---

id: home

title: React – The library for web and native user interfaces

permalink: index.html

{/* See HomeContent.js */}

---

title: React Blog

<Intro>

This blog is the official source for the updates from the React team. Anything important, including release notes or deprecation notices, will be posted here first. You can also follow the [@reactjs](https://twitter.com/reactjs) account on Twitter, but you won’t miss anything essential if you only read this blog.

</Intro>

<BlogCard title="React Canaries: Incremental Feature Rollout Outside Meta" date="May 3, 2023" url="/blog/2023/05/03/react-canaries">

Traditionally, new React features used to only be available at Meta first, and land in the open source releases later. We'd like to offer the React community an option to adopt individual new features as soon as their design is close to final--similar to how Meta uses React internally. We are introducing a new officially supported Canary release channel. It lets curated setups like frameworks decouple adoption of individual React features from the React release schedule.

</BlogCard>

<BlogCard title="React Labs: What We've Been Working On – March 2023" date="March 22, 2023" url="/blog/2023/03/22/react-labs-what-we-have-been-working-on-march-2023">

In React Labs posts, we write about projects in active research and development. Since our last update, we've made significant progress on React Server Components, Asset Loading, Optimizing Compiler, Offscreen Rendering, and Transition Tracing, and we'd like to share what we learned.

</BlogCard>

<BlogCard title="Introducing react.dev" date="March 16, 2023" url="/blog/2023/03/16/introducing-react-dev">

Today we are thrilled to launch react.dev, the new home for React and its documentation. In this post, we would like to give you a tour of the new site.

</BlogCard>

<BlogCard title="React Labs: What We've Been Working On – June 2022" date="June 15, 2022" url="/blog/2022/06/15/react-labs-what-we-have-been-working-on-june-2022">

React 18 was years in the making, and with it brought valuable lessons for the React team. Its release was the result of many years of research and exploring many paths. Some of those paths were successful; many more were dead-ends that led to new insights. One lesson we've learned is that it's frustrating for the community to wait for new features without having insight into these paths that we're exploring...

</BlogCard>

<BlogCard title="React v18.0" date="March 29, 2022" url="/blog/2022/03/29/react-v18">

React 18 is now available on npm! In our last post, we shared step-by-step instructions for upgrading your app to React 18. In this post, we'll give an overview of what's new in React 18, and what it means for the future...

</BlogCard>

<BlogCard title="How to Upgrade to React 18" date="March 8, 2022" url="/blog/2022/03/08/react-18-upgrade-guide">

As we shared in the release post, React 18 introduces features powered by our new concurrent renderer, with a gradual adoption strategy for existing applications. In this post, we will guide you through the steps for upgrading to React 18...

</BlogCard>

<BlogCard title="React Conf 2021 Recap" date="December 17, 2021" url="/blog/2021/12/17/react-conf-2021-recap">

Last week we hosted our 6th React Conf. In previous years, we've used the React Conf stage to deliver industry changing announcements such as React Native and React Hooks. This year, we shared our multi-platform vision for React, starting with the release of React 18 and gradual adoption of concurrent features...

</BlogCard>

<BlogCard title="The Plan for React 18" date="June 8, 2021" url="/blog/2021/06/08/the-plan-for-react-18">

The React team is excited to share a few updates:

- We've started work on the React 18 release, which will be our next major version.

- We've created a Working Group to prepare the community for gradual adoption of new features in React 18.

- We've published a React 18 Alpha so that library authors can try it and provide feedback...

</BlogCard>

<BlogCard title="Introducing Zero-Bundle-Size React Server Components" date="December 21, 2020" url="/blog/2020/12/21/data-fetching-with-react-server-components">

2020 has been a long year. As it comes to an end we wanted to share a special Holiday Update on our research into zero-bundle-size React Server Components. To introduce React Server Components, we have prepared a talk and a demo. If you want, you can check them out during the holidays, or later when work picks back up in the new year...

</BlogCard>

All release notes {/*all-release-notes*/}

Not every React release deserves its own blog post, but you can find a detailed changelog for every release in the [`CHANGELOG.md`](https://github.com/facebook/react/blob/main/CHANGELOG.md) file in the React repository, as well as on the [Releases](https://github.com/facebook/react/releases) page.

Older posts {/*older-posts*/}

See the [older posts.](https://reactjs.org/blog/all.html)

---

title: "React Conf 2021 Recap"

December 17, 2021 by [Jesslyn Tannady](https://twitter.com/jtannady) and [Rick Hanlon](https://twitter.com/rickhanlonii)

<Intro>

Last week we hosted our 6th React Conf. In previous years, we've used the React Conf stage to deliver industry changing announcements such as [_React Native_](https://engineering.fb.com/2015/03/26/android/react-native-bringing-modern-web-techniques-to-mobile/) and [_React Hooks_](https://reactjs.org/docs/hooks-intro.html). This year, we shared our multi-platform vision for React, starting with the release of React 18 and gradual adoption of concurrent features.

</Intro>

This was the first time React Conf was hosted online, and it was streamed for free, translated to 8 different languages. Participants from all over the world joined our conference Discord and the replay event for accessibility in all timezones. Over 50,000 people registered, with over 60,000 views of 19 talks, and 5,000 participants in Discord across both events.

All the talks are [available to stream online](https://www.youtube.com/watch?v=FZ0cG47msEk&list=PLNG_1j3cPCaZZ7etkzWA7JfdmKWT0pMsa).

Here’s a summary of what was shared on stage:

React 18 and concurrent features {/*react-18-and-concurrent-features*/}

In the keynote, we shared our vision for the future of React starting with React 18.

React 18 adds the long-awaited concurrent renderer and updates to Suspense without any major breaking changes. Apps can upgrade to React 18 and begin gradually adopting concurrent features with the amount of effort on par with any other major release.

This means there is no concurrent mode, only concurrent features.

In the keynote, we also shared our vision for Suspense, Server Components, new React working groups, and our long-term many-platform vision for React Native.

Watch the full keynote from [Andrew Clark](https://twitter.com/acdlite), [Juan Tejada](https://twitter.com/_jstejada), [Lauren Tan](https://twitter.com/potetotes), and [Rick Hanlon](https://twitter.com/rickhanlonii) here:

<YouTubeIframe src="https://www.youtube.com/embed/FZ0cG47msEk" />

React 18 for Application Developers {/*react-18-for-application-developers*/}

In the keynote, we also announced that the React 18 RC is available to try now. Pending further feedback, this is the exact version of React that we will publish to stable early next year.

To try the React 18 RC, upgrade your dependencies:

```bash
npm install react@rc react-dom@rc
```

and switch to the new `createRoot` API:

```js
// before
const container = document.getElementById('root');
ReactDOM.render(<App />, container);

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

For a demo of upgrading to React 18, see [Shruti Kapoor](https://twitter.com/shrutikapoor08)'s talk here:

<YouTubeIframe src="https://www.youtube.com/embed/ytudH8je5ko" />

Streaming Server Rendering with Suspense {/*streaming-server-rendering-with-suspense*/}

React 18 also includes improvements to server-side rendering performance using Suspense.

Streaming server rendering lets you generate HTML from React components on the server, and stream that HTML to your users. In React 18, you can use `Suspense` to break down your app into smaller independent units which can be streamed independently of each other without blocking the rest of the app. This means users will see your content sooner and be able to start interacting with it much faster.

For a deep dive, see [Shaundai Person](https://twitter.com/shaundai)'s talk here:

<YouTubeIframe src="https://www.youtube.com/embed/pj5N-Khihgc" />

The first React working group {/*the-first-react-working-group*/}

For React 18, we created our first Working Group to collaborate with a panel of experts, developers, library maintainers, and educators. Together we worked to create our gradual adoption strategy and refine new APIs such as `useId`, `useSyncExternalStore`, and `useInsertionEffect`.

For an overview of this work, see [Aakansha' Doshi](https://twitter.com/aakansha1216)'s talk:

<YouTubeIframe src="https://www.youtube.com/embed/qn7gRClrC9U" />

React Developer Tooling {/*react-developer-tooling*/}

To support the new features in this release, we also announced the newly formed React DevTools team and a new Timeline Profiler to help developers debug their React apps.

For more information and a demo of new DevTools features, see [Brian Vaughn](https://twitter.com/brian_d_vaughn)'s talk:

<YouTubeIframe src="https://www.youtube.com/embed/oxDfrke8rZg" />

React without memo {/*react-without-memo*/}

Looking further into the future, [Xuan Huang (■■)](https://twitter.com/Huxpro) shared an update from our React Labs research into an auto-memoizing compiler. Check out this talk for more information and a demo of the compiler prototype:

<YouTubeIframe src="https://www.youtube.com/embed/lGEMwh32soc" />

React docs keynote {/*react-docs-keynote*/}

[Rachel Nabors](https://twitter.com/rachelnabors) kicked off a section of talks about learning and designing with React with a keynote about our investment in React's new docs ([now shipped as react.dev](/blog/2023/03/16/introducing-react-dev)):

<YouTubeIframe src="https://www.youtube.com/embed/mneDaMYOKP8" />

And more... {/*and-more*/}

We also heard talks on learning and designing with React:

* Debbie O'Brien: [Things I learnt from the new React docs](https://youtu.be/-7odLW_hG7s).

* Sarah Rainsberger: [Learning in the Browser](https://youtu.be/5X-WEQflCL0).

* Linton Ye: [The ROI of Designing with React](https://youtu.be/7cPWmID5XAk).

* Delba de Oliveira: [Interactive playgrounds with React](https://youtu.be/zL8cz2W0z34).

Talks from the Relay, React Native, and PyTorch teams:

* Robert Balicki: [Re-introducing Relay](https://youtu.be/lhVGdErZuN4).

* Eric Rozell and Steven Moyes: [React Native Desktop](https://youtu.be/9L4FFrvwJwY).

* Roman Rädle: [On-device Machine Learning for React Native](https://youtu.be/NLj73vrc2I8)

And talks from the community on accessibility, tooling, and Server Components:

* Daishi Kato: [React 18 for External Store Libraries](https://youtu.be/oPfSC5bQPR8).

* Diego Haz: [Building Accessible Components in React 18](https://youtu.be/dcm8fjBfro8).

* Tafu Nakazaki: [Accessible Japanese Form Components with React](https://youtu.be/S4a0QlsH0pU).

* Lyle Troxell: [UI tools for artists](https://youtu.be/b3l4WxipFsE).

* Helen Lin: [Hydrogen + React 18](https://youtu.be/HS6vIYkSNks).

Thank you {/*thank-you*/}

This was our first year planning a conference ourselves, and we have a lot of people to thank.

First, thanks to all of our speakers [Aakansha Doshi](https://twitter.com/aakansha1216), [Andrew Clark](https://twitter.com/acdlite), [Brian Vaughn](https://twitter.com/brian_d_vaughn), [Daishi Kato](https://twitter.com/dai_shi), [Debbie O'Brien](https://twitter.com/debs_obrien), [Delba de Oliveira](https://twitter.com/delba_oliveira), [Diego Haz](https://twitter.com/diegohaz), [Eric Rozell](https://twitter.com/EricRozell), [Helen Lin](https://twitter.com/wizardlyhel), [Juan Tejada](https://twitter.com/_jstejada), [Lauren Tan](https://twitter.com/potetotes), [Linton Ye](https://twitter.com/lintonye), [Lyle Troxell](https://twitter.com/lyle), [Rachel Nabors](https://twitter.com/rachelnabors), [Rick Hanlon](https://twitter.com/rickhanlonii), [Robert Balicki](https://twitter.com/StatisticsFTW), [Roman Rädle](https://twitter.com/raedle), [Sarah Rainsberger](https://twitter.com/sarah11918), [Shaundai Person](https://twitter.com/shaundai), [Shruti Kapoor](https://twitter.com/shrutikapoor08), [Steven Moyes](https://twitter.com/moyessa), [Tafu Nakazaki](https://twitter.com/hawaiiman0), and [Xuan Huang (■■)](https://twitter.com/Huxpro).

Thanks to everyone who helped provide feedback on talks including [Andrew Clark](https://twitter.com/acdlite), [Dan Abramov](https://twitter.com/dan_abramov), [Dave McCabe](https://twitter.com/mcc_abe), [Eli White](https://twitter.com/Eli_White), [Joe Savona](https://twitter.com/en_JS), [Lauren Tan](https://twitter.com/potetotes), [Rachel Nabors](https://twitter.com/rachelnabors), and [Tim Yung](https://twitter.com/yungsters).

Thanks to [Lauren Tan](https://twitter.com/potetotes) for setting up the conference Discord and serving as our Discord admin.

Thanks to [Seth Webster](https://twitter.com/sethwebster) for feedback on overall direction and making sure we were focused on diversity and inclusion.

Thanks to [Rachel Nabors](https://twitter.com/rachelnabors) for spearheading our moderation effort, and [Aisha Blake](https://twitter.com/AishaBlake) for creating our moderation guide, leading our

moderation team, training the translators and moderators, and helping to moderate both events.

Thanks to our moderators [Jesslyn Tannady](https://twitter.com/jtannady), [Suzie Grange](https://twitter.com/missuze), [Becca Bailey](https://twitter.com/beccaliz), [Luna Wei](https://twitter.com/lunaleaps), [Joe Previte](https://twitter.com/jsjoeio), [Nicola Corti](https://twitter.com/Cortinico), [Gijs Weterings](https://twitter.com/gweterings), [Claudio Procida](https://twitter.com/claudiopro), Julia Neumann, Mengdi Chen, Jean Zhang, Ricky Li, and [Xuan Huang (■■)](https://twitter.com/Huxpro).

Thanks to [Manjula Dube](https://twitter.com/manjula_dube), [Sahil Mhapsekar](https://twitter.com/apheri0), and Vihang Patel from [React India](https://www.reactindia.io/), and [Jasmine Xie](https://twitter.com/jasmine_xby), [QiChang Li](https://twitter.com/QCL15), and [YanLun Li](https://twitter.com/anneincoding) from [React China](https://twitter.com/ReactChina) for helping moderate our replay event and keep it engaging for the community.

Thanks to Vercel for publishing their [Virtual Event Starter Kit](https://vercel.com/virtual-event-starter-kit), which the conference website was built on, and to [Lee Robinson](https://twitter.com/leeerob) and [Delba de Oliveira](https://twitter.com/delba_oliveira) for sharing their experience running Next.js Conf.

Thanks to [Leah Silber](https://twitter.com/wifelette) for sharing her experience running conferences, learnings from running [RustConf](https://rustconf.com/), and for her book [Event Driven](https://leanpub.com/eventdriven/) and the advice it contains for running conferences.

Thanks to [Kevin Lewis](https://twitter.com/_phzn) and [Rachel Nabors](https://twitter.com/rachelnabors) for sharing their experience running Women of React Conf.

Thanks to [Aakansha Doshi](https://twitter.com/aakansha1216), [Laurie Barth](https://twitter.com/laurieontech), [Michael Chan](https://twitter.com/chantastic), and [Shaundai Person](https://twitter.com/shaundai) for their advice and ideas throughout planning.

Thanks to [Dan Lebowitz](https://twitter.com/lebo) for help designing and building the conference website and tickets.

Thanks to Laura Podolak Waddell, Desmond Osei-Acheampong, Mark Rossi, Josh Toberman and others on the Facebook Video Productions team for recording the videos for the Keynote and Meta employee talks.

Thanks to our partner HitPlay for helping to organize the conference, editing all the videos in the stream, translating all the talks, and moderating the Discord in multiple languages.

Finally, thanks to all of our participants for making this a great React Conf!

---

title: "The Plan for React 18"

June 8, 2021 by [Andrew Clark](https://twitter.com/acdlite), [Brian Vaughn](https://github.com/bvaughn), [Christine Abernathy](https://twitter.com/abernathyca), [Dan Abramov](https://twitter.com/dan_abramov), [Rachel Nabors](https://twitter.com/rachelnabors), [Rick Hanlon](https://twitter.com/rickhanlonii), [Sebastian Markbåge](https://twitter.com/sebmarkbage), and

[Seth Webster](https://twitter.com/sethwebster)

<Intro>

The React team is excited to share a few updates:

1. We've started work on the React 18 release, which will be our next major version.

2. We've created a Working Group to prepare the community for gradual adoption of new features in React 18.

3. We've published a React 18 Alpha so that library authors can try it and provide feedback.

These updates are primarily aimed at maintainers of third-party libraries. If you're learning, teaching, or using React to build user-facing applications, you can safely ignore this post. But you are welcome to follow the discussions in the React 18 Working Group if you're curious!

</Intro>

What's coming in React 18 {/*whats-coming-in-react-18*/}

When it's released, React 18 will include out-of-the-box improvements (like [automatic batching](https://github.com/reactwg/react-18/discussions/21)), new APIs (like [`startTransition`](https://github.com/reactwg/react-18/discussions/41)), and a [new streaming server renderer](https://github.com/reactwg/react-18/discussions/37) with built-in support for `React.lazy`.

These features are possible thanks to a new opt-in mechanism we're adding in React 18. It's called "concurrent rendering" and it lets React prepare multiple versions of the UI at the same time. This change is mostly behind-the-scenes, but it unlocks new possibilities to improve both real and perceived performance of your app.

If you've been following our research into the future of React (we don't expect you to!), you might have heard of something called "concurrent mode" or that it might break your app. In response to this feedback from the community, we've redesigned the upgrade strategy for gradual adoption. Instead of an all-or-nothing "mode", concurrent rendering will only be enabled for updates triggered by one of the new features. In practice, this means **you will be able to adopt React 18 without rewrites and try the new features at your own pace.**

A gradual adoption strategy {/*a-gradual-adoption-strategy*/}

Since concurrency in React 18 is opt-in, there are no significant out-of-the-box breaking changes to component behavior. **You can upgrade to React 18 with minimal or no changes to your application code, with a level of effort comparable to a typical major React release**. Based on our experience converting several apps to React 18, we expect that many users will be able to upgrade within a single afternoon.

We successfully shipped concurrent features to tens of thousands of components at Facebook, and in our experience, we've found that most React components "just work" without additional changes. We're committed to making sure this is a smooth upgrade for the entire community, so today we're announcing the React 18 Working Group.

Working with the community {/*working-with-the-community*/}

We’re trying something new for this release: We've invited a panel of experts, developers, library authors, and educators from across the React community to participate in our [React 18 Working Group](https://github.com/reactwg/react-18) to provide feedback, ask questions, and collaborate on the release. We couldn't invite everyone we wanted to this initial, small group, but if this experiment works out, we hope there will be more in the future!

The goal of the React 18 Working Group is to prepare the ecosystem for a smooth, gradual adoption of React 18 by existing applications and libraries. The Working Group is hosted on [GitHub Discussions](https://github.com/reactwg/react-18/discussions) and is available for the public to read. Members of the working group can leave feedback, ask questions, and share ideas. The core team will also use the discussions repo to share our research findings. As the stable release gets closer, any important information will also be posted on this blog.

For more information on upgrading to React 18, or additional resources about the release, see the [React 18 announcement post](https://github.com/reactwg/react-18/discussions/4).

Accessing the React 18 Working Group {/*accessing-the-react-18-working-group*/}

Everyone can read the discussions in the [React 18 Working Group repo](https://github.com/reactwg/react-18).

Because we expect an initial surge of interest in the Working Group, only invited members will be allowed to create or comment on threads. However, the threads are fully visible to the public, so everyone has access to the same information. We believe this is a good compromise between creating a productive environment for working group members, while maintaining transparency with the wider community.

As always, you can submit bug reports, questions, and general feedback to our [issue tracker](https://github.com/facebook/react/issues).

How to try React 18 Alpha today {/*how-to-try-react-18-alpha-today*/}

New alphas are [regularly published to npm using the `@alpha` tag](https://github.com/reactwg/react-18/discussions/9). These releases are built using the most recent commit to our main repo. When a feature or bugfix is merged, it will appear in an alpha the following weekday.

There may be significant behavioral or API changes between alpha releases. Please remember that **alpha releases are not recommended for user-facing, production applications**.

Projected React 18 release timeline {/*projected-react-18-release-timeline*/}

We don't have a specific release date scheduled, but we expect it will take several months of feedback and iteration before React 18 is ready for most production applications.

* Library Alpha: Available today

* Public Beta: At least several months

* Release Candidate (RC): At least several weeks after Beta

* General Availability: At least several weeks after RC

More details about our projected release timeline are [available in the Working Group](https://github.com/reactwg/react-18/discussions/9). We'll post updates on this blog when we're closer to a public release.

---

title: "React Canaries: Enabling Incremental Feature Rollout Outside Meta"

May 3, 2023 by [Dan Abramov](https://twitter.com/dan_abramov), [Sophie Alpert](https://twitter.com/sophiebits), [Rick Hanlon](https://twitter.com/rickhanlonii), [Sebastian Markbåge](https://twitter.com/sebmarkbage), and [Andrew Clark](https://twitter.com/acdlite)

<Intro>

We'd like to offer the React community an option to adopt individual new features as soon as their design is close to final, before they're released in a stable version--similar to how Meta has long used bleeding-edge versions of React internally. We are introducing a new officially supported [Canary release channel](/community/versioning-policy#canary-channel). It lets curated setups like frameworks decouple adoption of individual React features from the React release schedule.

</Intro>

---

tl;dr {/*tldr*/}

* We're introducing an officially supported [Canary release channel](/community/versioning-policy#canary-channel) for React. Since it's officially supported, if any regressions land, we'll treat them with a similar urgency to bugs in stable releases.

* Canaries let you start using individual new React features before they land in the semver-stable releases.

* Unlike the [Experimental](/community/versioning-policy#experimental-channel) channel, React Canaries only include features that we reasonably believe to be ready for adoption. We encourage frameworks to consider bundling pinned Canary React releases.

* We will announce breaking changes and new features on our blog as they land in Canary releases.

* **As always, React continues to follow semver for every Stable release.**

How React features are usually developed {/*how-react-features-are-usually-developed*/}

Typically, every React feature has gone through the same stages:

1. We develop an initial version and prefix it with `experimental_` or `unstable_`. The feature is only available in the `experimental` release channel. At this point, the feature is expected to change significantly.

2. We find a team at Meta willing to help us test this feature and provide feedback on it. This leads to a round of changes. As the feature becomes more stable, we work with more teams at Meta to try it out.

3. Eventually, we feel confident in the design. We remove the prefix from the API name, and make the feature available on the `main` branch by default, which most Meta products use. At this point, any team at Meta can use this feature.

4. As we build confidence in the direction, we also post an RFC for the new feature. At this point we know the design works for a broad set of cases, but we might make some last minute adjustments.

5. When we are close to cutting an open source release, we write documentation for the feature and finally release the feature in a stable React release.

This playbook works well for most features we've released so far. However, there can be a significant gap between when the feature is generally ready to use (step 3) and when it is released in open source (step 5).

We'd like to offer the React community an option to follow the same approach as Meta, and adopt individual new features earlier (as they become available) without having to wait for the next release cycle of React.

As always, all React features will eventually make it into a Stable release.

Can we just do more minor releases? {/*can-we-just-do-more-minor-releases*/}

Generally, we *do* use minor releases for introducing new features.

However, this isn't always possible. Sometimes, new features are interconnected with *other* new features which have not yet been fully completed and that we're still actively iterating on. We can't release them separately because their implementations are related. We can't version them separately because they affect the same packages (for example, `react` and `react-dom`). And we need to keep the ability to iterate on the pieces that aren't ready without a flurry of major version releases, which semver would require us to do.

At Meta, we've solved this problem by building React from the `main` branch, and manually updating it to a specific pinned commit every week. This is also the approach that React Native releases have been following for the last several years. Every *stable* release of React Native is pinned to a specific commit from the `main` branch of the React repository. This lets React Native include important bugfixes and incrementally adopt new React features at the framework level without getting coupled to the global React release schedule.

We would like to make this workflow available to other frameworks and curated setups. For example, it lets a framework *on top of* React include a React-related breaking change *before* this breaking change gets included into a stable React release. This is particularly useful because some breaking changes only affect framework integrations. This lets a framework release such a change in its own minor version without breaking semver.

Rolling releases with the Canaries channel will allow us to have a tighter feedback loop and ensure that new features get comprehensive testing in the community. This workflow is closer to how TC39, the JavaScript standards committee, [handles changes in numbered stages](https://tc39.es/process-document/). New React features may be available in frameworks built on React before they are in a React stable release, just as new JavaScript features ship in browsers before they are officially ratified as part of the specification.

Why not use experimental releases instead? {/*why-not-use-experimental-releases-instead*/}

Although you *can* technically use [Experimental releases](/community/versioning-policy#canary-channel), we recommend against using them in production because experimental APIs can undergo significant breaking changes on their way to stabilization (or can even be removed entirely). While Canaries can also contain mistakes (as with any release), going forward we plan to announce any significant breaking changes in Canaries on our blog. Canaries are the closest to the code Meta runs internally, so you can generally expect them to be relatively stable. However, you *do* need to keep the version pinned and manually scan the GitHub commit log when updating between the pinned commits.

We expect that most people using React outside a curated setup (like a framework) will want to continue using the Stable releases. However, if you're building a framework, you might want to consider bundling a Canary version of React pinned to a particular commit, and update it at your own pace. The benefit of that is that it lets you ship individual completed React features and bugfixes earlier for your users and at your own release schedule, similar to how React Native has been doing it for the last few years. The downside is that you would take on additional responsibility to review which React commits are being pulled in and communicate to your users which React changes are included with your releases.

If you're a framework author and want to try this approach, please get in touch with us.

Announcing breaking changes and new features early {/*announcing-breaking-changes-and-new-features-early*/}

Canary releases represent our best guess of what will go into the next stable React release at any given time.

Traditionally, we've only announced breaking changes at the *end* of the release cycle (when doing a major release). Now that Canary releases are an officially supported way to consume React, we plan to shift towards announcing breaking changes and significant new features *as they land* in Canaries. For example, if we merge a breaking change that will go out in a Canary, we will write a post about it on the React blog, including codemods and migration instructions if necessary. Then, if you're a framework author cutting a major release that updates the pinned React canary to include that change, you can link to our blog post from your release notes. Finally, when a stable major version of React is ready, we will link to those already published blog posts, which we hope will help our team make progress faster.

We plan to document APIs as they land in Canaries--even if these APIs are not yet available outside of them. APIs that are only available in Canaries will be marked with a special note on the corresponding pages. This will include APIs like [`use`](https://github.com/reactjs/rfcs/pull/229), and some others (like `cache` and `createServerContext`) which we'll send RFCs for.

Canaries must be pinned {/*canaries-must-be-pinned*/}

If you decide to adopt the Canary workflow for your app or framework, make sure you always pin the *exact* version of the Canary you're using. Since Canaries are pre-releases, they may still include breaking changes.

Example: React Server Components {/*example-react-server-components*/}

As we [announced in March](/blog/2023/03/22/react-labs-what-we-have-been-working-on-march-2023#react-server-components), the React Server Components conventions have been finalized, and we do not expect significant breaking changes related to their user-facing API contract. However, we can't release support for React Server Components in a stable version of React yet because we are still working on several intertwined framework-only features (such as [asset

loading](/blog/2023/03/22/react-labs-what-we-have-been-working-on-march-2023#asset-loading)) and expect more breaking changes there.

This means that React Server Components are ready to be adopted by frameworks. However, until the next major React release, the only way for a framework to adopt them is to ship a pinned Canary version of React. (To avoid bundling two copies of React, frameworks that wish to do this would need to enforce resolution of `react` and `react-dom` to the pinned Canary they ship with their framework, and explain that to their users. As an example, this is what Next.js App Router does.)

Testing libraries against both Stable and Canary versions
{/*testing-libraries-against-both-stable-and-canary-versions*/}

We do not expect library authors to test every single Canary release since it would be prohibitively difficult. However, just as when we [originally introduced the different React pre-release channels three years ago](https://legacy.reactjs.org/blog/2019/10/22/react-release-channels.html), we encourage libraries to run tests against *both* the latest Stable and latest Canary versions. If you see a change in behavior that wasn't announced, please file a bug in the React repository so that we can help diagnose it. We expect that as this practice becomes widely adopted, it will reduce the amount of effort necessary to upgrade libraries to new major versions of React, since accidental regressions would be found as they land.

<Note>

Strictly speaking, Canary is not a *new* release channel--it used to be called Next. However, we've decided to rename it to avoid confusion with Next.js. We're announcing it as a *new* release channel to communicate the new expectations, such as Canaries being an officially supported way to use React.

</Note>

Stable releases work like before {/*stable-releases-work-like-before*/}

We are not introducing any changes to stable React releases.

---

title: "React Labs: What We've Been Working On – March 2023"

March 22, 2023 by [Joseph Savona](https://twitter.com/en_JS), [Josh Story](https://twitter.com/joshcstory), [Lauren Tan](https://twitter.com/potetotes), [Mengdi Chen](https://twitter.com/mengdi_en), [Samuel Susla](https://twitter.com/SamuelSusla), [Sathya Gunasekaran](https://twitter.com/_gsathya), [Sebastian Markbåge](https://twitter.com/sebmarkbage), and [Andrew Clark](https://twitter.com/acdlite)

<Intro>

In React Labs posts, we write about projects in active research and development. We've made significant progress on them since our [last update](https://react.dev/blog/2022/06/15/react-labs-what-we-have-been-working-on-june-2022), and we'd like to share what we learned.

</Intro>

React Server Components {/*react-server-components*/}

React Server Components (or RSC) is a new application architecture designed by the React team.

We've first shared our research on RSC in an [introductory talk](/blog/2020/12/21/data-fetching-with-react-server-components) and an [RFC](https://github.com/reactjs/rfcs/pull/188). To recap them, we are introducing a new kind of component--Server Components--that run ahead of time and are excluded from your JavaScript bundle. Server Components can run during the build, letting you read from the filesystem or fetch static content. They can also run on the server, letting you access your data layer without having to build an API. You can pass data by props from Server Components to the interactive Client Components in the browser.

RSC combines the simple "request/response" mental model of server-centric Multi-Page Apps with the seamless interactivity of client-centric Single-Page Apps, giving you the best of both worlds.

Since our last update, we have merged the [React Server Components RFC](https://github.com/reactjs/rfcs/blob/main/text/0188-server-components.md) to ratify the proposal. We resolved outstanding issues with the [React Server Module Conventions](https://github.com/reactjs/rfcs/blob/main/text/0227-server-module-conventions.md) proposal, and reached consensus with our partners to go with the `"use client"` convention. These documents also act as specification for what an RSC-compatible implementation should support.

The biggest change is that we introduced [`async` / `await`](https://github.com/reactjs/rfcs/pull/229) as the primary way to do data fetching from Server Components. We also plan to support data loading from the client by introducing a new hook called `use` that unwraps Promises. Although we can't support `async / await` in arbitrary components in client-only apps, we plan to add support for it when you structure your client-only app similar to how RSC apps are structured.

Now that we have data fetching pretty well sorted, we're exploring the other direction: sending data from the client to the server, so that you can execute database mutations and implement forms. We're doing this by letting you pass Server Action functions across the server/client boundary, which the client can then call, providing seamless RPC. Server Actions also give you progressively enhanced forms before JavaScript loads.

React Server Components has shipped in [Next.js App Router](/learn/start-a-new-react-project#nextjs-app-router). This showcases a deep integration of a router that really buys into RSC as a primitive, but it's not the only way to build a RSC-compatible router and framework. There's a clear separation for features provided by the RSC spec and implementation. React Server Components is meant as a spec for components that work across compatible React frameworks.

We generally recommend using an existing framework, but if you need to build your own custom framework, it is possible. Building your own RSC-compatible framework is not as easy as we'd like it to be, mainly due to the deep bundler integration needed. The current generation of bundlers are great for use on the client, but they weren't designed with first-class support for splitting a single module graph between the server and the client. This is why we're now partnering directly with bundler developers to get the primitives for RSC built-in.

Asset Loading {/*asset-loading*/}

[Suspense](/reference/react/Suspense) lets you specify what to display on the screen while the data or code for your components is still being loaded. This lets your users progressively see more content while the page is loading as well as during the router navigations that load more data and code. However, from the user's perspective, data loading and rendering do not tell the whole story when considering whether new content is ready. By default, browsers load stylesheets, fonts, and images independently, which can lead to UI jumps and consecutive layout shifts.

We're working to fully integrate Suspense with the loading lifecycle of stylesheets, fonts, and images, so that React takes them into account to determine whether the content is ready to be displayed. Without any change to the way you author your React components, updates will behave in a more coherent and pleasing manner. As an optimization, we will also provide a manual way to preload assets like fonts directly from components.

We are currently implementing these features and will have more to share soon.

Document Metadata {/*document-metadata*/}

Different pages and screens in your app may have different metadata like the `<title>` tag, description, and other `<meta>` tags specific to this screen. From the maintenance perspective, it's more scalable to keep this information close to the React component for that page or screen. However, the HTML tags for this metadata need to be in the document `<head>` which is typically rendered in a component at the very root of your app.

Today, people solve this problem with one of the two techniques.

One technique is to render a special third-party component that moves `<title>`, `<meta>`, and other tags inside it into the document `<head>`. This works for major browsers but there are many clients which do not run client-side JavaScript, such as Open Graph parsers, and so this technique is not universally suitable.

Another technique is to server-render the page in two parts. First, the main content is rendered and all such tags are collected. Then, the `<head>` is rendered with these tags. Finally, the `<head>` and the main content are sent to the browser. This approach works, but it prevents you from taking advantage of the [React 18's Streaming Server Renderer](/reference/react-dom/server/renderToReadableStream) because you'd have to wait for all content to render before sending the `<head>`.

This is why we're adding built-in support for rendering `<title>`, `<meta>`, and metadata `<link>` tags anywhere in your component tree out of the box. It would work the same way in all environments, including fully client-side code, SSR, and in the future, RSC. We will share more details about this soon.

React Optimizing Compiler {/*react-optimizing-compiler*/}

Since our previous update we've been actively iterating on the design of [React Forget](/blog/2022/06/15/react-labs-what-we-have-been-working-on-june-2022#react-compiler), an optimizing compiler for React. We've previously talked about it as an "auto-memoizing compiler", and that is true in some sense. But building the compiler has helped us understand React's programming model even more deeply. A better way to understand React Forget is as an automatic *reactivity* compiler.

The core idea of React is that developers define their UI as a function of the current state. You work with plain JavaScript values — numbers, strings, arrays, objects — and use standard JavaScript idioms — if/else, for, etc — to describe your component logic. The mental model is that React will re-render whenever the application state changes. We believe this simple mental model and keeping close to JavaScript semantics is an important principle in React's programming model.

The catch is that React can sometimes be *too* reactive: it can re-render too much. For example, in JavaScript we don't have cheap ways to compare if two objects or arrays are equivalent (having the same keys and values), so creating a new object or array on each render may cause React to do more work than it strictly needs to. This means developers have to explicitly memoize components so as to not over-react to changes.

Our goal with React Forget is to ensure that React apps have just the right amount of reactivity by default: that apps re-render only when state values *meaningfully* change. From an implementation perspective this means automatically memoizing, but we believe that the reactivity framing is a better way to understand React and Forget. One way to think about this is that React currently re-renders when object identity changes. With Forget, React re-renders when the semantic value changes — but without incurring the runtime cost of deep comparisons.

In terms of concrete progress, since our last update we have substantially iterated on the design of the compiler to align with this automatic reactivity approach and to incorporate feedback from using the compiler internally. After some significant refactors to the compiler starting late last year, we've now begun using the compiler in production in limited areas at Meta. We plan to open-source it once we've proved it in production.

Finally, a lot of people have expressed interest in how the compiler works. We're looking forward to sharing a lot more details when we prove the compiler and open-source it. But there are a few bits we can share now:

The core of the compiler is almost completely decoupled from Babel, and the core compiler API is (roughly) old AST in, new AST out (while retaining source location data). Under the hood we use a custom code representation and transformation pipeline in order to do low-level semantic analysis. However, the primary public interface to the compiler will be via Babel and other build system plugins. For ease of testing we currently have a Babel plugin which is a very thin wrapper that calls the compiler to generate a new version of each function and swap it in.

As we refactored the compiler over the last few months, we wanted to focus on refining the core compilation model to ensure we could handle complexities such as conditionals, loops, reassignment, and mutation. However, JavaScript has a lot of ways to express each of those features: if/else, ternaries, for, for-in, for-of, etc. Trying to support the full language up-front would have delayed the point where we could validate the core model. Instead, we started with a small but representative subset of the language: let/const, if/else, for loops, objects, arrays, primitives, function calls, and a few other features. As we gained confidence in the core model and refined our internal abstractions, we expanded the supported language subset. We're also explicit about syntax we don't yet support, logging diagnostics and skipping compilation for unsupported input. We have utilities to try the compiler on Meta's codebases and see what unsupported features are most common so we can prioritize those next. We'll continue incrementally expanding towards supporting the whole language.

Making plain JavaScript in React components reactive requires a compiler with a deep understanding of semantics so that it can understand exactly what the code is doing. By taking this approach, we're creating a system for reactivity within JavaScript that lets you write product code of any complexity with the full expressivity of the language, instead of being limited to a domain specific language.

Offscreen Rendering {/*offscreen-rendering*/}

Offscreen rendering is an upcoming capability in React for rendering screens in the background without additional performance overhead. You can think of it as a version of the [`content-visiblity` CSS property](https://developer.mozilla.org/en-US/docs/Web/CSS/content-visibility) that works not only for DOM elements but React components, too. During our research, we've discovered a variety of use cases:

- A router can prerender screens in the background so that when a user navigates to them, they're instantly available.

- A tab switching component can preserve the state of hidden tabs, so the user can switch between them without losing their progress.

- A virtualized list component can prerender additional rows above and below the visible window.

- When opening a modal or popup, the rest of the app can be put into "background" mode so that events and updates are disabled for everything except the modal.

Most React developers will not interact with React's offscreen APIs directly. Instead, offscreen rendering will be integrated into things like routers and UI libraries, and then developers who use those libraries will automatically benefit without additional work.

The idea is that you should be able to render any React tree offscreen without changing the way you write your components. When a component is rendered offscreen, it does not actually *mount* until the component becomes visible — its effects are not fired. For example, if a component uses `useEffect` to log analytics when it appears for the first time, prerendering won't mess up the accuracy of those analytics. Similarly, when a component goes offscreen, its effects are unmounted, too. A key feature of offscreen rendering is that you can toggle the visibility of a component without losing its state.

Since our last update, we've tested an experimental version of prerendering internally at Meta in our React Native apps on Android and iOS, with positive performance results. We've also improved how offscreen rendering works with Suspense — suspending inside an offscreen tree will not trigger Suspense fallbacks. Our remaining work involves finalizing the primitives that are exposed to library developers. We expect to publish an RFC later this year, alongside an experimental API for testing and feedback.

Transition Tracing {/*transition-tracing*/}

The Transition Tracing API lets you detect when [React Transitions](/reference/react/useTransition) become slower and investigate why they may be slow. Following our last update, we have completed the initial design of the API and published an [RFC](https://github.com/reactjs/rfcs/pull/238). The basic capabilities have also been implemented. The project is currently on hold. We welcome feedback on the RFC and look forward to resuming its development to provide a better performance measurement tool for React. This will be particularly useful with routers built on top of React Transitions, like the [Next.js App Router](/learn/start-a-new-react-project#nextjs-app-router).

* * *

In addition to this update, our team has made recent guest appearances on community podcasts and livestreams to speak more on our work and answer questions.

* [Dan Abramov](https://twitter.com/dan_abramov) and [Joe Savona](https://twitter.com/en_JS) were interviewed by [Kent C. Dodds on his YouTube channel](https://www.youtube.com/watch?v=h7tur48JSaw), where they discussed concerns around

React Server Components.

* [Dan Abramov](https://twitter.com/dan_abramov) and [Joe Savona](https://twitter.com/en_JS) were guests on the [JSParty podcast](https://jsparty.fm/267) and shared their thoughts about the future of React.

Thanks to [Andrew Clark](https://twitter.com/acdlite), [Dan Abramov](https://twitter.com/dan_abramov), [Dave McCabe](https://twitter.com/mcc_abe), [Luna Wei](https://twitter.com/lunaleaps), [Matt Carroll](https://twitter.com/mattcarrollcode), [Sean Keegan](https://twitter.com/DevRelSean), [Sebastian Silbermann](https://twitter.com/sebsilbermann), [Seth Webster](https://twitter.com/sethwebster), and [Sophie Alpert](https://twitter.com/sophiebits) for reviewing this post.

Thanks for reading, and see you in the next update!

---

title: "Introducing react.dev"

March 16, 2023 by [Dan Abramov](https://twitter.com/dan_abramov) and [Rachel Nabors](https://twitter.com/rachelnabors)

<Intro>

Today we are thrilled to launch react.dev, the new home for React and its documentation. In this post, we would like to give you a tour of the new site.

</Intro>

---

tl;dr {/*tldr*/}

* The new React site (react.dev) teaches modern React with function components and Hooks.

* We've included diagrams, illustrations, challenges, and over 600 new interactive examples.

* The previous React documentation site has now moved to legacy.reactjs.org.

New site, new domain, new homepage {/*new-site-new-domain-new-homepage*/}

First, a little bit of housekeeping.

To celebrate the launch of the new docs and, more importantly, to clearly separate the old and the new content, we've moved to the shorter react.dev domain. The old reactjs.org domain will now redirect here.

The old React docs are now archived at legacy.reactjs.org. All existing links to the old content will automatically redirect there to avoid "breaking the web", but the legacy site will not get many more updates.

Believe it or not, React will soon be ten years old. In JavaScript years, it's like a whole century! We've [refreshed the React homepage](https://react.dev) to reflect why we think React is a great way to create user interfaces today, and updated the getting started guides to more prominently mention modern React-based frameworks.

If you haven't seen the new homepage yet, check it out!

Going all-in on modern React with Hooks {/*going-all-in-on-modern-react-with-hooks*/}

When we released React Hooks in 2018, the Hooks docs assumed the reader is familiar with class components. This helped the community adopt Hooks very swiftly, but after a while the old docs failed to serve the new readers. New readers had to learn React twice: once with class components and then once again with Hooks.

The new docs teach React with Hooks from the beginning. The docs are divided in two main sections:

* **[Learn React](/learn)** is a self-paced course that teaches React from scratch.
* **[API Reference](/reference)** provides the details and usage examples for every React API.

Let's have a closer look at what you can find in each section.

<Note>

There are still a few rare class component use cases that do not yet have a Hook-based equivalent. Class components remain supported, and are documented in the [Legacy API](/reference/react/legacy) section of the new site.

</Note>

Quick start {/*quick-start*/}

The Learn section begins with the [Quick Start](/learn) page. It is a short introductory tour of React. It introduces the syntax for concepts like components, props, and state, but doesn't go into much detail on how to use them.

If you like to learn by doing, we recommend checking out the [Tic-Tac-Toe Tutorial](/learn/tutorial-tic-tac-toe) next. It walks you through building a little game with React, while teaching the skills you'll use every day. Here's what you'll build:

<Sandpack>

```js App.js
import { useState } from 'react';

function Square({ value, onSquareClick }) {
return (
<button className="square" onClick={onSquareClick}>
{value}
</button>
```

```
);
}

function Board({ xIsNext, squares, onPlay }) {
function handleClick(i) {
if (calculateWinner(squares) || squares[i]) {
return;
}
const nextSquares = squares.slice();
if (xIsNext) {
nextSquares[i] = 'X';
} else {
nextSquares[i] = 'O';
}
onPlay(nextSquares);
}

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

return (
<>
<div className="status">{status}</div>
<div className="board-row">
<Square value={squares[0]} onSquareClick={() => handleClick(0)} />
<Square value={squares[1]} onSquareClick={() => handleClick(1)} />
<Square value={squares[2]} onSquareClick={() => handleClick(2)} />
</div>
<div className="board-row">
<Square value={squares[3]} onSquareClick={() => handleClick(3)} />
<Square value={squares[4]} onSquareClick={() => handleClick(4)} />
<Square value={squares[5]} onSquareClick={() => handleClick(5)} />
```

```
</div>
<div className="board-row">
<Square value={squares[6]} onSquareClick={() => handleClick(6)} />
<Square value={squares[7]} onSquareClick={() => handleClick(7)} />
<Square value={squares[8]} onSquareClick={() => handleClick(8)} />
</div>
</>
);
}

export default function Game() {
const [history, setHistory] = useState([Array(9).fill(null)]);
const [currentMove, setCurrentMove] = useState(0);
const xIsNext = currentMove % 2 === 0;
const currentSquares = history[currentMove];

function handlePlay(nextSquares) {
const nextHistory = [...history.slice(0, currentMove + 1), nextSquares];
setHistory(nextHistory);
setCurrentMove(nextHistory.length - 1);
}

function jumpTo(nextMove) {
setCurrentMove(nextMove);
}

const moves = history.map((squares, move) => {
let description;
if (move > 0) {
description = 'Go to move #' + move;
} else {
description = 'Go to game start';
}
return (
<li key={move}>
<button onClick={() => jumpTo(move)}>{description}</button>
</li>
);
```

```
});

return (
<div className="game">
<div className="game-board">
<Board xIsNext={xIsNext} squares={currentSquares} onPlay={handlePlay} />
</div>
<div className="game-info">
<ol>{moves}</ol>
</div>
</div>
);
}

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;
}
```

```css styles.css
* {
```

```
box-sizing: border-box;
}

body {
font-family: sans-serif;
margin: 20px;
padding: 0;
}

.square {
background: #fff;
border: 1px solid #999;
float: left;
font-size: 24px;
font-weight: bold;
line-height: 34px;
height: 34px;
margin-right: -1px;
margin-top: -1px;
padding: 0;
text-align: center;
width: 34px;
}

.board-row:after {
clear: both;
content: '';
display: table;
}

.status {
margin-bottom: 10px;
}
.game {
display: flex;
flex-direction: row;
}

.game-info {
```

margin-left: 20px;

}
```

</Sandpack>

We'd also like to highlight [Thinking in React](/learn/thinking-in-react)—that's the tutorial that made React "click" for many of us. **We've updated both of these classic tutorials to use function components and Hooks,** so they're as good as new.

<Note>

The example above is a *sandbox*. We've added a lot of sandboxes—over 600!—everywhere throughout the site. You can edit any sandbox, or press "Fork" in the upper right corner to open it in a separate tab. Sandboxes let you quickly play with the React APIs, explore your ideas, and check your understanding.

</Note>

## Learn React step by step {/*learn-react-step-by-step*/}

We'd like everyone in the world to have an equal opportunity to learn React for free on their own.

This is why the Learn section is organized like a self-paced course split into chapters. The first two chapters describe the fundamentals of React. If you're new to React, or want to refresh it in your memory, start here:

- **[Describing the UI](/learn/describing-the-ui)** teaches how to display information with components.
- **[Adding Interactivity](/learn/adding-interactivity)** teaches how to update the screen in response to user input.

The next two chapters are more advanced, and will give you a deeper insight into the trickier parts:

- **[Managing State](/learn/managing-state)** teaches how to organize your logic as your app grows in complexity.
- **[Escape Hatches](/learn/escape-hatches)** teaches how you can "step outside" React, and when it makes most sense to do so.

Every chapter consists of several related pages. Most of these pages teach a specific skill or a technique—for example, [Writing Markup with JSX](/learn/writing-markup-with-jsx), [Updating Objects in State](/learn/updating-objects-in-state), or [Sharing State Between Components](/learn/sharing-state-between-components). Some of the pages focus on explaining an idea—like [Render and Commit](/learn/render-and-commit), or [State as a Snapshot](/learn/state-as-a-snapshot). And there are a few, like [You Might Not Need an Effect](/learn/you-might-not-need-an-effect), that share our suggestions based on what we've learned over these years.

You don't have to read these chapters as a sequence. Who has the time for this?! But you could. Pages in the Learn section only rely on concepts introduced by the earlier pages. If you want to read it like a book, go for it!
```

Check your understanding with challenges {/*check-your-understanding-with-challenges*/}

Most pages in the Learn section end with a few challenges to check your understanding. For example, here are a few challenges from the page about [Conditional Rendering](/learn/conditional-rendering#challenges).

You don't have to solve them right now! Unless you *really* want to.

<Challenges noTitle={true}>

Show an icon for incomplete items with `? :` {/*show-an-icon-for-incomplete-items-with--*/}

Use the conditional operator (`cond ? a : b`) to render a ■ if `isPacked` isn't `true`.

<Sandpack>

```js
function Item({ name, isPacked }) {
return (
<li className="item">
{name} {isPacked && '✔'}
</li>
);
}

export default function PackingList() {
return (
<section>
<h1>Sally Ride's Packing List</h1>
<ul>
<Item
isPacked={true}
name="Space suit"
/>
<Item
isPacked={true}
name="Helmet with a golden leaf"
/>
<Item
isPacked={false}
name="Photo of Tam"
```

/>

</section>
);
}
```

</Sandpack>

<Solution>

<Sandpack>

```js
function Item({ name, isPacked }) {
return (
<li className="item">
{name} {isPacked ? '✔' : '■'}
</li>
);
}

export default function PackingList() {
return (
<section>
<h1>Sally Ride's Packing List</h1>
<ul>
<Item
isPacked={true}
name="Space suit"
/>
<Item
isPacked={true}
name="Helmet with a golden leaf"
/>
<Item
isPacked={false}
name="Photo of Tam"
/>

</ul>

</section>

);

}

```

</Sandpack>

</Solution>

Show the item importance with `&&` {/*show-the-item-importance-with-*/}

In this example, each `Item` receives a numerical `importance` prop. Use the `&&` operator to render "_(Importance: X)_" in italics, but only for items that have non-zero importance. Your item list should end up looking like this:

* Space suit _(Importance: 9)_

* Helmet with a golden leaf

* Photo of Tam _(Importance: 6)_

Don't forget to add a space between the two labels!

<Sandpack>

```js
function Item({ name, importance }) {

return (

<li className="item">

{name}

</li>

);

}

export default function PackingList() {

return (

<section>

<h1>Sally Ride's Packing List</h1>

<ul>

<Item

importance={9}

name="Space suit"

/>

<Item
importance={0}
name="Helmet with a golden leaf"
/>
<Item
importance={6}
name="Photo of Tam"
/>
</ul>
</section>
);
}
```

</Sandpack>

<Solution>

This should do the trick:

<Sandpack>

```js
function Item({ name, importance }) {
return (
<li className="item">
{name}
{importance > 0 && ' '}
{importance > 0 &&
(Importance: {importance})
}
</li>
);
}

export default function PackingList() {
return (
<section>
<h1>Sally Ride's Packing List</h1>

<ul>
<Item
importance={9}
name="Space suit"
/>
<Item
importance={0}
name="Helmet with a golden leaf"
/>
<Item
importance={6}
name="Photo of Tam"
/>
</ul>
</section>
);
}
```

</Sandpack>

Note that you must write `importance > 0 && ...` rather than `importance && ...` so that if the `importance` is `0`, `0` isn't rendered as the result!

In this solution, two separate conditions are used to insert a space between then name and the importance label. Alternatively, you could use a fragment with a leading space: `importance > 0 && <> <i>...</i></>` or add a space immediately inside the `<i>`: `importance > 0 && <i> ...</i>`.

</Solution>

</Challenges>

Notice the "Show solution" button in the left bottom corner. It's handy if you want to check yourself!

Build an intuition with diagrams and illustrations
{/*build-an-intuition-with-diagrams-and-illustrations*/}

When we couldn't figure out how to explain something with code and words alone, we've added diagrams that help provide some intuition. For example, here is one of the diagrams from [Preserving and Resetting State](/learn/preserving-and-resetting-state):

<Diagram name="preserving_state_diff_same_pt1" height={350} width={794} alt="Diagram with three sections, with an arrow transitioning each section in between. The first section contains a React component labeled 'div' with a single child labeled 'section', which has a single child labeled 'Counter'
```

containing a state bubble labeled 'count' with value 3. The middle section has the same 'div' parent, but the child components have now been deleted, indicated by a yellow 'proof' image. The third section has the same 'div' parent again, now with a new child labeled 'div', highlighted in yellow, also with a new child labeled 'Counter' containing a state bubble labeled 'count' with value 0, all highlighted in yellow.">

When `section` changes to `div`, the `section` is deleted and the new `div` is added

</Diagram>

You'll also see some illustrations throughout the docs--here's one of the [browser painting the screen](/learn/render-and-commit#epilogue-browser-paint):

<Illustration alt="A browser painting 'still life with card element'."
src="/images/docs/illustrations/i_browser-paint.png" />

We've confirmed with the browser vendors that this depiction is 100% scientifically accurate.

A new, detailed API Reference {/*a-new-detailed-api-reference*/}

In the [API Reference](/reference/react), every React API now has a dedicated page. This includes all kinds of APIs:

- Built-in Hooks like [`useState`](/reference/react/useState).
- Built-in components like [`<Suspense>`](/reference/react/Suspense).
- Built-in browser components like [`<input>`](/reference/react-dom/components/input).
- Framework-oriented APIs like [`renderToPipeableStream`](/reference/react-dom/server/renderToReadableStream).
- Other React APIs like [`memo`](/reference/react/memo).

You'll notice that every API page is split into at least two segments: *Reference* and *Usage*.

[Reference](/reference/react/useState#reference) describes the formal API signature by listing its arguments and return values. It's concise, but it can feel a bit abstract if you're not familiar with that API. It describes what an API does, but not how to use it.

[Usage](/reference/react/useState#usage) shows why and how you would use this API in practice, like a colleague or a friend might explain. It shows the **canonical scenarios for how each API was meant to be used by the React team.** We've added color-coded snippets, examples of using different APIs together, and recipes that you can copy and paste from:

<Recipes titleText="Basic useState examples" titleId="examples-basic">

Counter (number) {/*counter-number*/}

In this example, the `count` state variable holds a number. Clicking the button increments it.

<Sandpack>

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

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

function handleClick() {
setCount(count + 1);
}

return (
<button onClick={handleClick}>
You pressed me {count} times
</button>
);
}
```

</Sandpack>

<Solution />

#### Text field (string) {/*text-field-string*/}

In this example, the `text` state variable holds a string. When you type, `handleChange` reads the latest input value from the browser input DOM element, and calls `setText` to update the state. This allows you to display the current `text` below.

<Sandpack>

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

export default function MyInput() {
const [text, setText] = useState('hello');

function handleChange(e) {
setText(e.target.value);
}

return (
<>
<input value={text} onChange={handleChange} />
You typed: {text}
<button onClick={() => setText('hello')}>
Reset

</button>
</>
);
}
```

</Sandpack>

<Solution />

Checkbox (boolean) {/*checkbox-boolean*/}

In this example, the `liked` state variable holds a boolean. When you click the input, `setLiked` updates the `liked` state variable with whether the browser checkbox input is checked. The `liked` variable is used to render the text below the checkbox.

<Sandpack>

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

export default function MyCheckbox() {
const [liked, setLiked] = useState(true);

function handleChange(e) {
setLiked(e.target.checked);
}

return (
<>
<label>
<input
type="checkbox"
checked={liked}
onChange={handleChange}
/>
I liked this
</label>
You {liked ? 'liked' : 'did not like'} this.
</>
);
}

```

</Sandpack>

<Solution />

Form (two variables) {/*form-two-variables*/}

You can declare more than one state variable in the same component. Each state variable is completely independent.

<Sandpack>

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

export default function Form() {
const [name, setName] = useState('Taylor');
const [age, setAge] = useState(42);

return (
<>
<input
value={name}
onChange={e => setName(e.target.value)}
/>
<button onClick={() => setAge(age + 1)}>
Increment age
</button>
<p>Hello, {name}. You are {age}.</p>
</>
);
}
```

```css
button { display: block; margin-top: 10px; }
```

</Sandpack>

<Solution />

</Recipes>
```

Some API pages also include [Troubleshooting](/reference/react/useEffect#troubleshooting) (for common problems) and [Alternatives](/reference/react-dom/findDOMNode#alternatives) (for deprecated APIs).

We hope that this approach will make the API reference useful not only as a way to look up an argument, but as a way to see all the different things you can do with any given API—and how it connects to the other ones.

What's next? {/*whats-next*/}

That's a wrap for our little tour! Have a look around the new website, see what you like or don't like, and keep the feedback coming in the [anonymous survey](https://www.surveymonkey.co.uk/r/PYRPF3X) or in our [issue tracker](https://github.com/reactjs/reactjs.org/issues).

We acknowledge this project has taken a long time to ship. We wanted to maintain a high quality bar that the React community deserves. While writing these docs and creating all of the examples, we found mistakes in some of our own explanations, bugs in React, and even gaps in the React design that we are now working to address. We hope that the new documentation will help us hold React itself to a higher bar in the future.

We've heard many of your requests to expand the content and functionality of the website, for example:

- Providing a TypeScript version for all examples;
- Creating the updated performance, testing, and accessibility guides;
- Documenting React Server Components independently from the frameworks that support them;
- Working with our international community to get the new docs translated;
- Adding missing features to the new website (for example, RSS for this blog).

Now that react.dev is out, we will be able to shift our focus from "catching up" with the third-party React educational resources to adding new information and further improving our new website.

We think there's never been a better time to learn React.

Who worked on this? {/*who-worked-on-this*/}

On the React team, [Rachel Nabors](https://twitter.com/rachelnabors/) led the project (and provided the illustrations), and [Dan Abramov](https://twitter.com/dan_abramov) designed the curriculum. They co-authored most of the content together as well.

Of course, no project this large happens in isolation. We have a lot of people to thank!

[Sylwia Vargas](https://twitter.com/SylwiaVargas) overhauled our examples to go beyond "foo/bar/baz" and kittens, and feature scientists, artists and cities from around the world. [Maggie Appleton](https://twitter.com/Mappletons) turned our doodles into a clear diagram system.

Thanks to [David McCabe](https://twitter.com/mcc_abe), [Sophie Alpert](https://twitter.com/sophiebits), [Rick Hanlon](https://twitter.com/rickhanlonii), [Andrew Clark](https://twitter.com/acdlite), and [Matt Carroll](https://twitter.com/mattcarrollcode) for additional writing contributions. We'd also like to thank [Natalia Tepluhina](https://twitter.com/n_tepluhina) and [Sebastian Markbåge](https://twitter.com/sebmarkbage) for their ideas and feedback.

Thanks to [Dan Lebowitz](https://twitter.com/lebo) for the site design and [Razvan Gradinar](https://dribbble.com/GradinarRazvan) for the sandbox design.

On the development front, thanks to [Jared Palmer](https://twitter.com/jaredpalmer) for prototype development. Thanks to [Dane Grant](https://twitter.com/danecando) and [Dustin Goodman](https://twitter.com/dustinsgoodman) from [ThisDotLabs](https://www.thisdot.co/) for their support on UI development. Thanks to [Ives van Hoorne](https://twitter.com/CompuIves), [Alex Moldovan](https://twitter.com/alexnmoldovan), [Jasper De Moor](https://twitter.com/JasperDeMoor), and [Danilo Woznica](https://twitter.com/danilowoz) from [CodeSandbox](https://codesandbox.io/) for their work with sandbox integration. Thanks to [Rick Hanlon](https://twitter.com/rickhanlonii) for spot development and design work, finessing our colors and finer details. Thanks to [Harish Kumar](https://www.strek.in/) and [Luna Ruan](https://twitter.com/lunaruan) for adding new features to the site and helping maintain it.

Huge thanks to the folks who volunteered their time to participate in the alpha and beta testing program. Your enthusiasm and invaluable feedback helped us shape these docs. A special shout out to our beta tester, [Debbie O'Brien](https://twitter.com/debs_obrien), who gave a talk about her experience using the React docs at React Conf 2021.

Finally, thanks to the React community for being the inspiration behind this effort. You are the reason we do this, and we hope that the new docs will help you use React to build any user interface that you want.

---

title: "React Labs: What We've Been Working On – June 2022"

June 15, 2022 by [Andrew Clark](https://twitter.com/acdlite), [Dan Abramov](https://twitter.com/dan_abramov), [Jan Kassens](https://twitter.com/kassens), [Joseph Savona](https://twitter.com/en_JS), [Josh Story](https://twitter.com/joshcstory), [Lauren Tan](https://twitter.com/potetotes), [Luna Ruan](https://twitter.com/lunaruan), [Mengdi Chen](https://twitter.com/mengdi_en), [Rick Hanlon](https://twitter.com/rickhanlonii), [Robert Zhang](https://twitter.com/jiaxuanzhang01), [Sathya Gunasekaran](https://twitter.com/_gsathya), [Sebastian Markbåge](https://twitter.com/sebmarkbage), and [Xuan Huang](https://twitter.com/Huxpro)

<Intro>

[React 18](https://reactjs.org/blog/2022/03/29/react-v18) was years in the making, and with it brought valuable lessons for the React team. Its release was the result of many years of research and exploring many paths. Some of those paths were successful; many more were dead-ends that led to new insights. One lesson we've learned is that it's frustrating for the community to wait for new features without having insight into these paths that we're exploring.

</Intro>

We typically have a number of projects being worked on at any time, ranging from the more experimental to the clearly defined. Looking ahead, we'd like to start regularly sharing more about what we've been working on with the community across these projects.

To set expectations, this is not a roadmap with clear timelines. Many of these projects are under active research and are difficult to put concrete ship dates on. They may possibly never even ship in their current iteration depending on what we learn. Instead, we want to share with you the problem spaces we're actively thinking about, and what we've learned so far.

Server Components {/*server-components*/}

We announced an [experimental demo of React Server Components](https://legacy.reactjs.org/blog/2020/12/21/data-fetching-with-react-server-components.html) (RSC) in December 2020. Since then we've been finishing up its dependencies in React 18, and working on changes inspired by experimental feedback.

In particular, we're abandoning the idea of having forked I/O libraries (eg react-fetch), and instead adopting an async/await model for better compatibility. This doesn't technically block RSC's release because you can also use routers for data fetching. Another change is that we're also moving away from the file extension approach in favor of [annotating boundaries](https://github.com/reactjs/rfcs/pull/189#issuecomment-1116482278).

We're working together with Vercel and Shopify to unify bundler support for shared semantics in both Webpack and Vite. Before launch, we want to make sure that the semantics of RSCs are the same across the whole React ecosystem. This is the major blocker for reaching stable.

Asset Loading {/*asset-loading*/}

Currently, assets like scripts, external styles, fonts, and images are typically preloaded and loaded using external systems. This can make it tricky to coordinate across new environments like streaming, server components, and more.

We're looking at adding APIs to preload and load deduplicated external assets through React APIs that work in all React environments.

We're also looking at having these support Suspense so you can have images, CSS, and fonts that block display until they're loaded but don't block streaming and concurrent rendering. This can help avoid ["popcorning"](https://twitter.com/sebmarkbage/status/1516852731251724293) as the visuals pop and layout shifts.

Static Server Rendering Optimizations {/*static-server-rendering-optimizations*/}

Static Site Generation (SSG) and Incremental Static Regeneration (ISR) are great ways to get performance for cacheable pages, but we think we can add features to improve performance of dynamic Server Side Rendering (SSR) – especially when most but not all of the content is cacheable. We're exploring ways to optimize server rendering utilizing compilation and static passes.

React Optimizing Compiler {/*react-compiler*/}

We gave an [early preview](https://www.youtube.com/watch?v=lGEMwh32soc) of React Forget at React Conf 2021. It's a compiler that automatically generates the equivalent of `useMemo` and `useCallback` calls to minimize the cost of re-rendering, while retaining React's programming model.

Recently, we finished a rewrite of the compiler to make it more reliable and capable. This new architecture allows us to analyze and memoize more complex patterns such as the use of [local mutations](/learn/keeping-components-pure#local-mutation-your-components-little-secret), and opens up many new compile-time optimization opportunities beyond just being on par with memoization

hooks.

We’re also working on a playground for exploring many aspects of the compiler. While the goal of the playground is to make development of the compiler easier, we think that it will make it easier to try it out and build intuition for what the compiler does. It reveals various insights into how it works under the hood, and live renders the compiler’s outputs as you type. This will be shipped together with the compiler when it’s released.

Offscreen {/*offscreen*/}

Today, if you want to hide and show a component, you have two options. One is to add or remove it from the tree completely. The problem with this approach is that the state of your UI is lost each time you unmount, including state stored in the DOM, like scroll position.

The other option is to keep the component mounted and toggle the appearance visually using CSS. This preserves the state of your UI, but it comes at a performance cost, because React must keep rendering the hidden component and all of its children whenever it receives new updates.

Offscreen introduces a third option: hide the UI visually, but deprioritize its content. The idea is similar in spirit to the `content-visibility` CSS property: when content is hidden, it doesn't need to stay in sync with the rest of the UI. React can defer the rendering work until the rest of the app is idle, or until the content becomes visible again.

Offscreen is a low level capability that unlocks high level features. Similar to React's other concurrent features like `startTransition`, in most cases you won't interact with the Offscreen API directly, but instead via an opinionated framework to implement patterns like:

* **Instant transitions.** Some routing frameworks already prefetch data to speed up subsequent navigations, like when hovering over a link. With Offscreen, they'll also be able to prerender the next screen in the background.

* **Reusable state.** Similarly, when navigating between routes or tabs, you can use Offscreen to preserve the state of the previous screen so you can switch back and pick up where you left off.

* **Virtualized list rendering.** When displaying large lists of items, virtualized list frameworks will prerender more rows than are currently visible. You can use Offscreen to prerender the hidden rows at a lower priority than the visible items in the list.

* **Backgrounded content.** We're also exploring a related feature for deprioritizing content in the background without hiding it, like when displaying a modal overlay.

Transition Tracing {/*transition-tracing*/}

Currently, React has two profiling tools. The [original Profiler](https://legacy.reactjs.org/blog/2018/09/10/introducing-the-react-profiler.html) shows an overview of all the commits in a profiling session. For each commit, it also shows all components that rendered and the amount of time it took for them to render. We also have a beta version of a [Timeline Profiler](https://github.com/reactwg/react-18/discussions/76) introduced in React 18 that shows when components schedule updates and when React works on these updates. Both of these profilers help developers identify performance problems in their code.

We’ve realized that developers don’t find knowing about individual slow commits or components out of context that useful. It’s more useful to know about what actually causes the slow commits. And that developers want to be able to track specific interactions (eg a button click, an initial load, or a page

navigation) to watch for performance regressions and to understand why an interaction was slow and how to fix it.

We previously tried to solve this issue by creating an [Interaction Tracing API](https://gist.github.com/bvaughn/8de925562903afd2e7a12554adcdda16), but it had some fundamental design flaws that reduced the accuracy of tracking why an interaction was slow and sometimes resulted in interactions never ending. We ended up [removing this API](https://github.com/facebook/react/pull/20037) because of these issues.

We are working on a new version for the Interaction Tracing API (tentatively called Transition Tracing because it is initiated via `startTransition`) that solves these problems.

New React Docs {/*new-react-docs*/}

Last year, we announced the beta version of the new React documentation website ([later shipped as react.dev](/blog/2023/03/16/introducing-react-dev)) of the new React documentation website. The new learning materials teach Hooks first and has new diagrams, illustrations, as well as many interactive examples and challenges. We took a break from that work to focus on the React 18 release, but now that React 18 is out, we're actively working to finish and ship the new documentation.

We are currently writing a detailed section about effects, as we've heard that is one of the more challenging topics for both new and experienced React users. [Synchronizing with Effects](/learn/synchronizing-with-effects) is the first published page in the series, and there are more to come in the following weeks. When we first started writing a detailed section about effects, we've realized that many common effect patterns can be simplified by adding a new primitive to React. We've shared some initial thoughts on that in the [useEvent RFC](https://github.com/reactjs/rfcs/pull/220). It is currently in early research, and we are still iterating on the idea. We appreciate the community's comments on the RFC so far, as well as the [feedback](https://github.com/reactjs/reactjs.org/issues/3308) and contributions to the ongoing documentation rewrite. We'd specifically like to thank [Harish Kumar](https://github.com/harish-sethuraman) for submitting and reviewing many improvements to the new website implementation.

Thanks to [Sophie Alpert](https://twitter.com/sophiebits) for reviewing this blog post!

---

title: "How to Upgrade to React 18"

March 08, 2022 by [Rick Hanlon](https://twitter.com/rickhanlonii)

<Intro>

As we shared in the [release post](/blog/2022/03/29/react-v18), React 18 introduces features powered by our new concurrent renderer, with a gradual adoption strategy for existing applications. In this post, we will guide you through the steps for upgrading to React 18.

Please [report any issues](https://github.com/facebook/react/issues/new/choose) you encounter while upgrading to React 18.

</Intro>

<Note>

For React Native users, React 18 will ship in a future version of React Native. This is because React 18 relies on the New React Native Architecture to benefit from the new capabilities presented in this blogpost. For more information, see the [React Conf keynote here](https://www.youtube.com/watch?v=FZ0cG47msEk&t=1530s).

</Note>

Installing {/*installing*/}

To install the latest version of React:

```bash
npm install react react-dom
```

Or if you're using yarn:

```bash
yarn add react react-dom
```

Updates to Client Rendering APIs {/*updates-to-client-rendering-apis*/}

When you first install React 18, you will see a warning in the console:

<ConsoleBlock level="error">

ReactDOM.render is no longer supported in React 18. Use createRoot instead. Until you switch to the new API, your app will behave as if it's running React 17. Learn more: https://reactjs.org/link/switch-to-createroot

</ConsoleBlock>

React 18 introduces a new root API which provides better ergonomics for managing roots. The new root API also enables the new concurrent renderer, which allows you to opt-into concurrent features.

```js
// Before
import { render } from 'react-dom';
const container = document.getElementById('app');
render(<App tab="home" />, container);

// After
```

```
import { createRoot } from 'react-dom/client';
const container = document.getElementById('app');
const root = createRoot(container); // createRoot(container!) if you use TypeScript
root.render(<App tab="home" />);
```

We've also changed `unmountComponentAtNode` to `root.unmount`:

```js
// Before
unmountComponentAtNode(container);

// After
root.unmount();
```

We've also removed the callback from render, since it usually does not have the expected result when using Suspense:

```js
// Before
const container = document.getElementById('app');
render(<App tab="home" />, container, () => {
console.log('rendered');
});

// After
function AppWithCallbackAfterRender() {
useEffect(() => {
console.log('rendered');
});

return <App tab="home" />
}

const container = document.getElementById('app');
const root = createRoot(container);
root.render(<AppWithCallbackAfterRender />);
```

<Note>

There is no one-to-one replacement for the old render callback API — it depends on your use case. See the working group post for [Replacing render with createRoot](https://github.com/reactwg/react-18/discussions/5) for more information.

</Note>

Finally, if your app uses server-side rendering with hydration, upgrade `hydrate` to `hydrateRoot`:

```js
// Before
import { hydrate } from 'react-dom';
const container = document.getElementById('app');
hydrate(<App tab="home" />, container);

// After
import { hydrateRoot } from 'react-dom/client';
const container = document.getElementById('app');
const root = hydrateRoot(container, <App tab="home" />);
// Unlike with createRoot, you don't need a separate root.render() call here.
```

For more information, see the [working group discussion here](https://github.com/reactwg/react-18/discussions/5).

<Note>

If your app doesn't work after upgrading, check whether it's wrapped in `<StrictMode>`. [Strict Mode has gotten stricter in React 18](#updates-to-strict-mode), and not all your components may be resilient to the new checks it adds in development mode. If removing Strict Mode fixes your app, you can remove it during the upgrade, and then add it back (either at the top or for a part of the tree) after you fix the issues that it's pointing out.

</Note>

Updates to Server Rendering APIs {/*updates-to-server-rendering-apis*/}

In this release, we're revamping our `react-dom/server` APIs to fully support Suspense on the server and Streaming SSR. As part of these changes, we're deprecating the old Node streaming API, which does not support incremental Suspense streaming on the server.

Using this API will now warn:

* `renderToNodeStream`: **Deprecated ■■■**

Instead, for streaming in Node environments, use:

* `renderToPipeableStream`: **New ■**

We're also introducing a new API to support streaming SSR with Suspense for modern edge runtime environments, such as Deno and Cloudflare workers:

* `renderToReadableStream`: **New ■**

The following APIs will continue working, but with limited support for Suspense:

* `renderToString`: **Limited** ■■

* `renderToStaticMarkup`: **Limited** ■■

Finally, this API will continue to work for rendering e-mails:

* `renderToStaticNodeStream`

For more information on the changes to server rendering APIs, see the working group post on [Upgrading to React 18 on the server](https://github.com/reactwg/react-18/discussions/22), a [deep dive on the new Suspense SSR Architecture](https://github.com/reactwg/react-18/discussions/37), and [Shaundai Person's](https://twitter.com/shaundai) talk on [Streaming Server Rendering with Suspense](https://www.youtube.com/watch?v=pj5N-Khihgc) at React Conf 2021.

Updates to TypeScript definitions {/*updates-to-typescript-definitions*/}

If your project uses TypeScript, you will need to update your `@types/react` and `@types/react-dom` dependencies to the latest versions. The new types are safer and catch issues that used to be ignored by the type checker. The most notable change is that the `children` prop now needs to be listed explicitly when defining props, for example:

```typescript{3}
interface MyButtonProps {
color: string;
children?: React.ReactNode;
}
```

See the [React 18 typings pull request](https://github.com/DefinitelyTyped/DefinitelyTyped/pull/56210) for a full list of type-only changes. It links to example fixes in library types so you can see how to adjust your code. You can use the [automated migration script](https://github.com/eps1lon/types-react-codemod) to help port your application code to the new and safer typings faster.

If you find a bug in the typings, please [file an issue](https://github.com/DefinitelyTyped/DefinitelyTyped/discussions/new?category=issues-with-a-types-package) in the DefinitelyTyped repo.

Automatic Batching {/*automatic-batching*/}

React 18 adds out-of-the-box performance improvements by doing more batching by default. Batching is when React groups multiple state updates into a single re-render for better performance. Before React 18, we only batched updates inside React event handlers. Updates inside of promises, setTimeout, native event handlers, or any other event were not batched in React by default:

```js
```

```
// Before React 18 only React events were batched

function handleClick() {
setCount(c => c + 1);
setFlag(f => !f);
// React will only re-render once at the end (that's batching!)
}

setTimeout(() => {
setCount(c => c + 1);
setFlag(f => !f);
// React will render twice, once for each state update (no batching)
}, 1000);
```

Starting in React 18 with `createRoot`, all updates will be automatically batched, no matter where they originate from. This means that updates inside of timeouts, promises, native event handlers or any other event will batch the same way as updates inside of React events:

```js
// After React 18 updates inside of timeouts, promises,
// native event handlers or any other event are batched.

function handleClick() {
setCount(c => c + 1);
setFlag(f => !f);
// React will only re-render once at the end (that's batching!)
}

setTimeout(() => {
setCount(c => c + 1);
setFlag(f => !f);
// React will only re-render once at the end (that's batching!)
}, 1000);
```

This is a breaking change, but we expect this to result in less work rendering, and therefore better performance in your applications. To opt-out of automatic batching, you can use `flushSync`:

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

```
function handleClick() {
flushSync(() => {
setCounter(c => c + 1);
});
// React has updated the DOM by now
flushSync(() => {
setFlag(f => !f);
});
// React has updated the DOM by now
}
```

For more information, see the [Automatic batching deep dive](https://github.com/reactwg/react-18/discussions/21).

New APIs for Libraries {/*new-apis-for-libraries*/}

In the React 18 Working Group we worked with library maintainers to create new APIs needed to support concurrent rendering for use cases specific to their use case in areas like styles, and external stores. To support React 18, some libraries may need to switch to one of the following APIs:

* `useSyncExternalStore` is a new hook that allows external stores to support concurrent reads by forcing updates to the store to be synchronous. This new API is recommended for any library that integrates with state external to React. For more information, see the [useSyncExternalStore overview post](https://github.com/reactwg/react-18/discussions/70) and [useSyncExternalStore API details](https://github.com/reactwg/react-18/discussions/86).

* `useInsertionEffect` is a new hook that allows CSS-in-JS libraries to address performance issues of injecting styles in render. Unless you've already built a CSS-in-JS library we don't expect you to ever use this. This hook will run after the DOM is mutated, but before layout effects read the new layout. This solves an issue that already exists in React 17 and below, but is even more important in React 18 because React yields to the browser during concurrent rendering, giving it a chance to recalculate layout. For more information, see the [Library Upgrade Guide for `<style>`](https://github.com/reactwg/react-18/discussions/110).

React 18 also introduces new APIs for concurrent rendering such as `startTransition`, `useDeferredValue` and `useId`, which we share more about in the [release post](/blog/2022/03/29/react-v18).

Updates to Strict Mode {/*updates-to-strict-mode*/}

In the future, we'd like to add a feature that allows React to add and remove sections of the UI while preserving state. For example, when a user tabs away from a screen and back, React should be able to immediately show the previous screen. To do this, React would unmount and remount trees using the same component state as before.

This feature will give React better performance out-of-the-box, but requires components to be resilient to effects being mounted and destroyed multiple times. Most effects will work without any changes, but

some effects assume they are only mounted or destroyed once.

To help surface these issues, React 18 introduces a new development-only check to Strict Mode. This new check will automatically unmount and remount every component, whenever a component mounts for the first time, restoring the previous state on the second mount.

Before this change, React would mount the component and create the effects:

```
* React mounts the component.
* Layout effects are created.
* Effect effects are created.
```

With Strict Mode in React 18, React will simulate unmounting and remounting the component in development mode:

```
* React mounts the component.
* Layout effects are created.
* Effect effects are created.
* React simulates unmounting the component.
* Layout effects are destroyed.
* Effects are destroyed.
* React simulates mounting the component with the previous state.
* Layout effect setup code runs
* Effect setup code runs
```

For more information, see the Working Group posts for [Adding Reusable State to StrictMode](https://github.com/reactwg/react-18/discussions/19) and [How to support Reusable State in Effects](https://github.com/reactwg/react-18/discussions/18).

Configuring Your Testing Environment {/*configuring-your-testing-environment*/}

When you first update your tests to use `createRoot`, you may see this warning in your test console:

<ConsoleBlock level="error">

The current testing environment is not configured to support act(...)

</ConsoleBlock>

To fix this, set `globalThis.IS_REACT_ACT_ENVIRONMENT` to `true` before running your test:

```js
```

// In your test setup file

globalThis.IS_REACT_ACT_ENVIRONMENT = true;

```

The purpose of the flag is to tell React that it's running in a unit test-like environment. React will log helpful warnings if you forget to wrap an update with `act`.

You can also set the flag to `false` to tell React that `act` isn't needed. This can be useful for end-to-end tests that simulate a full browser environment.

Eventually, we expect testing libraries will configure this for you automatically. For example, the [next version of React Testing Library has built-in support for React 18](https://github.com/testing-library/react-testing-library/issues/509#issuecomment-917989936) without any additional configuration.

[More background on the `act` testing API and related changes](https://github.com/reactwg/react-18/discussions/102) is available in the working group.

## Dropping Support for Internet Explorer {/*dropping-support-for-internet-explorer*/}

In this release, React is dropping support for Internet Explorer, which is [going out of support on June 15, 2022](https://blogs.windows.com/windowsexperience/2021/05/19/the-future-of-internet-explorer-on-windows-10-is-in-microsoft-edge). We're making this change now because new features introduced in React 18 are built using modern browser features such as microtasks which cannot be adequately polyfilled in IE.

If you need to support Internet Explorer we recommend you stay with React 17.

## Deprecations {/*deprecations*/}

* `react-dom`: `ReactDOM.render` has been deprecated. Using it will warn and run your app in React 17 mode.

* `react-dom`: `ReactDOM.hydrate` has been deprecated. Using it will warn and run your app in React 17 mode.

* `react-dom`: `ReactDOM.unmountComponentAtNode` has been deprecated.

* `react-dom`: `ReactDOM.renderSubtreeIntoContainer` has been deprecated.

* `react-dom/server`: `ReactDOMServer.renderToNodeStream` has been deprecated.

## Other Breaking Changes {/*other-breaking-changes*/}

* **Consistent useEffect timing**: React now always synchronously flushes effect functions if the update was triggered during a discrete user input event such as a click or a keydown event. Previously, the behavior wasn't always predictable or consistent.

* **Stricter hydration errors**: Hydration mismatches due to missing or extra text content are now treated like errors instead of warnings. React will no longer attempt to "patch up" individual nodes by inserting or deleting a node on the client in an attempt to match the server markup, and will revert to client rendering up to the closest `&lt;Suspense&gt;` boundary in the tree. This ensures the hydrated tree is consistent and avoids potential privacy and security holes that can be caused by hydration mismatches.
```

* **Suspense trees are always consistent:** If a component suspends before it's fully added to the tree, React will not add it to the tree in an incomplete state or fire its effects. Instead, React will throw away the new tree completely, wait for the asynchronous operation to finish, and then retry rendering again from scratch. React will render the retry attempt concurrently, and without blocking the browser.

* **Layout Effects with Suspense**: When a tree re-suspends and reverts to a fallback, React will now clean up layout effects, and then re-create them when the content inside the boundary is shown again. This fixes an issue which prevented component libraries from correctly measuring layout when used with Suspense.

* **New JS Environment Requirements**: React now depends on modern browsers features including `Promise`, `Symbol`, and `Object.assign`. If you support older browsers and devices such as Internet Explorer which do not provide modern browser features natively or have non-compliant implementations, consider including a global polyfill in your bundled application.

Other Notable Changes {/*other-notable-changes*/}

React {/*react*/}

* **Components can now render `undefined`:** React no longer warns if you return `undefined` from a component. This makes the allowed component return values consistent with values that are allowed in the middle of a component tree. We suggest to use a linter to prevent mistakes like forgetting a `return` statement before JSX.

* **In tests, `act` warnings are now opt-in:** If you're running end-to-end tests, the `act` warnings are unnecessary. We've introduced an [opt-in](https://github.com/reactwg/react-18/discussions/102) mechanism so you can enable them only for unit tests where they are useful and beneficial.

* **No warning about `setState` on unmounted components:** Previously, React warned about memory leaks when you call `setState` on an unmounted component. This warning was added for subscriptions, but people primarily run into it in scenarios where setting state is fine, and workarounds make the code worse. We've [removed](https://github.com/facebook/react/pull/22114) this warning.

* **No suppression of console logs:** When you use Strict Mode, React renders each component twice to help you find unexpected side effects. In React 17, we've suppressed console logs for one of the two renders to make the logs easier to read. In response to [community feedback](https://github.com/facebook/react/issues/21783) about this being confusing, we've removed the suppression. Instead, if you have React DevTools installed, the second log's renders will be displayed in grey, and there will be an option (off by default) to suppress them completely.

* **Improved memory usage:** React now cleans up more internal fields on unmount, making the impact from unfixed memory leaks that may exist in your application code less severe.

React DOM Server {/*react-dom-server*/}

* **`renderToString`:** Will no longer error when suspending on the server. Instead, it will emit the fallback HTML for the closest `<Suspense>` boundary and then retry rendering the same content on the client. It is still recommended that you switch to a streaming API like `renderToPipeableStream` or `renderToReadableStream` instead.

* **`renderToStaticMarkup`:** Will no longer error when suspending on the server. Instead, it will emit the fallback HTML for the closest `<Suspense>` boundary.

Changelog {/*changelog*/}

You can view the [full changelog here](https://github.com/facebook/react/blob/main/CHANGELOG.md).

---

title: "React v18.0"

March 29, 2022 by [The React Team](/community/team)

<Intro>

React 18 is now available on npm! In our last post, we shared step-by-step instructions for [upgrading your app to React 18](/blog/2022/03/08/react-18-upgrade-guide). In this post, we'll give an overview of what's new in React 18, and what it means for the future.

</Intro>

Our latest major version includes out-of-the-box improvements like automatic batching, new APIs like startTransition, and streaming server-side rendering with support for Suspense.

Many of the features in React 18 are built on top of our new concurrent renderer, a behind-the-scenes change that unlocks powerful new capabilities. Concurrent React is opt-in — it's only enabled when you use a concurrent feature — but we think it will have a big impact on the way people build applications.

We've spent years researching and developing support for concurrency in React, and we've taken extra care to provide a gradual adoption path for existing users. Last summer, [we formed the React 18 Working Group](/blog/2021/06/08/the-plan-for-react-18) to gather feedback from experts in the community and ensure a smooth upgrade experience for the entire React ecosystem.

In case you missed it, we shared a lot of this vision at React Conf 2021:

* In [the keynote](https://www.youtube.com/watch?v=FZ0cG47msEk&list=PLNG_1j3cPCaZZ7etkzWA7JfdmKWT0pMsa), we explain how React 18 fits into our mission to make it easy for developers to build great user experiences

* [Shruti Kapoor](https://twitter.com/shrutikapoor08) [demonstrated how to use the new features in React 18](https://www.youtube.com/watch?v=ytudH8je5ko&list=PLNG_1j3cPCaZZ7etkzWA7JfdmKWT0pMsa&index=2)

* [Shaundai Person](https://twitter.com/shaundai) gave us an overview of [streaming server rendering with Suspense](https://www.youtube.com/watch?v=pj5N-Khihgc&list=PLNG_1j3cPCaZZ7etkzWA7JfdmKWT0pMsa&index=3)

Below is a full overview of what to expect in this release, starting with Concurrent Rendering.

<Note>

For React Native users, React 18 will ship in React Native with the New React Native Architecture. For more information, see the [React Conf keynote here](https://www.youtube.com/watch?v=FZ0cG47msEk&t=1530s).

</Note>

What is Concurrent React? {/*what-is-concurrent-react*/}

The most important addition in React 18 is something we hope you never have to think about: concurrency. We think this is largely true for application developers, though the story may be a bit more complicated for library maintainers.

Concurrency is not a feature, per se. It's a new behind-the-scenes mechanism that enables React to prepare multiple versions of your UI at the same time. You can think of concurrency as an implementation detail — it's valuable because of the features that it unlocks. React uses sophisticated techniques in its internal implementation, like priority queues and multiple buffering. But you won't see those concepts anywhere in our public APIs.

When we design APIs, we try to hide implementation details from developers. As a React developer, you focus on *what* you want the user experience to look like, and React handles *how* to deliver that experience. So we don’t expect React developers to know how concurrency works under the hood.

However, Concurrent React is more important than a typical implementation detail — it's a foundational update to React's core rendering model. So while it's not super important to know how concurrency works, it may be worth knowing what it is at a high level.

A key property of Concurrent React is that rendering is interruptible. When you first upgrade to React 18, before adding any concurrent features, updates are rendered the same as in previous versions of React — in a single, uninterrupted, synchronous transaction. With synchronous rendering, once an update starts rendering, nothing can interrupt it until the user can see the result on screen.

In a concurrent render, this is not always the case. React may start rendering an update, pause in the middle, then continue later. It may even abandon an in-progress render altogether. React guarantees that the UI will appear consistent even if a render is interrupted. To do this, it waits to perform DOM mutations until the end, once the entire tree has been evaluated. With this capability, React can prepare new screens in the background without blocking the main thread. This means the UI can respond immediately to user input even if it’s in the middle of a large rendering task, creating a fluid user experience.

Another example is reusable state. Concurrent React can remove sections of the UI from the screen, then add them back later while reusing the previous state. For example, when a user tabs away from a screen and back, React should be able to restore the previous screen in the same state it was in before. In an upcoming minor, we're planning to add a new component called `<Offscreen>` that implements this pattern. Similarly, you’ll be able to use Offscreen to prepare new UI in the background so that it’s ready before the user reveals it.

Concurrent rendering is a powerful new tool in React and most of our new features are built to take advantage of it, including Suspense, transitions, and streaming server rendering. But React 18 is just the beginning of what we aim to build on this new foundation.

Gradually Adopting Concurrent Features {/*gradually-adopting-concurrent-features*/}

Technically, concurrent rendering is a breaking change. Because concurrent rendering is interruptible, components behave slightly differently when it is enabled.

In our testing, we've upgraded thousands of components to React 18. What we've found is that nearly all existing components "just work" with concurrent rendering, without any changes. However, some of them may require some additional migration effort. Although the changes are usually small, you'll still have the ability to make them at your own pace. The new rendering behavior in React 18 is **only

enabled in the parts of your app that use new features.**

The overall upgrade strategy is to get your application running on React 18 without breaking existing code. Then you can gradually start adding concurrent features at your own pace. You can use [`<StrictMode>`](/reference/react/StrictMode) to help surface concurrency-related bugs during development. Strict Mode doesn't affect production behavior, but during development it will log extra warnings and double-invoke functions that are expected to be idempotent. It won't catch everything, but it's effective at preventing the most common types of mistakes.

After you upgrade to React 18, you'll be able to start using concurrent features immediately. For example, you can use startTransition to navigate between screens without blocking user input. Or useDeferredValue to throttle expensive re-renders.

However, long term, we expect the main way you'll add concurrency to your app is by using a concurrent-enabled library or framework. In most cases, you won't interact with concurrent APIs directly. For example, instead of developers calling startTransition whenever they navigate to a new screen, router libraries will automatically wrap navigations in startTransition.

It may take some time for libraries to upgrade to be concurrent compatible. We've provided new APIs to make it easier for libraries to take advantage of concurrent features. In the meantime, please be patient with maintainers as we work to gradually migrate the React ecosystem.

For more info, see our previous post: [How to upgrade to React 18](/blog/2022/03/08/react-18-upgrade-guide).

Suspense in Data Frameworks {/*suspense-in-data-frameworks*/}

In React 18, you can start using [Suspense](/reference/react/Suspense) for data fetching in opinionated frameworks like Relay, Next.js, Hydrogen, or Remix. Ad hoc data fetching with Suspense is technically possible, but still not recommended as a general strategy.

In the future, we may expose additional primitives that could make it easier to access your data with Suspense, perhaps without the use of an opinionated framework. However, Suspense works best when it's deeply integrated into your application's architecture: your router, your data layer, and your server rendering environment. So even long term, we expect that libraries and frameworks will play a crucial role in the React ecosystem.

As in previous versions of React, you can also use Suspense for code splitting on the client with React.lazy. But our vision for Suspense has always been about much more than loading code — the goal is to extend support for Suspense so that eventually, the same declarative Suspense fallback can handle any asynchronous operation (loading code, data, images, etc).

Server Components is Still in Development {/*server-components-is-still-in-development*/}

[**Server Components**](/blog/2020/12/21/data-fetching-with-react-server-components) is an upcoming feature that allows developers to build apps that span the server and client, combining the rich interactivity of client-side apps with the improved performance of traditional server rendering. Server Components is not inherently coupled to Concurrent React, but it's designed to work best with concurrent features like Suspense and streaming server rendering.

Server Components is still experimental, but we expect to release an initial version in a minor 18.x release. In the meantime, we're working with frameworks like Next.js, Hydrogen, and Remix to advance the proposal and get it ready for broad adoption.

What's New in React 18 {/*whats-new-in-react-18*/}

New Feature: Automatic Batching {/*new-feature-automatic-batching*/}

Batching is when React groups multiple state updates into a single re-render for better performance. Without automatic batching, we only batched updates inside React event handlers. Updates inside of promises, setTimeout, native event handlers, or any other event were not batched in React by default. With automatic batching, these updates will be batched automatically:

```js
// Before: only React events were batched.
setTimeout(() => {
setCount(c => c + 1);
setFlag(f => !f);
// React will render twice, once for each state update (no batching)
}, 1000);

// After: updates inside of timeouts, promises,
// native event handlers or any other event are batched.
setTimeout(() => {
setCount(c => c + 1);
setFlag(f => !f);
// React will only re-render once at the end (that's batching!)
}, 1000);
```

For more info, see this post for [Automatic batching for fewer renders in React 18](https://github.com/reactwg/react-18/discussions/21).

New Feature: Transitions {/*new-feature-transitions*/}

A transition is a new concept in React to distinguish between urgent and non-urgent updates.

* **Urgent updates** reflect direct interaction, like typing, clicking, pressing, and so on.
* **Transition updates** transition the UI from one view to another.

Urgent updates like typing, clicking, or pressing, need immediate response to match our intuitions about how physical objects behave. Otherwise they feel "wrong". However, transitions are different because the user doesn’t expect to see every intermediate value on screen.

For example, when you select a filter in a dropdown, you expect the filter button itself to respond immediately when you click. However, the actual results may transition separately. A small delay would be imperceptible and often expected. And if you change the filter again before the results are done rendering, you only care to see the latest results.

Typically, for the best user experience, a single user input should result in both an urgent update and a non-urgent one. You can use startTransition API inside an input event to inform React which updates are urgent and which are "transitions":

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

// Urgent: Show what was typed
setInputValue(input);

// Mark any state updates inside as transitions
startTransition(() => {
// Transition: Show the results
setSearchQuery(input);
});
```

Updates wrapped in startTransition are handled as non-urgent and will be interrupted if more urgent updates like clicks or key presses come in. If a transition gets interrupted by the user (for example, by typing multiple characters in a row), React will throw out the stale rendering work that wasn’t finished and render only the latest update.

* `useTransition`: a hook to start transitions, including a value to track the pending state.
* `startTransition`: a method to start transitions when the hook cannot be used.

Transitions will opt in to concurrent rendering, which allows the update to be interrupted. If the content re-suspends, transitions also tell React to continue showing the current content while rendering the transition content in the background (see the [Suspense RFC](https://github.com/reactjs/rfcs/blob/main/text/0213-suspense-in-react-18.md) for more info).

[See docs for transitions here](/reference/react/useTransition).

New Suspense Features {/*new-suspense-features*/}

Suspense lets you declaratively specify the loading state for a part of the component tree if it's not yet ready to be displayed:

```js
<Suspense fallback={<Spinner />}>
<Comments />
</Suspense>
```

Suspense makes the "UI loading state" a first-class declarative concept in the React programming model. This lets us build higher-level features on top of it.

We introduced a limited version of Suspense several years ago. However, the only supported use case was code splitting with React.lazy, and it wasn't supported at all when rendering on the server.

In React 18, we've added support for Suspense on the server and expanded its capabilities using concurrent rendering features.

Suspense in React 18 works best when combined with the transition API. If you suspend during a transition, React will prevent already-visible content from being replaced by a fallback. Instead, React will delay the render until enough data has loaded to prevent a bad loading state.

For more, see the RFC for [Suspense in React 18](https://github.com/reactjs/rfcs/blob/main/text/0213-suspense-in-react-18.md).

New Client and Server Rendering APIs {/*new-client-and-server-rendering-apis*/}

In this release we took the opportunity to redesign the APIs we expose for rendering on the client and server. These changes allow users to continue using the old APIs in React 17 mode while they upgrade to the new APIs in React 18.

React DOM Client {/*react-dom-client*/}

These new APIs are now exported from `react-dom/client`:

* `createRoot`: New method to create a root to `render` or `unmount`. Use it instead of `ReactDOM.render`. New features in React 18 don't work without it.

* `hydrateRoot`: New method to hydrate a server rendered application. Use it instead of `ReactDOM.hydrate` in conjunction with the new React DOM Server APIs. New features in React 18 don't work without it.

Both `createRoot` and `hydrateRoot` accept a new option called `onRecoverableError` in case you want to be notified when React recovers from errors during rendering or hydration for logging. By default, React will use [`reportError`](https://developer.mozilla.org/en-US/docs/Web/API/reportError), or `console.error` in the older browsers.

[See docs for React DOM Client here](/reference/react-dom/client).

React DOM Server {/*react-dom-server*/}

These new APIs are now exported from `react-dom/server` and have full support for streaming Suspense on the server:

* `renderToPipeableStream`: for streaming in Node environments.

* `renderToReadableStream`: for modern edge runtime environments, such as Deno and Cloudflare workers.

The existing `renderToString` method keeps working but is discouraged.

[See docs for React DOM Server here](/reference/react-dom/server).

New Strict Mode Behaviors {/*new-strict-mode-behaviors*/}

In the future, we'd like to add a feature that allows React to add and remove sections of the UI while preserving state. For example, when a user tabs away from a screen and back, React should be able to immediately show the previous screen. To do this, React would unmount and remount trees using the same component state as before.

This feature will give React apps better performance out-of-the-box, but requires components to be resilient to effects being mounted and destroyed multiple times. Most effects will work without any changes, but some effects assume they are only mounted or destroyed once.

To help surface these issues, React 18 introduces a new development-only check to Strict Mode. This new check will automatically unmount and remount every component, whenever a component mounts for the first time, restoring the previous state on the second mount.

Before this change, React would mount the component and create the effects:

```

* React mounts the component.

* Layout effects are created.

* Effects are created.

```

With Strict Mode in React 18, React will simulate unmounting and remounting the component in development mode:

```

* React mounts the component.

* Layout effects are created.

* Effects are created.

* React simulates unmounting the component.

* Layout effects are destroyed.

* Effects are destroyed.

* React simulates mounting the component with the previous state.

* Layout effects are created.

* Effects are created.

```

[See docs for ensuring reusable state here](/reference/react/StrictMode#fixing-bugs-found-by-re-running-effects-in-development).

New Hooks {/*new-hooks*/}

useId {/*useid*/}

`useId` is a new hook for generating unique IDs on both the client and server, while avoiding hydration mismatches. It is primarily useful for component libraries integrating with accessibility APIs that require

unique IDs. This solves an issue that already exists in React 17 and below, but it's even more important in React 18 because of how the new streaming server renderer delivers HTML out-of-order. [See docs here](/reference/react/useId).

> Note
>
> `useId` is **not** for generating [keys in a list](/learn/rendering-lists#where-to-get-your-key). Keys should be generated from your data.

useTransition {/*usetransition*/}

`useTransition` and `startTransition` let you mark some state updates as not urgent. Other state updates are considered urgent by default. React will allow urgent state updates (for example, updating a text input) to interrupt non-urgent state updates (for example, rendering a list of search results). [See docs here](/reference/react/useTransition)

useDeferredValue {/*usedeferredvalue*/}

`useDeferredValue` lets you defer re-rendering a non-urgent part of the tree. It is similar to debouncing, but has a few advantages compared to it. There is no fixed time delay, so React will attempt the deferred render right after the first render is reflected on the screen. The deferred render is interruptible and doesn't block user input. [See docs here](/reference/react/useDeferredValue).

useSyncExternalStore {/*usesyncexternalstore*/}

`useSyncExternalStore` is a new hook that allows external stores to support concurrent reads by forcing updates to the store to be synchronous. It removes the need for useEffect when implementing subscriptions to external data sources, and is recommended for any library that integrates with state external to React. [See docs here](/reference/react/useSyncExternalStore).

> Note
>
> `useSyncExternalStore` is intended to be used by libraries, not application code.

useInsertionEffect {/*useinsertioneffect*/}

`useInsertionEffect` is a new hook that allows CSS-in-JS libraries to address performance issues of injecting styles in render. Unless you've already built a CSS-in-JS library we don't expect you to ever use this. This hook will run after the DOM is mutated, but before layout effects read the new layout. This solves an issue that already exists in React 17 and below, but is even more important in React 18 because React yields to the browser during concurrent rendering, giving it a chance to recalculate layout. [See docs here](/reference/react/useInsertionEffect).

> Note
>
> `useInsertionEffect` is intended to be used by libraries, not application code.

How to Upgrade {/*how-to-upgrade*/}

See [How to Upgrade to React 18](/blog/2022/03/08/react-18-upgrade-guide) for step-by-step instructions and a full list of breaking and notable changes.

Changelog {/*changelog*/}

React {/*react*/}

* Add `useTransition` and `useDeferredValue` to separate urgent updates from transitions. ([#10426](https://github.com/facebook/react/pull/10426), [#10715](https://github.com/facebook/react/pull/10715), [#15593](https://github.com/facebook/react/pull/15593), [#15272](https://github.com/facebook/react/pull/15272), [#15578](https://github.com/facebook/react/pull/15578), [#15769](https://github.com/facebook/react/pull/15769), [#17058](https://github.com/facebook/react/pull/17058), [#18796](https://github.com/facebook/react/pull/18796), [#19121](https://github.com/facebook/react/pull/19121), [#19703](https://github.com/facebook/react/pull/19703), [#19719](https://github.com/facebook/react/pull/19719), [#19724](https://github.com/facebook/react/pull/19724), [#20672](https://github.com/facebook/react/pull/20672), [#20976](https://github.com/facebook/react/pull/20976) by [@acdlite](https://github.com/acdlite), [@lunaruan](https://github.com/lunaruan), [@rickhanlonii](https://github.com/rickhanlonii), and [@sebmarkbage](https://github.com/sebmarkbage))

* Add `useId` for generating unique IDs. ([#17322](https://github.com/facebook/react/pull/17322), [#18576](https://github.com/facebook/react/pull/18576), [#22644](https://github.com/facebook/react/pull/22644), [#22672](https://github.com/facebook/react/pull/22672), [#21260](https://github.com/facebook/react/pull/21260) by [@acdlite](https://github.com/acdlite), [@lunaruan](https://github.com/lunaruan), and [@sebmarkbage](https://github.com/sebmarkbage))

* Add `useSyncExternalStore` to help external store libraries integrate with React. ([#15022](https://github.com/facebook/react/pull/15022), [#18000](https://github.com/facebook/react/pull/18000), [#18771](https://github.com/facebook/react/pull/18771), [#22211](https://github.com/facebook/react/pull/22211), [#22292](https://github.com/facebook/react/pull/22292), [#22239](https://github.com/facebook/react/pull/22239), [#22347](https://github.com/facebook/react/pull/22347), [#23150](https://github.com/facebook/react/pull/23150) by [@acdlite](https://github.com/acdlite), [@bvaughn](https://github.com/bvaughn), and [@drarmstr](https://github.com/drarmstr))

* Add `startTransition` as a version of `useTransition` without pending feedback. ([#19696](https://github.com/facebook/react/pull/19696) by [@rickhanlonii](https://github.com/rickhanlonii))

* Add `useInsertionEffect` for CSS-in-JS libraries. ([#21913](https://github.com/facebook/react/pull/21913) by [@rickhanlonii](https://github.com/rickhanlonii))

* Make Suspense remount layout effects when content reappears. ([#19322](https://github.com/facebook/react/pull/19322), [#19374](https://github.com/facebook/react/pull/19374), [#19523](https://github.com/facebook/react/pull/19523),

[#20625](https://github.com/facebook/react/pull/20625),
[#21079](https://github.com/facebook/react/pull/21079) by [@acdlite](https://github.com/acdlite),
[@bvaughn](https://github.com/bvaughn), and [@lunaruan](https://github.com/lunaruan))

* Make `<StrictMode>` re-run effects to check for restorable state.
([#19523](https://github.com/facebook/react/pull/19523) ,
[#21418](https://github.com/facebook/react/pull/21418) by [@bvaughn](https://github.com/bvaughn)
and [@lunaruan](https://github.com/lunaruan))

* Assume Symbols are always available. ([#23348](https://github.com/facebook/react/pull/23348) by
[@sebmarkbage](https://github.com/sebmarkbage))

* Remove `object-assign` polyfill. ([#23351](https://github.com/facebook/react/pull/23351) by
[@sebmarkbage](https://github.com/sebmarkbage))

* Remove unsupported `unstable_changedBits` API.
([#20953](https://github.com/facebook/react/pull/20953) by [@acdlite](https://github.com/acdlite))

* Allow components to render undefined. ([#21869](https://github.com/facebook/react/pull/21869) by
[@rickhanlonii](https://github.com/rickhanlonii))

* Flush `useEffect` resulting from discrete events like clicks synchronously.
([#21150](https://github.com/facebook/react/pull/21150) by [@acdlite](https://github.com/acdlite))

* Suspense `fallback={undefined}` now behaves the same as `null` and isn't ignored.
([#21854](https://github.com/facebook/react/pull/21854) by
[@rickhanlonii](https://github.com/rickhanlonii))

* Consider all `lazy()` resolving to the same component equivalent.
([#20357](https://github.com/facebook/react/pull/20357) by
[@sebmarkbage](https://github.com/sebmarkbage))

* Don't patch console during first render. ([#22308](https://github.com/facebook/react/pull/22308) by
[@lunaruan](https://github.com/lunaruan))

* Improve memory usage. ([#21039](https://github.com/facebook/react/pull/21039) by
[@bgirard](https://github.com/bgirard))

* Improve messages if string coercion throws (Temporal.*, Symbol, etc.)
([#22064](https://github.com/facebook/react/pull/22064) by
[@justingrant](https://github.com/justingrant))

* Use `setImmediate` when available over `MessageChannel`.
([#20834](https://github.com/facebook/react/pull/20834) by [@gaearon](https://github.com/gaearon))

* Fix context failing to propagate inside suspended trees.
([#23095](https://github.com/facebook/react/pull/23095) by [@gaearon](https://github.com/gaearon))

* Fix `useReducer` observing incorrect props by removing the eager bailout mechanism.
([#22445](https://github.com/facebook/react/pull/22445) by
[@josephsavona](https://github.com/josephsavona))

* Fix `setState` being ignored in Safari when appending iframes.
([#23111](https://github.com/facebook/react/pull/23111) by [@gaearon](https://github.com/gaearon))

* Fix a crash when rendering `ZonedDateTime` in the tree.
([#20617](https://github.com/facebook/react/pull/20617) by [@dimaqq](https://github.com/dimaqq))

* Fix a crash when document is set to `null` in tests.
([#22695](https://github.com/facebook/react/pull/22695) by [@SimenB](https://github.com/SimenB))

* Fix `onLoad` not triggering when concurrent features are on.
([#23316](https://github.com/facebook/react/pull/23316) by [@gnoff](https://github.com/gnoff))

* Fix a warning when a selector returns `NaN`. ([#23333](https://github.com/facebook/react/pull/23333) by [@hachibeeDI](https://github.com/hachibeeDI))

* Fix a crash when document is set to `null` in tests.
([#22695](https://github.com/facebook/react/pull/22695) by [@SimenB](https://github.com/SimenB))

* Fix the generated license header. ([#23004](https://github.com/facebook/react/pull/23004) by [@vitaliemiron](https://github.com/vitaliemiron))

* Add `package.json` as one of the entry points.
([#22954](https://github.com/facebook/react/pull/22954) by [@Jack](https://github.com/Jack-Works))

* Allow suspending outside a Suspense boundary.
([#23267](https://github.com/facebook/react/pull/23267) by [@acdlite](https://github.com/acdlite))

* Log a recoverable error whenever hydration fails.
([#23319](https://github.com/facebook/react/pull/23319) by [@acdlite](https://github.com/acdlite))

React DOM {/*react-dom*/}

* Add `createRoot` and `hydrateRoot`. ([#10239](https://github.com/facebook/react/pull/10239),
[#11225](https://github.com/facebook/react/pull/11225),
[#12117](https://github.com/facebook/react/pull/12117),
[#13732](https://github.com/facebook/react/pull/13732),
[#15502](https://github.com/facebook/react/pull/15502),
[#15532](https://github.com/facebook/react/pull/15532),
[#17035](https://github.com/facebook/react/pull/17035),
[#17165](https://github.com/facebook/react/pull/17165),
[#20669](https://github.com/facebook/react/pull/20669),
[#20748](https://github.com/facebook/react/pull/20748),
[#20888](https://github.com/facebook/react/pull/20888),
[#21072](https://github.com/facebook/react/pull/21072),
[#21417](https://github.com/facebook/react/pull/21417),
[#21652](https://github.com/facebook/react/pull/21652),
[#21687](https://github.com/facebook/react/pull/21687),
[#23207](https://github.com/facebook/react/pull/23207),
[#23385](https://github.com/facebook/react/pull/23385) by [@acdlite](https://github.com/acdlite),
[@bvaughn](https://github.com/bvaughn), [@gaearon](https://github.com/gaearon),
[@lunaruan](https://github.com/lunaruan), [@rickhanlonii](https://github.com/rickhanlonii),
[@trueadm](https://github.com/trueadm), and [@sebmarkbage](https://github.com/sebmarkbage))

* Add selective hydration. ([#14717](https://github.com/facebook/react/pull/14717),
[#14884](https://github.com/facebook/react/pull/14884),
[#16725](https://github.com/facebook/react/pull/16725),
[#16880](https://github.com/facebook/react/pull/16880),
[#17004](https://github.com/facebook/react/pull/17004),
[#22416](https://github.com/facebook/react/pull/22416),
[#22629](https://github.com/facebook/react/pull/22629),
[#22448](https://github.com/facebook/react/pull/22448),
[#22856](https://github.com/facebook/react/pull/22856),
[#23176](https://github.com/facebook/react/pull/23176) by [@acdlite](https://github.com/acdlite),
[@gaearon](https://github.com/gaearon), [@salazarm](https://github.com/salazarm), and
[@sebmarkbage](https://github.com/sebmarkbage))

* Add `aria-description` to the list of known ARIA attributes.
([#22142](https://github.com/facebook/react/pull/22142) by
[@mahyareb](https://github.com/mahyareb))

* Add `onResize` event to video elements. ([#21973](https://github.com/facebook/react/pull/21973) by
[@rileyjshaw](https://github.com/rileyjshaw))

* Add `imageSizes` and `imageSrcSet` to known props.
([#22550](https://github.com/facebook/react/pull/22550) by [@eps1lon](https://github.com/eps1lon))

* Allow non-string `<option>` children if `value` is provided.
([#21431](https://github.com/facebook/react/pull/21431) by
[@sebmarkbage](https://github.com/sebmarkbage))

* Fix `aspectRatio` style not being applied. ([#21100](https://github.com/facebook/react/pull/21100) by
[@gaearon](https://github.com/gaearon))

* Warn if `renderSubtreeIntoContainer` is called.
([#23355](https://github.com/facebook/react/pull/23355) by [@acdlite](https://github.com/acdlite))

React DOM Server {/*react-dom-server-1*/}

* Add the new streaming renderer. ([#14144](https://github.com/facebook/react/pull/14144),
[#20970](https://github.com/facebook/react/pull/20970),
[#21056](https://github.com/facebook/react/pull/21056),
[#21255](https://github.com/facebook/react/pull/21255),
[#21200](https://github.com/facebook/react/pull/21200),
[#21257](https://github.com/facebook/react/pull/21257),
[#21276](https://github.com/facebook/react/pull/21276),
[#22443](https://github.com/facebook/react/pull/22443),
[#22450](https://github.com/facebook/react/pull/22450),
[#23247](https://github.com/facebook/react/pull/23247),
[#24025](https://github.com/facebook/react/pull/24025),
[#24030](https://github.com/facebook/react/pull/24030) by
[@sebmarkbage](https://github.com/sebmarkbage))

* Fix context providers in SSR when handling multiple requests.
([#23171](https://github.com/facebook/react/pull/23171) by [@frandiox](https://github.com/frandiox))

* Revert to client render on text mismatch. ([#23354](https://github.com/facebook/react/pull/23354) by
[@acdlite](https://github.com/acdlite))

* Deprecate `renderToNodeStream`. ([#23359](https://github.com/facebook/react/pull/23359) by
[@sebmarkbage](https://github.com/sebmarkbage))

* Fix a spurious error log in the new server renderer.
([#24043](https://github.com/facebook/react/pull/24043) by [@eps1lon](https://github.com/eps1lon))

* Fix a bug in the new server renderer. ([#22617](https://github.com/facebook/react/pull/22617) by
[@shuding](https://github.com/shuding))

* Ignore function and symbol values inside custom elements on the server.
([#21157](https://github.com/facebook/react/pull/21157) by
[@sebmarkbage](https://github.com/sebmarkbage))

React DOM Test Utils {/*react-dom-test-utils*/}

* Throw when `act` is used in production. ([#21686](https://github.com/facebook/react/pull/21686) by [@acdlite](https://github.com/acdlite))

* Support disabling spurious act warnings with `global.IS_REACT_ACT_ENVIRONMENT`. ([#22561](https://github.com/facebook/react/pull/22561) by [@acdlite](https://github.com/acdlite))

* Expand act warning to cover all APIs that might schedule React work. ([#22607](https://github.com/facebook/react/pull/22607) by [@acdlite](https://github.com/acdlite))

* Make `act` batch updates. ([#21797](https://github.com/facebook/react/pull/21797) by [@acdlite](https://github.com/acdlite))

* Remove warning for dangling passive effects. ([#22609](https://github.com/facebook/react/pull/22609) by [@acdlite](https://github.com/acdlite))

React Refresh {/*react-refresh*/}

* Track late-mounted roots in Fast Refresh. ([#22740](https://github.com/facebook/react/pull/22740) by [@anc95](https://github.com/anc95))

* Add `exports` field to `package.json`. ([#23087](https://github.com/facebook/react/pull/23087) by [@otakustay](https://github.com/otakustay))

Server Components (Experimental) {/*server-components-experimental*/}

* Add Server Context support. ([#23244](https://github.com/facebook/react/pull/23244) by [@salazarm](https://github.com/salazarm))

* Add `lazy` support. ([#24068](https://github.com/facebook/react/pull/24068) by [@gnoff](https://github.com/gnoff))

* Update webpack plugin for webpack 5 ([#22739](https://github.com/facebook/react/pull/22739) by [@michenly](https://github.com/michenly))

* Fix a mistake in the Node loader. ([#22537](https://github.com/facebook/react/pull/22537) by [@btea](https://github.com/btea))

* Use `globalThis` instead of `window` for edge environments. ([#22777](https://github.com/facebook/react/pull/22777) by [@huozhi](https://github.com/huozhi))

---

title: "Introducing Zero-Bundle-Size React Server Components"

December 21, 2020 by [Dan Abramov](https://twitter.com/dan_abramov), [Lauren Tan](https://twitter.com/potetotes), [Joseph Savona](https://twitter.com/en_JS), and [Sebastian Markbåge](https://twitter.com/sebmarkbage)

<Intro>

2020 has been a long year. As it comes to an end we wanted to share a special Holiday Update on our research into zero-bundle-size **React Server Components**.

</Intro>

To introduce React Server Components, we have prepared a talk and a demo. If you want, you can check them out during the holidays, or later when work picks back up in the new year.

<YouTubeIframe src="https://www.youtube.com/embed/TQQPAU21ZUw" />

React Server Components are still in research and development. We are sharing this work in the spirit of transparency and to get initial feedback from the React community. There will be plenty of time for that, so **don't feel like you have to catch up right now!**

If you want to check them out, we recommend going in the following order:

1. **Watch the talk** to learn about React Server Components and see the demo.

2. **[Clone the demo](http://github.com/reactjs/server-components-demo)** to play with React Server Components on your computer.

3. **[Read the RFC (with FAQ at the end)](https://github.com/reactjs/rfcs/pull/188)** for a deeper technical breakdown and to provide feedback.

We are excited to hear from you on the RFC or in replies to the [@reactjs](https://twitter.com/reactjs) Twitter handle. Happy holidays, stay safe, and see you next year!

---

title: Escape Hatches

<Intro>

Some of your components may need to control and synchronize with systems outside of React. For example, you might need to focus an input using the browser API, play and pause a video player implemented without React, or connect and listen to messages from a remote server. In this chapter, you'll learn the escape hatches that let you "step outside" React and connect to external systems. Most of your application logic and data flow should not rely on these features.

</Intro>

<YouWillLearn isChapter={true}>

* [How to "remember" information without re-rendering](/learn/referencing-values-with-refs)
* [How to access DOM elements managed by React](/learn/manipulating-the-dom-with-refs)
* [How to synchronize components with external systems](/learn/synchronizing-with-effects)
* [How to remove unnecessary Effects from your components](/learn/you-might-not-need-an-effect)
* [How an Effect's lifecycle is different from a component's](/learn/lifecycle-of-reactive-effects)
* [How to prevent some values from re-triggering Effects](/learn/separating-events-from-effects)
* [How to make your Effect re-run less often](/learn/removing-effect-dependencies)
* [How to share logic between components](/learn/reusing-logic-with-custom-hooks)

</YouWillLearn>

Referencing values with refs {/*referencing-values-with-refs*/}

When you want a component to "remember" some information, but you don't want that information to [trigger new renders](/learn/render-and-commit), you can use a *ref*:

```js
const ref = useRef(0);
```

Like state, refs are retained by React between re-renders. However, setting state re-renders a component. Changing a ref does not! You can access the current value of that ref through the `ref.current` property.

<Sandpack>

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

export default function Counter() {
let ref = useRef(0);

function handleClick() {
ref.current = ref.current + 1;
alert('You clicked ' + ref.current + ' times!');
}

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

</Sandpack>

A ref is like a secret pocket of your component that React doesn't track. For example, you can use refs to store [timeout IDs](https://developer.mozilla.org/en-US/docs/Web/API/setTimeout#return_value), [DOM elements](https://developer.mozilla.org/en-US/docs/Web/API/Element), and other objects that don't impact the component's rendering output.

<LearnMore path="/learn/referencing-values-with-refs">

Read **[Referencing Values with Refs](/learn/referencing-values-with-refs)** to learn how to use refs to remember information.

</LearnMore>

Manipulating the DOM with refs {/*manipulating-the-dom-with-refs*/}

React automatically updates the DOM to match your render output, so your components won't often need to manipulate it. However, sometimes you might need access to the DOM elements managed by React—for example, to focus a node, scroll to it, or measure its size and position. There is no built-in way to do those things in React, so you will need a ref to the DOM node. For example, clicking the button will focus the input using a ref:

<Sandpack>

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

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

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

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

</Sandpack>

<LearnMore path="/learn/manipulating-the-dom-with-refs">

Read **[Manipulating the DOM with Refs](/learn/manipulating-the-dom-with-refs)** to learn how to access DOM elements managed by React.

</LearnMore>

Synchronizing with Effects {/*synchronizing-with-effects*/}

Some components need to synchronize with external systems. For example, you might want to control a non-React component based on the React state, set up a server connection, or send an analytics log when a component appears on the screen. Unlike event handlers, which let you handle particular events, *Effects* let you run some code after rendering. Use them to synchronize your component with a system outside of React.

Press Play/Pause a few times and see how the video player stays synchronized to the `isPlaying` prop value:

<Sandpack>

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

function VideoPlayer({ src, isPlaying }) {
const ref = useRef(null);

useEffect(() => {
if (isPlaying) {
ref.current.play();
} else {
ref.current.pause();
}
}, [isPlaying]);

return <video ref={ref} src={src} loop playsInline />;
}

export default function App() {
const [isPlaying, setIsPlaying] = useState(false);
return (
<>
<button onClick={() => setIsPlaying(!isPlaying)}>
{isPlaying ? 'Pause' : 'Play'}
</button>
<VideoPlayer
isPlaying={isPlaying}
src="https://interactive-examples.mdn.mozilla.net/media/cc0-videos/flower.mp4"
/>
</>
);
}
```

```

```css
button { display: block; margin-bottom: 20px; }
video { width: 250px; }
```

</Sandpack>

Many Effects also "clean up" after themselves. For example, an Effect that sets up a connection to a chat server should return a *cleanup function* that tells React how to disconnect your component from that server:

<Sandpack>

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

export default function ChatRoom() {
useEffect(() => {
const connection = createConnection();
connection.connect();
return () => connection.disconnect();
}, []);
return <h1>Welcome to the chat!</h1>;
}
```

```js chat.js
export function createConnection() {
// A real implementation would actually connect to the server
return {
connect() {
console.log('■ Connecting...');
},
disconnect() {
console.log('■ Disconnected.');
}
};
}
```
```

```

```css
input { display: block; margin-bottom: 20px; }
```

</Sandpack>

In development, React will immediately run and clean up your Effect one extra time. This is why you see `"■ Connecting..."` printed twice. This ensures that you don't forget to implement the cleanup function.

<LearnMore path="/learn/synchronizing-with-effects">

Read **[Synchronizing with Effects](/learn/synchronizing-with-effects)** to learn how to synchronize components with external systems.

</LearnMore>

You Might Not Need An Effect {/*you-might-not-need-an-effect*/}

Effects are an escape hatch from the React paradigm. They let you "step outside" of React and synchronize your components with some external system. If there is no external system involved (for example, if you want to update a component's state when some props or state change), you shouldn't need an Effect. Removing unnecessary Effects will make your code easier to follow, faster to run, and less error-prone.

There are two common cases in which you don't need Effects:

- **You don't need Effects to transform data for rendering.**
- **You don't need Effects to handle user events.**

For example, you don't need an Effect to adjust some state based on other state:

```js {5-9}
function Form() {
const [firstName, setFirstName] = useState('Taylor');
const [lastName, setLastName] = useState('Swift');

// ■ Avoid: redundant state and unnecessary Effect
const [fullName, setFullName] = useState('');
useEffect(() => {
setFullName(firstName + ' ' + lastName);
}, [firstName, lastName]);
// ...
}
```
```

Instead, calculate as much as you can while rendering:

```js {4-5}
function Form() {
const [firstName, setFirstName] = useState('Taylor');
const [lastName, setLastName] = useState('Swift');
// ■ Good: calculated during rendering
const fullName = firstName + ' ' + lastName;
// ...
}
```

However, you *do* need Effects to synchronize with external systems.

<LearnMore path="/learn/you-might-not-need-an-effect">

Read **[You Might Not Need an Effect](/learn/you-might-not-need-an-effect)** to learn how to remove unnecessary Effects.

</LearnMore>

Lifecycle of reactive effects {/*lifecycle-of-reactive-effects*/}

Effects have a different lifecycle from components. Components may mount, update, or unmount. An Effect can only do two things: to start synchronizing something, and later to stop synchronizing it. This cycle can happen multiple times if your Effect depends on props and state that change over time.

This Effect depends on the value of the `roomId` prop. Props are *reactive values,* which means they can change on a re-render. Notice that the Effect *re-synchronizes* (and re-connects to the server) if `roomId` changes:

<Sandpack>

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

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

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

```
return <h1>Welcome to the {roomId} room!</h1>;
}

export default function App() {
const [roomId, setRoomId] = useState('general');
return (
<>
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
</select>
</label>
<hr />
<ChatRoom roomId={roomId} />
</>
);
}
```

```js chat.js
export function createConnection(serverUrl, roomId) {
// A real implementation would actually connect to the server
return {
connect() {
console.log('■ Connecting to "' + roomId + '" room at ' + serverUrl + '...');
},
disconnect() {
console.log('■ Disconnected from "' + roomId + '" room at ' + serverUrl);
}
};
}
```

```

```css
input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }
```

</Sandpack>

React provides a linter rule to check that you've specified your Effect's dependencies correctly. If you forget to specify `roomId` in the list of dependencies in the above example, the linter will find that bug automatically.

<LearnMore path="/learn/lifecycle-of-reactive-effects">

Read **[Lifecycle of Reactive Events](/learn/lifecycle-of-reactive-effects)** to learn how an Effect's lifecycle is different from a component's.

</LearnMore>

Separating events from Effects {/*separating-events-from-effects*/}

<Wip>

This section describes an **experimental API that has not yet been released** in a stable version of React.

</Wip>

Event handlers only re-run when you perform the same interaction again. Unlike event handlers, Effects re-synchronize if any of the values they read, like props or state, are different than during last render. Sometimes, you want a mix of both behaviors: an Effect that re-runs in response to some values but not others.

All code inside Effects is *reactive.* It will run again if some reactive value it reads has changed due to a re-render. For example, this Effect will re-connect to the chat if either `roomId` or `theme` have changed:

<Sandpack>

```json package.json hidden
{
"dependencies": {
"react": "latest",
"react-dom": "latest",
"react-scripts": "latest",
"toastify-js": "1.12.0"
},
```
```

```
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
}
}
```

```js
import { useState, useEffect } from 'react';
import { createConnection, sendMessage } from './chat.js';
import { showNotification } from './notifications.js';

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

function ChatRoom({ roomId, theme }) {
useEffect(() => {
const connection = createConnection(serverUrl, roomId);
connection.on('connected', () => {
showNotification('Connected!', theme);
});
connection.connect();
return () => connection.disconnect();
}, [roomId, theme]);

return <h1>Welcome to the {roomId} room!</h1>
}

export default function App() {
const [roomId, setRoomId] = useState('general');
const [isDark, setIsDark] = useState(false);
return (
<>
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
```

```
>
<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
</select>
</label>
<label>
<input
type="checkbox"
checked={isDark}
onChange={e => setIsDark(e.target.checked)}
/>
Use dark theme
</label>
<hr />
<ChatRoom
roomId={roomId}
theme={isDark ? 'dark' : 'light'}
/>
</>
);
}
```

```js chat.js
export function createConnection(serverUrl, roomId) {
// A real implementation would actually connect to the server
let connectedCallback;
let timeout;
return {
connect() {
timeout = setTimeout(() => {
if (connectedCallback) {
connectedCallback();
}
}, 100);
```

```
},
on(event, callback) {
if (connectedCallback) {
throw Error('Cannot add the handler twice.');
}
if (event !== 'connected') {
throw Error('Only "connected" event is supported.');
}
connectedCallback = callback;
},
disconnect() {
clearTimeout(timeout);
}
};
}
```

```js notifications.js
import Toastify from 'toastify-js';
import 'toastify-js/src/toastify.css';

export function showNotification(message, theme) {
Toastify({
text: message,
duration: 2000,
gravity: 'top',
position: 'right',
style: {
background: theme === 'dark' ? 'black' : 'white',
color: theme === 'dark' ? 'white' : 'black',
},
}).showToast();
}
```

```css
label { display: block; margin-top: 10px; }
```

```

</Sandpack>

This is not ideal. You want to re-connect to the chat only if the `roomId` has changed. Switching the `theme` shouldn't re-connect to the chat! Move the code reading `theme` out of your Effect into an *Effect Event*:

<Sandpack>

```json package.json hidden
{
"dependencies": {
"react": "experimental",
"react-dom": "experimental",
"react-scripts": "latest",
"toastify-js": "1.12.0"
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
}
}
```

```js
import { useState, useEffect } from 'react';
import { experimental_useEffectEvent as useEffectEvent } from 'react';
import { createConnection, sendMessage } from './chat.js';
import { showNotification } from './notifications.js';

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

function ChatRoom({ roomId, theme }) {
const onConnected = useEffectEvent(() => {
showNotification('Connected!', theme);
});

useEffect(() => {

```
const connection = createConnection(serverUrl, roomId);
connection.on('connected', () => {
onConnected();
});
connection.connect();
return () => connection.disconnect();
}, [roomId]);

return <h1>Welcome to the {roomId} room!</h1>
}

export default function App() {
const [roomId, setRoomId] = useState('general');
const [isDark, setIsDark] = useState(false);
return (
<>
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
</select>
</label>
<label>
<input
type="checkbox"
checked={isDark}
onChange={e => setIsDark(e.target.checked)}
/>
Use dark theme
</label>
<hr />
<ChatRoom
```

```
roomId={roomId}
theme={isDark ? 'dark' : 'light'}
/>
</>
);
}
```

```js chat.js
export function createConnection(serverUrl, roomId) {
// A real implementation would actually connect to the server
let connectedCallback;
let timeout;
return {
connect() {
timeout = setTimeout(() => {
if (connectedCallback) {
connectedCallback();
}
}, 100);
},
on(event, callback) {
if (connectedCallback) {
throw Error('Cannot add the handler twice.');
}
if (event !== 'connected') {
throw Error('Only "connected" event is supported.');
}
connectedCallback = callback;
},
disconnect() {
clearTimeout(timeout);
}
};
}
```

```js notifications.js hidden
import Toastify from 'toastify-js';
import 'toastify-js/src/toastify.css';

export function showNotification(message, theme) {
Toastify({
text: message,
duration: 2000,
gravity: 'top',
position: 'right',
style: {
background: theme === 'dark' ? 'black' : 'white',
color: theme === 'dark' ? 'white' : 'black',
},
}).showToast();
}
```

```css
label { display: block; margin-top: 10px; }
```

</Sandpack>

Code inside Effect Events isn't reactive, so changing the `theme` no longer makes your Effect re-connect.

<LearnMore path="/learn/separating-events-from-effects">

Read **[Separating Events from Effects](/learn/separating-events-from-effects)** to learn how to prevent some values from re-triggering Effects.

</LearnMore>

Removing Effect dependencies {/*removing-effect-dependencies*/}

When you write an Effect, the linter will verify that you've included every reactive value (like props and state) that the Effect reads in the list of your Effect's dependencies. This ensures that your Effect remains synchronized with the latest props and state of your component. Unnecessary dependencies may cause your Effect to run too often, or even create an infinite loop. The way you remove them depends on the case.

For example, this Effect depends on the `options` object which gets re-created every time you edit the input:

<Sandpack>

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

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

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

const options = {
serverUrl: serverUrl,
roomId: roomId
};

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

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

export default function App() {
const [roomId, setRoomId] = useState('general');
return (
<>
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
```

<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
</select>
</label>
<hr />
<ChatRoom roomId={roomId} />
</>
);
}
```

```js chat.js
export function createConnection({ serverUrl, roomId }) {
// A real implementation would actually connect to the server
return {
connect() {
console.log('■ Connecting to "' + roomId + '" room at ' + serverUrl + '...');
},
disconnect() {
console.log('■ Disconnected from "' + roomId + '" room at ' + serverUrl);
}
};
}
```

```css
input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }
```

</Sandpack>

You don't want the chat to re-connect every time you start typing a message in that chat. To fix this problem, move creation of the `options` object inside the Effect so that the Effect only depends on the `roomId` string:

<Sandpack>

```js

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

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

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

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

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

export default function App() {
const [roomId, setRoomId] = useState('general');
return (
<>
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
```

</select>
</label>
<hr />
<ChatRoom roomId={roomId} />
</>
);
}
```

```js chat.js
export function createConnection({ serverUrl, roomId }) {
// A real implementation would actually connect to the server
return {
connect() {
console.log('■ Connecting to "' + roomId + '" room at ' + serverUrl + '...');
},
disconnect() {
console.log('■ Disconnected from "' + roomId + '" room at ' + serverUrl);
}
};
}
```

```css
input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }
```

</Sandpack>

Notice that you didn't start by editing the dependency list to remove the `options` dependency. That would be wrong. Instead, you changed the surrounding code so that the dependency became *unnecessary.* Think of the dependency list as a list of all the reactive values used by your Effect's code. You don't intentionally choose what to put on that list. The list describes your code. To change the dependency list, change the code.

<LearnMore path="/learn/removing-effect-dependencies">

Read **[Removing Effect Dependencies](/learn/removing-effect-dependencies)** to learn how to make your Effect re-run less often.

</LearnMore>
```

Reusing logic with custom Hooks {/*reusing-logic-with-custom-hooks*/}

React comes with built-in Hooks like `useState`, `useContext`, and `useEffect`. Sometimes, you'll wish that there was a Hook for some more specific purpose: for example, to fetch data, to keep track of whether the user is online, or to connect to a chat room. To do this, you can create your own Hooks for your application's needs.

In this example, the `usePointerPosition` custom Hook tracks the cursor position, while `useDelayedValue` custom Hook returns a value that's "lagging behind" the value you passed by a certain number of milliseconds. Move the cursor over the sandbox preview area to see a moving trail of dots following the cursor:

<Sandpack>

```js
import { usePointerPosition } from './usePointerPosition.js';
import { useDelayedValue } from './useDelayedValue.js';

export default function Canvas() {
const pos1 = usePointerPosition();
const pos2 = useDelayedValue(pos1, 100);
const pos3 = useDelayedValue(pos2, 200);
const pos4 = useDelayedValue(pos3, 100);
const pos5 = useDelayedValue(pos4, 50);
return (
<>
<Dot position={pos1} opacity={1} />
<Dot position={pos2} opacity={0.8} />
<Dot position={pos3} opacity={0.6} />
<Dot position={pos4} opacity={0.4} />
<Dot position={pos5} opacity={0.2} />
</>
);
}

function Dot({ position, opacity }) {
return (
<div style={{
position: 'absolute',
backgroundColor: 'pink',
borderRadius: '50%',
```

````
opacity,
transform: `translate(${position.x}px, ${position.y}px)`,
pointerEvents: 'none',
left: -20,
top: -20,
width: 40,
height: 40,
}} />
);
}
```

```js usePointerPosition.js
import { useState, useEffect } from 'react';

export function usePointerPosition() {
const [position, setPosition] = useState({ x: 0, y: 0 });
useEffect(() => {
function handleMove(e) {
setPosition({ x: e.clientX, y: e.clientY });
}
window.addEventListener('pointermove', handleMove);
return () => window.removeEventListener('pointermove', handleMove);
}, []);
return position;
}
```

```js useDelayedValue.js
import { useState, useEffect } from 'react';

export function useDelayedValue(value, delay) {
const [delayedValue, setDelayedValue] = useState(value);

useEffect(() => {
setTimeout(() => {
setDelayedValue(value);
}, delay);
}, [value, delay]);
````

return delayedValue;

}

```

```css

body { min-height: 300px; }

```

</Sandpack>

You can create custom Hooks, compose them together, pass data between them, and reuse them between components. As your app grows, you will write fewer Effects by hand because you'll be able to reuse custom Hooks you already wrote. There are also many excellent custom Hooks maintained by the React community.

<LearnMore path="/learn/reusing-logic-with-custom-hooks">

Read **[Reusing Logic with Custom Hooks](/learn/reusing-logic-with-custom-hooks)** to learn how to share logic between components.

</LearnMore>

What's next? {/*whats-next*/}

Head over to [Referencing Values with Refs](/learn/referencing-values-with-refs) to start reading this chapter page by page!

---

title: Passing Props to a Component

<Intro>

React components use *props* to communicate with each other. Every parent component can pass some information to its child components by giving them props. Props might remind you of HTML attributes, but you can pass any JavaScript value through them, including objects, arrays, and functions.

</Intro>

<YouWillLearn>

* How to pass props to a component

* How to read props from a component

* How to specify default values for props

* How to pass some JSX to a component

* How props change over time
```

</YouWillLearn>

Familiar props {/*familiar-props*/}

Props are the information that you pass to a JSX tag. For example, `className`, `src`, `alt`, `width`, and `height` are some of the props you can pass to an `<img>`:

<Sandpack>

```js
function Avatar() {
return (
<img
className="avatar"
src="https://i.imgur.com/1bX5QH6.jpg"
alt="Lin Lanying"
width={100}
height={100}
/>
);
}

export default function Profile() {
return (
<Avatar />
);
}
```

```css
body { min-height: 120px; }
.avatar { margin: 20px; border-radius: 50%; }
```

</Sandpack>

The props you can pass to an `<img>` tag are predefined (ReactDOM conforms to [the HTML standard](https://www.w3.org/TR/html52/semantics-embedded-content.html#the-img-element)). But you can pass any props to *your own* components, such as `<Avatar>`, to customize them. Here's how!

Passing props to a component {/*passing-props-to-a-component*/}

In this code, the `Profile` component isn't passing any props to its child component, `Avatar`:

```js
export default function Profile() {
return (
<Avatar />
);
}
```

You can give `Avatar` some props in two steps.

Step 1: Pass props to the child component {/*step-1-pass-props-to-the-child-component*/}

First, pass some props to `Avatar`. For example, let's pass two props: `person` (an object), and `size` (a number):

```js
export default function Profile() {
return (
<Avatar
person={{ name: 'Lin Lanying', imageId: '1bX5QH6' }}
size={100}
/>
);
}
```

<Note>

If double curly braces after `person=` confuse you, recall [they're merely an object](/learn/javascript-in-jsx-with-curly-braces#using-double-curlies-css-and-other-objects-in-jsx) inside the JSX curlies.

</Note>

Now you can read these props inside the `Avatar` component.

Step 2: Read props inside the child component {/*step-2-read-props-inside-the-child-component*/}

You can read these props by listing their names `person, size` separated by the commas inside `({` and `})` directly after `function Avatar`. This lets you use them inside the `Avatar` code, like you would with a variable.

```js
```

```
function Avatar({ person, size }) {
// person and size are available here
}
```

Add some logic to `Avatar` that uses the `person` and `size` props for rendering, and you're done.

Now you can configure `Avatar` to render in many different ways with different props. Try tweaking the values!

<Sandpack>

```js App.js
import { getImageUrl } from './utils.js';

function Avatar({ person, size }) {
return (
<img
className="avatar"
src={getImageUrl(person)}
alt={person.name}
width={size}
height={size}
/>
);
}

export default function Profile() {
return (
<div>
<Avatar
size={100}
person={{
name: 'Katsuko Saruhashi',
imageId: 'YfeOqp2'
}}
/>
<Avatar
size={80}
person={{
```

```
name: 'Aklilu Lemma',
imageId: 'OKS67lh'
}}
/>
<Avatar
size={50}
person={{
name: 'Lin Lanying',
imageId: '1bX5QH6'
}}
/>
</div>
);
}
```

```js utils.js
export function getImageUrl(person, size = 's') {
return (
'https://i.imgur.com/' +
person.imageId +
size +
'.jpg'
);
}
```

```css
body { min-height: 120px; }
.avatar { margin: 10px; border-radius: 50%; }
```

</Sandpack>

Props let you think about parent and child components independently. For example, you can change the `person` or the `size` props inside `Profile` without having to think about how `Avatar` uses them. Similarly, you can change how the `Avatar` uses these props, without looking at the `Profile`.

You can think of props like "knobs" that you can adjust. They serve the same role as arguments serve for functions—in fact, props _are_ the only argument to your component! React component functions

accept a single argument, a `props` object:

```js
function Avatar(props) {
let person = props.person;
let size = props.size;
// ...
}
```

Usually you don't need the whole `props` object itself, so you destructure it into individual props.

<Pitfall>

Don't miss the pair of `{` and `}` curlies inside of `(` and `)` when declaring props:

```js
function Avatar({ person, size }) {
// ...
}
```

This syntax is called ["destructuring"](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment#Unpacking_fields_from_objects_passed_as_a_function_parameter) and is equivalent to reading properties from a function parameter:

```js
function Avatar(props) {
let person = props.person;
let size = props.size;
// ...
}
```

</Pitfall>

Specifying a default value for a prop {/*specifying-a-default-value-for-a-prop*/}

If you want to give a prop a default value to fall back on when no value is specified, you can do it with the destructuring by putting `=` and the default value right after the parameter:

```js
function Avatar({ person, size = 100 }) {
// ...
```

}
```

Now, if `<Avatar person={...} />` is rendered with no `size` prop, the `size` will be set to `100`.

The default value is only used if the `size` prop is missing or if you pass `size={undefined}`. But if you pass `size={null}` or `size={0}`, the default value will **not** be used.

## Forwarding props with the JSX spread syntax {/*forwarding-props-with-the-jsx-spread-syntax*/}

Sometimes, passing props gets very repetitive:

```js
function Profile({ person, size, isSepia, thickBorder }) {
return (
<div className="card">
<Avatar
person={person}
size={size}
isSepia={isSepia}
thickBorder={thickBorder}
/>
</div>
);
}
```

There's nothing wrong with repetitive code—it can be more legible. But at times you may value conciseness. Some components forward all of their props to their children, like how this `Profile` does with `Avatar`. Because they don't use any of their props directly, it can make sense to use a more concise "spread" syntax:

```js
function Profile(props) {
return (
<div className="card">
<Avatar {...props} />
</div>
);
}
```
```

This forwards all of `Profile`'s props to the `Avatar` without listing each of their names.

Use spread syntax with restraint. If you're using it in every other component, something is wrong. Often, it indicates that you should split your components and pass children as JSX. More on that next!

Passing JSX as children {/*passing-jsx-as-children*/}

It is common to nest built-in browser tags:

```js
<div>
<img />
</div>
```

Sometimes you'll want to nest your own components the same way:

```js
<Card>
<Avatar />
</Card>
```

When you nest content inside a JSX tag, the parent component will receive that content in a prop called `children`. For example, the `Card` component below will receive a `children` prop set to `<Avatar />` and render it in a wrapper div:

<Sandpack>

```js App.js
import Avatar from './Avatar.js';

function Card({ children }) {
return (
<div className="card">
{children}
</div>
);
}

export default function Profile() {
return (
<Card>
<Avatar
```

```
size={100}
person={{
name: 'Katsuko Saruhashi',
imageId: 'YfeOqp2'
}}
/>
</Card>
);
}
```

```js Avatar.js
import { getImageUrl } from './utils.js';

export default function Avatar({ person, size }) {
return (
<img
className="avatar"
src={getImageUrl(person)}
alt={person.name}
width={size}
height={size}
/>
);
}
```

```js utils.js
export function getImageUrl(person, size = 's') {
return (
'https://i.imgur.com/' +
person.imageId +
size +
'.jpg'
);
}
```

```css
.card {
width: fit-content;
margin: 5px;
padding: 5px;
font-size: 20px;
text-align: center;
border: 1px solid #aaa;
border-radius: 20px;
background: #fff;
}
.avatar {
margin: 20px;
border-radius: 50%;
}
```

</Sandpack>

Try replacing the `<Avatar>` inside `<Card>` with some text to see how the `Card` component can wrap any nested content. It doesn't need to "know" what's being rendered inside of it. You will see this flexible pattern in many places.

You can think of a component with a `children` prop as having a "hole" that can be "filled in" by its parent components with arbitrary JSX. You will often use the `children` prop for visual wrappers: panels, grids, etc.

<Illustration src="/images/docs/illustrations/i_children-prop.png" alt='A puzzle-like Card tile with a slot for "children" pieces like text and Avatar' />

How props change over time {/*how-props-change-over-time*/}

The `Clock` component below receives two props from its parent component: `color` and `time`. (The parent component's code is omitted because it uses [state](/learn/state-a-components-memory), which we won't dive into just yet.)

Try changing the color in the select box below:

<Sandpack>

```js Clock.js active
export default function Clock({ color, time }) {
return (
<h1 style={{ color: color }}>
```

{time}
</h1>
);
}
```

```js App.js hidden
import { useState, useEffect } from 'react';
import Clock from './Clock.js';

function useTime() {
const [time, setTime] = useState(() => new Date());
useEffect(() => {
const id = setInterval(() => {
setTime(new Date());
}, 1000);
return () => clearInterval(id);
}, []);
return time;
}

export default function App() {
const time = useTime();
const [color, setColor] = useState('lightcoral');
return (

Pick a color:{' '}
<select value={color} onChange={e => setColor(e.target.value)}>
<option value="lightcoral">lightcoral</option>
<option value="midnightblue">midnightblue</option>
<option value="rebeccapurple">rebeccapurple</option>
</select>

<Clock color={color} time={time.toLocaleTimeString()} />

);

}
```

</Sandpack>

This example illustrates that **a component may receive different props over time.** Props are not always static! Here, the `time` prop changes every second, and the `color` prop changes when you select another color. Props reflect a component's data at any point in time, rather than only in the beginning.

However, props are [immutable](https://en.wikipedia.org/wiki/Immutable_object)—a term from computer science meaning "unchangeable". When a component needs to change its props (for example, in response to a user interaction or new data), it will have to "ask" its parent component to pass it _different props_—a new object! Its old props will then be cast aside, and eventually the JavaScript engine will reclaim the memory taken by them.

Don't try to "change props". When you need to respond to the user input (like changing the selected color), you will need to "set state", which you can learn about in [State: A Component's Memory.](/learn/state-a-components-memory)

<Recap>

* To pass props, add them to the JSX, just like you would with HTML attributes.
* To read props, use the `function Avatar({ person, size })` destructuring syntax.
* You can specify a default value like `size = 100`, which is used for missing and `undefined` props.
* You can forward all props with `<Avatar {...props} />` JSX spread syntax, but don't overuse it!
* Nested JSX like `<Card><Avatar /></Card>` will appear as `Card` component's `children` prop.
* Props are read-only snapshots in time: every render receives a new version of props.
* You can't change props. When you need interactivity, you'll need to set state.

</Recap>

<Challenges>

Extract a component {/*extract-a-component*/}

This `Gallery` component contains some very similar markup for two profiles. Extract a `Profile` component out of it to reduce the duplication. You'll need to choose what props to pass to it.

<Sandpack>

```js App.js
import { getImageUrl } from './utils.js';

export default function Gallery() {
return (

```
<h1>Notable Scientists</h1>
<section className="profile">
<h2>Maria Sk■odowska-Curie</h2>
<img
className="avatar"
src={getImageUrl('szV5sdG')}
alt="Maria Sk■odowska-Curie"
width={70}
height={70}
/>
<ul>
<li>
<b>Profession: </b>
physicist and chemist
</li>
<li>
<b>Awards: 4 </b>
(Nobel Prize in Physics, Nobel Prize in Chemistry, Davy Medal, Matteucci Medal)
</li>
<li>
<b>Discovered: </b>
polonium (element)
</li>
</ul>
</section>
<section className="profile">
<h2>Katsuko Saruhashi</h2>
<img
className="avatar"
src={getImageUrl('YfeOqp2')}
alt="Katsuko Saruhashi"
width={70}
height={70}
/>
<ul>
```

```
<li>
<b>Profession: </b>
geochemist
</li>
<li>
<b>Awards: 2 </b>
(Miyake Prize for geochemistry, Tanaka Prize)
</li>
<li>
<b>Discovered: </b>
a method for measuring carbon dioxide in seawater
</li>
</ul>
</section>
</div>
);
}
```

```js utils.js
export function getImageUrl(imageId, size = 's') {
return (
'https://i.imgur.com/' +
imageId +
size +
'.jpg'
);
}
```

```css
.avatar { margin: 5px; border-radius: 50%; min-height: 70px; }
.profile {
border: 1px solid #aaa;
border-radius: 6px;
margin-top: 20px;
padding: 10px;
```

}

h1, h2 { margin: 5px; }

h1 { margin-bottom: 10px; }

ul { padding: 0px 10px 0px 20px; }

li { margin: 5px; }

```

</Sandpack>

<Hint>

Start by extracting the markup for one of the scientists. Then find the pieces that don't match it in the second example, and make them configurable by props.

</Hint>

<Solution>

In this solution, the `Profile` component accepts multiple props: `imageId` (a string), `name` (a string), `profession` (a string), `awards` (an array of strings), `discovery` (a string), and `imageSize` (a number).

Note that the `imageSize` prop has a default value, which is why we don't pass it to the component.

<Sandpack>

```js App.js
import { getImageUrl } from './utils.js';

function Profile({

imageId,

name,

profession,

awards,

discovery,

imageSize = 70

}) {

return (

<section className="profile">

<h2>{name}</h2>

<img

className="avatar"

src={getImageUrl(imageId)}

```
alt={name}
width={imageSize}
height={imageSize}
/>

Profession: {profession}

Awards: {awards.length}
({awards.join(', ')})

Discovered:
{discovery}

</section>
);
}

export default function Gallery() {
return (
<div>
<h1>Notable Scientists</h1>
<Profile
imageId="szV5sdG"
name="Maria Sk■odowska-Curie"
profession="physicist and chemist"
discovery="polonium (chemical element)"
awards={[
'Nobel Prize in Physics',
'Nobel Prize in Chemistry',
'Davy Medal',
'Matteucci Medal'
]}
/>
<Profile
```

imageId='YfeOqp2'
name='Katsuko Saruhashi'
profession='geochemist'
discovery="a method for measuring carbon dioxide in seawater"
awards={[
'Miyake Prize for geochemistry',
'Tanaka Prize'
]}
/>

);
}
```

```js utils.js
export function getImageUrl(imageId, size = 's') {
return (
'https://i.imgur.com/' +
imageId +
size +
'.jpg'
);
}
```

```css
.avatar { margin: 5px; border-radius: 50%; min-height: 70px; }
.profile {
border: 1px solid #aaa;
border-radius: 6px;
margin-top: 20px;
padding: 10px;
}
h1, h2 { margin: 5px; }
h1 { margin-bottom: 10px; }
ul { padding: 0px 10px 0px 20px; }
li { margin: 5px; }

```

</Sandpack>

Note how you don't need a separate `awardCount` prop if `awards` is an array. Then you can use `awards.length` to count the number of awards. Remember that props can take any values, and that includes arrays too!

Another solution, which is more similar to the earlier examples on this page, is to group all information about a person in a single object, and pass that object as one prop:

<Sandpack>

```js App.js
import { getImageUrl } from './utils.js';

function Profile({ person, imageSize = 70 }) {
const imageSrc = getImageUrl(person)

return (
<section className="profile">
<h2>{person.name}</h2>
<img
className="avatar"
src={imageSrc}
alt={person.name}
width={imageSize}
height={imageSize}
/>
<ul>
<li>
<b>Profession:</b> {person.profession}
</li>
<li>
<b>Awards: {person.awards.length} </b>
({person.awards.join(', ')})
</li>
<li>
<b>Discovered: </b>
{person.discovery}
</li>
```
```

```
</ul>
</section>
)
}

export default function Gallery() {
return (
<div>
<h1>Notable Scientists</h1>
<Profile person={{
imageId: 'szV5sdG',
name: 'Maria Sk■odowska-Curie',
profession: 'physicist and chemist',
discovery: 'polonium (chemical element)',
awards: [
'Nobel Prize in Physics',
'Nobel Prize in Chemistry',
'Davy Medal',
'Matteucci Medal'
],
}} />
<Profile person={{
imageId: 'YfeOqp2',
name: 'Katsuko Saruhashi',
profession: 'geochemist',
discovery: 'a method for measuring carbon dioxide in seawater',
awards: [
'Miyake Prize for geochemistry',
'Tanaka Prize'
],
}} />
</div>
);
}
```

```js utils.js
```

```
export function getImageUrl(person, size = 's') {
return (
'https://i.imgur.com/' +
person.imageId +
size +
'.jpg'
);
}
```

```css
.avatar { margin: 5px; border-radius: 50%; min-height: 70px; }
.profile {
border: 1px solid #aaa;
border-radius: 6px;
margin-top: 20px;
padding: 10px;
}
h1, h2 { margin: 5px; }
h1 { margin-bottom: 10px; }
ul { padding: 0px 10px 0px 20px; }
li { margin: 5px; }
```

</Sandpack>

Although the syntax looks slightly different because you're describing properties of a JavaScript object rather than a collection of JSX attributes, these examples are mostly equivalent, and you can pick either approach.

</Solution>

Adjust the image size based on a prop {/*adjust-the-image-size-based-on-a-prop*/}

In this example, `Avatar` receives a numeric `size` prop which determines the `<img>` width and height. The `size` prop is set to `40` in this example. However, if you open the image in a new tab, you'll notice that the image itself is larger (`160` pixels). The real image size is determined by which thumbnail size you're requesting.

Change the `Avatar` component to request the closest image size based on the `size` prop. Specifically, if the `size` is less than `90`, pass `'s'` ("small") rather than `'b'` ("big") to the `getImageUrl` function. Verify that your changes work by rendering avatars with different values of the `size` prop and opening images in a new tab.

<Sandpack>

```js App.js
import { getImageUrl } from './utils.js';

function Avatar({ person, size }) {
return (
<img
className="avatar"
src={getImageUrl(person, 'b')}
alt={person.name}
width={size}
height={size}
/>
);
}

export default function Profile() {
return (
<Avatar
size={40}
person={{
name: 'Gregorio Y. Zara',
imageId: '7vQD0fP'
}}
/>
);
}
```

```js utils.js
export function getImageUrl(person, size) {
return (
'https://i.imgur.com/' +
person.imageId +
size +
'.jpg'
);
```

}
```

```css
.avatar { margin: 20px; border-radius: 50%; }
```

</Sandpack>

<Solution>

Here is how you could go about it:

<Sandpack>

```js App.js
import { getImageUrl } from './utils.js';

function Avatar({ person, size }) {
let thumbnailSize = 's';
if (size > 90) {
thumbnailSize = 'b';
}
return (
<img
className="avatar"
src={getImageUrl(person, thumbnailSize)}
alt={person.name}
width={size}
height={size}
/>
);
}

export default function Profile() {
return (
<>
<Avatar
size={40}
person={{
name: 'Gregorio Y. Zara',

```
imageId: '7vQD0fP'
}}
/>
<Avatar
size={120}
person={{
name: 'Gregorio Y. Zara',
imageId: '7vQD0fP'
}}
/>
</>
);
}
```

```js utils.js
export function getImageUrl(person, size) {
return (
'https://i.imgur.com/' +
person.imageId +
size +
'.jpg'
);
}
```

```css
.avatar { margin: 20px; border-radius: 50%; }
```

</Sandpack>

You could also show a sharper image for high DPI screens by taking [`window.devicePixelRatio`](https://developer.mozilla.org/en-US/docs/Web/API/Window/devicePixelRatio) into account:

<Sandpack>

```js App.js
import { getImageUrl } from './utils.js';
```

```
const ratio = window.devicePixelRatio;

function Avatar({ person, size }) {
let thumbnailSize = 's';
if (size * ratio > 90) {
thumbnailSize = 'b';
}
return (
<img
className="avatar"
src={getImageUrl(person, thumbnailSize)}
alt={person.name}
width={size}
height={size}
/>
);
}

export default function Profile() {
return (
<>
<Avatar
size={40}
person={{
name: 'Gregorio Y. Zara',
imageId: '7vQD0fP'
}}
/>
<Avatar
size={70}
person={{
name: 'Gregorio Y. Zara',
imageId: '7vQD0fP'
}}
/>
<Avatar
size={120}
```

person={{
name: 'Gregorio Y. Zara',
imageId: '7vQD0fP'
}}
/>
</>
);
}
```

```js utils.js
export function getImageUrl(person, size) {
return (
'https://i.imgur.com/' +
person.imageId +
size +
'.jpg'
);
}
```

```css
.avatar { margin: 20px; border-radius: 50%; }
```

</Sandpack>

Props let you encapsulate logic like this inside the `Avatar` component (and change it later if needed) so that everyone can use the `<Avatar>` component without thinking about how the images are requested and resized.

</Solution>

Passing JSX in a `children` prop {/*passing-jsx-in-a-children-prop*/}

Extract a `Card` component from the markup below, and use the `children` prop to pass different JSX to it:

<Sandpack>

```js
export default function Profile() {

return (

<h1>Photo</h1>

<img

className="avatar"

src="https://i.imgur.com/OKS67lhm.jpg"

alt="Aklilu Lemma"

width={70}

height={70}

/>

<h1>About</h1>

Aklilu Lemma was a distinguished Ethiopian scientist who discovered a natural treatment to schistosomiasis.

);

}

```

```css

.card {

width: fit-content;

margin: 20px;

padding: 20px;

border: 1px solid #aaa;

border-radius: 20px;

background: #fff;

}

.card-content {

````
text-align: center;
}
.avatar {
margin: 10px;
border-radius: 50%;
}
h1 {
margin: 5px;
padding: 0;
font-size: 24px;
}
```

</Sandpack>

<Hint>

Any JSX you put inside of a component's tag will be passed as the `children` prop to that component.

</Hint>

<Solution>

This is how you can use the `Card` component in both places:

<Sandpack>

```js
function Card({ children }) {
return (
<div className="card">
<div className="card-content">
{children}
</div>
</div>
);
}

export default function Profile() {
return (
<div>
````

<Card>

<h1>Photo</h1>

<img

className="avatar"

src="https://i.imgur.com/OKS67lhm.jpg"

alt="Aklilu Lemma"

width={100}

height={100}

/>

</Card>

<Card>

<h1>About</h1>

Aklilu Lemma was a distinguished Ethiopian scientist who discovered a natural treatment to schistosomiasis.

</Card>

);

}

```

```css

.card {

width: fit-content;

margin: 20px;

padding: 20px;

border: 1px solid #aaa;

border-radius: 20px;

background: #fff;

}

.card-content {

text-align: center;

}

.avatar {

margin: 10px;

border-radius: 50%;

}

h1 {
margin: 5px;
padding: 0;
font-size: 24px;
}
```

</Sandpack>

You can also make `title` a separate prop if you want every `Card` to always have a title:

<Sandpack>

```js
function Card({ children, title }) {
return (

<h1>{title}</h1>
{children}

);
}

export default function Profile() {
return (

<Card title="Photo">
<img
className="avatar"
src="https://i.imgur.com/OKS67lhm.jpg"
alt="Aklilu Lemma"
width={100}
height={100}
/>
</Card>
<Card title="About">

Aklilu Lemma was a distinguished Ethiopian scientist who discovered a natural treatment to schistosomiasis.

</Card>

);

}

```

```css

.card {

width: fit-content;

margin: 20px;

padding: 20px;

border: 1px solid #aaa;

border-radius: 20px;

background: #fff;

}

.card-content {

text-align: center;

}

.avatar {

margin: 10px;

border-radius: 50%;

}

h1 {

margin: 5px;

padding: 0;

font-size: 24px;

}

```

</Sandpack>

</Solution>

</Challenges>

---

title: React Developer Tools
```

<Intro>

Use React Developer Tools to inspect React [components](/learn/your-first-component), edit [props](/learn/passing-props-to-a-component) and [state](/learn/state-a-components-memory), and identify performance problems.

</Intro>

<YouWillLearn>

* How to install React Developer Tools

</YouWillLearn>

Browser extension {/*browser-extension*/}

The easiest way to debug websites built with React is to install the React Developer Tools browser extension. It is available for several popular browsers:

* [Install for **Chrome**](https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi?hl=en)
* [Install for **Firefox**](https://addons.mozilla.org/en-US/firefox/addon/react-devtools/)
* [Install for **Edge**](https://microsoftedge.microsoft.com/addons/detail/react-developer-tools/gpphkfbcpidddadnkolkpfckpihlkkil)

Now, if you visit a website **built with React,** you will see the _Components_ and _Profiler_ panels.

![React Developer Tools extension](/images/docs/react-devtools-extension.png)

Safari and other browsers {/*safari-and-other-browsers*/}

For other browsers (for example, Safari), install the [`react-devtools`](https://www.npmjs.com/package/react-devtools) npm package:

```bash
# Yarn
yarn global add react-devtools

# Npm
npm install -g react-devtools
```

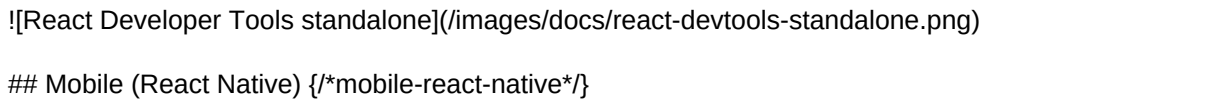
Next open the developer tools from the terminal:

```bash
react-devtools
```

Then connect your website by adding the following `<script>` tag to the beginning of your website's `<head>`:

```html {3}
<html>
<head>
<script src="http://localhost:8097"></script>
```

Reload your website in the browser now to view it in developer tools.



Mobile (React Native) {/*mobile-react-native*/}

React Developer Tools can be used to inspect apps built with [React Native](https://reactnative.dev/) as well.

The easiest way to use React Developer Tools is to install it globally:

```bash
# Yarn
yarn global add react-devtools

# Npm
npm install -g react-devtools
```

Next open the developer tools from the terminal.

```bash
react-devtools
```

It should connect to any local React Native app that's running.

> Try reloading the app if developer tools doesn't connect after a few seconds.

[Learn more about debugging React Native.](https://reactnative.dev/docs/debugging)

---

title: Scaling Up with Reducer and Context

<Intro>

Reducers let you consolidate a component's state update logic. Context lets you pass information deep down to other components. You can combine reducers and context together to manage state of a complex screen.

</Intro>

<YouWillLearn>

* How to combine a reducer with context
* How to avoid passing state and dispatch through props
* How to keep context and state logic in a separate file

</YouWillLearn>

Combining a reducer with context {/*combining-a-reducer-with-context*/}

In this example from [the introduction to reducers](/learn/extracting-state-logic-into-a-reducer), the state is managed by a reducer. The reducer function contains all of the state update logic and is declared at the bottom of this file:

<Sandpack>

```js App.js
import { useReducer } from 'react';
import AddTask from './AddTask.js';
import TaskList from './TaskList.js';

export default function TaskApp() {
const [tasks, dispatch] = useReducer(
tasksReducer,
initialTasks
);

function handleAddTask(text) {
dispatch({
type: 'added',
id: nextId++,
text: text,
});
}

function handleChangeTask(task) {
dispatch({
type: 'changed',
task: task
});
}
```

```
function handleDeleteTask(taskId) {
dispatch({
type: 'deleted',
id: taskId
});
}

return (
<>
<h1>Day off in Kyoto</h1>
<AddTask
onAddTask={handleAddTask}
/>
<TaskList
tasks={tasks}
onChangeTask={handleChangeTask}
onDeleteTask={handleDeleteTask}
/>
</>
);
}

function tasksReducer(tasks, action) {
switch (action.type) {
case 'added': {
return [...tasks, {
id: action.id,
text: action.text,
done: false
}];
}
case 'changed': {
return tasks.map(t => {
if (t.id === action.task.id) {
return action.task;
} else {
return t;
```

```
}
});
}
case 'deleted': {
return tasks.filter(t => t.id !== action.id);
}
default: {
throw Error('Unknown action: ' + action.type);
}
}
}

let nextId = 3;
const initialTasks = [
{ id: 0, text: 'Philosopher’s Path', done: true },
{ id: 1, text: 'Visit the temple', done: false },
{ id: 2, text: 'Drink matcha', done: false }
];
```

```js AddTask.js
import { useState } from 'react';

export default function AddTask({ onAddTask }) {
const [text, setText] = useState('');
return (
<>
<input
placeholder="Add task"
value={text}
onChange={e => setText(e.target.value)}
/>
<button onClick={() => {
setText('');
onAddTask(text);
}}>Add</button>
</>
```

```
)
}
```

```js TaskList.js
import { useState } from 'react';

export default function TaskList({
tasks,
onChangeTask,
onDeleteTask
}) {
return (
<ul>
{tasks.map(task => (
<li key={task.id}>
<Task
task={task}
onChange={onChangeTask}
onDelete={onDeleteTask}
/>
</li>
))}
</ul>
);
}

function Task({ task, onChange, onDelete }) {
const [isEditing, setIsEditing] = useState(false);
let taskContent;
if (isEditing) {
taskContent = (
<>
<input
value={task.text}
onChange={e => {
onChange({
```

```
...task,
text: e.target.value
});
}} />
<button onClick={() => setIsEditing(false)}>
Save
</button>
</>
);
} else {
taskContent = (
<>
{task.text}
<button onClick={() => setIsEditing(true)}>
Edit
</button>
</>
);
}
return (
<label>
<input
type="checkbox"
checked={task.done}
onChange={e => {
onChange({
...task,
done: e.target.checked
});
}}
/>
{taskContent}
<button onClick={() => onDelete(task.id)}>
Delete
</button>
```

</label>

);

}

```

```css

button { margin: 5px; }

li { list-style-type: none; }

ul, li { margin: 0; padding: 0; }

```

</Sandpack>

A reducer helps keep the event handlers short and concise. However, as your app grows, you might run into another difficulty. **Currently, the `tasks` state and the `dispatch` function are only available in the top-level `TaskApp` component.** To let other components read the list of tasks or change it, you have to explicitly [pass down](/learn/passing-props-to-a-component) the current state and the event handlers that change it as props.

For example, `TaskApp` passes a list of tasks and the event handlers to `TaskList`:

```js

<TaskList

tasks={tasks}

onChangeTask={handleChangeTask}

onDeleteTask={handleDeleteTask}

/>

```

And `TaskList` passes the event handlers to `Task`:

```js

<Task

task={task}

onChange={onChangeTask}

onDelete={onDeleteTask}

/>

```

In a small example like this, this works well, but if you have tens or hundreds of components in the middle, passing down all state and functions can be quite frustrating!
```

This is why, as an alternative to passing them through props, you might want to put both the `tasks` state and the `dispatch` function [into context.](/learn/passing-data-deeply-with-context) **This way, any component below `TaskApp` in the tree can read the tasks and dispatch actions without the repetitive "prop drilling".**

Here is how you can combine a reducer with context:

1. **Create** the context.

2. **Put** state and dispatch into context.

3. **Use** context anywhere in the tree.

Step 1: Create the context {/*step-1-create-the-context*/}

The `useReducer` Hook returns the current `tasks` and the `dispatch` function that lets you update them:

```js
const [tasks, dispatch] = useReducer(tasksReducer, initialTasks);
```

To pass them down the tree, you will
[create](/learn/passing-data-deeply-with-context#step-2-use-the-context) two separate contexts:

- `TasksContext` provides the current list of tasks.

- `TasksDispatchContext` provides the function that lets components dispatch actions.

Export them from a separate file so that you can later import them from other files:

<Sandpack>

```js App.js
import { useReducer } from 'react';

import AddTask from './AddTask.js';

import TaskList from './TaskList.js';

export default function TaskApp() {

const [tasks, dispatch] = useReducer(

tasksReducer,

initialTasks

);

function handleAddTask(text) {

dispatch({

type: 'added',

id: nextId++,
```

```
text: text,
});
}

function handleChangeTask(task) {
dispatch({
type: 'changed',
task: task
});
}

function handleDeleteTask(taskId) {
dispatch({
type: 'deleted',
id: taskId
});
}

return (
<>
<h1>Day off in Kyoto</h1>
<AddTask
onAddTask={handleAddTask}
/>
<TaskList
tasks={tasks}
onChangeTask={handleChangeTask}
onDeleteTask={handleDeleteTask}
/>
</>
);
}

function tasksReducer(tasks, action) {
switch (action.type) {
case 'added': {
return [...tasks, {
id: action.id,
```

```
text: action.text,
done: false
}];
}
case 'changed': {
return tasks.map(t => {
if (t.id === action.task.id) {
return action.task;
} else {
return t;
}
});
}
case 'deleted': {
return tasks.filter(t => t.id !== action.id);
}
default: {
throw Error('Unknown action: ' + action.type);
}
}
}

let nextId = 3;
const initialTasks = [
{ id: 0, text: 'Philosopher’s Path', done: true },
{ id: 1, text: 'Visit the temple', done: false },
{ id: 2, text: 'Drink matcha', done: false }
];
```

```js TasksContext.js active
import { createContext } from 'react';

export const TasksContext = createContext(null);
export const TasksDispatchContext = createContext(null);
```

```js AddTask.js
```

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

export default function AddTask({ onAddTask }) {
const [text, setText] = useState('');
return (
<>
<input
placeholder="Add task"
value={text}
onChange={e => setText(e.target.value)}
/>
<button onClick={() => {
setText('');
onAddTask(text);
}}>Add</button>
</>
)
}
```

```js TaskList.js
import { useState } from 'react';

export default function TaskList({
tasks,
onChangeTask,
onDeleteTask
}) {
return (
<ul>
{tasks.map(task => (
<li key={task.id}>
<Task
task={task}
onChange={onChangeTask}
onDelete={onDeleteTask}
/>
```

```
</li>
))}
</ul>
);
}

function Task({ task, onChange, onDelete }) {
const [isEditing, setIsEditing] = useState(false);
let taskContent;
if (isEditing) {
taskContent = (
<>
<input
value={task.text}
onChange={e => {
onChange({
...task,
text: e.target.value
});
}} />
<button onClick={() => setIsEditing(false)}>
Save
</button>
</>
);
} else {
taskContent = (
<>
{task.text}
<button onClick={() => setIsEditing(true)}>
Edit
</button>
</>
);
}
return (
```

<label>

<input

type="checkbox"

checked={task.done}

onChange={e => {

onChange({

...task,

done: e.target.checked

});

}}

/>

{taskContent}

<button onClick={() => onDelete(task.id)}>

Delete

</button>

</label>

);

}

```

```css

button { margin: 5px; }

li { list-style-type: none; }

ul, li { margin: 0; padding: 0; }

```

</Sandpack>

Here, you're passing `null` as the default value to both contexts. The actual values will be provided by the `TaskApp` component.

Step 2: Put state and dispatch into context {/*step-2-put-state-and-dispatch-into-context*/}

Now you can import both contexts in your `TaskApp` component. Take the `tasks` and `dispatch` returned by `useReducer()` and [provide them](/learn/passing-data-deeply-with-context#step-3-provide-the-context) to the entire tree below:

```js {4,7-8}

import { TasksContext, TasksDispatchContext } from './TasksContext.js';

export default function TaskApp() {

```
const [tasks, dispatch] = useReducer(tasksReducer, initialTasks);
// ...
return (
<TasksContext.Provider value={tasks}>
<TasksDispatchContext.Provider value={dispatch}>
...
</TasksDispatchContext.Provider>
</TasksContext.Provider>
);
}
```

For now, you pass the information both via props and in context:

<Sandpack>

```js App.js
import { useReducer } from 'react';
import AddTask from './AddTask.js';
import TaskList from './TaskList.js';
import { TasksContext, TasksDispatchContext } from './TasksContext.js';

export default function TaskApp() {
const [tasks, dispatch] = useReducer(
tasksReducer,
initialTasks
);

function handleAddTask(text) {
dispatch({
type: 'added',
id: nextId++,
text: text,
});
}

function handleChangeTask(task) {
dispatch({
type: 'changed',
```

```
task: task
});
}

function handleDeleteTask(taskId) {
dispatch({
type: 'deleted',
id: taskId
});
}

return (
<TasksContext.Provider value={tasks}>
<TasksDispatchContext.Provider value={dispatch}>
<h1>Day off in Kyoto</h1>
<AddTask
onAddTask={handleAddTask}
/>
<TaskList
tasks={tasks}
onChangeTask={handleChangeTask}
onDeleteTask={handleDeleteTask}
/>
</TasksDispatchContext.Provider>
</TasksContext.Provider>
);
}

function tasksReducer(tasks, action) {
switch (action.type) {
case 'added': {
return [...tasks, {
id: action.id,
text: action.text,
done: false
}];
}
```

```
case 'changed': {
return tasks.map(t => {
if (t.id === action.task.id) {
return action.task;
} else {
return t;
}
});
}
case 'deleted': {
return tasks.filter(t => t.id !== action.id);
}
default: {
throw Error('Unknown action: ' + action.type);
}
}
}

let nextId = 3;
const initialTasks = [
{ id: 0, text: 'Philosopher’s Path', done: true },
{ id: 1, text: 'Visit the temple', done: false },
{ id: 2, text: 'Drink matcha', done: false }
];
```

```js TasksContext.js
import { createContext } from 'react';

export const TasksContext = createContext(null);
export const TasksDispatchContext = createContext(null);
```

```js AddTask.js
import { useState } from 'react';

export default function AddTask({ onAddTask }) {
const [text, setText] = useState('');
return (
```

```
<>
<input
placeholder="Add task"
value={text}
onChange={e => setText(e.target.value)}
/>
<button onClick={() => {
setText('');
onAddTask(text);
}}>Add</button>
</>
)
}
```

```js TaskList.js
import { useState } from 'react';

export default function TaskList({
tasks,
onChangeTask,
onDeleteTask
}) {
return (
<ul>
{tasks.map(task => (
<li key={task.id}>
<Task
task={task}
onChange={onChangeTask}
onDelete={onDeleteTask}
/>
</li>
))}
</ul>
);
}
```

```
function Task({ task, onChange, onDelete }) {
const [isEditing, setIsEditing] = useState(false);
let taskContent;
if (isEditing) {
taskContent = (
<>
<input
value={task.text}
onChange={e => {
onChange({
...task,
text: e.target.value
});
}} />
<button onClick={() => setIsEditing(false)}>
Save
</button>
</>
);
} else {
taskContent = (
<>
{task.text}
<button onClick={() => setIsEditing(true)}>
Edit
</button>
</>
);
}
return (
<label>
<input
type="checkbox"
checked={task.done}
onChange={e => {
```

```
onChange({
...task,
done: e.target.checked
});
}}
/>
{taskContent}
<button onClick={() => onDelete(task.id)}>
Delete
</button>
</label>
);
}
```

```css
button { margin: 5px; }
li { list-style-type: none; }
ul, li { margin: 0; padding: 0; }
```

</Sandpack>

In the next step, you will remove prop passing.

Step 3: Use context anywhere in the tree {/*step-3-use-context-anywhere-in-the-tree*/}

Now you don't need to pass the list of tasks or the event handlers down the tree:

```js {4-5}
<TasksContext.Provider value={tasks}>
<TasksDispatchContext.Provider value={dispatch}>
<h1>Day off in Kyoto</h1>
<AddTask />
<TaskList />
</TasksDispatchContext.Provider>
</TasksContext.Provider>
```

Instead, any component that needs the task list can read it from the `TaskContext`:

```js {2}
export default function TaskList() {
const tasks = useContext(TasksContext);
// ...
```

To update the task list, any component can read the `dispatch` function from context and call it:

```js {3,9-13}
export default function AddTask() {
const [text, setText] = useState('');
const dispatch = useContext(TasksDispatchContext);
// ...
return (
// ...
<button onClick={() => {
setText('');
dispatch({
type: 'added',
id: nextId++,
text: text,
});
}}>Add</button>
// ...
```

The `TaskApp` component does not pass any event handlers down, and the `TaskList` does not pass any event handlers to the `Task` component either. Each component reads the context that it needs:

<Sandpack>

```js App.js
import { useReducer } from 'react';
import AddTask from './AddTask.js';
import TaskList from './TaskList.js';
import { TasksContext, TasksDispatchContext } from './TasksContext.js';

export default function TaskApp() {
const [tasks, dispatch] = useReducer(
tasksReducer,
```

```
initialTasks
);

return (
<TasksContext.Provider value={tasks}>
<TasksDispatchContext.Provider value={dispatch}>
<h1>Day off in Kyoto</h1>
<AddTask />
<TaskList />
</TasksDispatchContext.Provider>
</TasksContext.Provider>
);
}

function tasksReducer(tasks, action) {
switch (action.type) {
case 'added': {
return [...tasks, {
id: action.id,
text: action.text,
done: false
}];
}
case 'changed': {
return tasks.map(t => {
if (t.id === action.task.id) {
return action.task;
} else {
return t;
}
});
}
case 'deleted': {
return tasks.filter(t => t.id !== action.id);
}
default: {
throw Error('Unknown action: ' + action.type);
```

```
}
}
}

const initialTasks = [
{ id: 0, text: 'Philosopher’s Path', done: true },
{ id: 1, text: 'Visit the temple', done: false },
{ id: 2, text: 'Drink matcha', done: false }
];
```

```js TasksContext.js
import { createContext } from 'react';

export const TasksContext = createContext(null);
export const TasksDispatchContext = createContext(null);
```

```js AddTask.js
import { useState, useContext } from 'react';
import { TasksDispatchContext } from './TasksContext.js';

export default function AddTask() {
const [text, setText] = useState('');
const dispatch = useContext(TasksDispatchContext);
return (
<>
<input
placeholder="Add task"
value={text}
onChange={e => setText(e.target.value)}
/>
<button onClick={() => {
setText('');
dispatch({
type: 'added',
id: nextId++,
text: text,
});
```

```
}}>Add</button>
</>
);
}

let nextId = 3;
```

```js TaskList.js active
import { useState, useContext } from 'react';
import { TasksContext, TasksDispatchContext } from './TasksContext.js';

export default function TaskList() {
const tasks = useContext(TasksContext);
return (
<ul>
{tasks.map(task => (
<li key={task.id}>
<Task task={task} />
</li>
))}
</ul>
);
}

function Task({ task }) {
const [isEditing, setIsEditing] = useState(false);
const dispatch = useContext(TasksDispatchContext);
let taskContent;
if (isEditing) {
taskContent = (
<>
<input
value={task.text}
onChange={e => {
dispatch({
type: 'changed',
task: {
```

```
...task,
text: e.target.value
}
});
}} />
<button onClick={() => setIsEditing(false)}>
Save
</button>
</>
);
} else {
taskContent = (
<>
{task.text}
<button onClick={() => setIsEditing(true)}>
Edit
</button>
</>
);
}
return (
<label>
<input
type="checkbox"
checked={task.done}
onChange={e => {
dispatch({
type: 'changed',
task: {
...task,
done: e.target.checked
}
});
}}
/>
```

{taskContent}

<button onClick={() => {

dispatch({

type: 'deleted',

id: task.id

});

}}>

Delete

</button>

</label>

);

}

```

```css

button { margin: 5px; }

li { list-style-type: none; }

ul, li { margin: 0; padding: 0; }

```

</Sandpack>

The state still "lives" in the top-level `TaskApp` component, managed with `useReducer`. But its `tasks` and `dispatch` are now available to every component below in the tree by importing and using these contexts.

Moving all wiring into a single file {/*moving-all-wiring-into-a-single-file*/}

You don't have to do this, but you could further declutter the components by moving both reducer and context into a single file. Currently, `TasksContext.js` contains only two context declarations:

```js

import { createContext } from 'react';

export const TasksContext = createContext(null);

export const TasksDispatchContext = createContext(null);

```

This file is about to get crowded! You'll move the reducer into that same file. Then you'll declare a new `TasksProvider` component in the same file. This component will tie all the pieces together:

1. It will manage the state with a reducer.

2. It will provide both contexts to components below.
```

3. It will [take `children` as a prop](/learn/passing-props-to-a-component#passing-jsx-as-children) so you can pass JSX to it.

```js
export function TasksProvider({ children }) {
const [tasks, dispatch] = useReducer(tasksReducer, initialTasks);

return (
<TasksContext.Provider value={tasks}>
<TasksDispatchContext.Provider value={dispatch}>
{children}
</TasksDispatchContext.Provider>
</TasksContext.Provider>
);
}
```

This removes all the complexity and wiring from your `TaskApp` component:

<Sandpack>

```js App.js
import AddTask from './AddTask.js';
import TaskList from './TaskList.js';
import { TasksProvider } from './TasksContext.js';

export default function TaskApp() {
return (
<TasksProvider>
<h1>Day off in Kyoto</h1>
<AddTask />
<TaskList />
</TasksProvider>
);
}
```

```js TasksContext.js
import { createContext, useReducer } from 'react';

export const TasksContext = createContext(null);
```

```
export const TasksDispatchContext = createContext(null);

export function TasksProvider({ children }) {
const [tasks, dispatch] = useReducer(
tasksReducer,
initialTasks
);

return (
<TasksContext.Provider value={tasks}>
<TasksDispatchContext.Provider value={dispatch}>
{children}
</TasksDispatchContext.Provider>
</TasksContext.Provider>
);
}

function tasksReducer(tasks, action) {
switch (action.type) {
case 'added': {
return [...tasks, {
id: action.id,
text: action.text,
done: false
}];
}
case 'changed': {
return tasks.map(t => {
if (t.id === action.task.id) {
return action.task;
} else {
return t;
}
});
}
case 'deleted': {
return tasks.filter(t => t.id !== action.id);
```

```
}
default: {
throw Error('Unknown action: ' + action.type);
}
}
}

const initialTasks = [
{ id: 0, text: 'Philosopher’s Path', done: true },
{ id: 1, text: 'Visit the temple', done: false },
{ id: 2, text: 'Drink matcha', done: false }
];
```

```js AddTask.js
import { useState, useContext } from 'react';
import { TasksDispatchContext } from './TasksContext.js';

export default function AddTask() {
const [text, setText] = useState('');
const dispatch = useContext(TasksDispatchContext);
return (
<>
<input
placeholder="Add task"
value={text}
onChange={e => setText(e.target.value)}
/>
<button onClick={() => {
setText('');
dispatch({
type: 'added',
id: nextId++,
text: text,
});
}}>Add</button>
</>
```

```
);
}

let nextId = 3;
```

```js TaskList.js
import { useState, useContext } from 'react';
import { TasksContext, TasksDispatchContext } from './TasksContext.js';

export default function TaskList() {
const tasks = useContext(TasksContext);
return (
<ul>
{tasks.map(task => (
<li key={task.id}>
<Task task={task} />
</li>
))}
</ul>
);
}

function Task({ task }) {
const [isEditing, setIsEditing] = useState(false);
const dispatch = useContext(TasksDispatchContext);
let taskContent;
if (isEditing) {
taskContent = (
<>
<input
value={task.text}
onChange={e => {
dispatch({
type: 'changed',
task: {
...task,
text: e.target.value
```

```
}
});
}} />
<button onClick={() => setIsEditing(false)}>
Save
</button>
</>
);
} else {
taskContent = (
<>
{task.text}
<button onClick={() => setIsEditing(true)}>
Edit
</button>
</>
);
}
return (
<label>
<input
type="checkbox"
checked={task.done}
onChange={e => {
dispatch({
type: 'changed',
task: {
...task,
done: e.target.checked
}
});
}}
/>
{taskContent}
<button onClick={() => {
```

```
dispatch({
type: 'deleted',
id: task.id
});
}}>
Delete
</button>
</label>
);
}
```

```css
button { margin: 5px; }
li { list-style-type: none; }
ul, li { margin: 0; padding: 0; }
```

</Sandpack>

You can also export functions that _use_ the context from `TasksContext.js`:

```js
export function useTasks() {
return useContext(TasksContext);
}

export function useTasksDispatch() {
return useContext(TasksDispatchContext);
}
```

When a component needs to read context, it can do it through these functions:

```js
const tasks = useTasks();
const dispatch = useTasksDispatch();
```

This doesn't change the behavior in any way, but it lets you later split these contexts further or add some logic to these functions. **Now all of the context and reducer wiring is in `TasksContext.js`. This

keeps the components clean and uncluttered, focused on what they display rather than where they get the data:**

<Sandpack>

```js App.js
import AddTask from './AddTask.js';
import TaskList from './TaskList.js';
import { TasksProvider } from './TasksContext.js';

export default function TaskApp() {
return (
<TasksProvider>
<h1>Day off in Kyoto</h1>
<AddTask />
<TaskList />
</TasksProvider>
);
}
```

```js TasksContext.js
import { createContext, useContext, useReducer } from 'react';

const TasksContext = createContext(null);

const TasksDispatchContext = createContext(null);

export function TasksProvider({ children }) {
const [tasks, dispatch] = useReducer(
tasksReducer,
initialTasks
);

return (
<TasksContext.Provider value={tasks}>
<TasksDispatchContext.Provider value={dispatch}>
{children}
</TasksDispatchContext.Provider>
</TasksContext.Provider>
);
```

```
}

export function useTasks() {
return useContext(TasksContext);
}

export function useTasksDispatch() {
return useContext(TasksDispatchContext);
}

function tasksReducer(tasks, action) {
switch (action.type) {
case 'added': {
return [...tasks, {
id: action.id,
text: action.text,
done: false
}];
}
case 'changed': {
return tasks.map(t => {
if (t.id === action.task.id) {
return action.task;
} else {
return t;
}
});
}
case 'deleted': {
return tasks.filter(t => t.id !== action.id);
}
default: {
throw Error('Unknown action: ' + action.type);
}
}
}

const initialTasks = [
```

```
{ id: 0, text: 'Philosopher’s Path', done: true },
{ id: 1, text: 'Visit the temple', done: false },
{ id: 2, text: 'Drink matcha', done: false }
];
```

```js AddTask.js
import { useState } from 'react';
import { useTasksDispatch } from './TasksContext.js';

export default function AddTask() {
const [text, setText] = useState('');
const dispatch = useTasksDispatch();
return (
<>
<input
placeholder="Add task"
value={text}
onChange={e => setText(e.target.value)}
/>
<button onClick={() => {
setText('');
dispatch({
type: 'added',
id: nextId++,
text: text,
});
}}>Add</button>
</>
);
}

let nextId = 3;
```

```js TaskList.js active
import { useState } from 'react';
import { useTasks, useTasksDispatch } from './TasksContext.js';
```

```
export default function TaskList() {
const tasks = useTasks();
return (
<ul>
{tasks.map(task => (
<li key={task.id}>
<Task task={task} />
</li>
))}
</ul>
);
}

function Task({ task }) {
const [isEditing, setIsEditing] = useState(false);
const dispatch = useTasksDispatch();
let taskContent;
if (isEditing) {
taskContent = (
<>
<input
value={task.text}
onChange={e => {
dispatch({
type: 'changed',
task: {
...task,
text: e.target.value
}
});
}} />
<button onClick={() => setIsEditing(false)}>
Save
</button>
</>
);
```

```
} else {
taskContent = (
<>
{task.text}
<button onClick={() => setIsEditing(true)}>
Edit
</button>
</>
);
}
return (
<label>
<input
type="checkbox"
checked={task.done}
onChange={e => {
dispatch({
type: 'changed',
task: {
...task,
done: e.target.checked
}
});
}}
/>
{taskContent}
<button onClick={() => {
dispatch({
type: 'deleted',
id: task.id
});
}}>
Delete
</button>
</label>
```

);

}

```

```css

button { margin: 5px; }

li { list-style-type: none; }

ul, li { margin: 0; padding: 0; }

```

</Sandpack>

You can think of `TasksProvider` as a part of the screen that knows how to deal with tasks, `useTasks` as a way to read them, and `useTasksDispatch` as a way to update them from any component below in the tree.

<Note>

Functions like `useTasks` and `useTasksDispatch` are called *[Custom Hooks.](/learn/reusing-logic-with-custom-hooks)* Your function is considered a custom Hook if its name starts with `use`. This lets you use other Hooks, like `useContext`, inside it.

</Note>

As your app grows, you may have many context-reducer pairs like this. This is a powerful way to scale your app and [lift state up](/learn/sharing-state-between-components) without too much work whenever you want to access the data deep in the tree.

<Recap>

- You can combine reducer with context to let any component read and update state above it.

- To provide state and the dispatch function to components below:

1. Create two contexts (for state and for dispatch functions).

2. Provide both contexts from the component that uses the reducer.

3. Use either context from components that need to read them.

- You can further declutter the components by moving all wiring into one file.

- You can export a component like `TasksProvider` that provides context.

- You can also export custom Hooks like `useTasks` and `useTasksDispatch` to read it.

- You can have many context-reducer pairs like this in your app.

</Recap>

---

title: Quick Start

```

<Intro>

Welcome to the React documentation! This page will give you an introduction to the 80% of React concepts that you will use on a daily basis.

</Intro>

<YouWillLearn>

- How to create and nest components
- How to add markup and styles
- How to display data
- How to render conditions and lists
- How to respond to events and update the screen
- How to share data between components

</YouWillLearn>

Creating and nesting components {/*components*/}

React apps are made out of *components*. A component is a piece of the UI (user interface) that has its own logic and appearance. A component can be as small as a button, or as large as an entire page.

React components are JavaScript functions that return markup:

```js
function MyButton() {
return (
<button>I'm a button</button>
);
}
```

Now that you've declared `MyButton`, you can nest it into another component:

```js {5}
export default function MyApp() {
return (
<div>
<h1>Welcome to my app</h1>
<MyButton />
</div>
);
```

}
```

Notice that `<MyButton />` starts with a capital letter. That's how you know it's a React component. React component names must always start with a capital letter, while HTML tags must be lowercase.

Have a look at the result:

<Sandpack>

```js
function MyButton() {
return (
<button>
I'm a button
</button>
);
}

export default function MyApp() {
return (
<div>
<h1>Welcome to my app</h1>
<MyButton />
</div>
);
}
```

</Sandpack>

The `export default` keywords specify the main component in the file. If you're not familiar with some piece of JavaScript syntax, [MDN](https://developer.mozilla.org/en-US/docs/web/javascript/reference/statements/export) and [javascript.info](https://javascript.info/import-export) have great references.

Writing markup with JSX {/*writing-markup-with-jsx*/}

The markup syntax you've seen above is called *JSX*. It is optional, but most React projects use JSX for its convenience. All of the [tools we recommend for local development](/learn/installation) support JSX out of the box.

JSX is stricter than HTML. You have to close tags like `<br />`. Your component also can't return multiple JSX tags. You have to wrap them into a shared parent, like a `<div>...</div>` or an empty `<>...</>` wrapper:
```

```js {3,6}
function AboutPage() {
return (
<>
<h1>About</h1>
<p>Hello there.<br />How do you do?</p>
</>
);
}
```

If you have a lot of HTML to port to JSX, you can use an [online converter.](https://transform.tools/html-to-jsx)

Adding styles {/*adding-styles*/}

In React, you specify a CSS class with `className`. It works the same way as the HTML [`class`](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/class) attribute:

```js
<img className="avatar" />
```

Then you write the CSS rules for it in a separate CSS file:

```css
/* In your CSS */
.avatar {
border-radius: 50%;
}
```

React does not prescribe how you add CSS files. In the simplest case, you'll add a [`<link>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/link) tag to your HTML. If you use a build tool or a framework, consult its documentation to learn how to add a CSS file to your project.

Displaying data {/*displaying-data*/}

JSX lets you put markup into JavaScript. Curly braces let you "escape back" into JavaScript so that you can embed some variable from your code and display it to the user. For example, this will display `user.name`:

```js {3}
```

return (

<h1>

{user.name}

</h1>

);

```

You can also "escape into JavaScript" from JSX attributes, but you have to use curly braces *instead of* quotes. For example, `className="avatar"` passes the `"avatar"` string as the CSS class, but `src={user.imageUrl}` reads the JavaScript `user.imageUrl` variable value, and then passes that value as the `src` attribute:

```js {3,4}

return (

<img

className="avatar"

src={user.imageUrl}

/>

);

```

You can put more complex expressions inside the JSX curly braces too, for example, [string concatenation](https://javascript.info/operators#string-concatenation-with-binary):

<Sandpack>

```js

const user = {

name: 'Hedy Lamarr',

imageUrl: 'https://i.imgur.com/yXOvdOSs.jpg',

imageSize: 90,

};

export default function Profile() {

return (

<>

<h1>{user.name}</h1>

<img

className="avatar"

src={user.imageUrl}

alt={'Photo of ' + user.name}

````
style={{
width: user.imageSize,
height: user.imageSize
}}
/>
</>
);
}
```

```css
.avatar {
border-radius: 50%;
}

.large {
border: 4px solid gold;
}
```

</Sandpack>

In the above example, `style={{}}` is not a special syntax, but a regular `{}` object inside the `style={ }` JSX curly braces. You can use the `style` attribute when your styles depend on JavaScript variables.

Conditional rendering {/*conditional-rendering*/}

In React, there is no special syntax for writing conditions. Instead, you'll use the same techniques as you use when writing regular JavaScript code. For example, you can use an [`if`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/if...else) statement to conditionally include JSX:

```js
let content;
if (isLoggedIn) {
content = <AdminPanel />;
} else {
content = <LoginForm />;
}
return (
<div>
````

{content}

);

```

If you prefer more compact code, you can use the [conditional `?` operator.](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_Operator) Unlike `if`, it works inside JSX:

```js

<div>

{isLoggedIn ? (

<AdminPanel />

) : (

<LoginForm />

)}

</div>

```

When you don't need the `else` branch, you can also use a shorter [logical `&&` syntax](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_AND#short-circuit_evaluation):

```js

<div>

{isLoggedIn && <AdminPanel />}

</div>

```

All of these approaches also work for conditionally specifying attributes. If you're unfamiliar with some of this JavaScript syntax, you can start by always using `if...else`.

Rendering lists {/*rendering-lists*/}

You will rely on JavaScript features like [`for` loop](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for) and the [array `map()` function](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map) to render lists of components.

For example, let's say you have an array of products:

```js

const products = [

{ title: 'Cabbage', id: 1 },

{ title: 'Garlic', id: 2 },

{ title: 'Apple', id: 3 },

];

```

Inside your component, use the `map()` function to transform an array of products into an array of `<li>` items:

```js

const listItems = products.map(product =>

<li key={product.id}>

{product.title}

</li>

);

return (

<ul>{listItems}</ul>

);

```

Notice how `<li>` has a `key` attribute. For each item in a list, you should pass a string or a number that uniquely identifies that item among its siblings. Usually, a key should be coming from your data, such as a database ID. React uses your keys to know what happened if you later insert, delete, or reorder the items.

<Sandpack>

```js

const products = [

{ title: 'Cabbage', isFruit: false, id: 1 },

{ title: 'Garlic', isFruit: false, id: 2 },

{ title: 'Apple', isFruit: true, id: 3 },

];

export default function ShoppingList() {

const listItems = products.map(product =>

<li

key={product.id}

style={{

color: product.isFruit ? 'magenta' : 'darkgreen'

}}

>

{product.title}

</li>

);

return (

<ul>{listItems}</ul>

);

}

```

</Sandpack>

Responding to events {/*responding-to-events*/}

You can respond to events by declaring *event handler* functions inside your components:

```js {2-4,7}

function MyButton() {

function handleClick() {

alert('You clicked me!');

}

return (

<button onClick={handleClick}>

Click me

</button>

);

}

```

Notice how `onClick={handleClick}` has no parentheses at the end! Do not _call_ the event handler function: you only need to *pass it down*. React will call your event handler when the user clicks the button.

Updating the screen {/*updating-the-screen*/}

Often, you'll want your component to "remember" some information and display it. For example, maybe you want to count the number of times a button is clicked. To do this, add *state* to your component.

First, import [`useState`](/reference/react/useState) from React:

```js

import { useState } from 'react';

```

Now you can declare a *state variable* inside your component:

```js
function MyButton() {
const [count, setCount] = useState(0);
// ...
```

You'll get two things from `useState`: the current state (`count`), and the function that lets you update it (`setCount`). You can give them any names, but the convention is to write `[something, setSomething]`.

The first time the button is displayed, `count` will be `0` because you passed `0` to `useState()`. When you want to change state, call `setCount()` and pass the new value to it. Clicking this button will increment the counter:

```js {5}
function MyButton() {
const [count, setCount] = useState(0);

function handleClick() {
setCount(count + 1);
}

return (
<button onClick={handleClick}>
Clicked {count} times
</button>
);
}
```

React will call your component function again. This time, `count` will be `1`. Then it will be `2`. And so on.

If you render the same component multiple times, each will get its own state. Click each button separately:

<Sandpack>

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

export default function MyApp() {
```
```

```
return (
<div>
<h1>Counters that update separately</h1>
<MyButton />
<MyButton />
</div>
);
}

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

function handleClick() {
setCount(count + 1);
}

return (
<button onClick={handleClick}>
Clicked {count} times
</button>
);
}
```

```css
button {
display: block;
margin-bottom: 5px;
}
```

</Sandpack>

Notice how each button "remembers" its own `count` state and doesn't affect other buttons.

Using Hooks {/*using-hooks*/}

Functions starting with `use` are called *Hooks*. `useState` is a built-in Hook provided by React. You can find other built-in Hooks in the [API reference.](/reference/react) You can also write your own Hooks by combining the existing ones.

Hooks are more restrictive than other functions. You can only call Hooks *at the top* of your components (or other Hooks). If you want to use `useState` in a condition or a loop, extract a new component and put it there.

Sharing data between components {/*sharing-data-between-components*/}

In the previous example, each `MyButton` had its own independent `count`, and when each button was clicked, only the `count` for the button clicked changed:

<DiagramGroup>

<Diagram name="sharing_data_child" height={367} width={407} alt="Diagram showing a tree of three components, one parent labeled MyApp and two children labeled MyButton. Both MyButton components contain a count with value zero.">

Initially, each `MyButton`'s `count` state is `0`

</Diagram>

<Diagram name="sharing_data_child_clicked" height={367} width={407} alt="The same diagram as the previous, with the count of the first child MyButton component highlighted indicating a click with the count value incremented to one. The second MyButton component still contains value zero." >

The first `MyButton` updates its `count` to `1`

</Diagram>

</DiagramGroup>

However, often you'll need components to *share data and always update together*.

To make both `MyButton` components display the same `count` and update together, you need to move the state from the individual buttons "upwards" to the closest component containing all of them.

In this example, it is `MyApp`:

<DiagramGroup>

<Diagram name="sharing_data_parent" height={385} width={410} alt="Diagram showing a tree of three components, one parent labeled MyApp and two children labeled MyButton. MyApp contains a count value of zero which is passed down to both of the MyButton components, which also show value zero." >

Initially, `MyApp`'s `count` state is `0` and is passed down to both children

</Diagram>

<Diagram name="sharing_data_parent_clicked" height={385} width={410} alt="The same diagram as the previous, with the count of the parent MyApp component highlighted indicating a click with the value incremented to one. The flow to both of the children MyButton components is also highlighted, and the count value in each child is set to one indicating the value was passed down." >

On click, `MyApp` updates its `count` state to `1` and passes it down to both children

</Diagram>

</DiagramGroup>

Now when you click either button, the `count` in `MyApp` will change, which will change both of the counts in `MyButton`. Here's how you can express this in code.

First, *move the state up* from `MyButton` into `MyApp`:

```js {2-6,18}
export default function MyApp() {
const [count, setCount] = useState(0);

function handleClick() {
setCount(count + 1);
}

return (
<div>
<h1>Counters that update separately</h1>
<MyButton />
<MyButton />
</div>
);
}

function MyButton() {
// ... we're moving code from here ...
}
```

Then, *pass the state down* from `MyApp` to each `MyButton`, together with the shared click handler. You can pass information to `MyButton` using the JSX curly braces, just like you previously did with built-in tags like `<img>`:

```js {11-12}
export default function MyApp() {
const [count, setCount] = useState(0);

function handleClick() {
setCount(count + 1);
}
```

return (

<h1>Counters that update together</h1>

<MyButton count={count} onClick={handleClick} />

<MyButton count={count} onClick={handleClick} />

);

}

```

The information you pass down like this is called _props_. Now the `MyApp` component contains the `count` state and the `handleClick` event handler, and *passes both of them down as props* to each of the buttons.

Finally, change `MyButton` to *read* the props you have passed from its parent component:

```js {1,3}

function MyButton({ count, onClick }) {

return (

<button onClick={onClick}>

Clicked {count} times

</button>

);

}

```

When you click the button, the `onClick` handler fires. Each button's `onClick` prop was set to the `handleClick` function inside `MyApp`, so the code inside of it runs. That code calls `setCount(count + 1)`, incrementing the `count` state variable. The new `count` value is passed as a prop to each button, so they all show the new value. This is called "lifting state up". By moving state up, you've shared it between components.

<Sandpack>

```js

import { useState } from 'react';

export default function MyApp() {

const [count, setCount] = useState(0);

function handleClick() {

setCount(count + 1);

}

```
return (
<div>
<h1>Counters that update together</h1>
<MyButton count={count} onClick={handleClick} />
<MyButton count={count} onClick={handleClick} />
</div>
);
}

function MyButton({ count, onClick }) {
return (
<button onClick={onClick}>
Clicked {count} times
</button>
);
}
```

```css
button {
display: block;
margin-bottom: 5px;
}
```

</Sandpack>

Next Steps {/*next-steps*/}

By now, you know the basics of how to write React code!

Check out the [Tutorial](/learn/tutorial-tic-tac-toe) to put them into practice and build your first mini-app with React.

---

title: 'Lifecycle of Reactive Effects'

<Intro>

Effects have a different lifecycle from components. Components may mount, update, or unmount. An Effect can only do two things: to start synchronizing something, and later to stop synchronizing it. This cycle can happen multiple times if your Effect depends on props and state that change over time. React

provides a linter rule to check that you've specified your Effect's dependencies correctly. This keeps your Effect synchronized to the latest props and state.

</Intro>

<YouWillLearn>

- How an Effect's lifecycle is different from a component's lifecycle
- How to think about each individual Effect in isolation
- When your Effect needs to re-synchronize, and why
- How your Effect's dependencies are determined
- What it means for a value to be reactive
- What an empty dependency array means
- How React verifies your dependencies are correct with a linter
- What to do when you disagree with the linter

</YouWillLearn>

The lifecycle of an Effect {/*the-lifecycle-of-an-effect*/}

Every React component goes through the same lifecycle:

- A component _mounts_ when it's added to the screen.
- A component _updates_ when it receives new props or state, usually in response to an interaction.
- A component _unmounts_ when it's removed from the screen.

It's a good way to think about components, but _not_ about Effects. Instead, try to think about each Effect independently from your component's lifecycle. An Effect describes how to [synchronize an external system](/learn/synchronizing-with-effects) to the current props and state. As your code changes, synchronization will need to happen more or less often.

To illustrate this point, consider this Effect connecting your component to a chat server:

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

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

```
// ...
}
```

Your Effect's body specifies how to **start synchronizing:**

```js {2-3}
// ...
const connection = createConnection(serverUrl, roomId);
connection.connect();
return () => {
connection.disconnect();
};
// ...
```

The cleanup function returned by your Effect specifies how to **stop synchronizing:**

```js {5}
// ...
const connection = createConnection(serverUrl, roomId);
connection.connect();
return () => {
connection.disconnect();
};
// ...
```

Intuitively, you might think that React would **start synchronizing** when your component mounts and **stop synchronizing** when your component unmounts. However, this is not the end of the story! Sometimes, it may also be necessary to **start and stop synchronizing multiple times** while the component remains mounted.

Let's look at _why_ this is necessary, _when_ it happens, and _how_ you can control this behavior.

<Note>

Some Effects don't return a cleanup function at all. [More often than not,](/learn/synchronizing-with-effects#how-to-handle-the-effect-firing-twice-in-development) you'll want to return one--but if you don't, React will behave as if you returned an empty cleanup function.

</Note>

Why synchronization may need to happen more than once
{/*why-synchronization-may-need-to-happen-more-than-once*/}

Imagine this `ChatRoom` component receives a `roomId` prop that the user picks in a dropdown. Let's say that initially the user picks the `"general"` room as the `roomId`. Your app displays the `"general"` chat room:

```js {3}
const serverUrl = 'https://localhost:1234';

function ChatRoom({ roomId /* "general" */ }) {
// ...
return <h1>Welcome to the {roomId} room!</h1>;
}
```

After the UI is displayed, React will run your Effect to **start synchronizing.** It connects to the `"general"` room:

```js {3,4}
function ChatRoom({ roomId /* "general" */ }) {
useEffect(() => {
const connection = createConnection(serverUrl, roomId); // Connects to the "general" room
connection.connect();
return () => {
connection.disconnect(); // Disconnects from the "general" room
};
}, [roomId]);
// ...
```

So far, so good.

Later, the user picks a different room in the dropdown (for example, `"travel"`). First, React will update the UI:

```js {1}
function ChatRoom({ roomId /* "travel" */ }) {
// ...
return <h1>Welcome to the {roomId} room!</h1>;
}
```

Think about what should happen next. The user sees that `"travel"` is the selected chat room in the UI. However, the Effect that ran the last time is still connected to the `"general"` room. **The `roomId` prop has changed, so what your Effect did back then (connecting to the `"general"` room) no longer matches the UI.**

At this point, you want React to do two things:

1. Stop synchronizing with the old `roomId` (disconnect from the `"general"` room)

2. Start synchronizing with the new `roomId` (connect to the `"travel"` room)

Luckily, you've already taught React how to do both of these things! Your Effect's body specifies how to start synchronizing, and your cleanup function specifies how to stop synchronizing. All that React needs to do now is to call them in the correct order and with the correct props and state. Let's see how exactly that happens.

How React re-synchronizes your Effect {/*how-react-re-synchronizes-your-effect*/}

Recall that your `ChatRoom` component has received a new value for its `roomId` prop. It used to be `"general"`, and now it is `"travel"`. React needs to re-synchronize your Effect to re-connect you to a different room.

To **stop synchronizing,** React will call the cleanup function that your Effect returned after connecting to the `"general"` room. Since `roomId` was `"general"`, the cleanup function disconnects from the `"general"` room:

```js {6}
function ChatRoom({ roomId /* "general" */ }) {
useEffect(() => {
const connection = createConnection(serverUrl, roomId); // Connects to the "general" room
connection.connect();
return () => {
connection.disconnect(); // Disconnects from the "general" room
};
// ...
```

Then React will run the Effect that you've provided during this render. This time, `roomId` is `"travel"` so it will **start synchronizing** to the `"travel"` chat room (until its cleanup function is eventually called too):

```js {3,4}
function ChatRoom({ roomId /* "travel" */ }) {
useEffect(() => {
const connection = createConnection(serverUrl, roomId); // Connects to the "travel" room
connection.connect();
```

// ...
```

Thanks to this, you're now connected to the same room that the user chose in the UI. Disaster averted!

Every time after your component re-renders with a different `roomId`, your Effect will re-synchronize. For example, let's say the user changes `roomId` from `"travel"` to `"music"`. React will again **stop synchronizing** your Effect by calling its cleanup function (disconnecting you from the `"travel"` room). Then it will **start synchronizing** again by running its body with the new `roomId` prop (connecting you to the `"music"` room).

Finally, when the user goes to a different screen, `ChatRoom` unmounts. Now there is no need to stay connected at all. React will **stop synchronizing** your Effect one last time and disconnect you from the `"music"` chat room.

### Thinking from the Effect's perspective {/*thinking-from-the-effects-perspective*/}

Let's recap everything that's happened from the `ChatRoom` component's perspective:

1. `ChatRoom` mounted with `roomId` set to `"general"`
1. `ChatRoom` updated with `roomId` set to `"travel"`
1. `ChatRoom` updated with `roomId` set to `"music"`
1. `ChatRoom` unmounted

During each of these points in the component's lifecycle, your Effect did different things:

1. Your Effect connected to the `"general"` room
1. Your Effect disconnected from the `"general"` room and connected to the `"travel"` room
1. Your Effect disconnected from the `"travel"` room and connected to the `"music"` room
1. Your Effect disconnected from the `"music"` room

Now let's think about what happened from the perspective of the Effect itself:

```js
useEffect(() => {
// Your Effect connected to the room specified with roomId...
const connection = createConnection(serverUrl, roomId);
connection.connect();
return () => {
// ...until it disconnected
connection.disconnect();
};
}, [roomId]);
```
```

This code's structure might inspire you to see what happened as a sequence of non-overlapping time periods:

1. Your Effect connected to the `"general"` room (until it disconnected)

1. Your Effect connected to the `"travel"` room (until it disconnected)

1. Your Effect connected to the `"music"` room (until it disconnected)

Previously, you were thinking from the component's perspective. When you looked from the component's perspective, it was tempting to think of Effects as "callbacks" or "lifecycle events" that fire at a specific time like "after a render" or "before unmount". This way of thinking gets complicated very fast, so it's best to avoid.

Instead, always focus on a single start/stop cycle at a time. It shouldn't matter whether a component is mounting, updating, or unmounting. All you need to do is to describe how to start synchronization and how to stop it. If you do it well, your Effect will be resilient to being started and stopped as many times as it's needed.

This might remind you how you don't think whether a component is mounting or updating when you write the rendering logic that creates JSX. You describe what should be on the screen, and React [figures out the rest.](/learn/reacting-to-input-with-state)

How React verifies that your Effect can re-synchronize
{/*how-react-verifies-that-your-effect-can-re-synchronize*/}

Here is a live example that you can play with. Press "Open chat" to mount the `ChatRoom` component:

<Sandpack>

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

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

function ChatRoom({ roomId }) {
useEffect(() => {
const connection = createConnection(serverUrl, roomId);
connection.connect();
return () => connection.disconnect();
}, [roomId]);
return <h1>Welcome to the {roomId} room!</h1>;
}

export default function App() {
const [roomId, setRoomId] = useState('general');
const [show, setShow] = useState(false);
```

```
return (
<>
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
</select>
</label>
<button onClick={() => setShow(!show)}>
{show ? 'Close chat' : 'Open chat'}
</button>
{show && <hr />}
{show && <ChatRoom roomId={roomId} />}
</>
);
}
```

```js chat.js
export function createConnection(serverUrl, roomId) {
// A real implementation would actually connect to the server
return {
connect() {
console.log('■ Connecting to "' + roomId + '" room at ' + serverUrl + '...');
},
disconnect() {
console.log('■ Disconnected from "' + roomId + '" room at ' + serverUrl);
}
};
}
```

```css
input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }
```

</Sandpack>

Notice that when the component mounts for the first time, you see three logs:

1. `■ Connecting to "general" room at https://localhost:1234...` *(development-only)*
1. `■ Disconnected from "general" room at https://localhost:1234.` *(development-only)*
1. `■ Connecting to "general" room at https://localhost:1234...`

The first two logs are development-only. In development, React always remounts each component once.

React verifies that your Effect can re-synchronize by forcing it to do that immediately in development. This might remind you of opening a door and closing it an extra time to check if the door lock works. React starts and stops your Effect one extra time in development to check [you've implemented its cleanup well.](/learn/synchronizing-with-effects#how-to-handle-the-effect-firing-twice-in-development)

The main reason your Effect will re-synchronize in practice is if some data it uses has changed. In the sandbox above, change the selected chat room. Notice how, when the `roomId` changes, your Effect re-synchronizes.

However, there are also more unusual cases in which re-synchronization is necessary. For example, try editing the `serverUrl` in the sandbox above while the chat is open. Notice how the Effect re-synchronizes in response to your edits to the code. In the future, React may add more features that rely on re-synchronization.

How React knows that it needs to re-synchronize the Effect {/*how-react-knows-that-it-needs-to-re-synchronize-the-effect*/}

You might be wondering how React knew that your Effect needed to re-synchronize after `roomId` changes. It's because *you told React* that its code depends on `roomId` by including it in the [list of dependencies:](/learn/synchronizing-with-effects#step-2-specify-the-effect-dependencies)

```js {1,3,8}
function ChatRoom({ roomId }) { // The roomId prop may change over time
useEffect(() => {
const connection = createConnection(serverUrl, roomId); // This Effect reads roomId
connection.connect();
return () => {
connection.disconnect();
};
```

}, [roomId]); // So you tell React that this Effect "depends on" roomId

// ...

```

Here's how this works:

1. You knew `roomId` is a prop, which means it can change over time.

2. You knew that your Effect reads `roomId` (so its logic depends on a value that may change later).

3. This is why you specified it as your Effect's dependency (so that it re-synchronizes when `roomId` changes).

Every time after your component re-renders, React will look at the array of dependencies that you have passed. If any of the values in the array is different from the value at the same spot that you passed during the previous render, React will re-synchronize your Effect.

For example, if you passed `["general"]` during the initial render, and later you passed `["travel"]` during the next render, React will compare `"general"` and `"travel"`. These are different values (compared with [`Object.is`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/is)), so React will re-synchronize your Effect. On the other hand, if your component re-renders but `roomId` has not changed, your Effect will remain connected to the same room.

### Each Effect represents a separate synchronization process
{/*each-effect-represents-a-separate-synchronization-process*/}

Resist adding unrelated logic to your Effect only because this logic needs to run at the same time as an Effect you already wrote. For example, let's say you want to send an analytics event when the user visits the room. You already have an Effect that depends on `roomId`, so you might feel tempted to add the analytics call there:

```js {3}
function ChatRoom({ roomId }) {

useEffect(() => {

logVisit(roomId);

const connection = createConnection(serverUrl, roomId);

connection.connect();

return () => {

connection.disconnect();

};

}, [roomId]);

// ...

}
```

But imagine you later add another dependency to this Effect that needs to re-establish the connection. If this Effect re-synchronizes, it will also call `logVisit(roomId)` for the same room, which you did not
```

intend. Logging the visit **is a separate process** from connecting. Write them as two separate Effects:

```js {2-4}
function ChatRoom({ roomId }) {
useEffect(() => {
logVisit(roomId);
}, [roomId]);

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

Each Effect in your code should represent a separate and independent synchronization process.

In the above example, deleting one Effect wouldn't break the other Effect's logic. This is a good indication that they synchronize different things, and so it made sense to split them up. On the other hand, if you split up a cohesive piece of logic into separate Effects, the code may look "cleaner" but will be [more difficult to maintain.](/learn/you-might-not-need-an-effect#chains-of-computations) This is why you should think whether the processes are same or separate, not whether the code looks cleaner.

Effects "react" to reactive values {/*effects-react-to-reactive-values*/}

Your Effect reads two variables (`serverUrl` and `roomId`), but you only specified `roomId` as a dependency:

```js {5,10}
const serverUrl = 'https://localhost:1234';

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

```

Why doesn't `serverUrl` need to be a dependency?

This is because the `serverUrl` never changes due to a re-render. It's always the same no matter how many times the component re-renders and why. Since `serverUrl` never changes, it wouldn't make sense to specify it as a dependency. After all, dependencies only do something when they change over time!

On the other hand, `roomId` may be different on a re-render. **Props, state, and other values declared inside the component are _reactive_ because they're calculated during rendering and participate in the React data flow.**

If `serverUrl` was a state variable, it would be reactive. Reactive values must be included in dependencies:

```js {2,5,10}
function ChatRoom({ roomId }) { // Props change over time
const [serverUrl, setServerUrl] = useState('https://localhost:1234'); // State may change over time

useEffect(() => {
const connection = createConnection(serverUrl, roomId); // Your Effect reads props and state
connection.connect();
return () => {
connection.disconnect();
};
}, [roomId, serverUrl]); // So you tell React that this Effect "depends on" on props and state
// ...
}
```

By including `serverUrl` as a dependency, you ensure that the Effect re-synchronizes after it changes.

Try changing the selected chat room or edit the server URL in this sandbox:

<Sandpack>

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

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

useEffect(() => {

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

return (
<>
<label>
Server URL:{' '}
<input
value={serverUrl}
onChange={e => setServerUrl(e.target.value)}
/>
</label>
<h1>Welcome to the {roomId} room!</h1>
</>
);
}

export default function App() {
const [roomId, setRoomId] = useState('general');
return (
<>
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
</select>
</label>
<hr />
<ChatRoom roomId={roomId} />
</>
```

);
}
```

```js chat.js
export function createConnection(serverUrl, roomId) {
// A real implementation would actually connect to the server
return {
connect() {
console.log('■ Connecting to "' + roomId + '" room at ' + serverUrl + '...');
},
disconnect() {
console.log('■ Disconnected from "' + roomId + '" room at ' + serverUrl);
}
};
}
```

```css
input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }
```

</Sandpack>

Whenever you change a reactive value like `roomId` or `serverUrl`, the Effect re-connects to the chat server.

What an Effect with empty dependencies means
{/*what-an-effect-with-empty-dependencies-means*/}

What happens if you move both `serverUrl` and `roomId` outside the component?

```js {1,2}
const serverUrl = 'https://localhost:1234';
const roomId = 'general';

function ChatRoom() {
useEffect(() => {
const connection = createConnection(serverUrl, roomId);
connection.connect();

```
return () => {
connection.disconnect();
};
}, []); // ■ All dependencies declared
// ...
}
```

Now your Effect's code does not use *any* reactive values, so its dependencies can be empty (`[]`).

Thinking from the component's perspective, the empty `[]` dependency array means this Effect connects to the chat room only when the component mounts, and disconnects only when the component unmounts. (Keep in mind that React would still [re-synchronize it an extra time](#how-react-verifies-that-your-effect-can-re-synchronize) in development to stress-test your logic.)

<Sandpack>

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

const serverUrl = 'https://localhost:1234';
const roomId = 'general';

function ChatRoom() {
useEffect(() => {
const connection = createConnection(serverUrl, roomId);
connection.connect();
return () => connection.disconnect();
}, []);
return <h1>Welcome to the {roomId} room!</h1>;
}

export default function App() {
const [show, setShow] = useState(false);
return (
<>
<button onClick={() => setShow(!show)}>
{show ? 'Close chat' : 'Open chat'}
</button>
{show && <hr />}
```

```
{show && <ChatRoom />}
</>
);
}
```

```js chat.js
export function createConnection(serverUrl, roomId) {
// A real implementation would actually connect to the server
return {
connect() {
console.log('■ Connecting to "' + roomId + '" room at ' + serverUrl + '...');
},
disconnect() {
console.log('■ Disconnected from "' + roomId + '" room at ' + serverUrl);
}
};
}
```

```css
input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }
```

</Sandpack>

However, if you [think from the Effect's perspective,](#thinking-from-the-effects-perspective) you don't need to think about mounting and unmounting at all. What's important is you've specified what your Effect does to start and stop synchronizing. Today, it has no reactive dependencies. But if you ever want the user to change `roomId` or `serverUrl` over time (and they would become reactive), your Effect's code won't change. You will only need to add them to the dependencies.

All variables declared in the component body are reactive
{/*all-variables-declared-in-the-component-body-are-reactive*/}

Props and state aren't the only reactive values. Values that you calculate from them are also reactive. If the props or state change, your component will re-render, and the values calculated from them will also change. This is why all variables from the component body used by the Effect should be in the Effect dependency list.

Let's say that the user can pick a chat server in the dropdown, but they can also configure a default server in settings. Suppose you've already put the settings state in a

[context](/learn/scaling-up-with-reducer-and-context) so you read the `settings` from that context. Now you calculate the `serverUrl` based on the selected server from props and the default server:

```js {3,5,10}
function ChatRoom({ roomId, selectedServerUrl }) { // roomId is reactive
const settings = useContext(SettingsContext); // settings is reactive
const serverUrl = selectedServerUrl ?? settings.defaultServerUrl; // serverUrl is reactive
useEffect(() => {
const connection = createConnection(serverUrl, roomId); // Your Effect reads roomId and serverUrl
connection.connect();
return () => {
connection.disconnect();
};
}, [roomId, serverUrl]); // So it needs to re-synchronize when either of them changes!
// ...
}
```

In this example, `serverUrl` is not a prop or a state variable. It's a regular variable that you calculate during rendering. But it's calculated during rendering, so it can change due to a re-render. This is why it's reactive.

All values inside the component (including props, state, and variables in your component's body) are reactive. Any reactive value can change on a re-render, so you need to include reactive values as Effect's dependencies.

In other words, Effects "react" to all values from the component body.

<DeepDive>

Can global or mutable values be dependencies?
{/*can-global-or-mutable-values-be-dependencies*/}

Mutable values (including global variables) aren't reactive.

A mutable value like [`location.pathname`](https://developer.mozilla.org/en-US/docs/Web/API/Location/pathname) can't be a dependency. It's mutable, so it can change at any time completely outside of the React rendering data flow. Changing it wouldn't trigger a re-render of your component. Therefore, even if you specified it in the dependencies, React *wouldn't know* to re-synchronize the Effect when it changes. This also breaks the rules of React because reading mutable data during rendering (which is when you calculate the dependencies) breaks [purity of rendering.](/learn/keeping-components-pure) Instead, you should read and subscribe to an external mutable value with [`useSyncExternalStore`.](/learn/you-might-not-need-an-effect#subscribing-to-an-external-store)

A mutable value like [`ref.current`](/reference/react/useRef#reference) or things you read from it also can't be a dependency. The ref object returned by `useRef` itself can be a dependency, but its `current` property is intentionally mutable. It lets you [keep track of something without triggering a re-render.](/learn/referencing-values-with-refs) But since changing it doesn't trigger a re-render, it's not a reactive value, and React won't know to re-run your Effect when it changes.

As you'll learn below on this page, a linter will check for these issues automatically.

</DeepDive>

React verifies that you specified every reactive value as a dependency {/*react-verifies-that-you-specified-every-reactive-value-as-a-dependency*/}

If your linter is [configured for React,](/learn/editor-setup#linting) it will check that every reactive value used by your Effect's code is declared as its dependency. For example, this is a lint error because both `roomId` and `serverUrl` are reactive:

<Sandpack>

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

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

useEffect(() => {
const connection = createConnection(serverUrl, roomId);
connection.connect();
return () => connection.disconnect();
}, []); // <-- Something's wrong here!

return (
<>
<label>
Server URL:{' '}
<input
value={serverUrl}
onChange={e => setServerUrl(e.target.value)}
/>
</label>
<h1>Welcome to the {roomId} room!</h1>
</>
);
```

````
}

export default function App() {
const [roomId, setRoomId] = useState('general');
return (
<>
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
</select>
</label>
<hr />
<ChatRoom roomId={roomId} />
</>
);
}
```

```js chat.js
export function createConnection(serverUrl, roomId) {
// A real implementation would actually connect to the server
return {
connect() {
console.log('■ Connecting to "' + roomId + '" room at ' + serverUrl + '...');
},
disconnect() {
console.log('■ Disconnected from "' + roomId + '" room at ' + serverUrl);
}
};
}
```
````
````

```css
input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }
```

</Sandpack>

This may look like a React error, but really React is pointing out a bug in your code. Both `roomId` and `serverUrl` may change over time, but you're forgetting to re-synchronize your Effect when they change. You will remain connected to the initial `roomId` and `serverUrl` even after the user picks different values in the UI.

To fix the bug, follow the linter's suggestion to specify `roomId` and `serverUrl` as dependencies of your Effect:

```js {9}
function ChatRoom({ roomId }) { // roomId is reactive
const [serverUrl, setServerUrl] = useState('https://localhost:1234'); // serverUrl is reactive
useEffect(() => {
const connection = createConnection(serverUrl, roomId);
connection.connect();
return () => {
connection.disconnect();
};
}, [serverUrl, roomId]); // ■ All dependencies declared
// ...
}
```

Try this fix in the sandbox above. Verify that the linter error is gone, and the chat re-connects when needed.

<Note>

In some cases, React *knows* that a value never changes even though it's declared inside the component. For example, the [`set` function](/reference/react/useState#setstate) returned from `useState` and the ref object returned by [`useRef`](/reference/react/useRef) are *stable*--they are guaranteed to not change on a re-render. Stable values aren't reactive, so you may omit them from the list. Including them is allowed: they won't change, so it doesn't matter.

</Note>

What to do when you don't want to re-synchronize
{/*what-to-do-when-you-dont-want-to-re-synchronize*/}

In the previous example, you've fixed the lint error by listing `roomId` and `serverUrl` as dependencies.

However, you could instead "prove" to the linter that these values aren't reactive values, i.e. that they *can't* change as a result of a re-render. For example, if `serverUrl` and `roomId` don't depend on rendering and always have the same values, you can move them outside the component. Now they don't need to be dependencies:

```js {1,2,11}
const serverUrl = 'https://localhost:1234'; // serverUrl is not reactive
const roomId = 'general'; // roomId is not reactive

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

You can also move them *inside the Effect.* They aren't calculated during rendering, so they're not reactive:

```js {3,4,10}
function ChatRoom() {
useEffect(() => {
const serverUrl = 'https://localhost:1234'; // serverUrl is not reactive
const roomId = 'general'; // roomId is not reactive
const connection = createConnection(serverUrl, roomId);
connection.connect();
return () => {
connection.disconnect();
};
}, []); // ■ All dependencies declared
// ...
}
```

Effects are reactive blocks of code. They re-synchronize when the values you read inside of them change. Unlike event handlers, which only run once per interaction, Effects run whenever synchronization is necessary.

You can't "choose" your dependencies. Your dependencies must include every [reactive value](#all-variables-declared-in-the-component-body-are-reactive) you read in the Effect. The linter enforces this. Sometimes this may lead to problems like infinite loops and to your Effect re-synchronizing too often. Don't fix these problems by suppressing the linter! Here's what to try instead:

* **Check that your Effect represents an independent synchronization process.** If your Effect doesn't synchronize anything, [it might be unnecessary.](/learn/you-might-not-need-an-effect) If it synchronizes several independent things, [split it up.](#each-effect-represents-a-separate-synchronization-process)

* **If you want to read the latest value of props or state without "reacting" to it and re-synchronizing the Effect,** you can split your Effect into a reactive part (which you'll keep in the Effect) and a non-reactive part (which you'll extract into something called an _Effect Event_). [Read about separating Events from Effects.](/learn/separating-events-from-effects)

* **Avoid relying on objects and functions as dependencies.** If you create objects and functions during rendering and then read them from an Effect, they will be different on every render. This will cause your Effect to re-synchronize every time. [Read more about removing unnecessary dependencies from Effects.](/learn/removing-effect-dependencies)

<Pitfall>

The linter is your friend, but its powers are limited. The linter only knows when the dependencies are *wrong*. It doesn't know *the best* way to solve each case. If the linter suggests a dependency, but adding it causes a loop, it doesn't mean the linter should be ignored. You need to change the code inside (or outside) the Effect so that that value isn't reactive and doesn't *need* to be a dependency.

If you have an existing codebase, you might have some Effects that suppress the linter like this:

```js {3-4}
useEffect(() => {
// ...
// ■ Avoid suppressing the linter like this:
// eslint-ignore-next-line react-hooks/exhaustive-deps
}, []);
```

On the [next](/learn/separating-events-from-effects) [pages](/learn/removing-effect-dependencies), you'll learn how to fix this code without breaking the rules. It's always worth fixing!

</Pitfall>

<Recap>

- Components can mount, update, and unmount.

- Each Effect has a separate lifecycle from the surrounding component.

- Each Effect describes a separate synchronization process that can *start* and *stop*.

- When you write and read Effects, think from each individual Effect's perspective (how to start and stop synchronization) rather than from the component's perspective (how it mounts, updates, or unmounts).

- Values declared inside the component body are "reactive".

- Reactive values should re-synchronize the Effect because they can change over time.

- The linter verifies that all reactive values used inside the Effect are specified as dependencies.

- All errors flagged by the linter are legitimate. There's always a way to fix the code to not break the rules.

</Recap>

<Challenges>

Fix reconnecting on every keystroke {/*fix-reconnecting-on-every-keystroke*/}

In this example, the `ChatRoom` component connects to the chat room when the component mounts, disconnects when it unmounts, and reconnects when you select a different chat room. This behavior is correct, so you need to keep it working.

However, there is a problem. Whenever you type into the message box input at the bottom, `ChatRoom` *also* reconnects to the chat. (You can notice this by clearing the console and typing into the input.) Fix the issue so that this doesn't happen.

<Hint>

You might need to add a dependency array for this Effect. What dependencies should be there?

</Hint>

<Sandpack>

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

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

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

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

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

export default function App() {
const [roomId, setRoomId] = useState('general');
return (
<>
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
</select>
</label>
<hr />
<ChatRoom roomId={roomId} />
</>
);
}
```

```js chat.js
export function createConnection(serverUrl, roomId) {
// A real implementation would actually connect to the server
return {
```

```
connect() {
console.log('■ Connecting to "' + roomId + '" room at ' + serverUrl + '...');
},
disconnect() {
console.log('■ Disconnected from "' + roomId + '" room at ' + serverUrl);
}
};
}
```

```css
input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }
```

</Sandpack>

<Solution>

This Effect didn't have a dependency array at all, so it re-synchronized after every re-render. First, add a dependency array. Then, make sure that every reactive value used by the Effect is specified in the array. For example, `roomId` is reactive (because it's a prop), so it should be included in the array. This ensures that when the user selects a different room, the chat reconnects. On the other hand, `serverUrl` is defined outside the component. This is why it doesn't need to be in the array.

<Sandpack>

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

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

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

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

return (
```

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

export default function App() {
const [roomId, setRoomId] = useState('general');
return (
<>
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
</select>
</label>
<hr />
<ChatRoom roomId={roomId} />
</>
);
}
```

```js chat.js
export function createConnection(serverUrl, roomId) {
// A real implementation would actually connect to the server
return {
connect() {
```

```
console.log('■ Connecting to "' + roomId + '" room at ' + serverUrl + '...');
},
disconnect() {
console.log('■ Disconnected from "' + roomId + '" room at ' + serverUrl);
}
};
}
```

```css
input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }
```

</Sandpack>

</Solution>

Switch synchronization on and off {/*switch-synchronization-on-and-off*/}

In this example, an Effect subscribes to the window [`pointermove`](https://developer.mozilla.org/en-US/docs/Web/API/Element/pointermove_event) event to move a pink dot on the screen. Try hovering over the preview area (or touching the screen if you're on a mobile device), and see how the pink dot follows your movement.

There is also a checkbox. Ticking the checkbox toggles the `canMove` state variable, but this state variable is not used anywhere in the code. Your task is to change the code so that when `canMove` is `false` (the checkbox is ticked off), the dot should stop moving. After you toggle the checkbox back on (and set `canMove` to `true`), the box should follow the movement again. In other words, whether the dot can move or not should stay synchronized to whether the checkbox is checked.

<Hint>

You can't declare an Effect conditionally. However, the code inside the Effect can use conditions!

</Hint>

<Sandpack>

```js
import { useState, useEffect } from 'react';

export default function App() {
const [position, setPosition] = useState({ x: 0, y: 0 });
const [canMove, setCanMove] = useState(true);
```

```
useEffect(() => {
function handleMove(e) {
setPosition({ x: e.clientX, y: e.clientY });
}
window.addEventListener('pointermove', handleMove);
return () => window.removeEventListener('pointermove', handleMove);
}, []);

return (
<>
<label>
<input type="checkbox"
checked={canMove}
onChange={e => setCanMove(e.target.checked)}
/>
The dot is allowed to move
</label>
<hr />
<div style={{
position: 'absolute',
backgroundColor: 'pink',
borderRadius: '50%',
opacity: 0.6,
transform: `translate(${position.x}px, ${position.y}px)`,
pointerEvents: 'none',
left: -20,
top: -20,
width: 40,
height: 40,
}} />
</>
);
}
```

```css
body {
```

height: 200px;
}
```

</Sandpack>

<Solution>

One solution is to wrap the `setPosition` call into an `if (canMove) { ... }` condition:

<Sandpack>

```js
import { useState, useEffect } from 'react';

export default function App() {
const [position, setPosition] = useState({ x: 0, y: 0 });
const [canMove, setCanMove] = useState(true);

useEffect(() => {
function handleMove(e) {
if (canMove) {
setPosition({ x: e.clientX, y: e.clientY });
}
}
window.addEventListener('pointermove', handleMove);
return () => window.removeEventListener('pointermove', handleMove);
}, [canMove]);

return (
<>
<label>
<input type="checkbox"
checked={canMove}
onChange={e => setCanMove(e.target.checked)}
/>
The dot is allowed to move
</label>
<hr />
<div style={{
position: 'absolute',

backgroundColor: 'pink',
borderRadius: '50%',
opacity: 0.6,
transform: `translate(${position.x}px, ${position.y}px)`,
pointerEvents: 'none',
left: -20,
top: -20,
width: 40,
height: 40,
}} />
</>
);
}
```

```css
body {
height: 200px;
}
```

</Sandpack>

Alternatively, you could wrap the *event subscription* logic into an `if (canMove) { ... }` condition:

<Sandpack>

```js
import { useState, useEffect } from 'react';

export default function App() {
const [position, setPosition] = useState({ x: 0, y: 0 });
const [canMove, setCanMove] = useState(true);

useEffect(() => {
function handleMove(e) {
setPosition({ x: e.clientX, y: e.clientY });
}
if (canMove) {
window.addEventListener('pointermove', handleMove);

```
return () => window.removeEventListener('pointermove', handleMove);
}
}, [canMove]);

return (
<>
<label>
<input type="checkbox"
checked={canMove}
onChange={e => setCanMove(e.target.checked)}
/>
The dot is allowed to move
</label>
<hr />
<div style={{
position: 'absolute',
backgroundColor: 'pink',
borderRadius: '50%',
opacity: 0.6,
transform: `translate(${position.x}px, ${position.y}px)`,
pointerEvents: 'none',
left: -20,
top: -20,
width: 40,
height: 40,
}} />
</>
);
}
```

```css
body {
height: 200px;
}
```

</Sandpack>

In both of these cases, `canMove` is a reactive variable that you read inside the Effect. This is why it must be specified in the list of Effect dependencies. This ensures that the Effect re-synchronizes after every change to its value.

</Solution>

Investigate a stale value bug {/*investigate-a-stale-value-bug*/}

In this example, the pink dot should move when the checkbox is on, and should stop moving when the checkbox is off. The logic for this has already been implemented: the `handleMove` event handler checks the `canMove` state variable.

However, for some reason, the `canMove` state variable inside `handleMove` appears to be "stale": it's always `true`, even after you tick off the checkbox. How is this possible? Find the mistake in the code and fix it.

<Hint>

If you see a linter rule being suppressed, remove the suppression! That's where the mistakes usually are.

</Hint>

<Sandpack>

```js
import { useState, useEffect } from 'react';

export default function App() {
const [position, setPosition] = useState({ x: 0, y: 0 });
const [canMove, setCanMove] = useState(true);

function handleMove(e) {
if (canMove) {
setPosition({ x: e.clientX, y: e.clientY });
}
}

useEffect(() => {
window.addEventListener('pointermove', handleMove);
return () => window.removeEventListener('pointermove', handleMove);
// eslint-disable-next-line react-hooks/exhaustive-deps
}, []);

return (
```

<>
<label>
<input type="checkbox"
checked={canMove}
onChange={e => setCanMove(e.target.checked)}
/>
The dot is allowed to move
</label>
<hr />
<div style={{
position: 'absolute',
backgroundColor: 'pink',
borderRadius: '50%',
opacity: 0.6,
transform: `translate(${position.x}px, ${position.y}px)`,
pointerEvents: 'none',
left: -20,
top: -20,
width: 40,
height: 40,
}} />
</>
);
}
```

```css
body {
height: 200px;
}
```

</Sandpack>

<Solution>

The problem with the original code was suppressing the dependency linter. If you remove the suppression, you'll see that this Effect depends on the `handleMove` function. This makes sense: `handleMove` is declared inside the component body, which makes it a reactive value. Every reactive
```

value must be specified as a dependency, or it can potentially get stale over time!

The author of the original code has "lied" to React by saying that the Effect does not depend (`[]`) on any reactive values. This is why React did not re-synchronize the Effect after `canMove` has changed (and `handleMove` with it). Because React did not re-synchronize the Effect, the `handleMove` attached as a listener is the `handleMove` function created during the initial render. During the initial render, `canMove` was `true`, which is why `handleMove` from the initial render will forever see that value.

If you never suppress the linter, you will never see problems with stale values. There are a few different ways to solve this bug, but you should always start by removing the linter suppression. Then change the code to fix the lint error.

You can change the Effect dependencies to `[handleMove]`, but since it's going to be a newly defined function for every render, you might as well remove dependencies array altogether. Then the Effect *will* re-synchronize after every re-render:

<Sandpack>

```js
import { useState, useEffect } from 'react';

export default function App() {
const [position, setPosition] = useState({ x: 0, y: 0 });
const [canMove, setCanMove] = useState(true);

function handleMove(e) {
if (canMove) {
setPosition({ x: e.clientX, y: e.clientY });
}
}

useEffect(() => {
window.addEventListener('pointermove', handleMove);
return () => window.removeEventListener('pointermove', handleMove);
});

return (
<>
<label>
<input type="checkbox"
checked={canMove}
onChange={e => setCanMove(e.target.checked)}
/>
```

The dot is allowed to move

</label>

<hr />

<div style={{

position: 'absolute',

backgroundColor: 'pink',

borderRadius: '50%',

opacity: 0.6,

transform: `translate(${position.x}px, ${position.y}px)`,

pointerEvents: 'none',

left: -20,

top: -20,

width: 40,

height: 40,

}} />

</>

);

}

```

```css

body {

height: 200px;

}

```

</Sandpack>

This solution works, but it's not ideal. If you put `console.log('Resubscribing')` inside the Effect, you'll notice that it resubscribes after every re-render. Resubscribing is fast, but it would still be nice to avoid doing it so often.

A better fix would be to move the `handleMove` function *inside* the Effect. Then `handleMove` won't be a reactive value, and so your Effect won't depend on a function. Instead, it will need to depend on `canMove` which your code now reads from inside the Effect. This matches the behavior you wanted, since your Effect will now stay synchronized with the value of `canMove`:

<Sandpack>

```js

import { useState, useEffect } from 'react';

```
export default function App() {
const [position, setPosition] = useState({ x: 0, y: 0 });
const [canMove, setCanMove] = useState(true);

useEffect(() => {
function handleMove(e) {
if (canMove) {
setPosition({ x: e.clientX, y: e.clientY });
}
}

window.addEventListener('pointermove', handleMove);
return () => window.removeEventListener('pointermove', handleMove);
}, [canMove]);

return (
<>
<label>
<input type="checkbox"
checked={canMove}
onChange={e => setCanMove(e.target.checked)}
/>
The dot is allowed to move
</label>
<hr />
<div style={{
position: 'absolute',
backgroundColor: 'pink',
borderRadius: '50%',
opacity: 0.6,
transform: `translate(${position.x}px, ${position.y}px)`,
pointerEvents: 'none',
left: -20,
top: -20,
width: 40,
height: 40,
}} />
```

</>

);

}

```

```css

body {

height: 200px;

}

```

</Sandpack>

Try adding `console.log('Resubscribing')` inside the Effect body and notice that now it only resubscribes when you toggle the checkbox (`canMove` changes) or edit the code. This makes it better than the previous approach that always resubscribed.

You'll learn a more general approach to this type of problem in [Separating Events from Effects.](/learn/separating-events-from-effects)

</Solution>

Fix a connection switch {/*fix-a-connection-switch*/}

In this example, the chat service in `chat.js` exposes two different APIs: `createEncryptedConnection` and `createUnencryptedConnection`. The root `App` component lets the user choose whether to use encryption or not, and then passes down the corresponding API method to the child `ChatRoom` component as the `createConnection` prop.

Notice that initially, the console logs say the connection is not encrypted. Try toggling the checkbox on: nothing will happen. However, if you change the selected room after that, then the chat will reconnect *and* enable encryption (as you'll see from the console messages). This is a bug. Fix the bug so that toggling the checkbox *also* causes the chat to reconnect.

<Hint>

Suppressing the linter is always suspicious. Could this be a bug?

</Hint>

<Sandpack>

```js App.js

import { useState } from 'react';

import ChatRoom from './ChatRoom.js';

import {

createEncryptedConnection,

```
createUnencryptedConnection,
} from './chat.js';

export default function App() {
const [roomId, setRoomId] = useState('general');
const [isEncrypted, setIsEncrypted] = useState(false);
return (
<>
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
</select>
</label>
<label>
<input
type="checkbox"
checked={isEncrypted}
onChange={e => setIsEncrypted(e.target.checked)}
/>
Enable encryption
</label>
<hr />
<ChatRoom
roomId={roomId}
createConnection={isEncrypted ?
createEncryptedConnection :
createUnencryptedConnection
}
/>
</>
```

);
}
```

```js ChatRoom.js active
import { useState, useEffect } from 'react';

export default function ChatRoom({ roomId, createConnection }) {
useEffect(() => {
const connection = createConnection(roomId);
connection.connect();
return () => connection.disconnect();
// eslint-disable-next-line react-hooks/exhaustive-deps
}, [roomId]);

return <h1>Welcome to the {roomId} room!</h1>;
}
```

```js chat.js
export function createEncryptedConnection(roomId) {
// A real implementation would actually connect to the server
return {
connect() {
console.log('■ ■ Connecting to "' + roomId + '... (encrypted)');
},
disconnect() {
console.log('■ ■ Disconnected from "' + roomId + '" room (encrypted)');
}
};
}

export function createUnencryptedConnection(roomId) {
// A real implementation would actually connect to the server
return {
connect() {
console.log('■ Connecting to "' + roomId + '... (unencrypted)');
},
disconnect() {

```
console.log('■ Disconnected from "' + roomId + '" room (unencrypted)');
}
};
}
```

```css
label { display: block; margin-bottom: 10px; }
```

</Sandpack>

<Solution>

If you remove the linter suppression, you will see a lint error. The problem is that `createConnection` is a prop, so it's a reactive value. It can change over time! (And indeed, it should--when the user ticks the checkbox, the parent component passes a different value of the `createConnection` prop.) This is why it should be a dependency. Include it in the list to fix the bug:

<Sandpack>

```js App.js
import { useState } from 'react';
import ChatRoom from './ChatRoom.js';
import {
createEncryptedConnection,
createUnencryptedConnection,
} from './chat.js';

export default function App() {
const [roomId, setRoomId] = useState('general');
const [isEncrypted, setIsEncrypted] = useState(false);
return (
<>
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
<option value="general">general</option>
```

<option value="travel">travel</option>
<option value="music">music</option>
</select>
</label>
<label>
<input
type="checkbox"
checked={isEncrypted}
onChange={e => setIsEncrypted(e.target.checked)}
/>
Enable encryption
</label>
<hr />
<ChatRoom
roomId={roomId}
createConnection={isEncrypted ?
createEncryptedConnection :
createUnencryptedConnection
}
/>
</>
);
}
```

```js ChatRoom.js active
import { useState, useEffect } from 'react';

export default function ChatRoom({ roomId, createConnection }) {
useEffect(() => {
const connection = createConnection(roomId);
connection.connect();
return () => connection.disconnect();
}, [roomId, createConnection]);

return <h1>Welcome to the {roomId} room!</h1>;
}

```

```js chat.js
export function createEncryptedConnection(roomId) {
// A real implementation would actually connect to the server
return {
connect() {
console.log('■ ■ Connecting to "' + roomId + '... (encrypted)');
},
disconnect() {
console.log('■ ■ Disconnected from "' + roomId + '" room (encrypted)');
}
};
}

export function createUnencryptedConnection(roomId) {
// A real implementation would actually connect to the server
return {
connect() {
console.log('■ Connecting to "' + roomId + '... (unencrypted)');
},
disconnect() {
console.log('■ Disconnected from "' + roomId + '" room (unencrypted)');
}
};
}
```

```css
label { display: block; margin-bottom: 10px; }
```

</Sandpack>

It is correct that `createConnection` is a dependency. However, this code is a bit fragile because someone could edit the `App` component to pass an inline function as the value of this prop. In that case, its value would be different every time the `App` component re-renders, so the Effect might re-synchronize too often. To avoid this, you can pass `isEncrypted` down instead:

<Sandpack>
```

```js App.js
import { useState } from 'react';
import ChatRoom from './ChatRoom.js';

export default function App() {
const [roomId, setRoomId] = useState('general');
const [isEncrypted, setIsEncrypted] = useState(false);
return (
<>
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
</select>
</label>
<label>
<input
type="checkbox"
checked={isEncrypted}
onChange={e => setIsEncrypted(e.target.checked)}
/>
Enable encryption
</label>
<hr />
<ChatRoom
roomId={roomId}
isEncrypted={isEncrypted}
/>
</>
);
}
```

```

```js ChatRoom.js active
import { useState, useEffect } from 'react';
import {
createEncryptedConnection,
createUnencryptedConnection,
} from './chat.js';

export default function ChatRoom({ roomId, isEncrypted }) {
useEffect(() => {
const createConnection = isEncrypted ?
createEncryptedConnection :
createUnencryptedConnection;
const connection = createConnection(roomId);
connection.connect();
return () => connection.disconnect();
}, [roomId, isEncrypted]);

return <h1>Welcome to the {roomId} room!</h1>;
}
```

```js chat.js
export function createEncryptedConnection(roomId) {
// A real implementation would actually connect to the server
return {
connect() {
console.log('■ ■ Connecting to "' + roomId + '... (encrypted)');
},
disconnect() {
console.log('■ ■ Disconnected from "' + roomId + '" room (encrypted)');
}
};
}

export function createUnencryptedConnection(roomId) {
// A real implementation would actually connect to the server
return {

```
connect() {
console.log('■ Connecting to "' + roomId + '... (unencrypted)');
},
disconnect() {
console.log('■ Disconnected from "' + roomId + '" room (unencrypted)');
}
};
}
```

```css
label { display: block; margin-bottom: 10px; }
```

</Sandpack>

In this version, the `App` component passes a boolean prop instead of a function. Inside the Effect, you decide which function to use. Since both `createEncryptedConnection` and `createUnencryptedConnection` are declared outside the component, they aren't reactive, and don't need to be dependencies. You'll learn more about this in [Removing Effect Dependencies.](/learn/removing-effect-dependencies)

</Solution>

Populate a chain of select boxes {/*populate-a-chain-of-select-boxes*/}

In this example, there are two select boxes. One select box lets the user pick a planet. Another select box lets the user pick a place *on that planet.* The second box doesn't work yet. Your task is to make it show the places on the chosen planet.

Look at how the first select box works. It populates the `planetList` state with the result from the `"/planets"` API call. The currently selected planet's ID is kept in the `planetId` state variable. You need to find where to add some additional code so that the `placeList` state variable is populated with the result of the `"/planets/" + planetId + "/places"` API call.

If you implement this right, selecting a planet should populate the place list. Changing a planet should change the place list.

<Hint>

If you have two independent synchronization processes, you need to write two separate Effects.

</Hint>

<Sandpack>

```js App.js
```

```
import { useState, useEffect } from 'react';
import { fetchData } from './api.js';

export default function Page() {
const [planetList, setPlanetList] = useState([])
const [planetId, setPlanetId] = useState('');

const [placeList, setPlaceList] = useState([]);
const [placeId, setPlaceId] = useState('');

useEffect(() => {
let ignore = false;
fetchData('/planets').then(result => {
if (!ignore) {
console.log('Fetched a list of planets.');
setPlanetList(result);
setPlanetId(result[0].id); // Select the first planet
}
});
return () => {
ignore = true;
}
}, []);

return (
<>
<label>
Pick a planet:{' '}
<select value={planetId} onChange={e => {
setPlanetId(e.target.value);
}}>
{planetList.map(planet =>
<option key={planet.id} value={planet.id}>{planet.name}</option>
)}
</select>
</label>
<label>
Pick a place:{' '}
```

```
<select value={placeId} onChange={e => {
setPlaceId(e.target.value);
}}>
{placeList.map(place =>
<option key={place.id} value={place.id}>{place.name}</option>
)}
</select>
</label>
<hr />
<p>You are going to: {placeId || '???'} on {planetId || '???'} </p>
</>
);
}
```

```js api.js hidden
export function fetchData(url) {
if (url === '/planets') {
return fetchPlanets();
} else if (url.startsWith('/planets/')) {
const match = url.match(/^\/planets\/([\w-]+)\/places(\/)?$/);
if (!match || !match[1] || !match[1].length) {
throw Error('Expected URL like "/planets/earth/places". Received: "' + url + '".');
}
return fetchPlaces(match[1]);
} else throw Error('Expected URL like "/planets" or "/planets/earth/places". Received: "' + url + '".');
}

async function fetchPlanets() {
return new Promise(resolve => {
setTimeout(() => {
resolve([{
id: 'earth',
name: 'Earth'
}, {
id: 'venus',
name: 'Venus'
```

```
}, {
id: 'mars',
name: 'Mars'
}]);
}, 1000);
});
}

async function fetchPlaces(planetId) {
if (typeof planetId !== 'string') {
throw Error(
'fetchPlaces(planetId) expects a string argument. ' +
'Instead received: ' + planetId + '.'
);
}
return new Promise(resolve => {
setTimeout(() => {
if (planetId === 'earth') {
resolve([{
id: 'laos',
name: 'Laos'
}, {
id: 'spain',
name: 'Spain'
}, {
id: 'vietnam',
name: 'Vietnam'
}]);
} else if (planetId === 'venus') {
resolve([{
id: 'aurelia',
name: 'Aurelia'
}, {
id: 'diana-chasma',
name: 'Diana Chasma'
}, {
```

id: 'kumsong-vallis',
name: 'K■ms■ng Vallis'
}]);
} else if (planetId === 'mars') {
resolve([{
id: 'aluminum-city',
name: 'Aluminum City'
}, {
id: 'new-new-york',
name: 'New New York'
}, {
id: 'vishniac',
name: 'Vishniac'
}]);
} else throw Error('Unknown planet ID: ' + planetId);
}, 1000);
});
}
```

```css
label { display: block; margin-bottom: 10px; }
```

</Sandpack>

<Solution>

There are two independent synchronization processes:

- The first select box is synchronized to the remote list of planets.
- The second select box is synchronized to the remote list of places for the current `planetId`.

This is why it makes sense to describe them as two separate Effects. Here's an example of how you could do this:

<Sandpack>

```js App.js
import { useState, useEffect } from 'react';
import { fetchData } from './api.js';

```
export default function Page() {
const [planetList, setPlanetList] = useState([])
const [planetId, setPlanetId] = useState('');

const [placeList, setPlaceList] = useState([]);
const [placeId, setPlaceId] = useState('');

useEffect(() => {
let ignore = false;
fetchData('/planets').then(result => {
if (!ignore) {
console.log('Fetched a list of planets.');
setPlanetList(result);
setPlanetId(result[0].id); // Select the first planet
}
});
return () => {
ignore = true;
}
}, []);

useEffect(() => {
if (planetId === '') {
// Nothing is selected in the first box yet
return;
}

let ignore = false;
fetchData('/planets/' + planetId + '/places').then(result => {
if (!ignore) {
console.log('Fetched a list of places on '' + planetId + ''.');
setPlaceList(result);
setPlaceId(result[0].id); // Select the first place
}
});
return () => {
ignore = true;
}
```

```
}, [planetId]);

return (
<>
<label>
Pick a planet:{' '}
<select value={planetId} onChange={e => {
setPlanetId(e.target.value);
}}>
{planetList.map(planet =>
<option key={planet.id} value={planet.id}>{planet.name}</option>
)}
</select>
</label>
<label>
Pick a place:{' '}
<select value={placeId} onChange={e => {
setPlaceId(e.target.value);
}}>
{placeList.map(place =>
<option key={place.id} value={place.id}>{place.name}</option>
)}
</select>
</label>
<hr />
<p>You are going to: {placeId || '???'} on {planetId || '???'} </p>
</>
);
}
```

```js api.js hidden
export function fetchData(url) {
if (url === '/planets') {
return fetchPlanets();
} else if (url.startsWith('/planets/')) {
const match = url.match(/^\/planets\/([\w-]+)\/places(\/)?$/);
```

```
if (!match || !match[1] || !match[1].length) {
throw Error('Expected URL like "/planets/earth/places". Received: "' + url + '".');
}
return fetchPlaces(match[1]);
} else throw Error('Expected URL like "/planets" or "/planets/earth/places". Received: "' + url + '".');
}

async function fetchPlanets() {
return new Promise(resolve => {
setTimeout(() => {
resolve([{
id: 'earth',
name: 'Earth'
}, {
id: 'venus',
name: 'Venus'
}, {
id: 'mars',
name: 'Mars'
}]);
}, 1000);
});
}

async function fetchPlaces(planetId) {
if (typeof planetId !== 'string') {
throw Error(
'fetchPlaces(planetId) expects a string argument. ' +
'Instead received: ' + planetId + '.'
);
}
return new Promise(resolve => {
setTimeout(() => {
if (planetId === 'earth') {
resolve([{
id: 'laos',
name: 'Laos'
```

```
}, {
id: 'spain',
name: 'Spain'
}, {
id: 'vietnam',
name: 'Vietnam'
}]);
} else if (planetId === 'venus') {
resolve([{
id: 'aurelia',
name: 'Aurelia'
}, {
id: 'diana-chasma',
name: 'Diana Chasma'
}, {
id: 'kumsong-vallis',
name: 'K■ms■ng Vallis'
}]);
} else if (planetId === 'mars') {
resolve([{
id: 'aluminum-city',
name: 'Aluminum City'
}, {
id: 'new-new-york',
name: 'New New York'
}, {
id: 'vishniac',
name: 'Vishniac'
}]);
} else throw Error('Unknown planet ID: ' + planetId);
}, 1000);
});
}
```

```css
```

label { display: block; margin-bottom: 10px; }
```

</Sandpack>

This code is a bit repetitive. However, that's not a good reason to combine it into a single Effect! If you did this, you'd have to combine both Effect's dependencies into one list, and then changing the planet would refetch the list of all planets. Effects are not a tool for code reuse.

Instead, to reduce repetition, you can extract some logic into a custom Hook like `useSelectOptions` below:

<Sandpack>

```js App.js
import { useState } from 'react';
import { useSelectOptions } from './useSelectOptions.js';

export default function Page() {
const [
planetList,
planetId,
setPlanetId
] = useSelectOptions('/planets');

const [
placeList,
placeId,
setPlaceId
] = useSelectOptions(planetId ? `/planets/${planetId}/places` : null);

return (
<>
<label>
Pick a planet:{' '}
<select value={planetId} onChange={e => {
setPlanetId(e.target.value);
}}>
{planetList?.map(planet =>
<option key={planet.id} value={planet.id}>{planet.name}</option>
)}
</select>

```
</label>
<label>
Pick a place:{' '}
<select value={placeId} onChange={e => {
setPlaceId(e.target.value);
}}>
{placeList?.map(place =>
<option key={place.id} value={place.id}>{place.name}</option>
)}
</select>
</label>
<hr />
<p>You are going to: {placeId || '...'} on {planetId || '...'} </p>
</>
);
}
```

```js useSelectOptions.js
import { useState, useEffect } from 'react';
import { fetchData } from './api.js';

export function useSelectOptions(url) {
const [list, setList] = useState(null);
const [selectedId, setSelectedId] = useState('');
useEffect(() => {
if (url === null) {
return;
}

let ignore = false;
fetchData(url).then(result => {
if (!ignore) {
setList(result);
setSelectedId(result[0].id);
}
});
```

```
return () => {
ignore = true;
}
}, [url]);
return [list, selectedId, setSelectedId];
}
```

```js api.js hidden
export function fetchData(url) {
if (url === '/planets') {
return fetchPlanets();
} else if (url.startsWith('/planets/')) {
const match = url.match(/^\/planets\/([\w-]+)\/places(\/)?$/);
if (!match || !match[1] || !match[1].length) {
throw Error('Expected URL like "/planets/earth/places". Received: "' + url + '".');
}
return fetchPlaces(match[1]);
} else throw Error('Expected URL like "/planets" or "/planets/earth/places". Received: "' + url + '".');
}

async function fetchPlanets() {
return new Promise(resolve => {
setTimeout(() => {
resolve([{
id: 'earth',
name: 'Earth'
}, {
id: 'venus',
name: 'Venus'
}, {
id: 'mars',
name: 'Mars'
}]);
}, 1000);
});
}
```

```
async function fetchPlaces(planetId) {
if (typeof planetId !== 'string') {
throw Error(
'fetchPlaces(planetId) expects a string argument. ' +
'Instead received: ' + planetId + '.'
);
}
return new Promise(resolve => {
setTimeout(() => {
if (planetId === 'earth') {
resolve([{
id: 'laos',
name: 'Laos'
}, {
id: 'spain',
name: 'Spain'
}, {
id: 'vietnam',
name: 'Vietnam'
}]);
} else if (planetId === 'venus') {
resolve([{
id: 'aurelia',
name: 'Aurelia'
}, {
id: 'diana-chasma',
name: 'Diana Chasma'
}, {
id: 'kumsong-vallis',
name: 'K■ms■ng Vallis'
}]);
} else if (planetId === 'mars') {
resolve([{
id: 'aluminum-city',
name: 'Aluminum City'
```

```
}, {
id: 'new-new-york',
name: 'New New York'
}, {
id: 'vishniac',
name: 'Vishniac'
}]);
} else throw Error('Unknown planet ID: ' + planetId);
}, 1000);
});
}
```

```css
label { display: block; margin-bottom: 10px; }
```

</Sandpack>

Check the `useSelectOptions.js` tab in the sandbox to see how it works. Ideally, most Effects in your application should eventually be replaced by custom Hooks, whether written by you or by the community. Custom Hooks hide the synchronization logic, so the calling component doesn't know about the Effect. As you keep working on your app, you'll develop a palette of Hooks to choose from, and eventually you won't need to write Effects in your components very often.

</Solution>

</Challenges>

---

title: JavaScript in JSX with Curly Braces

<Intro>

JSX lets you write HTML-like markup inside a JavaScript file, keeping rendering logic and content in the same place. Sometimes you will want to add a little JavaScript logic or reference a dynamic property inside that markup. In this situation, you can use curly braces in your JSX to open a window to JavaScript.

</Intro>

<YouWillLearn>

* How to pass strings with quotes

* How to reference a JavaScript variable inside JSX with curly braces
* How to call a JavaScript function inside JSX with curly braces
* How to use a JavaScript object inside JSX with curly braces

</YouWillLearn>

Passing strings with quotes {/*passing-strings-with-quotes*/}

When you want to pass a string attribute to JSX, you put it in single or double quotes:

<Sandpack>

```js
export default function Avatar() {
return (
<img
className="avatar"
src="https://i.imgur.com/7vQD0fPs.jpg"
alt="Gregorio Y. Zara"
/>
);
}
```

```css
.avatar { border-radius: 50%; height: 90px; }
```

</Sandpack>

Here, `"https://i.imgur.com/7vQD0fPs.jpg"` and `"Gregorio Y. Zara"` are being passed as strings.

But what if you want to dynamically specify the `src` or `alt` text? You could **use a value from JavaScript by replacing `"` and `"` with `{` and `}`**:

<Sandpack>

```js
export default function Avatar() {
const avatar = 'https://i.imgur.com/7vQD0fPs.jpg';
const description = 'Gregorio Y. Zara';
return (
<img
```

````
className="avatar"
src={avatar}
alt={description}
/>
);
}
```

```css
.avatar { border-radius: 50%; height: 90px; }
```

</Sandpack>

Notice the difference between `className="avatar"`, which specifies an `"avatar"` CSS class name that makes the image round, and `src={avatar}` that reads the value of the JavaScript variable called `avatar`. That's because curly braces let you work with JavaScript right there in your markup!

## Using curly braces: A window into the JavaScript world
{/*using-curly-braces-a-window-into-the-javascript-world*/}

JSX is a special way of writing JavaScript. That means it's possible to use JavaScript inside it—with curly braces `{ }`. The example below first declares a name for the scientist, `name`, then embeds it with curly braces inside the `<h1>`:

<Sandpack>

```js
export default function TodoList() {
const name = 'Gregorio Y. Zara';
return (
<h1>{name}'s To Do List</h1>
);
}
```

</Sandpack>

Try changing the `name`'s value from `'Gregorio Y. Zara'` to `'Hedy Lamarr'`. See how the list title changes?

Any JavaScript expression will work between curly braces, including function calls like `formatDate()`:

<Sandpack>

```js
const today = new Date();

function formatDate(date) {
return new Intl.DateTimeFormat(
'en-US',
{ weekday: 'long' }
).format(date);
}

export default function TodoList() {
return (
<h1>To Do List for {formatDate(today)}</h1>
);
}
```

</Sandpack>

### Where to use curly braces {/*where-to-use-curly-braces*/}

You can only use curly braces in two ways inside JSX:

1. **As text** directly inside a JSX tag: `<h1>{name}'s To Do List</h1>` works, but `<{tag}>Gregorio Y. Zara's To Do List</{tag}>` will not.
2. **As attributes** immediately following the `=` sign: `src={avatar}` will read the `avatar` variable, but `src="{avatar}"` will pass the string `"{avatar}"`.

## Using "double curlies": CSS and other objects in JSX {/*using-double-curlies-css-and-other-objects-in-jsx*/}

In addition to strings, numbers, and other JavaScript expressions, you can even pass objects in JSX. Objects are also denoted with curly braces, like `{ name: "Hedy Lamarr", inventions: 5 }`. Therefore, to pass a JS object in JSX, you must wrap the object in another pair of curly braces: `person={{ name: "Hedy Lamarr", inventions: 5 }}`.

You may see this with inline CSS styles in JSX. React does not require you to use inline styles (CSS classes work great for most cases). But when you need an inline style, you pass an object to the `style` attribute:

<Sandpack>

```js
export default function TodoList() {
return (
```

<ul style={{
backgroundColor: 'black',
color: 'pink'
}}>
<li>Improve the videophone</li>
<li>Prepare aeronautics lectures</li>
<li>Work on the alcohol-fuelled engine</li>
</ul>
);
}
```

```css
body { padding: 0; margin: 0 }
ul { padding: 20px 20px 20px 40px; margin: 0; }
```

</Sandpack>

Try changing the values of `backgroundColor` and `color`.

You can really see the JavaScript object inside the curly braces when you write it like this:

```js {2-5}
<ul style={
{
backgroundColor: 'black',
color: 'pink'
}
}>
```

The next time you see `{{` and `}}` in JSX, know that it's nothing more than an object inside the JSX curlies!

<Pitfall>

Inline `style` properties are written in camelCase. For example, HTML `<ul style="background-color: black">` would be written as `<ul style={{ backgroundColor: 'black' }}>` in your component.

</Pitfall>
```

## More fun with JavaScript objects and curly braces
{/*more-fun-with-javascript-objects-and-curly-braces*/}

You can move several expressions into one object, and reference them in your JSX inside curly braces:

<Sandpack>

```js
const person = {
name: 'Gregorio Y. Zara',
theme: {
backgroundColor: 'black',
color: 'pink'
}
};

export default function TodoList() {
return (
<div style={person.theme}>
<h1>{person.name}'s Todos</h1>
<img
className="avatar"
src="https://i.imgur.com/7vQD0fPs.jpg"
alt="Gregorio Y. Zara"
/>
<ul>
<li>Improve the videophone</li>
<li>Prepare aeronautics lectures</li>
<li>Work on the alcohol-fuelled engine</li>
</ul>
</div>
);
}
```

```css
body { padding: 0; margin: 0 }
body > div > div { padding: 20px; }
.avatar { border-radius: 50%; height: 90px; }
```

```

</Sandpack>

In this example, the `person` JavaScript object contains a `name` string and a `theme` object:

```js
const person = {
name: 'Gregorio Y. Zara',
theme: {
backgroundColor: 'black',
color: 'pink'
}
};
```

The component can use these values from `person` like so:

```js
<div style={person.theme}>
<h1>{person.name}'s Todos</h1>
```

JSX is very minimal as a templating language because it lets you organize data and logic using JavaScript.

<Recap>

Now you know almost everything about JSX:

* JSX attributes inside quotes are passed as strings.
* Curly braces let you bring JavaScript logic and variables into your markup.
* They work inside the JSX tag content or immediately after `=` in attributes.
* `{{` and `}}` is not special syntax: it's a JavaScript object tucked inside JSX curly braces.

</Recap>

<Challenges>

#### Fix the mistake {/*fix-the-mistake*/}

This code crashes with an error saying `Objects are not valid as a React child`:

<Sandpack>

```js

```
const person = {
name: 'Gregorio Y. Zara',
theme: {
backgroundColor: 'black',
color: 'pink'
}
};

export default function TodoList() {
return (
<div style={person.theme}>
<h1>{person}'s Todos</h1>
<img
className="avatar"
src="https://i.imgur.com/7vQD0fPs.jpg"
alt="Gregorio Y. Zara"
/>
<ul>
<li>Improve the videophone</li>
<li>Prepare aeronautics lectures</li>
<li>Work on the alcohol-fuelled engine</li>
</ul>
</div>
);
}
```

```css
body { padding: 0; margin: 0 }
body > div > div { padding: 20px; }
.avatar { border-radius: 50%; height: 90px; }
```

</Sandpack>

Can you find the problem?

<Hint>Look for what's inside the curly braces. Are we putting the right thing there?</Hint>

<Solution>

This is happening because this example renders *an object itself* into the markup rather than a string: `<h1>{person}'s Todos</h1>` is trying to render the entire `person` object! Including raw objects as text content throws an error because React doesn't know how you want to display them.

To fix it, replace `<h1>{person}'s Todos</h1>` with `<h1>{person.name}'s Todos</h1>`:

<Sandpack>

```js
const person = {
name: 'Gregorio Y. Zara',
theme: {
backgroundColor: 'black',
color: 'pink'
}
};

export default function TodoList() {
return (
<div style={person.theme}>
<h1>{person.name}'s Todos</h1>
<img
className="avatar"
src="https://i.imgur.com/7vQD0fPs.jpg"
alt="Gregorio Y. Zara"
/>
<ul>
<li>Improve the videophone</li>
<li>Prepare aeronautics lectures</li>
<li>Work on the alcohol-fuelled engine</li>
</ul>
</div>
);
}
```

```css
body { padding: 0; margin: 0 }
```

body > div > div { padding: 20px; }
.avatar { border-radius: 50%; height: 90px; }
```

</Sandpack>

</Solution>

#### Extract information into an object {/*extract-information-into-an-object*/}

Extract the image URL into the `person` object.

<Sandpack>

```js
const person = {
name: 'Gregorio Y. Zara',
theme: {
backgroundColor: 'black',
color: 'pink'
}
};

export default function TodoList() {
return (

<h1>{person.name}'s Todos</h1>
<img
className="avatar"
src="https://i.imgur.com/7vQD0fPs.jpg"
alt="Gregorio Y. Zara"
/>
<ul>
<li>Improve the videophone</li>
<li>Prepare aeronautics lectures</li>
<li>Work on the alcohol-fuelled engine</li>
</ul>

);
}

```

```css
body { padding: 0; margin: 0 }
body > div > div { padding: 20px; }
.avatar { border-radius: 50%; height: 90px; }
```

</Sandpack>

<Solution>

Move the image URL into a property called `person.imageUrl` and read it from the `<img>` tag using the curlies:

<Sandpack>

```js
const person = {
name: 'Gregorio Y. Zara',
imageUrl: "https://i.imgur.com/7vQD0fPs.jpg",
theme: {
backgroundColor: 'black',
color: 'pink'
}
};

export default function TodoList() {
return (
<div style={person.theme}>
<h1>{person.name}'s Todos</h1>
<img
className="avatar"
src={person.imageUrl}
alt="Gregorio Y. Zara"
/>
<ul>
<li>Improve the videophone</li>
<li>Prepare aeronautics lectures</li>
<li>Work on the alcohol-fuelled engine</li>
```
```

</ul>

);

}

```

```css

body { padding: 0; margin: 0 }

body > div > div { padding: 20px; }

.avatar { border-radius: 50%; height: 90px; }

```

</Sandpack>

</Solution>

#### Write an expression inside JSX curly braces {/*write-an-expression-inside-jsx-curly-braces*/}

In the object below, the full image URL is split into four parts: base URL, `imageId`, `imageSize`, and file extension.

We want the image URL to combine these attributes together: base URL (always `'https://i.imgur.com/'`), `imageId` (`'7vQD0fP'`), `imageSize` (`'s'`), and file extension (always `'.jpg'`). However, something is wrong with how the `<img>` tag specifies its `src`.

Can you fix it?

<Sandpack>

```js

const baseUrl = 'https://i.imgur.com/';

const person = {

name: 'Gregorio Y. Zara',

imageId: '7vQD0fP',

imageSize: 's',

theme: {

backgroundColor: 'black',

color: 'pink'

}

};

export default function TodoList() {

return (

<h1>{person.name}'s Todos</h1>
<img
className="avatar"
src="{baseUrl}{person.imageId}{person.imageSize}.jpg"
alt={person.name}
/>
<ul>
<li>Improve the videophone</li>
<li>Prepare aeronautics lectures</li>
<li>Work on the alcohol-fuelled engine</li>
</ul>

);
}
```

```css
body { padding: 0; margin: 0 }
body > div > div { padding: 20px; }
.avatar { border-radius: 50%; }
```

</Sandpack>

To check that your fix worked, try changing the value of `imageSize` to `'b'`. The image should resize after your edit.

<Solution>

You can write it as `src={baseUrl + person.imageId + person.imageSize + '.jpg'}`.

1. `{` opens the JavaScript expression
2. `baseUrl + person.imageId + person.imageSize + '.jpg'` produces the correct URL string
3. `}` closes the JavaScript expression

<Sandpack>

```js
const baseUrl = 'https://i.imgur.com/';
const person = {

````
name: 'Gregorio Y. Zara',
imageId: '7vQD0fP',
imageSize: 's',
theme: {
backgroundColor: 'black',
color: 'pink'
}
};

export default function TodoList() {
return (
<div style={person.theme}>
<h1>{person.name}'s Todos</h1>
<img
className="avatar"
src={baseUrl + person.imageId + person.imageSize + '.jpg'}
alt={person.name}
/>

Improve the videophone
Prepare aeronautics lectures
Work on the alcohol-fuelled engine

</div>
);
}
```

```css
body { padding: 0; margin: 0 }
body > div > div { padding: 20px; }
.avatar { border-radius: 50%; }
```

</Sandpack>

You can also move this expression into a separate function like `getImageUrl` below:

<Sandpack>
````
````

```js App.js
import { getImageUrl } from './utils.js'

const person = {
name: 'Gregorio Y. Zara',
imageId: '7vQD0fP',
imageSize: 's',
theme: {
backgroundColor: 'black',
color: 'pink'
}
};

export default function TodoList() {
return (
<div style={person.theme}>
<h1>{person.name}'s Todos</h1>
<img
className="avatar"
src={getImageUrl(person)}
alt={person.name}
/>
<ul>
<li>Improve the videophone</li>
<li>Prepare aeronautics lectures</li>
<li>Work on the alcohol-fuelled engine</li>
</ul>
</div>
);
}
```

```js utils.js
export function getImageUrl(person) {
return (
'https://i.imgur.com/' +
person.imageId +
```

person.imageSize +

'.jpg'

);

}

```

```css

body { padding: 0; margin: 0 }

body > div > div { padding: 20px; }

.avatar { border-radius: 50%; }

```

</Sandpack>

Variables and functions can help you keep the markup simple!

</Solution>

</Challenges>

---

title: Sharing State Between Components

<Intro>

Sometimes, you want the state of two components to always change together. To do it, remove state from both of them, move it to their closest common parent, and then pass it down to them via props. This is known as *lifting state up,* and it's one of the most common things you will do writing React code.

</Intro>

<YouWillLearn>

- How to share state between components by lifting it up

- What are controlled and uncontrolled components

</YouWillLearn>

Lifting state up by example {/*lifting-state-up-by-example*/}

In this example, a parent `Accordion` component renders two separate `Panel`s:

* `Accordion`

- `Panel`

- `Panel`
```

Each `Panel` component has a boolean `isActive` state that determines whether its content is visible.

Press the Show button for both panels:

<Sandpack>

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

function Panel({ title, children }) {
const [isActive, setIsActive] = useState(false);
return (
<section className="panel">
<h3>{title}</h3>
{isActive ? (
<p>{children}</p>
) : (
<button onClick={() => setIsActive(true)}>
Show
</button>
)}
</section>
);
}

export default function Accordion() {
return (
<>
<h2>Almaty, Kazakhstan</h2>
<Panel title="About">
With a population of about 2 million, Almaty is Kazakhstan's largest city. From 1929 to 1997, it was its capital city.
</Panel>
<Panel title="Etymology">
The name comes from <span lang="kk-KZ">■■■■</span>, the Kazakh word for "apple" and is often translated as "full of apples". In fact, the region surrounding Almaty is thought to be the ancestral home of the apple, and the wild <i lang="la">Malus sieversii</i> is considered a likely candidate for the ancestor of the modern domestic apple.
</Panel>
</>
```

);
}
```

```css
h3, p { margin: 5px 0px; }
.panel {
padding: 10px;
border: 1px solid #aaa;
}
```

</Sandpack>

Notice how pressing one panel's button does not affect the other panel--they are independent.

<DiagramGroup>

<Diagram name="sharing_state_child" height={367} width={477} alt="Diagram showing a tree of three components, one parent labeled Accordion and two children labeled Panel. Both Panel components contain isActive with value false.">

Initially, each `Panel`'s `isActive` state is `false`, so they both appear collapsed

</Diagram>

<Diagram name="sharing_state_child_clicked" height={367} width={480} alt="The same diagram as the previous, with the isActive of the first child Panel component highlighted indicating a click with the isActive value set to true. The second Panel component still contains value false." >

Clicking either `Panel`'s button will only update that `Panel`'s `isActive` state alone

</Diagram>

</DiagramGroup>

But now let's say you want to change it so that only one panel is expanded at any given time. With that design, expanding the second panel should collapse the first one. How would you do that?

To coordinate these two panels, you need to "lift their state up" to a parent component in three steps:

1. **Remove** state from the child components.
2. **Pass** hardcoded data from the common parent.
3. **Add** state to the common parent and pass it down together with the event handlers.

This will allow the `Accordion` component to coordinate both `Panel`s and only expand one at a time.
```

Step 1: Remove state from the child components
{/*step-1-remove-state-from-the-child-components*/}

You will give control of the `Panel`'s `isActive` to its parent component. This means that the parent component will pass `isActive` to `Panel` as a prop instead. Start by **removing this line** from the `Panel` component:

```js
const [isActive, setIsActive] = useState(false);
```

And instead, add `isActive` to the `Panel`'s list of props:

```js
function Panel({ title, children, isActive }) {
```

Now the `Panel`'s parent component can *control* `isActive` by [passing it down as a prop.](/learn/passing-props-to-a-component) Conversely, the `Panel` component now has *no control* over the value of `isActive`--it's now up to the parent component!

Step 2: Pass hardcoded data from the common parent
{/*step-2-pass-hardcoded-data-from-the-common-parent*/}

To lift state up, you must locate the closest common parent component of *both* of the child components that you want to coordinate:

* `Accordion` *(closest common parent)*
- `Panel`
- `Panel`

In this example, it's the `Accordion` component. Since it's above both panels and can control their props, it will become the "source of truth" for which panel is currently active. Make the `Accordion` component pass a hardcoded value of `isActive` (for example, `true`) to both panels:

<Sandpack>

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

export default function Accordion() {
return (
<>
<h2>Almaty, Kazakhstan</h2>
<Panel title="About" isActive={true}>
With a population of about 2 million, Almaty is Kazakhstan's largest city. From 1929 to 1997, it was its capital city.
```

</Panel>

<Panel title="Etymology" isActive={true}>

The name comes from ■■■■, the Kazakh word for "apple" and is often translated as "full of apples". In fact, the region surrounding Almaty is thought to be the ancestral home of the apple, and the wild Malus sieversii is considered a likely candidate for the ancestor of the modern domestic apple.

</Panel>

</>

);

}

function Panel({ title, children, isActive }) {

return (

<section className="panel">

<h3>{title}</h3>

{isActive ? (

{children}

) : (

<button onClick={() => setIsActive(true)}>

Show

</button>

)}

</section>

);

}

```

```css

h3, p { margin: 5px 0px; }

.panel {

padding: 10px;

border: 1px solid #aaa;

}

```

</Sandpack>

Try editing the hardcoded `isActive` values in the `Accordion` component and see the result on the screen.
```

Step 3: Add state to the common parent {/*step-3-add-state-to-the-common-parent*/}

Lifting state up often changes the nature of what you're storing as state.

In this case, only one panel should be active at a time. This means that the `Accordion` common parent component needs to keep track of *which* panel is the active one. Instead of a `boolean` value, it could use a number as the index of the active `Panel` for the state variable:

```js
const [activeIndex, setActiveIndex] = useState(0);
```

When the `activeIndex` is `0`, the first panel is active, and when it's `1`, it's the second one.

Clicking the "Show" button in either `Panel` needs to change the active index in `Accordion`. A `Panel` can't set the `activeIndex` state directly because it's defined inside the `Accordion`. The `Accordion` component needs to *explicitly allow* the `Panel` component to change its state by [passing an event handler down as a prop](/learn/responding-to-events#passing-event-handlers-as-props):

```js
<>
<Panel
isActive={activeIndex === 0}
onShow={() => setActiveIndex(0)}
>
...
</Panel>
<Panel
isActive={activeIndex === 1}
onShow={() => setActiveIndex(1)}
>
...
</Panel>
</>
```

The `<button>` inside the `Panel` will now use the `onShow` prop as its click event handler:

<Sandpack>

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

export default function Accordion() {
```

```
const [activeIndex, setActiveIndex] = useState(0);
return (
<>
<h2>Almaty, Kazakhstan</h2>
<Panel
title="About"
isActive={activeIndex === 0}
onShow={() => setActiveIndex(0)}
>
With a population of about 2 million, Almaty is Kazakhstan's largest city. From 1929 to 1997, it was its capital city.
</Panel>
<Panel
title="Etymology"
isActive={activeIndex === 1}
onShow={() => setActiveIndex(1)}
>
The name comes from <span lang="kk-KZ">■■■■</span>, the Kazakh word for "apple" and is often translated as "full of apples". In fact, the region surrounding Almaty is thought to be the ancestral home of the apple, and the wild <i lang="la">Malus sieversii</i> is considered a likely candidate for the ancestor of the modern domestic apple.
</Panel>
</>
);
}

function Panel({
title,
children,
isActive,
onShow
}) {
return (
<section className="panel">
<h3>{title}</h3>
{isActive ? (
<p>{children}</p>
```

) : (

<button onClick={onShow}>

Show

</button>

)}

</section>

);

}

```

```css

h3, p { margin: 5px 0px; }

.panel {

padding: 10px;

border: 1px solid #aaa;

}

```

</Sandpack>

This completes lifting state up! Moving state into the common parent component allowed you to coordinate the two panels. Using the active index instead of two "is shown" flags ensured that only one panel is active at a given time. And passing down the event handler to the child allowed the child to change the parent's state.

<DiagramGroup>

<Diagram name="sharing_state_parent" height={385} width={487} alt="Diagram showing a tree of three components, one parent labeled Accordion and two children labeled Panel. Accordion contains an activeIndex value of zero which turns into isActive value of true passed to the first Panel, and isActive value of false passed to the second Panel." >

Initially, `Accordion`'s `activeIndex` is `0`, so the first `Panel` receives `isActive = true`

</Diagram>

<Diagram name="sharing_state_parent_clicked" height={385} width={521} alt="The same diagram as the previous, with the activeIndex value of the parent Accordion component highlighted indicating a click with the value changed to one. The flow to both of the children Panel components is also highlighted, and the isActive value passed to each child is set to the opposite: false for the first Panel and true for the second one." >

When `Accordion`'s `activeIndex` state changes to `1`, the second `Panel` receives `isActive = true` instead

</Diagram>
```

</DiagramGroup>

<DeepDive>

Controlled and uncontrolled components {/*controlled-and-uncontrolled-components*/}

It is common to call a component with some local state "uncontrolled". For example, the original `Panel` component with an `isActive` state variable is uncontrolled because its parent cannot influence whether the panel is active or not.

In contrast, you might say a component is "controlled" when the important information in it is driven by props rather than its own local state. This lets the parent component fully specify its behavior. The final `Panel` component with the `isActive` prop is controlled by the `Accordion` component.

Uncontrolled components are easier to use within their parents because they require less configuration. But they're less flexible when you want to coordinate them together. Controlled components are maximally flexible, but they require the parent components to fully configure them with props.

In practice, "controlled" and "uncontrolled" aren't strict technical terms--each component usually has some mix of both local state and props. However, this is a useful way to talk about how components are designed and what capabilities they offer.

When writing a component, consider which information in it should be controlled (via props), and which information should be uncontrolled (via state). But you can always change your mind and refactor later.

</DeepDive>

A single source of truth for each state {/*a-single-source-of-truth-for-each-state*/}

In a React application, many components will have their own state. Some state may "live" close to the leaf components (components at the bottom of the tree) like inputs. Other state may "live" closer to the top of the app. For example, even client-side routing libraries are usually implemented by storing the current route in the React state, and passing it down by props!

For each unique piece of state, you will choose the component that "owns" it. This principle is also known as having a ["single source of truth".](https://en.wikipedia.org/wiki/Single_source_of_truth) It doesn't mean that all state lives in one place--but that for _each_ piece of state, there is a _specific_ component that holds that piece of information. Instead of duplicating shared state between components, *lift it up* to their common shared parent, and *pass it down* to the children that need it.

Your app will change as you work on it. It is common that you will move state down or back up while you're still figuring out where each piece of the state "lives". This is all part of the process!

To see what this feels like in practice with a few more components, read [Thinking in React.](/learn/thinking-in-react)

<Recap>

* When you want to coordinate two components, move their state to their common parent.
* Then pass the information down through props from their common parent.
* Finally, pass the event handlers down so that the children can change the parent's state.

* It's useful to consider components as "controlled" (driven by props) or "uncontrolled" (driven by state).

</Recap>

<Challenges>

Synced inputs {/*synced-inputs*/}

These two inputs are independent. Make them stay in sync: editing one input should update the other input with the same text, and vice versa.

<Hint>

You'll need to lift their state up into the parent component.

</Hint>

<Sandpack>

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

export default function SyncedInputs() {
return (
<>
<Input label="First input" />
<Input label="Second input" />
</>
);
}

function Input({ label }) {
const [text, setText] = useState('');

function handleChange(e) {
setText(e.target.value);
}

return (
<label>
{label}
{' '}
<input
value={text}
```

onChange={handleChange}

/>

</label>

);

}

```

```css

input { margin: 5px; }

label { display: block; }

```

</Sandpack>

<Solution>

Move the `text` state variable into the parent component along with the `handleChange` handler. Then pass them down as props to both of the `Input` components. This will keep them in sync.

<Sandpack>

```js

import { useState } from 'react';

export default function SyncedInputs() {

const [text, setText] = useState('');

function handleChange(e) {

setText(e.target.value);

}

return (

<>

<Input

label="First input"

value={text}

onChange={handleChange}

/>

<Input

label="Second input"

value={text}

onChange={handleChange}

```
/>
</>
);
}

function Input({ label, value, onChange }) {
return (
<label>
{label}
{' '}
<input
value={value}
onChange={onChange}
/>
</label>
);
}
```

```css
input { margin: 5px; }
label { display: block; }
```

</Sandpack>

</Solution>

Filtering a list {/*filtering-a-list*/}

In this example, the `SearchBar` has its own `query` state that controls the text input. Its parent `FilterableList` component displays a `List` of items, but it doesn't take the search query into account.

Use the `filterItems(foods, query)` function to filter the list according to the search query. To test your changes, verify that typing "s" into the input filters down the list to "Sushi", "Shish kebab", and "Dim sum".

Note that `filterItems` is already implemented and imported so you don't need to write it yourself!

<Hint>

You will want to remove the `query` state and the `handleChange` handler from the `SearchBar`, and move them to the `FilterableList`. Then pass them down to `SearchBar` as `query` and `onChange` props.

</Hint>

<Sandpack>

```js
import { useState } from 'react';
import { foods, filterItems } from './data.js';

export default function FilterableList() {
return (
<>
<SearchBar />
<hr />
<List items={foods} />
</>
);
}

function SearchBar() {
const [query, setQuery] = useState('');

function handleChange(e) {
setQuery(e.target.value);
}

return (
<label>
Search:{' '}
<input
value={query}
onChange={handleChange}
/>
</label>
);
}

function List({ items }) {
return (
<table>
<tbody>
```

```
{items.map(food => (
<tr key={food.id}>
<td>{food.name}</td>
<td>{food.description}</td>
</tr>
))}
</tbody>
</table>
);
}
```

```js data.js
export function filterItems(items, query) {
query = query.toLowerCase();
return items.filter(item =>
item.name.split(' ').some(word =>
word.toLowerCase().startsWith(query)
)
);
}

export const foods = [{
id: 0,
name: 'Sushi',
description: 'Sushi is a traditional Japanese dish of prepared vinegared rice'
}, {
id: 1,
name: 'Dal',
description: 'The most common way of preparing dal is in the form of a soup to which onions, tomatoes and various spices may be added'
}, {
id: 2,
name: 'Pierogi',
description: 'Pierogi are filled dumplings made by wrapping unleavened dough around a savoury or sweet filling and cooking in boiling water'
}, {
```

id: 3,

name: 'Shish kebab',

description: 'Shish kebab is a popular meal of skewered and grilled cubes of meat.'

}, {

id: 4,

name: 'Dim sum',

description: 'Dim sum is a large range of small dishes that Cantonese people traditionally enjoy in restaurants for breakfast and lunch'

}];

```

</Sandpack>

<Solution>

Lift the `query` state up into the `FilterableList` component. Call `filterItems(foods, query)` to get the filtered list and pass it down to the `List`. Now changing the query input is reflected in the list:

<Sandpack>

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

import { foods, filterItems } from './data.js';

export default function FilterableList() {

const [query, setQuery] = useState('');

const results = filterItems(foods, query);

function handleChange(e) {

setQuery(e.target.value);

}

return (

<>

<SearchBar

query={query}

onChange={handleChange}

/>

<hr />

<List items={results} />

</>

```
);
}

function SearchBar({ query, onChange }) {
return (
<label>
Search:{' '}
<input
value={query}
onChange={onChange}
/>
</label>
);
}

function List({ items }) {
return (
<table>
<tbody>
{items.map(food => (
<tr key={food.id}>
<td>{food.name}</td>
<td>{food.description}</td>
</tr>
))}
</tbody>
</table>
);
}
```

```js data.js
export function filterItems(items, query) {
query = query.toLowerCase();
return items.filter(item =>
item.name.split(' ').some(word =>
word.toLowerCase().startsWith(query)
```

)

);

}

export const foods = [{

id: 0,

name: 'Sushi',

description: 'Sushi is a traditional Japanese dish of prepared vinegared rice'

}, {

id: 1,

name: 'Dal',

description: 'The most common way of preparing dal is in the form of a soup to which onions, tomatoes and various spices may be added'

}, {

id: 2,

name: 'Pierogi',

description: 'Pierogi are filled dumplings made by wrapping unleavened dough around a savoury or sweet filling and cooking in boiling water'

}, {

id: 3,

name: 'Shish kebab',

description: 'Shish kebab is a popular meal of skewered and grilled cubes of meat.'

}, {

id: 4,

name: 'Dim sum',

description: 'Dim sum is a large range of small dishes that Cantonese people traditionally enjoy in restaurants for breakfast and lunch'

}];

```

</Sandpack>

</Solution>

</Challenges>

---

title: Editor Setup

---

<Intro>
```

A properly configured editor can make code clearer to read and faster to write. It can even help you catch bugs as you write them! If this is your first time setting up an editor or you're looking to tune up your current editor, we have a few recommendations.

</Intro>

<YouWillLearn>

* What the most popular editors are
* How to format your code automatically

</YouWillLearn>

Your editor {/*your-editor*/}

[VS Code](https://code.visualstudio.com/) is one of the most popular editors in use today. It has a large marketplace of extensions and integrates well with popular services like GitHub. Most of the features listed below can be added to VS Code as extensions as well, making it highly configurable!

Other popular text editors used in the React community include:

* [WebStorm](https://www.jetbrains.com/webstorm/) is an integrated development environment designed specifically for JavaScript.
* [Sublime Text](https://www.sublimetext.com/) has support for JSX and TypeScript, [syntax highlighting](https://stackoverflow.com/a/70960574/458193) and autocomplete built in.
* [Vim](https://www.vim.org/) is a highly configurable text editor built to make creating and changing any kind of text very efficient. It is included as "vi" with most UNIX systems and with Apple OS X.

Recommended text editor features {/*recommended-text-editor-features*/}

Some editors come with these features built in, but others might require adding an extension. Check to see what support your editor of choice provides to be sure!

Linting {/*linting*/}

Code linters find problems in your code as you write, helping you fix them early. [ESLint](https://eslint.org/) is a popular, open source linter for JavaScript.

* [Install ESLint with the recommended configuration for React](https://www.npmjs.com/package/eslint-config-react-app) (be sure you have [Node installed!](https://nodejs.org/en/download/current/))
* [Integrate ESLint in VSCode with the official extension](https://marketplace.visualstudio.com/items?itemName=dbaeumer.vscode-eslint)

Make sure that you've enabled all the [`eslint-plugin-react-hooks`](https://www.npmjs.com/package/eslint-plugin-react-hooks) rules for your project. They are essential and catch the most severe bugs early. The recommended [`eslint-config-react-app`](https://www.npmjs.com/package/eslint-config-react-app) preset already includes them.

Formatting {/*formatting*/}

The last thing you want to do when sharing your code with another contributor is get into an discussion about [tabs vs spaces](https://www.google.com/search?q=tabs+vs+spaces)! Fortunately, [Prettier](https://prettier.io/) will clean up your code by reformatting it to conform to preset, configurable rules. Run Prettier, and all your tabs will be converted to spaces—and your indentation, quotes, etc will also all be changed to conform to the configuration. In the ideal setup, Prettier will run when you save your file, quickly making these edits for you.

You can install the [Prettier extension in VSCode](https://marketplace.visualstudio.com/items?itemName=esbenp.prettier-vscode) by following these steps:

1. Launch VS Code
2. Use Quick Open (press Ctrl/Cmd+P)
3. Paste in `ext install esbenp.prettier-vscode`
4. Press Enter

Formatting on save {/*formatting-on-save*/}

Ideally, you should format your code on every save. VS Code has settings for this!

1. In VS Code, press `CTRL/CMD + SHIFT + P`.
2. Type "settings"
3. Hit Enter
4. In the search bar, type "format on save"
5. Be sure the "format on save" option is ticked!

> If your ESLint preset has formatting rules, they may conflict with Prettier. We recommend disabling all formatting rules in your ESLint preset using [`eslint-config-prettier`](https://github.com/prettier/eslint-config-prettier) so that ESLint is *only* used for catching logical mistakes. If you want to enforce that files are formatted before a pull request is merged, use [`prettier --check`](https://prettier.io/docs/en/cli.html#--check) for your continuous integration.

---

title: Preserving and Resetting State

<Intro>

State is isolated between components. React keeps track of which state belongs to which component based on their place in the UI tree. You can control when to preserve state and when to reset it between re-renders.

</Intro>

<YouWillLearn>

* How React "sees" component structures
* When React chooses to preserve or reset the state

* How to force React to reset component's state

* How keys and types affect whether the state is preserved

</YouWillLearn>

The UI tree {/*the-ui-tree*/}

Browsers use many tree structures to model UI. The [DOM](https://developer.mozilla.org/docs/Web/API/Document_Object_Model/Introduction) represents HTML elements, the [CSSOM](https://developer.mozilla.org/docs/Web/API/CSS_Object_Model) does the same for CSS. There's even an [Accessibility tree](https://developer.mozilla.org/docs/Glossary/Accessibility_tree)!

React also uses tree structures to manage and model the UI you make. React makes **UI trees** from your JSX. Then React DOM updates the browser DOM elements to match that UI tree. (React Native translates these trees into elements specific to mobile platforms.)

<DiagramGroup>

<Diagram name="preserving_state_dom_tree" height={193} width={864} alt="Diagram with three sections arranged horizontally. In the first section, there are three rectangles stacked vertically, with labels 'Component A', 'Component B', and 'Component C'. Transitioning to the next pane is an arrow with the React logo on top labeled 'React'. The middle section contains a tree of components, with the root labeled 'A' and two children labeled 'B' and 'C'. The next section is again transitioned using an arrow with the React logo on top labeled 'React'. The third and final section is a wireframe of a browser, containing a tree of 8 nodes, which has only a subset highlighted (indicating the subtree from the middle section).">

From components, React creates a UI tree which React DOM uses to render the DOM

</Diagram>

</DiagramGroup>

State is tied to a position in the tree {/*state-is-tied-to-a-position-in-the-tree*/}

When you give a component state, you might think the state "lives" inside the component. But the state is actually held inside React. React associates each piece of state it's holding with the correct component by where that component sits in the UI tree.

Here, there is only one `<Counter />` JSX tag, but it's rendered at two different positions:

<Sandpack>

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

export default function App() {
const counter = <Counter />;
return (
```

```
<div>
{counter}
{counter}
</div>
);
}

function Counter() {
const [score, setScore] = useState(0);
const [hover, setHover] = useState(false);

let className = 'counter';
if (hover) {
className += ' hover';
}

return (
<div
className={className}
onPointerEnter={() => setHover(true)}
onPointerLeave={() => setHover(false)}
>
<h1>{score}</h1>
<button onClick={() => setScore(score + 1)}>
Add one
</button>
</div>
);
}
```

```css
label {
display: block;
clear: both;
}

.counter {
width: 100px;
```

text-align: center;

border: 1px solid gray;

border-radius: 4px;

padding: 20px;

margin: 0 20px 20px 0;

float: left;

}

.hover {

background: #ffffd8;

}

```

</Sandpack>

Here's how these look as a tree:

<DiagramGroup>

<Diagram name="preserving_state_tree" height={248} width={395} alt="Diagram of a tree of React components. The root node is labeled 'div' and has two children. Each of the children are labeled 'Counter' and both contain a state bubble labeled 'count' with value 0.">

React tree

</Diagram>

</DiagramGroup>

**These are two separate counters because each is rendered at its own position in the tree.** You don't usually have to think about these positions to use React, but it can be useful to understand how it works.

In React, each component on the screen has fully isolated state. For example, if you render two `Counter` components side by side, each of them will get its own, independent, `score` and `hover` states.

Try clicking both counters and notice they don't affect each other:

<Sandpack>

```js

import { useState } from 'react';

export default function App() {

return (

```
<Counter />
<Counter />
</div>
);
}

function Counter() {
const [score, setScore] = useState(0);
const [hover, setHover] = useState(false);

let className = 'counter';
if (hover) {
className += ' hover';
}

return (
<div
className={className}
onPointerEnter={() => setHover(true)}
onPointerLeave={() => setHover(false)}
>
<h1>{score}</h1>
<button onClick={() => setScore(score + 1)}>
Add one
</button>
</div>
);
}
```

```css
.counter {
width: 100px;
text-align: center;
border: 1px solid gray;
border-radius: 4px;
padding: 20px;
margin: 0 20px 20px 0;
```

```
float: left;
}
.hover {
background: #ffffd8;
}
```

</Sandpack>

As you can see, when one counter is updated, only the state for that component is updated:

<DiagramGroup>

<Diagram name="preserving_state_increment" height={248} width={441} alt="Diagram of a tree of React components. The root node is labeled 'div' and has two children. The left child is labeled 'Counter' and contains a state bubble labeled 'count' with value 0. The right child is labeled 'Counter' and contains a state bubble labeled 'count' with value 1. The state bubble of the right child is highlighted in yellow to indicate its value has updated.">

Updating state

</Diagram>

</DiagramGroup>

React will keep the state around for as long as you render the same component at the same position. To see this, increment both counters, then remove the second component by unchecking "Render the second counter" checkbox, and then add it back by ticking it again:

<Sandpack>

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

export default function App() {
const [showB, setShowB] = useState(true);
return (
<div>
<Counter />
{showB && <Counter />}
<label>
<input
type="checkbox"
checked={showB}
```

```
onChange={e => {
setShowB(e.target.checked)
}}
/>
Render the second counter
</label>
</div>
);
}

function Counter() {
const [score, setScore] = useState(0);
const [hover, setHover] = useState(false);

let className = 'counter';
if (hover) {
className += ' hover';
}

return (
<div
className={className}
onPointerEnter={() => setHover(true)}
onPointerLeave={() => setHover(false)}
>
<h1>{score}</h1>
<button onClick={() => setScore(score + 1)}>
Add one
</button>
</div>
);
}
```

```css
label {
display: block;
clear: both;
```

```
}

.counter {
width: 100px;
text-align: center;
border: 1px solid gray;
border-radius: 4px;
padding: 20px;
margin: 0 20px 20px 0;
float: left;
}

.hover {
background: #ffffd8;
}
```

</Sandpack>

Notice how the moment you stop rendering the second counter, its state disappears completely. That's because when React removes a component, it destroys its state.

<DiagramGroup>

<Diagram name="preserving_state_remove_component" height={253} width={422} alt="Diagram of a tree of React components. The root node is labeled 'div' and has two children. The left child is labeled 'Counter' and contains a state bubble labeled 'count' with value 0. The right child is missing, and in its place is a yellow 'poof' image, highlighting the component being deleted from the tree.">

Deleting a component

</Diagram>

</DiagramGroup>

When you tick "Render the second counter", a second `Counter` and its state are initialized from scratch (`score = 0`) and added to the DOM.

<DiagramGroup>

<Diagram name="preserving_state_add_component" height={258} width={500} alt="Diagram of a tree of React components. The root node is labeled 'div' and has two children. The left child is labeled 'Counter' and contains a state bubble labeled 'count' with value 0. The right child is labeled 'Counter' and contains a state bubble labeled 'count' with value 0. The entire right child node is highlighted in yellow, indicating that it was just added to the tree.">

Adding a component

</Diagram>

</DiagramGroup>

React preserves a component's state for as long as it's being rendered at its position in the UI tree. If it gets removed, or a different component gets rendered at the same position, React discards its state.

Same component at the same position preserves state
{/*same-component-at-the-same-position-preserves-state*/}

In this example, there are two different `<Counter />` tags:

<Sandpack>

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

export default function App() {
const [isFancy, setIsFancy] = useState(false);
return (
<div>
{isFancy ? (
<Counter isFancy={true} />
) : (
<Counter isFancy={false} />
)}
<label>
<input
type="checkbox"
checked={isFancy}
onChange={e => {
setIsFancy(e.target.checked)
}}
/>
Use fancy styling
</label>
</div>
);
}

function Counter({ isFancy }) {
```

```
const [score, setScore] = useState(0);
const [hover, setHover] = useState(false);

let className = 'counter';
if (hover) {
className += ' hover';
}
if (isFancy) {
className += ' fancy';
}

return (
<div
className={className}
onPointerEnter={() => setHover(true)}
onPointerLeave={() => setHover(false)}
>
<h1>{score}</h1>
<button onClick={() => setScore(score + 1)}>
Add one
</button>
</div>
);
}
```

```css
label {
display: block;
clear: both;
}

.counter {
width: 100px;
text-align: center;
border: 1px solid gray;
border-radius: 4px;
padding: 20px;
```

margin: 0 20px 20px 0;

float: left;

}

.fancy {

border: 5px solid gold;

color: #ff6767;

}

.hover {

background: #ffffd8;

}

```

</Sandpack>

When you tick or clear the checkbox, the counter state does not get reset. Whether `isFancy` is `true` or `false`, you always have a `<Counter />` as the first child of the `div` returned from the root `App` component:

<DiagramGroup>

<Diagram name="preserving_state_same_component" height={461} width={600} alt="Diagram with two sections separated by an arrow transitioning between them. Each section contains a layout of components with a parent labeled 'App' containing a state bubble labeled isFancy. This component has one child labeled 'div', which leads to a prop bubble containing isFancy (highlighted in purple) passed down to the only child. The last child is labeled 'Counter' and contains a state bubble with label 'count' and value 3 in both diagrams. In the left section of the diagram, nothing is highlighted and the isFancy parent state value is false. In the right section of the diagram, the isFancy parent state value has changed to true and it is highlighted in yellow, and so is the props bubble below, which has also changed its isFancy value to true.">

Updating the `App` state does not reset the `Counter` because `Counter` stays in the same position

</Diagram>

</DiagramGroup>

It's the same component at the same position, so from React's perspective, it's the same counter.

<Pitfall>

Remember that **it's the position in the UI tree--not in the JSX markup--that matters to React!** This component has two `return` clauses with different `<Counter />` JSX tags inside and outside the `if`:

<Sandpack>

```js

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

export default function App() {
const [isFancy, setIsFancy] = useState(false);
if (isFancy) {
return (
<div>
<Counter isFancy={true} />
<label>
<input
type="checkbox"
checked={isFancy}
onChange={e => {
setIsFancy(e.target.checked)
}}
/>
Use fancy styling
</label>
</div>
);
}
return (
<div>
<Counter isFancy={false} />
<label>
<input
type="checkbox"
checked={isFancy}
onChange={e => {
setIsFancy(e.target.checked)
}}
/>
Use fancy styling
</label>
</div>
);
```

```
}

function Counter({ isFancy }) {
const [score, setScore] = useState(0);
const [hover, setHover] = useState(false);

let className = 'counter';
if (hover) {
className += ' hover';
}
if (isFancy) {
className += ' fancy';
}

return (
<div
className={className}
onPointerEnter={() => setHover(true)}
onPointerLeave={() => setHover(false)}
>
<h1>{score}</h1>
<button onClick={() => setScore(score + 1)}>
Add one
</button>
</div>
);
}
```

```css
label {
display: block;
clear: both;
}

.counter {
width: 100px;
text-align: center;
border: 1px solid gray;
```

```
border-radius: 4px;
padding: 20px;
margin: 0 20px 20px 0;
float: left;
}

.fancy {
border: 5px solid gold;
color: #ff6767;
}

.hover {
background: #ffffd8;
}
```

</Sandpack>

You might expect the state to reset when you tick checkbox, but it doesn't! This is because **both of these `<Counter />` tags are rendered at the same position.** React doesn't know where you place the conditions in your function. All it "sees" is the tree you return.

In both cases, the `App` component returns a `<div>` with `<Counter />` as a first child. To React, these two counters have the same "address": the first child of the first child of the root. This is how React matches them up between the previous and next renders, regardless of how you structure your logic.

</Pitfall>

Different components at the same position reset state
{/*different-components-at-the-same-position-reset-state*/}

In this example, ticking the checkbox will replace `<Counter>` with a `<p>`:

<Sandpack>

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

export default function App() {
const [isPaused, setIsPaused] = useState(false);
return (
<div>
{isPaused ? (
<p>See you later!</p>
```

```
) : (
<Counter />
)}
<label>
<input
type="checkbox"
checked={isPaused}
onChange={e => {
setIsPaused(e.target.checked)
}}
/>
Take a break
</label>
</div>
);
}

function Counter() {
const [score, setScore] = useState(0);
const [hover, setHover] = useState(false);

let className = 'counter';
if (hover) {
className += ' hover';
}

return (
<div
className={className}
onPointerEnter={() => setHover(true)}
onPointerLeave={() => setHover(false)}
>
<h1>{score}</h1>
<button onClick={() => setScore(score + 1)}>
Add one
</button>
</div>
```

);
}
```

```css
label {
display: block;
clear: both;
}

.counter {
width: 100px;
text-align: center;
border: 1px solid gray;
border-radius: 4px;
padding: 20px;
margin: 0 20px 20px 0;
float: left;
}

.hover {
background: #ffffd8;
}
```

</Sandpack>

Here, you switch between _different_ component types at the same position. Initially, the first child of the `<div>` contained a `Counter`. But when you swapped in a `p`, React removed the `Counter` from the UI tree and destroyed its state.

<DiagramGroup>

<Diagram name="preserving_state_diff_pt1" height={290} width={753} alt="Diagram with three sections, with an arrow transitioning each section in between. The first section contains a React component labeled 'div' with a single child labeled 'Counter' containing a state bubble labeled 'count' with value 3. The middle section has the same 'div' parent, but the child component has now been deleted, indicated by a yellow 'proof' image. The third section has the same 'div' parent again, now with a new child labeled 'p', highlighted in yellow.">

When `Counter` changes to `p`, the `Counter` is deleted and the `p` is added

</Diagram>
```

</DiagramGroup>

<DiagramGroup>

<Diagram name="preserving_state_diff_pt2" height={290} width={753} alt="Diagram with three sections, with an arrow transitioning each section in between. The first section contains a React component labeled 'p'. The middle section has the same 'div' parent, but the child component has now been deleted, indicated by a yellow 'proof' image. The third section has the same 'div' parent again, now with a new child labeled 'Counter' containing a state bubble labeled 'count' with value 0, highlighted in yellow.">

When switching back, the `p` is deleted and the `Counter` is added

</Diagram>

</DiagramGroup>

Also, **when you render a different component in the same position, it resets the state of its entire subtree.** To see how this works, increment the counter and then tick the checkbox:

<Sandpack>

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

export default function App() {
const [isFancy, setIsFancy] = useState(false);
return (
<div>
{isFancy ? (
<div>
<Counter isFancy={true} />
</div>
) : (
<section>
<Counter isFancy={false} />
</section>
)}
<label>
<input
type="checkbox"
checked={isFancy}
onChange={e => {
```

````
setIsFancy(e.target.checked)
}}
/>
Use fancy styling
</label>
</div>
);
}

function Counter({ isFancy }) {
const [score, setScore] = useState(0);
const [hover, setHover] = useState(false);

let className = 'counter';
if (hover) {
className += ' hover';
}
if (isFancy) {
className += ' fancy';
}

return (
<div
className={className}
onPointerEnter={() => setHover(true)}
onPointerLeave={() => setHover(false)}
>
<h1>{score}</h1>
<button onClick={() => setScore(score + 1)}>
Add one
</button>
</div>
);
}
```

```css
label {
````

```
display: block;
clear: both;
}

.counter {
width: 100px;
text-align: center;
border: 1px solid gray;
border-radius: 4px;
padding: 20px;
margin: 0 20px 20px 0;
float: left;
}

.fancy {
border: 5px solid gold;
color: #ff6767;
}

.hover {
background: #ffffd8;
}
```

</Sandpack>

The counter state gets reset when you click the checkbox. Although you render a `Counter`, the first child of the `div` changes from a `div` to a `section`. When the child `div` was removed from the DOM, the whole tree below it (including the `Counter` and its state) was destroyed as well.

<DiagramGroup>

<Diagram name="preserving_state_diff_same_pt1" height={350} width={794} alt="Diagram with three sections, with an arrow transitioning each section in between. The first section contains a React component labeled 'div' with a single child labeled 'section', which has a single child labeled 'Counter' containing a state bubble labeled 'count' with value 3. The middle section has the same 'div' parent, but the child components have now been deleted, indicated by a yellow 'proof' image. The third section has the same 'div' parent again, now with a new child labeled 'div', highlighted in yellow, also with a new child labeled 'Counter' containing a state bubble labeled 'count' with value 0, all highlighted in yellow.">

When `section` changes to `div`, the `section` is deleted and the new `div` is added

</Diagram>

</DiagramGroup>

<DiagramGroup>

<Diagram name="preserving_state_diff_same_pt2" height={350} width={794} alt="Diagram with three sections, with an arrow transitioning each section in between. The first section contains a React component labeled 'div' with a single child labeled 'div', which has a single child labeled 'Counter' containing a state bubble labeled 'count' with value 0. The middle section has the same 'div' parent, but the child components have now been deleted, indicated by a yellow 'proof' image. The third section has the same 'div' parent again, now with a new child labeled 'section', highlighted in yellow, also with a new child labeled 'Counter' containing a state bubble labeled 'count' with value 0, all highlighted in yellow.">

When switching back, the `div` is deleted and the new `section` is added

</Diagram>

</DiagramGroup>

As a rule of thumb, **if you want to preserve the state between re-renders, the structure of your tree needs to "match up"** from one render to another. If the structure is different, the state gets destroyed because React destroys state when it removes a component from the tree.

<Pitfall>

This is why you should not nest component function definitions.

Here, the `MyTextField` component function is defined *inside* `MyComponent`:

<Sandpack>

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

export default function MyComponent() {
const [counter, setCounter] = useState(0);

function MyTextField() {
const [text, setText] = useState('');

return (
<input
value={text}
onChange={e => setText(e.target.value)}
/>
);
}

return (
<>
```

<MyTextField />

<button onClick={() => {

setCounter(counter + 1)

}}>Clicked {counter} times</button>

</>

);

}

```

</Sandpack>

Every time you click the button, the input state disappears! This is because a *different* `MyTextField` function is created for every render of `MyComponent`. You're rendering a *different* component in the same position, so React resets all state below. This leads to bugs and performance problems. To avoid this problem, **always declare component functions at the top level, and don't nest their definitions.**

</Pitfall>

## Resetting state at the same position {/*resetting-state-at-the-same-position*/}

By default, React preserves state of a component while it stays at the same position. Usually, this is exactly what you want, so it makes sense as the default behavior. But sometimes, you may want to reset a component's state. Consider this app that lets two players keep track of their scores during each turn:

<Sandpack>

```js

import { useState } from 'react';

export default function Scoreboard() {

const [isPlayerA, setIsPlayerA] = useState(true);

return (

{isPlayerA ? (

<Counter person="Taylor" />

) : (

<Counter person="Sarah" />

)}

<button onClick={() => {

setIsPlayerA(!isPlayerA);

}}>

```
Next player!
</button>
</div>
);
}

function Counter({ person }) {
const [score, setScore] = useState(0);
const [hover, setHover] = useState(false);

let className = 'counter';
if (hover) {
className += ' hover';
}

return (
<div
className={className}
onPointerEnter={() => setHover(true)}
onPointerLeave={() => setHover(false)}
>
<h1>{person}'s score: {score}</h1>
<button onClick={() => setScore(score + 1)}>
Add one
</button>
</div>
);
}
```

```css
h1 {
font-size: 18px;
}

.counter {
width: 100px;
text-align: center;
border: 1px solid gray;
```

border-radius: 4px;

padding: 20px;

margin: 0 20px 20px 0;

}

.hover {

background: #ffffd8;

}

```

</Sandpack>

Currently, when you change the player, the score is preserved. The two `Counter`s appear in the same position, so React sees them as *the same* `Counter` whose `person` prop has changed.

But conceptually, in this app they should be two separate counters. They might appear in the same place in the UI, but one is a counter for Taylor, and another is a counter for Sarah.

There are two ways to reset state when switching between them:

1. Render components in different positions

2. Give each component an explicit identity with `key`

### Option 1: Rendering a component in different positions
{/*option-1-rendering-a-component-in-different-positions*/}

If you want these two `Counter`s to be independent, you can render them in two different positions:

<Sandpack>

```js

import { useState } from 'react';

export default function Scoreboard() {

const [isPlayerA, setIsPlayerA] = useState(true);

return (

{isPlayerA &&

<Counter person="Taylor" />

}

{!isPlayerA &&

<Counter person="Sarah" />

}

```
<button onClick={() => {
setIsPlayerA(!isPlayerA);
}}>
Next player!
</button>
</div>
);
}

function Counter({ person }) {
const [score, setScore] = useState(0);
const [hover, setHover] = useState(false);

let className = 'counter';
if (hover) {
className += ' hover';
}

return (
<div
className={className}
onPointerEnter={() => setHover(true)}
onPointerLeave={() => setHover(false)}
>
<h1>{person}'s score: {score}</h1>
<button onClick={() => setScore(score + 1)}>
Add one
</button>
</div>
);
}
```

```css
h1 {
font-size: 18px;
}

.counter {
```

```
width: 100px;
text-align: center;
border: 1px solid gray;
border-radius: 4px;
padding: 20px;
margin: 0 20px 20px 0;
}

.hover {
background: #ffffd8;
}
```

</Sandpack>

* Initially, `isPlayerA` is `true`. So the first position contains `Counter` state, and the second one is empty.

* When you click the "Next player" button the first position clears but the second one now contains a `Counter`.

<DiagramGroup>

<Diagram name="preserving_state_diff_position_p1" height={375} width={504} alt="Diagram with a tree of React components. The parent is labeled 'Scoreboard' with a state bubble labeled isPlayerA with value 'true'. The only child, arranged to the left, is labeled Counter with a state bubble labeled 'count' and value 0. All of the left child is highlighted in yellow, indicating it was added.">

Initial state

</Diagram>

<Diagram name="preserving_state_diff_position_p2" height={375} width={504} alt="Diagram with a tree of React components. The parent is labeled 'Scoreboard' with a state bubble labeled isPlayerA with value 'false'. The state bubble is highlighted in yellow, indicating that it has changed. The left child is replaced with a yellow 'poof' image indicating that it has been deleted and there is a new child on the right, highlighted in yellow indicating that it was added. The new child is labeled 'Counter' and contains a state bubble labeled 'count' with value 0.">

Clicking "next"

</Diagram>

<Diagram name="preserving_state_diff_position_p3" height={375} width={504} alt="Diagram with a tree of React components. The parent is labeled 'Scoreboard' with a state bubble labeled isPlayerA with value 'true'. The state bubble is highlighted in yellow, indicating that it has changed. There is a new child on the left, highlighted in yellow indicating that it was added. The new child is labeled 'Counter' and contains a state bubble labeled 'count' with value 0. The right child is replaced with a yellow 'poof' image indicating that it has been deleted.">

Clicking "next" again

</Diagram>

</DiagramGroup>

Each `Counter`'s state gets destroyed each time its removed from the DOM. This is why they reset every time you click the button.

This solution is convenient when you only have a few independent components rendered in the same place. In this example, you only have two, so it's not a hassle to render both separately in the JSX.

Option 2: Resetting state with a key {/*option-2-resetting-state-with-a-key*/}

There is also another, more generic, way to reset a component's state.

You might have seen `key`s when [rendering lists.](/learn/rendering-lists#keeping-list-items-in-order-with-key) Keys aren't just for lists! You can use keys to make React distinguish between any components. By default, React uses order within the parent ("first counter", "second counter") to discern between components. But keys let you tell React that this is not just a *first* counter, or a *second* counter, but a specific counter--for example, *Taylor's* counter. This way, React will know *Taylor's* counter wherever it appears in the tree!

In this example, the two `<Counter />`s don't share state even though they appear in the same place in JSX:

<Sandpack>

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

export default function Scoreboard() {
const [isPlayerA, setIsPlayerA] = useState(true);
return (
<div>
{isPlayerA ? (
<Counter key="Taylor" person="Taylor" />
) : (
<Counter key="Sarah" person="Sarah" />
)}
<button onClick={() => {
setIsPlayerA(!isPlayerA);
}}>
Next player!
</button>
```

```
</div>
);
}

function Counter({ person }) {
const [score, setScore] = useState(0);
const [hover, setHover] = useState(false);

let className = 'counter';
if (hover) {
className += ' hover';
}

return (
<div
className={className}
onPointerEnter={() => setHover(true)}
onPointerLeave={() => setHover(false)}
>
<h1>{person}'s score: {score}</h1>
<button onClick={() => setScore(score + 1)}>
Add one
</button>
</div>
);
}
```

```css
h1 {
font-size: 18px;
}

.counter {
width: 100px;
text-align: center;
border: 1px solid gray;
border-radius: 4px;
padding: 20px;
```

margin: 0 20px 20px 0;

}

.hover {

background: #ffffd8;

}

```

</Sandpack>

Switching between Taylor and Sarah does not preserve the state. This is because **you gave them different `key`s:**

```js

{isPlayerA ? (

<Counter key="Taylor" person="Taylor" />

) : (

<Counter key="Sarah" person="Sarah" />

)}

```

Specifying a `key` tells React to use the `key` itself as part of the position, instead of their order within the parent. This is why, even though you render them in the same place in JSX, React sees them as two different counters, and so they will never share state. Every time a counter appears on the screen, its state is created. Every time it is removed, its state is destroyed. Toggling between them resets their state over and over.

<Note>

Remember that keys are not globally unique. They only specify the position *within the parent*.

</Note>

Resetting a form with a key {/*resetting-a-form-with-a-key*/}

Resetting state with a key is particularly useful when dealing with forms.

In this chat app, the `<Chat>` component contains the text input state:

<Sandpack>

```js App.js

import { useState } from 'react';

import Chat from './Chat.js';

import ContactList from './ContactList.js';

```
export default function Messenger() {
const [to, setTo] = useState(contacts[0]);
return (
<div>
<ContactList
contacts={contacts}
selectedContact={to}
onSelect={contact => setTo(contact)}
/>
<Chat contact={to} />
</div>
)
}

const contacts = [
{ id: 0, name: 'Taylor', email: 'taylor@mail.com' },
{ id: 1, name: 'Alice', email: 'alice@mail.com' },
{ id: 2, name: 'Bob', email: 'bob@mail.com' }
];
```

```js ContactList.js
export default function ContactList({
selectedContact,
contacts,
onSelect
}) {
return (
<section className="contact-list">
<ul>
{contacts.map(contact =>
<li key={contact.id}>
<button onClick={() => {
onSelect(contact);
}}>
{contact.name}
</button>
```

```
</li>
)}
</ul>
</section>
);
}
```

```js Chat.js
import { useState } from 'react';

export default function Chat({ contact }) {
const [text, setText] = useState('');
return (
<section className="chat">
<textarea
value={text}
placeholder={'Chat to ' + contact.name}
onChange={e => setText(e.target.value)}
/>
<br />
<button>Send to {contact.email}</button>
</section>
);
}
```

```css
.chat, .contact-list {
float: left;
margin-bottom: 20px;
}
ul, li {
list-style: none;
margin: 0;
padding: 0;
}
```

li button {

width: 100px;

padding: 10px;

margin-right: 10px;

}

textarea {

height: 150px;

}

```

</Sandpack>

Try entering something into the input, and then press "Alice" or "Bob" to choose a different recipient. You will notice that the input state is preserved because the `<Chat>` is rendered at the same position in the tree.

**In many apps, this may be the desired behavior, but not in a chat app!** You don't want to let the user send a message they already typed to a wrong person due to an accidental click. To fix it, add a `key`:

```js

<Chat key={to.id} contact={to} />

```

This ensures that when you select a different recipient, the `Chat` component will be recreated from scratch, including any state in the tree below it. React will also re-create the DOM elements instead of reusing them.

Now switching the recipient always clears the text field:

<Sandpack>

```js App.js

import { useState } from 'react';

import Chat from './Chat.js';

import ContactList from './ContactList.js';

export default function Messenger() {

const [to, setTo] = useState(contacts[0]);

return (

<ContactList

contacts={contacts}

selectedContact={to}

```
onSelect={contact => setTo(contact)}
/>
<Chat key={to.id} contact={to} />
</div>
)
}

const contacts = [
{ id: 0, name: 'Taylor', email: 'taylor@mail.com' },
{ id: 1, name: 'Alice', email: 'alice@mail.com' },
{ id: 2, name: 'Bob', email: 'bob@mail.com' }
];
```

```js ContactList.js
export default function ContactList({
selectedContact,
contacts,
onSelect
}) {
return (
<section className="contact-list">
<ul>
{contacts.map(contact =>
<li key={contact.id}>
<button onClick={() => {
onSelect(contact);
}}>
{contact.name}
</button>
</li>
)}
</ul>
</section>
);
}
```

```js Chat.js
import { useState } from 'react';

export default function Chat({ contact }) {
const [text, setText] = useState('');
return (
<section className="chat">
<textarea
value={text}
placeholder={'Chat to ' + contact.name}
onChange={e => setText(e.target.value)}
/>
<br />
<button>Send to {contact.email}</button>
</section>
);
}
```

```css
.chat, .contact-list {
float: left;
margin-bottom: 20px;
}
ul, li {
list-style: none;
margin: 0;
padding: 0;
}
li button {
width: 100px;
padding: 10px;
margin-right: 10px;
}
textarea {
height: 150px;
}
```

```

</Sandpack>

<DeepDive>

#### Preserving state for removed components {/*preserving-state-for-removed-components*/}

In a real chat app, you'd probably want to recover the input state when the user selects the previous recipient again. There are a few ways to keep the state "alive" for a component that's no longer visible:

- You could render _all_ chats instead of just the current one, but hide all the others with CSS. The chats would not get removed from the tree, so their local state would be preserved. This solution works great for simple UIs. But it can get very slow if the hidden trees are large and contain a lot of DOM nodes.
- You could [lift the state up](/learn/sharing-state-between-components) and hold the pending message for each recipient in the parent component. This way, when the child components get removed, it doesn't matter, because it's the parent that keeps the important information. This is the most common solution.
- You might also use a different source in addition to React state. For example, you probably want a message draft to persist even if the user accidentally closes the page. To implement this, you could have the `Chat` component initialize its state by reading from the [`localStorage`](https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage), and save the drafts there too.

No matter which strategy you pick, a chat _with Alice_ is conceptually distinct from a chat _with Bob_, so it makes sense to give a `key` to the `<Chat>` tree based on the current recipient.

</DeepDive>

<Recap>

- React keeps state for as long as the same component is rendered at the same position.
- State is not kept in JSX tags. It's associated with the tree position in which you put that JSX.
- You can force a subtree to reset its state by giving it a different key.
- Don't nest component definitions, or you'll reset state by accident.

</Recap>

<Challenges>

#### Fix disappearing input text {/*fix-disappearing-input-text*/}

This example shows a message when you press the button. However, pressing the button also accidentally resets the input. Why does this happen? Fix it so that pressing the button does not reset the input text.

<Sandpack>
```

```js App.js
import { useState } from 'react';

export default function App() {
const [showHint, setShowHint] = useState(false);
if (showHint) {
return (
<div>
<p><i>Hint: Your favorite city?</i></p>
<Form />
<button onClick={() => {
setShowHint(false);
}}>Hide hint</button>
</div>
);
}
return (
<div>
<Form />
<button onClick={() => {
setShowHint(true);
}}>Show hint</button>
</div>
);
}

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

```css
textarea { display: block; margin: 10px 0; }
```

</Sandpack>

<Solution>

The problem is that `Form` is rendered in different positions. In the `if` branch, it is the second child of the `<div>`, but in the `else` branch, it is the first child. Therefore, the component type in each position changes. The first position changes between holding a `p` and a `Form`, while the second position changes between holding a `Form` and a `button`. React resets the state every time the component type changes.

The easiest solution is to unify the branches so that `Form` always renders in the same position:

<Sandpack>

```js App.js
import { useState } from 'react';

export default function App() {
const [showHint, setShowHint] = useState(false);
return (
<div>
{showHint &&
<p><i>Hint: Your favorite city?</i></p>
}
<Form />
{showHint ? (
<button onClick={() => {
setShowHint(false);
}}>Hide hint</button>
) : (
<button onClick={() => {
setShowHint(true);
}}>Show hint</button>
)}
</div>
);
}
```

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

```css
textarea { display: block; margin: 10px 0; }
```

</Sandpack>

Technically, you could also add `null` before `<Form />` in the `else` branch to match the `if` branch structure:

<Sandpack>

```js App.js
import { useState } from 'react';

export default function App() {
const [showHint, setShowHint] = useState(false);
if (showHint) {
return (

Hint: Your favorite city?
<Form />
<button onClick={() => {
setShowHint(false);
}}>Hide hint</button>

);
}
return (

{null}

<Form />

<button onClick={() => {

setShowHint(true);

}}>Show hint</button>

);

}

function Form() {

const [text, setText] = useState('');

return (

<textarea

value={text}

onChange={e => setText(e.target.value)}

/>

);

}

```

```css

textarea { display: block; margin: 10px 0; }

```

</Sandpack>

This way, `Form` is always the second child, so it stays in the same position and keeps its state. But this approach is much less obvious and introduces a risk that someone else will remove that `null`.

</Solution>

Swap two form fields {/*swap-two-form-fields*/}

This form lets you enter first and last name. It also has a checkbox controlling which field goes first. When you tick the checkbox, the "Last name" field will appear before the "First name" field.

It almost works, but there is a bug. If you fill in the "First name" input and tick the checkbox, the text will stay in the first input (which is now "Last name"). Fix it so that the input text *also* moves when you reverse the order.

<Hint>
```

It seems like for these fields, their position within the parent is not enough. Is there some way to tell React how to match up the state between re-renders?

</Hint>

<Sandpack>

```js App.js
import { useState } from 'react';

export default function App() {
const [reverse, setReverse] = useState(false);
let checkbox = (
<label>
<input
type="checkbox"
checked={reverse}
onChange={e => setReverse(e.target.checked)}
/>
Reverse order
</label>
);
if (reverse) {
return (
<>
<Field label="Last name" />
<Field label="First name" />
{checkbox}
</>
);
} else {
return (
<>
<Field label="First name" />
<Field label="Last name" />
{checkbox}
</>
);
```

}

}

function Field({ label }) {

const [text, setText] = useState('');

return (

<label>

{label}:{' '}

<input

type="text"

value={text}

placeholder={label}

onChange={e => setText(e.target.value)}

/>

</label>

);

}

```

```css

label { display: block; margin: 10px 0; }

```

</Sandpack>

<Solution>

Give a `key` to both `<Field>` components in both `if` and `else` branches. This tells React how to "match up" the correct state for either `<Field>` even if their order within the parent changes:

<Sandpack>

```js App.js

import { useState } from 'react';

export default function App() {

const [reverse, setReverse] = useState(false);

let checkbox = (

<label>

<input

type="checkbox"

```
checked={reverse}
onChange={e => setReverse(e.target.checked)}
/>
Reverse order
</label>
);
if (reverse) {
return (
<>
<Field key="lastName" label="Last name" />
<Field key="firstName" label="First name" />
{checkbox}
</>
);
} else {
return (
<>
<Field key="firstName" label="First name" />
<Field key="lastName" label="Last name" />
{checkbox}
</>
);
}
}

function Field({ label }) {
const [text, setText] = useState('');
return (
<label>
{label}:{' '}
<input
type="text"
value={text}
placeholder={label}
onChange={e => setText(e.target.value)}
/>
```

</label>

);

}

```

```css

label { display: block; margin: 10px 0; }

```

</Sandpack>

</Solution>

Reset a detail form {/*reset-a-detail-form*/}

This is an editable contact list. You can edit the selected contact's details and then either press "Save" to update it, or "Reset" to undo your changes.

When you select a different contact (for example, Alice), the state updates but the form keeps showing the previous contact's details. Fix it so that the form gets reset when the selected contact changes.

<Sandpack>

```js App.js

import { useState } from 'react';

import ContactList from './ContactList.js';

import EditContact from './EditContact.js';

export default function ContactManager() {

const [

contacts,

setContacts

] = useState(initialContacts);

const [

selectedId,

setSelectedId

] = useState(0);

const selectedContact = contacts.find(c =>

c.id === selectedId

);

function handleSave(updatedData) {

const nextContacts = contacts.map(c => {

```
if (c.id === updatedData.id) {
return updatedData;
} else {
return c;
}
});
setContacts(nextContacts);
}

return (
<div>
<ContactList
contacts={contacts}
selectedId={selectedId}
onSelect={id => setSelectedId(id)}
/>
<hr />
<EditContact
initialData={selectedContact}
onSave={handleSave}
/>
</div>
)
}

const initialContacts = [
{ id: 0, name: 'Taylor', email: 'taylor@mail.com' },
{ id: 1, name: 'Alice', email: 'alice@mail.com' },
{ id: 2, name: 'Bob', email: 'bob@mail.com' }
];
```

```js ContactList.js
export default function ContactList({
contacts,
selectedId,
onSelect
```

```
}) {
return (
<section>
<ul>
{contacts.map(contact =>
<li key={contact.id}>
<button onClick={() => {
onSelect(contact.id);
}}>
{contact.id === selectedId ?
<b>{contact.name}</b> :
contact.name
}
</button>
</li>
)}
</ul>
</section>
);
}
```

```js EditContact.js
import { useState } from 'react';

export default function EditContact({ initialData, onSave }) {
const [name, setName] = useState(initialData.name);
const [email, setEmail] = useState(initialData.email);
return (
<section>
<label>
Name:{' '}
<input
type="text"
value={name}
onChange={e => setName(e.target.value)}
/>
```

</label>
<label>
Email:{' '}
<input
type="email"
value={email}
onChange={e => setEmail(e.target.value)}
/>
</label>
<button onClick={() => {
const updatedData = {
id: initialData.id,
name: name,
email: email
};
onSave(updatedData);
}}>
Save
</button>
<button onClick={() => {
setName(initialData.name);
setEmail(initialData.email);
}}>
Reset
</button>
</section>
);
}
```

```css
ul, li {
list-style: none;
margin: 0;
padding: 0;
}

li { display: inline-block; }

li button {

padding: 10px;

}

label {

display: block;

margin: 10px 0;

}

button {

margin-right: 10px;

margin-bottom: 10px;

}

```

</Sandpack>

<Solution>

Give `key={selectedId}` to the `EditContact` component. This way, switching between different contacts will reset the form:

<Sandpack>

```js App.js

import { useState } from 'react';

import ContactList from './ContactList.js';

import EditContact from './EditContact.js';

export default function ContactManager() {

const [

contacts,

setContacts

] = useState(initialContacts);

const [

selectedId,

setSelectedId

] = useState(0);

const selectedContact = contacts.find(c =>

c.id === selectedId

);

```
function handleSave(updatedData) {
const nextContacts = contacts.map(c => {
if (c.id === updatedData.id) {
return updatedData;
} else {
return c;
}
});
setContacts(nextContacts);
}

return (
<div>
<ContactList
contacts={contacts}
selectedId={selectedId}
onSelect={id => setSelectedId(id)}
/>
<hr />
<EditContact
key={selectedId}
initialData={selectedContact}
onSave={handleSave}
/>
</div>
)
}

const initialContacts = [
{ id: 0, name: 'Taylor', email: 'taylor@mail.com' },
{ id: 1, name: 'Alice', email: 'alice@mail.com' },
{ id: 2, name: 'Bob', email: 'bob@mail.com' }
];
```

```js ContactList.js
export default function ContactList({
```

```
contacts,
selectedId,
onSelect
}) {
return (
<section>
<ul>
{contacts.map(contact =>
<li key={contact.id}>
<button onClick={() => {
onSelect(contact.id);
}}>
{contact.id === selectedId ?
<b>{contact.name}</b> :
contact.name
}
</button>
</li>
)}
</ul>
</section>
);
}
```

```js EditContact.js
import { useState } from 'react';

export default function EditContact({ initialData, onSave }) {
const [name, setName] = useState(initialData.name);
const [email, setEmail] = useState(initialData.email);
return (
<section>
<label>
Name:{' '}
<input
type="text"
```

value={name}
onChange={e => setName(e.target.value)}
/>
</label>
<label>
Email:{' '}
<input
type="email"
value={email}
onChange={e => setEmail(e.target.value)}
/>
</label>
<button onClick={() => {
const updatedData = {
id: initialData.id,
name: name,
email: email
};
onSave(updatedData);
}}>
Save
</button>
<button onClick={() => {
setName(initialData.name);
setEmail(initialData.email);
}}>
Reset
</button>
</section>
);
}
```

```css
ul, li {
list-style: none;

margin: 0;

padding: 0;

}

li { display: inline-block; }

li button {

padding: 10px;

}

label {

display: block;

margin: 10px 0;

}

button {

margin-right: 10px;

margin-bottom: 10px;

}

```

</Sandpack>

</Solution>

Clear an image while it's loading {/*clear-an-image-while-its-loading*/}

When you press "Next", the browser starts loading the next image. However, because it's displayed in the same `<img>` tag, by default you would still see the previous image until the next one loads. This may be undesirable if it's important for the text to always match the image. Change it so that the moment you press "Next", the previous image immediately clears.

<Hint>

Is there a way to tell React to re-create the DOM instead of reusing it?

</Hint>

<Sandpack>

```js

import { useState } from 'react';

export default function Gallery() {

const [index, setIndex] = useState(0);

const hasNext = index < images.length - 1;

function handleClick() {

```
if (hasNext) {
setIndex(index + 1);
} else {
setIndex(0);
}
}

let image = images[index];
return (
<>
<button onClick={handleClick}>
Next
</button>
<h3>
Image {index + 1} of {images.length}
</h3>

<p>
{image.place}
</p>
</>
);
}

let images = [{
place: 'Penang, Malaysia',
src: 'https://i.imgur.com/FJeJR8M.jpg'
}, {
place: 'Lisbon, Portugal',
src: 'https://i.imgur.com/dB2LRbj.jpg'
}, {
place: 'Bilbao, Spain',
src: 'https://i.imgur.com/z08o2TS.jpg'
}, {
place: 'Valparaíso, Chile',
src: 'https://i.imgur.com/Y3utgTi.jpg'
}, {
```

place: 'Schwyz, Switzerland',

src: 'https://i.imgur.com/JBbMpWY.jpg'

}, {

place: 'Prague, Czechia',

src: 'https://i.imgur.com/QwUKKmF.jpg'

}, {

place: 'Ljubljana, Slovenia',

src: 'https://i.imgur.com/3aIiwfm.jpg'

}];

```

```css

img { width: 150px; height: 150px; }

```

</Sandpack>

<Solution>

You can provide a `key` to the `<img>` tag. When that `key` changes, React will re-create the `<img>` DOM node from scratch. This causes a brief flash when each image loads, so it's not something you'd want to do for every image in your app. But it makes sense if you want to ensure the image always matches the text.

<Sandpack>

```js

import { useState } from 'react';

export default function Gallery() {

const [index, setIndex] = useState(0);

const hasNext = index < images.length - 1;

function handleClick() {

if (hasNext) {

setIndex(index + 1);

} else {

setIndex(0);

}

}

let image = images[index];

```
return (
<>
<button onClick={handleClick}>
Next
</button>
<h3>
Image {index + 1} of {images.length}
</h3>

<p>
{image.place}
</p>
</>
);
}

let images = [{
place: 'Penang, Malaysia',
src: 'https://i.imgur.com/FJeJR8M.jpg'
}, {
place: 'Lisbon, Portugal',
src: 'https://i.imgur.com/dB2LRbj.jpg'
}, {
place: 'Bilbao, Spain',
src: 'https://i.imgur.com/z08o2TS.jpg'
}, {
place: 'Valparaíso, Chile',
src: 'https://i.imgur.com/Y3utgTi.jpg'
}, {
place: 'Schwyz, Switzerland',
src: 'https://i.imgur.com/JBbMpWY.jpg'
}, {
place: 'Prague, Czechia',
src: 'https://i.imgur.com/QwUKKmF.jpg'
}, {
place: 'Ljubljana, Slovenia',
```

src: 'https://i.imgur.com/3aIiwfm.jpg'

}];

```

```css

img { width: 150px; height: 150px; }

```

</Sandpack>

</Solution>

Fix misplaced state in the list {/*fix-misplaced-state-in-the-list*/}

In this list, each `Contact` has state that determines whether "Show email" has been pressed for it. Press "Show email" for Alice, and then tick the "Show in reverse order" checkbox. You will notice that it's _Taylor's_ email that is expanded now, but Alice's--which has moved to the bottom--appears collapsed.

Fix it so that the expanded state is associated with each contact, regardless of the chosen ordering.

<Sandpack>

```js App.js

import { useState } from 'react';

import Contact from './Contact.js';

export default function ContactList() {

const [reverse, setReverse] = useState(false);

const displayedContacts = [...contacts];

if (reverse) {

displayedContacts.reverse();

}

return (

<>

<label>

<input

type="checkbox"

value={reverse}

onChange={e => {

setReverse(e.target.checked)

}}

/>{' '}
Show in reverse order
</label>
<ul>
{displayedContacts.map((contact, i) =>
<li key={i}>
<Contact contact={contact} />
</li>
)}
</ul>
</>
);
}

const contacts = [
{ id: 0, name: 'Alice', email: 'alice@mail.com' },
{ id: 1, name: 'Bob', email: 'bob@mail.com' },
{ id: 2, name: 'Taylor', email: 'taylor@mail.com' }
];
```

```js Contact.js
import { useState } from 'react';

export default function Contact({ contact }) {
const [expanded, setExpanded] = useState(false);
return (
<>
{contact.name}
{expanded &&
{contact.email}
}
<button onClick={() => {
setExpanded(!expanded);
}}>
{expanded ? 'Hide' : 'Show'} email
</button>

</>
);
}
```

```css
ul, li {
list-style: none;
margin: 0;
padding: 0;
}
li {
margin-bottom: 20px;
}
label {
display: block;
margin: 10px 0;
}
button {
margin-right: 10px;
margin-bottom: 10px;
}
```

</Sandpack>

<Solution>

The problem is that this example was using index as a `key`:

```js
{displayedContacts.map((contact, i) =>
<li key={i}>
```

However, you want the state to be associated with _each particular contact_.

Using the contact ID as a `key` instead fixes the issue:

<Sandpack>
```

```js App.js
import { useState } from 'react';
import Contact from './Contact.js';

export default function ContactList() {
const [reverse, setReverse] = useState(false);

const displayedContacts = [...contacts];
if (reverse) {
displayedContacts.reverse();
}

return (
<>
<label>
<input
type="checkbox"
value={reverse}
onChange={e => {
setReverse(e.target.checked)
}}
/>{' '}
Show in reverse order
</label>
<ul>
{displayedContacts.map(contact =>
<li key={contact.id}>
<Contact contact={contact} />
</li>
)}
</ul>
</>
);
}

const contacts = [
{ id: 0, name: 'Alice', email: 'alice@mail.com' },
{ id: 1, name: 'Bob', email: 'bob@mail.com' },
```

```
{ id: 2, name: 'Taylor', email: 'taylor@mail.com' }
];
```

```js Contact.js
import { useState } from 'react';

export default function Contact({ contact }) {
const [expanded, setExpanded] = useState(false);
return (
<>
<p><b>{contact.name}</b></p>
{expanded &&
<p><i>{contact.email}</i></p>
}
<button onClick={() => {
setExpanded(!expanded);
}}>
{expanded ? 'Hide' : 'Show'} email
</button>
</>
);
}
```

```css
ul, li {
list-style: none;
margin: 0;
padding: 0;
}
li {
margin-bottom: 20px;
}
label {
display: block;
margin: 10px 0;
```

}

button {

margin-right: 10px;

margin-bottom: 10px;

}

```

</Sandpack>

State is associated with the tree position. A `key` lets you specify a named position instead of relying on order.

</Solution>

</Challenges>

---

title: 'Reusing Logic with Custom Hooks'

---

<Intro>

React comes with several built-in Hooks like `useState`, `useContext`, and `useEffect`. Sometimes, you'll wish that there was a Hook for some more specific purpose: for example, to fetch data, to keep track of whether the user is online, or to connect to a chat room. You might not find these Hooks in React, but you can create your own Hooks for your application's needs.

</Intro>

<YouWillLearn>

- What custom Hooks are, and how to write your own

- How to reuse logic between components

- How to name and structure your custom Hooks

- When and why to extract custom Hooks

</YouWillLearn>

## Custom Hooks: Sharing logic between components
{/*custom-hooks-sharing-logic-between-components*/}

Imagine you're developing an app that heavily relies on the network (as most apps do). You want to warn the user if their network connection has accidentally gone off while they were using your app. How would you go about it? It seems like you'll need two things in your component:

1. A piece of state that tracks whether the network is online.

2. An Effect that subscribes to the global [`online`](https://developer.mozilla.org/en-US/docs/Web/API/Window/online_event) and
```

[`offline`](https://developer.mozilla.org/en-US/docs/Web/API/Window/offline_event) events, and updates that state.

This will keep your component [synchronized](/learn/synchronizing-with-effects) with the network status. You might start with something like this:

<Sandpack>

```js
import { useState, useEffect } from 'react';

export default function StatusBar() {
const [isOnline, setIsOnline] = useState(true);
useEffect(() => {
function handleOnline() {
setIsOnline(true);
}
function handleOffline() {
setIsOnline(false);
}
window.addEventListener('online', handleOnline);
window.addEventListener('offline', handleOffline);
return () => {
window.removeEventListener('online', handleOnline);
window.removeEventListener('offline', handleOffline);
};
}, []);

return <h1>{isOnline ? '■ Online' : '■ Disconnected'}</h1>;
}
```

</Sandpack>

Try turning your network on and off, and notice how this `StatusBar` updates in response to your actions.

Now imagine you *also* want to use the same logic in a different component. You want to implement a Save button that will become disabled and show "Reconnecting..." instead of "Save" while the network is off.

To start, you can copy and paste the `isOnline` state and the Effect into `SaveButton`:

<Sandpack>

```js
import { useState, useEffect } from 'react';

export default function SaveButton() {
const [isOnline, setIsOnline] = useState(true);
useEffect(() => {
function handleOnline() {
setIsOnline(true);
}
function handleOffline() {
setIsOnline(false);
}
window.addEventListener('online', handleOnline);
window.addEventListener('offline', handleOffline);
return () => {
window.removeEventListener('online', handleOnline);
window.removeEventListener('offline', handleOffline);
};
}, []);

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

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

</Sandpack>

Verify that, if you turn off the network, the button will change its appearance.

These two components work fine, but the duplication in logic between them is unfortunate. It seems like even though they have different *visual appearance,* you want to reuse the logic between them.

Extracting your own custom Hook from a component
{/*extracting-your-own-custom-hook-from-a-component*/}

Imagine for a moment that, similar to [`useState`](/reference/react/useState) and [`useEffect`](/reference/react/useEffect), there was a built-in `useOnlineStatus` Hook. Then both of these components could be simplified and you could remove the duplication between them:

```js {2,7}
function StatusBar() {
const isOnline = useOnlineStatus();
return <h1>{isOnline ? '■ Online' : '■ Disconnected'}</h1>;
}

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

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

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

Although there is no such built-in Hook, you can write it yourself. Declare a function called `useOnlineStatus` and move all the duplicated code into it from the components you wrote earlier:

```js {2-16}
function useOnlineStatus() {
const [isOnline, setIsOnline] = useState(true);
useEffect(() => {
function handleOnline() {
setIsOnline(true);
}
function handleOffline() {
setIsOnline(false);
}
window.addEventListener('online', handleOnline);
window.addEventListener('offline', handleOffline);
```

```
return () => {
window.removeEventListener('online', handleOnline);
window.removeEventListener('offline', handleOffline);
};
}, []);
return isOnline;
}
```

At the end of the function, return `isOnline`. This lets your components read that value:

<Sandpack>

```js
import { useOnlineStatus } from './useOnlineStatus.js';

function StatusBar() {
const isOnline = useOnlineStatus();
return <h1>{isOnline ? '■ Online' : '■ Disconnected'}</h1>;
}

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

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

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

export default function App() {
return (
<>
<SaveButton />
<StatusBar />
</>
```

```
);
}
```

```js useOnlineStatus.js
import { useState, useEffect } from 'react';

export function useOnlineStatus() {
const [isOnline, setIsOnline] = useState(true);
useEffect(() => {
function handleOnline() {
setIsOnline(true);
}
function handleOffline() {
setIsOnline(false);
}
window.addEventListener('online', handleOnline);
window.addEventListener('offline', handleOffline);
return () => {
window.removeEventListener('online', handleOnline);
window.removeEventListener('offline', handleOffline);
};
}, []);
return isOnline;
}
```

</Sandpack>

Verify that switching the network on and off updates both components.

Now your components don't have as much repetitive logic. **More importantly, the code inside them describes *what they want to do* (use the online status!) rather than *how to do it* (by subscribing to the browser events).**

When you extract logic into custom Hooks, you can hide the gnarly details of how you deal with some external system or a browser API. The code of your components expresses your intent, not the implementation.

Hook names always start with `use` {/*hook-names-always-start-with-use*/}

React applications are built from components. Components are built from Hooks, whether built-in or custom. You'll likely often use custom Hooks created by others, but occasionally you might write one yourself!

You must follow these naming conventions:

1. **React component names must start with a capital letter,** like `StatusBar` and `SaveButton`. React components also need to return something that React knows how to display, like a piece of JSX.

2. **Hook names must start with `use` followed by a capital letter,** like [`useState`](/reference/react/useState) (built-in) or `useOnlineStatus` (custom, like earlier on the page). Hooks may return arbitrary values.

This convention guarantees that you can always look at a component and know where its state, Effects, and other React features might "hide". For example, if you see a `getColor()` function call inside your component, you can be sure that it can't possibly contain React state inside because its name doesn't start with `use`. However, a function call like `useOnlineStatus()` will most likely contain calls to other Hooks inside!

<Note>

If your linter is [configured for React,](/learn/editor-setup#linting) it will enforce this naming convention. Scroll up to the sandbox above and rename `useOnlineStatus` to `getOnlineStatus`. Notice that the linter won't allow you to call `useState` or `useEffect` inside of it anymore. Only Hooks and components can call other Hooks!

</Note>

<DeepDive>

Should all functions called during rendering start with the use prefix? {/*should-all-functions-called-during-rendering-start-with-the-use-prefix*/}

No. Functions that don't *call* Hooks don't need to *be* Hooks.

If your function doesn't call any Hooks, avoid the `use` prefix. Instead, write it as a regular function *without* the `use` prefix. For example, `useSorted` below doesn't call Hooks, so call it `getSorted` instead:

```js
// ■ Avoid: A Hook that doesn't use Hooks
function useSorted(items) {
return items.slice().sort();
}

// ■ Good: A regular function that doesn't use Hooks
function getSorted(items) {
return items.slice().sort();
}
```

```

This ensures that your code can call this regular function anywhere, including conditions:

```js
function List({ items, shouldSort }) {
let displayedItems = items;
if (shouldSort) {
// ■ It's ok to call getSorted() conditionally because it's not a Hook
displayedItems = getSorted(items);
}
// ...
}
```

You should give `use` prefix to a function (and thus make it a Hook) if it uses at least one Hook inside of it:

```js
// ■ Good: A Hook that uses other Hooks
function useAuth() {
return useContext(Auth);
}
```

Technically, this isn't enforced by React. In principle, you could make a Hook that doesn't call other Hooks. This is often confusing and limiting so it's best to avoid that pattern. However, there may be rare cases where it is helpful. For example, maybe your function doesn't use any Hooks right now, but you plan to add some Hook calls to it in the future. Then it makes sense to name it with the `use` prefix:

```js {3-4}
// ■ Good: A Hook that will likely use some other Hooks later
function useAuth() {
// TODO: Replace with this line when authentication is implemented:
// return useContext(Auth);
return TEST_USER;
}
```

Then components won't be able to call it conditionally. This will become important when you actually add Hook calls inside. If you don't plan to use Hooks inside it (now or later), don't make it a Hook.
```

</DeepDive>

Custom Hooks let you share stateful logic, not state itself
{/*custom-hooks-let-you-share-stateful-logic-not-state-itself*/}

In the earlier example, when you turned the network on and off, both components updated together. However, it's wrong to think that a single `isOnline` state variable is shared between them. Look at this code:

```js {2,7}
function StatusBar() {
const isOnline = useOnlineStatus();
// ...
}

function SaveButton() {
const isOnline = useOnlineStatus();
// ...
}
```

It works the same way as before you extracted the duplication:

```js {2-5,10-13}
function StatusBar() {
const [isOnline, setIsOnline] = useState(true);
useEffect(() => {
// ...
}, []);
// ...
}

function SaveButton() {
const [isOnline, setIsOnline] = useState(true);
useEffect(() => {
// ...
}, []);
// ...
}
```

These are two completely independent state variables and Effects! They happened to have the same value at the same time because you synchronized them with the same external value (whether the network is on).

To better illustrate this, we'll need a different example. Consider this `Form` component:

<Sandpack>

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

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

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

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

return (
<>
<label>
First name:
<input value={firstName} onChange={handleFirstNameChange} />
</label>
<label>
Last name:
<input value={lastName} onChange={handleLastNameChange} />
</label>
<p><b>Good morning, {firstName} {lastName}.</b></p>
</>
);
}
```

```css
label { display: block; }
```

input { margin-left: 10px; }
```

</Sandpack>

There's some repetitive logic for each form field:

1. There's a piece of state (`firstName` and `lastName`).
1. There's a change handler (`handleFirstNameChange` and `handleLastNameChange`).
1. There's a piece of JSX that specifies the `value` and `onChange` attributes for that input.

You can extract the repetitive logic into this `useFormInput` custom Hook:

<Sandpack>

```js
import { useFormInput } from './useFormInput.js';

export default function Form() {
const firstNameProps = useFormInput('Mary');
const lastNameProps = useFormInput('Poppins');

return (
<>
<label>
First name:
<input {...firstNameProps} />
</label>
<label>
Last name:
<input {...lastNameProps} />
</label>
<p><b>Good morning, {firstNameProps.value} {lastNameProps.value}.</b></p>
</>
);
}
```

```js useFormInput.js active
import { useState } from 'react';

export function useFormInput(initialValue) {

```
const [value, setValue] = useState(initialValue);

function handleChange(e) {
setValue(e.target.value);
}

const inputProps = {
value: value,
onChange: handleChange
};

return inputProps;
}
```

```css
label { display: block; }
input { margin-left: 10px; }
```

</Sandpack>

Notice that it only declares *one* state variable called `value`.

However, the `Form` component calls `useFormInput` *two times:*

```js
function Form() {
const firstNameProps = useFormInput('Mary');
const lastNameProps = useFormInput('Poppins');
// ...
```

This is why it works like declaring two separate state variables!

Custom Hooks let you share *stateful logic* but not *state itself.* Each call to a Hook is completely independent from every other call to the same Hook. This is why the two sandboxes above are completely equivalent. If you'd like, scroll back up and compare them. The behavior before and after extracting a custom Hook is identical.

When you need to share the state itself between multiple components, [lift it up and pass it down](/learn/sharing-state-between-components) instead.

Passing reactive values between Hooks {/*passing-reactive-values-between-hooks*/}

The code inside your custom Hooks will re-run during every re-render of your component. This is why, like components, custom Hooks [need to be pure.](/learn/keeping-components-pure) Think of custom Hooks' code as part of your component's body!

Because custom Hooks re-render together with your component, they always receive the latest props and state. To see what this means, consider this chat room example. Change the server URL or the chat room:

<Sandpack>

```js App.js
import { useState } from 'react';
import ChatRoom from './ChatRoom.js';

export default function App() {
const [roomId, setRoomId] = useState('general');
return (
<>
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
</select>
</label>
<hr />
<ChatRoom
roomId={roomId}
/>
</>
);
}
```

```js ChatRoom.js active
import { useState, useEffect } from 'react';
```

````
import { createConnection } from './chat.js';
import { showNotification } from './notifications.js';

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

useEffect(() => {
const options = {
serverUrl: serverUrl,
roomId: roomId
};
const connection = createConnection(options);
connection.on('message', (msg) => {
showNotification('New message: ' + msg);
});
connection.connect();
return () => connection.disconnect();
}, [roomId, serverUrl]);

return (
<>
<label>
Server URL:
<input value={serverUrl} onChange={e => setServerUrl(e.target.value)} />
</label>
<h1>Welcome to the {roomId} room!</h1>
</>
);
}
```

```js chat.js
export function createConnection({ serverUrl, roomId }) {
// A real implementation would actually connect to the server
if (typeof serverUrl !== 'string') {
throw Error('Expected serverUrl to be a string. Received: ' + serverUrl);
}
if (typeof roomId !== 'string') {
````

```
throw Error('Expected roomId to be a string. Received: ' + roomId);
}
let intervalId;
let messageCallback;
return {
connect() {
console.log('■ Connecting to '' + roomId + '' room at ' + serverUrl + '...');
clearInterval(intervalId);
intervalId = setInterval(() => {
if (messageCallback) {
if (Math.random() > 0.5) {
messageCallback('hey')
} else {
messageCallback('lol');
}
}
}, 3000);
},
disconnect() {
clearInterval(intervalId);
messageCallback = null;
console.log('■ Disconnected from '' + roomId + '' room at ' + serverUrl + '');
},
on(event, callback) {
if (messageCallback) {
throw Error('Cannot add the handler twice.');
}
if (event !== 'message') {
throw Error('Only "message" event is supported.');
}
messageCallback = callback;
},
};
}
```

```js notifications.js
import Toastify from 'toastify-js';
import 'toastify-js/src/toastify.css';

export function showNotification(message, theme = 'dark') {
Toastify({
text: message,
duration: 2000,
gravity: 'top',
position: 'right',
style: {
background: theme === 'dark' ? 'black' : 'white',
color: theme === 'dark' ? 'white' : 'black',
},
}).showToast();
}
```

```json package.json hidden
{
"dependencies": {
"react": "latest",
"react-dom": "latest",
"react-scripts": "latest",
"toastify-js": "1.12.0"
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
}
}
```

```css
input { display: block; margin-bottom: 20px; }
```

button { margin-left: 10px; }
```

</Sandpack>

When you change `serverUrl` or `roomId`, the Effect ["reacts" to your changes](/learn/lifecycle-of-reactive-effects#effects-react-to-reactive-values) and re-synchronizes. You can tell by the console messages that the chat re-connects every time that you change your Effect's dependencies.

Now move the Effect's code into a custom Hook:

```js {2-13}
export function useChatRoom({ serverUrl, roomId }) {
useEffect(() => {
const options = {
serverUrl: serverUrl,
roomId: roomId
};
const connection = createConnection(options);
connection.connect();
connection.on('message', (msg) => {
showNotification('New message: ' + msg);
});
return () => connection.disconnect();
}, [roomId, serverUrl]);
}
```

This lets your `ChatRoom` component call your custom Hook without worrying about how it works inside:

```js {4-7}
export default function ChatRoom({ roomId }) {
const [serverUrl, setServerUrl] = useState('https://localhost:1234');

useChatRoom({
roomId: roomId,
serverUrl: serverUrl
});

return (

```
<>
<label>
Server URL:
<input value={serverUrl} onChange={e => setServerUrl(e.target.value)} />
</label>
<h1>Welcome to the {roomId} room!</h1>
</>
);
}
```

This looks much simpler! (But it does the same thing.)

Notice that the logic *still responds* to prop and state changes. Try editing the server URL or the selected room:

<Sandpack>

```js App.js
import { useState } from 'react';
import ChatRoom from './ChatRoom.js';

export default function App() {
const [roomId, setRoomId] = useState('general');
return (
<>
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
</select>
</label>
<hr />
<ChatRoom
```

```
roomId={roomId}
/>
</>
);
}
```

```js ChatRoom.js active
import { useState } from 'react';
import { useChatRoom } from './useChatRoom.js';

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

useChatRoom({
roomId: roomId,
serverUrl: serverUrl
});

return (
<>
<label>
Server URL:
<input value={serverUrl} onChange={e => setServerUrl(e.target.value)} />
</label>
<h1>Welcome to the {roomId} room!</h1>
</>
);
}
```

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

export function useChatRoom({ serverUrl, roomId }) {
useEffect(() => {
const options = {
```

````
serverUrl: serverUrl,
roomId: roomId
};
const connection = createConnection(options);
connection.connect();
connection.on('message', (msg) => {
showNotification('New message: ' + msg);
});
return () => connection.disconnect();
}, [roomId, serverUrl]);
}
```

```js chat.js
export function createConnection({ serverUrl, roomId }) {
// A real implementation would actually connect to the server
if (typeof serverUrl !== 'string') {
throw Error('Expected serverUrl to be a string. Received: ' + serverUrl);
}
if (typeof roomId !== 'string') {
throw Error('Expected roomId to be a string. Received: ' + roomId);
}
let intervalId;
let messageCallback;
return {
connect() {
console.log('■ Connecting to "' + roomId + '" room at ' + serverUrl + '...');
clearInterval(intervalId);
intervalId = setInterval(() => {
if (messageCallback) {
if (Math.random() > 0.5) {
messageCallback('hey')
} else {
messageCallback('lol');
}
}
````

```
}, 3000);
},
disconnect() {
clearInterval(intervalId);
messageCallback = null;
console.log('■ Disconnected from "' + roomId + '" room at ' + serverUrl + '');
},
on(event, callback) {
if (messageCallback) {
throw Error('Cannot add the handler twice.');
}
if (event !== 'message') {
throw Error('Only "message" event is supported.');
}
messageCallback = callback;
},
};
}
```

```js notifications.js
import Toastify from 'toastify-js';
import 'toastify-js/src/toastify.css';

export function showNotification(message, theme = 'dark') {
Toastify({
text: message,
duration: 2000,
gravity: 'top',
position: 'right',
style: {
background: theme === 'dark' ? 'black' : 'white',
color: theme === 'dark' ? 'white' : 'black',
},
}).showToast();
}
```

```json package.json hidden
{
"dependencies": {
"react": "latest",
"react-dom": "latest",
"react-scripts": "latest",
"toastify-js": "1.12.0"
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
}
}
```

```css
input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }
```

</Sandpack>

Notice how you're taking the return value of one Hook:

```js {2}
export default function ChatRoom({ roomId }) {
const [serverUrl, setServerUrl] = useState('https://localhost:1234');

useChatRoom({
roomId: roomId,
serverUrl: serverUrl
});
// ...
```

and pass it as an input to another Hook:

```js {6}
```

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

useChatRoom({
roomId: roomId,
serverUrl: serverUrl
});
// ...
```

Every time your `ChatRoom` component re-renders, it passes the latest `roomId` and `serverUrl` to your Hook. This is why your Effect re-connects to the chat whenever their values are different after a re-render. (If you ever worked with audio or video processing software, chaining Hooks like this might remind you of chaining visual or audio effects. It's as if the output of `useState` "feeds into" the input of the `useChatRoom`.)

Passing event handlers to custom Hooks {/*passing-event-handlers-to-custom-hooks*/}

<Wip>

This section describes an **experimental API that has not yet been released** in a stable version of React.

</Wip>

As you start using `useChatRoom` in more components, you might want to let components customize its behavior. For example, currently, the logic for what to do when a message arrives is hardcoded inside the Hook:

```js {9-11}
export function useChatRoom({ serverUrl, roomId }) {
useEffect(() => {
const options = {
serverUrl: serverUrl,
roomId: roomId
};
const connection = createConnection(options);
connection.connect();
connection.on('message', (msg) => {
showNotification('New message: ' + msg);
});
return () => connection.disconnect();
}, [roomId, serverUrl]);
```

}
```

Let's say you want to move this logic back to your component:

```js {7-9}
export default function ChatRoom({ roomId }) {
const [serverUrl, setServerUrl] = useState('https://localhost:1234');

useChatRoom({
roomId: roomId,
serverUrl: serverUrl,
onReceiveMessage(msg) {
showNotification('New message: ' + msg);
}
});
// ...
```

To make this work, change your custom Hook to take `onReceiveMessage` as one of its named options:

```js {1,10,13}
export function useChatRoom({ serverUrl, roomId, onReceiveMessage }) {
useEffect(() => {
const options = {
serverUrl: serverUrl,
roomId: roomId
};
const connection = createConnection(options);
connection.connect();
connection.on('message', (msg) => {
onReceiveMessage(msg);
});
return () => connection.disconnect();
}, [roomId, serverUrl, onReceiveMessage]); // ■ All dependencies declared
}
```
```

This will work, but there's one more improvement you can do when your custom Hook accepts event handlers.

Adding a dependency on `onReceiveMessage` is not ideal because it will cause the chat to re-connect every time the component re-renders. [Wrap this event handler into an Effect Event to remove it from the dependencies:](/learn/removing-effect-dependencies#wrapping-an-event-handler-from-the-props)

```js {1,4,5,15,18}
import { useEffect, useEffectEvent } from 'react';
// ...

export function useChatRoom({ serverUrl, roomId, onReceiveMessage }) {
const onMessage = useEffectEvent(onReceiveMessage);

useEffect(() => {
const options = {
serverUrl: serverUrl,
roomId: roomId
};
const connection = createConnection(options);
connection.connect();
connection.on('message', (msg) => {
onMessage(msg);
});
return () => connection.disconnect();
}, [roomId, serverUrl]); // ■ All dependencies declared
}
```

Now the chat won't re-connect every time that the `ChatRoom` component re-renders. Here is a fully working demo of passing an event handler to a custom Hook that you can play with:

<Sandpack>

```js App.js
import { useState } from 'react';
import ChatRoom from './ChatRoom.js';

export default function App() {
const [roomId, setRoomId] = useState('general');
return (
<>
```

```
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
</select>
</label>
<hr />
<ChatRoom
roomId={roomId}
/>
</>
);
}
```

```js ChatRoom.js active
import { useState } from 'react';
import { useChatRoom } from './useChatRoom.js';
import { showNotification } from './notifications.js';

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

useChatRoom({
roomId: roomId,
serverUrl: serverUrl,
onReceiveMessage(msg) {
showNotification('New message: ' + msg);
}
});

return (
<>
```

<label>
Server URL:
<input value={serverUrl} onChange={e => setServerUrl(e.target.value)} />
</label>
<h1>Welcome to the {roomId} room!</h1>
</>
);
}
```

```js useChatRoom.js
import { useEffect } from 'react';
import { experimental_useEffectEvent as useEffectEvent } from 'react';
import { createConnection } from './chat.js';

export function useChatRoom({ serverUrl, roomId, onReceiveMessage }) {
const onMessage = useEffectEvent(onReceiveMessage);

useEffect(() => {
const options = {
serverUrl: serverUrl,
roomId: roomId
};
const connection = createConnection(options);
connection.connect();
connection.on('message', (msg) => {
onMessage(msg);
});
return () => connection.disconnect();
}, [roomId, serverUrl]);
}
```

```js chat.js
export function createConnection({ serverUrl, roomId }) {
// A real implementation would actually connect to the server
if (typeof serverUrl !== 'string') {
throw Error('Expected serverUrl to be a string. Received: ' + serverUrl);

```
}
if (typeof roomId !== 'string') {
throw Error('Expected roomId to be a string. Received: ' + roomId);
}
let intervalId;
let messageCallback;
return {
connect() {
console.log('■ Connecting to "' + roomId + '" room at ' + serverUrl + '...');
clearInterval(intervalId);
intervalId = setInterval(() => {
if (messageCallback) {
if (Math.random() > 0.5) {
messageCallback('hey')
} else {
messageCallback('lol');
}
}
}, 3000);
},
disconnect() {
clearInterval(intervalId);
messageCallback = null;
console.log('■ Disconnected from "' + roomId + '" room at ' + serverUrl + '');
},
on(event, callback) {
if (messageCallback) {
throw Error('Cannot add the handler twice.');
}
if (event !== 'message') {
throw Error('Only "message" event is supported.');
}
messageCallback = callback;
},
};
```

}
```

```js notifications.js
import Toastify from 'toastify-js';
import 'toastify-js/src/toastify.css';

export function showNotification(message, theme = 'dark') {
Toastify({
text: message,
duration: 2000,
gravity: 'top',
position: 'right',
style: {
background: theme === 'dark' ? 'black' : 'white',
color: theme === 'dark' ? 'white' : 'black',
},
}).showToast();
}
```

```json package.json hidden
{
"dependencies": {
"react": "experimental",
"react-dom": "experimental",
"react-scripts": "latest",
"toastify-js": "1.12.0"
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
}
}
```
```

```css
input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }
```

</Sandpack>

Notice how you no longer need to know *how* `useChatRoom` works in order to use it. You could add it to any other component, pass any other options, and it would work the same way. That's the power of custom Hooks.

When to use custom Hooks {/*when-to-use-custom-hooks*/}

You don't need to extract a custom Hook for every little duplicated bit of code. Some duplication is fine. For example, extracting a `useFormInput` Hook to wrap a single `useState` call like earlier is probably unnecessary.

However, whenever you write an Effect, consider whether it would be clearer to also wrap it in a custom Hook. [You shouldn't need Effects very often,](/learn/you-might-not-need-an-effect) so if you're writing one, it means that you need to "step outside React" to synchronize with some external system or to do something that React doesn't have a built-in API for. Wrapping it into a custom Hook lets you precisely communicate your intent and how the data flows through it.

For example, consider a `ShippingForm` component that displays two dropdowns: one shows the list of cities, and another shows the list of areas in the selected city. You might start with some code that looks like this:

```js {3-16,20-35}
function ShippingForm({ country }) {
const [cities, setCities] = useState(null);
// This Effect fetches cities for a country
useEffect(() => {
let ignore = false;
fetch(`/api/cities?country=${country}`)
.then(response => response.json())
.then(json => {
if (!ignore) {
setCities(json);
}
});
return () => {
ignore = true;
};
```

```
}, [country]);

const [city, setCity] = useState(null);
const [areas, setAreas] = useState(null);
// This Effect fetches areas for the selected city
useEffect(() => {
if (city) {
let ignore = false;
fetch(`/api/areas?city=${city}`)
.then(response => response.json())
.then(json => {
if (!ignore) {
setAreas(json);
}
});
return () => {
ignore = true;
};
}
}, [city]);

// ...
```

Although this code is quite repetitive, [it's correct to keep these Effects separate from each other.](/learn/removing-effect-dependencies#is-your-effect-doing-several-unrelated-things) They synchronize two different things, so you shouldn't merge them into one Effect. Instead, you can simplify the `ShippingForm` component above by extracting the common logic between them into your own `useData` Hook:

```
```js {2-18}
function useData(url) {
const [data, setData] = useState(null);
useEffect(() => {
if (url) {
let ignore = false;
fetch(url)
.then(response => response.json())
.then(json => {
```
```

```
if (!ignore) {
setData(json);
}
});
return () => {
ignore = true;
};
}
}, [url]);
return data;
}
```

Now you can replace both Effects in the `ShippingForm` components with calls to `useData`:

```js {2,4}
function ShippingForm({ country }) {
const cities = useData(`/api/cities?country=${country}`);
const [city, setCity] = useState(null);
const areas = useData(city ? `/api/areas?city=${city}` : null);
// ...
```

Extracting a custom Hook makes the data flow explicit. You feed the `url` in and you get the `data` out. By "hiding" your Effect inside `useData`, you also prevent someone working on the `ShippingForm` component from adding [unnecessary dependencies](/learn/removing-effect-dependencies) to it. With time, most of your app's Effects will be in custom Hooks.

<DeepDive>

Keep your custom Hooks focused on concrete high-level use cases {/*keep-your-custom-hooks-focused-on-concrete-high-level-use-cases*/}

Start by choosing your custom Hook's name. If you struggle to pick a clear name, it might mean that your Effect is too coupled to the rest of your component's logic, and is not yet ready to be extracted.

Ideally, your custom Hook's name should be clear enough that even a person who doesn't write code often could have a good guess about what your custom Hook does, what it takes, and what it returns:

* ■ `useData(url)`
* ■ `useImpressionLog(eventName, extraData)`
* ■ `useChatRoom(options)`

When you synchronize with an external system, your custom Hook name may be more technical and use jargon specific to that system. It's good as long as it would be clear to a person familiar with that system:

* ■ `useMediaQuery(query)`
* ■ `useSocket(url)`
* ■ `useIntersectionObserver(ref, options)`

Keep custom Hooks focused on concrete high-level use cases. Avoid creating and using custom "lifecycle" Hooks that act as alternatives and convenience wrappers for the `useEffect` API itself:

* ■ `useMount(fn)`
* ■ `useEffectOnce(fn)`
* ■ `useUpdateEffect(fn)`

For example, this `useMount` Hook tries to ensure some code only runs "on mount":

```js {4-5,14-15}
function ChatRoom({ roomId }) {
const [serverUrl, setServerUrl] = useState('https://localhost:1234');

// ■ Avoid: using custom "lifecycle" Hooks
useMount(() => {
const connection = createConnection({ roomId, serverUrl });
connection.connect();

post('/analytics/event', { eventName: 'visit_chat' });
});
// ...
}

// ■ Avoid: creating custom "lifecycle" Hooks
function useMount(fn) {
useEffect(() => {
fn();
}, []); // ■ React Hook useEffect has a missing dependency: 'fn'
}
```

Custom "lifecycle" Hooks like `useMount` don't fit well into the React paradigm. For example, this code example has a mistake (it doesn't "react" to `roomId` or `serverUrl` changes), but the linter won't warn you about it because the linter only checks direct `useEffect` calls. It won't know about your Hook.

If you're writing an Effect, start by using the React API directly:

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

// ■ Good: two raw Effects separated by purpose

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

useEffect(() => {
post('/analytics/event', { eventName: 'visit_chat', roomId });
}, [roomId]);

// ...
}
```

Then, you can (but don't have to) extract custom Hooks for different high-level use cases:

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

// ■ Great: custom Hooks named after their purpose
useChatRoom({ serverUrl, roomId });
useImpressionLog('visit_chat', { roomId });
// ...
}
```

A good custom Hook makes the calling code more declarative by constraining what it does. For example, `useChatRoom(options)` can only connect to the chat room, while `useImpressionLog(eventName, extraData)` can only send an impression log to the analytics. If your custom Hook API doesn't constrain the use cases and is very abstract, in the long run it's likely to introduce more problems than it solves.

</DeepDive>

Custom Hooks help you migrate to better patterns
{/*custom-hooks-help-you-migrate-to-better-patterns*/}

Effects are an ["escape hatch"](/learn/escape-hatches): you use them when you need to "step outside React" and when there is no better built-in solution for your use case. With time, the React team's goal is to reduce the number of the Effects in your app to the minimum by providing more specific solutions to more specific problems. Wrapping your Effects in custom Hooks makes it easier to upgrade your code when these solutions become available.

Let's return to this example:

<Sandpack>

```js
import { useOnlineStatus } from './useOnlineStatus.js';

function StatusBar() {
const isOnline = useOnlineStatus();
return <h1>{isOnline ? '■ Online' : '■ Disconnected'}</h1>;
}

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

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

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

export default function App() {
return (
<>
<SaveButton />
<StatusBar />
</>
);
}
```

```js useOnlineStatus.js active
```

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

export function useOnlineStatus() {
const [isOnline, setIsOnline] = useState(true);
useEffect(() => {
function handleOnline() {
setIsOnline(true);
}
function handleOffline() {
setIsOnline(false);
}
window.addEventListener('online', handleOnline);
window.addEventListener('offline', handleOffline);
return () => {
window.removeEventListener('online', handleOnline);
window.removeEventListener('offline', handleOffline);
};
}, []);
return isOnline;
}
```

</Sandpack>

In the above example, `useOnlineStatus` is implemented with a pair of [`useState`](/reference/react/useState) and [`useEffect`.](/reference/react/useEffect) However, this isn't the best possible solution. There is a number of edge cases it doesn't consider. For example, it assumes that when the component mounts, `isOnline` is already `true`, but this may be wrong if the network already went offline. You can use the browser [`navigator.onLine`](https://developer.mozilla.org/en-US/docs/Web/API/Navigator/onLine) API to check for that, but using it directly would not work on the server for generating the initial HTML. In short, this code could be improved.

Luckily, React 18 includes a dedicated API called [`useSyncExternalStore`](/reference/react/useSyncExternalStore) which takes care of all of these problems for you. Here is how your `useOnlineStatus` Hook, rewritten to take advantage of this new API:

<Sandpack>

```js
import { useOnlineStatus } from './useOnlineStatus.js';
```

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

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

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

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

export default function App() {
return (
<>
<SaveButton />
<StatusBar />
</>
);
}
```

```js useOnlineStatus.js active
import { useSyncExternalStore } from 'react';

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

```
}

export function useOnlineStatus() {
return useSyncExternalStore(
subscribe,
() => navigator.onLine, // How to get the value on the client
() => true // How to get the value on the server
);
}
```

</Sandpack>

Notice how **you didn't need to change any of the components** to make this migration:

```js {2,7}
function StatusBar() {
const isOnline = useOnlineStatus();
// ...
}

function SaveButton() {
const isOnline = useOnlineStatus();
// ...
}
```

This is another reason for why wrapping Effects in custom Hooks is often beneficial:

1. You make the data flow to and from your Effects very explicit.
2. You let your components focus on the intent rather than on the exact implementation of your Effects.
3. When React adds new features, you can remove those Effects without changing any of your components.

Similar to a [design system,](https://uxdesign.cc/everything-you-need-to-know-about-design-systems-54b109851969) you might find it helpful to start extracting common idioms from your app's components into custom Hooks. This will keep your components' code focused on the intent, and let you avoid writing raw Effects very often. Many excellent custom Hooks are maintained by the React community.

<DeepDive>

Will React provide any built-in solution for data fetching?
{/*will-react-provide-any-built-in-solution-for-data-fetching*/}

We're still working out the details, but we expect that in the future, you'll write data fetching like this:

```js {1,4,6}
import { use } from 'react'; // Not available yet!

function ShippingForm({ country }) {
const cities = use(fetch(`/api/cities?country=${country}`));
const [city, setCity] = useState(null);
const areas = city ? use(fetch(`/api/areas?city=${city}`)) : null;
// ...
```

If you use custom Hooks like `useData` above in your app, it will require fewer changes to migrate to the eventually recommended approach than if you write raw Effects in every component manually. However, the old approach will still work fine, so if you feel happy writing raw Effects, you can continue to do that.

</DeepDive>

There is more than one way to do it {/*there-is-more-than-one-way-to-do-it*/}

Let's say you want to implement a fade-in animation *from scratch* using the browser [`requestAnimationFrame`](https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame) API. You might start with an Effect that sets up an animation loop. During each frame of the animation, you could change the opacity of the DOM node you [hold in a ref](/learn/manipulating-the-dom-with-refs) until it reaches `1`. Your code might start like this:

<Sandpack>

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

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

useEffect(() => {
const duration = 1000;
const node = ref.current;

let startTime = performance.now();
let frameId = null;

function onFrame(now) {
const timePassed = now - startTime;
const progress = Math.min(timePassed / duration, 1);
onProgress(progress);
```

```
if (progress < 1) {
// We still have more frames to paint
frameId = requestAnimationFrame(onFrame);
}
}

function onProgress(progress) {
node.style.opacity = progress;
}

function start() {
onProgress(0);
startTime = performance.now();
frameId = requestAnimationFrame(onFrame);
}

function stop() {
cancelAnimationFrame(frameId);
startTime = null;
frameId = null;
}

start();
return () => stop();
}, []);

return (
<h1 className="welcome" ref={ref}>
Welcome
</h1>
);
}

export default function App() {
const [show, setShow] = useState(false);
return (
<>
<button onClick={() => setShow(!show)}>
{show ? 'Remove' : 'Show'}
```

</button>
<hr />
{show && <Welcome />}
</>
);
}
```

```css
label, button { display: block; margin-bottom: 20px; }
html, body { min-height: 300px; }
.welcome {
opacity: 0;
color: white;
padding: 50px;
text-align: center;
font-size: 50px;
background-image: radial-gradient(circle, rgba(63,94,251,1) 0%, rgba(252,70,107,1) 100%);
}
```

</Sandpack>

To make the component more readable, you might extract the logic into a `useFadeIn` custom Hook:

<Sandpack>

```js
import { useState, useEffect, useRef } from 'react';
import { useFadeIn } from './useFadeIn.js';

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

useFadeIn(ref, 1000);

return (
<h1 className="welcome" ref={ref}>
Welcome
</h1>
);

```
}

export default function App() {
const [show, setShow] = useState(false);
return (
<>
<button onClick={() => setShow(!show)}>
{show ? 'Remove' : 'Show'}
</button>
<hr />
{show && <Welcome />}
</>
);
}
```

```js useFadeIn.js
import { useEffect } from 'react';

export function useFadeIn(ref, duration) {
useEffect(() => {
const node = ref.current;

let startTime = performance.now();
let frameId = null;

function onFrame(now) {
const timePassed = now - startTime;
const progress = Math.min(timePassed / duration, 1);
onProgress(progress);
if (progress < 1) {
// We still have more frames to paint
frameId = requestAnimationFrame(onFrame);
}
}

function onProgress(progress) {
node.style.opacity = progress;
}
```

```
function start() {
onProgress(0);
startTime = performance.now();
frameId = requestAnimationFrame(onFrame);
}

function stop() {
cancelAnimationFrame(frameId);
startTime = null;
frameId = null;
}

start();
return () => stop();
}, [ref, duration]);
}
```

```css
label, button { display: block; margin-bottom: 20px; }
html, body { min-height: 300px; }
.welcome {
opacity: 0;
color: white;
padding: 50px;
text-align: center;
font-size: 50px;
background-image: radial-gradient(circle, rgba(63,94,251,1) 0%, rgba(252,70,107,1) 100%);
}
```

</Sandpack>

You could keep the `useFadeIn` code as is, but you could also refactor it more. For example, you could extract the logic for setting up the animation loop out of `useFadeIn` into a custom `useAnimationLoop` Hook:

<Sandpack>

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

```
import { useFadeIn } from './useFadeIn.js';

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

useFadeIn(ref, 1000);

return (
<h1 className="welcome" ref={ref}>
Welcome
</h1>
);
}

export default function App() {
const [show, setShow] = useState(false);
return (
<>
<button onClick={() => setShow(!show)}>
{show ? 'Remove' : 'Show'}
</button>
<hr />
{show && <Welcome />}
</>
);
}
```

```js useFadeIn.js active
import { useState, useEffect } from 'react';
import { experimental_useEffectEvent as useEffectEvent } from 'react';

export function useFadeIn(ref, duration) {
const [isRunning, setIsRunning] = useState(true);

useAnimationLoop(isRunning, (timePassed) => {
const progress = Math.min(timePassed / duration, 1);
ref.current.style.opacity = progress;
if (progress === 1) {
setIsRunning(false);
```

```
}
});
}

function useAnimationLoop(isRunning, drawFrame) {
const onFrame = useEffectEvent(drawFrame);

useEffect(() => {
if (!isRunning) {
return;
}

const startTime = performance.now();
let frameId = null;

function tick(now) {
const timePassed = now - startTime;
onFrame(timePassed);
frameId = requestAnimationFrame(tick);
}

tick();
return () => cancelAnimationFrame(frameId);
}, [isRunning]);
}
```

```css
label, button { display: block; margin-bottom: 20px; }
html, body { min-height: 300px; }
.welcome {
opacity: 0;
color: white;
padding: 50px;
text-align: center;
font-size: 50px;
background-image: radial-gradient(circle, rgba(63,94,251,1) 0%, rgba(252,70,107,1) 100%);
}
```

```json package.json hidden
{
"dependencies": {
"react": "experimental",
"react-dom": "experimental",
"react-scripts": "latest"
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
}
}
```

</Sandpack>

However, you didn't *have to* do that. As with regular functions, ultimately you decide where to draw the boundaries between different parts of your code. You could also take a very different approach. Instead of keeping the logic in the Effect, you could move most of the imperative logic inside a JavaScript [class:](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes)

<Sandpack>

```js
import { useState, useEffect, useRef } from 'react';
import { useFadeIn } from './useFadeIn.js';

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

useFadeIn(ref, 1000);

return (
<h1 className="welcome" ref={ref}>
Welcome
</h1>
);
}

export default function App() {
```

```
const [show, setShow] = useState(false);
return (
<>
<button onClick={() => setShow(!show)}>
{show ? 'Remove' : 'Show'}
</button>
<hr />
{show && <Welcome />}
</>
);
}
```

```js useFadeIn.js active
import { useState, useEffect } from 'react';
import { FadeInAnimation } from './animation.js';

export function useFadeIn(ref, duration) {
useEffect(() => {
const animation = new FadeInAnimation(ref.current);
animation.start(duration);
return () => {
animation.stop();
};
}, [ref, duration]);
}
```

```js animation.js
export class FadeInAnimation {
constructor(node) {
this.node = node;
}
start(duration) {
this.duration = duration;
this.onProgress(0);
this.startTime = performance.now();
```

```
this.frameId = requestAnimationFrame(() => this.onFrame());
}
onFrame() {
const timePassed = performance.now() - this.startTime;
const progress = Math.min(timePassed / this.duration, 1);
this.onProgress(progress);
if (progress === 1) {
this.stop();
} else {
// We still have more frames to paint
this.frameId = requestAnimationFrame(() => this.onFrame());
}
}
onProgress(progress) {
this.node.style.opacity = progress;
}
stop() {
cancelAnimationFrame(this.frameId);
this.startTime = null;
this.frameId = null;
this.duration = 0;
}
}
```

```css
label, button { display: block; margin-bottom: 20px; }
html, body { min-height: 300px; }
.welcome {
opacity: 0;
color: white;
padding: 50px;
text-align: center;
font-size: 50px;
background-image: radial-gradient(circle, rgba(63,94,251,1) 0%, rgba(252,70,107,1) 100%);
}
```

```

</Sandpack>

Effects let you connect React to external systems. The more coordination between Effects is needed (for example, to chain multiple animations), the more it makes sense to extract that logic out of Effects and Hooks *completely* like in the sandbox above. Then, the code you extracted *becomes* the "external system". This lets your Effects stay simple because they only need to send messages to the system you've moved outside React.

The examples above assume that the fade-in logic needs to be written in JavaScript. However, this particular fade-in animation is both simpler and much more efficient to implement with a plain [CSS Animation:](https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Animations/Using_CSS_animations)

<Sandpack>

```js
import { useState, useEffect, useRef } from 'react';
import './welcome.css';

function Welcome() {
return (
<h1 className="welcome">
Welcome
</h1>
);
}

export default function App() {
const [show, setShow] = useState(false);
return (
<>
<button onClick={() => setShow(!show)}>
{show ? 'Remove' : 'Show'}
</button>
<hr />
{show && <Welcome />}
</>
);
}
```

```css styles.css

label, button { display: block; margin-bottom: 20px; }
html, body { min-height: 300px; }
```

```css welcome.css active
.welcome {
color: white;
padding: 50px;
text-align: center;
font-size: 50px;
background-image: radial-gradient(circle, rgba(63,94,251,1) 0%, rgba(252,70,107,1) 100%);
animation: fadeIn 1000ms;
}
@keyframes fadeIn {
0% { opacity: 0; }
100% { opacity: 1; }
}
```

</Sandpack>

Sometimes, you don't even need a Hook!

<Recap>

- Custom Hooks let you share logic between components.
- Custom Hooks must be named starting with `use` followed by a capital letter.
- Custom Hooks only share stateful logic, not state itself.
- You can pass reactive values from one Hook to another, and they stay up-to-date.
- All Hooks re-run every time your component re-renders.
- The code of your custom Hooks should be pure, like your component's code.
- Wrap event handlers received by custom Hooks into Effect Events.
- Don't create custom Hooks like `useMount`. Keep their purpose specific.
- It's up to you how and where to choose the boundaries of your code.

</Recap>

<Challenges>

Extract a `useCounter` Hook {/*extract-a-usecounter-hook*/}
```

This component uses a state variable and an Effect to display a number that increments every second. Extract this logic into a custom Hook called `useCounter`. Your goal is to make the `Counter` component implementation look exactly like this:

```js
export default function Counter() {
const count = useCounter();
return <h1>Seconds passed: {count}</h1>;
}
```

You'll need to write your custom Hook in `useCounter.js` and import it into the `Counter.js` file.

<Sandpack>

```js
import { useState, useEffect } from 'react';

export default function Counter() {
const [count, setCount] = useState(0);
useEffect(() => {
const id = setInterval(() => {
setCount(c => c + 1);
}, 1000);
return () => clearInterval(id);
}, []);
return <h1>Seconds passed: {count}</h1>;
}
```

```js useCounter.js
// Write your custom Hook in this file!
```

</Sandpack>

<Solution>

Your code should look like this:

<Sandpack>

```js
```

```
import { useCounter } from './useCounter.js';

export default function Counter() {
const count = useCounter();
return <h1>Seconds passed: {count}</h1>;
}
```

```js useCounter.js
import { useState, useEffect } from 'react';

export function useCounter() {
const [count, setCount] = useState(0);
useEffect(() => {
const id = setInterval(() => {
setCount(c => c + 1);
}, 1000);
return () => clearInterval(id);
}, []);
return count;
}
```

</Sandpack>

Notice that `App.js` doesn't need to import `useState` or `useEffect` anymore.

</Solution>

Make the counter delay configurable {/*make-the-counter-delay-configurable*/}

In this example, there is a `delay` state variable controlled by a slider, but its value is not used. Pass the `delay` value to your custom `useCounter` Hook, and change the `useCounter` Hook to use the passed `delay` instead of hardcoding `1000` ms.

<Sandpack>

```js
import { useState } from 'react';
import { useCounter } from './useCounter.js';

export default function Counter() {
const [delay, setDelay] = useState(1000);
```

````
const count = useCounter();
return (
<>
<label>
Tick duration: {delay} ms
<br />
<input
type="range"
value={delay}
min="10"
max="2000"
onChange={e => setDelay(Number(e.target.value))}
/>
</label>
<hr />
<h1>Ticks: {count}</h1>
</>
);
}
```

```js useCounter.js
import { useState, useEffect } from 'react';

export function useCounter() {
const [count, setCount] = useState(0);
useEffect(() => {
const id = setInterval(() => {
setCount(c => c + 1);
}, 1000);
return () => clearInterval(id);
}, []);
return count;
}
```

</Sandpack>

<Solution>

Pass the `delay` to your Hook with `useCounter(delay)`. Then, inside the Hook, use `delay` instead of the hardcoded `1000` value. You'll need to add `delay` to your Effect's dependencies. This ensures that a change in `delay` will reset the interval.

<Sandpack>

```js
import { useState } from 'react';
import { useCounter } from './useCounter.js';

export default function Counter() {
const [delay, setDelay] = useState(1000);
const count = useCounter(delay);
return (
<>
<label>
Tick duration: {delay} ms
<br />
<input
type="range"
value={delay}
min="10"
max="2000"
onChange={e => setDelay(Number(e.target.value))}
/>
</label>
<hr />
<h1>Ticks: {count}</h1>
</>
);
}
```

```js useCounter.js
import { useState, useEffect } from 'react';

export function useCounter(delay) {
const [count, setCount] = useState(0);
```

```
useEffect(() => {
const id = setInterval(() => {
setCount(c => c + 1);
}, delay);
return () => clearInterval(id);
}, [delay]);
return count;
}
```

</Sandpack>

</Solution>

#### Extract `useInterval` out of `useCounter` {/*extract-useinterval-out-of-usecounter*/}

Currently, your `useCounter` Hook does two things. It sets up an interval, and it also increments a state variable on every interval tick. Split out the logic that sets up the interval into a separate Hook called `useInterval`. It should take two arguments: the `onTick` callback, and the `delay`. After this change, your `useCounter` implementation should look like this:

```js
export function useCounter(delay) {
const [count, setCount] = useState(0);
useInterval(() => {
setCount(c => c + 1);
}, delay);
return count;
}
```

Write `useInterval` in the `useInterval.js` file and import it into the `useCounter.js` file.

<Sandpack>

```js
import { useState } from 'react';
import { useCounter } from './useCounter.js';

export default function Counter() {
const count = useCounter(1000);
return <h1>Seconds passed: {count}</h1>;
```

}
```

```js useCounter.js
import { useState, useEffect } from 'react';

export function useCounter(delay) {
const [count, setCount] = useState(0);
useEffect(() => {
const id = setInterval(() => {
setCount(c => c + 1);
}, delay);
return () => clearInterval(id);
}, [delay]);
return count;
}
```

```js useInterval.js
// Write your Hook here!
```

</Sandpack>

<Solution>

The logic inside `useInterval` should set up and clear the interval. It doesn't need to do anything else.

<Sandpack>

```js
import { useCounter } from './useCounter.js';

export default function Counter() {
const count = useCounter(1000);
return <h1>Seconds passed: {count}</h1>;
}
```

```js useCounter.js
import { useState } from 'react';
import { useInterval } from './useInterval.js';

```
export function useCounter(delay) {
const [count, setCount] = useState(0);
useInterval(() => {
setCount(c => c + 1);
}, delay);
return count;
}
```

```js useInterval.js active
import { useEffect } from 'react';

export function useInterval(onTick, delay) {
useEffect(() => {
const id = setInterval(onTick, delay);
return () => clearInterval(id);
}, [onTick, delay]);
}
```

</Sandpack>

Note that there is a bit of a problem with this solution, which you'll solve in the next challenge.

</Solution>

#### Fix a resetting interval {/*fix-a-resetting-interval*/}

In this example, there are *two* separate intervals.

The `App` component calls `useCounter`, which calls `useInterval` to update the counter every second. But the `App` component *also* calls `useInterval` to randomly update the page background color every two seconds.

For some reason, the callback that updates the page background never runs. Add some logs inside `useInterval`:

```js {2,5}
useEffect(() => {
console.log('■ Setting up an interval with delay ', delay)
const id = setInterval(onTick, delay);
return () => {
console.log('■ Clearing an interval with delay ', delay)
```

clearInterval(id);

};

}, [onTick, delay]);

```

Do the logs match what you expect to happen? If some of your Effects seem to re-synchronize unnecessarily, can you guess which dependency is causing that to happen? Is there some way to [remove that dependency](/learn/removing-effect-dependencies) from your Effect?

After you fix the issue, you should expect the page background to update every two seconds.

<Hint>

It looks like your `useInterval` Hook accepts an event listener as an argument. Can you think of some way to wrap that event listener so that it doesn't need to be a dependency of your Effect?

</Hint>

<Sandpack>

```json package.json hidden
{
"dependencies": {
"react": "experimental",
"react-dom": "experimental",
"react-scripts": "latest"
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
}
}
```

```js
import { useCounter } from './useCounter.js';
import { useInterval } from './useInterval.js';

export default function Counter() {
const count = useCounter(1000);

````
useInterval(() => {
const randomColor = `hsla(${Math.random() * 360}, 100%, 50%, 0.2)`;
document.body.style.backgroundColor = randomColor;
}, 2000);

return <h1>Seconds passed: {count}</h1>;
}
```

```js useCounter.js
import { useState } from 'react';
import { useInterval } from './useInterval.js';

export function useCounter(delay) {
const [count, setCount] = useState(0);
useInterval(() => {
setCount(c => c + 1);
}, delay);
return count;
}
```

```js useInterval.js
import { useEffect } from 'react';
import { experimental_useEffectEvent as useEffectEvent } from 'react';

export function useInterval(onTick, delay) {
useEffect(() => {
const id = setInterval(onTick, delay);
return () => {
clearInterval(id);
};
}, [onTick, delay]);
}
```

</Sandpack>

<Solution>
````
````

Inside `useInterval`, wrap the tick callback into an Effect Event, as you did [earlier on this page.](/learn/reusing-logic-with-custom-hooks#passing-event-handlers-to-custom-hooks)

This will allow you to omit `onTick` from dependencies of your Effect. The Effect won't re-synchronize on every re-render of the component, so the page background color change interval won't get reset every second before it has a chance to fire.

With this change, both intervals work as expected and don't interfere with each other:

<Sandpack>

```json package.json hidden
{
"dependencies": {
"react": "experimental",
"react-dom": "experimental",
"react-scripts": "latest"
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
}
}
```

```js
import { useCounter } from './useCounter.js';
import { useInterval } from './useInterval.js';

export default function Counter() {
const count = useCounter(1000);

useInterval(() => {
const randomColor = `hsla(${Math.random() * 360}, 100%, 50%, 0.2)`;
document.body.style.backgroundColor = randomColor;
}, 2000);

return <h1>Seconds passed: {count}</h1>;
}
```

```

```js useCounter.js
import { useState } from 'react';
import { useInterval } from './useInterval.js';

export function useCounter(delay) {
const [count, setCount] = useState(0);
useInterval(() => {
setCount(c => c + 1);
}, delay);
return count;
}
```

```js useInterval.js active
import { useEffect } from 'react';
import { experimental_useEffectEvent as useEffectEvent } from 'react';

export function useInterval(callback, delay) {
const onTick = useEffectEvent(callback);
useEffect(() => {
const id = setInterval(onTick, delay);
return () => clearInterval(id);
}, [delay]);
}
```

</Sandpack>

</Solution>

Implement a staggering movement {/*implement-a-staggering-movement*/}

In this example, the `usePointerPosition()` Hook tracks the current pointer position. Try moving your cursor or your finger over the preview area and see the red dot follow your movement. Its position is saved in the `pos1` variable.

In fact, there are five (!) different red dots being rendered. You don't see them because currently they all appear at the same position. This is what you need to fix. What you want to implement instead is a "staggered" movement: each dot should "follow" the previous dot's path. For example, if you quickly move your cursor, the first dot should follow it immediately, the second dot should follow the first dot with a small delay, the third dot should follow the second dot, and so on.
```

You need to implement the `useDelayedValue` custom Hook. Its current implementation returns the `value` provided to it. Instead, you want to return the value back from `delay` milliseconds ago. You might need some state and an Effect to do this.

After you implement `useDelayedValue`, you should see the dots move following one another.

<Hint>

You'll need to store the `delayedValue` as a state variable inside your custom Hook. When the `value` changes, you'll want to run an Effect. This Effect should update `delayedValue` after the `delay`. You might find it helpful to call `setTimeout`.

Does this Effect need cleanup? Why or why not?

</Hint>

<Sandpack>

```js
import { usePointerPosition } from './usePointerPosition.js';

function useDelayedValue(value, delay) {
// TODO: Implement this Hook
return value;
}

export default function Canvas() {
const pos1 = usePointerPosition();
const pos2 = useDelayedValue(pos1, 100);
const pos3 = useDelayedValue(pos2, 200);
const pos4 = useDelayedValue(pos3, 100);
const pos5 = useDelayedValue(pos3, 50);
return (
<>
<Dot position={pos1} opacity={1} />
<Dot position={pos2} opacity={0.8} />
<Dot position={pos3} opacity={0.6} />
<Dot position={pos4} opacity={0.4} />
<Dot position={pos5} opacity={0.2} />
</>
);
}
```

````
function Dot({ position, opacity }) {
return (
<div style={{
position: 'absolute',
backgroundColor: 'pink',
borderRadius: '50%',
opacity,
transform: `translate(${position.x}px, ${position.y}px)`,
pointerEvents: 'none',
left: -20,
top: -20,
width: 40,
height: 40,
}} />
);
}
```

```js usePointerPosition.js
import { useState, useEffect } from 'react';

export function usePointerPosition() {
const [position, setPosition] = useState({ x: 0, y: 0 });
useEffect(() => {
function handleMove(e) {
setPosition({ x: e.clientX, y: e.clientY });
}
window.addEventListener('pointermove', handleMove);
return () => window.removeEventListener('pointermove', handleMove);
}, []);
return position;
}
```

```css
body { min-height: 300px; }
```
````
````

</Sandpack>

<Solution>

Here is a working version. You keep the `delayedValue` as a state variable. When `value` updates, your Effect schedules a timeout to update the `delayedValue`. This is why the `delayedValue` always "lags behind" the actual `value`.

<Sandpack>

```js
import { useState, useEffect } from 'react';
import { usePointerPosition } from './usePointerPosition.js';

function useDelayedValue(value, delay) {
const [delayedValue, setDelayedValue] = useState(value);

useEffect(() => {
setTimeout(() => {
setDelayedValue(value);
}, delay);
}, [value, delay]);

return delayedValue;
}

export default function Canvas() {
const pos1 = usePointerPosition();
const pos2 = useDelayedValue(pos1, 100);
const pos3 = useDelayedValue(pos2, 200);
const pos4 = useDelayedValue(pos3, 100);
const pos5 = useDelayedValue(pos3, 50);
return (
<>
<Dot position={pos1} opacity={1} />
<Dot position={pos2} opacity={0.8} />
<Dot position={pos3} opacity={0.6} />
<Dot position={pos4} opacity={0.4} />
<Dot position={pos5} opacity={0.2} />
</>
);
```

```
}

function Dot({ position, opacity }) {
return (
<div style={{
position: 'absolute',
backgroundColor: 'pink',
borderRadius: '50%',
opacity,
transform: `translate(${position.x}px, ${position.y}px)`,
pointerEvents: 'none',
left: -20,
top: -20,
width: 40,
height: 40,
}} />
);
}
```

```js usePointerPosition.js
import { useState, useEffect } from 'react';

export function usePointerPosition() {
const [position, setPosition] = useState({ x: 0, y: 0 });
useEffect(() => {
function handleMove(e) {
setPosition({ x: e.clientX, y: e.clientY });
}
window.addEventListener('pointermove', handleMove);
return () => window.removeEventListener('pointermove', handleMove);
}, []);
return position;
}
```

```css
body { min-height: 300px; }
```

```

</Sandpack>

Note that this Effect *does not* need cleanup. If you called `clearTimeout` in the cleanup function, then each time the `value` changes, it would reset the already scheduled timeout. To keep the movement continuous, you want all the timeouts to fire.

</Solution>

</Challenges>

---

title: 'Synchronizing with Effects'

---

<Intro>

Some components need to synchronize with external systems. For example, you might want to control a non-React component based on the React state, set up a server connection, or send an analytics log when a component appears on the screen. *Effects* let you run some code after rendering so that you can synchronize your component with some system outside of React.

</Intro>

<YouWillLearn>

- What Effects are
- How Effects are different from events
- How to declare an Effect in your component
- How to skip re-running an Effect unnecessarily
- Why Effects run twice in development and how to fix them

</YouWillLearn>

## What are Effects and how are they different from events?
{/*what-are-effects-and-how-are-they-different-from-events*/}

Before getting to Effects, you need to be familiar with two types of logic inside React components:

- **Rendering code** (introduced in [Describing the UI](/learn/describing-the-ui)) lives at the top level of your component. This is where you take the props and state, transform them, and return the JSX you want to see on the screen. [Rendering code must be pure.](/learn/keeping-components-pure) Like a math formula, it should only _calculate_ the result, but not do anything else.

- **Event handlers** (introduced in [Adding Interactivity](/learn/adding-interactivity)) are nested functions inside your components that *do* things rather than just calculate them. An event handler might update an input field, submit an HTTP POST request to buy a product, or navigate the user to another screen. Event handlers contain ["side effects"](https://en.wikipedia.org/wiki/Side_effect_(computer_science)) (they change the program's
```

state) caused by a specific user action (for example, a button click or typing).

Sometimes this isn't enough. Consider a `ChatRoom` component that must connect to the chat server whenever it's visible on the screen. Connecting to a server is not a pure calculation (it's a side effect) so it can't happen during rendering. However, there is no single particular event like a click that causes `ChatRoom` to be displayed.

***Effects* let you specify side effects that are caused by rendering itself, rather than by a particular event.** Sending a message in the chat is an *event* because it is directly caused by the user clicking a specific button. However, setting up a server connection is an *Effect* because it should happen no matter which interaction caused the component to appear. Effects run at the end of a [commit](/learn/render-and-commit) after the screen updates. This is a good time to synchronize the React components with some external system (like network or a third-party library).

<Note>

Here and later in this text, capitalized "Effect" refers to the React-specific definition above, i.e. a side effect caused by rendering. To refer to the broader programming concept, we'll say "side effect".

</Note>

You might not need an Effect {/*you-might-not-need-an-effect*/}

Don't rush to add Effects to your components. Keep in mind that Effects are typically used to "step out" of your React code and synchronize with some *external* system. This includes browser APIs, third-party widgets, network, and so on. If your Effect only adjusts some state based on other state, [you might not need an Effect.](/learn/you-might-not-need-an-effect)

How to write an Effect {/*how-to-write-an-effect*/}

To write an Effect, follow these three steps:

1. **Declare an Effect.** By default, your Effect will run after every render.
2. **Specify the Effect dependencies.** Most Effects should only re-run *when needed* rather than after every render. For example, a fade-in animation should only trigger when a component appears. Connecting and disconnecting to a chat room should only happen when the component appears and disappears, or when the chat room changes. You will learn how to control this by specifying *dependencies.*
3. **Add cleanup if needed.** Some Effects need to specify how to stop, undo, or clean up whatever they were doing. For example, "connect" needs "disconnect", "subscribe" needs "unsubscribe", and "fetch" needs either "cancel" or "ignore". You will learn how to do this by returning a *cleanup function*.

Let's look at each of these steps in detail.

Step 1: Declare an Effect {/*step-1-declare-an-effect*/}

To declare an Effect in your component, import the [`useEffect` Hook](/reference/react/useEffect) from React:

```js
import { useEffect } from 'react';
```

```

Then, call it at the top level of your component and put some code inside your Effect:

```js {2-4}
function MyComponent() {
useEffect(() => {
// Code here will run after *every* render
});
return <div />;
}
```

Every time your component renders, React will update the screen *and then* run the code inside `useEffect`. In other words, **`useEffect` "delays" a piece of code from running until that render is reflected on the screen.**

Let's see how you can use an Effect to synchronize with an external system. Consider a `<VideoPlayer>` React component. It would be nice to control whether it's playing or paused by passing an `isPlaying` prop to it:

```js
<VideoPlayer isPlaying={isPlaying} />;
```

Your custom `VideoPlayer` component renders the built-in browser [`<video>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/video) tag:

```js
function VideoPlayer({ src, isPlaying }) {
// TODO: do something with isPlaying
return <video src={src} />;
}
```

However, the browser `<video>` tag does not have an `isPlaying` prop. The only way to control it is to manually call the [`play()`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement/play) and [`pause()`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement/pause) methods on the DOM element. **You need to synchronize the value of `isPlaying` prop, which tells whether the video _should_ currently be playing, with calls like `play()` and `pause()`.**

We'll need to first [get a ref](/learn/manipulating-the-dom-with-refs) to the `<video>` DOM node.

You might be tempted to try to call `play()` or `pause()` during rendering, but that isn't correct:
```

<Sandpack>

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

function VideoPlayer({ src, isPlaying }) {
const ref = useRef(null);

if (isPlaying) {
ref.current.play(); // Calling these while rendering isn't allowed.
} else {
ref.current.pause(); // Also, this crashes.
}

return <video ref={ref} src={src} loop playsInline />;
}

export default function App() {
const [isPlaying, setIsPlaying] = useState(false);
return (
<>
<button onClick={() => setIsPlaying(!isPlaying)}>
{isPlaying ? 'Pause' : 'Play'}
</button>
<VideoPlayer
isPlaying={isPlaying}
src="https://interactive-examples.mdn.mozilla.net/media/cc0-videos/flower.mp4"
/>
</>
);
}
```

```css
button { display: block; margin-bottom: 20px; }
video { width: 250px; }
```

</Sandpack>

The reason this code isn't correct is that it tries to do something with the DOM node during rendering. In React, [rendering should be a pure calculation](/learn/keeping-components-pure) of JSX and should not contain side effects like modifying the DOM.

Moreover, when `VideoPlayer` is called for the first time, its DOM does not exist yet! There isn't a DOM node yet to call `play()` or `pause()` on, because React doesn't know what DOM to create until you return the JSX.

The solution here is to **wrap the side effect with `useEffect` to move it out of the rendering calculation:**

```js {6,12}
import { useEffect, useRef } from 'react';

function VideoPlayer({ src, isPlaying }) {
const ref = useRef(null);

useEffect(() => {
if (isPlaying) {
ref.current.play();
} else {
ref.current.pause();
}
});

return <video ref={ref} src={src} loop playsInline />;
}
```

By wrapping the DOM update in an Effect, you let React update the screen first. Then your Effect runs.

When your `VideoPlayer` component renders (either the first time or if it re-renders), a few things will happen. First, React will update the screen, ensuring the `<video>` tag is in the DOM with the right props. Then React will run your Effect. Finally, your Effect will call `play()` or `pause()` depending on the value of `isPlaying`.

Press Play/Pause multiple times and see how the video player stays synchronized to the `isPlaying` value:

<Sandpack>

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

function VideoPlayer({ src, isPlaying }) {
const ref = useRef(null);
```

```
useEffect(() => {
if (isPlaying) {
ref.current.play();
} else {
ref.current.pause();
}
});

return <video ref={ref} src={src} loop playsInline />;
}

export default function App() {
const [isPlaying, setIsPlaying] = useState(false);
return (
<>
<button onClick={() => setIsPlaying(!isPlaying)}>
{isPlaying ? 'Pause' : 'Play'}
</button>
<VideoPlayer
isPlaying={isPlaying}
src="https://interactive-examples.mdn.mozilla.net/media/cc0-videos/flower.mp4"
/>
</>
);
}
```

```css
button { display: block; margin-bottom: 20px; }
video { width: 250px; }
```

</Sandpack>

In this example, the "external system" you synchronized to React state was the browser media API. You can use a similar approach to wrap legacy non-React code (like jQuery plugins) into declarative React components.

Note that controlling a video player is much more complex in practice. Calling `play()` may fail, the user might play or pause using the built-in browser controls, and so on. This example is very simplified and incomplete.

<Pitfall>

By default, Effects run after *every* render. This is why code like this will **produce an infinite loop:**

```js
const [count, setCount] = useState(0);
useEffect(() => {
setCount(count + 1);
});
```

Effects run as a *result* of rendering. Setting state *triggers* rendering. Setting state immediately in an Effect is like plugging a power outlet into itself. The Effect runs, it sets the state, which causes a re-render, which causes the Effect to run, it sets the state again, this causes another re-render, and so on.

Effects should usually synchronize your components with an *external* system. If there's no external system and you only want to adjust some state based on other state, [you might not need an Effect.](/learn/you-might-not-need-an-effect)

</Pitfall>

Step 2: Specify the Effect dependencies {/*step-2-specify-the-effect-dependencies*/}

By default, Effects run after *every* render. Often, this is **not what you want:**

- Sometimes, it's slow. Synchronizing with an external system is not always instant, so you might want to skip doing it unless it's necessary. For example, you don't want to reconnect to the chat server on every keystroke.
- Sometimes, it's wrong. For example, you don't want to trigger a component fade-in animation on every keystroke. The animation should only play once when the component appears for the first time.

To demonstrate the issue, here is the previous example with a few `console.log` calls and a text input that updates the parent component's state. Notice how typing causes the Effect to re-run:

<Sandpack>

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

function VideoPlayer({ src, isPlaying }) {
const ref = useRef(null);

useEffect(() => {
if (isPlaying) {
console.log('Calling video.play()');
ref.current.play();
```

```
} else {
console.log('Calling video.pause()');
ref.current.pause();
}
});

return <video ref={ref} src={src} loop playsInline />;
}

export default function App() {
const [isPlaying, setIsPlaying] = useState(false);
const [text, setText] = useState('');
return (
<>
<input value={text} onChange={e => setText(e.target.value)} />
<button onClick={() => setIsPlaying(!isPlaying)}>
{isPlaying ? 'Pause' : 'Play'}
</button>
<VideoPlayer
isPlaying={isPlaying}
src="https://interactive-examples.mdn.mozilla.net/media/cc0-videos/flower.mp4"
/>
</>
);
}
```

```css
input, button { display: block; margin-bottom: 20px; }
video { width: 250px; }
```

</Sandpack>

You can tell React to **skip unnecessarily re-running the Effect** by specifying an array of *dependencies* as the second argument to the `useEffect` call. Start by adding an empty `[]` array to the above example on line 14:

```js {3}
useEffect(() => {
```

```
// ...
}, []);
```

You should see an error saying `React Hook useEffect has a missing dependency: 'isPlaying'`:

<Sandpack>

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

function VideoPlayer({ src, isPlaying }) {
const ref = useRef(null);

useEffect(() => {
if (isPlaying) {
console.log('Calling video.play()');
ref.current.play();
} else {
console.log('Calling video.pause()');
ref.current.pause();
}
}, []); // This causes an error

return <video ref={ref} src={src} loop playsInline />;
}

export default function App() {
const [isPlaying, setIsPlaying] = useState(false);
const [text, setText] = useState('');
return (
<>
<input value={text} onChange={e => setText(e.target.value)} />
<button onClick={() => setIsPlaying(!isPlaying)}>
{isPlaying ? 'Pause' : 'Play'}
</button>
<VideoPlayer
isPlaying={isPlaying}
src="https://interactive-examples.mdn.mozilla.net/media/cc0-videos/flower.mp4"
/>
```

```
</>
);
}
```

```css
input, button { display: block; margin-bottom: 20px; }
video { width: 250px; }
```

</Sandpack>

The problem is that the code inside of your Effect *depends on* the `isPlaying` prop to decide what to do, but this dependency was not explicitly declared. To fix this issue, add `isPlaying` to the dependency array:

```js {2,7}
useEffect(() => {
if (isPlaying) { // It's used here...
// ...
} else {
// ...
}
}, [isPlaying]); // ...so it must be declared here!
```

Now all dependencies are declared, so there is no error. Specifying `[isPlaying]` as the dependency array tells React that it should skip re-running your Effect if `isPlaying` is the same as it was during the previous render. With this change, typing into the input doesn't cause the Effect to re-run, but pressing Play/Pause does:

<Sandpack>

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

function VideoPlayer({ src, isPlaying }) {
const ref = useRef(null);

useEffect(() => {
if (isPlaying) {
console.log('Calling video.play()');
ref.current.play();
```

```
} else {
console.log('Calling video.pause()');
ref.current.pause();
}
}, [isPlaying]);

return <video ref={ref} src={src} loop playsInline />;
}

export default function App() {
const [isPlaying, setIsPlaying] = useState(false);
const [text, setText] = useState('');
return (
<>
<input value={text} onChange={e => setText(e.target.value)} />
<button onClick={() => setIsPlaying(!isPlaying)}>
{isPlaying ? 'Pause' : 'Play'}
</button>
<VideoPlayer
isPlaying={isPlaying}
src="https://interactive-examples.mdn.mozilla.net/media/cc0-videos/flower.mp4"
/>
</>
);
}
```

```css
input, button { display: block; margin-bottom: 20px; }
video { width: 250px; }
```

</Sandpack>

The dependency array can contain multiple dependencies. React will only skip re-running the Effect if *all* of the dependencies you specify have exactly the same values as they had during the previous render. React compares the dependency values using the [`Object.is`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/is) comparison. See the [`useEffect` reference](/reference/react/useEffect#reference) for details.

Notice that you can't "choose" your dependencies. You will get a lint error if the dependencies you specified don't match what React expects based on the code inside your Effect. This helps catch many bugs in your code. If you don't want some code to re-run, [*edit the Effect code itself* to not "need" that dependency.](/learn/lifecycle-of-reactive-effects#what-to-do-when-you-dont-want-to-re-synchronize)

<Pitfall>

The behaviors without the dependency array and with an *empty* `[]` dependency array are different:

```js {3,7,11}
useEffect(() => {
// This runs after every render
});

useEffect(() => {
// This runs only on mount (when the component appears)
}, []);

useEffect(() => {
// This runs on mount *and also* if either a or b have changed since the last render
}, [a, b]);
```

We'll take a close look at what "mount" means in the next step.

</Pitfall>

<DeepDive>

Why was the ref omitted from the dependency array?
{/*why-was-the-ref-omitted-from-the-dependency-array*/}

This Effect uses _both_ `ref` and `isPlaying`, but only `isPlaying` is declared as a dependency:

```js {9}
function VideoPlayer({ src, isPlaying }) {
const ref = useRef(null);
useEffect(() => {
if (isPlaying) {
ref.current.play();
} else {
ref.current.pause();
}
}, [isPlaying]);
```

```

This is because the `ref` object has a *stable identity:* React guarantees [you'll always get the same object](/reference/react/useRef#returns) from the same `useRef` call on every render. It never changes, so it will never by itself cause the Effect to re-run. Therefore, it does not matter whether you include it or not. Including it is fine too:

```js {9}
function VideoPlayer({ src, isPlaying }) {
const ref = useRef(null);
useEffect(() => {
if (isPlaying) {
ref.current.play();
} else {
ref.current.pause();
}
}, [isPlaying, ref]);
```

The [`set` functions](/reference/react/useState#setstate) returned by `useState` also have stable identity, so you will often see them omitted from the dependencies too. If the linter lets you omit a dependency without errors, it is safe to do.

Omitting always-stable dependencies only works when the linter can "see" that the object is stable. For example, if `ref` was passed from a parent component, you would have to specify it in the dependency array. However, this is good because you can't know whether the parent component always passes the same ref, or passes one of several refs conditionally. So your Effect _would_ depend on which ref is passed.

</DeepDive>

Step 3: Add cleanup if needed {/*step-3-add-cleanup-if-needed*/}

Consider a different example. You're writing a `ChatRoom` component that needs to connect to the chat server when it appears. You are given a `createConnection()` API that returns an object with `connect()` and `disconnect()` methods. How do you keep the component connected while it is displayed to the user?

Start by writing the Effect logic:

```js
useEffect(() => {
const connection = createConnection();
connection.connect();
});
```
```

```

It would be slow to connect to the chat after every re-render, so you add the dependency array:

```js {4}
useEffect(() => {
const connection = createConnection();
connection.connect();
}, []);
```

The code inside the Effect does not use any props or state, so your dependency array is `[]` (empty). This tells React to only run this code when the component "mounts", i.e. appears on the screen for the first time.

Let's try running this code:

<Sandpack>

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

export default function ChatRoom() {
useEffect(() => {
const connection = createConnection();
connection.connect();
}, []);
return <h1>Welcome to the chat!</h1>;
}
```

```js chat.js
export function createConnection() {
// A real implementation would actually connect to the server
return {
connect() {
console.log('■ Connecting...');
},
disconnect() {
console.log('■ Disconnected.');
```
```

```
}
};
}
```

```css
input { display: block; margin-bottom: 20px; }
```

</Sandpack>

This Effect only runs on mount, so you might expect `"■ Connecting..."` to be printed once in the console. **However, if you check the console, `"■ Connecting..."` gets printed twice. Why does it happen?**

Imagine the `ChatRoom` component is a part of a larger app with many different screens. The user starts their journey on the `ChatRoom` page. The component mounts and calls `connection.connect()`. Then imagine the user navigates to another screen--for example, to the Settings page. The `ChatRoom` component unmounts. Finally, the user clicks Back and `ChatRoom` mounts again. This would set up a second connection--but the first connection was never destroyed! As the user navigates across the app, the connections would keep piling up.

Bugs like this are easy to miss without extensive manual testing. To help you spot them quickly, in development React remounts every component once immediately after its initial mount.

Seeing the `"■ Connecting..."` log twice helps you notice the real issue: your code doesn't close the connection when the component unmounts.

To fix the issue, return a *cleanup function* from your Effect:

```js {4-6}
useEffect(() => {
const connection = createConnection();
connection.connect();
return () => {
connection.disconnect();
};
}, []);
```

React will call your cleanup function each time before the Effect runs again, and one final time when the component unmounts (gets removed). Let's see what happens when the cleanup function is implemented:

<Sandpack>

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

export default function ChatRoom() {
useEffect(() => {
const connection = createConnection();
connection.connect();
return () => connection.disconnect();
}, []);
return <h1>Welcome to the chat!</h1>;
}
```

```js chat.js
export function createConnection() {
// A real implementation would actually connect to the server
return {
connect() {
console.log('■ Connecting...');
},
disconnect() {
console.log('■ Disconnected.');
}
};
}
```

```css
input { display: block; margin-bottom: 20px; }
```

</Sandpack>

Now you get three console logs in development:

1. `"■ Connecting..."`
2. `"■ Disconnected."`
3. `"■ Connecting..."`

This is the correct behavior in development. By remounting your component, React verifies that navigating away and back would not break your code. Disconnecting and then connecting again is exactly what should happen! When you implement the cleanup well, there should be no user-visible difference between running the Effect once vs running it, cleaning it up, and running it again. There's an extra connect/disconnect call pair because React is probing your code for bugs in development. This is normal--don't try to make it go away!

In production, you would only see `"■ Connecting..."` printed once. Remounting components only happens in development to help you find Effects that need cleanup. You can turn off [Strict Mode](/reference/react/StrictMode) to opt out of the development behavior, but we recommend keeping it on. This lets you find many bugs like the one above.

How to handle the Effect firing twice in development?
{/*how-to-handle-the-effect-firing-twice-in-development*/}

React intentionally remounts your components in development to find bugs like in the last example. **The right question isn't "how to run an Effect once", but "how to fix my Effect so that it works after remounting".**

Usually, the answer is to implement the cleanup function. The cleanup function should stop or undo whatever the Effect was doing. The rule of thumb is that the user shouldn't be able to distinguish between the Effect running once (as in production) and a _setup → cleanup → setup_ sequence (as you'd see in development).

Most of the Effects you'll write will fit into one of the common patterns below.

Controlling non-React widgets {/*controlling-non-react-widgets*/}

Sometimes you need to add UI widgets that aren't written to React. For example, let's say you're adding a map component to your page. It has a `setZoomLevel()` method, and you'd like to keep the zoom level in sync with a `zoomLevel` state variable in your React code. Your Effect would look similar to this:

```js
useEffect(() => {
const map = mapRef.current;
map.setZoomLevel(zoomLevel);
}, [zoomLevel]);
```

Note that there is no cleanup needed in this case. In development, React will call the Effect twice, but this is not a problem because calling `setZoomLevel` twice with the same value does not do anything. It may be slightly slower, but this doesn't matter because it won't remount needlessly in production.

Some APIs may not allow you to call them twice in a row. For example, the [`showModal`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLDialogElement/showModal) method of the built-in [`<dialog>`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLDialogElement) element throws if you call it twice. Implement the cleanup function and make it close the dialog:

```js {4}
```

```
useEffect(() => {
const dialog = dialogRef.current;
dialog.showModal();
return () => dialog.close();
}, []);
```

In development, your Effect will call `showModal()`, then immediately `close()`, and then `showModal()` again. This has the same user-visible behavior as calling `showModal()` once, as you would see in production.

Subscribing to events {/*subscribing-to-events*/}

If your Effect subscribes to something, the cleanup function should unsubscribe:

```js {6}
useEffect(() => {
function handleScroll(e) {
console.log(window.scrollX, window.scrollY);
}
window.addEventListener('scroll', handleScroll);
return () => window.removeEventListener('scroll', handleScroll);
}, []);
```

In development, your Effect will call `addEventListener()`, then immediately `removeEventListener()`, and then `addEventListener()` again with the same handler. So there would be only one active subscription at a time. This has the same user-visible behavior as calling `addEventListener()` once, as in production.

Triggering animations {/*triggering-animations*/}

If your Effect animates something in, the cleanup function should reset the animation to the initial values:

```js {4-6}
useEffect(() => {
const node = ref.current;
node.style.opacity = 1; // Trigger the animation
return () => {
node.style.opacity = 0; // Reset to the initial value
};
}, []);
```

```

In development, opacity will be set to `1`, then to `0`, and then to `1` again. This should have the same user-visible behavior as setting it to `1` directly, which is what would happen in production. If you use a third-party animation library with support for tweening, your cleanup function should reset the timeline to its initial state.

### Fetching data {/*fetching-data*/}

If your Effect fetches something, the cleanup function should either [abort the fetch](https://developer.mozilla.org/en-US/docs/Web/API/AbortController) or ignore its result:

```js {2,6,13-15}
useEffect(() => {
let ignore = false;

async function startFetching() {
const json = await fetchTodos(userId);
if (!ignore) {
setTodos(json);
}
}

startFetching();

return () => {
ignore = true;
};
}, [userId]);
```

You can't "undo" a network request that already happened, but your cleanup function should ensure that the fetch that's _not relevant anymore_ does not keep affecting your application. If the `userId` changes from `'Alice'` to `'Bob'`, cleanup ensures that the `'Alice'` response is ignored even if it arrives after `'Bob'`.

In development, you will see two fetches in the Network tab. There is nothing wrong with that. With the approach above, the first Effect will immediately get cleaned up so its copy of the `ignore` variable will be set to `true`. So even though there is an extra request, it won't affect the state thanks to the `if (!ignore)` check.

In production, there will only be one request. If the second request in development is bothering you, the best approach is to use a solution that deduplicates requests and caches their responses between components:

```js
function TodoList() {
```
```

```
const todos = useSomeDataLibrary(`/api/user/${userId}/todos`);
// ...
```

This will not only improve the development experience, but also make your application feel faster. For example, the user pressing the Back button won't have to wait for some data to load again because it will be cached. You can either build such a cache yourself or use one of the many alternatives to manual fetching in Effects.

<DeepDive>

What are good alternatives to data fetching in Effects? {/*what-are-good-alternatives-to-data-fetching-in-effects*/}

Writing `fetch` calls inside Effects is a [popular way to fetch data](https://www.robinwieruch.de/react-hooks-fetch-data/), especially in fully client-side apps. This is, however, a very manual approach and it has significant downsides:

- **Effects don't run on the server.** This means that the initial server-rendered HTML will only include a loading state with no data. The client computer will have to download all JavaScript and render your app only to discover that now it needs to load the data. This is not very efficient.
- **Fetching directly in Effects makes it easy to create "network waterfalls".** You render the parent component, it fetches some data, renders the child components, and then they start fetching their data. If the network is not very fast, this is significantly slower than fetching all data in parallel.
- **Fetching directly in Effects usually means you don't preload or cache data.** For example, if the component unmounts and then mounts again, it would have to fetch the data again.
- **It's not very ergonomic.** There's quite a bit of boilerplate code involved when writing `fetch` calls in a way that doesn't suffer from bugs like [race conditions.](https://maxrozen.com/race-conditions-fetching-data-react-with-useeffect)

This list of downsides is not specific to React. It applies to fetching data on mount with any library. Like with routing, data fetching is not trivial to do well, so we recommend the following approaches:

- **If you use a [framework](/learn/start-a-new-react-project#production-grade-react-frameworks), use its built-in data fetching mechanism.** Modern React frameworks have integrated data fetching mechanisms that are efficient and don't suffer from the above pitfalls.
- **Otherwise, consider using or building a client-side cache.** Popular open source solutions include [React Query](https://tanstack.com/query/latest), [useSWR](https://swr.vercel.app/), and [React Router 6.4+.](https://beta.reactrouter.com/en/main/start/overview) You can build your own solution too, in which case you would use Effects under the hood, but add logic for deduplicating requests, caching responses, and avoiding network waterfalls (by preloading data or hoisting data requirements to routes).

You can continue fetching data directly in Effects if neither of these approaches suit you.

</DeepDive>

Sending analytics {/*sending-analytics*/}

Consider this code that sends an analytics event on the page visit:

```js
useEffect(() => {
logVisit(url); // Sends a POST request
}, [url]);
```

In development, `logVisit` will be called twice for every URL, so you might be tempted to try to fix that. **We recommend keeping this code as is.** Like with earlier examples, there is no *user-visible* behavior difference between running it once and running it twice. From a practical point of view, `logVisit` should not do anything in development because you don't want the logs from the development machines to skew the production metrics. Your component remounts every time you save its file, so it logs extra visits in development anyway.

In production, there will be no duplicate visit logs.

To debug the analytics events you're sending, you can deploy your app to a staging environment (which runs in production mode) or temporarily opt out of [Strict Mode](/reference/react/StrictMode) and its development-only remounting checks. You may also send analytics from the route change event handlers instead of Effects. For more precise analytics, [intersection observers](https://developer.mozilla.org/en-US/docs/Web/API/Intersection_Observer_API) can help track which components are in the viewport and how long they remain visible.

Not an Effect: Initializing the application {/*not-an-effect-initializing-the-application*/}

Some logic should only run once when the application starts. You can put it outside your components:

```js {2-3}
if (typeof window !== 'undefined') { // Check if we're running in the browser.
checkAuthToken();
loadDataFromLocalStorage();
}

function App() {
// ...
}
```

This guarantees that such logic only runs once after the browser loads the page.

Not an Effect: Buying a product {/*not-an-effect-buying-a-product*/}

Sometimes, even if you write a cleanup function, there's no way to prevent user-visible consequences of running the Effect twice. For example, maybe your Effect sends a POST request like buying a product:

```js {2-3}
useEffect(() => {
```

// ■ Wrong: This Effect fires twice in development, exposing a problem in the code.

fetch('/api/buy', { method: 'POST' });

}, []);
```

You wouldn't want to buy the product twice. However, this is also why you shouldn't put this logic in an Effect. What if the user goes to another page and then presses Back? Your Effect would run again. You don't want to buy the product when the user *visits* a page; you want to buy it when the user *clicks* the Buy button.

Buying is not caused by rendering; it's caused by a specific interaction. It should run only when the user presses the button. **Delete the Effect and move your `/api/buy` request into the Buy button event handler:**

```js {2-3}
function handleClick() {

// ■ Buying is an event because it is caused by a particular interaction.

fetch('/api/buy', { method: 'POST' });

}
```

This illustrates that if remounting breaks the logic of your application, this usually uncovers existing bugs. From the user's perspective, visiting a page shouldn't be different from visiting it, clicking a link, and pressing Back. React verifies that your components abide by this principle by remounting them once in development.

Putting it all together {/*putting-it-all-together*/}

This playground can help you "get a feel" for how Effects work in practice.

This example uses [`setTimeout`](https://developer.mozilla.org/en-US/docs/Web/API/setTimeout) to schedule a console log with the input text to appear three seconds after the Effect runs. The cleanup function cancels the pending timeout. Start by pressing "Mount the component":

<Sandpack>

```js
import { useState, useEffect } from 'react';

function Playground() {

const [text, setText] = useState('a');

useEffect(() => {

function onTimeout() {

console.log('■ ' + text);

}

```
console.log('■ Schedule "' + text + '" log');
const timeoutId = setTimeout(onTimeout, 3000);

return () => {
console.log('■ Cancel "' + text + '" log');
clearTimeout(timeoutId);
};
}, [text]);

return (
<>
<label>
What to log:{' '}
<input
value={text}
onChange={e => setText(e.target.value)}
/>
</label>
<h1>{text}</h1>
</>
);
}

export default function App() {
const [show, setShow] = useState(false);
return (
<>
<button onClick={() => setShow(!show)}>
{show ? 'Unmount' : 'Mount'} the component
</button>
{show && <hr />}
{show && <Playground />}
</>
);
}
```

</Sandpack>

You will see three logs at first: `Schedule "a" log`, `Cancel "a" log`, and `Schedule "a" log` again. Three second later there will also be a log saying `a`. As you learned earlier, the extra schedule/cancel pair is because React remounts the component once in development to verify that you've implemented cleanup well.

Now edit the input to say `abc`. If you do it fast enough, you'll see `Schedule "ab" log` immediately followed by `Cancel "ab" log` and `Schedule "abc" log`. **React always cleans up the previous render's Effect before the next render's Effect.** This is why even if you type into the input fast, there is at most one timeout scheduled at a time. Edit the input a few times and watch the console to get a feel for how Effects get cleaned up.

Type something into the input and then immediately press "Unmount the component". Notice how unmounting cleans up the last render's Effect. Here, it clears the last timeout before it has a chance to fire.

Finally, edit the component above and comment out the cleanup function so that the timeouts don't get cancelled. Try typing `abcde` fast. What do you expect to happen in three seconds? Will `console.log(text)` inside the timeout print the *latest* `text` and produce five `abcde` logs? Give it a try to check your intuition!

Three seconds later, you should see a sequence of logs (`a`, `ab`, `abc`, `abcd`, and `abcde`) rather than five `abcde` logs. **Each Effect "captures" the `text` value from its corresponding render.** It doesn't matter that the `text` state changed: an Effect from the render with `text = 'ab'` will always see `'ab'`. In other words, Effects from each render are isolated from each other. If you're curious how this works, you can read about [closures](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures).

<DeepDive>

Each render has its own Effects {/*each-render-has-its-own-effects*/}

You can think of `useEffect` as "attaching" a piece of behavior to the render output. Consider this Effect:

```js
export default function ChatRoom({ roomId }) {
useEffect(() => {
const connection = createConnection(roomId);
connection.connect();
return () => connection.disconnect();
}, [roomId]);

return <h1>Welcome to {roomId}!</h1>;
}
```

Let's see what exactly happens as the user navigates around the app.

Initial render {/*initial-render*/}

The user visits `<ChatRoom roomId="general" />`. Let's [mentally substitute](/learn/state-as-a-snapshot#rendering-takes-a-snapshot-in-time) `roomId` with `'general'`:

```js
// JSX for the first render (roomId = "general")
return <h1>Welcome to general!</h1>;
```

The Effect is *also* a part of the rendering output. The first render's Effect becomes:

```js
// Effect for the first render (roomId = "general")
() => {
const connection = createConnection('general');
connection.connect();
return () => connection.disconnect();
},
// Dependencies for the first render (roomId = "general")
['general']
```

React runs this Effect, which connects to the `'general'` chat room.

Re-render with same dependencies {/*re-render-with-same-dependencies*/}

Let's say `<ChatRoom roomId="general" />` re-renders. The JSX output is the same:

```js
// JSX for the second render (roomId = "general")
return <h1>Welcome to general!</h1>;
```

React sees that the rendering output has not changed, so it doesn't update the DOM.

The Effect from the second render looks like this:

```js
// Effect for the second render (roomId = "general")
() => {
const connection = createConnection('general');
connection.connect();
return () => connection.disconnect();
```

},

// Dependencies for the second render (roomId = "general")

['general']

```

React compares `['general']` from the second render with `['general']` from the first render. **Because all dependencies are the same, React *ignores* the Effect from the second render.** It never gets called.

#### Re-render with different dependencies {/*re-render-with-different-dependencies*/}

Then, the user visits `<ChatRoom roomId="travel" />`. This time, the component returns different JSX:

```js
// JSX for the third render (roomId = "travel")
return <h1>Welcome to travel!</h1>;
```

React updates the DOM to change `"Welcome to general"` into `"Welcome to travel"`.

The Effect from the third render looks like this:

```js
// Effect for the third render (roomId = "travel")
() => {
const connection = createConnection('travel');
connection.connect();
return () => connection.disconnect();
},
// Dependencies for the third render (roomId = "travel")
['travel']
```

React compares `['travel']` from the third render with `['general']` from the second render. One dependency is different: `Object.is('travel', 'general')` is `false`. The Effect can't be skipped.

Before React can apply the Effect from the third render, it needs to clean up the last Effect that _did_ run. The second render's Effect was skipped, so React needs to clean up the first render's Effect. If you scroll up to the first render, you'll see that its cleanup calls `disconnect()` on the connection that was created with `createConnection('general')`. This disconnects the app from the `'general'` chat room.

After that, React runs the third render's Effect. It connects to the `'travel'` chat room.

Unmount {/*unmount*/}
```

Finally, let's say the user navigates away, and the `ChatRoom` component unmounts. React runs the last Effect's cleanup function. The last Effect was from the third render. The third render's cleanup destroys the `createConnection('travel')` connection. So the app disconnects from the `'travel'` room.

Development-only behaviors {/*development-only-behaviors*/}

When [Strict Mode](/reference/react/StrictMode) is on, React remounts every component once after mount (state and DOM are preserved). This [helps you find Effects that need cleanup](#step-3-add-cleanup-if-needed) and exposes bugs like race conditions early. Additionally, React will remount the Effects whenever you save a file in development. Both of these behaviors are development-only.

</DeepDive>

<Recap>

- Unlike events, Effects are caused by rendering itself rather than a particular interaction.
- Effects let you synchronize a component with some external system (third-party API, network, etc).
- By default, Effects run after every render (including the initial one).
- React will skip the Effect if all of its dependencies have the same values as during the last render.
- You can't "choose" your dependencies. They are determined by the code inside the Effect.
- Empty dependency array (`[]`) corresponds to the component "mounting", i.e. being added to the screen.
- In Strict Mode, React mounts components twice (in development only!) to stress-test your Effects.
- If your Effect breaks because of remounting, you need to implement a cleanup function.
- React will call your cleanup function before the Effect runs next time, and during the unmount.

</Recap>

<Challenges>

Focus a field on mount {/*focus-a-field-on-mount*/}

In this example, the form renders a `<MyInput />` component.

Use the input's [`focus()`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/focus) method to make `MyInput` automatically focus when it appears on the screen. There is already a commented out implementation, but it doesn't quite work. Figure out why it doesn't work, and fix it. (If you're familiar with the `autoFocus` attribute, pretend that it does not exist: we are reimplementing the same functionality from scratch.)

<Sandpack>

```js MyInput.js active
import { useEffect, useRef } from 'react';

export default function MyInput({ value, onChange }) {
const ref = useRef(null);
```

```
// TODO: This doesn't quite work. Fix it.
// ref.current.focus()

return (
<input
ref={ref}
value={value}
onChange={onChange}
/>
);
}
```

```js App.js hidden
import { useState } from 'react';
import MyInput from './MyInput.js';

export default function Form() {
const [show, setShow] = useState(false);
const [name, setName] = useState('Taylor');
const [upper, setUpper] = useState(false);
return (
<>
<button onClick={() => setShow(s => !s)}>{show ? 'Hide' : 'Show'} form</button>
<br />
<hr />
{show && (
<>
<label>
Enter your name:
<MyInput
value={name}
onChange={e => setName(e.target.value)}
/>
</label>
<label>
<input
```

type="checkbox"

checked={upper}

onChange={e => setUpper(e.target.checked)}

/>

Make it uppercase

</label>

Hello, {upper ? name.toUpperCase() : name}

</>

)}

</>

);

}

```

```css

label {

display: block;

margin-top: 20px;

margin-bottom: 20px;

}

body {

min-height: 150px;

}

```

</Sandpack>

To verify that your solution works, press "Show form" and verify that the input receives focus (becomes highlighted and the cursor is placed inside). Press "Hide form" and "Show form" again. Verify the input is highlighted again.

`MyInput` should only focus _on mount_ rather than after every render. To verify that the behavior is right, press "Show form" and then repeatedly press the "Make it uppercase" checkbox. Clicking the checkbox should _not_ focus the input above it.

<Solution>

Calling `ref.current.focus()` during render is wrong because it is a *side effect*. Side effects should either be placed inside an event handler or be declared with `useEffect`. In this case, the side effect is _caused_ by the component appearing rather than by any specific interaction, so it makes sense to put it in an Effect.
```

To fix the mistake, wrap the `ref.current.focus()` call into an Effect declaration. Then, to ensure that this Effect runs only on mount rather than after every render, add the empty `[]` dependencies to it.

<Sandpack>

```js MyInput.js active
import { useEffect, useRef } from 'react';

export default function MyInput({ value, onChange }) {
const ref = useRef(null);

useEffect(() => {
ref.current.focus();
}, []);

return (
<input
ref={ref}
value={value}
onChange={onChange}
/>
);
}
```

```js App.js hidden
import { useState } from 'react';
import MyInput from './MyInput.js';

export default function Form() {
const [show, setShow] = useState(false);
const [name, setName] = useState('Taylor');
const [upper, setUpper] = useState(false);
return (
<>
<button onClick={() => setShow(s => !s)}>{show ? 'Hide' : 'Show'} form</button>
<br />
<hr />
{show && (
<>
```

<label>
Enter your name:
<MyInput
value={name}
onChange={e => setName(e.target.value)}
/>
</label>
<label>
<input
type="checkbox"
checked={upper}
onChange={e => setUpper(e.target.checked)}
/>
Make it uppercase
</label>
Hello, {upper ? name.toUpperCase() : name}
</>
)}
</>
);
}
```

```css
label {
display: block;
margin-top: 20px;
margin-bottom: 20px;
}

body {
min-height: 150px;
}
```

</Sandpack>

</Solution>
```

Focus a field conditionally {/*focus-a-field-conditionally*/}

This form renders two `<MyInput />` components.

Press "Show form" and notice that the second field automatically gets focused. This is because both of the `<MyInput />` components try to focus the field inside. When you call `focus()` for two input fields in a row, the last one always "wins".

Let's say you want to focus the first field. The first `MyInput` component now receives a boolean `shouldFocus` prop set to `true`. Change the logic so that `focus()` is only called if the `shouldFocus` prop received by `MyInput` is `true`.

<Sandpack>

```js MyInput.js active
import { useEffect, useRef } from 'react';

export default function MyInput({ shouldFocus, value, onChange }) {
const ref = useRef(null);

// TODO: call focus() only if shouldFocus is true.
useEffect(() => {
ref.current.focus();
}, []);

return (
<input
ref={ref}
value={value}
onChange={onChange}
/>
);
}
```

```js App.js hidden
import { useState } from 'react';
import MyInput from './MyInput.js';

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

```
const [upper, setUpper] = useState(false);
const name = firstName + ' ' + lastName;
return (
<>
<button onClick={() => setShow(s => !s)}>{show ? 'Hide' : 'Show'} form</button>
<br />
<hr />
{show && (
<>
<label>
Enter your first name:
<MyInput
value={firstName}
onChange={e => setFirstName(e.target.value)}
shouldFocus={true}
/>
</label>
<label>
Enter your last name:
<MyInput
value={lastName}
onChange={e => setLastName(e.target.value)}
shouldFocus={false}
/>
</label>
<p>Hello, <b>{upper ? name.toUpperCase() : name}</b></p>
</>
)}
</>
);
}
```

```css
label {
display: block;
```

margin-top: 20px;

margin-bottom: 20px;

}

body {

min-height: 150px;

}

```

</Sandpack>

To verify your solution, press "Show form" and "Hide form" repeatedly. When the form appears, only the *first* input should get focused. This is because the parent component renders the first input with `shouldFocus={true}` and the second input with `shouldFocus={false}`. Also check that both inputs still work and you can type into both of them.

<Hint>

You can't declare an Effect conditionally, but your Effect can include conditional logic.

</Hint>

<Solution>

Put the conditional logic inside the Effect. You will need to specify `shouldFocus` as a dependency because you are using it inside the Effect. (This means that if some input's `shouldFocus` changes from `false` to `true`, it will focus after mount.)

<Sandpack>

```js MyInput.js active

import { useEffect, useRef } from 'react';

export default function MyInput({ shouldFocus, value, onChange }) {

const ref = useRef(null);

useEffect(() => {

if (shouldFocus) {

ref.current.focus();

}

}, [shouldFocus]);

return (

<input

ref={ref}

value={value}

```
onChange={onChange}
/>
);
}
```

```js App.js hidden
import { useState } from 'react';
import MyInput from './MyInput.js';

export default function Form() {
const [show, setShow] = useState(false);
const [firstName, setFirstName] = useState('Taylor');
const [lastName, setLastName] = useState('Swift');
const [upper, setUpper] = useState(false);
const name = firstName + ' ' + lastName;
return (
<>
<button onClick={() => setShow(s => !s)}>{show ? 'Hide' : 'Show'} form</button>
<br />
<hr />
{show && (
<>
<label>
Enter your first name:
<MyInput
value={firstName}
onChange={e => setFirstName(e.target.value)}
shouldFocus={true}
/>
</label>
<label>
Enter your last name:
<MyInput
value={lastName}
onChange={e => setLastName(e.target.value)}
shouldFocus={false}
```

/>

</label>

Hello, {upper ? name.toUpperCase() : name}

</>

)}

</>

);

}

```

```css

label {

display: block;

margin-top: 20px;

margin-bottom: 20px;

}

body {

min-height: 150px;

}

```

</Sandpack>

</Solution>

Fix an interval that fires twice {/*fix-an-interval-that-fires-twice*/}

This `Counter` component displays a counter that should increment every second. On mount, it calls [`setInterval`.](https://developer.mozilla.org/en-US/docs/Web/API/setInterval) This causes `onTick` to run every second. The `onTick` function increments the counter.

However, instead of incrementing once per second, it increments twice. Why is that? Find the cause of the bug and fix it.

<Hint>

Keep in mind that `setInterval` returns an interval ID, which you can pass to [`clearInterval`](https://developer.mozilla.org/en-US/docs/Web/API/clearInterval) to stop the interval.

</Hint>

<Sandpack>

```js Counter.js active

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

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

useEffect(() => {
function onTick() {
setCount(c => c + 1);
}

setInterval(onTick, 1000);
}, []);

return <h1>{count}</h1>;
}
```

```js App.js hidden
import { useState } from 'react';
import Counter from './Counter.js';

export default function Form() {
const [show, setShow] = useState(false);
return (
<>
<button onClick={() => setShow(s => !s)}>{show ? 'Hide' : 'Show'} counter</button>
<br />
<hr />
{show && <Counter />}
</>
);
}
```

```css
label {
display: block;
margin-top: 20px;
margin-bottom: 20px;
}
```

```
body {
min-height: 150px;
}
```

</Sandpack>

<Solution>

When [Strict Mode](/reference/react/StrictMode) is on (like in the sandboxes on this site), React remounts each component once in development. This causes the interval to be set up twice, and this is why each second the counter increments twice.

However, React's behavior is not the *cause* of the bug: the bug already exists in the code. React's behavior makes the bug more noticeable. The real cause is that this Effect starts a process but doesn't provide a way to clean it up.

To fix this code, save the interval ID returned by `setInterval`, and implement a cleanup function with [`clearInterval`](https://developer.mozilla.org/en-US/docs/Web/API/clearInterval):

<Sandpack>

```js Counter.js active
import { useState, useEffect } from 'react';

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

useEffect(() => {
function onTick() {
setCount(c => c + 1);
}

const intervalId = setInterval(onTick, 1000);
return () => clearInterval(intervalId);
}, []);

return <h1>{count}</h1>;
}
```

```js App.js hidden
import { useState } from 'react';
import Counter from './Counter.js';

export default function App() {
```

```
const [show, setShow] = useState(false);
return (
<>
<button onClick={() => setShow(s => !s)}>{show ? 'Hide' : 'Show'} counter</button>
<br />
<hr />
{show && <Counter />}
</>
);
}
```

```css
label {
display: block;
margin-top: 20px;
margin-bottom: 20px;
}

body {
min-height: 150px;
}
```

</Sandpack>

In development, React will still remount your component once to verify that you've implemented cleanup well. So there will be a `setInterval` call, immediately followed by `clearInterval`, and `setInterval` again. In production, there will be only one `setInterval` call. The user-visible behavior in both cases is the same: the counter increments once per second.

</Solution>

Fix fetching inside an Effect {/*fix-fetching-inside-an-effect*/}

This component shows the biography for the selected person. It loads the biography by calling an asynchronous function `fetchBio(person)` on mount and whenever `person` changes. That asynchronous function returns a [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) which eventually resolves to a string. When fetching is done, it calls `setBio` to display that string under the select box.

<Sandpack>

```js App.js
```

````
import { useState, useEffect } from 'react';
import { fetchBio } from './api.js';

export default function Page() {
const [person, setPerson] = useState('Alice');
const [bio, setBio] = useState(null);

useEffect(() => {
setBio(null);
fetchBio(person).then(result => {
setBio(result);
});
}, [person]);

return (
<>
<select value={person} onChange={e => {
setPerson(e.target.value);
}}>
<option value="Alice">Alice</option>
<option value="Bob">Bob</option>
<option value="Taylor">Taylor</option>
</select>
<hr />
<p><i>{bio ?? 'Loading...'}</i></p>
</>
);
}
```

```js api.js hidden
export async function fetchBio(person) {
const delay = person === 'Bob' ? 2000 : 200;
return new Promise(resolve => {
setTimeout(() => {
resolve('This is ' + person + ''s bio.');
}, delay);
})
````

}

```

</Sandpack>

There is a bug in this code. Start by selecting "Alice". Then select "Bob" and then immediately after that select "Taylor". If you do this fast enough, you will notice that bug: Taylor is selected, but the paragraph below says "This is Bob's bio."

Why does this happen? Fix the bug inside this Effect.

<Hint>

If an Effect fetches something asynchronously, it usually needs cleanup.

</Hint>

<Solution>

To trigger the bug, things need to happen in this order:

- Selecting `'Bob'` triggers `fetchBio('Bob')`
- Selecting `'Taylor'` triggers `fetchBio('Taylor')`
- **Fetching `'Taylor'` completes *before* fetching `'Bob'`**
- The Effect from the `'Taylor'` render calls `setBio('This is Taylor’s bio')`
- Fetching `'Bob'` completes
- The Effect from the `'Bob'` render calls `setBio('This is Bob’s bio')`

This is why you see Bob's bio even though Taylor is selected. Bugs like this are called [race conditions](https://en.wikipedia.org/wiki/Race_condition) because two asynchronous operations are "racing" with each other, and they might arrive in an unexpected order.

To fix this race condition, add a cleanup function:

<Sandpack>

```js App.js
import { useState, useEffect } from 'react';
import { fetchBio } from './api.js';

export default function Page() {
const [person, setPerson] = useState('Alice');
const [bio, setBio] = useState(null);
useEffect(() => {
let ignore = false;

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

return (
<>
<select value={person} onChange={e => {
setPerson(e.target.value);
}}>
<option value="Alice">Alice</option>
<option value="Bob">Bob</option>
<option value="Taylor">Taylor</option>
</select>
<hr />
<p><i>{bio ?? 'Loading...'}</i></p>
</>
);
}
```

```js api.js hidden
export async function fetchBio(person) {
const delay = person === 'Bob' ? 2000 : 200;
return new Promise(resolve => {
setTimeout(() => {
resolve('This is ' + person + ''s bio.');
}, delay);
})
}
```

```

</Sandpack>

Each render's Effect has its own `ignore` variable. Initially, the `ignore` variable is set to `false`. However, if an Effect gets cleaned up (such as when you select a different person), its `ignore` variable becomes `true`. So now it doesn't matter in which order the requests complete. Only the last person's Effect will have `ignore` set to `false`, so it will call `setBio(result)`. Past Effects have been cleaned up, so the `if (!ignore)` check will prevent them from calling `setBio`:

- Selecting `'Bob'` triggers `fetchBio('Bob')`
- Selecting `'Taylor'` triggers `fetchBio('Taylor')` **and cleans up the previous (Bob's) Effect**
- Fetching `'Taylor'` completes *before* fetching `'Bob'`
- The Effect from the `'Taylor'` render calls `setBio('This is Taylor’s bio')`
- Fetching `'Bob'` completes
- The Effect from the `'Bob'` render **does not do anything because its `ignore` flag was set to `true`**

In addition to ignoring the result of an outdated API call, you can also use [`AbortController`](https://developer.mozilla.org/en-US/docs/Web/API/AbortController) to cancel the requests that are no longer needed. However, by itself this is not enough to protect against race conditions. More asynchronous steps could be chained after the fetch, so using an explicit flag like `ignore` is the most reliable way to fix this type of problems.

</Solution>

</Challenges>

---

title: Updating Objects in State

---

<Intro>

State can hold any kind of JavaScript value, including objects. But you shouldn't change objects that you hold in the React state directly. Instead, when you want to update an object, you need to create a new one (or make a copy of an existing one), and then set the state to use that copy.

</Intro>

<YouWillLearn>

- How to correctly update an object in React state
- How to update a nested object without mutating it
- What immutability is, and how not to break it
- How to make object copying less repetitive with Immer

</YouWillLearn>
```

What's a mutation? {/*whats-a-mutation*/}

You can store any kind of JavaScript value in state.

```js
const [x, setX] = useState(0);
```

So far you've been working with numbers, strings, and booleans. These kinds of JavaScript values are "immutable", meaning unchangeable or "read-only". You can trigger a re-render to _replace_ a value:

```js
setX(5);
```

The `x` state changed from `0` to `5`, but the _number `0` itself_ did not change. It's not possible to make any changes to the built-in primitive values like numbers, strings, and booleans in JavaScript.

Now consider an object in state:

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

Technically, it is possible to change the contents of _the object itself_. **This is called a mutation:**

```js
position.x = 5;
```

However, although objects in React state are technically mutable, you should treat them **as if** they were immutable--like numbers, booleans, and strings. Instead of mutating them, you should always replace them.

Treat state as read-only {/*treat-state-as-read-only*/}

In other words, you should **treat any JavaScript object that you put into state as read-only.**

This example holds an object in state to represent the current pointer position. The red dot is supposed to move when you touch or move the cursor over the preview area. But the dot stays in the initial position:

<Sandpack>

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

export default function MovingDot() {
```

```
const [position, setPosition] = useState({
x: 0,
y: 0
});
return (
<div
onPointerMove={e => {
position.x = e.clientX;
position.y = e.clientY;
}}
style={{
position: 'relative',
width: '100vw',
height: '100vh',
}}>
<div style={{
position: 'absolute',
backgroundColor: 'red',
borderRadius: '50%',
transform: `translate(${position.x}px, ${position.y}px)`,
left: -10,
top: -10,
width: 20,
height: 20,
}} />
</div>
);
}
```

```css
body { margin: 0; padding: 0; height: 250px; }
```

</Sandpack>

The problem is with this bit of code.

```js
onPointerMove={e => {
position.x = e.clientX;
position.y = e.clientY;
}}
```

This code modifies the object assigned to `position` from [the previous render.](/learn/state-as-a-snapshot#rendering-takes-a-snapshot-in-time) But without using the state setting function, React has no idea that object has changed. So React does not do anything in response. It's like trying to change the order after you've already eaten the meal. While mutating state can work in some cases, we don't recommend it. You should treat the state value you have access to in a render as read-only.

To actually [trigger a re-render](/learn/state-as-a-snapshot#setting-state-triggers-renders) in this case, **create a *new* object and pass it to the state setting function:**

```js
onPointerMove={e => {
setPosition({
x: e.clientX,
y: e.clientY
});
}}
```

With `setPosition`, you're telling React:

* Replace `position` with this new object
* And render this component again

Notice how the red dot now follows your pointer when you touch or hover over the preview area:

<Sandpack>

```js
import { useState } from 'react';
export default function MovingDot() {
const [position, setPosition] = useState({
x: 0,
y: 0
});
return (
```

<div
onPointerMove={e => {
setPosition({
x: e.clientX,
y: e.clientY
});
}}
style={{
position: 'relative',
width: '100vw',
height: '100vh',
}}>
<div style={{
position: 'absolute',
backgroundColor: 'red',
borderRadius: '50%',
transform: `translate(${position.x}px, ${position.y}px)`,
left: -10,
top: -10,
width: 20,
height: 20,
}} />

);
}
```

```css
body { margin: 0; padding: 0; height: 250px; }
```

</Sandpack>

<DeepDive>

Local mutation is fine {/*local-mutation-is-fine*/}

Code like this is a problem because it modifies an *existing* object in state:
```

```js
position.x = e.clientX;
position.y = e.clientY;
```

But code like this is **absolutely fine** because you're mutating a fresh object you have *just created*:

```js
const nextPosition = {};
nextPosition.x = e.clientX;
nextPosition.y = e.clientY;
setPosition(nextPosition);
```

In fact, it is completely equivalent to writing this:

```js
setPosition({
x: e.clientX,
y: e.clientY
});
```

Mutation is only a problem when you change *existing* objects that are already in state. Mutating an object you've just created is okay because *no other code references it yet.* Changing it isn't going to accidentally impact something that depends on it. This is called a "local mutation". You can even do local mutation [while rendering.](/learn/keeping-components-pure#local-mutation-your-components-little-secret) Very convenient and completely okay!

</DeepDive>

Copying objects with the spread syntax {/*copying-objects-with-the-spread-syntax*/}

In the previous example, the `position` object is always created fresh from the current cursor position. But often, you will want to include *existing* data as a part of the new object you're creating. For example, you may want to update *only one* field in a form, but keep the previous values for all other fields.

These input fields don't work because the `onChange` handlers mutate the state:

<Sandpack>

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

```
export default function Form() {
const [person, setPerson] = useState({
firstName: 'Barbara',
lastName: 'Hepworth',
email: 'bhepworth@sculpture.com'
});

function handleFirstNameChange(e) {
person.firstName = e.target.value;
}

function handleLastNameChange(e) {
person.lastName = e.target.value;
}

function handleEmailChange(e) {
person.email = e.target.value;
}

return (
<>
<label>
First name:
<input
value={person.firstName}
onChange={handleFirstNameChange}
/>
</label>
<label>
Last name:
<input
value={person.lastName}
onChange={handleLastNameChange}
/>
</label>
<label>
Email:
<input
```

value={person.email}

onChange={handleEmailChange}

/>

</label>

{person.firstName}{' '}

{person.lastName}{' '}

({person.email})

</>

);

}

```

```css

label { display: block; }

input { margin-left: 5px; margin-bottom: 5px; }

```

</Sandpack>

For example, this line mutates the state from a past render:

```js

person.firstName = e.target.value;

```

The reliable way to get the behavior you're looking for is to create a new object and pass it to `setPerson`. But here, you want to also **copy the existing data into it** because only one of the fields has changed:

```js

setPerson({

firstName: e.target.value, // New first name from the input

lastName: person.lastName,

email: person.email

});

```

You can use the `...` [object spread](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax#spread_in_object_literals) syntax so that you don't need to copy every
```

property separately.

```js
setPerson({
...person, // Copy the old fields
firstName: e.target.value // But override this one
});
```

Now the form works!

Notice how you didn't declare a separate state variable for each input field. For large forms, keeping all data grouped in an object is very convenient--as long as you update it correctly!

<Sandpack>

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

export default function Form() {
const [person, setPerson] = useState({
firstName: 'Barbara',
lastName: 'Hepworth',
email: 'bhepworth@sculpture.com'
});

function handleFirstNameChange(e) {
setPerson({
...person,
firstName: e.target.value
});
}

function handleLastNameChange(e) {
setPerson({
...person,
lastName: e.target.value
});
}

function handleEmailChange(e) {
setPerson({
```

```
...person,
email: e.target.value
});
}

return (
<>
<label>
First name:
<input
value={person.firstName}
onChange={handleFirstNameChange}
/>
</label>
<label>
Last name:
<input
value={person.lastName}
onChange={handleLastNameChange}
/>
</label>
<label>
Email:
<input
value={person.email}
onChange={handleEmailChange}
/>
</label>
<p>
{person.firstName}{' '}
{person.lastName}{' '}
({person.email})
</p>
</>
);
}
```

```

```css
label { display: block; }
input { margin-left: 5px; margin-bottom: 5px; }
```

</Sandpack>

Note that the `...` spread syntax is "shallow"--it only copies things one level deep. This makes it fast, but it also means that if you want to update a nested property, you'll have to use it more than once.

<DeepDive>

Using a single event handler for multiple fields
{/*using-a-single-event-handler-for-multiple-fields*/}

You can also use the `[` and `]` braces inside your object definition to specify a property with dynamic name. Here is the same example, but with a single event handler instead of three different ones:

<Sandpack>

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

export default function Form() {
const [person, setPerson] = useState({
firstName: 'Barbara',
lastName: 'Hepworth',
email: 'bhepworth@sculpture.com'
});

function handleChange(e) {
setPerson({
...person,
[e.target.name]: e.target.value
});
}

return (
<>
<label>
First name:
<input

````
name="firstName"
value={person.firstName}
onChange={handleChange}
/>
</label>
<label>
Last name:
<input
name="lastName"
value={person.lastName}
onChange={handleChange}
/>
</label>
<label>
Email:
<input
name="email"
value={person.email}
onChange={handleChange}
/>
</label>
<p>
{person.firstName}{' '}
{person.lastName}{' '}
({person.email})
</p>
</>
);
}
```

```css
label { display: block; }
input { margin-left: 5px; margin-bottom: 5px; }
```

</Sandpack>

Here, `e.target.name` refers to the `name` property given to the `<input>` DOM element.

</DeepDive>

Updating a nested object {/*updating-a-nested-object*/}

Consider a nested object structure like this:

```js
const [person, setPerson] = useState({
name: 'Niki de Saint Phalle',
artwork: {
title: 'Blue Nana',
city: 'Hamburg',
image: 'https://i.imgur.com/Sd1AgUOm.jpg',
}
});
```

If you wanted to update `person.artwork.city`, it's clear how to do it with mutation:

```js
person.artwork.city = 'New Delhi';
```

But in React, you treat state as immutable! In order to change `city`, you would first need to produce the new `artwork` object (pre-populated with data from the previous one), and then produce the new `person` object which points at the new `artwork`:

```js
const nextArtwork = { ...person.artwork, city: 'New Delhi' };
const nextPerson = { ...person, artwork: nextArtwork };
setPerson(nextPerson);
```

Or, written as a single function call:

```js
setPerson({
...person, // Copy other fields
artwork: { // but replace the artwork
...person.artwork, // with the same one
city: 'New Delhi' // but in New Delhi!
```

}
});
```

This gets a bit wordy, but it works fine for many cases:

<Sandpack>

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

export default function Form() {
const [person, setPerson] = useState({
name: 'Niki de Saint Phalle',
artwork: {
title: 'Blue Nana',
city: 'Hamburg',
image: 'https://i.imgur.com/Sd1AgUOm.jpg',
}
});

function handleNameChange(e) {
setPerson({
...person,
name: e.target.value
});
}

function handleTitleChange(e) {
setPerson({
...person,
artwork: {
...person.artwork,
title: e.target.value
}
});
}

function handleCityChange(e) {
setPerson({

```
...person,
artwork: {
...person.artwork,
city: e.target.value
}
});
}

function handleImageChange(e) {
setPerson({
...person,
artwork: {
...person.artwork,
image: e.target.value
}
});
}

return (
<>
<label>
Name:
<input
value={person.name}
onChange={handleNameChange}
/>
</label>
<label>
Title:
<input
value={person.artwork.title}
onChange={handleTitleChange}
/>
</label>
<label>
City:
<input
```

value={person.artwork.city}
onChange={handleCityChange}
/>
</label>
<label>
Image:
<input
value={person.artwork.image}
onChange={handleImageChange}
/>
</label>

{person.artwork.title}
{' by '}
{person.name}

(located in {person.artwork.city})

<img
src={person.artwork.image}
alt={person.artwork.title}
/>
</>
);
}
```

```css
label { display: block; }
input { margin-left: 5px; margin-bottom: 5px; }
img { width: 200px; height: 200px; }
```

</Sandpack>

<DeepDive>

Objects are not really nested {/*objects-are-not-really-nested*/}
```

An object like this appears "nested" in code:

```js
let obj = {
name: 'Niki de Saint Phalle',
artwork: {
title: 'Blue Nana',
city: 'Hamburg',
image: 'https://i.imgur.com/Sd1AgUOm.jpg',
}
};
```

However, "nesting" is an inaccurate way to think about how objects behave. When the code executes, there is no such thing as a "nested" object. You are really looking at two different objects:

```js
let obj1 = {
title: 'Blue Nana',
city: 'Hamburg',
image: 'https://i.imgur.com/Sd1AgUOm.jpg',
};

let obj2 = {
name: 'Niki de Saint Phalle',
artwork: obj1
};
```

The `obj1` object is not "inside" `obj2`. For example, `obj3` could "point" at `obj1` too:

```js
let obj1 = {
title: 'Blue Nana',
city: 'Hamburg',
image: 'https://i.imgur.com/Sd1AgUOm.jpg',
};

let obj2 = {
name: 'Niki de Saint Phalle',
```

```
artwork: obj1
};

let obj3 = {
name: 'Copycat',
artwork: obj1
};
```

If you were to mutate `obj3.artwork.city`, it would affect both `obj2.artwork.city` and `obj1.city`. This is because `obj3.artwork`, `obj2.artwork`, and `obj1` are the same object. This is difficult to see when you think of objects as "nested". Instead, they are separate objects "pointing" at each other with properties.

</DeepDive>

Write concise update logic with Immer {/*write-concise-update-logic-with-immer*/}

If your state is deeply nested, you might want to consider [flattening it.](/learn/choosing-the-state-structure#avoid-deeply-nested-state) But, if you don't want to change your state structure, you might prefer a shortcut to nested spreads. [Immer](https://github.com/immerjs/use-immer) is a popular library that lets you write using the convenient but mutating syntax and takes care of producing the copies for you. With Immer, the code you write looks like you are "breaking the rules" and mutating an object:

```js
updatePerson(draft => {
draft.artwork.city = 'Lagos';
});
```

But unlike a regular mutation, it doesn't overwrite the past state!

<DeepDive>

How does Immer work? {/*how-does-immer-work*/}

The `draft` provided by Immer is a special type of object, called a [Proxy](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy), that "records" what you do with it. This is why you can mutate it freely as much as you like! Under the hood, Immer figures out which parts of the `draft` have been changed, and produces a completely new object that contains your edits.

</DeepDive>

To try Immer:

1. Run `npm install use-immer` to add Immer as a dependency
2. Then replace `import { useState } from 'react'` with `import { useImmer } from 'use-immer'`

Here is the above example converted to Immer:

<Sandpack>

```js
import { useImmer } from 'use-immer';

export default function Form() {
const [person, updatePerson] = useImmer({
name: 'Niki de Saint Phalle',
artwork: {
title: 'Blue Nana',
city: 'Hamburg',
image: 'https://i.imgur.com/Sd1AgUOm.jpg',
}
});

function handleNameChange(e) {
updatePerson(draft => {
draft.name = e.target.value;
});
}

function handleTitleChange(e) {
updatePerson(draft => {
draft.artwork.title = e.target.value;
});
}

function handleCityChange(e) {
updatePerson(draft => {
draft.artwork.city = e.target.value;
});
}

function handleImageChange(e) {
updatePerson(draft => {
draft.artwork.image = e.target.value;
});
}
```

```
return (
<>
<label>
Name:
<input
value={person.name}
onChange={handleNameChange}
/>
</label>
<label>
Title:
<input
value={person.artwork.title}
onChange={handleTitleChange}
/>
</label>
<label>
City:
<input
value={person.artwork.city}
onChange={handleCityChange}
/>
</label>
<label>
Image:
<input
value={person.artwork.image}
onChange={handleImageChange}
/>
</label>
<p>
<i>{person.artwork.title}</i>
{' by '}
{person.name}
<br />
```

(located in {person.artwork.city})
</p>
<img
src={person.artwork.image}
alt={person.artwork.title}
/>
</>
);
}
```

```json package.json
{
"dependencies": {
"immer": "1.7.3",
"react": "latest",
"react-dom": "latest",
"react-scripts": "latest",
"use-immer": "0.5.1"
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
}
}
```

```css
label { display: block; }
input { margin-left: 5px; margin-bottom: 5px; }
img { width: 200px; height: 200px; }
```

</Sandpack>
```

Notice how much more concise the event handlers have become. You can mix and match `useState` and `useImmer` in a single component as much as you like. Immer is a great way to keep the update handlers concise, especially if there's nesting in your state, and copying objects leads to repetitive code.

<DeepDive>

Why is mutating state not recommended in React?
{/*why-is-mutating-state-not-recommended-in-react*/}

There are a few reasons:

* **Debugging:** If you use `console.log` and don't mutate state, your past logs won't get clobbered by the more recent state changes. So you can clearly see how state has changed between renders.

* **Optimizations:** Common React [optimization strategies](/reference/react/memo) rely on skipping work if previous props or state are the same as the next ones. If you never mutate state, it is very fast to check whether there were any changes. If `prevObj === obj`, you can be sure that nothing could have changed inside of it.

* **New Features:** The new React features we're building rely on state being [treated like a snapshot.](/learn/state-as-a-snapshot) If you're mutating past versions of state, that may prevent you from using the new features.

* **Requirement Changes:** Some application features, like implementing Undo/Redo, showing a history of changes, or letting the user reset a form to earlier values, are easier to do when nothing is mutated. This is because you can keep past copies of state in memory, and reuse them when appropriate. If you start with a mutative approach, features like this can be difficult to add later on.

* **Simpler Implementation:** Because React does not rely on mutation, it does not need to do anything special with your objects. It does not need to hijack their properties, always wrap them into Proxies, or do other work at initialization as many "reactive" solutions do. This is also why React lets you put any object into state--no matter how large--without additional performance or correctness pitfalls.

In practice, you can often "get away" with mutating state in React, but we strongly advise you not to do that so that you can use new React features developed with this approach in mind. Future contributors and perhaps even your future self will thank you!

</DeepDive>

<Recap>

* Treat all state in React as immutable.

* When you store objects in state, mutating them will not trigger renders and will change the state in previous render "snapshots".

* Instead of mutating an object, create a *new* version of it, and trigger a re-render by setting state to it.

* You can use the `{...obj, something: 'newValue'}` object spread syntax to create copies of objects.

* Spread syntax is shallow: it only copies one level deep.

* To update a nested object, you need to create copies all the way up from the place you're updating.

* To reduce repetitive copying code, use Immer.

</Recap>

<Challenges>

Fix incorrect state updates {/*fix-incorrect-state-updates*/}

This form has a few bugs. Click the button that increases the score a few times. Notice that it does not increase. Then edit the first name, and notice that the score has suddenly "caught up" with your changes. Finally, edit the last name, and notice that the score has disappeared completely.

Your task is to fix all of these bugs. As you fix them, explain why each of them happens.

<Sandpack>

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

export default function Scoreboard() {
const [player, setPlayer] = useState({
firstName: 'Ranjani',
lastName: 'Shettar',
score: 10,
});

function handlePlusClick() {
player.score++;
}

function handleFirstNameChange(e) {
setPlayer({
...player,
firstName: e.target.value,
});
}

function handleLastNameChange(e) {
setPlayer({
lastName: e.target.value
});
}

return (
<>
<label>
Score: <b>{player.score}</b>
```

{' '}
<button onClick={handlePlusClick}>
+1
</button>
</label>
<label>
First name:
<input
value={player.firstName}
onChange={handleFirstNameChange}
/>
</label>
<label>
Last name:
<input
value={player.lastName}
onChange={handleLastNameChange}
/>
</label>
</>
);
}
```

```css
label { display: block; margin-bottom: 10px; }
input { margin-left: 5px; margin-bottom: 5px; }
```

</Sandpack>

<Solution>

Here is a version with both bugs fixed:

<Sandpack>

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

```
export default function Scoreboard() {
const [player, setPlayer] = useState({
firstName: 'Ranjani',
lastName: 'Shettar',
score: 10,
});

function handlePlusClick() {
setPlayer({
...player,
score: player.score + 1,
});
}

function handleFirstNameChange(e) {
setPlayer({
...player,
firstName: e.target.value,
});
}

function handleLastNameChange(e) {
setPlayer({
...player,
lastName: e.target.value
});
}

return (
<>
<label>
Score: <b>{player.score}</b>
{' '}
<button onClick={handlePlusClick}>
+1
</button>
</label>
<label>
```

First name:

<input

value={player.firstName}

onChange={handleFirstNameChange}

/>

</label>

<label>

Last name:

<input

value={player.lastName}

onChange={handleLastNameChange}

/>

</label>

</>

);

}

```

```css

label { display: block; }

input { margin-left: 5px; margin-bottom: 5px; }

```

</Sandpack>

The problem with `handlePlusClick` was that it mutated the `player` object. As a result, React did not know that there's a reason to re-render, and did not update the score on the screen. This is why, when you edited the first name, the state got updated, triggering a re-render which _also_ updated the score on the screen.

The problem with `handleLastNameChange` was that it did not copy the existing `...player` fields into the new object. This is why the score got lost after you edited the last name.

</Solution>

Find and fix the mutation {/*find-and-fix-the-mutation*/}

There is a draggable box on a static background. You can change the box's color using the select input.

But there is a bug. If you move the box first, and then change its color, the background (which isn't supposed to move!) will "jump" to the box position. But this should not happen: the `Background`'s `position` prop is set to `initialPosition`, which is `{ x: 0, y: 0 }`. Why is the background moving after the color change?
```

Find the bug and fix it.

<Hint>

If something unexpected changes, there is a mutation. Find the mutation in `App.js` and fix it.

</Hint>

<Sandpack>

```js App.js
import { useState } from 'react';
import Background from './Background.js';
import Box from './Box.js';

const initialPosition = {
x: 0,
y: 0
};

export default function Canvas() {
const [shape, setShape] = useState({
color: 'orange',
position: initialPosition
});

function handleMove(dx, dy) {
shape.position.x += dx;
shape.position.y += dy;
}

function handleColorChange(e) {
setShape({
...shape,
color: e.target.value
});
}

return (
<>
<select
value={shape.color}
```

```
onChange={handleColorChange}
>
<option value="orange">orange</option>
<option value="lightpink">lightpink</option>
<option value="aliceblue">aliceblue</option>
</select>
<Background
position={initialPosition}
/>
<Box
color={shape.color}
position={shape.position}
onMove={handleMove}
>
Drag me!
</Box>
</>
);
}
```

```js Box.js
import { useState } from 'react';

export default function Box({
children,
color,
position,
onMove
}) {
const [
lastCoordinates,
setLastCoordinates
] = useState(null);

function handlePointerDown(e) {
e.target.setPointerCapture(e.pointerId);
```

```
setLastCoordinates({
x: e.clientX,
y: e.clientY,
});
}

function handlePointerMove(e) {
if (lastCoordinates) {
setLastCoordinates({
x: e.clientX,
y: e.clientY,
});
const dx = e.clientX - lastCoordinates.x;
const dy = e.clientY - lastCoordinates.y;
onMove(dx, dy);
}
}

function handlePointerUp(e) {
setLastCoordinates(null);
}

return (
<div
onPointerDown={handlePointerDown}
onPointerMove={handlePointerMove}
onPointerUp={handlePointerUp}
style={{
width: 100,
height: 100,
cursor: 'grab',
backgroundColor: color,
position: 'absolute',
border: '1px solid black',
display: 'flex',
justifyContent: 'center',
alignItems: 'center',
```

transform: `translate(
${position.x}px,
${position.y}px
)`,
}}
>{children}
);
}
```

```js Background.js
export default function Background({
position
}) {
return (
<div style={{
position: 'absolute',
transform: `translate(
${position.x}px,
${position.y}px
)`,
width: 250,
height: 250,
backgroundColor: 'rgba(200, 200, 0, 0.2)',
}} />
);
};
```

```css
body { height: 280px; }
select { margin-bottom: 10px; }
```

</Sandpack>

<Solution>
```

The problem was in the mutation inside `handleMove`. It mutated `shape.position`, but that's the same object that `initialPosition` points at. This is why both the shape and the background move. (It's a mutation, so the change doesn't reflect on the screen until an unrelated update--the color change--triggers a re-render.)

The fix is to remove the mutation from `handleMove`, and use the spread syntax to copy the shape. Note that `+=` is a mutation, so you need to rewrite it to use a regular `+` operation.

<Sandpack>

```js App.js
import { useState } from 'react';
import Background from './Background.js';
import Box from './Box.js';

const initialPosition = {
x: 0,
y: 0
};

export default function Canvas() {
const [shape, setShape] = useState({
color: 'orange',
position: initialPosition
});

function handleMove(dx, dy) {
setShape({
...shape,
position: {
x: shape.position.x + dx,
y: shape.position.y + dy,
}
});
}

function handleColorChange(e) {
setShape({
...shape,
color: e.target.value
});
```

```
}

return (
<>
<select
value={shape.color}
onChange={handleColorChange}
>
<option value="orange">orange</option>
<option value="lightpink">lightpink</option>
<option value="aliceblue">aliceblue</option>
</select>
<Background
position={initialPosition}
/>
<Box
color={shape.color}
position={shape.position}
onMove={handleMove}
>
Drag me!
</Box>
</>
);
}
```

```js Box.js
import { useState } from 'react';

export default function Box({
children,
color,
position,
onMove
}) {
const [
```

```
lastCoordinates,
setLastCoordinates
] = useState(null);

function handlePointerDown(e) {
e.target.setPointerCapture(e.pointerId);
setLastCoordinates({
x: e.clientX,
y: e.clientY,
});
}

function handlePointerMove(e) {
if (lastCoordinates) {
setLastCoordinates({
x: e.clientX,
y: e.clientY,
});
const dx = e.clientX - lastCoordinates.x;
const dy = e.clientY - lastCoordinates.y;
onMove(dx, dy);
}
}

function handlePointerUp(e) {
setLastCoordinates(null);
}

return (
<div
onPointerDown={handlePointerDown}
onPointerMove={handlePointerMove}
onPointerUp={handlePointerUp}
style={{
width: 100,
height: 100,
cursor: 'grab',
backgroundColor: color,
```

```
position: 'absolute',
border: '1px solid black',
display: 'flex',
justifyContent: 'center',
alignItems: 'center',
transform: `translate(
${position.x}px,
${position.y}px
)`,
}}
>{children}</div>
);
}
```

```js Background.js
export default function Background({
position
}) {
return (
<div style={{
position: 'absolute',
transform: `translate(
${position.x}px,
${position.y}px
)`,
width: 250,
height: 250,
backgroundColor: 'rgba(200, 200, 0, 0.2)',
}} />
);
};
```

```css
body { height: 280px; }
select { margin-bottom: 10px; }
```

```

</Sandpack>

</Solution>

#### Update an object with Immer {/*update-an-object-with-immer*/}

This is the same buggy example as in the previous challenge. This time, fix the mutation by using Immer. For your convenience, `useImmer` is already imported, so you need to change the `shape` state variable to use it.

<Sandpack>

```js App.js
import { useState } from 'react';
import { useImmer } from 'use-immer';
import Background from './Background.js';
import Box from './Box.js';

const initialPosition = {
x: 0,
y: 0
};

export default function Canvas() {
const [shape, setShape] = useState({
color: 'orange',
position: initialPosition
});

function handleMove(dx, dy) {
shape.position.x += dx;
shape.position.y += dy;
}

function handleColorChange(e) {
setShape({
...shape,
color: e.target.value
});
}
```
```

```
return (
<>
<select
value={shape.color}
onChange={handleColorChange}
>
<option value="orange">orange</option>
<option value="lightpink">lightpink</option>
<option value="aliceblue">aliceblue</option>
</select>
<Background
position={initialPosition}
/>
<Box
color={shape.color}
position={shape.position}
onMove={handleMove}
>
Drag me!
</Box>
</>
);
}
```

```js Box.js
import { useState } from 'react';

export default function Box({
children,
color,
position,
onMove
}) {
const [
lastCoordinates,
setLastCoordinates
```

```
] = useState(null);

function handlePointerDown(e) {
e.target.setPointerCapture(e.pointerId);
setLastCoordinates({
x: e.clientX,
y: e.clientY,
});
}

function handlePointerMove(e) {
if (lastCoordinates) {
setLastCoordinates({
x: e.clientX,
y: e.clientY,
});
const dx = e.clientX - lastCoordinates.x;
const dy = e.clientY - lastCoordinates.y;
onMove(dx, dy);
}
}

function handlePointerUp(e) {
setLastCoordinates(null);
}

return (
<div
onPointerDown={handlePointerDown}
onPointerMove={handlePointerMove}
onPointerUp={handlePointerUp}
style={{
width: 100,
height: 100,
cursor: 'grab',
backgroundColor: color,
position: 'absolute',
border: '1px solid black',
```

````
display: 'flex',
justifyContent: 'center',
alignItems: 'center',
transform: `translate(
${position.x}px,
${position.y}px
)`,
}}
>{children}</div>
);
}
```

```js Background.js
export default function Background({
position
}) {
return (
<div style={{
position: 'absolute',
transform: `translate(
${position.x}px,
${position.y}px
)`,
width: 250,
height: 250,
backgroundColor: 'rgba(200, 200, 0, 0.2)',
}} />
);
};
```

```css
body { height: 280px; }
select { margin-bottom: 10px; }
```
````
````

```json package.json
{
"dependencies": {
"immer": "1.7.3",
"react": "latest",
"react-dom": "latest",
"react-scripts": "latest",
"use-immer": "0.5.1"
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
}
}
```

</Sandpack>

<Solution>

This is the solution rewritten with Immer. Notice how the event handlers are written in a mutating fashion, but the bug does not occur. This is because under the hood, Immer never mutates the existing objects.

<Sandpack>

```js App.js
import { useImmer } from 'use-immer';
import Background from './Background.js';
import Box from './Box.js';

const initialPosition = {
x: 0,
y: 0
};

export default function Canvas() {
const [shape, updateShape] = useImmer({
color: 'orange',
```

```
position: initialPosition
});

function handleMove(dx, dy) {
updateShape(draft => {
draft.position.x += dx;
draft.position.y += dy;
});
}

function handleColorChange(e) {
updateShape(draft => {
draft.color = e.target.value;
});
}

return (
<>
<select
value={shape.color}
onChange={handleColorChange}
>
<option value="orange">orange</option>
<option value="lightpink">lightpink</option>
<option value="aliceblue">aliceblue</option>
</select>
<Background
position={initialPosition}
/>
<Box
color={shape.color}
position={shape.position}
onMove={handleMove}
>
Drag me!
</Box>
</>
```

);
}
```

```js Box.js
import { useState } from 'react';

export default function Box({
children,
color,
position,
onMove
}) {
const [
lastCoordinates,
setLastCoordinates
] = useState(null);

function handlePointerDown(e) {
e.target.setPointerCapture(e.pointerId);
setLastCoordinates({
x: e.clientX,
y: e.clientY,
});
}

function handlePointerMove(e) {
if (lastCoordinates) {
setLastCoordinates({
x: e.clientX,
y: e.clientY,
});
const dx = e.clientX - lastCoordinates.x;
const dy = e.clientY - lastCoordinates.y;
onMove(dx, dy);
}
}

function handlePointerUp(e) {

```
setLastCoordinates(null);
}

return (
<div
onPointerDown={handlePointerDown}
onPointerMove={handlePointerMove}
onPointerUp={handlePointerUp}
style={{
width: 100,
height: 100,
cursor: 'grab',
backgroundColor: color,
position: 'absolute',
border: '1px solid black',
display: 'flex',
justifyContent: 'center',
alignItems: 'center',
transform: `translate(
${position.x}px,
${position.y}px
)`,
}}
>{children}</div>
);
}
```

```js Background.js
export default function Background({
position
}) {
return (
<div style={{
position: 'absolute',
transform: `translate(
${position.x}px,
```

```
${position.y}px
)`,
width: 250,
height: 250,
backgroundColor: 'rgba(200, 200, 0, 0.2)',
}} />
);
};
```

```css
body { height: 280px; }
select { margin-bottom: 10px; }
```

```json package.json
{
"dependencies": {
"immer": "1.7.3",
"react": "latest",
"react-dom": "latest",
"react-scripts": "latest",
"use-immer": "0.5.1"
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
}
}
```

</Sandpack>

</Solution>

</Challenges>

---

title: "State: A Component's Memory"

<Intro>

Components often need to change what's on the screen as a result of an interaction. Typing into the form should update the input field, clicking "next" on an image carousel should change which image is displayed, clicking "buy" should put a product in the shopping cart. Components need to "remember" things: the current input value, the current image, the shopping cart. In React, this kind of component-specific memory is called *state*.

</Intro>

<YouWillLearn>

* How to add a state variable with the [`useState`](/reference/react/useState) Hook

* What pair of values the `useState` Hook returns

* How to add more than one state variable

* Why state is called local

</YouWillLearn>

When a regular variable isn’t enough {/*when-a-regular-variable-isnt-enough*/}

Here's a component that renders a sculpture image. Clicking the "Next" button should show the next sculpture by changing the `index` to `1`, then `2`, and so on. However, this **won't work** (you can try it!):

<Sandpack>

```js
import { sculptureList } from './data.js';

export default function Gallery() {
let index = 0;

function handleClick() {
index = index + 1;
}

let sculpture = sculptureList[index];
return (
<>
<button onClick={handleClick}>
Next
</button>
```

<h2>

{sculpture.name}

by {sculpture.artist}

</h2>

<h3>

({index + 1} of {sculptureList.length})

</h3>

<img

src={sculpture.url}

alt={sculpture.alt}

/>

{sculpture.description}

</>

);

}

```

```js data.js

export const sculptureList = [{

name: 'Homenaje a la Neurocirugía',

artist: 'Marta Colvin Andrade',

description: 'Although Colvin is predominantly known for abstract themes that allude to pre-Hispanic symbols, this gigantic sculpture, an homage to neurosurgery, is one of her most recognizable public art pieces.',

url: 'https://i.imgur.com/Mx7dA2Y.jpg',

alt: 'A bronze statue of two crossed hands delicately holding a human brain in their fingertips.'

}, {

name: 'Floralis Genérica',

artist: 'Eduardo Catalano',

description: 'This enormous (75 ft. or 23m) silver flower is located in Buenos Aires. It is designed to move, closing its petals in the evening or when strong winds blow and opening them in the morning.',

url: 'https://i.imgur.com/ZF6s192m.jpg',

alt: 'A gigantic metallic flower sculpture with reflective mirror-like petals and strong stamens.'

}, {

name: 'Eternal Presence',

artist: 'John Woodrow Wilson',

description: 'Wilson was known for his preoccupation with equality, social justice, as well as the essential and spiritual qualities of humankind. This massive (7ft. or 2,13m) bronze represents what he described as "a symbolic Black presence infused with a sense of universal humanity."',

url: 'https://i.imgur.com/aTtVpES.jpg',

alt: 'The sculpture depicting a human head seems ever-present and solemn. It radiates calm and serenity.'

}, {

name: 'Moai',

artist: 'Unknown Artist',

description: 'Located on the Easter Island, there are 1,000 moai, or extant monumental statues, created by the early Rapa Nui people, which some believe represented deified ancestors.',

url: 'https://i.imgur.com/RCwLEoQm.jpg',

alt: 'Three monumental stone busts with the heads that are disproportionately large with somber faces.'

}, {

name: 'Blue Nana',

artist: 'Niki de Saint Phalle',

description: 'The Nanas are triumphant creatures, symbols of femininity and maternity. Initially, Saint Phalle used fabric and found objects for the Nanas, and later on introduced polyester to achieve a more vibrant effect.',

url: 'https://i.imgur.com/Sd1AgUOm.jpg',

alt: 'A large mosaic sculpture of a whimsical dancing female figure in a colorful costume emanating joy.'

}, {

name: 'Ultimate Form',

artist: 'Barbara Hepworth',

description: 'This abstract bronze sculpture is a part of The Family of Man series located at Yorkshire Sculpture Park. Hepworth chose not to create literal representations of the world but developed abstract forms inspired by people and landscapes.',

url: 'https://i.imgur.com/2heNQDcm.jpg',

alt: 'A tall sculpture made of three elements stacked on each other reminding of a human figure.'

}, {

name: 'Cavaliere',

artist: 'Lamidi Olonade Fakeye',

description: "Descended from four generations of woodcarvers, Fakeye's work blended traditional and contemporary Yoruba themes.",

url: 'https://i.imgur.com/wIdGuZwm.png',

alt: 'An intricate wood sculpture of a warrior with a focused face on a horse adorned with patterns.'

}, {

name: 'Big Bellies',

artist: 'Alina Szapocznikow',

description: "Szapocznikow is known for her sculptures of the fragmented body as a metaphor for the fragility and impermanence of youth and beauty. This sculpture depicts two very realistic large bellies stacked on top of each other, each around five feet (1,5m) tall.",

url: 'https://i.imgur.com/AlHTAdDm.jpg',

alt: 'The sculpture reminds a cascade of folds, quite different from bellies in classical sculptures.'

}, {

name: 'Terracotta Army',

artist: 'Unknown Artist',

description: 'The Terracotta Army is a collection of terracotta sculptures depicting the armies of Qin Shi Huang, the first Emperor of China. The army consisted of more than 8,000 soldiers, 130 chariots with 520 horses, and 150 cavalry horses.',

url: 'https://i.imgur.com/HMFmH6m.jpg',

alt: '12 terracotta sculptures of solemn warriors, each with a unique facial expression and armor.'

}, {

name: 'Lunar Landscape',

artist: 'Louise Nevelson',

description: 'Nevelson was known for scavenging objects from New York City debris, which she would later assemble into monumental constructions. In this one, she used disparate parts like a bedpost, juggling pin, and seat fragment, nailing and gluing them into boxes that reflect the influence of Cubism’s geometric abstraction of space and form.',

url: 'https://i.imgur.com/rN7hY6om.jpg',

alt: 'A black matte sculpture where the individual elements are initially indistinguishable.'

}, {

name: 'Aureole',

artist: 'Ranjani Shettar',

description: 'Shettar merges the traditional and the modern, the natural and the industrial. Her art focuses on the relationship between man and nature. Her work was described as compelling both abstractly and figuratively, gravity defying, and a "fine synthesis of unlikely materials."',

url: 'https://i.imgur.com/okTpbHhm.jpg',

alt: 'A pale wire-like sculpture mounted on concrete wall and descending on the floor. It appears light.'

}, {

name: 'Hippos',

artist: 'Taipei Zoo',

description: 'The Taipei Zoo commissioned a Hippo Square featuring submerged hippos at play.',

url: 'https://i.imgur.com/6o5Vuyu.jpg',

alt: 'A group of bronze hippo sculptures emerging from the sett sidewalk as if they were swimming.'

}];
```

```css
h2 { margin-top: 10px; margin-bottom: 0; }
h3 {
margin-top: 5px;
font-weight: normal;
font-size: 100%;
}
img { width: 120px; height: 120px; }
button {
display: block;
margin-top: 10px;
margin-bottom: 10px;
}
```

</Sandpack>

The `handleClick` event handler is updating a local variable, `index`. But two things prevent that change from being visible:

1. **Local variables don't persist between renders.** When React renders this component a second time, it renders it from scratch—it doesn't consider any changes to the local variables.
2. **Changes to local variables won't trigger renders.** React doesn't realize it needs to render the component again with the new data.

To update a component with new data, two things need to happen:

1. **Retain** the data between renders.
2. **Trigger** React to render the component with new data (re-rendering).

The [`useState`](/reference/react/useState) Hook provides those two things:

1. A **state variable** to retain the data between renders.
2. A **state setter function** to update the variable and trigger React to render the component again.

Adding a state variable {/*adding-a-state-variable*/}

To add a state variable, import `useState` from React at the top of the file:

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

```

Then, replace this line:

```js
let index = 0;
```

with

```js
const [index, setIndex] = useState(0);
```

`index` is a state variable and `setIndex` is the setter function.

> The `[` and `]` syntax here is called [array destructuring](https://javascript.info/destructuring-assignment) and it lets you read values from an array. The array returned by `useState` always has exactly two items.

This is how they work together in `handleClick`:

```js
function handleClick() {
setIndex(index + 1);
}
```

Now clicking the "Next" button switches the current sculpture:

<Sandpack>

```js
import { useState } from 'react';
import { sculptureList } from './data.js';

export default function Gallery() {
const [index, setIndex] = useState(0);

function handleClick() {
setIndex(index + 1);
}

let sculpture = sculptureList[index];
return (
```
```

```
<>
<button onClick={handleClick}>
Next
</button>
<h2>
<i>{sculpture.name} </i>
by {sculpture.artist}
</h2>
<h3>
({index + 1} of {sculptureList.length})
</h3>
<img
src={sculpture.url}
alt={sculpture.alt}
/>
<p>
{sculpture.description}
</p>
</>
);
}
```

```js data.js
export const sculptureList = [{
name: 'Homenaje a la Neurocirugía',
artist: 'Marta Colvin Andrade',
description: 'Although Colvin is predominantly known for abstract themes that allude to pre-Hispanic symbols, this gigantic sculpture, an homage to neurosurgery, is one of her most recognizable public art pieces.',
url: 'https://i.imgur.com/Mx7dA2Y.jpg',
alt: 'A bronze statue of two crossed hands delicately holding a human brain in their fingertips.'
}, {
name: 'Floralis Genérica',
artist: 'Eduardo Catalano',
description: 'This enormous (75 ft. or 23m) silver flower is located in Buenos Aires. It is designed to move, closing its petals in the evening or when strong winds blow and opening them in the morning.',
```

url: 'https://i.imgur.com/ZF6s192m.jpg',

alt: 'A gigantic metallic flower sculpture with reflective mirror-like petals and strong stamens.'

}, {

name: 'Eternal Presence',

artist: 'John Woodrow Wilson',

description: 'Wilson was known for his preoccupation with equality, social justice, as well as the essential and spiritual qualities of humankind. This massive (7ft. or 2,13m) bronze represents what he described as "a symbolic Black presence infused with a sense of universal humanity."',

url: 'https://i.imgur.com/aTtVpES.jpg',

alt: 'The sculpture depicting a human head seems ever-present and solemn. It radiates calm and serenity.'

}, {

name: 'Moai',

artist: 'Unknown Artist',

description: 'Located on the Easter Island, there are 1,000 moai, or extant monumental statues, created by the early Rapa Nui people, which some believe represented deified ancestors.',

url: 'https://i.imgur.com/RCwLEoQm.jpg',

alt: 'Three monumental stone busts with the heads that are disproportionately large with somber faces.'

}, {

name: 'Blue Nana',

artist: 'Niki de Saint Phalle',

description: 'The Nanas are triumphant creatures, symbols of femininity and maternity. Initially, Saint Phalle used fabric and found objects for the Nanas, and later on introduced polyester to achieve a more vibrant effect.',

url: 'https://i.imgur.com/Sd1AgUOm.jpg',

alt: 'A large mosaic sculpture of a whimsical dancing female figure in a colorful costume emanating joy.'

}, {

name: 'Ultimate Form',

artist: 'Barbara Hepworth',

description: 'This abstract bronze sculpture is a part of The Family of Man series located at Yorkshire Sculpture Park. Hepworth chose not to create literal representations of the world but developed abstract forms inspired by people and landscapes.',

url: 'https://i.imgur.com/2heNQDcm.jpg',

alt: 'A tall sculpture made of three elements stacked on each other reminding of a human figure.'

}, {

name: 'Cavaliere',

artist: 'Lamidi Olonade Fakeye',

description: "Descended from four generations of woodcarvers, Fakeye's work blended traditional and contemporary Yoruba themes.",

url: 'https://i.imgur.com/wIdGuZwm.png',

alt: 'An intricate wood sculpture of a warrior with a focused face on a horse adorned with patterns.'

}, {

name: 'Big Bellies',

artist: 'Alina Szapocznikow',

description: "Szapocznikow is known for her sculptures of the fragmented body as a metaphor for the fragility and impermanence of youth and beauty. This sculpture depicts two very realistic large bellies stacked on top of each other, each around five feet (1,5m) tall.",

url: 'https://i.imgur.com/AlHTAdDm.jpg',

alt: 'The sculpture reminds a cascade of folds, quite different from bellies in classical sculptures.'

}, {

name: 'Terracotta Army',

artist: 'Unknown Artist',

description: 'The Terracotta Army is a collection of terracotta sculptures depicting the armies of Qin Shi Huang, the first Emperor of China. The army consisted of more than 8,000 soldiers, 130 chariots with 520 horses, and 150 cavalry horses.',

url: 'https://i.imgur.com/HMFmH6m.jpg',

alt: '12 terracotta sculptures of solemn warriors, each with a unique facial expression and armor.'

}, {

name: 'Lunar Landscape',

artist: 'Louise Nevelson',

description: 'Nevelson was known for scavenging objects from New York City debris, which she would later assemble into monumental constructions. In this one, she used disparate parts like a bedpost, juggling pin, and seat fragment, nailing and gluing them into boxes that reflect the influence of Cubism’s geometric abstraction of space and form.',

url: 'https://i.imgur.com/rN7hY6om.jpg',

alt: 'A black matte sculpture where the individual elements are initially indistinguishable.'

}, {

name: 'Aureole',

artist: 'Ranjani Shettar',

description: 'Shettar merges the traditional and the modern, the natural and the industrial. Her art focuses on the relationship between man and nature. Her work was described as compelling both abstractly and figuratively, gravity defying, and a "fine synthesis of unlikely materials."',

url: 'https://i.imgur.com/okTpbHhm.jpg',

alt: 'A pale wire-like sculpture mounted on concrete wall and descending on the floor. It appears light.'

}, {

name: 'Hippos',

artist: 'Taipei Zoo',

description: 'The Taipei Zoo commissioned a Hippo Square featuring submerged hippos at play.',

url: 'https://i.imgur.com/6o5Vuyu.jpg',

alt: 'A group of bronze hippo sculptures emerging from the sett sidewalk as if they were swimming.'

}];

```

```css

h2 { margin-top: 10px; margin-bottom: 0; }

h3 {

margin-top: 5px;

font-weight: normal;

font-size: 100%;

}

img { width: 120px; height: 120px; }

button {

display: block;

margin-top: 10px;

margin-bottom: 10px;

}

```

</Sandpack>

Meet your first Hook {/*meet-your-first-hook*/}

In React, `useState`, as well as any other function starting with "`use`", is called a Hook.

Hooks are special functions that are only available while React is [rendering](/learn/render-and-commit#step-1-trigger-a-render) (which we'll get into in more detail on the next page). They let you "hook into" different React features.

State is just one of those features, but you will meet the other Hooks later.

<Pitfall>

Hooks—functions starting with `use`—can only be called at the top level of your components or [your own Hooks.](/learn/reusing-logic-with-custom-hooks) You can't call Hooks inside conditions, loops, or other nested functions. Hooks are functions, but it's helpful to think of them as unconditional declarations about your component's needs. You "use" React features at the top of your component similar to how you "import" modules at the top of your file.

</Pitfall>
```

Anatomy of `useState` {/*anatomy-of-usestate*/}

When you call [`useState`](/reference/react/useState), you are telling React that you want this component to remember something:

```js
const [index, setIndex] = useState(0);
```

In this case, you want React to remember `index`.

<Note>

The convention is to name this pair like `const [something, setSomething]`. You could name it anything you like, but conventions make things easier to understand across projects.

</Note>

The only argument to `useState` is the **initial value** of your state variable. In this example, the `index`'s initial value is set to `0` with `useState(0)`.

Every time your component renders, `useState` gives you an array containing two values:

1. The **state variable** (`index`) with the value you stored.
2. The **state setter function** (`setIndex`) which can update the state variable and trigger React to render the component again.

Here's how that happens in action:

```js
const [index, setIndex] = useState(0);
```

1. **Your component renders the first time.** Because you passed `0` to `useState` as the initial value for `index`, it will return `[0, setIndex]`. React remembers `0` is the latest state value.
2. **You update the state.** When a user clicks the button, it calls `setIndex(index + 1)`. `index` is `0`, so it's `setIndex(1)`. This tells React to remember `index` is `1` now and triggers another render.
3. **Your component's second render.** React still sees `useState(0)`, but because React *remembers* that you set `index` to `1`, it returns `[1, setIndex]` instead.
4. And so on!

Giving a component multiple state variables {/*giving-a-component-multiple-state-variables*/}

You can have as many state variables of as many types as you like in one component. This component has two state variables, a number `index` and a boolean `showMore` that's toggled when you click "Show details":

<Sandpack>

```js
import { useState } from 'react';
import { sculptureList } from './data.js';

export default function Gallery() {
const [index, setIndex] = useState(0);
const [showMore, setShowMore] = useState(false);

function handleNextClick() {
setIndex(index + 1);
}

function handleMoreClick() {
setShowMore(!showMore);
}

let sculpture = sculptureList[index];
return (
<>
<button onClick={handleNextClick}>
Next
</button>
<h2>
<i>{sculpture.name} </i>
by {sculpture.artist}
</h2>
<h3>
({index + 1} of {sculptureList.length})
</h3>
<button onClick={handleMoreClick}>
{showMore ? 'Hide' : 'Show'} details
</button>
{showMore && <p>{sculpture.description}</p>}
<img
src={sculpture.url}
alt={sculpture.alt}
/>
</>
```

);

}

```

```js data.js

export const sculptureList = [{

name: 'Homenaje a la Neurocirugía',

artist: 'Marta Colvin Andrade',

description: 'Although Colvin is predominantly known for abstract themes that allude to pre-Hispanic symbols, this gigantic sculpture, an homage to neurosurgery, is one of her most recognizable public art pieces.',

url: 'https://i.imgur.com/Mx7dA2Y.jpg',

alt: 'A bronze statue of two crossed hands delicately holding a human brain in their fingertips.'

}, {

name: 'Floralis Genérica',

artist: 'Eduardo Catalano',

description: 'This enormous (75 ft. or 23m) silver flower is located in Buenos Aires. It is designed to move, closing its petals in the evening or when strong winds blow and opening them in the morning.',

url: 'https://i.imgur.com/ZF6s192m.jpg',

alt: 'A gigantic metallic flower sculpture with reflective mirror-like petals and strong stamens.'

}, {

name: 'Eternal Presence',

artist: 'John Woodrow Wilson',

description: 'Wilson was known for his preoccupation with equality, social justice, as well as the essential and spiritual qualities of humankind. This massive (7ft. or 2,13m) bronze represents what he described as "a symbolic Black presence infused with a sense of universal humanity."',

url: 'https://i.imgur.com/aTtVpES.jpg',

alt: 'The sculpture depicting a human head seems ever-present and solemn. It radiates calm and serenity.'

}, {

name: 'Moai',

artist: 'Unknown Artist',

description: 'Located on the Easter Island, there are 1,000 moai, or extant monumental statues, created by the early Rapa Nui people, which some believe represented deified ancestors.',

url: 'https://i.imgur.com/RCwLEoQm.jpg',

alt: 'Three monumental stone busts with the heads that are disproportionately large with somber faces.'

}, {

name: 'Blue Nana',

artist: 'Niki de Saint Phalle',

description: 'The Nanas are triumphant creatures, symbols of femininity and maternity. Initially, Saint Phalle used fabric and found objects for the Nanas, and later on introduced polyester to achieve a more vibrant effect.',

url: 'https://i.imgur.com/Sd1AgUOm.jpg',

alt: 'A large mosaic sculpture of a whimsical dancing female figure in a colorful costume emanating joy.'

}, {

name: 'Ultimate Form',

artist: 'Barbara Hepworth',

description: 'This abstract bronze sculpture is a part of The Family of Man series located at Yorkshire Sculpture Park. Hepworth chose not to create literal representations of the world but developed abstract forms inspired by people and landscapes.',

url: 'https://i.imgur.com/2heNQDcm.jpg',

alt: 'A tall sculpture made of three elements stacked on each other reminding of a human figure.'

}, {

name: 'Cavaliere',

artist: 'Lamidi Olonade Fakeye',

description: "Descended from four generations of woodcarvers, Fakeye's work blended traditional and contemporary Yoruba themes.",

url: 'https://i.imgur.com/wIdGuZwm.png',

alt: 'An intricate wood sculpture of a warrior with a focused face on a horse adorned with patterns.'

}, {

name: 'Big Bellies',

artist: 'Alina Szapocznikow',

description: "Szapocznikow is known for her sculptures of the fragmented body as a metaphor for the fragility and impermanence of youth and beauty. This sculpture depicts two very realistic large bellies stacked on top of each other, each around five feet (1,5m) tall.",

url: 'https://i.imgur.com/AlHTAdDm.jpg',

alt: 'The sculpture reminds a cascade of folds, quite different from bellies in classical sculptures.'

}, {

name: 'Terracotta Army',

artist: 'Unknown Artist',

description: 'The Terracotta Army is a collection of terracotta sculptures depicting the armies of Qin Shi Huang, the first Emperor of China. The army consisted of more than 8,000 soldiers, 130 chariots with 520 horses, and 150 cavalry horses.',

url: 'https://i.imgur.com/HMFmH6m.jpg',

alt: '12 terracotta sculptures of solemn warriors, each with a unique facial expression and armor.'

}, {

name: 'Lunar Landscape',

artist: 'Louise Nevelson',

description: 'Nevelson was known for scavenging objects from New York City debris, which she would later assemble into monumental constructions. In this one, she used disparate parts like a bedpost, juggling pin, and seat fragment, nailing and gluing them into boxes that reflect the influence of Cubism’s geometric abstraction of space and form.',

url: 'https://i.imgur.com/rN7hY6om.jpg',

alt: 'A black matte sculpture where the individual elements are initially indistinguishable.'

}, {

name: 'Aureole',

artist: 'Ranjani Shettar',

description: 'Shettar merges the traditional and the modern, the natural and the industrial. Her art focuses on the relationship between man and nature. Her work was described as compelling both abstractly and figuratively, gravity defying, and a "fine synthesis of unlikely materials."',

url: 'https://i.imgur.com/okTpbHhm.jpg',

alt: 'A pale wire-like sculpture mounted on concrete wall and descending on the floor. It appears light.'

}, {

name: 'Hippos',

artist: 'Taipei Zoo',

description: 'The Taipei Zoo commissioned a Hippo Square featuring submerged hippos at play.',

url: 'https://i.imgur.com/6o5Vuyu.jpg',

alt: 'A group of bronze hippo sculptures emerging from the sett sidewalk as if they were swimming.'

}];

```

```css

h2 { margin-top: 10px; margin-bottom: 0; }

h3 {

margin-top: 5px;

font-weight: normal;

font-size: 100%;

}

img { width: 120px; height: 120px; }

button {

display: block;

margin-top: 10px;

margin-bottom: 10px;

}

```

</Sandpack>

It is a good idea to have multiple state variables if their state is unrelated, like `index` and `showMore` in this example. But if you find that you often change two state variables together, it might be easier to combine them into one. For example, if you have a form with many fields, it's more convenient to have a single state variable that holds an object than state variable per field. Read [Choosing the State Structure](/learn/choosing-the-state-structure) for more tips.

<DeepDive>

How does React know which state to return? {/*how-does-react-know-which-state-to-return*/}

You might have noticed that the `useState` call does not receive any information about *which* state variable it refers to. There is no "identifier" that is passed to `useState`, so how does it know which of the state variables to return? Does it rely on some magic like parsing your functions? The answer is no.

Instead, to enable their concise syntax, Hooks **rely on a stable call order on every render of the same component.** This works well in practice because if you follow the rule above ("only call Hooks at the top level"), Hooks will always be called in the same order. Additionally, a [linter plugin](https://www.npmjs.com/package/eslint-plugin-react-hooks) catches most mistakes.

Internally, React holds an array of state pairs for every component. It also maintains the current pair index, which is set to `0` before rendering. Each time you call `useState`, React gives you the next state pair and increments the index. You can read more about this mechanism in [React Hooks: Not Magic, Just Arrays.](https://medium.com/@ryardley/react-hooks-not-magic-just-arrays-cd4f1857236e)

This example **doesn't use React** but it gives you an idea of how `useState` works internally:

<Sandpack>

```js index.js active
let componentHooks = [];
let currentHookIndex = 0;

// How useState works inside React (simplified).
function useState(initialState) {
let pair = componentHooks[currentHookIndex];
if (pair) {
// This is not the first render,
// so the state pair already exists.
// Return it and prepare for next Hook call.
currentHookIndex++;
return pair;
}
```
```

```
// This is the first time we're rendering,
// so create a state pair and store it.
pair = [initialState, setState];

function setState(nextState) {
// When the user requests a state change,
// put the new value into the pair.
pair[0] = nextState;
updateDOM();
}

// Store the pair for future renders
// and prepare for the next Hook call.
componentHooks[currentHookIndex] = pair;
currentHookIndex++;
return pair;
}

function Gallery() {
// Each useState() call will get the next pair.
const [index, setIndex] = useState(0);
const [showMore, setShowMore] = useState(false);

function handleNextClick() {
setIndex(index + 1);
}

function handleMoreClick() {
setShowMore(!showMore);
}

let sculpture = sculptureList[index];
// This example doesn't use React, so
// return an output object instead of JSX.
return {
onNextClick: handleNextClick,
onMoreClick: handleMoreClick,
header: `${sculpture.name} by ${sculpture.artist}`,
counter: `${index + 1} of ${sculptureList.length}`,
```

```
more: `${showMore ? 'Hide' : 'Show'} details`,
description: showMore ? sculpture.description : null,
imageSrc: sculpture.url,
imageAlt: sculpture.alt
};
}

function updateDOM() {
// Reset the current Hook index
// before rendering the component.
currentHookIndex = 0;
let output = Gallery();

// Update the DOM to match the output.
// This is the part React does for you.
nextButton.onclick = output.onNextClick;
header.textContent = output.header;
moreButton.onclick = output.onMoreClick;
moreButton.textContent = output.more;
image.src = output.imageSrc;
image.alt = output.imageAlt;
if (output.description !== null) {
description.textContent = output.description;
description.style.display = '';
} else {
description.style.display = 'none';
}
}

let nextButton = document.getElementById('nextButton');
let header = document.getElementById('header');
let moreButton = document.getElementById('moreButton');
let description = document.getElementById('description');
let image = document.getElementById('image');
let sculptureList = [{
name: 'Homenaje a la Neurocirugía',
artist: 'Marta Colvin Andrade',
```

description: 'Although Colvin is predominantly known for abstract themes that allude to pre-Hispanic symbols, this gigantic sculpture, an homage to neurosurgery, is one of her most recognizable public art pieces.',

url: 'https://i.imgur.com/Mx7dA2Y.jpg',

alt: 'A bronze statue of two crossed hands delicately holding a human brain in their fingertips.'

}, {

name: 'Floralis Genérica',

artist: 'Eduardo Catalano',

description: 'This enormous (75 ft. or 23m) silver flower is located in Buenos Aires. It is designed to move, closing its petals in the evening or when strong winds blow and opening them in the morning.',

url: 'https://i.imgur.com/ZF6s192m.jpg',

alt: 'A gigantic metallic flower sculpture with reflective mirror-like petals and strong stamens.'

}, {

name: 'Eternal Presence',

artist: 'John Woodrow Wilson',

description: 'Wilson was known for his preoccupation with equality, social justice, as well as the essential and spiritual qualities of humankind. This massive (7ft. or 2,13m) bronze represents what he described as "a symbolic Black presence infused with a sense of universal humanity."',

url: 'https://i.imgur.com/aTtVpES.jpg',

alt: 'The sculpture depicting a human head seems ever-present and solemn. It radiates calm and serenity.'

}, {

name: 'Moai',

artist: 'Unknown Artist',

description: 'Located on the Easter Island, there are 1,000 moai, or extant monumental statues, created by the early Rapa Nui people, which some believe represented deified ancestors.',

url: 'https://i.imgur.com/RCwLEoQm.jpg',

alt: 'Three monumental stone busts with the heads that are disproportionately large with somber faces.'

}, {

name: 'Blue Nana',

artist: 'Niki de Saint Phalle',

description: 'The Nanas are triumphant creatures, symbols of femininity and maternity. Initially, Saint Phalle used fabric and found objects for the Nanas, and later on introduced polyester to achieve a more vibrant effect.',

url: 'https://i.imgur.com/Sd1AgUOm.jpg',

alt: 'A large mosaic sculpture of a whimsical dancing female figure in a colorful costume emanating joy.'

}, {

name: 'Ultimate Form',

artist: 'Barbara Hepworth',

description: 'This abstract bronze sculpture is a part of The Family of Man series located at Yorkshire Sculpture Park. Hepworth chose not to create literal representations of the world but developed abstract forms inspired by people and landscapes.',

url: 'https://i.imgur.com/2heNQDcm.jpg',

alt: 'A tall sculpture made of three elements stacked on each other reminding of a human figure.'

}, {

name: 'Cavaliere',

artist: 'Lamidi Olonade Fakeye',

description: "Descended from four generations of woodcarvers, Fakeye's work blended traditional and contemporary Yoruba themes.",

url: 'https://i.imgur.com/wIdGuZwm.png',

alt: 'An intricate wood sculpture of a warrior with a focused face on a horse adorned with patterns.'

}, {

name: 'Big Bellies',

artist: 'Alina Szapocznikow',

description: "Szapocznikow is known for her sculptures of the fragmented body as a metaphor for the fragility and impermanence of youth and beauty. This sculpture depicts two very realistic large bellies stacked on top of each other, each around five feet (1,5m) tall.",

url: 'https://i.imgur.com/AlHTAdDm.jpg',

alt: 'The sculpture reminds a cascade of folds, quite different from bellies in classical sculptures.'

}, {

name: 'Terracotta Army',

artist: 'Unknown Artist',

description: 'The Terracotta Army is a collection of terracotta sculptures depicting the armies of Qin Shi Huang, the first Emperor of China. The army consisted of more than 8,000 soldiers, 130 chariots with 520 horses, and 150 cavalry horses.',

url: 'https://i.imgur.com/HMFmH6m.jpg',

alt: '12 terracotta sculptures of solemn warriors, each with a unique facial expression and armor.'

}, {

name: 'Lunar Landscape',

artist: 'Louise Nevelson',

description: 'Nevelson was known for scavenging objects from New York City debris, which she would later assemble into monumental constructions. In this one, she used disparate parts like a bedpost, juggling pin, and seat fragment, nailing and gluing them into boxes that reflect the influence of Cubism’s geometric abstraction of space and form.',

url: 'https://i.imgur.com/rN7hY6om.jpg',

alt: 'A black matte sculpture where the individual elements are initially indistinguishable.'

}, {

name: 'Aureole',

artist: 'Ranjani Shettar',

description: 'Shettar merges the traditional and the modern, the natural and the industrial. Her art focuses on the relationship between man and nature. Her work was described as compelling both abstractly and figuratively, gravity defying, and a "fine synthesis of unlikely materials."',

url: 'https://i.imgur.com/okTpbHhm.jpg',

alt: 'A pale wire-like sculpture mounted on concrete wall and descending on the floor. It appears light.'

}, {

name: 'Hippos',

artist: 'Taipei Zoo',

description: 'The Taipei Zoo commissioned a Hippo Square featuring submerged hippos at play.',

url: 'https://i.imgur.com/6o5Vuyu.jpg',

alt: 'A group of bronze hippo sculptures emerging from the sett sidewalk as if they were swimming.'

}];

// Make UI match the initial state.

updateDOM();

```

```html public/index.html

<button id="nextButton">

Next

</button>

<h3 id="header"></h3>

<button id="moreButton"></button>

<p id="description"></p>

<img id="image">

<style>

* { box-sizing: border-box; }

body { font-family: sans-serif; margin: 20px; padding: 0; }

button { display: block; margin-bottom: 10px; }

</style>

```

```css

button { display: block; margin-bottom: 10px; }

```
```

</Sandpack>

You don't have to understand it to use React, but you might find this a helpful mental model.

</DeepDive>

State is isolated and private {/*state-is-isolated-and-private*/}

State is local to a component instance on the screen. In other words, **if you render the same component twice, each copy will have completely isolated state!** Changing one of them will not affect the other.

In this example, the `Gallery` component from earlier is rendered twice with no changes to its logic. Try clicking the buttons inside each of the galleries. Notice that their state is independent:

<Sandpack>

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

export default function Page() {
return (
<div className="Page">
<Gallery />
<Gallery />
</div>
);
}
```

```js Gallery.js
import { useState } from 'react';
import { sculptureList } from './data.js';

export default function Gallery() {
const [index, setIndex] = useState(0);
const [showMore, setShowMore] = useState(false);

function handleNextClick() {
setIndex(index + 1);
}

function handleMoreClick() {
setShowMore(!showMore);
```

```
}

let sculpture = sculptureList[index];
return (
<section>
<button onClick={handleNextClick}>
Next
</button>
<h2>
<i>{sculpture.name} </i>
by {sculpture.artist}
</h2>
<h3>
({index + 1} of {sculptureList.length})
</h3>
<button onClick={handleMoreClick}>
{showMore ? 'Hide' : 'Show'} details
</button>
{showMore && <p>{sculpture.description}</p>}
<img
src={sculpture.url}
alt={sculpture.alt}
/>
</section>
);
}
```

```js data.js
export const sculptureList = [{
name: 'Homenaje a la Neurocirugía',
artist: 'Marta Colvin Andrade',
description: 'Although Colvin is predominantly known for abstract themes that allude to pre-Hispanic symbols, this gigantic sculpture, an homage to neurosurgery, is one of her most recognizable public art pieces.',
url: 'https://i.imgur.com/Mx7dA2Y.jpg',
alt: 'A bronze statue of two crossed hands delicately holding a human brain in their fingertips.'
```

}, {

name: 'Floralis Genérica',

artist: 'Eduardo Catalano',

description: 'This enormous (75 ft. or 23m) silver flower is located in Buenos Aires. It is designed to move, closing its petals in the evening or when strong winds blow and opening them in the morning.',

url: 'https://i.imgur.com/ZF6s192m.jpg',

alt: 'A gigantic metallic flower sculpture with reflective mirror-like petals and strong stamens.'

}, {

name: 'Eternal Presence',

artist: 'John Woodrow Wilson',

description: 'Wilson was known for his preoccupation with equality, social justice, as well as the essential and spiritual qualities of humankind. This massive (7ft. or 2,13m) bronze represents what he described as "a symbolic Black presence infused with a sense of universal humanity."',

url: 'https://i.imgur.com/aTtVpES.jpg',

alt: 'The sculpture depicting a human head seems ever-present and solemn. It radiates calm and serenity.'

}, {

name: 'Moai',

artist: 'Unknown Artist',

description: 'Located on the Easter Island, there are 1,000 moai, or extant monumental statues, created by the early Rapa Nui people, which some believe represented deified ancestors.',

url: 'https://i.imgur.com/RCwLEoQm.jpg',

alt: 'Three monumental stone busts with the heads that are disproportionately large with somber faces.'

}, {

name: 'Blue Nana',

artist: 'Niki de Saint Phalle',

description: 'The Nanas are triumphant creatures, symbols of femininity and maternity. Initially, Saint Phalle used fabric and found objects for the Nanas, and later on introduced polyester to achieve a more vibrant effect.',

url: 'https://i.imgur.com/Sd1AgUOm.jpg',

alt: 'A large mosaic sculpture of a whimsical dancing female figure in a colorful costume emanating joy.'

}, {

name: 'Ultimate Form',

artist: 'Barbara Hepworth',

description: 'This abstract bronze sculpture is a part of The Family of Man series located at Yorkshire Sculpture Park. Hepworth chose not to create literal representations of the world but developed abstract forms inspired by people and landscapes.',

url: 'https://i.imgur.com/2heNQDcm.jpg',

alt: 'A tall sculpture made of three elements stacked on each other reminding of a human figure.'

}, {

name: 'Cavaliere',

artist: 'Lamidi Olonade Fakeye',

description: "Descended from four generations of woodcarvers, Fakeye's work blended traditional and contemporary Yoruba themes.",

url: 'https://i.imgur.com/wIdGuZwm.png',

alt: 'An intricate wood sculpture of a warrior with a focused face on a horse adorned with patterns.'

}, {

name: 'Big Bellies',

artist: 'Alina Szapocznikow',

description: "Szapocznikow is known for her sculptures of the fragmented body as a metaphor for the fragility and impermanence of youth and beauty. This sculpture depicts two very realistic large bellies stacked on top of each other, each around five feet (1,5m) tall.",

url: 'https://i.imgur.com/AlHTAdDm.jpg',

alt: 'The sculpture reminds a cascade of folds, quite different from bellies in classical sculptures.'

}, {

name: 'Terracotta Army',

artist: 'Unknown Artist',

description: 'The Terracotta Army is a collection of terracotta sculptures depicting the armies of Qin Shi Huang, the first Emperor of China. The army consisted of more than 8,000 soldiers, 130 chariots with 520 horses, and 150 cavalry horses.',

url: 'https://i.imgur.com/HMFmH6m.jpg',

alt: '12 terracotta sculptures of solemn warriors, each with a unique facial expression and armor.'

}, {

name: 'Lunar Landscape',

artist: 'Louise Nevelson',

description: 'Nevelson was known for scavenging objects from New York City debris, which she would later assemble into monumental constructions. In this one, she used disparate parts like a bedpost, juggling pin, and seat fragment, nailing and gluing them into boxes that reflect the influence of Cubism’s geometric abstraction of space and form.',

url: 'https://i.imgur.com/rN7hY6om.jpg',

alt: 'A black matte sculpture where the individual elements are initially indistinguishable.'

}, {

name: 'Aureole',

artist: 'Ranjani Shettar',

description: 'Shettar merges the traditional and the modern, the natural and the industrial. Her art focuses on the relationship between man and nature. Her work was described as compelling both abstractly and figuratively, gravity defying, and a "fine synthesis of unlikely materials."',

```
url: 'https://i.imgur.com/okTpbHhm.jpg',
alt: 'A pale wire-like sculpture mounted on concrete wall and descending on the floor. It appears light.'
}, {
name: 'Hippos',
artist: 'Taipei Zoo',
description: 'The Taipei Zoo commissioned a Hippo Square featuring submerged hippos at play.',
url: 'https://i.imgur.com/6o5Vuyu.jpg',
alt: 'A group of bronze hippo sculptures emerging from the sett sidewalk as if they were swimming.'
}];
```

```css
button { display: block; margin-bottom: 10px; }
.Page > * {
float: left;
width: 50%;
padding: 10px;
}
h2 { margin-top: 10px; margin-bottom: 0; }
h3 {
margin-top: 5px;
font-weight: normal;
font-size: 100%;
}
img { width: 120px; height: 120px; }
button {
display: block;
margin-top: 10px;
margin-bottom: 10px;
}
```

</Sandpack>

This is what makes state different from regular variables that you might declare at the top of your module. State is not tied to a particular function call or a place in the code, but it's "local" to the specific place on the screen. You rendered two `<Gallery />` components, so their state is stored separately.

Also notice how the `Page` component doesn't "know" anything about the `Gallery` state or even whether it has any. Unlike props, **state is fully private to the component declaring it.** The parent component can't change it. This lets you add state to any component or remove it without impacting the rest of the components.

What if you wanted both galleries to keep their states in sync? The right way to do it in React is to *remove* state from child components and add it to their closest shared parent. The next few pages will focus on organizing state of a single component, but we will return to this topic in [Sharing State Between Components.](/learn/sharing-state-between-components)

<Recap>

* Use a state variable when a component needs to "remember" some information between renders.

* State variables are declared by calling the `useState` Hook.

* Hooks are special functions that start with `use`. They let you "hook into" React features like state.

* Hooks might remind you of imports: they need to be called unconditionally. Calling Hooks, including `useState`, is only valid at the top level of a component or another Hook.

* The `useState` Hook returns a pair of values: the current state and the function to update it.

* You can have more than one state variable. Internally, React matches them up by their order.

* State is private to the component. If you render it in two places, each copy gets its own state.

</Recap>

<Challenges>

Complete the gallery {/*complete-the-gallery*/}

When you press "Next" on the last sculpture, the code crashes. Fix the logic to prevent the crash. You may do this by adding extra logic to event handler or by disabling the button when the action is not possible.

After fixing the crash, add a "Previous" button that shows the previous sculpture. It shouldn't crash on the first sculpture.

<Sandpack>

```js
import { useState } from 'react';
import { sculptureList } from './data.js';

export default function Gallery() {
const [index, setIndex] = useState(0);
const [showMore, setShowMore] = useState(false);

function handleNextClick() {
setIndex(index + 1);
```

```
}

function handleMoreClick() {
setShowMore(!showMore);
}

let sculpture = sculptureList[index];
return (
<>
<button onClick={handleNextClick}>
Next
</button>
<h2>
<i>{sculpture.name} </i>
by {sculpture.artist}
</h2>
<h3>
({index + 1} of {sculptureList.length})
</h3>
<button onClick={handleMoreClick}>
{showMore ? 'Hide' : 'Show'} details
</button>
{showMore && <p>{sculpture.description}</p>}
<img
src={sculpture.url}
alt={sculpture.alt}
/>
</>
);
}
```

```js data.js
export const sculptureList = [{
name: 'Homenaje a la Neurocirugía',
artist: 'Marta Colvin Andrade',
description: 'Although Colvin is predominantly known for abstract themes that allude to pre-Hispanic symbols, this gigantic sculpture, an homage to neurosurgery, is one of her most recognizable public art
```

pieces.',

url: 'https://i.imgur.com/Mx7dA2Y.jpg',

alt: 'A bronze statue of two crossed hands delicately holding a human brain in their fingertips.'

}, {

name: 'Floralis Genérica',

artist: 'Eduardo Catalano',

description: 'This enormous (75 ft. or 23m) silver flower is located in Buenos Aires. It is designed to move, closing its petals in the evening or when strong winds blow and opening them in the morning.',

url: 'https://i.imgur.com/ZF6s192m.jpg',

alt: 'A gigantic metallic flower sculpture with reflective mirror-like petals and strong stamens.'

}, {

name: 'Eternal Presence',

artist: 'John Woodrow Wilson',

description: 'Wilson was known for his preoccupation with equality, social justice, as well as the essential and spiritual qualities of humankind. This massive (7ft. or 2,13m) bronze represents what he described as "a symbolic Black presence infused with a sense of universal humanity."',

url: 'https://i.imgur.com/aTtVpES.jpg',

alt: 'The sculpture depicting a human head seems ever-present and solemn. It radiates calm and serenity.'

}, {

name: 'Moai',

artist: 'Unknown Artist',

description: 'Located on the Easter Island, there are 1,000 moai, or extant monumental statues, created by the early Rapa Nui people, which some believe represented deified ancestors.',

url: 'https://i.imgur.com/RCwLEoQm.jpg',

alt: 'Three monumental stone busts with the heads that are disproportionately large with somber faces.'

}, {

name: 'Blue Nana',

artist: 'Niki de Saint Phalle',

description: 'The Nanas are triumphant creatures, symbols of femininity and maternity. Initially, Saint Phalle used fabric and found objects for the Nanas, and later on introduced polyester to achieve a more vibrant effect.',

url: 'https://i.imgur.com/Sd1AgUOm.jpg',

alt: 'A large mosaic sculpture of a whimsical dancing female figure in a colorful costume emanating joy.'

}, {

name: 'Ultimate Form',

artist: 'Barbara Hepworth',

description: 'This abstract bronze sculpture is a part of The Family of Man series located at Yorkshire Sculpture Park. Hepworth chose not to create literal representations of the world but developed abstract forms inspired by people and landscapes.',

url: 'https://i.imgur.com/2heNQDcm.jpg',

alt: 'A tall sculpture made of three elements stacked on each other reminding of a human figure.'

}, {

name: 'Cavaliere',

artist: 'Lamidi Olonade Fakeye',

description: "Descended from four generations of woodcarvers, Fakeye's work blended traditional and contemporary Yoruba themes.",

url: 'https://i.imgur.com/wIdGuZwm.png',

alt: 'An intricate wood sculpture of a warrior with a focused face on a horse adorned with patterns.'

}, {

name: 'Big Bellies',

artist: 'Alina Szapocznikow',

description: "Szapocznikow is known for her sculptures of the fragmented body as a metaphor for the fragility and impermanence of youth and beauty. This sculpture depicts two very realistic large bellies stacked on top of each other, each around five feet (1,5m) tall.",

url: 'https://i.imgur.com/AlHTAdDm.jpg',

alt: 'The sculpture reminds a cascade of folds, quite different from bellies in classical sculptures.'

}, {

name: 'Terracotta Army',

artist: 'Unknown Artist',

description: 'The Terracotta Army is a collection of terracotta sculptures depicting the armies of Qin Shi Huang, the first Emperor of China. The army consisted of more than 8,000 soldiers, 130 chariots with 520 horses, and 150 cavalry horses.',

url: 'https://i.imgur.com/HMFmH6m.jpg',

alt: '12 terracotta sculptures of solemn warriors, each with a unique facial expression and armor.'

}, {

name: 'Lunar Landscape',

artist: 'Louise Nevelson',

description: 'Nevelson was known for scavenging objects from New York City debris, which she would later assemble into monumental constructions. In this one, she used disparate parts like a bedpost, juggling pin, and seat fragment, nailing and gluing them into boxes that reflect the influence of Cubism’s geometric abstraction of space and form.',

url: 'https://i.imgur.com/rN7hY6om.jpg',

alt: 'A black matte sculpture where the individual elements are initially indistinguishable.'

}, {

name: 'Aureole',

artist: 'Ranjani Shettar',

description: 'Shettar merges the traditional and the modern, the natural and the industrial. Her art focuses on the relationship between man and nature. Her work was described as compelling both abstractly and figuratively, gravity defying, and a "fine synthesis of unlikely materials."',

url: 'https://i.imgur.com/okTpbHhm.jpg',

alt: 'A pale wire-like sculpture mounted on concrete wall and descending on the floor. It appears light.'

}, {

name: 'Hippos',

artist: 'Taipei Zoo',

description: 'The Taipei Zoo commissioned a Hippo Square featuring submerged hippos at play.',

url: 'https://i.imgur.com/6o5Vuyu.jpg',

alt: 'A group of bronze hippo sculptures emerging from the sett sidewalk as if they were swimming.'

}];

```

```css

button { display: block; margin-bottom: 10px; }

.Page > * {

float: left;

width: 50%;

padding: 10px;

}

h2 { margin-top: 10px; margin-bottom: 0; }

h3 {

margin-top: 5px;

font-weight: normal;

font-size: 100%;

}

img { width: 120px; height: 120px; }

```

</Sandpack>

<Solution>

This adds a guarding condition inside both event handlers and disables the buttons when needed:

<Sandpack>

```js

```
import { useState } from 'react';
import { sculptureList } from './data.js';

export default function Gallery() {
const [index, setIndex] = useState(0);
const [showMore, setShowMore] = useState(false);

let hasPrev = index > 0;
let hasNext = index < sculptureList.length - 1;

function handlePrevClick() {
if (hasPrev) {
setIndex(index - 1);
}
}

function handleNextClick() {
if (hasNext) {
setIndex(index + 1);
}
}

function handleMoreClick() {
setShowMore(!showMore);
}

let sculpture = sculptureList[index];
return (
<>
<button
onClick={handlePrevClick}
disabled={!hasPrev}
>
Previous
</button>
<button
onClick={handleNextClick}
disabled={!hasNext}
>
```

Next

</button>

<h2>

{sculpture.name} 

by {sculpture.artist}

</h2>

<h3>

({index + 1} of {sculptureList.length})

</h3>

<button onClick={handleMoreClick}>

{showMore ? 'Hide' : 'Show'} details

</button>

{showMore && {sculpture.description} }

<img

src={sculpture.url}

alt={sculpture.alt}

/>

</>

);

}

```

```js data.js hidden

export const sculptureList = [{

name: 'Homenaje a la Neurocirugía',

artist: 'Marta Colvin Andrade',

description: 'Although Colvin is predominantly known for abstract themes that allude to pre-Hispanic symbols, this gigantic sculpture, an homage to neurosurgery, is one of her most recognizable public art pieces.',

url: 'https://i.imgur.com/Mx7dA2Y.jpg',

alt: 'A bronze statue of two crossed hands delicately holding a human brain in their fingertips.'

}, {

name: 'Floralis Genérica',

artist: 'Eduardo Catalano',

description: 'This enormous (75 ft. or 23m) silver flower is located in Buenos Aires. It is designed to move, closing its petals in the evening or when strong winds blow and opening them in the morning.',

url: 'https://i.imgur.com/ZF6s192m.jpg',

alt: 'A gigantic metallic flower sculpture with reflective mirror-like petals and strong stamens.'

}, {

name: 'Eternal Presence',

artist: 'John Woodrow Wilson',

description: 'Wilson was known for his preoccupation with equality, social justice, as well as the essential and spiritual qualities of humankind. This massive (7ft. or 2,13m) bronze represents what he described as "a symbolic Black presence infused with a sense of universal humanity."',

url: 'https://i.imgur.com/aTtVpES.jpg',

alt: 'The sculpture depicting a human head seems ever-present and solemn. It radiates calm and serenity.'

}, {

name: 'Moai',

artist: 'Unknown Artist',

description: 'Located on the Easter Island, there are 1,000 moai, or extant monumental statues, created by the early Rapa Nui people, which some believe represented deified ancestors.',

url: 'https://i.imgur.com/RCwLEoQm.jpg',

alt: 'Three monumental stone busts with the heads that are disproportionately large with somber faces.'

}, {

name: 'Blue Nana',

artist: 'Niki de Saint Phalle',

description: 'The Nanas are triumphant creatures, symbols of femininity and maternity. Initially, Saint Phalle used fabric and found objects for the Nanas, and later on introduced polyester to achieve a more vibrant effect.',

url: 'https://i.imgur.com/Sd1AgUOm.jpg',

alt: 'A large mosaic sculpture of a whimsical dancing female figure in a colorful costume emanating joy.'

}, {

name: 'Ultimate Form',

artist: 'Barbara Hepworth',

description: 'This abstract bronze sculpture is a part of The Family of Man series located at Yorkshire Sculpture Park. Hepworth chose not to create literal representations of the world but developed abstract forms inspired by people and landscapes.',

url: 'https://i.imgur.com/2heNQDcm.jpg',

alt: 'A tall sculpture made of three elements stacked on each other reminding of a human figure.'

}, {

name: 'Cavaliere',

artist: 'Lamidi Olonade Fakeye',

description: "Descended from four generations of woodcarvers, Fakeye's work blended traditional and contemporary Yoruba themes.",

url: 'https://i.imgur.com/wIdGuZwm.png',

alt: 'An intricate wood sculpture of a warrior with a focused face on a horse adorned with patterns.'

}, {

name: 'Big Bellies',

artist: 'Alina Szapocznikow',

description: "Szapocznikow is known for her sculptures of the fragmented body as a metaphor for the fragility and impermanence of youth and beauty. This sculpture depicts two very realistic large bellies stacked on top of each other, each around five feet (1,5m) tall.",

url: 'https://i.imgur.com/AlHTAdDm.jpg',

alt: 'The sculpture reminds a cascade of folds, quite different from bellies in classical sculptures.'

}, {

name: 'Terracotta Army',

artist: 'Unknown Artist',

description: 'The Terracotta Army is a collection of terracotta sculptures depicting the armies of Qin Shi Huang, the first Emperor of China. The army consisted of more than 8,000 soldiers, 130 chariots with 520 horses, and 150 cavalry horses.',

url: 'https://i.imgur.com/HMFmH6m.jpg',

alt: '12 terracotta sculptures of solemn warriors, each with a unique facial expression and armor.'

}, {

name: 'Lunar Landscape',

artist: 'Louise Nevelson',

description: 'Nevelson was known for scavenging objects from New York City debris, which she would later assemble into monumental constructions. In this one, she used disparate parts like a bedpost, juggling pin, and seat fragment, nailing and gluing them into boxes that reflect the influence of Cubism’s geometric abstraction of space and form.',

url: 'https://i.imgur.com/rN7hY6om.jpg',

alt: 'A black matte sculpture where the individual elements are initially indistinguishable.'

}, {

name: 'Aureole',

artist: 'Ranjani Shettar',

description: 'Shettar merges the traditional and the modern, the natural and the industrial. Her art focuses on the relationship between man and nature. Her work was described as compelling both abstractly and figuratively, gravity defying, and a "fine synthesis of unlikely materials."',

url: 'https://i.imgur.com/okTpbHhm.jpg',

alt: 'A pale wire-like sculpture mounted on concrete wall and descending on the floor. It appears light.'

}, {

name: 'Hippos',

artist: 'Taipei Zoo',

description: 'The Taipei Zoo commissioned a Hippo Square featuring submerged hippos at play.',

url: 'https://i.imgur.com/6o5Vuyu.jpg',

alt: 'A group of bronze hippo sculptures emerging from the sett sidewalk as if they were swimming.'

}];

```

```css

button { display: block; margin-bottom: 10px; }

.Page > * {

float: left;

width: 50%;

padding: 10px;

}

h2 { margin-top: 10px; margin-bottom: 0; }

h3 {

margin-top: 5px;

font-weight: normal;

font-size: 100%;

}

img { width: 120px; height: 120px; }

```

</Sandpack>

Notice how `hasPrev` and `hasNext` are used *both* for the returned JSX and inside the event handlers! This handy pattern works because event handler functions ["close over"](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures) any variables declared while rendering.

</Solution>

Fix stuck form inputs {/*fix-stuck-form-inputs*/}

When you type into the input fields, nothing appears. It's like the input values are "stuck" with empty strings. The `value` of the first `<input>` is set to always match the `firstName` variable, and the `value` for the second `<input>` is set to always match the `lastName` variable. This is correct. Both inputs have `onChange` event handlers, which try to update the variables based on the latest user input (`e.target.value`). However, the variables don't seem to "remember" their values between re-renders. Fix this by using state variables instead.

<Sandpack>

```js

```
export default function Form() {
let firstName = '';
let lastName = '';

function handleFirstNameChange(e) {
firstName = e.target.value;
}

function handleLastNameChange(e) {
lastName = e.target.value;
}

function handleReset() {
firstName = '';
lastName = '';
}

return (
<form onSubmit={e => e.preventDefault()}>
<input
placeholder="First name"
value={firstName}
onChange={handleFirstNameChange}
/>
<input
placeholder="Last name"
value={lastName}
onChange={handleLastNameChange}
/>
<h1>Hi, {firstName} {lastName}</h1>
<button onClick={handleReset}>Reset</button>
</form>
);
}
```

```css
h1 { margin-top: 10px; }
```

</Sandpack>

<Solution>

First, import `useState` from React. Then replace `firstName` and `lastName` with state variables declared by calling `useState`. Finally, replace every `firstName = ...` assignment with `setFirstName(...)`, and do the same for `lastName`. Don't forget to update `handleReset` too so that the reset button works.

<Sandpack>

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

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

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

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

function handleReset() {
setFirstName('');
setLastName('');
}

return (
<form onSubmit={e => e.preventDefault()}>
<input
placeholder="First name"
value={firstName}
onChange={handleFirstNameChange}
/>
<input
placeholder="Last name"
value={lastName}
onChange={handleLastNameChange}
```

/>

<h1>Hi, {firstName} {lastName}</h1>

<button onClick={handleReset}>Reset</button>

</form>

);

}

```

```css

h1 { margin-top: 10px; }

```

</Sandpack>

</Solution>

Fix a crash {/*fix-a-crash*/}

Here is a small form that is supposed to let the user leave some feedback. When the feedback is submitted, it's supposed to display a thank-you message. However, it crashes with an error message saying "Rendered fewer hooks than expected". Can you spot the mistake and fix it?

<Hint>

Are there any limitations on _where_ Hooks may be called? Does this component break any rules? Check if there are any comments disabling the linter checks--this is where the bugs often hide!

</Hint>

<Sandpack>

```js

import { useState } from 'react';

export default function FeedbackForm() {

const [isSent, setIsSent] = useState(false);

if (isSent) {

return <h1>Thank you!</h1>;

} else {

// eslint-disable-next-line

const [message, setMessage] = useState('');

return (

<form onSubmit={e => {

e.preventDefault();

alert(`Sending: "${message}"`);
setIsSent(true);
}}>
<textarea
placeholder="Message"
value={message}
onChange={e => setMessage(e.target.value)}
/>
<br />
<button type="submit">Send</button>
</form>
);
}
}
```

</Sandpack>

<Solution>

Hooks can only be called at the top level of the component function. Here, the first `isSent` definition follows this rule, but the `message` definition is nested in a condition.

Move it out of the condition to fix the issue:

<Sandpack>

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

export default function FeedbackForm() {
const [isSent, setIsSent] = useState(false);
const [message, setMessage] = useState('');

if (isSent) {
return <h1>Thank you!</h1>;
} else {
return (
<form onSubmit={e => {
e.preventDefault();
alert(`Sending: "${message}"`);

setIsSent(true);

}}>

<textarea

placeholder="Message"

value={message}

onChange={e => setMessage(e.target.value)}

/>

<br />

<button type="submit">Send</button>

</form>

);

}

}

```

</Sandpack>

Remember, Hooks must be called unconditionally and always in the same order!

You could also remove the unnecessary `else` branch to reduce the nesting. However, it's still important that all calls to Hooks happen *before* the first `return`.

<Sandpack>

```js

import { useState } from 'react';

export default function FeedbackForm() {

const [isSent, setIsSent] = useState(false);

const [message, setMessage] = useState('');

if (isSent) {

return <h1>Thank you!</h1>;

}

return (

<form onSubmit={e => {

e.preventDefault();

alert(`Sending: "${message}"`);

setIsSent(true);

}}>

<textarea

placeholder="Message"

value={message}

onChange={e => setMessage(e.target.value)}

/>

<br />

<button type="submit">Send</button>

</form>

);

}

```

</Sandpack>

Try moving the second `useState` call after the `if` condition and notice how this breaks it again.

If your linter is [configured for React](/learn/editor-setup#linting), you should see a lint error when you make a mistake like this. If you don't see an error when you try the faulty code locally, you need to set up linting for your project.

</Solution>

Remove unnecessary state {/*remove-unnecessary-state*/}

When the button is clicked, this example should ask for the user's name and then display an alert greeting them. You tried to use state to keep the name, but for some reason it always shows "Hello, !".

To fix this code, remove the unnecessary state variable. (We will discuss about [why this didn't work](/learn/state-as-a-snapshot) later.)

Can you explain why this state variable was unnecessary?

<Sandpack>

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

export default function FeedbackForm() {

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

function handleClick() {

setName(prompt('What is your name?'));

alert(`Hello, ${name}!`);

}

return (
<button onClick={handleClick}>
Greet
</button>
);
}
```

</Sandpack>

<Solution>

Here is a fixed version that uses a regular `name` variable declared in the function that needs it:

<Sandpack>

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

export default function FeedbackForm() {
function handleClick() {
const name = prompt('What is your name?');
alert(`Hello, ${name}!`);
}

return (
<button onClick={handleClick}>
Greet
</button>
);
}
```

</Sandpack>

A state variable is only necessary to keep information between re-renders of a component. Within a single event handler, a regular variable will do fine. Don't introduce state variables when a regular variable works well.

</Solution>

</Challenges>

```

title: 'Referencing Values with Refs'

<Intro>

When you want a component to "remember" some information, but you don't want that information to [trigger new renders](/learn/render-and-commit), you can use a *ref*.

</Intro>

<YouWillLearn>

- How to add a ref to your component
- How to update a ref's value
- How refs are different from state
- How to use refs safely

</YouWillLearn>

Adding a ref to your component {/*adding-a-ref-to-your-component*/}

You can add a ref to your component by importing the `useRef` Hook from React:

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

Inside your component, call the `useRef` Hook and pass the initial value that you want to reference as the only argument. For example, here is a ref to the value `0`:

```js
const ref = useRef(0);
```

`useRef` returns an object like this:

```js
{
current: 0 // The value you passed to useRef
}
```

<Illustration src="/images/docs/illustrations/i_ref.png" alt="An arrow with 'current' written on it stuffed into a pocket with 'ref' written on it." />

You can access the current value of that ref through the `ref.current` property. This value is intentionally mutable, meaning you can both read and write to it. It's like a secret pocket of your component that

React doesn't track. (This is what makes it an "escape hatch" from React's one-way data flow--more on that below!)

Here, a button will increment `ref.current` on every click:

<Sandpack>

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

export default function Counter() {
let ref = useRef(0);

function handleClick() {
ref.current = ref.current + 1;
alert('You clicked ' + ref.current + ' times!');
}

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

</Sandpack>

The ref points to a number, but, like [state](/learn/state-a-components-memory), you could point to anything: a string, an object, or even a function. Unlike state, ref is a plain JavaScript object with the `current` property that you can read and modify.

Note that **the component doesn't re-render with every increment.** Like state, refs are retained by React between re-renders. However, setting state re-renders a component. Changing a ref does not!

Example: building a stopwatch {/*example-building-a-stopwatch*/}

You can combine refs and state in a single component. For example, let's make a stopwatch that the user can start or stop by pressing a button. In order to display how much time has passed since the user pressed "Start", you will need to keep track of when the Start button was pressed and what the current time is. **This information is used for rendering, so you'll keep it in state:**

```js
const [startTime, setStartTime] = useState(null);
const [now, setNow] = useState(null);
```

When the user presses "Start", you'll use [`setInterval`](https://developer.mozilla.org/docs/Web/API/setInterval) in order to update the time every 10 milliseconds:

<Sandpack>

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

export default function Stopwatch() {
const [startTime, setStartTime] = useState(null);
const [now, setNow] = useState(null);

function handleStart() {
// Start counting.
setStartTime(Date.now());
setNow(Date.now());

setInterval(() => {
// Update the current time every 10ms.
setNow(Date.now());
}, 10);
}

let secondsPassed = 0;
if (startTime != null && now != null) {
secondsPassed = (now - startTime) / 1000;
}

return (
<>
<h1>Time passed: {secondsPassed.toFixed(3)}</h1>
<button onClick={handleStart}>
Start
</button>
</>
);
}
```

</Sandpack>

When the "Stop" button is pressed, you need to cancel the existing interval so that it stops updating the `now` state variable. You can do this by calling [`clearInterval`](https://developer.mozilla.org/en-US/docs/Web/API/clearInterval), but you need to give it the interval ID that was previously returned by the `setInterval` call when the user pressed Start. You need to keep the interval ID somewhere. **Since the interval ID is not used for rendering, you can keep it in a ref:**

<Sandpack>

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

export default function Stopwatch() {
const [startTime, setStartTime] = useState(null);
const [now, setNow] = useState(null);
const intervalRef = useRef(null);

function handleStart() {
setStartTime(Date.now());
setNow(Date.now());

clearInterval(intervalRef.current);
intervalRef.current = setInterval(() => {
setNow(Date.now());
}, 10);
}

function handleStop() {
clearInterval(intervalRef.current);
}

let secondsPassed = 0;
if (startTime != null && now != null) {
secondsPassed = (now - startTime) / 1000;
}

return (
<>
<h1>Time passed: {secondsPassed.toFixed(3)}</h1>
<button onClick={handleStart}>
Start
</button>
```

<button onClick={handleStop}>

Stop

</button>

</>

);

}

```

</Sandpack>

When a piece of information is used for rendering, keep it in state. When a piece of information is only needed by event handlers and changing it doesn't require a re-render, using a ref may be more efficient.

## Differences between refs and state {/*differences-between-refs-and-state*/}

Perhaps you're thinking refs seem less "strict" than state—you can mutate them instead of always having to use a state setting function, for instance. But in most cases, you'll want to use state. Refs are an "escape hatch" you won't need often. Here's how state and refs compare:

| refs | state |

| ---------- |
---------- |

| `useRef(initialValue)` returns `{ current: initialValue }` | `useState(initialValue)` returns the current value of a state variable and a state setter function ( `[value, setValue]`) |

| Doesn't trigger re-render when you change it. | Triggers re-render when you change it. |

| Mutable—you can modify and update `current`'s value outside of the rendering process. | "Immutable"—you must use the state setting function to modify state variables to queue a re-render. |

| You shouldn't read (or write) the `current` value during rendering. | You can read state at any time. However, each render has its own [snapshot](/learn/state-as-a-snapshot) of state which does not change.

Here is a counter button that's implemented with state:

<Sandpack>

```js

import { useState } from 'react';

export default function Counter() {

const [count, setCount] = useState(0);

function handleClick() {

setCount(count + 1);

}

return (

<button onClick={handleClick}>

You clicked {count} times

</button>

);

}

```

</Sandpack>

Because the `count` value is displayed, it makes sense to use a state value for it. When the counter's value is set with `setCount()`, React re-renders the component and the screen updates to reflect the new count.

If you tried to implement this with a ref, React would never re-render the component, so you'd never see the count change! See how clicking this button **does not update its text**:

<Sandpack>

```js

import { useRef } from 'react';

export default function Counter() {

let countRef = useRef(0);

function handleClick() {

// This doesn't re-render the component!

countRef.current = countRef.current + 1;

}

return (

<button onClick={handleClick}>

You clicked {countRef.current} times

</button>

);

}

```

</Sandpack>

This is why reading `ref.current` during render leads to unreliable code. If you need that, use state instead.

<DeepDive>
```

How does useRef work inside? {/*how-does-use-ref-work-inside*/}

Although both `useState` and `useRef` are provided by React, in principle `useRef` could be implemented _on top of_ `useState`. You can imagine that inside of React, `useRef` is implemented like this:

```js
// Inside of React
function useRef(initialValue) {
const [ref, unused] = useState({ current: initialValue });
return ref;
}
```

During the first render, `useRef` returns `{ current: initialValue }`. This object is stored by React, so during the next render the same object will be returned. Note how the state setter is unused in this example. It is unnecessary because `useRef` always needs to return the same object!

React provides a built-in version of `useRef` because it is common enough in practice. But you can think of it as a regular state variable without a setter. If you're familiar with object-oriented programming, refs might remind you of instance fields--but instead of `this.something` you write `somethingRef.current`.

</DeepDive>

When to use refs {/*when-to-use-refs*/}

Typically, you will use a ref when your component needs to "step outside" React and communicate with external APIs—often a browser API that won't impact the appearance of the component. Here are a few of these rare situations:

- Storing [timeout IDs](https://developer.mozilla.org/docs/Web/API/setTimeout)
- Storing and manipulating [DOM elements](https://developer.mozilla.org/docs/Web/API/Element), which we cover on [the next page](/learn/manipulating-the-dom-with-refs)
- Storing other objects that aren't necessary to calculate the JSX.

If your component needs to store some value, but it doesn't impact the rendering logic, choose refs.

Best practices for refs {/*best-practices-for-refs*/}

Following these principles will make your components more predictable:

- **Treat refs as an escape hatch.** Refs are useful when you work with external systems or browser APIs. If much of your application logic and data flow relies on refs, you might want to rethink your approach.
- **Don't read or write `ref.current` during rendering.** If some information is needed during rendering, use [state](/learn/state-a-components-memory) instead. Since React doesn't know when `ref.current` changes, even reading it while rendering makes your component's behavior difficult to predict. (The

only exception to this is code like `if (!ref.current) ref.current = new Thing()` which only sets the ref once during the first render.)

Limitations of React state don't apply to refs. For example, state acts like a [snapshot for every render](/learn/state-as-a-snapshot) and [doesn't update synchronously.](/learn/queueing-a-series-of-state-updates) But when you mutate the current value of a ref, it changes immediately:

```js
ref.current = 5;
console.log(ref.current); // 5
```

This is because **the ref itself is a regular JavaScript object,** and so it behaves like one.

You also don't need to worry about [avoiding mutation](/learn/updating-objects-in-state) when you work with a ref. As long as the object you're mutating isn't used for rendering, React doesn't care what you do with the ref or its contents.

Refs and the DOM {/*refs-and-the-dom*/}

You can point a ref to any value. However, the most common use case for a ref is to access a DOM element. For example, this is handy if you want to focus an input programmatically. When you pass a ref to a `ref` attribute in JSX, like `<div ref={myRef}>`, React will put the corresponding DOM element into `myRef.current`. You can read more about this in [Manipulating the DOM with Refs.](/learn/manipulating-the-dom-with-refs)

<Recap>

- Refs are an escape hatch to hold onto values that aren't used for rendering. You won't need them often.
- A ref is a plain JavaScript object with a single property called `current`, which you can read or set.
- You can ask React to give you a ref by calling the `useRef` Hook.
- Like state, refs let you retain information between re-renders of a component.
- Unlike state, setting the ref's `current` value does not trigger a re-render.
- Don't read or write `ref.current` during rendering. This makes your component hard to predict.

</Recap>

<Challenges>

Fix a broken chat input {/*fix-a-broken-chat-input*/}

Type a message and click "Send". You will notice there is a three second delay before you see the "Sent!" alert. During this delay, you can see an "Undo" button. Click it. This "Undo" button is supposed to stop the "Sent!" message from appearing. It does this by calling [`clearTimeout`](https://developer.mozilla.org/en-US/docs/Web/API/clearTimeout) for the timeout ID saved during `handleSend`. However, even after "Undo" is clicked, the "Sent!" message still appears.

Find why it doesn't work, and fix it.

<Hint>

Regular variables like `let timeoutID` don't "survive" between re-renders because every render runs your component (and initializes its variables) from scratch. Should you keep the timeout ID somewhere else?

</Hint>

<Sandpack>

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

export default function Chat() {
const [text, setText] = useState('');
const [isSending, setIsSending] = useState(false);
let timeoutID = null;

function handleSend() {
setIsSending(true);
timeoutID = setTimeout(() => {
alert('Sent!');
setIsSending(false);
}, 3000);
}

function handleUndo() {
setIsSending(false);
clearTimeout(timeoutID);
}

return (
<>
<input
disabled={isSending}
value={text}
onChange={e => setText(e.target.value)}
/>
<button
disabled={isSending}
```

onClick={handleSend}>

{isSending ? 'Sending...' : 'Send'}

</button>

{isSending &&

<button onClick={handleUndo}>

Undo

</button>

}

</>

);

}

```

</Sandpack>

<Solution>

Whenever your component re-renders (such as when you set state), all local variables get initialized from scratch. This is why you can't save the timeout ID in a local variable like `timeoutID` and then expect another event handler to "see" it in the future. Instead, store it in a ref, which React will preserve between renders.

<Sandpack>

```js

import { useState, useRef } from 'react';

export default function Chat() {

const [text, setText] = useState('');

const [isSending, setIsSending] = useState(false);

const timeoutRef = useRef(null);

function handleSend() {

setIsSending(true);

timeoutRef.current = setTimeout(() => {

alert('Sent!');

setIsSending(false);

}, 3000);

}

function handleUndo() {

setIsSending(false);

```
clearTimeout(timeoutRef.current);
}

return (
<>
<input
disabled={isSending}
value={text}
onChange={e => setText(e.target.value)}
/>
<button
disabled={isSending}
onClick={handleSend}>
{isSending ? 'Sending...' : 'Send'}
</button>
{isSending &&
<button onClick={handleUndo}>
Undo
</button>
}
</>
);
}
```

</Sandpack>

</Solution>

Fix a component failing to re-render {/*fix-a-component-failing-to-re-render*/}

This button is supposed to toggle between showing "On" and "Off". However, it always shows "Off". What is wrong with this code? Fix it.

<Sandpack>

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

export default function Toggle() {
const isOnRef = useRef(false);
```

return (

<button onClick={() => {

isOnRef.current = !isOnRef.current;

}}>

{isOnRef.current ? 'On' : 'Off'}

</button>

);

}

```

</Sandpack>

<Solution>

In this example, the current value of a ref is used to calculate the rendering output: `{isOnRef.current ? 'On' : 'Off'}`. This is a sign that this information should not be in a ref, and should have instead been put in state. To fix it, remove the ref and use state instead:

<Sandpack>

```js

import { useState } from 'react';

export default function Toggle() {

const [isOn, setIsOn] = useState(false);

return (

<button onClick={() => {

setIsOn(!isOn);

}}>

{isOn ? 'On' : 'Off'}

</button>

);

}

```

</Sandpack>

</Solution>

Fix debouncing {/*fix-debouncing*/}

In this example, all button click handlers are ["debounced".](https://redd.one/blog/debounce-vs-throttle) To see what this means, press one of the buttons. Notice how the message appears a second later. If
```

you press the button while waiting for the message, the timer will reset. So if you keep clicking the same button fast many times, the message won't appear until a second *after* you stop clicking. Debouncing lets you delay some action until the user "stops doing things".

This example works, but not quite as intended. The buttons are not independent. To see the problem, click one of the buttons, and then immediately click another button. You'd expect that after a delay, you would see both button's messages. But only the last button's message shows up. The first button's message gets lost.

Why are the buttons interfering with each other? Find and fix the issue.

<Hint>

The last timeout ID variable is shared between all `DebouncedButton` components. This is why clicking one button resets another button's timeout. Can you store a separate timeout ID for each button?

</Hint>

<Sandpack>

```js
let timeoutID;

function DebouncedButton({ onClick, children }) {
return (
<button onClick={() => {
clearTimeout(timeoutID);
timeoutID = setTimeout(() => {
onClick();
}, 1000);
}}>
{children}
</button>
);
}

export default function Dashboard() {
return (
<>
<DebouncedButton
onClick={() => alert('Spaceship launched!')}
>
Launch the spaceship
```

</DebouncedButton>

<DebouncedButton

onClick={() => alert('Soup boiled!')}

>

Boil the soup

</DebouncedButton>

<DebouncedButton

onClick={() => alert('Lullaby sung!')}

>

Sing a lullaby

</DebouncedButton>

</>

)

}

```

```css

button { display: block; margin: 10px; }

```

</Sandpack>

<Solution>

A variable like `timeoutID` is shared between all components. This is why clicking on the second button resets the first button's pending timeout. To fix this, you can keep timeout in a ref. Each button will get its own ref, so they won't conflict with each other. Notice how clicking two buttons fast will show both messages.

<Sandpack>

```js

import { useRef } from 'react';

function DebouncedButton({ onClick, children }) {

const timeoutRef = useRef(null);

return (

<button onClick={() => {

clearTimeout(timeoutRef.current);

timeoutRef.current = setTimeout(() => {

onClick();

}, 1000);
}}>
{children}
</button>
);
}

export default function Dashboard() {
return (
<>
<DebouncedButton
onClick={() => alert('Spaceship launched!')}
>
Launch the spaceship
</DebouncedButton>
<DebouncedButton
onClick={() => alert('Soup boiled!')}
>
Boil the soup
</DebouncedButton>
<DebouncedButton
onClick={() => alert('Lullaby sung!')}
>
Sing a lullaby
</DebouncedButton>
</>
)
}
```

```css
button { display: block; margin: 10px; }
```

</Sandpack>

</Solution>

Read the latest state {/*read-the-latest-state*/}
```

In this example, after you press "Send", there is a small delay before the message is shown. Type "hello", press Send, and then quickly edit the input again. Despite your edits, the alert would still show "hello" (which was the value of state [at the time](/learn/state-as-a-snapshot#state-over-time) the button was clicked).

Usually, this behavior is what you want in an app. However, there may be occasional cases where you want some asynchronous code to read the *latest* version of some state. Can you think of a way to make the alert show the *current* input text rather than what it was at the time of the click?

<Sandpack>

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

export default function Chat() {
const [text, setText] = useState('');

function handleSend() {
setTimeout(() => {
alert('Sending: ' + text);
}, 3000);
}

return (
<>
<input
value={text}
onChange={e => setText(e.target.value)}
/>
<button
onClick={handleSend}>
Send
</button>
</>
);
}
```

</Sandpack>

<Solution>

State works [like a snapshot](/learn/state-as-a-snapshot), so you can't read the latest state from an asynchronous operation like a timeout. However, you can keep the latest input text in a ref. A ref is mutable, so you can read the `current` property at any time. Since the current text is also used for rendering, in this example, you will need *both* a state variable (for rendering), *and* a ref (to read it in the timeout). You will need to update the current ref value manually.

<Sandpack>

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

export default function Chat() {
const [text, setText] = useState('');
const textRef = useRef(text);

function handleChange(e) {
setText(e.target.value);
textRef.current = e.target.value;
}

function handleSend() {
setTimeout(() => {
alert('Sending: ' + textRef.current);
}, 3000);
}

return (
<>
<input
value={text}
onChange={handleChange}
/>
<button
onClick={handleSend}>
Send
</button>
</>
);
}
```

</Sandpack>

</Solution>

</Challenges>

---

title: Your First Component

<Intro>

Components are one of the core concepts of React. They are the foundation upon which you build user interfaces (UI), which makes them the perfect place to start your React journey!

</Intro>

<YouWillLearn>

* What a component is

* What role components play in a React application

* How to write your first React component

</YouWillLearn>

Components: UI building blocks {/*components-ui-building-blocks*/}

On the Web, HTML lets us create rich structured documents with its built-in set of tags like `<h1>` and `<li>`:

```html
<article>
<h1>My First Component</h1>
<ol>
<li>Components: UI Building Blocks</li>
<li>Defining a Component</li>
<li>Using a Component</li>
</ol>
</article>
```

This markup represents this article `<article>`, its heading `<h1>`, and an (abbreviated) table of contents as an ordered list `<ol>`. Markup like this, combined with CSS for style, and JavaScript for interactivity, lies behind every sidebar, avatar, modal, dropdown—every piece of UI you see on the Web.

React lets you combine your markup, CSS, and JavaScript into custom "components", **reusable UI elements for your app.** The table of contents code you saw above could be turned into a `<TableOfContents />` component you could render on every page. Under the hood, it still uses the same HTML tags like `<article>`, `<h1>`, etc.

Just like with HTML tags, you can compose, order and nest components to design whole pages. For example, the documentation page you're reading is made out of React components:

```js
<PageLayout>
<NavigationHeader>
<SearchBar />
<Link to="/docs">Docs</Link>
</NavigationHeader>
<Sidebar />
<PageContent>
<TableOfContents />
<DocumentationText />
</PageContent>
</PageLayout>
```

As your project grows, you will notice that many of your designs can be composed by reusing components you already wrote, speeding up your development. Our table of contents above could be added to any screen with `<TableOfContents />`! You can even jumpstart your project with the thousands of components shared by the React open source community like [Chakra UI](https://chakra-ui.com/) and [Material UI.](https://material-ui.com/)

Defining a component {/*defining-a-component*/}

Traditionally when creating web pages, web developers marked up their content and then added interaction by sprinkling on some JavaScript. This worked great when interaction was a nice-to-have on the web. Now it is expected for many sites and all apps. React puts interactivity first while still using the same technology: **a React component is a JavaScript function that you can _sprinkle with markup_.** Here's what that looks like (you can edit the example below):

<Sandpack>

```js
export default function Profile() {
return (
<img
src="https://i.imgur.com/MK3eW3Am.jpg"
alt="Katherine Johnson"
```

/>

)

}

```

```css
img { height: 200px; }
```

</Sandpack>

And here's how to build a component:

Step 1: Export the component {/*step-1-export-the-component*/}

The `export default` prefix is a [standard JavaScript syntax](https://developer.mozilla.org/docs/web/javascript/reference/statements/export) (not specific to React). It lets you mark the main function in a file so that you can later import it from other files. (More on importing in [Importing and Exporting Components](/learn/importing-and-exporting-components)!)

Step 2: Define the function {/*step-2-define-the-function*/}

With `function Profile() { }` you define a JavaScript function with the name `Profile`.

<Pitfall>

React components are regular JavaScript functions, but **their names must start with a capital letter** or they won't work!

</Pitfall>

Step 3: Add markup {/*step-3-add-markup*/}

The component returns an `<img />` tag with `src` and `alt` attributes. `<img />` is written like HTML, but it is actually JavaScript under the hood! This syntax is called [JSX](/learn/writing-markup-with-jsx), and it lets you embed markup inside JavaScript.

Return statements can be written all on one line, as in this component:

```js
return <img src="https://i.imgur.com/MK3eW3As.jpg" alt="Katherine Johnson" />;
```

But if your markup isn't all on the same line as the `return` keyword, you must wrap it in a pair of parentheses:

```js
return (

<img src="https://i.imgur.com/MK3eW3As.jpg" alt="Katherine Johnson" />

);

```

<Pitfall>

Without parentheses, any code on the lines after `return` [will be ignored](https://stackoverflow.com/questions/2846283/what-are-the-rules-for-javascripts-automatic-semicolon-insertion-asi)!

</Pitfall>

Using a component {/*using-a-component*/}

Now that you've defined your `Profile` component, you can nest it inside other components. For example, you can export a `Gallery` component that uses multiple `Profile` components:

<Sandpack>

```js
function Profile() {

return (

<img

src="https://i.imgur.com/MK3eW3As.jpg"

alt="Katherine Johnson"

/>

);

}

export default function Gallery() {

return (

<section>

<h1>Amazing scientists</h1>

<Profile />

<Profile />

<Profile />

</section>

);

}

```
```

```css
img { margin: 0 10px 10px 0; height: 90px; }
```

</Sandpack>

What the browser sees {/*what-the-browser-sees*/}

Notice the difference in casing:

* `<section>` is lowercase, so React knows we refer to an HTML tag.
* `<Profile />` starts with a capital `P`, so React knows that we want to use our component called `Profile`.

And `Profile` contains even more HTML: `<img />`. In the end, this is what the browser sees:

```html
<section>
<h1>Amazing scientists</h1>
<img src="https://i.imgur.com/MK3eW3As.jpg" alt="Katherine Johnson" />
<img src="https://i.imgur.com/MK3eW3As.jpg" alt="Katherine Johnson" />
<img src="https://i.imgur.com/MK3eW3As.jpg" alt="Katherine Johnson" />
</section>
```

Nesting and organizing components {/*nesting-and-organizing-components*/}

Components are regular JavaScript functions, so you can keep multiple components in the same file. This is convenient when components are relatively small or tightly related to each other. If this file gets crowded, you can always move `Profile` to a separate file. You will learn how to do this shortly on the [page about imports.](/learn/importing-and-exporting-components)

Because the `Profile` components are rendered inside `Gallery`—even several times!—we can say that `Gallery` is a **parent component,** rendering each `Profile` as a "child". This is part of the magic of React: you can define a component once, and then use it in as many places and as many times as you like.

<Pitfall>

Components can render other components, but **you must never nest their definitions:**

```js {2-5}
export default function Gallery() {
// ■ Never define a component inside another component!
function Profile() {
// ...
```

```
}

// ...

}
```

The snippet above is [very slow and causes bugs.](/learn/preserving-and-resetting-state#different-components-at-the-same-position-reset-state) Instead, define every component at the top level:

```js {5-8}
export default function Gallery() {

// ...

}

// ■ Declare components at the top level

function Profile() {

// ...

}
```

When a child component needs some data from a parent, [pass it by props](/learn/passing-props-to-a-component) instead of nesting definitions.

</Pitfall>

<DeepDive>

Components all the way down {/*components-all-the-way-down*/}

Your React application begins at a "root" component. Usually, it is created automatically when you start a new project. For example, if you use [CodeSandbox](https://codesandbox.io/) or [Create React App](https://create-react-app.dev/), the root component is defined in `src/App.js`. If you use the framework [Next.js](https://nextjs.org/), the root component is defined in `pages/index.js`. In these examples, you've been exporting root components.

Most React apps use components all the way down. This means that you won't only use components for reusable pieces like buttons, but also for larger pieces like sidebars, lists, and ultimately, complete pages! Components are a handy way to organize UI code and markup, even if some of them are only used once.

[React-based frameworks](/learn/start-a-new-react-project) take this a step further. Instead of using an empty HTML file and letting React "take over" managing the page with JavaScript, they *also* generate the HTML automatically from your React components. This allows your app to show some content before the JavaScript code loads.

Still, many websites only use React to [add interactivity to existing HTML pages.](/learn/add-react-to-an-existing-project#using-react-for-a-part-of-your-existing-page) They have many root components instead of a single one for the entire page. You can use as much—or as

little—React as you need.

</DeepDive>

<Recap>

You've just gotten your first taste of React! Let's recap some key points.

* React lets you create components, **reusable UI elements for your app.**
* In a React app, every piece of UI is a component.
* React components are regular JavaScript functions except:

1. Their names always begin with a capital letter.
2. They return JSX markup.

</Recap>

<Challenges>

Export the component {/*export-the-component*/}

This sandbox doesn't work because the root component is not exported:

<Sandpack>

```js
function Profile() {
return (
<img
src="https://i.imgur.com/lICfvbD.jpg"
alt="Aklilu Lemma"
/>
);
}
```

```css
img { height: 181px; }
```

</Sandpack>

Try to fix it yourself before looking at the solution!

<Solution>

Add `export default` before the function definition like so:

<Sandpack>

```js
export default function Profile() {
return (
<img
src="https://i.imgur.com/lICfvbD.jpg"
alt="Aklilu Lemma"
/>
);
}
```

```css
img { height: 181px; }
```

</Sandpack>

You might be wondering why writing `export` alone is not enough to fix this example. You can learn the difference between `export` and `export default` in [Importing and Exporting Components.](/learn/importing-and-exporting-components)

</Solution>

Fix the return statement {/*fix-the-return-statement*/}

Something isn't right about this `return` statement. Can you fix it?

<Hint>

You may get an "Unexpected token" error while trying to fix this. In that case, check that the semicolon appears *after* the closing parenthesis. Leaving a semicolon inside `return ( )` will cause an error.

</Hint>

<Sandpack>

```js
export default function Profile() {
return
<img src="https://i.imgur.com/jA8hHMpm.jpg" alt="Katsuko Saruhashi" />;
}
```

```

```css
img { height: 180px; }
```

</Sandpack>

<Solution>

You can fix this component by moving the return statement to one line like so:

<Sandpack>

```js
export default function Profile() {
return <img src="https://i.imgur.com/jA8hHMpm.jpg" alt="Katsuko Saruhashi" />;
}
```

```css
img { height: 180px; }
```

</Sandpack>

Or by wrapping the returned JSX markup in parentheses that open right after `return`:

<Sandpack>

```js
export default function Profile() {
return (
<img
src="https://i.imgur.com/jA8hHMpm.jpg"
alt="Katsuko Saruhashi"
/>
);
}
```

```css
img { height: 180px; }
```
```

```

</Sandpack>

</Solution>

#### Spot the mistake {/*spot-the-mistake*/}

Something's wrong with how the `Profile` component is declared and used. Can you spot the mistake? (Try to remember how React distinguishes components from the regular HTML tags!)

<Sandpack>

```js
function profile() {
return (
<img
src="https://i.imgur.com/QIrZWGIs.jpg"
alt="Alan L. Hart"
/>
);
}

export default function Gallery() {
return (
<section>
<h1>Amazing scientists</h1>
<profile />
<profile />
<profile />
</section>
);
}
```

```css
img { margin: 0 10px 10px 0; height: 90px; }
```

</Sandpack>

<Solution>
```

React component names must start with a capital letter.

Change `function profile()` to `function Profile()`, and then change every `<profile />` to `<Profile />`:

<Sandpack>

```js
function Profile() {
return (
<img
src="https://i.imgur.com/QIrZWGIs.jpg"
alt="Alan L. Hart"
/>
);
}

export default function Gallery() {
return (
<section>
<h1>Amazing scientists</h1>
<Profile />
<Profile />
<Profile />
</section>
);
}
```

```css
img { margin: 0 10px 10px 0; }
```

</Sandpack>

</Solution>

Your own component {/*your-own-component*/}

Write a component from scratch. You can give it any valid name and return any markup. If you're out of ideas, you can write a `Congratulations` component that shows `<h1>Good job!</h1>`. Don't forget to export it!

<Sandpack>

```js
// Write your component below!
```

</Sandpack>

<Solution>

<Sandpack>

```js
export default function Congratulations() {
return (
<h1>Good job!</h1>
);
}
```

</Sandpack>

</Solution>

</Challenges>

---
title: Updating Arrays in State

<Intro>

Arrays are mutable in JavaScript, but you should treat them as immutable when you store them in state. Just like with objects, when you want to update an array stored in state, you need to create a new one (or make a copy of an existing one), and then set state to use the new array.

</Intro>

<YouWillLearn>

- How to add, remove, or change items in an array in React state
- How to update an object inside of an array
- How to make array copying less repetitive with Immer

</YouWillLearn>

Updating arrays without mutation {/*updating-arrays-without-mutation*/}

In JavaScript, arrays are just another kind of object. [Like with objects](/learn/updating-objects-in-state), **you should treat arrays in React state as read-only.** This means that you shouldn't reassign items inside an array like `arr[0] = 'bird'`, and you also shouldn't use methods that mutate the array, such as `push()` and `pop()`.

Instead, every time you want to update an array, you'll want to pass a *new* array to your state setting function. To do that, you can create a new array from the original array in your state by calling its non-mutating methods like `filter()` and `map()`. Then you can set your state to the resulting new array.

Here is a reference table of common array operations. When dealing with arrays inside React state, you will need to avoid the methods in the left column, and instead prefer the methods in the right column:

| | avoid (mutates the array) | prefer (returns a new array) |

| --- | --- | --- |

| adding | `push`, `unshift` | `concat`, `[...arr]` spread syntax ([example](#adding-to-an-array)) |

| removing | `pop`, `shift`, `splice` | `filter`, `slice` ([example](#removing-from-an-array)) |

| replacing | `splice`, `arr[i] = ...` assignment | `map` ([example](#replacing-items-in-an-array)) |

| sorting | `reverse`, `sort` | copy the array first ([example](#making-other-changes-to-an-array)) |

Alternatively, you can [use Immer](#write-concise-update-logic-with-immer) which lets you use methods from both columns.

<Pitfall>

Unfortunately, [`slice`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/slice) and [`splice`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/splice) are named similarly but are very different:

* `slice` lets you copy an array or a part of it.

* `splice` **mutates** the array (to insert or delete items).

In React, you will be using `slice` (no `p`!) a lot more often because you don't want to mutate objects or arrays in state. [Updating Objects](/learn/updating-objects-in-state) explains what mutation is and why it's not recommended for state.

</Pitfall>

Adding to an array {/*adding-to-an-array*/}

`push()` will mutate an array, which you don't want:

<Sandpack>

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

let nextId = 0;
```

```
export default function List() {
const [name, setName] = useState('');
const [artists, setArtists] = useState([]);

return (
<>
<h1>Inspiring sculptors:</h1>
<input
value={name}
onChange={e => setName(e.target.value)}
/>
<button onClick={() => {
artists.push({
id: nextId++,
name: name,
});
}}>Add</button>
<ul>
{artists.map(artist => (
<li key={artist.id}>{artist.name}</li>
))}
</ul>
</>
);
}
```

```css
button { margin-left: 5px; }
```

</Sandpack>

Instead, create a *new* array which contains the existing items *and* a new item at the end. There are multiple ways to do this, but the easiest one is to use the `...` [array spread](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax#spread_in_array_literals) syntax:

```js
setArtists( // Replace the state
```

```
[ // with a new array
...artists, // that contains all the old items
{ id: nextId++, name: name } // and one new item at the end
]
);
```

Now it works correctly:

<Sandpack>

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

let nextId = 0;

export default function List() {
const [name, setName] = useState('');
const [artists, setArtists] = useState([]);

return (
<>
<h1>Inspiring sculptors:</h1>
<input
value={name}
onChange={e => setName(e.target.value)}
/>
<button onClick={() => {
setArtists([
...artists,
{ id: nextId++, name: name }
]);
}}>Add</button>
<ul>
{artists.map(artist => (
<li key={artist.id}>{artist.name}</li>
))}
</ul>
</>
```

);
}
```

```css
button { margin-left: 5px; }
```

</Sandpack>

The array spread syntax also lets you prepend an item by placing it *before* the original `...artists`:

```js
setArtists([
{ id: nextId++, name: name },
...artists // Put old items at the end
]);
```

In this way, spread can do the job of both `push()` by adding to the end of an array and `unshift()` by adding to the beginning of an array. Try it in the sandbox above!

Removing from an array {/*removing-from-an-array*/}

The easiest way to remove an item from an array is to *filter it out*. In other words, you will produce a new array that will not contain that item. To do this, use the `filter` method, for example:

<Sandpack>

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

let initialArtists = [
{ id: 0, name: 'Marta Colvin Andrade' },
{ id: 1, name: 'Lamidi Olonade Fakeye'},
{ id: 2, name: 'Louise Nevelson'},
];

export default function List() {
const [artists, setArtists] = useState(
initialArtists
);

return (

```
<>
<h1>Inspiring sculptors:</h1>
<ul>
{artists.map(artist => (
<li key={artist.id}>
{artist.name}{' '}
<button onClick={() => {
setArtists(
artists.filter(a =>
a.id !== artist.id
)
);
}}>
Delete
</button>
</li>
))}
</ul>
</>
);
}
```

</Sandpack>

Click the "Delete" button a few times, and look at its click handler.

```js
setArtists(
artists.filter(a => a.id !== artist.id)
);
```

Here, `artists.filter(a => a.id !== artist.id)` means "create an array that consists of those `artists` whose IDs are different from `artist.id`". In other words, each artist's "Delete" button will filter _that_ artist out of the array, and then request a re-render with the resulting array. Note that `filter` does not modify the original array.

Transforming an array {/*transforming-an-array*/}

If you want to change some or all items of the array, you can use `map()` to create a **new** array. The function you will pass to `map` can decide what to do with each item, based on its data or its index (or both).

In this example, an array holds coordinates of two circles and a square. When you press the button, it moves only the circles down by 50 pixels. It does this by producing a new array of data using `map()`:

<Sandpack>

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

let initialShapes = [
{ id: 0, type: 'circle', x: 50, y: 100 },
{ id: 1, type: 'square', x: 150, y: 100 },
{ id: 2, type: 'circle', x: 250, y: 100 },
];

export default function ShapeEditor() {
const [shapes, setShapes] = useState(
initialShapes
);

function handleClick() {
const nextShapes = shapes.map(shape => {
if (shape.type === 'square') {
// No change
return shape;
} else {
// Return a new circle 50px below
return {
...shape,
y: shape.y + 50,
};
}
});
// Re-render with the new array
setShapes(nextShapes);
}

return (
```

```
<>
<button onClick={handleClick}>
Move circles down!
</button>
{shapes.map(shape => (
<div
key={shape.id}
style={{
background: 'purple',
position: 'absolute',
left: shape.x,
top: shape.y,
borderRadius:
shape.type === 'circle'
? '50%' : '',
width: 20,
height: 20,
}} />
))}
</>
);
}
```

```css
body { height: 300px; }
```

</Sandpack>

Replacing items in an array {/*replacing-items-in-an-array*/}

It is particularly common to want to replace one or more items in an array. Assignments like `arr[0] = 'bird'` are mutating the original array, so instead you'll want to use `map` for this as well.

To replace an item, create a new array with `map`. Inside your `map` call, you will receive the item index as the second argument. Use it to decide whether to return the original item (the first argument) or something else:

<Sandpack>

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

let initialCounters = [
0, 0, 0
];

export default function CounterList() {
const [counters, setCounters] = useState(
initialCounters
);

function handleIncrementClick(index) {
const nextCounters = counters.map((c, i) => {
if (i === index) {
// Increment the clicked counter
return c + 1;
} else {
// The rest haven't changed
return c;
}
});
setCounters(nextCounters);
}

return (
<ul>
{counters.map((counter, i) => (
<li key={i}>
{counter}
<button onClick={() => {
handleIncrementClick(i);
}}>+1</button>
</li>
))}
</ul>
);
}
```

```

```css
button { margin: 5px; }
```

</Sandpack>

Inserting into an array {/*inserting-into-an-array*/}

Sometimes, you may want to insert an item at a particular position that's neither at the beginning nor at the end. To do this, you can use the `...` array spread syntax together with the `slice()` method. The `slice()` method lets you cut a "slice" of the array. To insert an item, you will create an array that spreads the slice _before_ the insertion point, then the new item, and then the rest of the original array.

In this example, the Insert button always inserts at the index `1`:

<Sandpack>

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

let nextId = 3;
const initialArtists = [
{ id: 0, name: 'Marta Colvin Andrade' },
{ id: 1, name: 'Lamidi Olonade Fakeye'},
{ id: 2, name: 'Louise Nevelson'},
];

export default function List() {
const [name, setName] = useState('');
const [artists, setArtists] = useState(
initialArtists
);

function handleClick() {
const insertAt = 1; // Could be any index
const nextArtists = [
// Items before the insertion point:
...artists.slice(0, insertAt),
// New item:
{ id: nextId++, name: name },
// Items after the insertion point:

```
...artists.slice(insertAt)
];
setArtists(nextArtists);
setName('');
}

return (
<>
<h1>Inspiring sculptors:</h1>
<input
value={name}
onChange={e => setName(e.target.value)}
/>
<button onClick={handleClick}>
Insert
</button>
<ul>
{artists.map(artist => (
<li key={artist.id}>{artist.name}</li>
))}
</ul>
</>
);
}
```

```css
button { margin-left: 5px; }
```

</Sandpack>

Making other changes to an array {/*making-other-changes-to-an-array*/}

There are some things you can't do with the spread syntax and non-mutating methods like `map()` and `filter()` alone. For example, you may want to reverse or sort an array. The JavaScript `reverse()` and `sort()` methods are mutating the original array, so you can't use them directly.

However, you can copy the array first, and then make changes to it.

For example:

<Sandpack>

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

let nextId = 3;
const initialList = [
{ id: 0, title: 'Big Bellies' },
{ id: 1, title: 'Lunar Landscape' },
{ id: 2, title: 'Terracotta Army' },
];

export default function List() {
const [list, setList] = useState(initialList);

function handleClick() {
const nextList = [...list];
nextList.reverse();
setList(nextList);
}

return (
<>
<button onClick={handleClick}>
Reverse
</button>
<ul>
{list.map(artwork => (
<li key={artwork.id}>{artwork.title}</li>
))}
</ul>
</>
);
}
```

</Sandpack>

Here, you use the `[...list]` spread syntax to create a copy of the original array first. Now that you have a copy, you can use mutating methods like `nextList.reverse()` or `nextList.sort()`, or even assign individual items with `nextList[0] = "something"`.

However, **even if you copy an array, you can't mutate existing items _inside_ of it directly.** This is because copying is shallow--the new array will contain the same items as the original one. So if you modify an object inside the copied array, you are mutating the existing state. For example, code like this is a problem.

```js
const nextList = [...list];
nextList[0].seen = true; // Problem: mutates list[0]
setList(nextList);
```

Although `nextList` and `list` are two different arrays, **`nextList[0]` and `list[0]` point to the same object.** So by changing `nextList[0].seen`, you are also changing `list[0].seen`. This is a state mutation, which you should avoid! You can solve this issue in a similar way to [updating nested JavaScript objects](/learn/updating-objects-in-state#updating-a-nested-object)--by copying individual items you want to change instead of mutating them. Here's how.

Updating objects inside arrays {/*updating-objects-inside-arrays*/}

Objects are not _really_ located "inside" arrays. They might appear to be "inside" in code, but each object in an array is a separate value, to which the array "points". This is why you need to be careful when changing nested fields like `list[0]`. Another person's artwork list may point to the same element of the array!

When updating nested state, you need to create copies from the point where you want to update, and all the way up to the top level. Let's see how this works.

In this example, two separate artwork lists have the same initial state. They are supposed to be isolated, but because of a mutation, their state is accidentally shared, and checking a box in one list affects the other list:

<Sandpack>

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

let nextId = 3;
const initialList = [
{ id: 0, title: 'Big Bellies', seen: false },
{ id: 1, title: 'Lunar Landscape', seen: false },
{ id: 2, title: 'Terracotta Army', seen: true },
];

export default function BucketList() {
const [myList, setMyList] = useState(initialList);
const [yourList, setYourList] = useState(
```

```
initialList
);

function handleToggleMyList(artworkId, nextSeen) {
const myNextList = [...myList];
const artwork = myNextList.find(
a => a.id === artworkId
);
artwork.seen = nextSeen;
setMyList(myNextList);
}

function handleToggleYourList(artworkId, nextSeen) {
const yourNextList = [...yourList];
const artwork = yourNextList.find(
a => a.id === artworkId
);
artwork.seen = nextSeen;
setYourList(yourNextList);
}

return (
<>
<h1>Art Bucket List</h1>
<h2>My list of art to see:</h2>
<ItemList
artworks={myList}
onToggle={handleToggleMyList} />
<h2>Your list of art to see:</h2>
<ItemList
artworks={yourList}
onToggle={handleToggleYourList} />
</>
);
}

function ItemList({ artworks, onToggle }) {
return (
```

```
<ul>
{artworks.map(artwork => (
<li key={artwork.id}>
<label>
<input
type="checkbox"
checked={artwork.seen}
onChange={e => {
onToggle(
artwork.id,
e.target.checked
);
}}
/>
{artwork.title}
</label>
</li>
))}
</ul>
);
}
```

</Sandpack>

The problem is in code like this:

```js
const myNextList = [...myList];
const artwork = myNextList.find(a => a.id === artworkId);
artwork.seen = nextSeen; // Problem: mutates an existing item
setMyList(myNextList);
```

Although the `myNextList` array itself is new, the *items themselves* are the same as in the original `myList` array. So changing `artwork.seen` changes the *original* artwork item. That artwork item is also in `yourList`, which causes the bug. Bugs like this can be difficult to think about, but thankfully they disappear if you avoid mutating state.

You can use `map` to substitute an old item with its updated version without mutation.

```js
setMyList(myList.map(artwork => {
if (artwork.id === artworkId) {
// Create a *new* object with changes
return { ...artwork, seen: nextSeen };
} else {
// No changes
return artwork;
}
}));
```

Here, `...` is the object spread syntax used to [create a copy of an object.](/learn/updating-objects-in-state#copying-objects-with-the-spread-syntax)

With this approach, none of the existing state items are being mutated, and the bug is fixed:

<Sandpack>

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

let nextId = 3;
const initialList = [
{ id: 0, title: 'Big Bellies', seen: false },
{ id: 1, title: 'Lunar Landscape', seen: false },
{ id: 2, title: 'Terracotta Army', seen: true },
];

export default function BucketList() {
const [myList, setMyList] = useState(initialList);
const [yourList, setYourList] = useState(
initialList
);

function handleToggleMyList(artworkId, nextSeen) {
setMyList(myList.map(artwork => {
if (artwork.id === artworkId) {
// Create a *new* object with changes
return { ...artwork, seen: nextSeen };
```

```
} else {
// No changes
return artwork;
}
}));
}

function handleToggleYourList(artworkId, nextSeen) {
setYourList(yourList.map(artwork => {
if (artwork.id === artworkId) {
// Create a *new* object with changes
return { ...artwork, seen: nextSeen };
} else {
// No changes
return artwork;
}
}));
}

return (
<>
<h1>Art Bucket List</h1>
<h2>My list of art to see:</h2>
<ItemList
artworks={myList}
onToggle={handleToggleMyList} />
<h2>Your list of art to see:</h2>
<ItemList
artworks={yourList}
onToggle={handleToggleYourList} />
</>
);
}

function ItemList({ artworks, onToggle }) {
return (
<ul>
```

```
{artworks.map(artwork => (
<li key={artwork.id}>
<label>
<input
type="checkbox"
checked={artwork.seen}
onChange={e => {
onToggle(
artwork.id,
e.target.checked
);
}}
/>
{artwork.title}
</label>
</li>
))}
</ul>
);
}
```

</Sandpack>

In general, **you should only mutate objects that you have just created.** If you were inserting a *new* artwork, you could mutate it, but if you're dealing with something that's already in state, you need to make a copy.

Write concise update logic with Immer {/*write-concise-update-logic-with-immer*/}

Updating nested arrays without mutation can get a little bit repetitive. [Just as with objects](/learn/updating-objects-in-state#write-concise-update-logic-with-immer):

- Generally, you shouldn't need to update state more than a couple of levels deep. If your state objects are very deep, you might want to [restructure them differently](/learn/choosing-the-state-structure#avoid-deeply-nested-state) so that they are flat.
- If you don't want to change your state structure, you might prefer to use [Immer](https://github.com/immerjs/use-immer), which lets you write using the convenient but mutating syntax and takes care of producing the copies for you.

Here is the Art Bucket List example rewritten with Immer:

<Sandpack>

```js
import { useState } from 'react';
import { useImmer } from 'use-immer';

let nextId = 3;
const initialList = [
{ id: 0, title: 'Big Bellies', seen: false },
{ id: 1, title: 'Lunar Landscape', seen: false },
{ id: 2, title: 'Terracotta Army', seen: true },
];

export default function BucketList() {
const [myList, updateMyList] = useImmer(
initialList
);
const [yourList, updateYourList] = useImmer(
initialList
);

function handleToggleMyList(id, nextSeen) {
updateMyList(draft => {
const artwork = draft.find(a =>
a.id === id
);
artwork.seen = nextSeen;
});
}

function handleToggleYourList(artworkId, nextSeen) {
updateYourList(draft => {
const artwork = draft.find(a =>
a.id === artworkId
);
artwork.seen = nextSeen;
});
}
```

```
return (
<>
<h1>Art Bucket List</h1>
<h2>My list of art to see:</h2>
<ItemList
artworks={myList}
onToggle={handleToggleMyList} />
<h2>Your list of art to see:</h2>
<ItemList
artworks={yourList}
onToggle={handleToggleYourList} />
</>
);
}

function ItemList({ artworks, onToggle }) {
return (
<ul>
{artworks.map(artwork => (
<li key={artwork.id}>
<label>
<input
type="checkbox"
checked={artwork.seen}
onChange={e => {
onToggle(
artwork.id,
e.target.checked
);
}}
/>
{artwork.title}
</label>
</li>
))}
</ul>
```

);
}
```

```json package.json
{
"dependencies": {
"immer": "1.7.3",
"react": "latest",
"react-dom": "latest",
"react-scripts": "latest",
"use-immer": "0.5.1"
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
}
}
```

</Sandpack>

Note how with Immer, **mutation like `artwork.seen = nextSeen` is now okay:**

```js
updateMyTodos(draft => {
const artwork = draft.find(a => a.id === artworkId);
artwork.seen = nextSeen;
});
```

This is because you're not mutating the _original_ state, but you're mutating a special `draft` object provided by Immer. Similarly, you can apply mutating methods like `push()` and `pop()` to the content of the `draft`.

Behind the scenes, Immer always constructs the next state from scratch according to the changes that you've done to the `draft`. This keeps your event handlers very concise without ever mutating state.

<Recap>
```

- You can put arrays into state, but you can't change them.
- Instead of mutating an array, create a *new* version of it, and update the state to it.
- You can use the `[...arr, newItem]` array spread syntax to create arrays with new items.
- You can use `filter()` and `map()` to create new arrays with filtered or transformed items.
- You can use Immer to keep your code concise.

</Recap>

<Challenges>

Update an item in the shopping cart {/*update-an-item-in-the-shopping-cart*/}

Fill in the `handleIncreaseClick` logic so that pressing "+" increases the corresponding number:

<Sandpack>

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

const initialProducts = [{
id: 0,
name: 'Baklava',
count: 1,
}, {
id: 1,
name: 'Cheese',
count: 5,
}, {
id: 2,
name: 'Spaghetti',
count: 2,
}];

export default function ShoppingCart() {
const [
products,
setProducts
] = useState(initialProducts)

function handleIncreaseClick(productId) {
```

}

return (

{products.map(product => (
<li key={product.id}>
{product.name}
{' '}
({product.count})
<button onClick={() => {
handleIncreaseClick(product.id);
}}>
+
</button>

))}

);
}
```

```css
button { margin: 5px; }
```

</Sandpack>

<Solution>

You can use the `map` function to create a new array, and then use the `...` object spread syntax to create a copy of the changed object for the new array:

<Sandpack>

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

const initialProducts = [{
id: 0,
name: 'Baklava',
count: 1,

```
}, {
id: 1,
name: 'Cheese',
count: 5,
}, {
id: 2,
name: 'Spaghetti',
count: 2,
}];

export default function ShoppingCart() {
const [
products,
setProducts
] = useState(initialProducts)

function handleIncreaseClick(productId) {
setProducts(products.map(product => {
if (product.id === productId) {
return {
...product,
count: product.count + 1
};
} else {
return product;
}
}))
}

return (
<ul>
{products.map(product => (
<li key={product.id}>
{product.name}
{' '}
(<b>{product.count}</b>)
<button onClick={() => {
```

handleIncreaseClick(product.id);

}}>

+

</button>

))}

);

}

```

```css

button { margin: 5px; }

```

</Sandpack>

</Solution>

Remove an item from the shopping cart {/*remove-an-item-from-the-shopping-cart*/}

This shopping cart has a working "+" button, but the "–" button doesn't do anything. You need to add an event handler to it so that pressing it decreases the `count` of the corresponding product. If you press "–" when the count is 1, the product should automatically get removed from the cart. Make sure it never shows 0.

<Sandpack>

```js

import { useState } from 'react';

const initialProducts = [{

id: 0,

name: 'Baklava',

count: 1,

}, {

id: 1,

name: 'Cheese',

count: 5,

}, {

id: 2,

name: 'Spaghetti',

```
count: 2,
}];

export default function ShoppingCart() {
const [
products,
setProducts
] = useState(initialProducts)

function handleIncreaseClick(productId) {
setProducts(products.map(product => {
if (product.id === productId) {
return {
...product,
count: product.count + 1
};
} else {
return product;
}
}))
}

return (

{products.map(product => (
<li key={product.id}>
{product.name}
{' '}
({product.count})
<button onClick={() => {
handleIncreaseClick(product.id);
}}>
+
</button>
<button>
–
</button>
```

))}

);
}
```

```css
button { margin: 5px; }
```

</Sandpack>

<Solution>

You can first use `map` to produce a new array, and then `filter` to remove products with a `count` set to `0`:

<Sandpack>

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

const initialProducts = [{
id: 0,
name: 'Baklava',
count: 1,
}, {
id: 1,
name: 'Cheese',
count: 5,
}, {
id: 2,
name: 'Spaghetti',
count: 2,
}];

export default function ShoppingCart() {
const [
products,
setProducts

```
] = useState(initialProducts)

function handleIncreaseClick(productId) {
setProducts(products.map(product => {
if (product.id === productId) {
return {
...product,
count: product.count + 1
};
} else {
return product;
}
}))
}

function handleDecreaseClick(productId) {
let nextProducts = products.map(product => {
if (product.id === productId) {
return {
...product,
count: product.count - 1
};
} else {
return product;
}
});
nextProducts = nextProducts.filter(p =>
p.count > 0
);
setProducts(nextProducts)
}

return (
<ul>
{products.map(product => (
<li key={product.id}>
{product.name}
```

{' '}
({product.count})
<button onClick={() => {
handleIncreaseClick(product.id);
}}>
+
</button>
<button onClick={() => {
handleDecreaseClick(product.id);
}}>
–
</button>

))}

);
}
```

```css
button { margin: 5px; }
```

</Sandpack>

</Solution>

Fix the mutations using non-mutative methods {/*fix-the-mutations-using-non-mutative-methods*/}

In this example, all of the event handlers in `App.js` use mutation. As a result, editing and deleting todos doesn't work. Rewrite `handleAddTodo`, `handleChangeTodo`, and `handleDeleteTodo` to use the non-mutative methods:

<Sandpack>

```js App.js
import { useState } from 'react';
import AddTodo from './AddTodo.js';
import TaskList from './TaskList.js';

let nextId = 3;

```
const initialTodos = [
{ id: 0, title: 'Buy milk', done: true },
{ id: 1, title: 'Eat tacos', done: false },
{ id: 2, title: 'Brew tea', done: false },
];

export default function TaskApp() {
const [todos, setTodos] = useState(
initialTodos
);

function handleAddTodo(title) {
todos.push({
id: nextId++,
title: title,
done: false
});
}

function handleChangeTodo(nextTodo) {
const todo = todos.find(t =>
t.id === nextTodo.id
);
todo.title = nextTodo.title;
todo.done = nextTodo.done;
}

function handleDeleteTodo(todoId) {
const index = todos.findIndex(t =>
t.id === todoId
);
todos.splice(index, 1);
}

return (
<>
<AddTodo
onAddTodo={handleAddTodo}
/>
```

```
<TaskList
todos={todos}
onChangeTodo={handleChangeTodo}
onDeleteTodo={handleDeleteTodo}
/>
</>
);
}
```

```js AddTodo.js
import { useState } from 'react';

export default function AddTodo({ onAddTodo }) {
const [title, setTitle] = useState('');
return (
<>
<input
placeholder="Add todo"
value={title}
onChange={e => setTitle(e.target.value)}
/>
<button onClick={() => {
setTitle('');
onAddTodo(title);
}}>Add</button>
</>
)
}
```

```js TaskList.js
import { useState } from 'react';

export default function TaskList({
todos,
onChangeTodo,
onDeleteTodo
```

```
}) {
return (
<ul>
{todos.map(todo => (
<li key={todo.id}>
<Task
todo={todo}
onChange={onChangeTodo}
onDelete={onDeleteTodo}
/>
</li>
))}
</ul>
);
}

function Task({ todo, onChange, onDelete }) {
const [isEditing, setIsEditing] = useState(false);
let todoContent;
if (isEditing) {
todoContent = (
<>
<input
value={todo.title}
onChange={e => {
onChange({
...todo,
title: e.target.value
});
}} />
<button onClick={() => setIsEditing(false)}>
Save
</button>
</>
);
} else {
```

```
todoContent = (
<>
{todo.title}
<button onClick={() => setIsEditing(true)}>
Edit
</button>
</>
);
}
return (
<label>
<input
type="checkbox"
checked={todo.done}
onChange={e => {
onChange({
...todo,
done: e.target.checked
});
}}
/>
{todoContent}
<button onClick={() => onDelete(todo.id)}>
Delete
</button>
</label>
);
}
```

```css
button { margin: 5px; }
li { list-style-type: none; }
ul, li { margin: 0; padding: 0; }
```

</Sandpack>

<Solution>

In `handleAddTodo`, you can use the array spread syntax. In `handleChangeTodo`, you can create a new array with `map`. In `handleDeleteTodo`, you can create a new array with `filter`. Now the list works correctly:

<Sandpack>

```js App.js
import { useState } from 'react';
import AddTodo from './AddTodo.js';
import TaskList from './TaskList.js';

let nextId = 3;
const initialTodos = [
{ id: 0, title: 'Buy milk', done: true },
{ id: 1, title: 'Eat tacos', done: false },
{ id: 2, title: 'Brew tea', done: false },
];

export default function TaskApp() {
const [todos, setTodos] = useState(
initialTodos
);

function handleAddTodo(title) {
setTodos([
...todos,
{
id: nextId++,
title: title,
done: false
}
]);
}

function handleChangeTodo(nextTodo) {
setTodos(todos.map(t => {
if (t.id === nextTodo.id) {
return nextTodo;
} else {
```

```
return t;
}
}));
}

function handleDeleteTodo(todoId) {
setTodos(
todos.filter(t => t.id !== todoId)
);
}

return (
<>
<AddTodo
onAddTodo={handleAddTodo}
/>
<TaskList
todos={todos}
onChangeTodo={handleChangeTodo}
onDeleteTodo={handleDeleteTodo}
/>
</>
);
}
```

```js AddTodo.js
import { useState } from 'react';

export default function AddTodo({ onAddTodo }) {
const [title, setTitle] = useState('');
return (
<>
<input
placeholder="Add todo"
value={title}
onChange={e => setTitle(e.target.value)}
/>
```

```
<button onClick={() => {
setTitle('');
onAddTodo(title);
}}>Add</button>
</>
)
}
```

```js TaskList.js
import { useState } from 'react';

export default function TaskList({
todos,
onChangeTodo,
onDeleteTodo
}) {
return (
<ul>
{todos.map(todo => (
<li key={todo.id}>
<Task
todo={todo}
onChange={onChangeTodo}
onDelete={onDeleteTodo}
/>
</li>
))}
</ul>
);
}

function Task({ todo, onChange, onDelete }) {
const [isEditing, setIsEditing] = useState(false);
let todoContent;
if (isEditing) {
todoContent = (
```

```
<>
<input
value={todo.title}
onChange={e => {
onChange({
...todo,
title: e.target.value
});
}} />
<button onClick={() => setIsEditing(false)}>
Save
</button>
</>
);
} else {
todoContent = (
<>
{todo.title}
<button onClick={() => setIsEditing(true)}>
Edit
</button>
</>
);
}
return (
<label>
<input
type="checkbox"
checked={todo.done}
onChange={e => {
onChange({
...todo,
done: e.target.checked
});
}}
```

/>

{todoContent}

<button onClick={() => onDelete(todo.id)}>

Delete

</button>

</label>

);

}

```

```css

button { margin: 5px; }

li { list-style-type: none; }

ul, li { margin: 0; padding: 0; }

```

</Sandpack>

</Solution>

Fix the mutations using Immer {/*fix-the-mutations-using-immer*/}

This is the same example as in the previous challenge. This time, fix the mutations by using Immer. For your convenience, `useImmer` is already imported, so you need to change the `todos` state variable to use it.

<Sandpack>

```js App.js

import { useState } from 'react';

import { useImmer } from 'use-immer';

import AddTodo from './AddTodo.js';

import TaskList from './TaskList.js';

let nextId = 3;

const initialTodos = [

{ id: 0, title: 'Buy milk', done: true },

{ id: 1, title: 'Eat tacos', done: false },

{ id: 2, title: 'Brew tea', done: false },

];

export default function TaskApp() {

```
const [todos, setTodos] = useState(
initialTodos
);

function handleAddTodo(title) {
todos.push({
id: nextId++,
title: title,
done: false
});
}

function handleChangeTodo(nextTodo) {
const todo = todos.find(t =>
t.id === nextTodo.id
);
todo.title = nextTodo.title;
todo.done = nextTodo.done;
}

function handleDeleteTodo(todoId) {
const index = todos.findIndex(t =>
t.id === todoId
);
todos.splice(index, 1);
}

return (
<>
<AddTodo
onAddTodo={handleAddTodo}
/>
<TaskList
todos={todos}
onChangeTodo={handleChangeTodo}
onDeleteTodo={handleDeleteTodo}
/>
</>
```

```
);
}
```

```js AddTodo.js
import { useState } from 'react';

export default function AddTodo({ onAddTodo }) {
const [title, setTitle] = useState('');
return (
<>
<input
placeholder="Add todo"
value={title}
onChange={e => setTitle(e.target.value)}
/>
<button onClick={() => {
setTitle('');
onAddTodo(title);
}}>Add</button>
</>
)
}
```

```js TaskList.js
import { useState } from 'react';

export default function TaskList({
todos,
onChangeTodo,
onDeleteTodo
}) {
return (
<ul>
{todos.map(todo => (
<li key={todo.id}>
<Task
```

```
todo={todo}
onChange={onChangeTodo}
onDelete={onDeleteTodo}
/>
</li>
))}
</ul>
);
}

function Task({ todo, onChange, onDelete }) {
const [isEditing, setIsEditing] = useState(false);
let todoContent;
if (isEditing) {
todoContent = (
<>
<input
value={todo.title}
onChange={e => {
onChange({
...todo,
title: e.target.value
});
}} />
<button onClick={() => setIsEditing(false)}>
Save
</button>
</>
);
} else {
todoContent = (
<>
{todo.title}
<button onClick={() => setIsEditing(true)}>
Edit
</button>
```

```
</>
);
}
return (
<label>
<input
type="checkbox"
checked={todo.done}
onChange={e => {
onChange({
...todo,
done: e.target.checked
});
}}
/>
{todoContent}
<button onClick={() => onDelete(todo.id)}>
Delete
</button>
</label>
);
}
```

```css
button { margin: 5px; }
li { list-style-type: none; }
ul, li { margin: 0; padding: 0; }
```

```json package.json
{
"dependencies": {
"immer": "1.7.3",
"react": "latest",
"react-dom": "latest",
"react-scripts": "latest",
```

"use-immer": "0.5.1"

},

"scripts": {

"start": "react-scripts start",

"build": "react-scripts build",

"test": "react-scripts test --env=jsdom",

"eject": "react-scripts eject"

}

}

```

</Sandpack>

<Solution>

With Immer, you can write code in the mutative fashion, as long as you're only mutating parts of the `draft` that Immer gives you. Here, all mutations are performed on the `draft` so the code works:

<Sandpack>

```js App.js

import { useState } from 'react';

import { useImmer } from 'use-immer';

import AddTodo from './AddTodo.js';

import TaskList from './TaskList.js';

let nextId = 3;

const initialTodos = [

{ id: 0, title: 'Buy milk', done: true },

{ id: 1, title: 'Eat tacos', done: false },

{ id: 2, title: 'Brew tea', done: false },

];

export default function TaskApp() {

const [todos, updateTodos] = useImmer(

initialTodos

);

function handleAddTodo(title) {

updateTodos(draft => {

draft.push({

```
id: nextId++,
title: title,
done: false
});
});
}

function handleChangeTodo(nextTodo) {
updateTodos(draft => {
const todo = draft.find(t =>
t.id === nextTodo.id
);
todo.title = nextTodo.title;
todo.done = nextTodo.done;
});
}

function handleDeleteTodo(todoId) {
updateTodos(draft => {
const index = draft.findIndex(t =>
t.id === todoId
);
draft.splice(index, 1);
});
}

return (
<>
<AddTodo
onAddTodo={handleAddTodo}
/>
<TaskList
todos={todos}
onChangeTodo={handleChangeTodo}
onDeleteTodo={handleDeleteTodo}
/>
</>
```

```
);
}
```

```js AddTodo.js
import { useState } from 'react';

export default function AddTodo({ onAddTodo }) {
const [title, setTitle] = useState('');
return (
<>
<input
placeholder="Add todo"
value={title}
onChange={e => setTitle(e.target.value)}
/>
<button onClick={() => {
setTitle('');
onAddTodo(title);
}}>Add</button>
</>
)
}
```

```js TaskList.js
import { useState } from 'react';

export default function TaskList({
todos,
onChangeTodo,
onDeleteTodo
}) {
return (
<ul>
{todos.map(todo => (
<li key={todo.id}>
<Task
```

```
todo={todo}
onChange={onChangeTodo}
onDelete={onDeleteTodo}
/>
</li>
))}
</ul>
);
}

function Task({ todo, onChange, onDelete }) {
const [isEditing, setIsEditing] = useState(false);
let todoContent;
if (isEditing) {
todoContent = (
<>
<input
value={todo.title}
onChange={e => {
onChange({
...todo,
title: e.target.value
});
}} />
<button onClick={() => setIsEditing(false)}>
Save
</button>
</>
);
} else {
todoContent = (
<>
{todo.title}
<button onClick={() => setIsEditing(true)}>
Edit
</button>
```

```
</>
);
}
return (
<label>
<input
type="checkbox"
checked={todo.done}
onChange={e => {
onChange({
...todo,
done: e.target.checked
});
}}
/>
{todoContent}
<button onClick={() => onDelete(todo.id)}>
Delete
</button>
</label>
);
}
```

```css
button { margin: 5px; }
li { list-style-type: none; }
ul, li { margin: 0; padding: 0; }
```

```json package.json
{
"dependencies": {
"immer": "1.7.3",
"react": "latest",
"react-dom": "latest",
"react-scripts": "latest",
```

"use-immer": "0.5.1"

},

"scripts": {

"start": "react-scripts start",

"build": "react-scripts build",

"test": "react-scripts test --env=jsdom",

"eject": "react-scripts eject"

}

}

```

</Sandpack>

You can also mix and match the mutative and non-mutative approaches with Immer.

For example, in this version `handleAddTodo` is implemented by mutating the Immer `draft`, while `handleChangeTodo` and `handleDeleteTodo` use the non-mutative `map` and `filter` methods:

<Sandpack>

```js App.js

import { useState } from 'react';

import { useImmer } from 'use-immer';

import AddTodo from './AddTodo.js';

import TaskList from './TaskList.js';

let nextId = 3;

const initialTodos = [

{ id: 0, title: 'Buy milk', done: true },

{ id: 1, title: 'Eat tacos', done: false },

{ id: 2, title: 'Brew tea', done: false },

];

export default function TaskApp() {

const [todos, updateTodos] = useImmer(

initialTodos

);

function handleAddTodo(title) {

updateTodos(draft => {

draft.push({

```
id: nextId++,
title: title,
done: false
});
});
}

function handleChangeTodo(nextTodo) {
updateTodos(todos.map(todo => {
if (todo.id === nextTodo.id) {
return nextTodo;
} else {
return todo;
}
}));
}

function handleDeleteTodo(todoId) {
updateTodos(
todos.filter(t => t.id !== todoId)
);
}

return (
<>
<AddTodo
onAddTodo={handleAddTodo}
/>
<TaskList
todos={todos}
onChangeTodo={handleChangeTodo}
onDeleteTodo={handleDeleteTodo}
/>
</>
);
}
```

```js AddTodo.js
import { useState } from 'react';

export default function AddTodo({ onAddTodo }) {
const [title, setTitle] = useState('');
return (
<>
<input
placeholder="Add todo"
value={title}
onChange={e => setTitle(e.target.value)}
/>
<button onClick={() => {
setTitle('');
onAddTodo(title);
}}>Add</button>
</>
)
}
```

```js TaskList.js
import { useState } from 'react';

export default function TaskList({
todos,
onChangeTodo,
onDeleteTodo
}) {
return (
<ul>
{todos.map(todo => (
<li key={todo.id}>
<Task
todo={todo}
onChange={onChangeTodo}
onDelete={onDeleteTodo}
```

```
/>
</li>
))}
</ul>
);
}

function Task({ todo, onChange, onDelete }) {
const [isEditing, setIsEditing] = useState(false);
let todoContent;
if (isEditing) {
todoContent = (
<>
<input
value={todo.title}
onChange={e => {
onChange({
...todo,
title: e.target.value
});
}} />
<button onClick={() => setIsEditing(false)}>
Save
</button>
</>
);
} else {
todoContent = (
<>
{todo.title}
<button onClick={() => setIsEditing(true)}>
Edit
</button>
</>
);
}
```

```
return (
<label>
<input
type="checkbox"
checked={todo.done}
onChange={e => {
onChange({
...todo,
done: e.target.checked
});
}}
/>
{todoContent}
<button onClick={() => onDelete(todo.id)}>
Delete
</button>
</label>
);
}
```

```css
button { margin: 5px; }
li { list-style-type: none; }
ul, li { margin: 0; padding: 0; }
```

```json package.json
{
"dependencies": {
"immer": "1.7.3",
"react": "latest",
"react-dom": "latest",
"react-scripts": "latest",
"use-immer": "0.5.1"
},
"scripts": {
```

"start": "react-scripts start",

"build": "react-scripts build",

"test": "react-scripts test --env=jsdom",

"eject": "react-scripts eject"

}

}

```

</Sandpack>

With Immer, you can pick the style that feels the most natural for each separate case.

</Solution>

</Challenges>

---

title: Queueing a Series of State Updates

---

<Intro>

Setting a state variable will queue another render. But sometimes you might want to perform multiple operations on the value before queueing the next render. To do this, it helps to understand how React batches state updates.

</Intro>

<YouWillLearn>

* What "batching" is and how React uses it to process multiple state updates

* How to apply several updates to the same state variable in a row

</YouWillLearn>

## React batches state updates {/*react-batches-state-updates*/}

You might expect that clicking the "+3" button will increment the counter three times because it calls `setNumber(number + 1)` three times:

<Sandpack>

```js

import { useState } from 'react';

export default function Counter() {

const [number, setNumber] = useState(0);

```
return (
<>
<h1>{number}</h1>
<button onClick={() => {
setNumber(number + 1);
setNumber(number + 1);
setNumber(number + 1);
}}>+3</button>
</>
)
}
```

```css
button { display: inline-block; margin: 10px; font-size: 20px; }
h1 { display: inline-block; margin: 10px; width: 30px; text-align: center; }
```

</Sandpack>

However, as you might recall from the previous section, [each render's state values are fixed](/learn/state-as-a-snapshot#rendering-takes-a-snapshot-in-time), so the value of `number` inside the first render's event handler is always `0`, no matter how many times you call `setNumber(1)`:

```js
setNumber(0 + 1);
setNumber(0 + 1);
setNumber(0 + 1);
```

But there is one other factor at play here. **React waits until *all* code in the event handlers has run before processing your state updates.** This is why the re-render only happens *after* all these `setNumber()` calls.

This might remind you of a waiter taking an order at the restaurant. A waiter doesn't run to the kitchen at the mention of your first dish! Instead, they let you finish your order, let you make changes to it, and even take orders from other people at the table.

<Illustration src="/images/docs/illustrations/i_react-batching.png" alt="An elegant cursor at a restaurant places and order multiple times with React, playing the part of the waiter. After she calls setState() multiple times, the waiter writes down the last one she requested as her final order." />

This lets you update multiple state variables--even from multiple components--without triggering too many [re-renders.](/learn/render-and-commit#re-renders-when-state-updates) But this also means that

the UI won't be updated until _after_ your event handler, and any code in it, completes. This behavior, also known as **batching,** makes your React app run much faster. It also avoids dealing with confusing "half-finished" renders where only some of the variables have been updated.

React does not batch across *multiple* intentional events like clicks--each click is handled separately. Rest assured that React only does batching when it's generally safe to do. This ensures that, for example, if the first button click disables a form, the second click would not submit it again.

Updating the same state multiple times before the next render
{/*updating-the-same-state-multiple-times-before-the-next-render*/}

It is an uncommon use case, but if you would like to update the same state variable multiple times before the next render, instead of passing the *next state value* like `setNumber(number + 1)`, you can pass a *function* that calculates the next state based on the previous one in the queue, like `setNumber(n => n + 1)`. It is a way to tell React to "do something with the state value" instead of just replacing it.

Try incrementing the counter now:

<Sandpack>

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

export default function Counter() {
const [number, setNumber] = useState(0);

return (
<>
<h1>{number}</h1>
<button onClick={() => {
setNumber(n => n + 1);
setNumber(n => n + 1);
setNumber(n => n + 1);
}}>+3</button>
</>
)
}
```

```css
button { display: inline-block; margin: 10px; font-size: 20px; }
h1 { display: inline-block; margin: 10px; width: 30px; text-align: center; }
```

</Sandpack>

Here, `n => n + 1` is called an **updater function.** When you pass it to a state setter:

1. React queues this function to be processed after all the other code in the event handler has run.
2. During the next render, React goes through the queue and gives you the final updated state.

```js
setNumber(n => n + 1);
setNumber(n => n + 1);
setNumber(n => n + 1);
```

Here's how React works through these lines of code while executing the event handler:

1. `setNumber(n => n + 1)`: `n => n + 1` is a function. React adds it to a queue.
1. `setNumber(n => n + 1)`: `n => n + 1` is a function. React adds it to a queue.
1. `setNumber(n => n + 1)`: `n => n + 1` is a function. React adds it to a queue.

When you call `useState` during the next render, React goes through the queue. The previous `number` state was `0`, so that's what React passes to the first updater function as the `n` argument. Then React takes the return value of your previous updater function and passes it to the next updater as `n`, and so on:

queued update	`n`	returns
`n => n + 1`	`0`	`0 + 1 = 1`
`n => n + 1`	`1`	`1 + 1 = 2`
`n => n + 1`	`2`	`2 + 1 = 3`

React stores `3` as the final result and returns it from `useState`.

This is why clicking "+3" in the above example correctly increments the value by 3.

What happens if you update state after replacing it
{/*what-happens-if-you-update-state-after-replacing-it*/}

What about this event handler? What do you think `number` will be in the next render?

```js
<button onClick={() => {
setNumber(number + 5);
setNumber(n => n + 1);
}}>
```

<Sandpack>

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

export default function Counter() {
const [number, setNumber] = useState(0);

return (
<>
<h1>{number}</h1>
<button onClick={() => {
setNumber(number + 5);
setNumber(n => n + 1);
}}>Increase the number</button>
</>
)
}
```

```css
button { display: inline-block; margin: 10px; font-size: 20px; }
h1 { display: inline-block; margin: 10px; width: 30px; text-align: center; }
```

</Sandpack>

Here's what this event handler tells React to do:

1. `setNumber(number + 5)`: `number` is `0`, so `setNumber(0 + 5)`. React adds *"replace with `5`"* to its queue.
2. `setNumber(n => n + 1)`: `n => n + 1` is an updater function. React adds *that function* to its queue.

During the next render, React goes through the state queue:

queued update	`n`	returns
"replace with `5`"	`0` (unused)	`5`
`n => n + 1`	`5`	`5 + 1 = 6`

React stores `6` as the final result and returns it from `useState`.

<Note>

You may have noticed that `setState(5)` actually works like `setState(n => 5)`, but `n` is unused!

</Note>

What happens if you replace state after updating it
{/*what-happens-if-you-replace-state-after-updating-it*/}

Let's try one more example. What do you think `number` will be in the next render?

```js
<button onClick={() => {
setNumber(number + 5);
setNumber(n => n + 1);
setNumber(42);
}}>
```

<Sandpack>

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

export default function Counter() {
const [number, setNumber] = useState(0);

return (
<>
<h1>{number}</h1>
<button onClick={() => {
setNumber(number + 5);
setNumber(n => n + 1);
setNumber(42);
}}>Increase the number</button>
</>
)
}
```

```css
button { display: inline-block; margin: 10px; font-size: 20px; }
h1 { display: inline-block; margin: 10px; width: 30px; text-align: center; }
```

</Sandpack>

Here's how React works through these lines of code while executing this event handler:

1. `setNumber(number + 5)`: `number` is `0`, so `setNumber(0 + 5)`. React adds *"replace with `5`"* to its queue.

2. `setNumber(n => n + 1)`: `n => n + 1` is an updater function. React adds *that function* to its queue.

3. `setNumber(42)`: React adds *"replace with `42`"* to its queue.

During the next render, React goes through the state queue:

| queued update | `n` | returns |

|---------------|---------|-----|

| "replace with `5`" | `0` (unused) | `5` |

| `n => n + 1` | `5` | `5 + 1 = 6` |

| "replace with `42`" | `6` (unused) | `42` |

Then React stores `42` as the final result and returns it from `useState`.

To summarize, here's how you can think of what you're passing to the `setNumber` state setter:

* **An updater function** (e.g. `n => n + 1`) gets added to the queue.

* **Any other value** (e.g. number `5`) adds "replace with `5`" to the queue, ignoring what's already queued.

After the event handler completes, React will trigger a re-render. During the re-render, React will process the queue. Updater functions run during rendering, so **updater functions must be [pure](/learn/keeping-components-pure)** and only *return* the result. Don't try to set state from inside of them or run other side effects. In Strict Mode, React will run each updater function twice (but discard the second result) to help you find mistakes.

Naming conventions {/*naming-conventions*/}

It's common to name the updater function argument by the first letters of the corresponding state variable:

```js
setEnabled(e => !e);
setLastName(ln => ln.reverse());
setFriendCount(fc => fc * 2);
```

If you prefer more verbose code, another common convention is to repeat the full state variable name, like `setEnabled(enabled => !enabled)`, or to use a prefix like `setEnabled(prevEnabled => !prevEnabled)`.

<Recap>

* Setting state does not change the variable in the existing render, but it requests a new render.

* React processes state updates after event handlers have finished running. This is called batching.

* To update some state multiple times in one event, you can use `setNumber(n => n + 1)` updater function.

</Recap>

<Challenges>

Fix a request counter {/*fix-a-request-counter*/}

You're working on an art marketplace app that lets the user submit multiple orders for an art item at the same time. Each time the user presses the "Buy" button, the "Pending" counter should increase by one. After three seconds, the "Pending" counter should decrease, and the "Completed" counter should increase.

However, the "Pending" counter does not behave as intended. When you press "Buy", it decreases to `-1` (which should not be possible!). And if you click fast twice, both counters seem to behave unpredictably.

Why does this happen? Fix both counters.

<Sandpack>

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

export default function RequestTracker() {
const [pending, setPending] = useState(0);
const [completed, setCompleted] = useState(0);

async function handleClick() {
setPending(pending + 1);
await delay(3000);
setPending(pending - 1);
setCompleted(completed + 1);
}

return (
<>
<h3>
Pending: {pending}
</h3>
<h3>
```

Completed: {completed}

</h3>

<button onClick={handleClick}>

Buy

</button>

</>

);

}

function delay(ms) {

return new Promise(resolve => {

setTimeout(resolve, ms);

});

}

```

</Sandpack>

<Solution>

Inside the `handleClick` event handler, the values of `pending` and `completed` correspond to what they were at the time of the click event. For the first render, `pending` was `0`, so `setPending(pending - 1)` becomes `setPending(-1)`, which is wrong. Since you want to *increment* or *decrement* the counters, rather than set them to a concrete value determined during the click, you can instead pass the updater functions:

<Sandpack>

```js

import { useState } from 'react';

export default function RequestTracker() {

const [pending, setPending] = useState(0);

const [completed, setCompleted] = useState(0);

async function handleClick() {

setPending(p => p + 1);

await delay(3000);

setPending(p => p - 1);

setCompleted(c => c + 1);

}

return (

```
<>
<h3>
Pending: {pending}
</h3>
<h3>
Completed: {completed}
</h3>
<button onClick={handleClick}>
Buy
</button>
</>
);
}

function delay(ms) {
return new Promise(resolve => {
setTimeout(resolve, ms);
});
}
```

</Sandpack>

This ensures that when you increment or decrement a counter, you do it in relation to its *latest* state rather than what the state was at the time of the click.

</Solution>

Implement the state queue yourself {/*implement-the-state-queue-yourself*/}

In this challenge, you will reimplement a tiny part of React from scratch! It's not as hard as it sounds.

Scroll through the sandbox preview. Notice that it shows **four test cases.** They correspond to the examples you've seen earlier on this page. Your task is to implement the `getFinalState` function so that it returns the correct result for each of those cases. If you implement it correctly, all four tests should pass.

You will receive two arguments: `baseState` is the initial state (like `0`), and the `queue` is an array which contains a mix of numbers (like `5`) and updater functions (like `n => n + 1`) in the order they were added.

Your task is to return the final state, just like the tables on this page show!

<Hint>

If you're feeling stuck, start with this code structure:

```js
export function getFinalState(baseState, queue) {
let finalState = baseState;

for (let update of queue) {
if (typeof update === 'function') {
// TODO: apply the updater function
} else {
// TODO: replace the state
}
}

return finalState;
}
```

Fill out the missing lines!

</Hint>

<Sandpack>

```js processQueue.js active
export function getFinalState(baseState, queue) {
let finalState = baseState;

// TODO: do something with the queue...

return finalState;
}
```

```js App.js
import { getFinalState } from './processQueue.js';

function increment(n) {
return n + 1;
}
increment.toString = () => 'n => n+1';

export default function App() {
```

```
return (
<>
<TestCase
baseState={0}
queue={[1, 1, 1]}
expected={1}
/>
<hr />
<TestCase
baseState={0}
queue={[
increment,
increment,
increment
]}
expected={3}
/>
<hr />
<TestCase
baseState={0}
queue={[
5,
increment,
]}
expected={6}
/>
<hr />
<TestCase
baseState={0}
queue={[
5,
increment,
42,
]}
expected={42}
```

```
/>
</>
);
}

function TestCase({
baseState,
queue,
expected
}) {
const actual = getFinalState(baseState, queue);
return (
<>
<p>Base state: <b>{baseState}</b></p>
<p>Queue: <b>[{queue.join(', ')}]</b></p>
<p>Expected result: <b>{expected}</b></p>
<p style={{
color: actual === expected ?
'green' :
'red'
}}>
Your result: <b>{actual}</b>
{' '}
({actual === expected ?
'correct' :
'wrong'
})
</p>
</>
);
}
```

</Sandpack>

<Solution>

This is the exact algorithm described on this page that React uses to calculate the final state:

<Sandpack>

```js processQueue.js active
export function getFinalState(baseState, queue) {
let finalState = baseState;

for (let update of queue) {
if (typeof update === 'function') {
// Apply the updater function.
finalState = update(finalState);
} else {
// Replace the next state.
finalState = update;
}
}

return finalState;
}
```

```js App.js
import { getFinalState } from './processQueue.js';

function increment(n) {
return n + 1;
}
increment.toString = () => 'n => n+1';

export default function App() {
return (
<>
<TestCase
baseState={0}
queue={[1, 1, 1]}
expected={1}
/>
<hr />
<TestCase
baseState={0}
```

```
queue={[
increment,
increment,
increment
]}
expected={3}
/>
<hr />
<TestCase
baseState={0}
queue={[
5,
increment,
]}
expected={6}
/>
<hr />
<TestCase
baseState={0}
queue={[
5,
increment,
42,
]}
expected={42}
/>
</>
);
}

function TestCase({
baseState,
queue,
expected
}) {
const actual = getFinalState(baseState, queue);
```

```
return (
<>
<p>Base state: <b>{baseState}</b></p>
<p>Queue: <b>[{queue.join(', ')}]</b></p>
<p>Expected result: <b>{expected}</b></p>
<p style={{
color: actual === expected ?
'green' :
'red'
}}>
Your result: <b>{actual}</b>
{' '}
({actual === expected ?
'correct' :
'wrong'
})
</p>
</>
);
}
```

</Sandpack>

Now you know how this part of React works!

</Solution>

</Challenges>

---
title: Choosing the State Structure

<Intro>

Structuring state well can make a difference between a component that is pleasant to modify and debug, and one that is a constant source of bugs. Here are some tips you should consider when structuring state.

</Intro>

<YouWillLearn>

* When to use a single vs multiple state variables

* What to avoid when organizing state

* How to fix common issues with the state structure

</YouWillLearn>

Principles for structuring state {/*principles-for-structuring-state*/}

When you write a component that holds some state, you'll have to make choices about how many state variables to use and what the shape of their data should be. While it's possible to write correct programs even with a suboptimal state structure, there are a few principles that can guide you to make better choices:

1. **Group related state.** If you always update two or more state variables at the same time, consider merging them into a single state variable.

2. **Avoid contradictions in state.** When the state is structured in a way that several pieces of state may contradict and "disagree" with each other, you leave room for mistakes. Try to avoid this.

3. **Avoid redundant state.** If you can calculate some information from the component's props or its existing state variables during rendering, you should not put that information into that component's state.

4. **Avoid duplication in state.** When the same data is duplicated between multiple state variables, or within nested objects, it is difficult to keep them in sync. Reduce duplication when you can.

5. **Avoid deeply nested state.** Deeply hierarchical state is not very convenient to update. When possible, prefer to structure state in a flat way.

The goal behind these principles is to *make state easy to update without introducing mistakes*. Removing redundant and duplicate data from state helps ensure that all its pieces stay in sync. This is similar to how a database engineer might want to ["normalize" the database structure](https://docs.microsoft.com/en-us/office/troubleshoot/access/database-normalization-description) to reduce the chance of bugs. To paraphrase Albert Einstein, **"Make your state as simple as it can be--but no simpler."**

Now let's see how these principles apply in action.

Group related state {/*group-related-state*/}

You might sometimes be unsure between using a single or multiple state variables.

Should you do this?

```js
const [x, setX] = useState(0);
const [y, setY] = useState(0);
```

Or this?

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

Technically, you can use either of these approaches. But **if some two state variables always change together, it might be a good idea to unify them into a single state variable.** Then you won't forget to always keep them in sync, like in this example where moving the cursor updates both coordinates of the red dot:

<Sandpack>

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

export default function MovingDot() {
const [position, setPosition] = useState({
x: 0,
y: 0
});
return (
<div
onPointerMove={e => {
setPosition({
x: e.clientX,
y: e.clientY
});
}}
style={{
position: 'relative',
width: '100vw',
height: '100vh',
}}>
<div style={{
position: 'absolute',
backgroundColor: 'red',
borderRadius: '50%',
transform: `translate(${position.x}px, ${position.y}px)`,
left: -10,
top: -10,
```

width: 20,

height: 20,

}} />

)

}

```

```css

body { margin: 0; padding: 0; height: 250px; }

```

</Sandpack>

Another case where you'll group data into an object or an array is when you don't know how many pieces of state you'll need. For example, it's helpful when you have a form where the user can add custom fields.

<Pitfall>

If your state variable is an object, remember that [you can't update only one field in it](/learn/updating-objects-in-state) without explicitly copying the other fields. For example, you can't do `setPosition({ x: 100 })` in the above example because it would not have the `y` property at all! Instead, if you wanted to set `x` alone, you would either do `setPosition({ ...position, x: 100 })`, or split them into two state variables and do `setX(100)`.

</Pitfall>

Avoid contradictions in state {/*avoid-contradictions-in-state*/}

Here is a hotel feedback form with `isSending` and `isSent` state variables:

<Sandpack>

```js

import { useState } from 'react';

export default function FeedbackForm() {

const [text, setText] = useState('');

const [isSending, setIsSending] = useState(false);

const [isSent, setIsSent] = useState(false);

async function handleSubmit(e) {

e.preventDefault();

setIsSending(true);

```
await sendMessage(text);
setIsSending(false);
setIsSent(true);
}

if (isSent) {
return <h1>Thanks for feedback!</h1>
}

return (
<form onSubmit={handleSubmit}>
<p>How was your stay at The Prancing Pony?</p>
<textarea
disabled={isSending}
value={text}
onChange={e => setText(e.target.value)}
/>

<button
disabled={isSending}
type="submit"
>
Send
</button>
{isSending && <p>Sending...</p>}
</form>
);
}

// Pretend to send a message.
function sendMessage(text) {
return new Promise(resolve => {
setTimeout(resolve, 2000);
});
}
```

</Sandpack>

While this code works, it leaves the door open for "impossible" states. For example, if you forget to call `setIsSent` and `setIsSending` together, you may end up in a situation where both `isSending` and `isSent` are `true` at the same time. The more complex your component is, the harder it is to understand what happened.

Since `isSending` and `isSent` should never be `true` at the same time, it is better to replace them with one `status` state variable that may take one of *three* valid states: `'typing'` (initial), `'sending'`, and `'sent'`:

<Sandpack>

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

export default function FeedbackForm() {
const [text, setText] = useState('');
const [status, setStatus] = useState('typing');

async function handleSubmit(e) {
e.preventDefault();
setStatus('sending');
await sendMessage(text);
setStatus('sent');
}

const isSending = status === 'sending';
const isSent = status === 'sent';

if (isSent) {
return <h1>Thanks for feedback!</h1>
}

return (
<form onSubmit={handleSubmit}>
<p>How was your stay at The Prancing Pony?</p>
<textarea
disabled={isSending}
value={text}
onChange={e => setText(e.target.value)}
/>
<br />
<button
```

disabled={isSending}

type="submit"

>

Send

</button>

{isSending && Sending... }

</form>

);

}

// Pretend to send a message.

function sendMessage(text) {

return new Promise(resolve => {

setTimeout(resolve, 2000);

});

}

```

</Sandpack>

You can still declare some constants for readability:

```js

const isSending = status === 'sending';

const isSent = status === 'sent';

```

But they're not state variables, so you don't need to worry about them getting out of sync with each other.

Avoid redundant state {/*avoid-redundant-state*/}

If you can calculate some information from the component's props or its existing state variables during rendering, you **should not** put that information into that component's state.

For example, take this form. It works, but can you find any redundant state in it?

<Sandpack>

```js

import { useState } from 'react';

export default function Form() {

```
const [firstName, setFirstName] = useState('');
const [lastName, setLastName] = useState('');
const [fullName, setFullName] = useState('');

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

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

return (
<>
<h2>Let’s check you in</h2>
<label>
First name:{' '}
<input
value={firstName}
onChange={handleFirstNameChange}
/>
</label>
<label>
Last name:{' '}
<input
value={lastName}
onChange={handleLastNameChange}
/>
</label>
<p>
Your ticket will be issued to: {fullName}
</p>
</>
);
}
```

```

```css
label { display: block; margin-bottom: 5px; }
```

</Sandpack>

This form has three state variables: `firstName`, `lastName`, and `fullName`. However, `fullName` is redundant. **You can always calculate `fullName` from `firstName` and `lastName` during render, so remove it from state.**

This is how you can do it:

<Sandpack>

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

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

const fullName = firstName + ' ' + lastName;

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

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

return (
<>
<h2>Let’s check you in</h2>
<label>
First name:{' '}
<input
value={firstName}
onChange={handleFirstNameChange}
/>
</label>

<label>

Last name:{' '}

<input

value={lastName}

onChange={handleLastNameChange}

/>

</label>

Your ticket will be issued to: {fullName}

</>

);

}

```

```css

label { display: block; margin-bottom: 5px; }

```

</Sandpack>

Here, `fullName` is *not* a state variable. Instead, it's calculated during render:

```js

const fullName = firstName + ' ' + lastName;

```

As a result, the change handlers don't need to do anything special to update it. When you call `setFirstName` or `setLastName`, you trigger a re-render, and then the next `fullName` will be calculated from the fresh data.

<DeepDive>

Don't mirror props in state {/*don-t-mirror-props-in-state*/}

A common example of redundant state is code like this:

```js

function Message({ messageColor }) {

const [color, setColor] = useState(messageColor);

```
```

Here, a `color` state variable is initialized to the `messageColor` prop. The problem is that **if the parent component passes a different value of `messageColor` later (for example, `'red'` instead of `'blue'`), the `color` *state variable* would not be updated!** The state is only initialized during the first render.

This is why "mirroring" some prop in a state variable can lead to confusion. Instead, use the `messageColor` prop directly in your code. If you want to give it a shorter name, use a constant:

```js
function Message({ messageColor }) {
const color = messageColor;
```

This way it won't get out of sync with the prop passed from the parent component.

"Mirroring" props into state only makes sense when you *want* to ignore all updates for a specific prop. By convention, start the prop name with `initial` or `default` to clarify that its new values are ignored:

```js
function Message({ initialColor }) {
// The `color` state variable holds the *first* value of `initialColor`.
// Further changes to the `initialColor` prop are ignored.
const [color, setColor] = useState(initialColor);
```

</DeepDive>

Avoid duplication in state {/*avoid-duplication-in-state*/}

This menu list component lets you choose a single travel snack out of several:

<Sandpack>

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

const initialItems = [
{ title: 'pretzels', id: 0 },
{ title: 'crispy seaweed', id: 1 },
{ title: 'granola bar', id: 2 },
];

export default function Menu() {
const [items, setItems] = useState(initialItems);
const [selectedItem, setSelectedItem] = useState(
```

items[0]
);

return (
<>
<h2>What's your travel snack?</h2>

{items.map(item => (
<li key={item.id}>
{item.title}
{' '}
<button onClick={() => {
setSelectedItem(item);
}}>Choose</button>

))}

You picked {selectedItem.title}.
</>
);
}
```

```css
button { margin-top: 10px; }
```

</Sandpack>

Currently, it stores the selected item as an object in the `selectedItem` state variable. However, this is not great: **the contents of the `selectedItem` is the same object as one of the items inside the `items` list.** This means that the information about the item itself is duplicated in two places.

Why is this a problem? Let's make each item editable:

<Sandpack>

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

const initialItems = [

```
{ title: 'pretzels', id: 0 },
{ title: 'crispy seaweed', id: 1 },
{ title: 'granola bar', id: 2 },
];

export default function Menu() {
const [items, setItems] = useState(initialItems);
const [selectedItem, setSelectedItem] = useState(
items[0]
);

function handleItemChange(id, e) {
setItems(items.map(item => {
if (item.id === id) {
return {
...item,
title: e.target.value,
};
} else {
return item;
}
}));
}

return (
<>
<h2>What's your travel snack?</h2>
<ul>
{items.map((item, index) => (
<li key={item.id}>
<input
value={item.title}
onChange={e => {
handleItemChange(item.id, e)
}}
/>
{' '}
```

<button onClick={() => {

setSelectedItem(item);

}}>Choose</button>

))}

You picked {selectedItem.title}.

</>

);

}

```

```css

button { margin-top: 10px; }

```

</Sandpack>

Notice how if you first click "Choose" on an item and *then* edit it, **the input updates but the label at the bottom does not reflect the edits.** This is because you have duplicated state, and you forgot to update `selectedItem`.

Although you could update `selectedItem` too, an easier fix is to remove duplication. In this example, instead of a `selectedItem` object (which creates a duplication with objects inside `items`), you hold the `selectedId` in state, and *then* get the `selectedItem` by searching the `items` array for an item with that ID:

<Sandpack>

```js

import { useState } from 'react';

const initialItems = [

{ title: 'pretzels', id: 0 },

{ title: 'crispy seaweed', id: 1 },

{ title: 'granola bar', id: 2 },

];

export default function Menu() {

const [items, setItems] = useState(initialItems);

const [selectedId, setSelectedId] = useState(0);

const selectedItem = items.find(item =>

```
item.id === selectedId
);

function handleItemChange(id, e) {
setItems(items.map(item => {
if (item.id === id) {
return {
...item,
title: e.target.value,
};
} else {
return item;
}
}));
}

return (
<>
<h2>What's your travel snack?</h2>

{items.map((item, index) => (
<li key={item.id}>
<input
value={item.title}
onChange={e => {
handleItemChange(item.id, e)
}}
/>
{' '}
<button onClick={() => {
setSelectedId(item.id);
}}>Choose</button>

))}

<p>You picked {selectedItem.title}.</p>
</>
```

);

}

```

```css

button { margin-top: 10px; }

```

</Sandpack>

(Alternatively, you may hold the selected index in state.)

The state used to be duplicated like this:

* `items = [{ id: 0, title: 'pretzels'}, ...]`

* `selectedItem = {id: 0, title: 'pretzels'}`

But after the change it's like this:

* `items = [{ id: 0, title: 'pretzels'}, ...]`

* `selectedId = 0`

The duplication is gone, and you only keep the essential state!

Now if you edit the *selected* item, the message below will update immediately. This is because `setItems` triggers a re-render, and `items.find(...)` would find the item with the updated title. You didn't need to hold *the selected item* in state, because only the *selected ID* is essential. The rest could be calculated during render.

Avoid deeply nested state {/*avoid-deeply-nested-state*/}

Imagine a travel plan consisting of planets, continents, and countries. You might be tempted to structure its state using nested objects and arrays, like in this example:

<Sandpack>

```js

import { useState } from 'react';

import { initialTravelPlan } from './places.js';

function PlaceTree({ place }) {

const childPlaces = place.childPlaces;

return (

<li>

{place.title}

{childPlaces.length > 0 && (

```

{childPlaces.map(place => (
<PlaceTree key={place.id} place={place} />
))}

)}

);
}

export default function TravelPlan() {
const [plan, setPlan] = useState(initialTravelPlan);
const planets = plan.childPlaces;
return (
<>
<h2>Places to visit</h2>

{planets.map(place => (
<PlaceTree key={place.id} place={place} />
))}

</>
);
}
```

```js places.js active
export const initialTravelPlan = {
id: 0,
title: '(Root)',
childPlaces: [{
id: 1,
title: 'Earth',
childPlaces: [{
id: 2,
title: 'Africa',
childPlaces: [{
```

id: 3,
title: 'Botswana',
childPlaces: []
}, {
id: 4,
title: 'Egypt',
childPlaces: []
}, {
id: 5,
title: 'Kenya',
childPlaces: []
}, {
id: 6,
title: 'Madagascar',
childPlaces: []
}, {
id: 7,
title: 'Morocco',
childPlaces: []
}, {
id: 8,
title: 'Nigeria',
childPlaces: []
}, {
id: 9,
title: 'South Africa',
childPlaces: []
}]
}, {
id: 10,
title: 'Americas',
childPlaces: [{
id: 11,
title: 'Argentina',
childPlaces: []

```
}, {
id: 12,
title: 'Brazil',
childPlaces: []
}, {
id: 13,
title: 'Barbados',
childPlaces: []
}, {
id: 14,
title: 'Canada',
childPlaces: []
}, {
id: 15,
title: 'Jamaica',
childPlaces: []
}, {
id: 16,
title: 'Mexico',
childPlaces: []
}, {
id: 17,
title: 'Trinidad and Tobago',
childPlaces: []
}, {
id: 18,
title: 'Venezuela',
childPlaces: []
}]
}, {
id: 19,
title: 'Asia',
childPlaces: [{
id: 20,
title: 'China',
```

```
childPlaces: []
}, {
id: 21,
title: 'Hong Kong',
childPlaces: []
}, {
id: 22,
title: 'India',
childPlaces: []
}, {
id: 23,
title: 'Singapore',
childPlaces: []
}, {
id: 24,
title: 'South Korea',
childPlaces: []
}, {
id: 25,
title: 'Thailand',
childPlaces: []
}, {
id: 26,
title: 'Vietnam',
childPlaces: []
}]
}, {
id: 27,
title: 'Europe',
childPlaces: [{
id: 28,
title: 'Croatia',
childPlaces: [],
}, {
id: 29,
```

```
title: 'France',
childPlaces: [],
}, {
id: 30,
title: 'Germany',
childPlaces: [],
}, {
id: 31,
title: 'Italy',
childPlaces: [],
}, {
id: 32,
title: 'Portugal',
childPlaces: [],
}, {
id: 33,
title: 'Spain',
childPlaces: [],
}, {
id: 34,
title: 'Turkey',
childPlaces: [],
}]
}, {
id: 35,
title: 'Oceania',
childPlaces: [{
id: 36,
title: 'Australia',
childPlaces: [],
}, {
id: 37,
title: 'Bora Bora (French Polynesia)',
childPlaces: [],
}, {
```

id: 38,
title: 'Easter Island (Chile)',
childPlaces: [],
}, {
id: 39,
title: 'Fiji',
childPlaces: [],
}, {
id: 40,
title: 'Hawaii (the USA)',
childPlaces: [],
}, {
id: 41,
title: 'New Zealand',
childPlaces: [],
}, {
id: 42,
title: 'Vanuatu',
childPlaces: [],
}]
}]
}, {
id: 43,
title: 'Moon',
childPlaces: [{
id: 44,
title: 'Rheita',
childPlaces: []
}, {
id: 45,
title: 'Piccolomini',
childPlaces: []
}, {
id: 46,
title: 'Tycho',

childPlaces: []

}]

}, {

id: 47,

title: 'Mars',

childPlaces: [{

id: 48,

title: 'Corn Town',

childPlaces: []

}, {

id: 49,

title: 'Green Hill',

childPlaces: []

}]

}]

};

```

</Sandpack>

Now let's say you want to add a button to delete a place you've already visited. How would you go about it? [Updating nested state](/learn/updating-objects-in-state#updating-a-nested-object) involves making copies of objects all the way up from the part that changed. Deleting a deeply nested place would involve copying its entire parent place chain. Such code can be very verbose.

**If the state is too nested to update easily, consider making it "flat".** Here is one way you can restructure this data. Instead of a tree-like structure where each `place` has an array of *its child places*, you can have each place hold an array of *its child place IDs*. Then store a mapping from each place ID to the corresponding place.

This data restructuring might remind you of seeing a database table:

<Sandpack>

```js

import { useState } from 'react';

import { initialTravelPlan } from './places.js';

function PlaceTree({ id, placesById }) {

const place = placesById[id];

const childIds = place.childIds;

return (

```

{place.title}
{childIds.length > 0 && (

{childIds.map(childId => (
<PlaceTree
key={childId}
id={childId}
placesById={placesById}
/>
))}

)}

);
}

export default function TravelPlan() {
const [plan, setPlan] = useState(initialTravelPlan);
const root = plan[0];
const planetIds = root.childIds;
return (
<>
<h2>Places to visit</h2>

{planetIds.map(id => (
<PlaceTree
key={id}
id={id}
placesById={plan}
/>
))}

</>
);
}
```

```

```js places.js active
export const initialTravelPlan = {
0: {
id: 0,
title: '(Root)',
childIds: [1, 43, 47],
},
1: {
id: 1,
title: 'Earth',
childIds: [2, 10, 19, 27, 35]
},
2: {
id: 2,
title: 'Africa',
childIds: [3, 4, 5, 6 , 7, 8, 9]
},
3: {
id: 3,
title: 'Botswana',
childIds: []
},
4: {
id: 4,
title: 'Egypt',
childIds: []
},
5: {
id: 5,
title: 'Kenya',
childIds: []
},
6: {
id: 6,

```
title: 'Madagascar',
childIds: []
},
7: {
id: 7,
title: 'Morocco',
childIds: []
},
8: {
id: 8,
title: 'Nigeria',
childIds: []
},
9: {
id: 9,
title: 'South Africa',
childIds: []
},
10: {
id: 10,
title: 'Americas',
childIds: [11, 12, 13, 14, 15, 16, 17, 18],
},
11: {
id: 11,
title: 'Argentina',
childIds: []
},
12: {
id: 12,
title: 'Brazil',
childIds: []
},
13: {
id: 13,
```

```
title: 'Barbados',
childIds: []
},
14: {
id: 14,
title: 'Canada',
childIds: []
},
15: {
id: 15,
title: 'Jamaica',
childIds: []
},
16: {
id: 16,
title: 'Mexico',
childIds: []
},
17: {
id: 17,
title: 'Trinidad and Tobago',
childIds: []
},
18: {
id: 18,
title: 'Venezuela',
childIds: []
},
19: {
id: 19,
title: 'Asia',
childIds: [20, 21, 22, 23, 24, 25, 26],
},
20: {
id: 20,
```

```
title: 'China',
childIds: []
},
21: {
id: 21,
title: 'Hong Kong',
childIds: []
},
22: {
id: 22,
title: 'India',
childIds: []
},
23: {
id: 23,
title: 'Singapore',
childIds: []
},
24: {
id: 24,
title: 'South Korea',
childIds: []
},
25: {
id: 25,
title: 'Thailand',
childIds: []
},
26: {
id: 26,
title: 'Vietnam',
childIds: []
},
27: {
id: 27,
```

```
title: 'Europe',
childIds: [28, 29, 30, 31, 32, 33, 34],
},
28: {
id: 28,
title: 'Croatia',
childIds: []
},
29: {
id: 29,
title: 'France',
childIds: []
},
30: {
id: 30,
title: 'Germany',
childIds: []
},
31: {
id: 31,
title: 'Italy',
childIds: []
},
32: {
id: 32,
title: 'Portugal',
childIds: []
},
33: {
id: 33,
title: 'Spain',
childIds: []
},
34: {
id: 34,
```

```
title: 'Turkey',
childIds: []
},
35: {
id: 35,
title: 'Oceania',
childIds: [36, 37, 38, 39, 40, 41, 42],
},
36: {
id: 36,
title: 'Australia',
childIds: []
},
37: {
id: 37,
title: 'Bora Bora (French Polynesia)',
childIds: []
},
38: {
id: 38,
title: 'Easter Island (Chile)',
childIds: []
},
39: {
id: 39,
title: 'Fiji',
childIds: []
},
40: {
id: 40,
title: 'Hawaii (the USA)',
childIds: []
},
41: {
id: 41,
```

```
title: 'New Zealand',
childIds: []
},
42: {
id: 42,
title: 'Vanuatu',
childIds: []
},
43: {
id: 43,
title: 'Moon',
childIds: [44, 45, 46]
},
44: {
id: 44,
title: 'Rheita',
childIds: []
},
45: {
id: 45,
title: 'Piccolomini',
childIds: []
},
46: {
id: 46,
title: 'Tycho',
childIds: []
},
47: {
id: 47,
title: 'Mars',
childIds: [48, 49]
},
48: {
id: 48,
```

```
title: 'Corn Town',
childIds: []
},
49: {
id: 49,
title: 'Green Hill',
childIds: []
}
};
```

</Sandpack>

Now that the state is "flat" (also known as "normalized"), updating nested items becomes easier.

In order to remove a place now, you only need to update two levels of state:

- The updated version of its *parent* place should exclude the removed ID from its `childIds` array.
- The updated version of the root "table" object should include the updated version of the parent place.

Here is an example of how you could go about it:

<Sandpack>

```js
import { useState } from 'react';
import { initialTravelPlan } from './places.js';

export default function TravelPlan() {
const [plan, setPlan] = useState(initialTravelPlan);

function handleComplete(parentId, childId) {
const parent = plan[parentId];
// Create a new version of the parent place
// that doesn't include this child ID.
const nextParent = {
...parent,
childIds: parent.childIds
.filter(id => id !== childId)
};
// Update the root state object...
```

```
setPlan({
...plan,
// ...so that it has the updated parent.
[parentId]: nextParent
});
}

const root = plan[0];
const planetIds = root.childIds;
return (
<>
<h2>Places to visit</h2>
<ol>
{planetIds.map(id => (
<PlaceTree
key={id}
id={id}
parentId={0}
placesById={plan}
onComplete={handleComplete}
/>
))}
</ol>
</>
);
}

function PlaceTree({ id, parentId, placesById, onComplete }) {
const place = placesById[id];
const childIds = place.childIds;
return (
<li>
{place.title}
<button onClick={() => {
onComplete(parentId, id);
}}>
Complete
```

```
</button>
{childIds.length > 0 &&
<ol>
{childIds.map(childId => (
<PlaceTree
key={childId}
id={childId}
parentId={id}
placesById={placesById}
onComplete={onComplete}
/>
))}
</ol>
}
</li>
);
}
```

```js places.js
export const initialTravelPlan = {
0: {
id: 0,
title: '(Root)',
childIds: [1, 43, 47],
},
1: {
id: 1,
title: 'Earth',
childIds: [2, 10, 19, 27, 35]
},
2: {
id: 2,
title: 'Africa',
childIds: [3, 4, 5, 6 , 7, 8, 9]
},
```

```
3: {
id: 3,
title: 'Botswana',
childIds: []
},
4: {
id: 4,
title: 'Egypt',
childIds: []
},
5: {
id: 5,
title: 'Kenya',
childIds: []
},
6: {
id: 6,
title: 'Madagascar',
childIds: []
},
7: {
id: 7,
title: 'Morocco',
childIds: []
},
8: {
id: 8,
title: 'Nigeria',
childIds: []
},
9: {
id: 9,
title: 'South Africa',
childIds: []
},
```

```
10: {
id: 10,
title: 'Americas',
childIds: [11, 12, 13, 14, 15, 16, 17, 18],
},
11: {
id: 11,
title: 'Argentina',
childIds: []
},
12: {
id: 12,
title: 'Brazil',
childIds: []
},
13: {
id: 13,
title: 'Barbados',
childIds: []
},
14: {
id: 14,
title: 'Canada',
childIds: []
},
15: {
id: 15,
title: 'Jamaica',
childIds: []
},
16: {
id: 16,
title: 'Mexico',
childIds: []
},
```

```
17: {
id: 17,
title: 'Trinidad and Tobago',
childIds: []
},
18: {
id: 18,
title: 'Venezuela',
childIds: []
},
19: {
id: 19,
title: 'Asia',
childIds: [20, 21, 22, 23, 24, 25, 26],
},
20: {
id: 20,
title: 'China',
childIds: []
},
21: {
id: 21,
title: 'Hong Kong',
childIds: []
},
22: {
id: 22,
title: 'India',
childIds: []
},
23: {
id: 23,
title: 'Singapore',
childIds: []
},
```

```
24: {
id: 24,
title: 'South Korea',
childIds: []
},
25: {
id: 25,
title: 'Thailand',
childIds: []
},
26: {
id: 26,
title: 'Vietnam',
childIds: []
},
27: {
id: 27,
title: 'Europe',
childIds: [28, 29, 30, 31, 32, 33, 34],
},
28: {
id: 28,
title: 'Croatia',
childIds: []
},
29: {
id: 29,
title: 'France',
childIds: []
},
30: {
id: 30,
title: 'Germany',
childIds: []
},
```

```
31: {
id: 31,
title: 'Italy',
childIds: []
},
32: {
id: 32,
title: 'Portugal',
childIds: []
},
33: {
id: 33,
title: 'Spain',
childIds: []
},
34: {
id: 34,
title: 'Turkey',
childIds: []
},
35: {
id: 35,
title: 'Oceania',
childIds: [36, 37, 38, 39, 40, 41,, 42],
},
36: {
id: 36,
title: 'Australia',
childIds: []
},
37: {
id: 37,
title: 'Bora Bora (French Polynesia)',
childIds: []
},
```

```
38: {
id: 38,
title: 'Easter Island (Chile)',
childIds: []
},
39: {
id: 39,
title: 'Fiji',
childIds: []
},
40: {
id: 40,
title: 'Hawaii (the USA)',
childIds: []
},
41: {
id: 41,
title: 'New Zealand',
childIds: []
},
42: {
id: 42,
title: 'Vanuatu',
childIds: []
},
43: {
id: 43,
title: 'Moon',
childIds: [44, 45, 46]
},
44: {
id: 44,
title: 'Rheita',
childIds: []
},
```

```
45: {
id: 45,
title: 'Piccolomini',
childIds: []
},
46: {
id: 46,
title: 'Tycho',
childIds: []
},
47: {
id: 47,
title: 'Mars',
childIds: [48, 49]
},
48: {
id: 48,
title: 'Corn Town',
childIds: []
},
49: {
id: 49,
title: 'Green Hill',
childIds: []
}
};
```

```css
button { margin: 10px; }
```

</Sandpack>

You can nest state as much as you like, but making it "flat" can solve numerous problems. It makes state easier to update, and it helps ensure you don't have duplication in different parts of a nested object.

<DeepDive>

Improving memory usage {/*improving-memory-usage*/}

Ideally, you would also remove the deleted items (and their children!) from the "table" object to improve memory usage. This version does that. It also [uses Immer](/learn/updating-objects-in-state#write-concise-update-logic-with-immer) to make the update logic more concise.

<Sandpack>

```js
import { useImmer } from 'use-immer';
import { initialTravelPlan } from './places.js';

export default function TravelPlan() {
const [plan, updatePlan] = useImmer(initialTravelPlan);

function handleComplete(parentId, childId) {
updatePlan(draft => {
// Remove from the parent place's child IDs.
const parent = draft[parentId];
parent.childIds = parent.childIds
.filter(id => id !== childId);

// Forget this place and all its subtree.
deleteAllChildren(childId);
function deleteAllChildren(id) {
const place = draft[id];
place.childIds.forEach(deleteAllChildren);
delete draft[id];
}
});
}

const root = plan[0];
const planetIds = root.childIds;
return (
<>
<h2>Places to visit</h2>
<ol>
{planetIds.map(id => (
<PlaceTree
```

```
key={id}
id={id}
parentId={0}
placesById={plan}
onComplete={handleComplete}
/>
))}
</ol>
</>
);
}

function PlaceTree({ id, parentId, placesById, onComplete }) {
const place = placesById[id];
const childIds = place.childIds;
return (
<li>
{place.title}
<button onClick={() => {
onComplete(parentId, id);
}}>
Complete
</button>
{childIds.length > 0 &&
<ol>
{childIds.map(childId => (
<PlaceTree
key={childId}
id={childId}
parentId={id}
placesById={placesById}
onComplete={onComplete}
/>
))}
</ol>
}
```


);
}
```

```js places.js
export const initialTravelPlan = {
0: {
id: 0,
title: '(Root)',
childIds: [1, 43, 47],
},
1: {
id: 1,
title: 'Earth',
childIds: [2, 10, 19, 27, 35]
},
2: {
id: 2,
title: 'Africa',
childIds: [3, 4, 5, 6 , 7, 8, 9]
},
3: {
id: 3,
title: 'Botswana',
childIds: []
},
4: {
id: 4,
title: 'Egypt',
childIds: []
},
5: {
id: 5,
title: 'Kenya',
childIds: []

```
},
6: {
id: 6,
title: 'Madagascar',
childIds: []
},
7: {
id: 7,
title: 'Morocco',
childIds: []
},
8: {
id: 8,
title: 'Nigeria',
childIds: []
},
9: {
id: 9,
title: 'South Africa',
childIds: []
},
10: {
id: 10,
title: 'Americas',
childIds: [11, 12, 13, 14, 15, 16, 17, 18],
},
11: {
id: 11,
title: 'Argentina',
childIds: []
},
12: {
id: 12,
title: 'Brazil',
childIds: []
```

},
13: {
id: 13,
title: 'Barbados',
childIds: []
},
14: {
id: 14,
title: 'Canada',
childIds: []
},
15: {
id: 15,
title: 'Jamaica',
childIds: []
},
16: {
id: 16,
title: 'Mexico',
childIds: []
},
17: {
id: 17,
title: 'Trinidad and Tobago',
childIds: []
},
18: {
id: 18,
title: 'Venezuela',
childIds: []
},
19: {
id: 19,
title: 'Asia',
childIds: [20, 21, 22, 23, 24, 25, 26],

},
20: {
id: 20,
title: 'China',
childIds: []
},
21: {
id: 21,
title: 'Hong Kong',
childIds: []
},
22: {
id: 22,
title: 'India',
childIds: []
},
23: {
id: 23,
title: 'Singapore',
childIds: []
},
24: {
id: 24,
title: 'South Korea',
childIds: []
},
25: {
id: 25,
title: 'Thailand',
childIds: []
},
26: {
id: 26,
title: 'Vietnam',
childIds: []

```
},
27: {
id: 27,
title: 'Europe',
childIds: [28, 29, 30, 31, 32, 33, 34],
},
28: {
id: 28,
title: 'Croatia',
childIds: []
},
29: {
id: 29,
title: 'France',
childIds: []
},
30: {
id: 30,
title: 'Germany',
childIds: []
},
31: {
id: 31,
title: 'Italy',
childIds: []
},
32: {
id: 32,
title: 'Portugal',
childIds: []
},
33: {
id: 33,
title: 'Spain',
childIds: []
```

},
34: {
id: 34,
title: 'Turkey',
childIds: []
},
35: {
id: 35,
title: 'Oceania',
childIds: [36, 37, 38, 39, 40, 41,, 42],
},
36: {
id: 36,
title: 'Australia',
childIds: []
},
37: {
id: 37,
title: 'Bora Bora (French Polynesia)',
childIds: []
},
38: {
id: 38,
title: 'Easter Island (Chile)',
childIds: []
},
39: {
id: 39,
title: 'Fiji',
childIds: []
},
40: {
id: 40,
title: 'Hawaii (the USA)',
childIds: []

},
41: {
id: 41,
title: 'New Zealand',
childIds: []
},
42: {
id: 42,
title: 'Vanuatu',
childIds: []
},
43: {
id: 43,
title: 'Moon',
childIds: [44, 45, 46]
},
44: {
id: 44,
title: 'Rheita',
childIds: []
},
45: {
id: 45,
title: 'Piccolomini',
childIds: []
},
46: {
id: 46,
title: 'Tycho',
childIds: []
},
47: {
id: 47,
title: 'Mars',
childIds: [48, 49]

```
},
48: {
id: 48,
title: 'Corn Town',
childIds: []
},
49: {
id: 49,
title: 'Green Hill',
childIds: []
}
};
```

```css
button { margin: 10px; }
```

```json package.json
{
"dependencies": {
"immer": "1.7.3",
"react": "latest",
"react-dom": "latest",
"react-scripts": "latest",
"use-immer": "0.5.1"
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
}
}
```

</Sandpack>

</DeepDive>

Sometimes, you can also reduce state nesting by moving some of the nested state into the child components. This works well for ephemeral UI state that doesn't need to be stored, like whether an item is hovered.

<Recap>

* If two state variables always update together, consider merging them into one.
* Choose your state variables carefully to avoid creating "impossible" states.
* Structure your state in a way that reduces the chances that you'll make a mistake updating it.
* Avoid redundant and duplicate state so that you don't need to keep it in sync.
* Don't put props *into* state unless you specifically want to prevent updates.
* For UI patterns like selection, keep ID or index in state instead of the object itself.
* If updating deeply nested state is complicated, try flattening it.

</Recap>

<Challenges>

Fix a component that's not updating {/*fix-a-component-thats-not-updating*/}

This `Clock` component receives two props: `color` and `time`. When you select a different color in the select box, the `Clock` component receives a different `color` prop from its parent component. However, for some reason, the displayed color doesn't update. Why? Fix the problem.

<Sandpack>

```js Clock.js active
import { useState } from 'react';

export default function Clock(props) {
const [color, setColor] = useState(props.color);
return (
<h1 style={{ color: color }}>
{props.time}
</h1>
);
}
```

```js App.js hidden
import { useState, useEffect } from 'react';
import Clock from './Clock.js';
```

```
function useTime() {
const [time, setTime] = useState(() => new Date());
useEffect(() => {
const id = setInterval(() => {
setTime(new Date());
}, 1000);
return () => clearInterval(id);
}, []);
return time;
}

export default function App() {
const time = useTime();
const [color, setColor] = useState('lightcoral');
return (
<div>
<p>
Pick a color:{' '}
<select value={color} onChange={e => setColor(e.target.value)}>
<option value="lightcoral">lightcoral</option>
<option value="midnightblue">midnightblue</option>
<option value="rebeccapurple">rebeccapurple</option>
</select>
</p>
<Clock color={color} time={time.toLocaleTimeString()} />
</div>
);
}
```

</Sandpack>

<Solution>

The issue is that this component has `color` state initialized with the initial value of the `color` prop. But when the `color` prop changes, this does not affect the state variable! So they get out of sync. To fix this issue, remove the state variable altogether, and use the `color` prop directly.

<Sandpack>

```js Clock.js active
import { useState } from 'react';

export default function Clock(props) {
return (
<h1 style={{ color: props.color }}>
{props.time}
</h1>
);
}
```

```js App.js hidden
import { useState, useEffect } from 'react';
import Clock from './Clock.js';

function useTime() {
const [time, setTime] = useState(() => new Date());
useEffect(() => {
const id = setInterval(() => {
setTime(new Date());
}, 1000);
return () => clearInterval(id);
}, []);
return time;
}

export default function App() {
const time = useTime();
const [color, setColor] = useState('lightcoral');
return (
<div>
<p>
Pick a color:{' '}
<select value={color} onChange={e => setColor(e.target.value)}>
<option value="lightcoral">lightcoral</option>
<option value="midnightblue">midnightblue</option>
<option value="rebeccapurple">rebeccapurple</option>
```

```
</select>
</p>
<Clock color={color} time={time.toLocaleTimeString()} />
</div>
);
}
```

</Sandpack>

Or, using the destructuring syntax:

<Sandpack>

```js Clock.js active
import { useState } from 'react';

export default function Clock({ color, time }) {
return (
<h1 style={{ color: color }}>
{time}
</h1>
);
}
```

```js App.js hidden
import { useState, useEffect } from 'react';
import Clock from './Clock.js';

function useTime() {
const [time, setTime] = useState(() => new Date());
useEffect(() => {
const id = setInterval(() => {
setTime(new Date());
}, 1000);
return () => clearInterval(id);
}, []);
return time;
}
```

```
export default function App() {
const time = useTime();
const [color, setColor] = useState('lightcoral');
return (
<div>
<p>
Pick a color:{' '}
<select value={color} onChange={e => setColor(e.target.value)}>
<option value="lightcoral">lightcoral</option>
<option value="midnightblue">midnightblue</option>
<option value="rebeccapurple">rebeccapurple</option>
</select>
</p>
<Clock color={color} time={time.toLocaleTimeString()} />
</div>
);
}
```

</Sandpack>

</Solution>

Fix a broken packing list {/*fix-a-broken-packing-list*/}

This packing list has a footer that shows how many items are packed, and how many items there are overall. It seems to work at first, but it is buggy. For example, if you mark an item as packed and then delete it, the counter will not be updated correctly. Fix the counter so that it's always correct.

<Hint>

Is any state in this example redundant?

</Hint>

<Sandpack>

```js App.js
import { useState } from 'react';
import AddItem from './AddItem.js';
import PackingList from './PackingList.js';

let nextId = 3;
```

```
const initialItems = [
{ id: 0, title: 'Warm socks', packed: true },
{ id: 1, title: 'Travel journal', packed: false },
{ id: 2, title: 'Watercolors', packed: false },
];

export default function TravelPlan() {
const [items, setItems] = useState(initialItems);
const [total, setTotal] = useState(3);
const [packed, setPacked] = useState(1);

function handleAddItem(title) {
setTotal(total + 1);
setItems([
...items,
{
id: nextId++,
title: title,
packed: false
}
]);
}

function handleChangeItem(nextItem) {
if (nextItem.packed) {
setPacked(packed + 1);
} else {
setPacked(packed - 1);
}
setItems(items.map(item => {
if (item.id === nextItem.id) {
return nextItem;
} else {
return item;
}
}));
}
```

```
function handleDeleteItem(itemId) {
setTotal(total - 1);
setItems(
items.filter(item => item.id !== itemId)
);
}

return (
<>
<AddItem
onAddItem={handleAddItem}
/>
<PackingList
items={items}
onChangeItem={handleChangeItem}
onDeleteItem={handleDeleteItem}
/>
<hr />
<b>{packed} out of {total} packed!</b>
</>
);
}
```

```js AddItem.js hidden
import { useState } from 'react';

export default function AddItem({ onAddItem }) {
const [title, setTitle] = useState('');
return (
<>
<input
placeholder="Add item"
value={title}
onChange={e => setTitle(e.target.value)}
/>
<button onClick={() => {
```

```
setTitle('');
onAddItem(title);
}}>Add</button>
</>
)
}
```

```js PackingList.js hidden
import { useState } from 'react';

export default function PackingList({
items,
onChangeItem,
onDeleteItem
}) {
return (
<ul>
{items.map(item => (
<li key={item.id}>
<label>
<input
type="checkbox"
checked={item.packed}
onChange={e => {
onChangeItem({
...item,
packed: e.target.checked
});
}}
/>
{' '}
{item.title}
</label>
<button onClick={() => onDeleteItem(item.id)}>
Delete
</button>
```


))}

);
}
```

```css
button { margin: 5px; }
li { list-style-type: none; }
ul, li { margin: 0; padding: 0; }
```

</Sandpack>

<Solution>

Although you could carefully change each event handler to update the `total` and `packed` counters correctly, the root problem is that these state variables exist at all. They are redundant because you can always calculate the number of items (packed or total) from the `items` array itself. Remove the redundant state to fix the bug:

<Sandpack>

```js App.js
import { useState } from 'react';
import AddItem from './AddItem.js';
import PackingList from './PackingList.js';

let nextId = 3;
const initialItems = [
{ id: 0, title: 'Warm socks', packed: true },
{ id: 1, title: 'Travel journal', packed: false },
{ id: 2, title: 'Watercolors', packed: false },
];

export default function TravelPlan() {
const [items, setItems] = useState(initialItems);

const total = items.length;
const packed = items
.filter(item => item.packed)

```
.length;

function handleAddItem(title) {
setItems([
...items,
{
id: nextId++,
title: title,
packed: false
}
]);
}

function handleChangeItem(nextItem) {
setItems(items.map(item => {
if (item.id === nextItem.id) {
return nextItem;
} else {
return item;
}
}));
}

function handleDeleteItem(itemId) {
setItems(
items.filter(item => item.id !== itemId)
);
}

return (
<>
<AddItem
onAddItem={handleAddItem}
/>
<PackingList
items={items}
onChangeItem={handleChangeItem}
onDeleteItem={handleDeleteItem}
```

```
/>
<hr />
<b>{packed} out of {total} packed!</b>
</>
);
}
```

```js AddItem.js hidden
import { useState } from 'react';

export default function AddItem({ onAddItem }) {
const [title, setTitle] = useState('');
return (
<>
<input
placeholder="Add item"
value={title}
onChange={e => setTitle(e.target.value)}
/>
<button onClick={() => {
setTitle('');
onAddItem(title);
}}>Add</button>
</>
)
}
```

```js PackingList.js hidden
import { useState } from 'react';

export default function PackingList({
items,
onChangeItem,
onDeleteItem
}) {
return (
```

```
<ul>
{items.map(item => (
<li key={item.id}>
<label>
<input
type="checkbox"
checked={item.packed}
onChange={e => {
onChangeItem({
...item,
packed: e.target.checked
});
}}
/>
{' '}
{item.title}
</label>
<button onClick={() => onDeleteItem(item.id)}>
Delete
</button>
</li>
))}
</ul>
);
}
```

```css
button { margin: 5px; }
li { list-style-type: none; }
ul, li { margin: 0; padding: 0; }
```

</Sandpack>

Notice how the event handlers are only concerned with calling `setItems` after this change. The item counts are now calculated during the next render from `items`, so they are always up-to-date.

</Solution>

Fix the disappearing selection {/*fix-the-disappearing-selection*/}

There is a list of `letters` in state. When you hover or focus a particular letter, it gets highlighted. The currently highlighted letter is stored in the `highlightedLetter` state variable. You can "star" and "unstar" individual letters, which updates the `letters` array in state.

This code works, but there is a minor UI glitch. When you press "Star" or "Unstar", the highlighting disappears for a moment. However, it reappears as soon as you move your pointer or switch to another letter with keyboard. Why is this happening? Fix it so that the highlighting doesn't disappear after the button click.

<Sandpack>

```js App.js
import { useState } from 'react';
import { initialLetters } from './data.js';
import Letter from './Letter.js';

export default function MailClient() {
const [letters, setLetters] = useState(initialLetters);
const [highlightedLetter, setHighlightedLetter] = useState(null);

function handleHover(letter) {
setHighlightedLetter(letter);
}

function handleStar(starred) {
setLetters(letters.map(letter => {
if (letter.id === starred.id) {
return {
...letter,
isStarred: !letter.isStarred
};
} else {
return letter;
}
}));
}

return (
<>
```

<h2>Inbox</h2>

{letters.map(letter => (
<Letter
key={letter.id}
letter={letter}
isHighlighted={
letter === highlightedLetter
}
onHover={handleHover}
onToggleStar={handleStar}
/>
))}

</>
);
}
```

```js Letter.js
export default function Letter({
letter,
isHighlighted,
onHover,
onToggleStar,
}) {
return (
<li
className={
isHighlighted ? 'highlighted' : ''
}
onFocus={() => {
onHover(letter);
}}
onPointerMove={() => {
onHover(letter);

}}
>
<button onClick={() => {
onToggleStar(letter);
}}>
{letter.isStarred ? 'Unstar' : 'Star'}
</button>
{letter.subject}
</li>
)
}
```

```js data.js
export const initialLetters = [{
id: 0,
subject: 'Ready for adventure?',
isStarred: true,
}, {
id: 1,
subject: 'Time to check in!',
isStarred: false,
}, {
id: 2,
subject: 'Festival Begins in Just SEVEN Days!',
isStarred: false,
}];
```

```css
button { margin: 5px; }
li { border-radius: 5px; }
.highlighted { background: #d2eaff; }
```

</Sandpack>

<Solution>
```

The problem is that you're holding the letter object in `highlightedLetter`. But you're also holding the same information in the `letters` array. So your state has duplication! When you update the `letters` array after the button click, you create a new letter object which is different from `highlightedLetter`. This is why `highlightedLetter === letter` check becomes `false`, and the highlight disappears. It reappears the next time you call `setHighlightedLetter` when the pointer moves.

To fix the issue, remove the duplication from state. Instead of storing *the letter itself* in two places, store the `highlightedId` instead. Then you can check `isHighlighted` for each letter with `letter.id === highlightedId`, which will work even if the `letter` object has changed since the last render.

<Sandpack>

```js App.js
import { useState } from 'react';
import { initialLetters } from './data.js';
import Letter from './Letter.js';

export default function MailClient() {
const [letters, setLetters] = useState(initialLetters);
const [highlightedId, setHighlightedId ] = useState(null);

function handleHover(letterId) {
setHighlightedId(letterId);
}

function handleStar(starredId) {
setLetters(letters.map(letter => {
if (letter.id === starredId) {
return {
...letter,
isStarred: !letter.isStarred
};
} else {
return letter;
}
}));
}

return (
<>
<h2>Inbox</h2>
<ul>
```

{letters.map(letter => (
<Letter
key={letter.id}
letter={letter}
isHighlighted={
letter.id === highlightedId
}
onHover={handleHover}
onToggleStar={handleStar}
/>
))}

</>
);
}
```

```js Letter.js
export default function Letter({
letter,
isHighlighted,
onHover,
onToggleStar,
}) {
return (
<li
className={
isHighlighted ? 'highlighted' : ''
}
onFocus={() => {
onHover(letter.id);
}}
onPointerMove={() => {
onHover(letter.id);
}}
>

<button onClick={() => {
onToggleStar(letter.id);
}}>
{letter.isStarred ? 'Unstar' : 'Star'}
</button>
{letter.subject}
</li>
)
}
```

```js data.js
export const initialLetters = [{
id: 0,
subject: 'Ready for adventure?',
isStarred: true,
}, {
id: 1,
subject: 'Time to check in!',
isStarred: false,
}, {
id: 2,
subject: 'Festival Begins in Just SEVEN Days!',
isStarred: false,
}];
```

```css
button { margin: 5px; }
li { border-radius: 5px; }
.highlighted { background: #d2eaff; }
```

</Sandpack>

</Solution>

Implement multiple selection {/*implement-multiple-selection*/}
```

In this example, each `Letter` has an `isSelected` prop and an `onToggle` handler that marks it as selected. This works, but the state is stored as a `selectedId` (either `null` or an ID), so only one letter can get selected at any given time.

Change the state structure to support multiple selection. (How would you structure it? Think about this before writing the code.) Each checkbox should become independent from the others. Clicking a selected letter should uncheck it. Finally, the footer should show the correct number of the selected items.

<Hint>

Instead of a single selected ID, you might want to hold an array or a [Set](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set) of selected IDs in state.

</Hint>

<Sandpack>

```js App.js
import { useState } from 'react';
import { letters } from './data.js';
import Letter from './Letter.js';

export default function MailClient() {
const [selectedId, setSelectedId] = useState(null);

// TODO: allow multiple selection
const selectedCount = 1;

function handleToggle(toggledId) {
// TODO: allow multiple selection
setSelectedId(toggledId);
}

return (
<>
<h2>Inbox</h2>
<ul>
{letters.map(letter => (
<Letter
key={letter.id}
letter={letter}
isSelected={
```

```
// TODO: allow multiple selection
letter.id === selectedId
}
onToggle={handleToggle}
/>
))}
<hr />
<p>
<b>
You selected {selectedCount} letters
</b>
</p>
</ul>
</>
);
}
```

```js Letter.js
export default function Letter({
letter,
onToggle,
isSelected,
}) {
return (
<li className={
isSelected ? 'selected' : ''
}>
<label>
<input
type="checkbox"
checked={isSelected}
onChange={() => {
onToggle(letter.id);
}}
/>
```

{letter.subject}

</label>

)

}

```

```js data.js
export const letters = [{
id: 0,
subject: 'Ready for adventure?',
isStarred: true,
}, {
id: 1,
subject: 'Time to check in!',
isStarred: false,
}, {
id: 2,
subject: 'Festival Begins in Just SEVEN Days!',
isStarred: false,
}];
```

```css
input { margin: 5px; }
li { border-radius: 5px; }
label { width: 100%; padding: 5px; display: inline-block; }
.selected { background: #d2eaff; }
```

</Sandpack>

<Solution>

Instead of a single `selectedId`, keep a `selectedIds` *array* in state. For example, if you select the first and the last letter, it would contain `[0, 2]`. When nothing is selected, it would be an empty `[]` array:

<Sandpack>

```js App.js

```
import { useState } from 'react';
import { letters } from './data.js';
import Letter from './Letter.js';

export default function MailClient() {
const [selectedIds, setSelectedIds] = useState([]);

const selectedCount = selectedIds.length;

function handleToggle(toggledId) {
// Was it previously selected?
if (selectedIds.includes(toggledId)) {
// Then remove this ID from the array.
setSelectedIds(selectedIds.filter(id =>
id !== toggledId
));
} else {
// Otherwise, add this ID to the array.
setSelectedIds([
...selectedIds,
toggledId
]);
}
}

return (
<>
<h2>Inbox</h2>

{letters.map(letter => (
<Letter
key={letter.id}
letter={letter}
isSelected={
selectedIds.includes(letter.id)
}
onToggle={handleToggle}
/>
```

```
))}
<hr />
<p>
<b>
You selected {selectedCount} letters
</b>
</p>
</ul>
</>
);
}
```

```js Letter.js
export default function Letter({
letter,
onToggle,
isSelected,
}) {
return (
<li className={
isSelected ? 'selected' : ''
}>
<label>
<input
type="checkbox"
checked={isSelected}
onChange={() => {
onToggle(letter.id);
}}
/>
{letter.subject}
</label>
</li>
)
}
```

```

```js data.js
export const letters = [{
id: 0,
subject: 'Ready for adventure?',
isStarred: true,
}, {
id: 1,
subject: 'Time to check in!',
isStarred: false,
}, {
id: 2,
subject: 'Festival Begins in Just SEVEN Days!',
isStarred: false,
}];
```

```css
input { margin: 5px; }
li { border-radius: 5px; }
label { width: 100%; padding: 5px; display: inline-block; }
.selected { background: #d2eaff; }
```

</Sandpack>

One minor downside of using an array is that for each item, you're calling `selectedIds.includes(letter.id)` to check whether it's selected. If the array is very large, this can become a performance problem because array search with [`includes()`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/includes) takes linear time, and you're doing this search for each individual item.

To fix this, you can hold a [Set](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set) in state instead, which provides a fast [`has()`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set/has) operation:

<Sandpack>

```js App.js
import { useState } from 'react';
```
```

```
import { letters } from './data.js';
import Letter from './Letter.js';

export default function MailClient() {
const [selectedIds, setSelectedIds] = useState(
new Set()
);

const selectedCount = selectedIds.size;

function handleToggle(toggledId) {
// Create a copy (to avoid mutation).
const nextIds = new Set(selectedIds);
if (nextIds.has(toggledId)) {
nextIds.delete(toggledId);
} else {
nextIds.add(toggledId);
}
setSelectedIds(nextIds);
}

return (
<>
<h2>Inbox</h2>
<ul>
{letters.map(letter => (
<Letter
key={letter.id}
letter={letter}
isSelected={
selectedIds.has(letter.id)
}
onToggle={handleToggle}
/>
))}
<hr />
<p>
<b>
```

````
You selected {selectedCount} letters
</b>
</p>
</ul>
</>
);
}
```

```js Letter.js
export default function Letter({
letter,
onToggle,
isSelected,
}) {
return (
<li className={
isSelected ? 'selected' : ''
}>
<label>
<input
type="checkbox"
checked={isSelected}
onChange={() => {
onToggle(letter.id);
}}
/>
{letter.subject}
</label>
</li>
)
}
```

```js data.js
export const letters = [{
id: 0,
````

subject: 'Ready for adventure?',

isStarred: true,

}, {

id: 1,

subject: 'Time to check in!',

isStarred: false,

}, {

id: 2,

subject: 'Festival Begins in Just SEVEN Days!',

isStarred: false,

}];

```

```css

input { margin: 5px; }

li { border-radius: 5px; }

label { width: 100%; padding: 5px; display: inline-block; }

.selected { background: #d2eaff; }

```

</Sandpack>

Now each item does a `selectedIds.has(letter.id)` check, which is very fast.

Keep in mind that you [should not mutate objects in state](/learn/updating-objects-in-state), and that includes Sets, too. This is why the `handleToggle` function creates a *copy* of the Set first, and then updates that copy.

</Solution>

</Challenges>

---

title: Responding to Events

<Intro>

React lets you add *event handlers* to your JSX. Event handlers are your own functions that will be triggered in response to interactions like clicking, hovering, focusing form inputs, and so on.

</Intro>

<YouWillLearn>
```

* Different ways to write an event handler

* How to pass event handling logic from a parent component

* How events propagate and how to stop them

</YouWillLearn>

Adding event handlers {/*adding-event-handlers*/}

To add an event handler, you will first define a function and then [pass it as a prop](/learn/passing-props-to-a-component) to the appropriate JSX tag. For example, here is a button that doesn't do anything yet:

<Sandpack>

```js
export default function Button() {
return (
<button>
I don't do anything
</button>
);
}
```

</Sandpack>

You can make it show a message when a user clicks by following these three steps:

1. Declare a function called `handleClick` *inside* your `Button` component.
2. Implement the logic inside that function (use `alert` to show the message).
3. Add `onClick={handleClick}` to the `<button>` JSX.

<Sandpack>

```js
export default function Button() {
function handleClick() {
alert('You clicked me!');
}
return (
<button onClick={handleClick}>
Click me
```

</button>

);

}

```

```css

button { margin-right: 10px; }

```

</Sandpack>

You defined the `handleClick` function and then [passed it as a prop](/learn/passing-props-to-a-component) to `<button>`. `handleClick` is an **event handler.** Event handler functions:

* Are usually defined *inside* your components.

* Have names that start with `handle`, followed by the name of the event.

By convention, it is common to name event handlers as `handle` followed by the event name. You'll often see `onClick={handleClick}`, `onMouseEnter={handleMouseEnter}`, and so on.

Alternatively, you can define an event handler inline in the JSX:

```jsx

<button onClick={function handleClick() {

alert('You clicked me!');

}}>

```

Or, more concisely, using an arrow function:

```jsx

<button onClick={() => {

alert('You clicked me!');

}}>

```

All of these styles are equivalent. Inline event handlers are convenient for short functions.

<Pitfall>

Functions passed to event handlers must be passed, not called. For example:

| passing a function (correct) | calling a function (incorrect) |

| ---------- | ---------- |
```

| `<button onClick={handleClick}>` | `<button onClick={handleClick()}>` |

The difference is subtle. In the first example, the `handleClick` function is passed as an `onClick` event handler. This tells React to remember it and only call your function when the user clicks the button.

In the second example, the `()` at the end of `handleClick()` fires the function *immediately* during [rendering](/learn/render-and-commit), without any clicks. This is because JavaScript inside the [JSX `{` and `}`](/learn/javascript-in-jsx-with-curly-braces) executes right away.

When you write code inline, the same pitfall presents itself in a different way:

| passing a function (correct) | calling a function (incorrect) |

| ---------- | ---------- |

| `<button onClick={() => alert('...')}>` | `<button onClick={alert('...')}>` |

Passing inline code like this won't fire on click—it fires every time the component renders:

```jsx
// This alert fires when the component renders, not when clicked!
<button onClick={alert('You clicked me!')}>
```

If you want to define your event handler inline, wrap it in an anonymous function like so:

```jsx
<button onClick={() => alert('You clicked me!')}>
```

Rather than executing the code inside with every render, this creates a function to be called later.

In both cases, what you want to pass is a function:

* `<button onClick={handleClick}>` passes the `handleClick` function.
* `<button onClick={() => alert('...')}>` passes the `() => alert('...')` function.

[Read more about arrow functions.](https://javascript.info/arrow-functions-basics)

</Pitfall>

Reading props in event handlers {/*reading-props-in-event-handlers*/}

Because event handlers are declared inside of a component, they have access to the component's props. Here is a button that, when clicked, shows an alert with its `message` prop:

<Sandpack>

```js
function AlertButton({ message, children }) {
```

```
return (
<button onClick={() => alert(message)}>
{children}
</button>
);
}

export default function Toolbar() {
return (
<div>
<AlertButton message="Playing!">
Play Movie
</AlertButton>
<AlertButton message="Uploading!">
Upload Image
</AlertButton>
</div>
);
}
```

```css
button { margin-right: 10px; }
```

</Sandpack>

This lets these two buttons show different messages. Try changing the messages passed to them.

Passing event handlers as props {/*passing-event-handlers-as-props*/}

Often you'll want the parent component to specify a child's event handler. Consider buttons: depending on where you're using a `Button` component, you might want to execute a different function—perhaps one plays a movie and another uploads an image.

To do this, pass a prop the component receives from its parent as the event handler like so:

<Sandpack>

```js
function Button({ onClick, children }) {
return (
```

```
<button onClick={onClick}>
{children}
</button>
);
}

function PlayButton({ movieName }) {
function handlePlayClick() {
alert(`Playing ${movieName}!`);
}

return (
<Button onClick={handlePlayClick}>
Play "{movieName}"
</Button>
);
}

function UploadButton() {
return (
<Button onClick={() => alert('Uploading!')}>
Upload Image
</Button>
);
}

export default function Toolbar() {
return (
<div>
<PlayButton movieName="Kiki's Delivery Service" />
<UploadButton />
</div>
);
}
```

```css
button { margin-right: 10px; }
```

</Sandpack>

Here, the `Toolbar` component renders a `PlayButton` and an `UploadButton`:

- `PlayButton` passes `handlePlayClick` as the `onClick` prop to the `Button` inside.
- `UploadButton` passes `() => alert('Uploading!')` as the `onClick` prop to the `Button` inside.

Finally, your `Button` component accepts a prop called `onClick`. It passes that prop directly to the built-in browser `<button>` with `onClick={onClick}`. This tells React to call the passed function on click.

If you use a [design system](https://uxdesign.cc/everything-you-need-to-know-about-design-systems-54b109851969), it's common for components like buttons to contain styling but not specify behavior. Instead, components like `PlayButton` and `UploadButton` will pass event handlers down.

Naming event handler props {/*naming-event-handler-props*/}

Built-in components like `<button>` and `<div>` only support [browser event names](/reference/react-dom/components/common#common-props) like `onClick`. However, when you're building your own components, you can name their event handler props any way that you like.

By convention, event handler props should start with `on`, followed by a capital letter.

For example, the `Button` component's `onClick` prop could have been called `onSmash`:

<Sandpack>

```js
function Button({ onSmash, children }) {
return (
<button onClick={onSmash}>
{children}
</button>
);
}

export default function App() {
return (
<div>
<Button onSmash={() => alert('Playing!')}>
Play Movie
</Button>
<Button onSmash={() => alert('Uploading!')}>
Upload Image
</Button>
```

);

}

```

```css

button { margin-right: 10px; }

```

</Sandpack>

In this example, `<button onClick={onSmash}>` shows that the browser `<button>` (lowercase) still needs a prop called `onClick`, but the prop name received by your custom `Button` component is up to you!

When your component supports multiple interactions, you might name event handler props for app-specific concepts. For example, this `Toolbar` component receives `onPlayMovie` and `onUploadImage` event handlers:

<Sandpack>

```js

export default function App() {

return (

<Toolbar

onPlayMovie={() => alert('Playing!')}

onUploadImage={() => alert('Uploading!')}

/>

);

}

function Toolbar({ onPlayMovie, onUploadImage }) {

return (

<Button onClick={onPlayMovie}>

Play Movie

</Button>

<Button onClick={onUploadImage}>

Upload Image

</Button>

```
);
}

function Button({ onClick, children }) {
return (
<button onClick={onClick}>
{children}
</button>
);
}
```

```css
button { margin-right: 10px; }
```

</Sandpack>

Notice how the `App` component does not need to know *what* `Toolbar` will do with `onPlayMovie` or `onUploadImage`. That's an implementation detail of the `Toolbar`. Here, `Toolbar` passes them down as `onClick` handlers to its `Button`s, but it could later also trigger them on a keyboard shortcut. Naming props after app-specific interactions like `onPlayMovie` gives you the flexibility to change how they're used later.

<Note>

Make sure that you use the appropriate HTML tags for your event handlers. For example, to handle clicks, use [`<button onClick={handleClick}>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/button) instead of `<div onClick={handleClick}>`. Using a real browser `<button>` enables built-in browser behaviors like keyboard navigation. If you don't like the default browser styling of a button and want to make it look more like a link or a different UI element, you can achieve it with CSS. [Learn more about writing accessible markup.](https://developer.mozilla.org/en-US/docs/Learn/Accessibility/HTML)

</Note>

Event propagation {/*event-propagation*/}

Event handlers will also catch events from any children your component might have. We say that an event "bubbles" or "propagates" up the tree: it starts with where the event happened, and then goes up the tree.

This `<div>` contains two buttons. Both the `<div>` *and* each button have their own `onClick` handlers. Which handlers do you think will fire when you click a button?

<Sandpack>

```js
```

```
export default function Toolbar() {
return (
<div className="Toolbar" onClick={() => {
alert('You clicked on the toolbar!');
}}>
<button onClick={() => alert('Playing!')}>
Play Movie
</button>
<button onClick={() => alert('Uploading!')}>
Upload Image
</button>
</div>
);
}
```

```css
.Toolbar {
background: #aaa;
padding: 5px;
}
button { margin: 5px; }
```

</Sandpack>

If you click on either button, its `onClick` will run first, followed by the parent `<div>`'s `onClick`. So two messages will appear. If you click the toolbar itself, only the parent `<div>`'s `onClick` will run.

<Pitfall>

All events propagate in React except `onScroll`, which only works on the JSX tag you attach it to.

</Pitfall>

Stopping propagation {/*stopping-propagation*/}

Event handlers receive an **event object** as their only argument. By convention, it's usually called `e`, which stands for "event". You can use this object to read information about the event.

That event object also lets you stop the propagation. If you want to prevent an event from reaching parent components, you need to call `e.stopPropagation()` like this `Button` component does:

<Sandpack>

```js
function Button({ onClick, children }) {
return (
<button onClick={e => {
e.stopPropagation();
onClick();
}}>
{children}
</button>
);
}

export default function Toolbar() {
return (
<div className="Toolbar" onClick={() => {
alert('You clicked on the toolbar!');
}}>
<Button onClick={() => alert('Playing!')}>
Play Movie
</Button>
<Button onClick={() => alert('Uploading!')}>
Upload Image
</Button>
</div>
);
}
```

```css
.Toolbar {
background: #aaa;
padding: 5px;
}
button { margin: 5px; }
```

</Sandpack>

When you click on a button:

1. React calls the `onClick` handler passed to `<button>`.

2. That handler, defined in `Button`, does the following:

* Calls `e.stopPropagation()`, preventing the event from bubbling further.

* Calls the `onClick` function, which is a prop passed from the `Toolbar` component.

3. That function, defined in the `Toolbar` component, displays the button's own alert.

4. Since the propagation was stopped, the parent `<div>`'s `onClick` handler does *not* run.

As a result of `e.stopPropagation()`, clicking on the buttons now only shows a single alert (from the `<button>`) rather than the two of them (from the `<button>` and the parent toolbar `<div>`). Clicking a button is not the same thing as clicking the surrounding toolbar, so stopping the propagation makes sense for this UI.

<DeepDive>

Capture phase events {/*capture-phase-events*/}

In rare cases, you might need to catch all events on child elements, *even if they stopped propagation*. For example, maybe you want to log every click to analytics, regardless of the propagation logic. You can do this by adding `Capture` at the end of the event name:

```js
<div onClickCapture={() => { /* this runs first */ }}>
<button onClick={e => e.stopPropagation()} />
<button onClick={e => e.stopPropagation()} />
</div>
```

Each event propagates in three phases:

1. It travels down, calling all `onClickCapture` handlers.

2. It runs the clicked element's `onClick` handler.

3. It travels upwards, calling all `onClick` handlers.

Capture events are useful for code like routers or analytics, but you probably won't use them in app code.

</DeepDive>

Passing handlers as alternative to propagation {/*passing-handlers-as-alternative-to-propagation*/}

Notice how this click handler runs a line of code _and then_ calls the `onClick` prop passed by the parent:

```js {4,5}
function Button({ onClick, children }) {
return (
<button onClick={e => {
e.stopPropagation();
onClick();
}}>
{children}
</button>
);
}
```

You could add more code to this handler before calling the parent `onClick` event handler, too. This pattern provides an *alternative* to propagation. It lets the child component handle the event, while also letting the parent component specify some additional behavior. Unlike propagation, it's not automatic. But the benefit of this pattern is that you can clearly follow the whole chain of code that executes as a result of some event.

If you rely on propagation and it's difficult to trace which handlers execute and why, try this approach instead.

Preventing default behavior {/*preventing-default-behavior*/}

Some browser events have default behavior associated with them. For example, a `<form>` submit event, which happens when a button inside of it is clicked, will reload the whole page by default:

<Sandpack>

```js
export default function Signup() {
return (
<form onSubmit={() => alert('Submitting!')}>
<input />
<button>Send</button>
</form>
);
}
```

```css
button { margin-left: 5px; }
```

```

</Sandpack>

You can call `e.preventDefault()` on the event object to stop this from happening:

<Sandpack>

```js
export default function Signup() {
return (
<form onSubmit={e => {
e.preventDefault();
alert('Submitting!');
}}>
<input />
<button>Send</button>
</form>
);
}
```

```css
button { margin-left: 5px; }
```

</Sandpack>

Don't confuse `e.stopPropagation()` and `e.preventDefault()`. They are both useful, but are unrelated:

* [`e.stopPropagation()`](https://developer.mozilla.org/docs/Web/API/Event/stopPropagation) stops the event handlers attached to the tags above from firing.

* [`e.preventDefault()`](https://developer.mozilla.org/docs/Web/API/Event/preventDefault) prevents the default browser behavior for the few events that have it.

Can event handlers have side effects? {/*can-event-handlers-have-side-effects*/}

Absolutely! Event handlers are the best place for side effects.

Unlike rendering functions, event handlers don't need to be [pure](/learn/keeping-components-pure), so it's a great place to *change* something—for example, change an input's value in response to typing, or change a list in response to a button press. However, in order to change some information, you first need some way to store it. In React, this is done by using [state, a component's memory.](/learn/state-a-components-memory) You will learn all about it on the next page.
```

<Recap>

* You can handle events by passing a function as a prop to an element like `<button>`.
* Event handlers must be passed, **not called!** `onClick={handleClick}`, not `onClick={handleClick()}`.
* You can define an event handler function separately or inline.
* Event handlers are defined inside a component, so they can access props.
* You can declare an event handler in a parent and pass it as a prop to a child.
* You can define your own event handler props with application-specific names.
* Events propagate upwards. Call `e.stopPropagation()` on the first argument to prevent that.
* Events may have unwanted default browser behavior. Call `e.preventDefault()` to prevent that.
* Explicitly calling an event handler prop from a child handler is a good alternative to propagation.

</Recap>

<Challenges>

Fix an event handler {/*fix-an-event-handler*/}

Clicking this button is supposed to switch the page background between white and black. However, nothing happens when you click it. Fix the problem. (Don't worry about the logic inside `handleClick`—that part is fine.)

<Sandpack>

```js
export default function LightSwitch() {
function handleClick() {
let bodyStyle = document.body.style;
if (bodyStyle.backgroundColor === 'black') {
bodyStyle.backgroundColor = 'white';
} else {
bodyStyle.backgroundColor = 'black';
}
}

return (
<button onClick={handleClick()}>
Toggle the lights
</button>
);
}
```

```

</Sandpack>

<Solution>

The problem is that `<button onClick={handleClick()}>` _calls_ the `handleClick` function while rendering instead of _passing_ it. Removing the `()` call so that it's `<button onClick={handleClick}>` fixes the issue:

<Sandpack>

```js
export default function LightSwitch() {
function handleClick() {
let bodyStyle = document.body.style;
if (bodyStyle.backgroundColor === 'black') {
bodyStyle.backgroundColor = 'white';
} else {
bodyStyle.backgroundColor = 'black';
}
}
return (
<button onClick={handleClick}>
Toggle the lights
</button>
);
}
```

</Sandpack>

Alternatively, you could wrap the call into another function, like `<button onClick={() => handleClick()}>`:

<Sandpack>

```js
export default function LightSwitch() {
function handleClick() {
let bodyStyle = document.body.style;
if (bodyStyle.backgroundColor === 'black') {
bodyStyle.backgroundColor = 'white';
```
```

```
} else {
bodyStyle.backgroundColor = 'black';
}
}

return (
<button onClick={() => handleClick()}>
Toggle the lights
</button>
);
}
```

</Sandpack>

</Solution>

Wire up the events {/*wire-up-the-events*/}

This `ColorSwitch` component renders a button. It's supposed to change the page color. Wire it up to the `onChangeColor` event handler prop it receives from the parent so that clicking the button changes the color.

After you do this, notice that clicking the button also increments the page click counter. Your colleague who wrote the parent component insists that `onChangeColor` does not increment any counters. What else might be happening? Fix it so that clicking the button *only* changes the color, and does _not_ increment the counter.

<Sandpack>

```js ColorSwitch.js active
export default function ColorSwitch({
onChangeColor
}) {
return (
<button>
Change color
</button>
);
}
```

```js App.js hidden
```

```
import { useState } from 'react';
import ColorSwitch from './ColorSwitch.js';

export default function App() {
const [clicks, setClicks] = useState(0);

function handleClickOutside() {
setClicks(c => c + 1);
}

function getRandomLightColor() {
let r = 150 + Math.round(100 * Math.random());
let g = 150 + Math.round(100 * Math.random());
let b = 150 + Math.round(100 * Math.random());
return `rgb(${r}, ${g}, ${b})`;
}

function handleChangeColor() {
let bodyStyle = document.body.style;
bodyStyle.backgroundColor = getRandomLightColor();
}

return (
<div style={{ width: '100%', height: '100%' }} onClick={handleClickOutside}>
<ColorSwitch onChangeColor={handleChangeColor} />
<br />
<br />
<h2>Clicks on the page: {clicks}</h2>
</div>
);
}
```

</Sandpack>

<Solution>

First, you need to add the event handler, like `<button onClick={onChangeColor}>`.

However, this introduces the problem of the incrementing counter. If `onChangeColor` does not do this, as your colleague insists, then the problem is that this event propagates up, and some handler above does it. To solve this problem, you need to stop the propagation. But don't forget that you should still

call `onChangeColor`.

<Sandpack>

```js ColorSwitch.js active
export default function ColorSwitch({
onChangeColor
}) {
return (
<button onClick={e => {
e.stopPropagation();
onChangeColor();
}}>
Change color
</button>
);
}
```

```js App.js hidden
import { useState } from 'react';
import ColorSwitch from './ColorSwitch.js';

export default function App() {
const [clicks, setClicks] = useState(0);

function handleClickOutside() {
setClicks(c => c + 1);
}

function getRandomLightColor() {
let r = 150 + Math.round(100 * Math.random());
let g = 150 + Math.round(100 * Math.random());
let b = 150 + Math.round(100 * Math.random());
return `rgb(${r}, ${g}, ${b})`;
}

function handleChangeColor() {
let bodyStyle = document.body.style;
bodyStyle.backgroundColor = getRandomLightColor();
```

}

return (

<div style={{ width: '100%', height: '100%' }} onClick={handleClickOutside}>

<ColorSwitch onChangeColor={handleChangeColor} />

<h2>Clicks on the page: {clicks}</h2>

);

}

```

</Sandpack>

</Solution>

</Challenges>

---

title: Conditional Rendering

---

<Intro>

Your components will often need to display different things depending on different conditions. In React, you can conditionally render JSX using JavaScript syntax like `if` statements, `&&`, and `? :` operators.

</Intro>

<YouWillLearn>

* How to return different JSX depending on a condition

* How to conditionally include or exclude a piece of JSX

* Common conditional syntax shortcuts you'll encounter in React codebases

</YouWillLearn>

## Conditionally returning JSX {/*conditionally-returning-jsx*/}

Let's say you have a `PackingList` component rendering several `Item`s, which can be marked as packed or not:

<Sandpack>

```js

```
function Item({ name, isPacked }) {
return <li className="item">{name};
}

export default function PackingList() {
return (
<section>
<h1>Sally Ride's Packing List</h1>

<Item
isPacked={true}
name="Space suit"
/>
<Item
isPacked={true}
name="Helmet with a golden leaf"
/>
<Item
isPacked={false}
name="Photo of Tam"
/>

</section>
);
}
```

</Sandpack>

Notice that some of the `Item` components have their `isPacked` prop set to `true` instead of `false`. You want to add a checkmark (✔) to packed items if `isPacked={true}`.

You can write this as an [`if`/`else` statement](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/if...else) like so:

```js
if (isPacked) {
return <li className="item">{name} ✔</li>;
}
```

return <li className="item">{name};
```

If the `isPacked` prop is `true`, this code **returns a different JSX tree.** With this change, some of the items get a checkmark at the end:

<Sandpack>

```js
function Item({ name, isPacked }) {
if (isPacked) {
return <li className="item">{name} ✔</li>;
}
return <li className="item">{name}</li>;
}

export default function PackingList() {
return (
<section>
<h1>Sally Ride's Packing List</h1>
<ul>
<Item
isPacked={true}
name="Space suit"
/>
<Item
isPacked={true}
name="Helmet with a golden leaf"
/>
<Item
isPacked={false}
name="Photo of Tam"
/>
</ul>
</section>
);
}
```
```

</Sandpack>

Try editing what gets returned in either case, and see how the result changes!

Notice how you're creating branching logic with JavaScript's `if` and `return` statements. In React, control flow (like conditions) is handled by JavaScript.

Conditionally returning nothing with `null` {/*conditionally-returning-nothing-with-null*/}

In some situations, you won't want to render anything at all. For example, say you don't want to show packed items at all. A component must return something. In this case, you can return `null`:

```js
if (isPacked) {
return null;
}
return <li className="item">{name}</li>;
```

If `isPacked` is true, the component will return nothing, `null`. Otherwise, it will return JSX to render.

<Sandpack>

```js
function Item({ name, isPacked }) {
if (isPacked) {
return null;
}
return <li className="item">{name}</li>;
}

export default function PackingList() {
return (
<section>
<h1>Sally Ride's Packing List</h1>
<ul>
<Item
isPacked={true}
name="Space suit"
/>
<Item
isPacked={true}
```

name="Helmet with a golden leaf"

/>

<Item

isPacked={false}

name="Photo of Tam"

/>

</section>

);

}

```

</Sandpack>

In practice, returning `null` from a component isn't common because it might surprise a developer trying to render it. More often, you would conditionally include or exclude the component in the parent component's JSX. Here's how to do that!

## Conditionally including JSX {/*conditionally-including-jsx*/}

In the previous example, you controlled which (if any!) JSX tree would be returned by the component. You may already have noticed some duplication in the render output:

```js

<li className="item">{name} ✔</li>

```

is very similar to

```js

<li className="item">{name}</li>

```

Both of the conditional branches return `<li className="item">...</li>`:

```js

if (isPacked) {

return <li className="item">{name} ✔</li>;

}

return <li className="item">{name}</li>;

```
```

While this duplication isn't harmful, it could make your code harder to maintain. What if you want to change the `className`? You'd have to do it in two places in your code! In such a situation, you could conditionally include a little JSX to make your code more [DRY.](https://en.wikipedia.org/wiki/Don%27t_repeat_yourself)

Conditional (ternary) operator (`? :`) {/*conditional-ternary-operator--*/}

JavaScript has a compact syntax for writing a conditional expression -- the [conditional operator](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_Operator) or "ternary operator".

Instead of this:

```js
if (isPacked) {
return <li className="item">{name} ✔</li>;
}
return <li className="item">{name}</li>;
```

You can write this:

```js
return (
<li className="item">
{isPacked ? name + ' ✔' : name}
</li>
);
```

You can read it as *"if `isPacked` is true, then (`?`) render `name + ' ✔'`, otherwise (`:`) render `name`"*.

<DeepDive>

Are these two examples fully equivalent? {/*are-these-two-examples-fully-equivalent*/}

If you're coming from an object-oriented programming background, you might assume that the two examples above are subtly different because one of them may create two different "instances" of `<li>`. But JSX elements aren't "instances" because they don't hold any internal state and aren't real DOM nodes. They're lightweight descriptions, like blueprints. So these two examples, in fact, *are* completely equivalent. [Preserving and Resetting State](/learn/preserving-and-resetting-state) goes into detail about how this works.

</DeepDive>

Now let's say you want to wrap the completed item's text into another HTML tag, like `<del>` to strike it out. You can add even more newlines and parentheses so that it's easier to nest more JSX in each of the cases:

<Sandpack>

```js
function Item({ name, isPacked }) {
return (
<li className="item">
{isPacked ? (
<del>
{name + ' ✔'}
</del>
) : (
name
)}
</li>
);
}

export default function PackingList() {
return (
<section>
<h1>Sally Ride's Packing List</h1>
<ul>
<Item
isPacked={true}
name="Space suit"
/>
<Item
isPacked={true}
name="Helmet with a golden leaf"
/>
<Item
isPacked={false}
name="Photo of Tam"
/>
```


</section>

);

}

```

</Sandpack>

This style works well for simple conditions, but use it in moderation. If your components get messy with too much nested conditional markup, consider extracting child components to clean things up. In React, markup is a part of your code, so you can use tools like variables and functions to tidy up complex expressions.

### Logical AND operator (`&&`) {/*logical-and-operator-*/}

Another common shortcut you'll encounter is the [JavaScript logical AND (`&&`) operator.](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_AND#:~:text=The%20logical%20AND%20(%20%26%26%20)%20operator,it%20returns%20a%20Boolean%20value.) Inside React components, it often comes up when you want to render some JSX when the condition is true, **or render nothing otherwise.** With `&&`, you could conditionally render the checkmark only if `isPacked` is `true`:

```js

return (

<li className="item">

{name} {isPacked && '✔'}

</li>

);

```

You can read this as *"if `isPacked`, then (`&&`) render the checkmark, otherwise, render nothing"*.

Here it is in action:

<Sandpack>

```js

function Item({ name, isPacked }) {

return (

<li className="item">

{name} {isPacked && '✔'}

</li>

);

}

```
export default function PackingList() {
return (
<section>
<h1>Sally Ride's Packing List</h1>

<Item
isPacked={true}
name="Space suit"
/>
<Item
isPacked={true}
name="Helmet with a golden leaf"
/>
<Item
isPacked={false}
name="Photo of Tam"
/>

</section>
);
}
```

</Sandpack>

A [JavaScript && expression](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_AND) returns the value of its right side (in our case, the checkmark) if the left side (our condition) is `true`. But if the condition is `false`, the whole expression becomes `false`. React considers `false` as a "hole" in the JSX tree, just like `null` or `undefined`, and doesn't render anything in its place.

<Pitfall>

Don't put numbers on the left side of `&&`.

To test the condition, JavaScript converts the left side to a boolean automatically. However, if the left side is `0`, then the whole expression gets that value (`0`), and React will happily render `0` rather than nothing.

For example, a common mistake is to write code like `messageCount && <p>New messages</p>`. It's easy to assume that it renders nothing when `messageCount` is `0`, but it really renders the `0` itself!

To fix it, make the left side a boolean: `messageCount > 0 && <p>New messages</p>`.

</Pitfall>

Conditionally assigning JSX to a variable {/*conditionally-assigning-jsx-to-a-variable*/}

When the shortcuts get in the way of writing plain code, try using an `if` statement and a variable. You can reassign variables defined with [`let`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let), so start by providing the default content you want to display, the name:

```js
let itemContent = name;
```

Use an `if` statement to reassign a JSX expression to `itemContent` if `isPacked` is `true`:

```js
if (isPacked) {
itemContent = name + " ✔";
}
```

[Curly braces open the "window into JavaScript".](/learn/javascript-in-jsx-with-curly-braces#using-curly-braces-a-window-into-the-javascript-world) Embed the variable with curly braces in the returned JSX tree, nesting the previously calculated expression inside of JSX:

```js
<li className="item">
{itemContent}
</li>
```

This style is the most verbose, but it's also the most flexible. Here it is in action:

<Sandpack>

```js
function Item({ name, isPacked }) {
let itemContent = name;
if (isPacked) {
itemContent = name + " ✔";
}
return (
```

<li className="item">
{itemContent}

);
}

export default function PackingList() {
return (
<section>
<h1>Sally Ride's Packing List</h1>

<Item
isPacked={true}
name="Space suit"
/>
<Item
isPacked={true}
name="Helmet with a golden leaf"
/>
<Item
isPacked={false}
name="Photo of Tam"
/>

</section>
);
}
```

</Sandpack>

Like before, this works not only for text, but for arbitrary JSX too:

<Sandpack>

```js
function Item({ name, isPacked }) {
let itemContent = name;
if (isPacked) {

```
itemContent = (
<del>
{name + " ✔"}
</del>
);
}
return (
<li className="item">
{itemContent}
</li>
);
}

export default function PackingList() {
return (
<section>
<h1>Sally Ride's Packing List</h1>
<ul>
<Item
isPacked={true}
name="Space suit"
/>
<Item
isPacked={true}
name="Helmet with a golden leaf"
/>
<Item
isPacked={false}
name="Photo of Tam"
/>
</ul>
</section>
);
}
```

</Sandpack>

If you're not familiar with JavaScript, this variety of styles might seem overwhelming at first. However, learning them will help you read and write any JavaScript code -- and not just React components! Pick the one you prefer for a start, and then consult this reference again if you forget how the other ones work.

<Recap>

* In React, you control branching logic with JavaScript.

* You can return a JSX expression conditionally with an `if` statement.

* You can conditionally save some JSX to a variable and then include it inside other JSX by using the curly braces.

* In JSX, `{cond ? <A /> : <B />}` means *"if `cond`, render `<A />`, otherwise `<B />`"*.

* In JSX, `{cond && <A />}` means *"if `cond`, render `<A />`, otherwise nothing"*.

* The shortcuts are common, but you don't have to use them if you prefer plain `if`.

</Recap>

<Challenges>

Show an icon for incomplete items with `? :` {/*show-an-icon-for-incomplete-items-with--*/}

Use the conditional operator (`cond ? a : b`) to render a ■ if `isPacked` isn't `true`.

<Sandpack>

```js
function Item({ name, isPacked }) {
return (
<li className="item">
{name} {isPacked && '✔'}
</li>
);
}

export default function PackingList() {
return (
<section>
<h1>Sally Ride's Packing List</h1>
<ul>
<Item
isPacked={true}
name="Space suit"
```

/>
<Item
isPacked={true}
name="Helmet with a golden leaf"
/>
<Item
isPacked={false}
name="Photo of Tam"
/>

</section>
);
}
```

</Sandpack>

<Solution>

<Sandpack>

```js
function Item({ name, isPacked }) {
return (
<li className="item">
{name} {isPacked ? '✔' : '■'}
</li>
);
}

export default function PackingList() {
return (
<section>
<h1>Sally Ride's Packing List</h1>
<ul>
<Item
isPacked={true}
name="Space suit"
/>

<Item

isPacked={true}

name="Helmet with a golden leaf"

/>

<Item

isPacked={false}

name="Photo of Tam"

/>

</ul>

</section>

);

}

```

</Sandpack>

</Solution>

Show the item importance with `&&` {/*show-the-item-importance-with-*/}

In this example, each `Item` receives a numerical `importance` prop. Use the `&&` operator to render "_(Importance: X)_" in italics, but only for items that have non-zero importance. Your item list should end up looking like this:

* Space suit _(Importance: 9)_

* Helmet with a golden leaf

* Photo of Tam _(Importance: 6)_

Don't forget to add a space between the two labels!

<Sandpack>

```js

function Item({ name, importance }) {

return (

<li className="item">

{name}

</li>

);

}

export default function PackingList() {

return (
<section>
<h1>Sally Ride's Packing List</h1>
<ul>
<Item
importance={9}
name="Space suit"
/>
<Item
importance={0}
name="Helmet with a golden leaf"
/>
<Item
importance={6}
name="Photo of Tam"
/>
</ul>
</section>
);
}
```

</Sandpack>

<Solution>

This should do the trick:

<Sandpack>

```js
function Item({ name, importance }) {
return (
<li className="item">
{name}
{importance > 0 && ' '}
{importance > 0 &&
(Importance: {importance})
}

```
</li>
);
}

export default function PackingList() {
return (
<section>
<h1>Sally Ride's Packing List</h1>
<ul>
<Item
importance={9}
name="Space suit"
/>
<Item
importance={0}
name="Helmet with a golden leaf"
/>
<Item
importance={6}
name="Photo of Tam"
/>
</ul>
</section>
);
}
```

</Sandpack>

Note that you must write `importance > 0 && ...` rather than `importance && ...` so that if the `importance` is `0`, `0` isn't rendered as the result!

In this solution, two separate conditions are used to insert a space between the name and the importance label. Alternatively, you could use a fragment with a leading space: `importance > 0 && <> <i>...</i></>` or add a space immediately inside the `<i>`: `importance > 0 && <i> ...</i>`.

</Solution>

Refactor a series of `? :` to `if` and variables {/*refactor-a-series-of---to-if-and-variables*/}

This `Drink` component uses a series of `? :` conditions to show different information depending on whether the `name` prop is `"tea"` or `"coffee"`. The problem is that the information about each drink is spread across multiple conditions. Refactor this code to use a single `if` statement instead of three `? :` conditions.

<Sandpack>

```js
function Drink({ name }) {
return (
<section>
<h1>{name}</h1>
<dl>
<dt>Part of plant</dt>
<dd>{name === 'tea' ? 'leaf' : 'bean'}</dd>
<dt>Caffeine content</dt>
<dd>{name === 'tea' ? '15–70 mg/cup' : '80–185 mg/cup'}</dd>
<dt>Age</dt>
<dd>{name === 'tea' ? '4,000+ years' : '1,000+ years'}</dd>
</dl>
</section>
);
}

export default function DrinkList() {
return (
<div>
<Drink name="tea" />
<Drink name="coffee" />
</div>
);
}
```

</Sandpack>

Once you've refactored the code to use `if`, do you have further ideas on how to simplify it?

<Solution>

There are multiple ways you could go about this, but here is one starting point:

<Sandpack>

```js
function Drink({ name }) {
let part, caffeine, age;
if (name === 'tea') {
part = 'leaf';
caffeine = '15–70 mg/cup';
age = '4,000+ years';
} else if (name === 'coffee') {
part = 'bean';
caffeine = '80–185 mg/cup';
age = '1,000+ years';
}
return (
<section>
<h1>{name}</h1>
<dl>
<dt>Part of plant</dt>
<dd>{part}</dd>
<dt>Caffeine content</dt>
<dd>{caffeine}</dd>
<dt>Age</dt>
<dd>{age}</dd>
</dl>
</section>
);
}

export default function DrinkList() {
return (
<div>
<Drink name="tea" />
<Drink name="coffee" />
</div>
);
}
```

```

</Sandpack>

Here the information about each drink is grouped together instead of being spread across multiple conditions. This makes it easier to add more drinks in the future.

Another solution would be to remove the condition altogether by moving the information into objects:

<Sandpack>

```js
const drinks = {
tea: {
part: 'leaf',
caffeine: '15–70 mg/cup',
age: '4,000+ years'
},
coffee: {
part: 'bean',
caffeine: '80–185 mg/cup',
age: '1,000+ years'
}
};

function Drink({ name }) {
const info = drinks[name];
return (
<section>
<h1>{name}</h1>
<dl>
<dt>Part of plant</dt>
<dd>{info.part}</dd>
<dt>Caffeine content</dt>
<dd>{info.caffeine}</dd>
<dt>Age</dt>
<dd>{info.age}</dd>
</dl>
</section>
);
```
```

```
}

export default function DrinkList() {
return (
<div>
<Drink name="tea" />
<Drink name="coffee" />
</div>
);
}
```

</Sandpack>

</Solution>

</Challenges>

---
title: 'Manipulating the DOM with Refs'

<Intro>

React automatically updates the [DOM](https://developer.mozilla.org/docs/Web/API/Document_Object_Model/Introduction) to match your render output, so your components won't often need to manipulate it. However, sometimes you might need access to the DOM elements managed by React--for example, to focus a node, scroll to it, or measure its size and position. There is no built-in way to do those things in React, so you will need a *ref* to the DOM node.

</Intro>

<YouWillLearn>

- How to access a DOM node managed by React with the `ref` attribute
- How the `ref` JSX attribute relates to the `useRef` Hook
- How to access another component's DOM node
- In which cases it's safe to modify the DOM managed by React

</YouWillLearn>

Getting a ref to the node {/*getting-a-ref-to-the-node*/}

To access a DOM node managed by React, first, import the `useRef` Hook:

```js
```

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

Then, use it to declare a ref inside your component:

```js
const myRef = useRef(null);
```

Finally, pass your ref as the `ref` attribute to the JSX tag for which you want to get the DOM node:

```js
<div ref={myRef}>
```

The `useRef` Hook returns an object with a single property called `current`. Initially, `myRef.current` will be `null`. When React creates a DOM node for this `<div>`, React will put a reference to this node into `myRef.current`. You can then access this DOM node from your [event handlers](/learn/responding-to-events) and use the built-in [browser APIs](https://developer.mozilla.org/docs/Web/API/Element) defined on it.

```js
// You can use any browser APIs, for example:
myRef.current.scrollIntoView();
```

Example: Focusing a text input {/*example-focusing-a-text-input*/}

In this example, clicking the button will focus the input:

<Sandpack>

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

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

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

return (
<>
<input ref={inputRef} />
```

<button onClick={handleClick}>

Focus the input

</button>

</>

);

}

```

</Sandpack>

To implement this:

1. Declare `inputRef` with the `useRef` Hook.

2. Pass it as `<input ref={inputRef}>`. This tells React to **put this `<input>`'s DOM node into `inputRef.current`.**

3. In the `handleClick` function, read the input DOM node from `inputRef.current` and call [`focus()`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/focus) on it with `inputRef.current.focus()`.

4. Pass the `handleClick` event handler to `<button>` with `onClick`.

While DOM manipulation is the most common use case for refs, the `useRef` Hook can be used for storing other things outside React, like timer IDs. Similarly to state, refs remain between renders. Refs are like state variables that don't trigger re-renders when you set them. Read about refs in [Referencing Values with Refs.](/learn/referencing-values-with-refs)

### Example: Scrolling to an element {/*example-scrolling-to-an-element*/}

You can have more than a single ref in a component. In this example, there is a carousel of three images. Each button centers an image by calling the browser [`scrollIntoView()`](https://developer.mozilla.org/en-US/docs/Web/API/Element/scrollIntoView) method on the corresponding DOM node:

<Sandpack>

```js

import { useRef } from 'react';

export default function CatFriends() {

const firstCatRef = useRef(null);

const secondCatRef = useRef(null);

const thirdCatRef = useRef(null);

function handleScrollToFirstCat() {

firstCatRef.current.scrollIntoView({

behavior: 'smooth',

```
block: 'nearest',
inline: 'center'
});
}

function handleScrollToSecondCat() {
secondCatRef.current.scrollIntoView({
behavior: 'smooth',
block: 'nearest',
inline: 'center'
});
}

function handleScrollToThirdCat() {
thirdCatRef.current.scrollIntoView({
behavior: 'smooth',
block: 'nearest',
inline: 'center'
});
}

return (
<>
<nav>
<button onClick={handleScrollToFirstCat}>
Tom
</button>
<button onClick={handleScrollToSecondCat}>
Maru
</button>
<button onClick={handleScrollToThirdCat}>
Jellylorum
</button>
</nav>
<div>

```

```
<img
src="https://placekitten.com/g/200/200"
alt="Tom"
ref={firstCatRef}
/>
</li>
<li>
<img
src="https://placekitten.com/g/300/200"
alt="Maru"
ref={secondCatRef}
/>
</li>
<li>
<img
src="https://placekitten.com/g/250/200"
alt="Jellylorum"
ref={thirdCatRef}
/>
</li>
</ul>
</div>
</>
);
}
```

```css
div {
width: 100%;
overflow: hidden;
}

nav {
text-align: center;
}
```

```
button {
margin: .25rem;
}

ul,
li {
list-style: none;
white-space: nowrap;
}

li {
display: inline;
padding: 0.5rem;
}
```

</Sandpack>

<DeepDive>

How to manage a list of refs using a ref callback
{/*how-to-manage-a-list-of-refs-using-a-ref-callback*/}

In the above examples, there is a predefined number of refs. However, sometimes you might need a ref to each item in the list, and you don't know how many you will have. Something like this **wouldn't work**:

```js
<ul>
{items.map((item) => {
// Doesn't work!
const ref = useRef(null);
return <li ref={ref} />;
})}
</ul>
```

This is because **Hooks must only be called at the top-level of your component.** You can't call `useRef` in a loop, in a condition, or inside a `map()` call.

One possible way around this is to get a single ref to their parent element, and then use DOM manipulation methods like [`querySelectorAll`](https://developer.mozilla.org/en-US/docs/Web/API/Document/querySelectorAll) to "find" the individual child nodes from it. However, this is brittle and can break if your DOM structure

changes.

Another solution is to **pass a function to the `ref` attribute.** This is called a [`ref` callback.](/reference/react-dom/components/common#ref-callback) React will call your ref callback with the DOM node when it's time to set the ref, and with `null` when it's time to clear it. This lets you maintain your own array or a [Map](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map), and access any ref by its index or some kind of ID.

This example shows how you can use this approach to scroll to an arbitrary node in a long list:

<Sandpack>

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

export default function CatFriends() {
const itemsRef = useRef(null);

function scrollToId(itemId) {
const map = getMap();
const node = map.get(itemId);
node.scrollIntoView({
behavior: 'smooth',
block: 'nearest',
inline: 'center'
});
}

function getMap() {
if (!itemsRef.current) {
// Initialize the Map on first usage.
itemsRef.current = new Map();
}
return itemsRef.current;
}

return (
<>
<nav>
<button onClick={() => scrollToId(0)}>
Tom
```

```
</button>
<button onClick={() => scrollToId(5)}>
Maru
</button>
<button onClick={() => scrollToId(9)}>
Jellylorum
</button>
</nav>
<div>
<ul>
{catList.map(cat => (
<li
key={cat.id}
ref={(node) => {
const map = getMap();
if (node) {
map.set(cat.id, node);
} else {
map.delete(cat.id);
}
}}
>
<img
src={cat.imageUrl}
alt={'Cat #' + cat.id}
/>
</li>
))}
</ul>
</div>
</>
);
}

const catList = [];
for (let i = 0; i < 10; i++) {
```

```
catList.push({
id: i,
imageUrl: 'https://placekitten.com/250/200?image=' + i
});
}
```

```css
div {
width: 100%;
overflow: hidden;
}

nav {
text-align: center;
}

button {
margin: .25rem;
}

ul,
li {
list-style: none;
white-space: nowrap;
}

li {
display: inline;
padding: 0.5rem;
}
```

</Sandpack>

In this example, `itemsRef` doesn't hold a single DOM node. Instead, it holds a [Map](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Map) from item ID to a DOM node. ([Refs can hold any values!](/learn/referencing-values-with-refs)) The [`ref` callback](/reference/react-dom/components/common#ref-callback) on every list item takes care to update the Map:

```js
```

<li

key={cat.id}

ref={node => {

const map = getMap();

if (node) {

// Add to the Map

map.set(cat.id, node);

} else {

// Remove from the Map

map.delete(cat.id);

}

}}

>

```

This lets you read individual DOM nodes from the Map later.

</DeepDive>

## Accessing another component's DOM nodes {/*accessing-another-components-dom-nodes*/}

When you put a ref on a built-in component that outputs a browser element like `<input />`, React will set that ref's `current` property to the corresponding DOM node (such as the actual `<input />` in the browser).

However, if you try to put a ref on **your own** component, like `<MyInput />`, by default you will get `null`. Here is an example demonstrating it. Notice how clicking the button **does not** focus the input:

<Sandpack>

```js

import { useRef } from 'react';

function MyInput(props) {

return <input {...props} />;

}

export default function MyForm() {

const inputRef = useRef(null);

function handleClick() {

inputRef.current.focus();

}

return (

<>

<MyInput ref={inputRef} />

<button onClick={handleClick}>

Focus the input

</button>

</>

);

}

```

</Sandpack>

To help you notice the issue, React also prints an error to the console:

<ConsoleBlock level="error">

Warning: Function components cannot be given refs. Attempts to access this ref will fail. Did you mean to use React.forwardRef()?

</ConsoleBlock>

This happens because by default React does not let a component access the DOM nodes of other components. Not even for its own children! This is intentional. Refs are an escape hatch that should be used sparingly. Manually manipulating _another_ component's DOM nodes makes your code even more fragile.

Instead, components that _want_ to expose their DOM nodes have to **opt in** to that behavior. A component can specify that it "forwards" its ref to one of its children. Here's how `MyInput` can use the `forwardRef` API:

```js

const MyInput = forwardRef((props, ref) => {

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

});

```

This is how it works:

1. `<MyInput ref={inputRef} />` tells React to put the corresponding DOM node into `inputRef.current`. However, it's up to the `MyInput` component to opt into that--by default, it doesn't.

2. The `MyInput` component is declared using `forwardRef`. **This opts it into receiving the `inputRef` from above as the second `ref` argument** which is declared after `props`.

3. `MyInput` itself passes the `ref` it received to the `<input>` inside of it.
```

Now clicking the button to focus the input works:

<Sandpack>

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

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

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

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

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

</Sandpack>

In design systems, it is a common pattern for low-level components like buttons, inputs, and so on, to forward their refs to their DOM nodes. On the other hand, high-level components like forms, lists, or page sections usually won't expose their DOM nodes to avoid accidental dependencies on the DOM structure.

<DeepDive>

Exposing a subset of the API with an imperative handle {/*exposing-a-subset-of-the-api-with-an-imperative-handle*/}

In the above example, `MyInput` exposes the original DOM input element. This lets the parent component call `focus()` on it. However, this also lets the parent component do something else--for example, change its CSS styles. In uncommon cases, you may want to restrict the exposed functionality. You can do that with `useImperativeHandle`:

<Sandpack>

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

const MyInput = forwardRef((props, ref) => {
const realInputRef = useRef(null);
useImperativeHandle(ref, () => ({
// Only expose focus and nothing else
focus() {
realInputRef.current.focus();
},
}));
return <input {...props} ref={realInputRef} />;
});

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

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

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

</Sandpack>

Here, `realInputRef` inside `MyInput` holds the actual input DOM node. However, `useImperativeHandle` instructs React to provide your own special object as the value of a ref to the parent component. So `inputRef.current` inside the `Form` component will only have the `focus` method. In this case, the ref "handle" is not the DOM node, but the custom object you create inside `useImperativeHandle` call.

</DeepDive>

When React attaches the refs {/*when-react-attaches-the-refs*/}

In React, every update is split in [two phases](/learn/render-and-commit#step-3-react-commits-changes-to-the-dom):

* During **render,** React calls your components to figure out what should be on the screen.
* During **commit,** React applies changes to the DOM.

In general, you [don't want](/learn/referencing-values-with-refs#best-practices-for-refs) to access refs during rendering. That goes for refs holding DOM nodes as well. During the first render, the DOM nodes have not yet been created, so `ref.current` will be `null`. And during the rendering of updates, the DOM nodes haven't been updated yet. So it's too early to read them.

React sets `ref.current` during the commit. Before updating the DOM, React sets the affected `ref.current` values to `null`. After updating the DOM, React immediately sets them to the corresponding DOM nodes.

Usually, you will access refs from event handlers. If you want to do something with a ref, but there is no particular event to do it in, you might need an Effect. We will discuss effects on the next pages.

<DeepDive>

Flushing state updates synchronously with flushSync {/*flushing-state-updates-synchronously-with-flush-sync*/}

Consider code like this, which adds a new todo and scrolls the screen down to the last child of the list. Notice how, for some reason, it always scrolls to the todo that was *just before* the last added one:

<Sandpack>

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

export default function TodoList() {
const listRef = useRef(null);
const [text, setText] = useState('');
const [todos, setTodos] = useState(
initialTodos
);

function handleAdd() {
```

```
const newTodo = { id: nextId++, text: text };
setText('');
setTodos([ ...todos, newTodo]);
listRef.current.lastChild.scrollIntoView({
behavior: 'smooth',
block: 'nearest'
});
}

return (
<>
<button onClick={handleAdd}>
Add
</button>
<input
value={text}
onChange={e => setText(e.target.value)}
/>
<ul ref={listRef}>
{todos.map(todo => (
<li key={todo.id}>{todo.text}</li>
))}
</ul>
</>
);
}

let nextId = 0;
let initialTodos = [];
for (let i = 0; i < 20; i++) {
initialTodos.push({
id: nextId++,
text: 'Todo #' + (i + 1)
});
}
```

</Sandpack>

The issue is with these two lines:

```js
setTodos([ ...todos, newTodo]);
listRef.current.lastChild.scrollIntoView();
```

In React, [state updates are queued.](/learn/queueing-a-series-of-state-updates) Usually, this is what you want. However, here it causes a problem because `setTodos` does not immediately update the DOM. So the time you scroll the list to its last element, the todo has not yet been added. This is why scrolling always "lags behind" by one item.

To fix this issue, you can force React to update ("flush") the DOM synchronously. To do this, import `flushSync` from `react-dom` and **wrap the state update** into a `flushSync` call:

```js
flushSync(() => {
setTodos([ ...todos, newTodo]);
});
listRef.current.lastChild.scrollIntoView();
```

This will instruct React to update the DOM synchronously right after the code wrapped in `flushSync` executes. As a result, the last todo will already be in the DOM by the time you try to scroll to it:

<Sandpack>

```js
import { useState, useRef } from 'react';
import { flushSync } from 'react-dom';

export default function TodoList() {
const listRef = useRef(null);
const [text, setText] = useState('');
const [todos, setTodos] = useState(
initialTodos
);

function handleAdd() {
const newTodo = { id: nextId++, text: text };
flushSync(() => {
setText('');
```

```
setTodos([ ...todos, newTodo]);
});
listRef.current.lastChild.scrollIntoView({
behavior: 'smooth',
block: 'nearest'
});
}

return (
<>
<button onClick={handleAdd}>
Add
</button>
<input
value={text}
onChange={e => setText(e.target.value)}
/>
<ul ref={listRef}>
{todos.map(todo => (
<li key={todo.id}>{todo.text}</li>
))}
</ul>
</>
);
}

let nextId = 0;
let initialTodos = [];
for (let i = 0; i < 20; i++) {
initialTodos.push({
id: nextId++,
text: 'Todo #' + (i + 1)
});
}
```

</Sandpack>

</DeepDive>

Best practices for DOM manipulation with refs {/*best-practices-for-dom-manipulation-with-refs*/}

Refs are an escape hatch. You should only use them when you have to "step outside React". Common examples of this include managing focus, scroll position, or calling browser APIs that React does not expose.

If you stick to non-destructive actions like focusing and scrolling, you shouldn't encounter any problems. However, if you try to **modify** the DOM manually, you can risk conflicting with the changes React is making.

To illustrate this problem, this example includes a welcome message and two buttons. The first button toggles its presence using [conditional rendering](/learn/conditional-rendering) and [state](/learn/state-a-components-memory), as you would usually do in React. The second button uses the [`remove()` DOM API](https://developer.mozilla.org/en-US/docs/Web/API/Element/remove) to forcefully remove it from the DOM outside of React's control.

Try pressing "Toggle with setState" a few times. The message should disappear and appear again. Then press "Remove from the DOM". This will forcefully remove it. Finally, press "Toggle with setState":

<Sandpack>

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

export default function Counter() {
const [show, setShow] = useState(true);
const ref = useRef(null);

return (
<div>
<button
onClick={() => {
setShow(!show);
}}>
Toggle with setState
</button>
<button
onClick={() => {
ref.current.remove();
}}>
Remove from the DOM
</button>
```

{show && Hello world }

);

}

```

```css

p,

button {

display: block;

margin: 10px;

}

```

</Sandpack>

After you've manually removed the DOM element, trying to use `setState` to show it again will lead to a crash. This is because you've changed the DOM, and React doesn't know how to continue managing it correctly.

Avoid changing DOM nodes managed by React. Modifying, adding children to, or removing children from elements that are managed by React can lead to inconsistent visual results or crashes like above.

However, this doesn't mean that you can't do it at all. It requires caution. **You can safely modify parts of the DOM that React has _no reason_ to update.** For example, if some `<div>` is always empty in the JSX, React won't have a reason to touch its children list. Therefore, it is safe to manually add or remove elements there.

<Recap>

- Refs are a generic concept, but most often you'll use them to hold DOM elements.

- You instruct React to put a DOM node into `myRef.current` by passing `<div ref={myRef}>`.

- Usually, you will use refs for non-destructive actions like focusing, scrolling, or measuring DOM elements.

- A component doesn't expose its DOM nodes by default. You can opt into exposing a DOM node by using `forwardRef` and passing the second `ref` argument down to a specific node.

- Avoid changing DOM nodes managed by React.

- If you do modify DOM nodes managed by React, modify parts that React has no reason to update.

</Recap>

<Challenges>

Play and pause the video {/*play-and-pause-the-video*/}
```

In this example, the button toggles a state variable to switch between a playing and a paused state. However, in order to actually play or pause the video, toggling state is not enough. You also need to call [`play()`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement/play) and [`pause()`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement/pause) on the DOM element for the `<video>`. Add a ref to it, and make the button work.

<Sandpack>

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

export default function VideoPlayer() {
const [isPlaying, setIsPlaying] = useState(false);

function handleClick() {
const nextIsPlaying = !isPlaying;
setIsPlaying(nextIsPlaying);
}

return (
<>
<button onClick={handleClick}>
{isPlaying ? 'Pause' : 'Play'}
</button>
<video width="250">
<source
src="https://interactive-examples.mdn.mozilla.net/media/cc0-videos/flower.mp4"
type="video/mp4"
/>
</video>
</>
)
}
```

```css
button { display: block; margin-bottom: 20px; }
```

</Sandpack>

For an extra challenge, keep the "Play" button in sync with whether the video is playing even if the user right-clicks the video and plays it using the built-in browser media controls. You might want to listen to `onPlay` and `onPause` on the video to do that.

<Solution>

Declare a ref and put it on the `<video>` element. Then call `ref.current.play()` and `ref.current.pause()` in the event handler depending on the next state.

<Sandpack>

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

export default function VideoPlayer() {
const [isPlaying, setIsPlaying] = useState(false);
const ref = useRef(null);

function handleClick() {
const nextIsPlaying = !isPlaying;
setIsPlaying(nextIsPlaying);

if (nextIsPlaying) {
ref.current.play();
} else {
ref.current.pause();
}
}

return (
<>
<button onClick={handleClick}>
{isPlaying ? 'Pause' : 'Play'}
</button>
<video
width="250"
ref={ref}
onPlay={() => setIsPlaying(true)}
onPause={() => setIsPlaying(false)}
>
<source
src="https://interactive-examples.mdn.mozilla.net/media/cc0-videos/flower.mp4"
```

type="video/mp4"

/>

</video>

</>

)

}

```

```css

button { display: block; margin-bottom: 20px; }

```

</Sandpack>

In order to handle the built-in browser controls, you can add `onPlay` and `onPause` handlers to the `<video>` element and call `setIsPlaying` from them. This way, if the user plays the video using the browser controls, the state will adjust accordingly.

</Solution>

Focus the search field {/*focus-the-search-field*/}

Make it so that clicking the "Search" button puts focus into the field.

<Sandpack>

```js

export default function Page() {

return (

<>

<nav>

<button>Search</button>

</nav>

<input

placeholder="Looking for something?"

/>

</>

);

}

```

```css

````
button { display: block; margin-bottom: 10px; }
```

</Sandpack>

<Solution>

Add a ref to the input, and call `focus()` on the DOM node to focus it:

<Sandpack>

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

export default function Page() {
const inputRef = useRef(null);
return (
<>
<nav>
<button onClick={() => {
inputRef.current.focus();
}}>
Search
</button>
</nav>
<input
ref={inputRef}
placeholder="Looking for something?"
/>
</>
);
}
```

```css
button { display: block; margin-bottom: 10px; }
```

</Sandpack>

</Solution>

Scrolling an image carousel {/*scrolling-an-image-carousel*/}

This image carousel has a "Next" button that switches the active image. Make the gallery scroll horizontally to the active image on click. You will want to call [`scrollIntoView()`](https://developer.mozilla.org/en-US/docs/Web/API/Element/scrollIntoView) on the DOM node of the active image:

```js
node.scrollIntoView({
behavior: 'smooth',
block: 'nearest',
inline: 'center'
});
```

<Hint>

You don't need to have a ref to every image for this exercise. It should be enough to have a ref to the currently active image, or to the list itself. Use `flushSync` to ensure the DOM is updated *before* you scroll.

</Hint>

<Sandpack>

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

export default function CatFriends() {
const [index, setIndex] = useState(0);
return (
<>
<nav>
<button onClick={() => {
if (index < catList.length - 1) {
setIndex(index + 1);
} else {
setIndex(0);
}
}}>
Next
</button>
```

```
</nav>
<div>
<ul>
{catList.map((cat, i) => (
<li key={cat.id}>
<img
className={
index === i ?
'active' :
''
}
src={cat.imageUrl}
alt={'Cat #' + cat.id}
/>
</li>
))}
</ul>
</div>
</>
);
}

const catList = [];
for (let i = 0; i < 10; i++) {
catList.push({
id: i,
imageUrl: 'https://placekitten.com/250/200?image=' + i
});
}
```

```css
div {
width: 100%;
overflow: hidden;
}
```

nav {
text-align: center;
}

button {
margin: .25rem;
}

ul,
li {
list-style: none;
white-space: nowrap;
}

li {
display: inline;
padding: 0.5rem;
}

img {
padding: 10px;
margin: -10px;
transition: background 0.2s linear;
}

.active {
background: rgba(0, 100, 150, 0.4);
}
```

</Sandpack>

<Solution>

You can declare a `selectedRef`, and then pass it conditionally only to the current image:

```js
<li ref={index === i ? selectedRef : null}>
```

When `index === i`, meaning that the image is the selected one, the `<li>` will receive the `selectedRef`. React will make sure that `selectedRef.current` always points at the correct DOM node.
```

Note that the `flushSync` call is necessary to force React to update the DOM before the scroll. Otherwise, `selectedRef.current` would always point at the previously selected item.

<Sandpack>

```js
import { useRef, useState } from 'react';
import { flushSync } from 'react-dom';

export default function CatFriends() {
const selectedRef = useRef(null);
const [index, setIndex] = useState(0);

return (
<>
<nav>
<button onClick={() => {
flushSync(() => {
if (index < catList.length - 1) {
setIndex(index + 1);
} else {
setIndex(0);
}
});
selectedRef.current.scrollIntoView({
behavior: 'smooth',
block: 'nearest',
inline: 'center'
});
}}>
Next
</button>
</nav>
<div>
<ul>
{catList.map((cat, i) => (
<li
key={cat.id}
```

```
ref={index === i ?
selectedRef :
null
}
>
<img
className={
index === i ?
'active'
: ''
}
src={cat.imageUrl}
alt={'Cat #' + cat.id}
/>
</li>
))}
</ul>
</div>
</>
);
}

const catList = [];
for (let i = 0; i < 10; i++) {
catList.push({
id: i,
imageUrl: 'https://placekitten.com/250/200?image=' + i
});
}
```

```css
div {
width: 100%;
overflow: hidden;
}
```

```
nav {
text-align: center;
}

button {
margin: .25rem;
}

ul,
li {
list-style: none;
white-space: nowrap;
}

li {
display: inline;
padding: 0.5rem;
}

img {
padding: 10px;
margin: -10px;
transition: background 0.2s linear;
}

.active {
background: rgba(0, 100, 150, 0.4);
}
```

</Sandpack>

</Solution>

Focus the search field with separate components
{/*focus-the-search-field-with-separate-components*/}

Make it so that clicking the "Search" button puts focus into the field. Note that each component is defined in a separate file and shouldn't be moved out of it. How do you connect them together?

<Hint>

You'll need `forwardRef` to opt into exposing a DOM node from your own component like `SearchInput`.

</Hint>

<Sandpack>

```js App.js
import SearchButton from './SearchButton.js';
import SearchInput from './SearchInput.js';

export default function Page() {
return (
<>
<nav>
<SearchButton />
</nav>
<SearchInput />
</>
);
}
```

```js SearchButton.js
export default function SearchButton() {
return (
<button>
Search
</button>
);
}
```

```js SearchInput.js
export default function SearchInput() {
return (
<input
placeholder="Looking for something?"
/>
);
}
```

```css
button { display: block; margin-bottom: 10px; }
```

</Sandpack>

<Solution>

You'll need to add an `onClick` prop to the `SearchButton`, and make the `SearchButton` pass it down to the browser `<button>`. You'll also pass a ref down to `<SearchInput>`, which will forward it to the real `<input>` and populate it. Finally, in the click handler, you'll call `focus` on the DOM node stored inside that ref.

<Sandpack>

```js App.js
import { useRef } from 'react';
import SearchButton from './SearchButton.js';
import SearchInput from './SearchInput.js';

export default function Page() {
const inputRef = useRef(null);
return (
<>
<nav>
<SearchButton onClick={() => {
inputRef.current.focus();
}} />
</nav>
<SearchInput ref={inputRef} />
</>
);
}
```

```js SearchButton.js
export default function SearchButton({ onClick }) {
return (
<button onClick={onClick}>
Search
</button>
```

);
}
```

```js SearchInput.js
import { forwardRef } from 'react';

export default forwardRef(
function SearchInput(props, ref) {
return (
<input
ref={ref}
placeholder="Looking for something?"
/>
);
}
);
```

```css
button { display: block; margin-bottom: 10px; }
```

</Sandpack>

</Solution>

</Challenges>

---
title: Describing the UI

<Intro>

React is a JavaScript library for rendering user interfaces (UI). UI is built from small units like buttons, text, and images. React lets you combine them into reusable, nestable *components.* From web sites to phone apps, everything on the screen can be broken down into components. In this chapter, you'll learn to create, customize, and conditionally display React components.

</Intro>

<YouWillLearn isChapter={true}>

* [How to write your first React component](/learn/your-first-component)
```

* [When and how to create multi-component files](/learn/importing-and-exporting-components)

* [How to add markup to JavaScript with JSX](/learn/writing-markup-with-jsx)

* [How to use curly braces with JSX to access JavaScript functionality from your components](/learn/javascript-in-jsx-with-curly-braces)

* [How to configure components with props](/learn/passing-props-to-a-component)

* [How to conditionally render components](/learn/conditional-rendering)

* [How to render multiple components at a time](/learn/rendering-lists)

* [How to avoid confusing bugs by keeping components pure](/learn/keeping-components-pure)

</YouWillLearn>

Your first component {/*your-first-component*/}

React applications are built from isolated pieces of UI called *components*. A React component is a JavaScript function that you can sprinkle with markup. Components can be as small as a button, or as large as an entire page. Here is a `Gallery` component rendering three `Profile` components:

<Sandpack>

```js
function Profile() {
return (
<img
src="https://i.imgur.com/MK3eW3As.jpg"
alt="Katherine Johnson"
/>
);
}

export default function Gallery() {
return (
<section>
<h1>Amazing scientists</h1>
<Profile />
<Profile />
<Profile />
</section>
);
}
```

```css
img { margin: 0 10px 10px 0; height: 90px; }
```

</Sandpack>

<LearnMore path="/learn/your-first-component">

Read **[Your First Component](/learn/your-first-component)** to learn how to declare and use React components.

</LearnMore>

Importing and exporting components {/*importing-and-exporting-components*/}

You can declare many components in one file, but large files can get difficult to navigate. To solve this, you can *export* a component into its own file, and then *import* that component from another file:

<Sandpack>

```js App.js hidden
import Gallery from './Gallery.js';

export default function App() {
return (
<Gallery />
);
}
```

```js Gallery.js active
import Profile from './Profile.js';

export default function Gallery() {
return (
<section>
<h1>Amazing scientists</h1>
<Profile />
<Profile />
<Profile />
</section>
);
}
```

```

```js Profile.js
export default function Profile() {
return (
<img
src="https://i.imgur.com/QIrZWGIs.jpg"
alt="Alan L. Hart"
/>
);
}
```

```css
img { margin: 0 10px 10px 0; }
```

</Sandpack>

<LearnMore path="/learn/importing-and-exporting-components">

Read **[Importing and Exporting Components](/learn/importing-and-exporting-components)** to learn how to split components into their own files.

</LearnMore>

Writing markup with JSX {/*writing-markup-with-jsx*/}

Each React component is a JavaScript function that may contain some markup that React renders into the browser. React components use a syntax extension called JSX to represent that markup. JSX looks a lot like HTML, but it is a bit stricter and can display dynamic information.

If we paste existing HTML markup into a React component, it won't always work:

<Sandpack>

```js
export default function TodoList() {
return (
// This doesn't quite work!
<h1>Hedy Lamarr's Todos</h1>
<img
src="https://i.imgur.com/yXOvdOSs.jpg"
alt="Hedy Lamarr"
```
```

````
class="photo"
>
<ul>
<li>Invent new traffic lights
<li>Rehearse a movie scene
<li>Improve spectrum technology
</ul>
);
}
```

```css
img { height: 90px; }
```

</Sandpack>

If you have existing HTML like this, you can fix it using a [converter](https://transform.tools/html-to-jsx):

<Sandpack>

```js
export default function TodoList() {
return (
<>
<h1>Hedy Lamarr's Todos</h1>
<img
src="https://i.imgur.com/yXOvdOSs.jpg"
alt="Hedy Lamarr"
className="photo"
/>
<ul>
<li>Invent new traffic lights</li>
<li>Rehearse a movie scene</li>
<li>Improve spectrum technology</li>
</ul>
</>
);
}
````

```

```css
img { height: 90px; }
```

</Sandpack>

<LearnMore path="/learn/writing-markup-with-jsx">

Read **[Writing Markup with JSX](/learn/writing-markup-with-jsx)** to learn how to write valid JSX.

</LearnMore>

JavaScript in JSX with curly braces {/*javascript-in-jsx-with-curly-braces*/}

JSX lets you write HTML-like markup inside a JavaScript file, keeping rendering logic and content in the same place. Sometimes you will want to add a little JavaScript logic or reference a dynamic property inside that markup. In this situation, you can use curly braces in your JSX to "open a window" to JavaScript:

<Sandpack>

```js
const person = {
name: 'Gregorio Y. Zara',
theme: {
backgroundColor: 'black',
color: 'pink'
}
};

export default function TodoList() {
return (

<h1>{person.name}'s Todos</h1>
<img
className="avatar"
src="https://i.imgur.com/7vQD0fPs.jpg"
alt="Gregorio Y. Zara"
/>
<ul>
<li>Improve the videophone</li>

<li>Prepare aeronautics lectures</li>

<li>Work on the alcohol-fuelled engine</li>

</ul>

);

}

```

```css

body { padding: 0; margin: 0 }

body > div > div { padding: 20px; }

.avatar { border-radius: 50%; height: 90px; }

```

</Sandpack>

<LearnMore path="/learn/javascript-in-jsx-with-curly-braces">

Read **[JavaScript in JSX with Curly Braces](/learn/javascript-in-jsx-with-curly-braces)** to learn how to access JavaScript data from JSX.

</LearnMore>

Passing props to a component {/*passing-props-to-a-component*/}

React components use *props* to communicate with each other. Every parent component can pass some information to its child components by giving them props. Props might remind you of HTML attributes, but you can pass any JavaScript value through them, including objects, arrays, functions, and even JSX!

<Sandpack>

```js

import { getImageUrl } from './utils.js'

export default function Profile() {

return (

<Card>

<Avatar

size={100}

person={{

name: 'Katsuko Saruhashi',

imageId: 'YfeOqp2'

```
}}
/>
</Card>
);
}

function Avatar({ person, size }) {
return (
<img
className="avatar"
src={getImageUrl(person)}
alt={person.name}
width={size}
height={size}
/>
);
}

function Card({ children }) {
return (
<div className="card">
{children}
</div>
);
}
```

```js utils.js
export function getImageUrl(person, size = 's') {
return (
'https://i.imgur.com/' +
person.imageId +
size +
'.jpg'
);
}
```

```css
.card {
width: fit-content;
margin: 5px;
padding: 5px;
font-size: 20px;
text-align: center;
border: 1px solid #aaa;
border-radius: 20px;
background: #fff;
}
.avatar {
margin: 20px;
border-radius: 50%;
}
```

</Sandpack>

<LearnMore path="/learn/passing-props-to-a-component">

Read **[Passing Props to a Component](/learn/passing-props-to-a-component)** to learn how to pass and read props.

</LearnMore>

Conditional rendering {/*conditional-rendering*/}

Your components will often need to display different things depending on different conditions. In React, you can conditionally render JSX using JavaScript syntax like `if` statements, `&&`, and `? :` operators.

In this example, the JavaScript `&&` operator is used to conditionally render a checkmark:

<Sandpack>

```js
function Item({ name, isPacked }) {
return (
<li className="item">
{name} {isPacked && '✔'}
</li>
);
```

```
}

export default function PackingList() {
return (
<section>
<h1>Sally Ride's Packing List</h1>
<ul>
<Item
isPacked={true}
name="Space suit"
/>
<Item
isPacked={true}
name="Helmet with a golden leaf"
/>
<Item
isPacked={false}
name="Photo of Tam"
/>
</ul>
</section>
);
}
```

</Sandpack>

<LearnMore path="/learn/conditional-rendering">

Read **[Conditional Rendering](/learn/conditional-rendering)** to learn the different ways to render content conditionally.

</LearnMore>

Rendering lists {/*rendering-lists*/}

You will often want to display multiple similar components from a collection of data. You can use JavaScript's `filter()` and `map()` with React to filter and transform your array of data into an array of components.

For each array item, you will need to specify a `key`. Usually, you will want to use an ID from the database as a `key`. Keys let React keep track of each item's place in the list even if the list changes.

<Sandpack>

```js App.js
import { people } from './data.js';
import { getImageUrl } from './utils.js';

export default function List() {
const listItems = people.map(person =>
<li key={person.id}>
<img
src={getImageUrl(person)}
alt={person.name}
/>
<p>
<b>{person.name}:</b>
{' ' + person.profession + ' '}
known for {person.accomplishment}
</p>
</li>
);
return (
<article>
<h1>Scientists</h1>
<ul>{listItems}</ul>
</article>
);
}
```

```js data.js
export const people = [{
id: 0,
name: 'Creola Katherine Johnson',
profession: 'mathematician',
accomplishment: 'spaceflight calculations',
imageId: 'MK3eW3A'
}, {
```

id: 1,
name: 'Mario José Molina-Pasquel Henríquez',
profession: 'chemist',
accomplishment: 'discovery of Arctic ozone hole',
imageId: 'mynHUSa'
}, {
id: 2,
name: 'Mohammad Abdus Salam',
profession: 'physicist',
accomplishment: 'electromagnetism theory',
imageId: 'bE7W1ji'
}, {
id: 3,
name: 'Percy Lavon Julian',
profession: 'chemist',
accomplishment: 'pioneering cortisone drugs, steroids and birth control pills',
imageId: 'IOjWm71'
}, {
id: 4,
name: 'Subrahmanyan Chandrasekhar',
profession: 'astrophysicist',
accomplishment: 'white dwarf star mass calculations',
imageId: 'lrWQx8l'
}];
```

```js utils.js
export function getImageUrl(person) {
return (
'https://i.imgur.com/' +
person.imageId +
's.jpg'
);
}
```

```css

ul { list-style-type: none; padding: 0px 10px; }

li {

margin-bottom: 10px;

display: grid;

grid-template-columns: 1fr 1fr;

align-items: center;

}

img { width: 100px; height: 100px; border-radius: 50%; }

h1 { font-size: 22px; }

h2 { font-size: 20px; }

```

</Sandpack>

<LearnMore path="/learn/rendering-lists">

Read **[Rendering Lists](/learn/rendering-lists)** to learn how to render a list of components, and how to choose a key.

</LearnMore>

Keeping components pure {/*keeping-components-pure*/}

Some JavaScript functions are *pure.* A pure function:

* **Minds its own business.** It does not change any objects or variables that existed before it was called.

* **Same inputs, same output.** Given the same inputs, a pure function should always return the same result.

By strictly only writing your components as pure functions, you can avoid an entire class of baffling bugs and unpredictable behavior as your codebase grows. Here is an example of an impure component:

<Sandpack>

```js

let guest = 0;

function Cup() {

// Bad: changing a preexisting variable!

guest = guest + 1;

return <h2>Tea cup for guest #{guest}</h2>;

}

```
export default function TeaSet() {
return (
<>
<Cup />
<Cup />
<Cup />
</>
);
}
```

</Sandpack>

You can make this component pure by passing a prop instead of modifying a preexisting variable:

<Sandpack>

```js
function Cup({ guest }) {
return <h2>Tea cup for guest #{guest}</h2>;
}

export default function TeaSet() {
return (
<>
<Cup guest={1} />
<Cup guest={2} />
<Cup guest={3} />
</>
);
}
```

</Sandpack>

<LearnMore path="/learn/keeping-components-pure">

Read **[Keeping Components Pure](/learn/keeping-components-pure)** to learn how to write components as pure, predictable functions.

</LearnMore>

What's next? {/*whats-next*/}

Head over to [Your First Component](/learn/your-first-component) to start reading this chapter page by page!

Or, if you're already familiar with these topics, why not read about [Adding Interactivity](/learn/adding-interactivity)?

---

title: 'You Might Not Need an Effect'

<Intro>

Effects are an escape hatch from the React paradigm. They let you "step outside" of React and synchronize your components with some external system like a non-React widget, network, or the browser DOM. If there is no external system involved (for example, if you want to update a component's state when some props or state change), you shouldn't need an Effect. Removing unnecessary Effects will make your code easier to follow, faster to run, and less error-prone.

</Intro>

<YouWillLearn>

* Why and how to remove unnecessary Effects from your components
* How to cache expensive computations without Effects
* How to reset and adjust component state without Effects
* How to share logic between event handlers
* Which logic should be moved to event handlers
* How to notify parent components about changes

</YouWillLearn>

How to remove unnecessary Effects {/*how-to-remove-unnecessary-effects*/}

There are two common cases in which you don't need Effects:

* **You don't need Effects to transform data for rendering.** For example, let's say you want to filter a list before displaying it. You might feel tempted to write an Effect that updates a state variable when the list changes. However, this is inefficient. When you update the state, React will first call your component functions to calculate what should be on the screen. Then React will ["commit"](/learn/render-and-commit) these changes to the DOM, updating the screen. Then React will run your Effects. If your Effect *also* immediately updates the state, this restarts the whole process from scratch! To avoid the unnecessary render passes, transform all the data at the top level of your components. That code will automatically re-run whenever your props or state change.
* **You don't need Effects to handle user events.** For example, let's say you want to send an `/api/buy` POST request and show a notification when the user buys a product. In the Buy button click event handler, you know exactly what happened. By the time an Effect runs, you don't know *what* the user did (for example, which button was clicked). This is why you'll usually handle user events in the

corresponding event handlers.

You *do* need Effects to [synchronize](/learn/synchronizing-with-effects#what-are-effects-and-how-are-they-different-from-events) with external systems. For example, you can write an Effect that keeps a jQuery widget synchronized with the React state. You can also fetch data with Effects: for example, you can synchronize the search results with the current search query. Keep in mind that modern [frameworks](/learn/start-a-new-react-project#production-grade-react-frameworks) provide more efficient built-in data fetching mechanisms than writing Effects directly in your components.

To help you gain the right intuition, let's look at some common concrete examples!

Updating state based on props or state {/*updating-state-based-on-props-or-state*/}

Suppose you have a component with two state variables: `firstName` and `lastName`. You want to calculate a `fullName` from them by concatenating them. Moreover, you'd like `fullName` to update whenever `firstName` or `lastName` change. Your first instinct might be to add a `fullName` state variable and update it in an Effect:

```js {5-9}
function Form() {
const [firstName, setFirstName] = useState('Taylor');
const [lastName, setLastName] = useState('Swift');

// ■ Avoid: redundant state and unnecessary Effect
const [fullName, setFullName] = useState('');
useEffect(() => {
setFullName(firstName + ' ' + lastName);
}, [firstName, lastName]);
// ...
}
```

This is more complicated than necessary. It is inefficient too: it does an entire render pass with a stale value for `fullName`, then immediately re-renders with the updated value. Remove the state variable and the Effect:

```js {4-5}
function Form() {
const [firstName, setFirstName] = useState('Taylor');
const [lastName, setLastName] = useState('Swift');
// ■ Good: calculated during rendering
const fullName = firstName + ' ' + lastName;
// ...
}
```

```

**When something can be calculated from the existing props or state, [don't put it in state.](/learn/choosing-the-state-structure#avoid-redundant-state) Instead, calculate it during rendering.** This makes your code faster (you avoid the extra "cascading" updates), simpler (you remove some code), and less error-prone (you avoid bugs caused by different state variables getting out of sync with each other). If this approach feels new to you, [Thinking in React](/learn/thinking-in-react#step-3-find-the-minimal-but-complete-representation-of-ui-state) explains what should go into state.

### Caching expensive calculations {/*caching-expensive-calculations*/}

This component computes `visibleTodos` by taking the `todos` it receives by props and filtering them according to the `filter` prop. You might feel tempted to store the result in state and update it from an Effect:

```js {4-8}
function TodoList({ todos, filter }) {
const [newTodo, setNewTodo] = useState('');

// ■ Avoid: redundant state and unnecessary Effect
const [visibleTodos, setVisibleTodos] = useState([]);
useEffect(() => {
setVisibleTodos(getFilteredTodos(todos, filter));
}, [todos, filter]);

// ...
}
```

Like in the earlier example, this is both unnecessary and inefficient. First, remove the state and the Effect:

```js {3-4}
function TodoList({ todos, filter }) {
const [newTodo, setNewTodo] = useState('');
// ■ This is fine if getFilteredTodos() is not slow.
const visibleTodos = getFilteredTodos(todos, filter);
// ...
}
```

Usually, this code is fine! But maybe `getFilteredTodos()` is slow or you have a lot of `todos`. In that case you don't want to recalculate `getFilteredTodos()` if some unrelated state variable like `newTodo` has changed.
```

You can cache (or ["memoize"](https://en.wikipedia.org/wiki/Memoization)) an expensive calculation by wrapping it in a [`useMemo`](/reference/react/useMemo) Hook:

```js {5-8}
import { useMemo, useState } from 'react';

function TodoList({ todos, filter }) {
const [newTodo, setNewTodo] = useState('');
const visibleTodos = useMemo(() => {
// ■ Does not re-run unless todos or filter change
return getFilteredTodos(todos, filter);
}, [todos, filter]);
// ...
}
```

Or, written as a single line:

```js {5-6}
import { useMemo, useState } from 'react';

function TodoList({ todos, filter }) {
const [newTodo, setNewTodo] = useState('');
// ■ Does not re-run getFilteredTodos() unless todos or filter change
const visibleTodos = useMemo(() => getFilteredTodos(todos, filter), [todos, filter]);
// ...
}
```

This tells React that you don't want the inner function to re-run unless either `todos` or `filter` have changed. React will remember the return value of `getFilteredTodos()` during the initial render. During the next renders, it will check if `todos` or `filter` are different. If they're the same as last time, `useMemo` will return the last result it has stored. But if they are different, React will call the inner function again (and store its result).

The function you wrap in [`useMemo`](/reference/react/useMemo) runs during rendering, so this only works for [pure calculations.](/learn/keeping-components-pure)

<DeepDive>

How to tell if a calculation is expensive? {/*how-to-tell-if-a-calculation-is-expensive*/}

In general, unless you're creating or looping over thousands of objects, it's probably not expensive. If you want to get more confidence, you can add a console log to measure the time spent in a piece of code:

```js {1,3}
console.time('filter array');
const visibleTodos = getFilteredTodos(todos, filter);
console.timeEnd('filter array');
```

Perform the interaction you're measuring (for example, typing into the input). You will then see logs like `filter array: 0.15ms` in your console. If the overall logged time adds up to a significant amount (say, `1ms` or more), it might make sense to memoize that calculation. As an experiment, you can then wrap the calculation in `useMemo` to verify whether the total logged time has decreased for that interaction or not:

```js
console.time('filter array');
const visibleTodos = useMemo(() => {
return getFilteredTodos(todos, filter); // Skipped if todos and filter haven't changed
}, [todos, filter]);
console.timeEnd('filter array');
```

`useMemo` won't make the *first* render faster. It only helps you skip unnecessary work on updates.

Keep in mind that your machine is probably faster than your users' so it's a good idea to test the performance with an artificial slowdown. For example, Chrome offers a [CPU Throttling](https://developer.chrome.com/blog/new-in-devtools-61/#throttling) option for this.

Also note that measuring performance in development will not give you the most accurate results. (For example, when [Strict Mode](/reference/react/StrictMode) is on, you will see each component render twice rather than once.) To get the most accurate timings, build your app for production and test it on a device like your users have.

</DeepDive>

Resetting all state when a prop changes {/*resetting-all-state-when-a-prop-changes*/}

This `ProfilePage` component receives a `userId` prop. The page contains a comment input, and you use a `comment` state variable to hold its value. One day, you notice a problem: when you navigate from one profile to another, the `comment` state does not get reset. As a result, it's easy to accidentally post a comment on a wrong user's profile. To fix the issue, you want to clear out the `comment` state variable whenever the `userId` changes:

```js {4-7}
export default function ProfilePage({ userId }) {
const [comment, setComment] = useState('');

// ■ Avoid: Resetting state on prop change in an Effect
```

```
useEffect(() => {
setComment('');
}, [userId]);
// ...
}
```

This is inefficient because `ProfilePage` and its children will first render with the stale value, and then render again. It is also complicated because you'd need to do this in *every* component that has some state inside `ProfilePage`. For example, if the comment UI is nested, you'd want to clear out nested comment state too.

Instead, you can tell React that each user's profile is conceptually a _different_ profile by giving it an explicit key. Split your component in two and pass a `key` attribute from the outer component to the inner one:

```js {5,11-12}
export default function ProfilePage({ userId }) {
return (
<Profile
userId={userId}
key={userId}
/>
);
}

function Profile({ userId }) {
// ■ This and any other state below will reset on key change automatically
const [comment, setComment] = useState('');
// ...
}
```

Normally, React preserves the state when the same component is rendered in the same spot. **By passing `userId` as a `key` to the `Profile` component, you're asking React to treat two `Profile` components with different `userId` as two different components that should not share any state.** Whenever the key (which you've set to `userId`) changes, React will recreate the DOM and [reset the state](/learn/preserving-and-resetting-state#option-2-resetting-state-with-a-key) of the `Profile` component and all of its children. Now the `comment` field will clear out automatically when navigating between profiles.

Note that in this example, only the outer `ProfilePage` component is exported and visible to other files in the project. Components rendering `ProfilePage` don't need to pass the key to it: they pass `userId` as a regular prop. The fact `ProfilePage` passes it as a `key` to the inner `Profile` component is an

implementation detail.

Adjusting some state when a prop changes {/*adjusting-some-state-when-a-prop-changes*/}

Sometimes, you might want to reset or adjust a part of the state on a prop change, but not all of it.

This `List` component receives a list of `items` as a prop, and maintains the selected item in the `selection` state variable. You want to reset the `selection` to `null` whenever the `items` prop receives a different array:

```js {5-8}
function List({ items }) {
const [isReverse, setIsReverse] = useState(false);
const [selection, setSelection] = useState(null);

// ■ Avoid: Adjusting state on prop change in an Effect
useEffect(() => {
setSelection(null);
}, [items]);
// ...
}
```

This, too, is not ideal. Every time the `items` change, the `List` and its child components will render with a stale `selection` value at first. Then React will update the DOM and run the Effects. Finally, the `setSelection(null)` call will cause another re-render of the `List` and its child components, restarting this whole process again.

Start by deleting the Effect. Instead, adjust the state directly during rendering:

```js {5-11}
function List({ items }) {
const [isReverse, setIsReverse] = useState(false);
const [selection, setSelection] = useState(null);

// Better: Adjust the state while rendering
const [prevItems, setPrevItems] = useState(items);
if (items !== prevItems) {
setPrevItems(items);
setSelection(null);
}
// ...
}
```

```

[Storing information from previous renders](/reference/react/useState#storing-information-from-previous-renders) like this can be hard to understand, but it’s better than updating the same state in an Effect. In the above example, `setSelection` is called directly during a render. React will re-render the `List` *immediately* after it exits with a `return` statement. React has not rendered the `List` children or updated the DOM yet, so this lets the `List` children skip rendering the stale `selection` value.

When you update a component during rendering, React throws away the returned JSX and immediately retries rendering. To avoid very slow cascading retries, React only lets you update the *same* component's state during a render. If you update another component's state during a render, you'll see an error. A condition like `items !== prevItems` is necessary to avoid loops. You may adjust state like this, but any other side effects (like changing the DOM or setting timeouts) should stay in event handlers or Effects to [keep components pure.](/learn/keeping-components-pure)

**Although this pattern is more efficient than an Effect, most components shouldn't need it either.** No matter how you do it, adjusting state based on props or other state makes your data flow more difficult to understand and debug. Always check whether you can [reset all state with a key](#resetting-all-state-when-a-prop-changes) or [calculate everything during rendering](#updating-state-based-on-props-or-state) instead. For example, instead of storing (and resetting) the selected *item*, you can store the selected *item ID:*

```js {3-5}
function List({ items }) {
const [isReverse, setIsReverse] = useState(false);
const [selectedId, setSelectedId] = useState(null);
// ■ Best: Calculate everything during rendering
const selection = items.find(item => item.id === selectedId) ?? null;
// ...
}
```

Now there is no need to "adjust" the state at all. If the item with the selected ID is in the list, it remains selected. If it's not, the `selection` calculated during rendering will be `null` because no matching item was found. This behavior is different, but arguably better because most changes to `items` preserve the selection.

Sharing logic between event handlers {/*sharing-logic-between-event-handlers*/}

Let's say you have a product page with two buttons (Buy and Checkout) that both let you buy that product. You want to show a notification whenever the user puts the product in the cart. Calling `showNotification()` in both buttons' click handlers feels repetitive so you might be tempted to place this logic in an Effect:

```js {2-7}
function ProductPage({ product, addToCart }) {
// ■ Avoid: Event-specific logic inside an Effect
```
```

```
useEffect(() => {
if (product.isInCart) {
showNotification(`Added ${product.name} to the shopping cart!`);
}
}, [product]);

function handleBuyClick() {
addToCart(product);
}

function handleCheckoutClick() {
addToCart(product);
navigateTo('/checkout');
}
// ...
}
```

This Effect is unnecessary. It will also most likely cause bugs. For example, let's say that your app "remembers" the shopping cart between the page reloads. If you add a product to the cart once and refresh the page, the notification will appear again. It will keep appearing every time you refresh that product's page. This is because `product.isInCart` will already be `true` on the page load, so the Effect above will call `showNotification()`.

When you're not sure whether some code should be in an Effect or in an event handler, ask yourself *why* this code needs to run. Use Effects only for code that should run *because* the component was displayed to the user. In this example, the notification should appear because the user *pressed the button*, not because the page was displayed! Delete the Effect and put the shared logic into a function called from both event handlers:

```js {2-6,9,13}
function ProductPage({ product, addToCart }) {
// ■ Good: Event-specific logic is called from event handlers
function buyProduct() {
addToCart(product);
showNotification(`Added ${product.name} to the shopping cart!`);
}

function handleBuyClick() {
buyProduct();
}

function handleCheckoutClick() {
```

```
buyProduct();
navigateTo('/checkout');
}
// ...
}
```

This both removes the unnecessary Effect and fixes the bug.

Sending a POST request {/*sending-a-post-request*/}

This `Form` component sends two kinds of POST requests. It sends an analytics event when it mounts. When you fill in the form and click the Submit button, it will send a POST request to the `/api/register` endpoint:

```js {5-8,10-16}
function Form() {
const [firstName, setFirstName] = useState('');
const [lastName, setLastName] = useState('');

// ■ Good: This logic should run because the component was displayed
useEffect(() => {
post('/analytics/event', { eventName: 'visit_form' });
}, []);

// ■ Avoid: Event-specific logic inside an Effect
const [jsonToSubmit, setJsonToSubmit] = useState(null);
useEffect(() => {
if (jsonToSubmit !== null) {
post('/api/register', jsonToSubmit);
}
}, [jsonToSubmit]);

function handleSubmit(e) {
e.preventDefault();
setJsonToSubmit({ firstName, lastName });
}
// ...
}
```

Let's apply the same criteria as in the example before.

The analytics POST request should remain in an Effect. This is because the _reason_ to send the analytics event is that the form was displayed. (It would fire twice in development, but [see here](/learn/synchronizing-with-effects#sending-analytics) for how to deal with that.)

However, the `/api/register` POST request is not caused by the form being _displayed_. You only want to send the request at one specific moment in time: when the user presses the button. It should only ever happen _on that particular interaction_. Delete the second Effect and move that POST request into the event handler:

```js {12-13}
function Form() {
const [firstName, setFirstName] = useState('');
const [lastName, setLastName] = useState('');

// ■ Good: This logic runs because the component was displayed
useEffect(() => {
post('/analytics/event', { eventName: 'visit_form' });
}, []);

function handleSubmit(e) {
e.preventDefault();
// ■ Good: Event-specific logic is in the event handler
post('/api/register', { firstName, lastName });
}
// ...
}
```

When you choose whether to put some logic into an event handler or an Effect, the main question you need to answer is _what kind of logic_ it is from the user's perspective. If this logic is caused by a particular interaction, keep it in the event handler. If it's caused by the user _seeing_ the component on the screen, keep it in the Effect.

Chains of computations {/*chains-of-computations*/}

Sometimes you might feel tempted to chain Effects that each adjust a piece of state based on other state:

```js {7-29}
function Game() {
const [card, setCard] = useState(null);
const [goldCardCount, setGoldCardCount] = useState(0);
```

```
const [round, setRound] = useState(1);
const [isGameOver, setIsGameOver] = useState(false);

// ■ Avoid: Chains of Effects that adjust the state solely to trigger each other
useEffect(() => {
if (card !== null && card.gold) {
setGoldCardCount(c => c + 1);
}
}, [card]);

useEffect(() => {
if (goldCardCount > 3) {
setRound(r => r + 1)
setGoldCardCount(0);
}
}, [goldCardCount]);

useEffect(() => {
if (round > 5) {
setIsGameOver(true);
}
}, [round]);

useEffect(() => {
alert('Good game!');
}, [isGameOver]);

function handlePlaceCard(nextCard) {
if (isGameOver) {
throw Error('Game already ended.');
} else {
setCard(nextCard);
}
}

// ...
```

There are two problems with this code.

One problem is that it is very inefficient: the component (and its children) have to re-render between each `set` call in the chain. In the example above, in the worst case (`setCard` → render → `setGoldCardCount` → render → `setRound` → render → `setIsGameOver` → render) there are three unnecessary re-renders of the tree below.

Even if it weren't slow, as your code evolves, you will run into cases where the "chain" you wrote doesn't fit the new requirements. Imagine you are adding a way to step through the history of the game moves. You'd do it by updating each state variable to a value from the past. However, setting the `card` state to a value from the past would trigger the Effect chain again and change the data you're showing. Such code is often rigid and fragile.

In this case, it's better to calculate what you can during rendering, and adjust the state in the event handler:

```js {6-7,14-26}
function Game() {
const [card, setCard] = useState(null);
const [goldCardCount, setGoldCardCount] = useState(0);
const [round, setRound] = useState(1);

// ■ Calculate what you can during rendering
const isGameOver = round > 5;

function handlePlaceCard(nextCard) {
if (isGameOver) {
throw Error('Game already ended.');
}

// ■ Calculate all the next state in the event handler
setCard(nextCard);
if (nextCard.gold) {
if (goldCardCount <= 3) {
setGoldCardCount(goldCardCount + 1);
} else {
setGoldCardCount(0);
setRound(round + 1);
if (round === 5) {
alert('Good game!');
}
}
}
}
```

// ...
```

This is a lot more efficient. Also, if you implement a way to view game history, now you will be able to set each state variable to a move from the past without triggering the Effect chain that adjusts every other value. If you need to reuse logic between several event handlers, you can [extract a function](#sharing-logic-between-event-handlers) and call it from those handlers.

Remember that inside event handlers, [state behaves like a snapshot.](/learn/state-as-a-snapshot) For example, even after you call `setRound(round + 1)`, the `round` variable will reflect the value at the time the user clicked the button. If you need to use the next value for calculations, define it manually like `const nextRound = round + 1`.

In some cases, you *can't* calculate the next state directly in the event handler. For example, imagine a form with multiple dropdowns where the options of the next dropdown depend on the selected value of the previous dropdown. Then, a chain of Effects is appropriate because you are synchronizing with network.

### Initializing the application {/*initializing-the-application*/}

Some logic should only run once when the app loads.

You might be tempted to place it in an Effect in the top-level component:

```js {2-6}
function App() {
// ■ Avoid: Effects with logic that should only ever run once
useEffect(() => {
loadDataFromLocalStorage();
checkAuthToken();
}, []);
// ...
}
```

However, you'll quickly discover that it [runs twice in development.](/learn/synchronizing-with-effects#how-to-handle-the-effect-firing-twice-in-development) This can cause issues--for example, maybe it invalidates the authentication token because the function wasn't designed to be called twice. In general, your components should be resilient to being remounted. This includes your top-level `App` component.

Although it may not ever get remounted in practice in production, following the same constraints in all components makes it easier to move and reuse code. If some logic must run *once per app load* rather than *once per component mount*, add a top-level variable to track whether it has already executed:

```js {1,5-6,10}
let didInit = false;
```
```

```
function App() {
useEffect(() => {
if (!didInit) {
didInit = true;
// ■ Only runs once per app load
loadDataFromLocalStorage();
checkAuthToken();
}
}, []);
// ...
}
```

You can also run it during module initialization and before the app renders:

```js {1,5}
if (typeof window !== 'undefined') { // Check if we're running in the browser.
// ■ Only runs once per app load
checkAuthToken();
loadDataFromLocalStorage();
}

function App() {
// ...
}
```

Code at the top level runs once when your component is imported--even if it doesn't end up being rendered. To avoid slowdown or surprising behavior when importing arbitrary components, don't overuse this pattern. Keep app-wide initialization logic to root component modules like `App.js` or in your application's entry point.

Notifying parent components about state changes
{/*notifying-parent-components-about-state-changes*/}

Let's say you're writing a `Toggle` component with an internal `isOn` state which can be either `true` or `false`. There are a few different ways to toggle it (by clicking or dragging). You want to notify the parent component whenever the `Toggle` internal state changes, so you expose an `onChange` event and call it from an Effect:

```js {4-7}
function Toggle({ onChange }) {
```

```
const [isOn, setIsOn] = useState(false);

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

function handleClick() {
setIsOn(!isOn);
}

function handleDragEnd(e) {
if (isCloserToRightEdge(e)) {
setIsOn(true);
} else {
setIsOn(false);
}
}

// ...
}
```

Like earlier, this is not ideal. The `Toggle` updates its state first, and React updates the screen. Then React runs the Effect, which calls the `onChange` function passed from a parent component. Now the parent component will update its own state, starting another render pass. It would be better to do everything in a single pass.

Delete the Effect and instead update the state of *both* components within the same event handler:

```js {5-7,11,16,18}
function Toggle({ onChange }) {
const [isOn, setIsOn] = useState(false);

function updateToggle(nextIsOn) {
// ■ Good: Perform all updates during the event that caused them
setIsOn(nextIsOn);
onChange(nextIsOn);
}

function handleClick() {
updateToggle(!isOn);
```

```
}

function handleDragEnd(e) {
if (isCloserToRightEdge(e)) {
updateToggle(true);
} else {
updateToggle(false);
}
}

// ...
}
```

With this approach, both the `Toggle` component and its parent component update their state during the event. React [batches updates](/learn/queueing-a-series-of-state-updates) from different components together, so there will only be one render pass.

You might also be able to remove the state altogether, and instead receive `isOn` from the parent component:

```js {1,2}
// ■ Also good: the component is fully controlled by its parent
function Toggle({ isOn, onChange }) {
function handleClick() {
onChange(!isOn);
}

function handleDragEnd(e) {
if (isCloserToRightEdge(e)) {
onChange(true);
} else {
onChange(false);
}
}

// ...
}
```

["Lifting state up"](/learn/sharing-state-between-components) lets the parent component fully control the `Toggle` by toggling the parent's own state. This means the parent component will have to contain

more logic, but there will be less state overall to worry about. Whenever you try to keep two different state variables synchronized, try lifting state up instead!

Passing data to the parent {/*passing-data-to-the-parent*/}

This `Child` component fetches some data and then passes it to the `Parent` component in an Effect:

```js {9-14}
function Parent() {
const [data, setData] = useState(null);
// ...
return <Child onFetched={setData} />;
}

function Child({ onFetched }) {
const data = useSomeAPI();
// ■ Avoid: Passing data to the parent in an Effect
useEffect(() => {
if (data) {
onFetched(data);
}
}, [onFetched, data]);
// ...
}
```

In React, data flows from the parent components to their children. When you see something wrong on the screen, you can trace where the information comes from by going up the component chain until you find which component passes the wrong prop or has the wrong state. When child components update the state of their parent components in Effects, the data flow becomes very difficult to trace. Since both the child and the parent need the same data, let the parent component fetch that data, and *pass it down* to the child instead:

```js {4-5}
function Parent() {
const data = useSomeAPI();
// ...
// ■ Good: Passing data down to the child
return <Child data={data} />;
}

function Child({ data }) {
```

```
// ...
}
```

This is simpler and keeps the data flow predictable: the data flows down from the parent to the child.

Subscribing to an external store {/*subscribing-to-an-external-store*/}

Sometimes, your components may need to subscribe to some data outside of the React state. This data could be from a third-party library or a built-in browser API. Since this data can change without React's knowledge, you need to manually subscribe your components to it. This is often done with an Effect, for example:

```js {2-17}
function useOnlineStatus() {
// Not ideal: Manual store subscription in an Effect
const [isOnline, setIsOnline] = useState(true);
useEffect(() => {
function updateState() {
setIsOnline(navigator.onLine);
}

updateState();

window.addEventListener('online', updateState);
window.addEventListener('offline', updateState);
return () => {
window.removeEventListener('online', updateState);
window.removeEventListener('offline', updateState);
};
}, []);
return isOnline;
}

function ChatIndicator() {
const isOnline = useOnlineStatus();
// ...
}
```

Here, the component subscribes to an external data store (in this case, the browser `navigator.onLine` API). Since this API does not exist on the server (so it can't be used for the initial HTML), initially the

state is set to `true`. Whenever the value of that data store changes in the browser, the component updates its state.

Although it's common to use Effects for this, React has a purpose-built Hook for subscribing to an external store that is preferred instead. Delete the Effect and replace it with a call to [`useSyncExternalStore`](/reference/react/useSyncExternalStore):

```js {11-16}
function subscribe(callback) {
window.addEventListener('online', callback);
window.addEventListener('offline', callback);
return () => {
window.removeEventListener('online', callback);
window.removeEventListener('offline', callback);
};
}

function useOnlineStatus() {
// ■ Good: Subscribing to an external store with a built-in Hook
return useSyncExternalStore(
subscribe, // React won't resubscribe for as long as you pass the same function
() => navigator.onLine, // How to get the value on the client
() => true // How to get the value on the server
);
}

function ChatIndicator() {
const isOnline = useOnlineStatus();
// ...
}
```

This approach is less error-prone than manually syncing mutable data to React state with an Effect. Typically, you'll write a custom Hook like `useOnlineStatus()` above so that you don't need to repeat this code in the individual components. [Read more about subscribing to external stores from React components.](/reference/react/useSyncExternalStore)

Fetching data {/*fetching-data*/}

Many apps use Effects to kick off data fetching. It is quite common to write a data fetching Effect like this:

```js {5-10}
```

```
function SearchResults({ query }) {
const [results, setResults] = useState([]);
const [page, setPage] = useState(1);

useEffect(() => {
// ■ Avoid: Fetching without cleanup logic
fetchResults(query, page).then(json => {
setResults(json);
});
}, [query, page]);

function handleNextPageClick() {
setPage(page + 1);
}
// ...
}
```

You *don't* need to move this fetch to an event handler.

This might seem like a contradiction with the earlier examples where you needed to put the logic into the event handlers! However, consider that it's not *the typing event* that's the main reason to fetch. Search inputs are often prepopulated from the URL, and the user might navigate Back and Forward without touching the input.

It doesn't matter where `page` and `query` come from. While this component is visible, you want to keep `results` [synchronized](/learn/synchronizing-with-effects) with data from the network for the current `page` and `query`. This is why it's an Effect.

However, the code above has a bug. Imagine you type `"hello"` fast. Then the `query` will change from `"h"`, to `"he"`, `"hel"`, `"hell"`, and `"hello"`. This will kick off separate fetches, but there is no guarantee about which order the responses will arrive in. For example, the `"hell"` response may arrive *after* the `"hello"` response. Since it will call `setResults()` last, you will be displaying the wrong search results. This is called a ["race condition"](https://en.wikipedia.org/wiki/Race_condition): two different requests "raced" against each other and came in a different order than you expected.

To fix the race condition, you need to [add a cleanup function](/learn/synchronizing-with-effects#fetching-data) to ignore stale responses:

```js {5,7,9,11-13}
function SearchResults({ query }) {
const [results, setResults] = useState([]);
const [page, setPage] = useState(1);
useEffect(() => {
```

```
let ignore = false;
fetchResults(query, page).then(json => {
if (!ignore) {
setResults(json);
}
});
return () => {
ignore = true;
};
}, [query, page]);

function handleNextPageClick() {
setPage(page + 1);
}
// ...
}
```

This ensures that when your Effect fetches data, all responses except the last requested one will be ignored.

Handling race conditions is not the only difficulty with implementing data fetching. You might also want to think about caching responses (so that the user can click Back and see the previous screen instantly), how to fetch data on the server (so that the initial server-rendered HTML contains the fetched content instead of a spinner), and how to avoid network waterfalls (so that a child can fetch data without waiting for every parent).

These issues apply to any UI library, not just React. Solving them is not trivial, which is why modern [frameworks](/learn/start-a-new-react-project#production-grade-react-frameworks) provide more efficient built-in data fetching mechanisms than fetching data in Effects.

If you don't use a framework (and don't want to build your own) but would like to make data fetching from Effects more ergonomic, consider extracting your fetching logic into a custom Hook like in this example:

```js {4}
function SearchResults({ query }) {
const [page, setPage] = useState(1);
const params = new URLSearchParams({ query, page });
const results = useData(`/api/search?${params}`);

function handleNextPageClick() {
setPage(page + 1);
```

```
}
// ...
}

function useData(url) {
const [data, setData] = useState(null);
useEffect(() => {
let ignore = false;
fetch(url)
.then(response => response.json())
.then(json => {
if (!ignore) {
setData(json);
}
});
return () => {
ignore = true;
};
}, [url]);
return data;
}
```

You'll likely also want to add some logic for error handling and to track whether the content is loading. You can build a Hook like this yourself or use one of the many solutions already available in the React ecosystem. **Although this alone won't be as efficient as using a framework's built-in data fetching mechanism, moving the data fetching logic into a custom Hook will make it easier to adopt an efficient data fetching strategy later.**

In general, whenever you have to resort to writing Effects, keep an eye out for when you can extract a piece of functionality into a custom Hook with a more declarative and purpose-built API like `useData` above. The fewer raw `useEffect` calls you have in your components, the easier you will find to maintain your application.

<Recap>

- If you can calculate something during render, you don't need an Effect.
- To cache expensive calculations, add `useMemo` instead of `useEffect`.
- To reset the state of an entire component tree, pass a different `key` to it.
- To reset a particular bit of state in response to a prop change, set it during rendering.

- Code that runs because a component was *displayed* should be in Effects, the rest should be in events.

- If you need to update the state of several components, it's better to do it during a single event.

- Whenever you try to synchronize state variables in different components, consider lifting state up.

- You can fetch data with Effects, but you need to implement cleanup to avoid race conditions.

</Recap>

<Challenges>

Transform data without Effects {/*transform-data-without-effects*/}

The `TodoList` below displays a list of todos. When the "Show only active todos" checkbox is ticked, completed todos are not displayed in the list. Regardless of which todos are visible, the footer displays the count of todos that are not yet completed.

Simplify this component by removing all the unnecessary state and Effects.

<Sandpack>

```js
import { useState, useEffect } from 'react';
import { initialTodos, createTodo } from './todos.js';

export default function TodoList() {
const [todos, setTodos] = useState(initialTodos);
const [showActive, setShowActive] = useState(false);
const [activeTodos, setActiveTodos] = useState([]);
const [visibleTodos, setVisibleTodos] = useState([]);
const [footer, setFooter] = useState(null);

useEffect(() => {
setActiveTodos(todos.filter(todo => !todo.completed));
}, [todos]);

useEffect(() => {
setVisibleTodos(showActive ? activeTodos : todos);
}, [showActive, todos, activeTodos]);

useEffect(() => {
setFooter(
<footer>
{activeTodos.length} todos left
</footer>
```

```
);
}, [activeTodos]);

return (
<>
<label>
<input
type="checkbox"
checked={showActive}
onChange={e => setShowActive(e.target.checked)}
/>
Show only active todos
</label>
<NewTodo onAdd={newTodo => setTodos([...todos, newTodo])} />
<ul>
{visibleTodos.map(todo => (
<li key={todo.id}>
{todo.completed ? <s>{todo.text}</s> : todo.text}
</li>
))}
</ul>
{footer}
</>
);
}

function NewTodo({ onAdd }) {
const [text, setText] = useState('');

function handleAddClick() {
setText('');
onAdd(createTodo(text));
}

return (
<>
<input value={text} onChange={e => setText(e.target.value)} />
<button onClick={handleAddClick}>
```

Add

</button>

</>

);

}

```

```js todos.js

let nextId = 0;

export function createTodo(text, completed = false) {

return {

id: nextId++,

text,

completed

};

}

export const initialTodos = [

createTodo('Get apples', true),

createTodo('Get oranges', true),

createTodo('Get carrots'),

];

```

```css

label { display: block; }

input { margin-top: 10px; }

```

</Sandpack>

<Hint>

If you can calculate something during rendering, you don't need state or an Effect that updates it.

</Hint>

<Solution>

There are only two essential pieces of state in this example: the list of `todos` and the `showActive` state variable which represents whether the checkbox is ticked. All of the other state variables are [redundant](/learn/choosing-the-state-structure#avoid-redundant-state) and can be calculated during
```

rendering instead. This includes the `footer` which you can move directly into the surrounding JSX.

Your result should end up looking like this:

<Sandpack>

```js
import { useState } from 'react';
import { initialTodos, createTodo } from './todos.js';

export default function TodoList() {
const [todos, setTodos] = useState(initialTodos);
const [showActive, setShowActive] = useState(false);
const activeTodos = todos.filter(todo => !todo.completed);
const visibleTodos = showActive ? activeTodos : todos;

return (
<>
<label>
<input
type="checkbox"
checked={showActive}
onChange={e => setShowActive(e.target.checked)}
/>
Show only active todos
</label>
<NewTodo onAdd={newTodo => setTodos([...todos, newTodo])} />
<ul>
{visibleTodos.map(todo => (
<li key={todo.id}>
{todo.completed ? <s>{todo.text}</s> : todo.text}
</li>
))}
</ul>
<footer>
{activeTodos.length} todos left
</footer>
</>
);
```

```
}

function NewTodo({ onAdd }) {
const [text, setText] = useState('');

function handleAddClick() {
setText('');
onAdd(createTodo(text));
}

return (
<>
<input value={text} onChange={e => setText(e.target.value)} />
<button onClick={handleAddClick}>
Add
</button>
</>
);
}
```

```js todos.js
let nextId = 0;

export function createTodo(text, completed = false) {
return {
id: nextId++,
text,
completed
};
}

export const initialTodos = [
createTodo('Get apples', true),
createTodo('Get oranges', true),
createTodo('Get carrots'),
];
```

```css
```

label { display: block; }

input { margin-top: 10px; }

```

</Sandpack>

</Solution>

#### Cache a calculation without Effects {/*cache-a-calculation-without-effects*/}

In this example, filtering the todos was extracted into a separate function called `getVisibleTodos()`. This function contains a `console.log()` call inside of it which helps you notice when it's being called. Toggle "Show only active todos" and notice that it causes `getVisibleTodos()` to re-run. This is expected because visible todos change when you toggle which ones to display.

Your task is to remove the Effect that recomputes the `visibleTodos` list in the `TodoList` component. However, you need to make sure that `getVisibleTodos()` does *not* re-run (and so does not print any logs) when you type into the input.

<Hint>

One solution is to add a `useMemo` call to cache the visible todos. There is also another, less obvious solution.

</Hint>

<Sandpack>

```js
import { useState, useEffect } from 'react';

import { initialTodos, createTodo, getVisibleTodos } from './todos.js';

export default function TodoList() {

const [todos, setTodos] = useState(initialTodos);

const [showActive, setShowActive] = useState(false);

const [text, setText] = useState('');

const [visibleTodos, setVisibleTodos] = useState([]);

useEffect(() => {

setVisibleTodos(getVisibleTodos(todos, showActive));

}, [todos, showActive]);

function handleAddClick() {

setText('');

setTodos([...todos, createTodo(text)]);

}
```
```

```
return (
<>
<label>
<input
type="checkbox"
checked={showActive}
onChange={e => setShowActive(e.target.checked)}
/>
Show only active todos
</label>
<input value={text} onChange={e => setText(e.target.value)} />
<button onClick={handleAddClick}>
Add
</button>
<ul>
{visibleTodos.map(todo => (
<li key={todo.id}>
{todo.completed ? <s>{todo.text}</s> : todo.text}
</li>
))}
</ul>
</>
);
}
```

```js todos.js
let nextId = 0;
let calls = 0;

export function getVisibleTodos(todos, showActive) {
console.log(`getVisibleTodos() was called ${++calls} times`);
const activeTodos = todos.filter(todo => !todo.completed);
const visibleTodos = showActive ? activeTodos : todos;
return visibleTodos;
}
```

````
export function createTodo(text, completed = false) {
return {
id: nextId++,
text,
completed
};
}

export const initialTodos = [
createTodo('Get apples', true),
createTodo('Get oranges', true),
createTodo('Get carrots'),
];
```

```css
label { display: block; }
input { margin-top: 10px; }
```

</Sandpack>

<Solution>

Remove the state variable and the Effect, and instead add a `useMemo` call to cache the result of calling `getVisibleTodos()`:

<Sandpack>

```js
import { useState, useMemo } from 'react';
import { initialTodos, createTodo, getVisibleTodos } from './todos.js';

export default function TodoList() {
const [todos, setTodos] = useState(initialTodos);
const [showActive, setShowActive] = useState(false);
const [text, setText] = useState('');
const visibleTodos = useMemo(
() => getVisibleTodos(todos, showActive),
[todos, showActive]
);
````

```
function handleAddClick() {
setText('');
setTodos([...todos, createTodo(text)]);
}

return (
<>
<label>
<input
type="checkbox"
checked={showActive}
onChange={e => setShowActive(e.target.checked)}
/>
Show only active todos
</label>
<input value={text} onChange={e => setText(e.target.value)} />
<button onClick={handleAddClick}>
Add
</button>
<ul>
{visibleTodos.map(todo => (
<li key={todo.id}>
{todo.completed ? <s>{todo.text}</s> : todo.text}
</li>
))}
</ul>
</>
);
}
```

```js todos.js
let nextId = 0;
let calls = 0;

export function getVisibleTodos(todos, showActive) {
console.log(`getVisibleTodos() was called ${++calls} times`);
```

```
const activeTodos = todos.filter(todo => !todo.completed);
const visibleTodos = showActive ? activeTodos : todos;
return visibleTodos;
}

export function createTodo(text, completed = false) {
return {
id: nextId++,
text,
completed
};
}

export const initialTodos = [
createTodo('Get apples', true),
createTodo('Get oranges', true),
createTodo('Get carrots'),
];
```

```css
label { display: block; }
input { margin-top: 10px; }
```

</Sandpack>

With this change, `getVisibleTodos()` will be called only if `todos` or `showActive` change. Typing into the input only changes the `text` state variable, so it does not trigger a call to `getVisibleTodos()`.

There is also another solution which does not need `useMemo`. Since the `text` state variable can't possibly affect the list of todos, you can extract the `NewTodo` form into a separate component, and move the `text` state variable inside of it:

<Sandpack>

```js
import { useState, useMemo } from 'react';
import { initialTodos, createTodo, getVisibleTodos } from './todos.js';

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

```
const [showActive, setShowActive] = useState(false);
const visibleTodos = getVisibleTodos(todos, showActive);

return (
<>
<label>
<input
type="checkbox"
checked={showActive}
onChange={e => setShowActive(e.target.checked)}
/>
Show only active todos
</label>
<NewTodo onAdd={newTodo => setTodos([...todos, newTodo])} />
<ul>
{visibleTodos.map(todo => (
<li key={todo.id}>
{todo.completed ? <s>{todo.text}</s> : todo.text}
</li>
))}
</ul>
</>
);
}

function NewTodo({ onAdd }) {
const [text, setText] = useState('');

function handleAddClick() {
setText('');
onAdd(createTodo(text));
}

return (
<>
<input value={text} onChange={e => setText(e.target.value)} />
<button onClick={handleAddClick}>
Add
```

</button>
</>
);
}
```

```js todos.js
let nextId = 0;
let calls = 0;

export function getVisibleTodos(todos, showActive) {
console.log(`getVisibleTodos() was called ${++calls} times`);
const activeTodos = todos.filter(todo => !todo.completed);
const visibleTodos = showActive ? activeTodos : todos;
return visibleTodos;
}

export function createTodo(text, completed = false) {
return {
id: nextId++,
text,
completed
};
}

export const initialTodos = [
createTodo('Get apples', true),
createTodo('Get oranges', true),
createTodo('Get carrots'),
];
```

```css
label { display: block; }
input { margin-top: 10px; }
```

</Sandpack>
```

This approach satisfies the requirements too. When you type into the input, only the `text` state variable updates. Since the `text` state variable is in the child `NewTodo` component, the parent `TodoList` component won't get re-rendered. This is why `getVisibleTodos()` doesn't get called when you type. (It would still be called if the `TodoList` re-renders for another reason.)

</Solution>

Reset state without Effects {/*reset-state-without-effects*/}

This `EditContact` component receives a contact object shaped like `{ id, name, email }` as the `savedContact` prop. Try editing the name and email input fields. When you press Save, the contact's button above the form updates to the edited name. When you press Reset, any pending changes in the form are discarded. Play around with this UI to get a feel for it.

When you select a contact with the buttons at the top, the form resets to reflect that contact's details. This is done with an Effect inside `EditContact.js`. Remove this Effect. Find another way to reset the form when `savedContact.id` changes.

<Sandpack>

```js App.js hidden
import { useState } from 'react';
import ContactList from './ContactList.js';
import EditContact from './EditContact.js';

export default function ContactManager() {
const [
contacts,
setContacts
] = useState(initialContacts);
const [
selectedId,
setSelectedId
] = useState(0);
const selectedContact = contacts.find(c =>
c.id === selectedId
);

function handleSave(updatedData) {
const nextContacts = contacts.map(c => {
if (c.id === updatedData.id) {
return updatedData;
} else {
```

```
return c;
}
});
setContacts(nextContacts);
}

return (
<div>
<ContactList
contacts={contacts}
selectedId={selectedId}
onSelect={id => setSelectedId(id)}
/>
<hr />
<EditContact
savedContact={selectedContact}
onSave={handleSave}
/>
</div>
)
}

const initialContacts = [
{ id: 0, name: 'Taylor', email: 'taylor@mail.com' },
{ id: 1, name: 'Alice', email: 'alice@mail.com' },
{ id: 2, name: 'Bob', email: 'bob@mail.com' }
];
```

```js ContactList.js hidden
export default function ContactList({
contacts,
selectedId,
onSelect
}) {
return (
<section>
```

```
<ul>
{contacts.map(contact =>
<li key={contact.id}>
<button onClick={() => {
onSelect(contact.id);
}}>
{contact.id === selectedId ?
<b>{contact.name}</b> :
contact.name
}
</button>
</li>
)}
</ul>
</section>
);
}
```

```js EditContact.js active
import { useState, useEffect } from 'react';

export default function EditContact({ savedContact, onSave }) {
const [name, setName] = useState(savedContact.name);
const [email, setEmail] = useState(savedContact.email);

useEffect(() => {
setName(savedContact.name);
setEmail(savedContact.email);
}, [savedContact]);

return (
<section>
<label>
Name:{' '}
<input
type="text"
value={name}
```

```
onChange={e => setName(e.target.value)}
/>
</label>
<label>
Email:{' '}
<input
type="email"
value={email}
onChange={e => setEmail(e.target.value)}
/>
</label>
<button onClick={() => {
const updatedData = {
id: savedContact.id,
name: name,
email: email
};
onSave(updatedData);
}}>
Save
</button>
<button onClick={() => {
setName(savedContact.name);
setEmail(savedContact.email);
}}>
Reset
</button>
</section>
);
}
```

```css
ul, li {
list-style: none;
margin: 0;
```

padding: 0;

}

li { display: inline-block; }

li button {

padding: 10px;

}

label {

display: block;

margin: 10px 0;

}

button {

margin-right: 10px;

margin-bottom: 10px;

}

```

</Sandpack>

<Hint>

It would be nice if there was a way to tell React that when `savedContact.id` is different, the `EditContact` form is conceptually a _different contact's form_ and should not preserve state. Do you recall any such way?

</Hint>

<Solution>

Split the `EditContact` component in two. Move all the form state into the inner `EditForm` component. Export the outer `EditContact` component, and make it pass `savedContact.id` as the `key` to the inner `EditForm` component. As a result, the inner `EditForm` component resets all of the form state and recreates the DOM whenever you select a different contact.

<Sandpack>

```js App.js hidden

import { useState } from 'react';

import ContactList from './ContactList.js';

import EditContact from './EditContact.js';

export default function ContactManager() {

const [

contacts,

```
setContacts
] = useState(initialContacts);
const [
selectedId,
setSelectedId
] = useState(0);
const selectedContact = contacts.find(c =>
c.id === selectedId
);

function handleSave(updatedData) {
const nextContacts = contacts.map(c => {
if (c.id === updatedData.id) {
return updatedData;
} else {
return c;
}
});
setContacts(nextContacts);
}

return (
<div>
<ContactList
contacts={contacts}
selectedId={selectedId}
onSelect={id => setSelectedId(id)}
/>
<hr />
<EditContact
savedContact={selectedContact}
onSave={handleSave}
/>
</div>
)
}
```

````
const initialContacts = [
{ id: 0, name: 'Taylor', email: 'taylor@mail.com' },
{ id: 1, name: 'Alice', email: 'alice@mail.com' },
{ id: 2, name: 'Bob', email: 'bob@mail.com' }
];
```

```js ContactList.js hidden
export default function ContactList({
contacts,
selectedId,
onSelect
}) {
return (
<section>
<ul>
{contacts.map(contact =>
<li key={contact.id}>
<button onClick={() => {
onSelect(contact.id);
}}>
{contact.id === selectedId ?
<b>{contact.name}</b> :
contact.name
}
</button>
</li>
)}
</ul>
</section>
);
}
```

```js EditContact.js active
import { useState } from 'react';
````

```
export default function EditContact(props) {
return (
<EditForm
{...props}
key={props.savedContact.id}
/>
);
}

function EditForm({ savedContact, onSave }) {
const [name, setName] = useState(savedContact.name);
const [email, setEmail] = useState(savedContact.email);

return (
<section>
<label>
Name:{' '}
<input
type="text"
value={name}
onChange={e => setName(e.target.value)}
/>
</label>
<label>
Email:{' '}
<input
type="email"
value={email}
onChange={e => setEmail(e.target.value)}
/>
</label>
<button onClick={() => {
const updatedData = {
id: savedContact.id,
name: name,
email: email
};
```

onSave(updatedData);
}}>
Save
</button>
<button onClick={() => {
setName(savedContact.name);
setEmail(savedContact.email);
}}>
Reset
</button>
</section>
);
}
```

```css
ul, li {
list-style: none;
margin: 0;
padding: 0;
}
li { display: inline-block; }
li button {
padding: 10px;
}
label {
display: block;
margin: 10px 0;
}
button {
margin-right: 10px;
margin-bottom: 10px;
}
```

</Sandpack>
```

</Solution>

Submit a form without Effects {/*submit-a-form-without-effects*/}

This `Form` component lets you send a message to a friend. When you submit the form, the `showForm` state variable is set to `false`. This triggers an Effect calling `sendMessage(message)`, which sends the message (you can see it in the console). After the message is sent, you see a "Thank you" dialog with an "Open chat" button that lets you get back to the form.

Your app's users are sending way too many messages. To make chatting a little bit more difficult, you've decided to show the "Thank you" dialog *first* rather than the form. Change the `showForm` state variable to initialize to `false` instead of `true`. As soon as you make that change, the console will show that an empty message was sent. Something in this logic is wrong!

What's the root cause of this problem? And how can you fix it?

<Hint>

Should the message be sent _because_ the user saw the "Thank you" dialog? Or is it the other way around?

</Hint>

<Sandpack>

```js
import { useState, useEffect } from 'react';

export default function Form() {
const [showForm, setShowForm] = useState(true);
const [message, setMessage] = useState('');

useEffect(() => {
if (!showForm) {
sendMessage(message);
}
}, [showForm, message]);

function handleSubmit(e) {
e.preventDefault();
setShowForm(false);
}

if (!showForm) {
return (
<>
```

<h1>Thanks for using our services!</h1>
<button onClick={() => {
setMessage('');
setShowForm(true);
}}>
Open chat
</button>
</>
);
}

return (
<form onSubmit={handleSubmit}>
<textarea
placeholder="Message"
value={message}
onChange={e => setMessage(e.target.value)}
/>
<button type="submit" disabled={message === ''}>
Send
</button>
</form>
);
}

function sendMessage(message) {
console.log('Sending message: ' + message);
}
```

```css
label, textarea { margin-bottom: 10px; display: block; }
```

</Sandpack>

<Solution>

The `showForm` state variable determines whether to show the form or the "Thank you" dialog. However, you aren't sending the message because the "Thank you" dialog was _displayed_. You want
```

to send the message because the user has _submitted the form._ Delete the misleading Effect and move the `sendMessage` call inside the `handleSubmit` event handler:

<Sandpack>

```js
import { useState, useEffect } from 'react';

export default function Form() {
const [showForm, setShowForm] = useState(true);
const [message, setMessage] = useState('');

function handleSubmit(e) {
e.preventDefault();
setShowForm(false);
sendMessage(message);
}

if (!showForm) {
return (
<>
<h1>Thanks for using our services!</h1>
<button onClick={() => {
setMessage('');
setShowForm(true);
}}>
Open chat
</button>
</>
);
}

return (
<form onSubmit={handleSubmit}>
<textarea
placeholder="Message"
value={message}
onChange={e => setMessage(e.target.value)}
/>
<button type="submit" disabled={message === ''}>
```

Send

</button>

</form>

);

}

function sendMessage(message) {

console.log('Sending message: ' + message);

}

```

```css

label, textarea { margin-bottom: 10px; display: block; }

```

</Sandpack>

Notice how in this version, only _submitting the form_ (which is an event) causes the message to be sent. It works equally well regardless of whether `showForm` is initially set to `true` or `false`. (Set it to `false` and notice no extra console messages.)

</Solution>

</Challenges>

---

title: Thinking in React

<Intro>

React can change how you think about the designs you look at and the apps you build. When you build a user interface with React, you will first break it apart into pieces called *components*. Then, you will describe the different visual states for each of your components. Finally, you will connect your components together so that the data flows through them. In this tutorial, we'll guide you through the thought process of building a searchable product data table with React.

</Intro>

Start with the mockup {/*start-with-the-mockup*/}

Imagine that you already have a JSON API and a mockup from a designer.

The JSON API returns some data that looks like this:

```json

[

{ category: "Fruits", price: "$1", stocked: true, name: "Apple" },

{ category: "Fruits", price: "$1", stocked: true, name: "Dragonfruit" },

{ category: "Fruits", price: "$2", stocked: false, name: "Passionfruit" },

{ category: "Vegetables", price: "$2", stocked: true, name: "Spinach" },

{ category: "Vegetables", price: "$4", stocked: false, name: "Pumpkin" },

{ category: "Vegetables", price: "$1", stocked: true, name: "Peas" }

]

```

The mockup looks like this:

To implement a UI in React, you will usually follow the same five steps.

Step 1: Break the UI into a component hierarchy
{/*step-1-break-the-ui-into-a-component-hierarchy*/}

Start by drawing boxes around every component and subcomponent in the mockup and naming them. If you work with a designer, they may have already named these components in their design tool. Ask them!

Depending on your background, you can think about splitting up a design into components in different ways:

* **Programming**--use the same techniques for deciding if you should create a new function or object. One such technique is the [single responsibility principle](https://en.wikipedia.org/wiki/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.

* **CSS**--consider what you would make class selectors for. (However, components are a bit less granular.)

* **Design**--consider how you would organize the design's layers.

If your JSON is well-structured, you'll often find that it naturally maps to the component structure of your UI. That's because UI and data models often have the same information architecture--that is, the same shape. Separate your UI into components, where each component matches one piece of your data model.

There are five components on this screen:

<FullWidth>

<CodeDiagram flip>

1. `FilterableProductTable` (grey) contains the entire app.

2. `SearchBar` (blue) receives the user input.
```

3. `ProductTable` (lavender) displays and filters the list according to the user input.

4. `ProductCategoryRow` (green) displays a heading for each category.

5. `ProductRow` (yellow) displays a row for each product.

</CodeDiagram>

</FullWidth>

If you look at `ProductTable` (lavender), 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 you could go either way. For this example, it is a part of `ProductTable` because it appears inside the `ProductTable`'s list. However, if this header grows to be complex (e.g., if you add sorting), you can move it into its own `ProductTableHeader` component.

Now that you've identified the components in the mockup, arrange them into a hierarchy. Components that appear within another component in the mockup should appear as a child in the hierarchy:

* `FilterableProductTable`

* `SearchBar`

* `ProductTable`

* `ProductCategoryRow`

* `ProductRow`

Step 2: Build a static version in React {/*step-2-build-a-static-version-in-react*/}

Now that you have your component hierarchy, it's time to implement your app. The most straightforward approach is to build a version that renders the UI from your data model without adding any interactivity... yet! It's often easier to build the static version first and add interactivity later. Building a static version requires a lot of typing and no thinking, but adding interactivity requires a lot of thinking and not a lot of typing.

To build a static version of your app that renders your data model, you'll want to build [components](/learn/your-first-component) that reuse other components and pass data using [props.](/learn/passing-props-to-a-component) Props are a way of passing data from parent to child. (If you're familiar with the concept of [state](/learn/state-a-components-memory), 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 either build "top down" by starting with building the components higher up in the hierarchy (like `FilterableProductTable`) or "bottom up" by working from components lower down (like `ProductRow`). In simpler examples, it’s usually easier to go top-down, and on larger projects, it’s easier to go bottom-up.

<Sandpack>

```jsx App.js
function ProductCategoryRow({ category }) {
return (
```

```
<tr>
<th colSpan="2">
{category}
</th>
</tr>
);
}

function ProductRow({ product }) {
const name = product.stocked ? product.name :
<span style={{ color: 'red' }}>
{product.name}
</span>;

return (
<tr>
<td>{name}</td>
<td>{product.price}</td>
</tr>
);
}

function ProductTable({ products }) {
const rows = [];
let lastCategory = null;

products.forEach((product) => {
if (product.category !== lastCategory) {
rows.push(
<ProductCategoryRow
category={product.category}
key={product.category} />
);
}
rows.push(
<ProductRow
product={product}
key={product.name} />
```

```
);
lastCategory = product.category;
});

return (
<table>
<thead>
<tr>
<th>Name</th>
<th>Price</th>
</tr>
</thead>
<tbody>{rows}</tbody>
</table>
);
}

function SearchBar() {
return (
<form>
<input type="text" placeholder="Search..." />
<label>
<input type="checkbox" />
{' '}
Only show products in stock
</label>
</form>
);
}

function FilterableProductTable({ products }) {
return (
<div>
<SearchBar />
<ProductTable products={products} />
</div>
);
```

```
}

const PRODUCTS = [
{category: "Fruits", price: "$1", stocked: true, name: "Apple"},
{category: "Fruits", price: "$1", stocked: true, name: "Dragonfruit"},
{category: "Fruits", price: "$2", stocked: false, name: "Passionfruit"},
{category: "Vegetables", price: "$2", stocked: true, name: "Spinach"},
{category: "Vegetables", price: "$4", stocked: false, name: "Pumpkin"},
{category: "Vegetables", price: "$1", stocked: true, name: "Peas"}
];

export default function App() {
return <FilterableProductTable products={PRODUCTS} />;
}
```

```css
body {
padding: 5px
}
label {
display: block;
margin-top: 5px;
margin-bottom: 5px;
}
th {
padding-top: 10px;
}
td {
padding: 2px;
padding-right: 40px;
}
```

</Sandpack>

(If this code looks intimidating, go through the [Quick Start](/learn/) first!)

After building your components, you'll have a library of reusable components that render your data model. Because this is a static app, the components will only return JSX. The component at the top of

the hierarchy (`FilterableProductTable`) will take your data model as a prop. This is called _one-way data flow_ because the data flows down from the top-level component to the ones at the bottom of the tree.

<Pitfall>

At this point, you should not be using any state values. That’s for the next step!

</Pitfall>

Step 3: Find the minimal but complete representation of UI state
{/*step-3-find-the-minimal-but-complete-representation-of-ui-state*/}

To make the UI interactive, you need to let users change your underlying data model. You will use *state* for this.

Think of state as the minimal set of changing data that your app needs to remember. The most important principle for structuring state is to keep it [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 on-demand. For example, if you're building a shopping list, you can store the items as an array in state. If you want to also display the number of items in the list, don't store the number of items as another state value--instead, read the length of your array.

Now think of all of the pieces of data in this example application:

1. The original list of products
2. The search text the user has entered
3. The value of the checkbox
4. The filtered list of products

Which of these are state? Identify the ones that are not:

* Does it **remain unchanged** over time? If so, it isn't state.
* Is it **passed in from a parent** via props? If so, it isn't state.
* **Can you compute it** based on existing state or props in your component? If so, it *definitely* isn't state!

What's left is probably state.

Let's go through them one by one again:

1. The original list of products is **passed in as props, so it's not state.**
2. The search text seems to be state since it changes over time and can't be computed from anything.
3. The value of the checkbox seems to be state since it changes over time and can't be computed from anything.
4. The filtered list of products **isn't state because it can be computed** by taking the original list of products and filtering it according to the search text and value of the checkbox.

This means only the search text and the value of the checkbox are state! Nicely done!

<DeepDive>

Props vs State {/*props-vs-state*/}

There are two types of "model" data in React: props and state. The two are very different:

* [**Props** are like arguments you pass](/learn/passing-props-to-a-component) to a function. They let a parent component pass data to a child component and customize its appearance. For example, a `Form` can pass a `color` prop to a `Button`.

* [**State** is like a component's memory.](/learn/state-a-components-memory) It lets a component keep track of some information and change it in response to interactions. For example, a `Button` might keep track of `isHovered` state.

Props and state are different, but they work together. A parent component will often keep some information in state (so that it can change it), and *pass it down* to child components as their props. It's okay if the difference still feels fuzzy on the first read. It takes a bit of practice for it to really stick!

</DeepDive>

Step 4: Identify where your state should live {/*step-4-identify-where-your-state-should-live*/}

After identifying your app's minimal state data, you need to identify which component is responsible for changing this state, or *owns* the state. Remember: React uses one-way data flow, passing data down the component hierarchy from parent to child component. It may not be immediately clear which component should own what state. This can be challenging if you're new to this concept, but you can figure it out by following these steps!

For each piece of state in your application:

1. Identify *every* component that renders something based on that state.

2. Find their closest common parent component--a component above them all in the hierarchy.

3. Decide where the state should live:

1. Often, you can put the state directly into their common parent.

2. You can also put the state into some component above their common parent.

3. If you can't find a component where it makes sense to own the state, create a new component solely for holding the state and add it somewhere in the hierarchy above the common parent component.

In the previous step, you found two pieces of state in this application: the search input text, and the value of the checkbox. In this example, they always appear together, so it makes sense to put them into the same place.

Now let's run through our strategy for them:

1. **Identify components that use state:**

* `ProductTable` needs to filter the product list based on that state (search text and checkbox value).

* `SearchBar` needs to display that state (search text and checkbox value).

1. **Find their common parent:** The first parent component both components share is `FilterableProductTable`.

2. **Decide where the state lives**: We'll keep the filter text and checked state values in `FilterableProductTable`.

So the state values will live in `FilterableProductTable`.

Add state to the component with the [`useState()` Hook.](/reference/react/useState) Hooks are special functions that let you "hook into" React. Add two state variables at the top of `FilterableProductTable` and specify their initial state:

```js
function FilterableProductTable({ products }) {
const [filterText, setFilterText] = useState('');
const [inStockOnly, setInStockOnly] = useState(false);
```

Then, pass `filterText` and `inStockOnly` to `ProductTable` and `SearchBar` as props:

```js
<div>
<SearchBar
filterText={filterText}
inStockOnly={inStockOnly} />
<ProductTable
products={products}
filterText={filterText}
inStockOnly={inStockOnly} />
</div>
```

You can start seeing how your application will behave. Edit the `filterText` initial value from `useState('')` to `useState('fruit')` in the sandbox code below. You'll see both the search input text and the table update:

<Sandpack>

```jsx App.js
import { useState } from 'react';

function FilterableProductTable({ products }) {
const [filterText, setFilterText] = useState('');
const [inStockOnly, setInStockOnly] = useState(false);
```

```
return (
<div>
<SearchBar
filterText={filterText}
inStockOnly={inStockOnly} />
<ProductTable
products={products}
filterText={filterText}
inStockOnly={inStockOnly} />
</div>
);
}

function ProductCategoryRow({ category }) {
return (
<tr>
<th colSpan="2">
{category}
</th>
</tr>
);
}

function ProductRow({ product }) {
const name = product.stocked ? product.name :
<span style={{ color: 'red' }}>
{product.name}
</span>;

return (
<tr>
<td>{name}</td>
<td>{product.price}</td>
</tr>
);
}

function ProductTable({ products, filterText, inStockOnly }) {
```

```
const rows = [];
let lastCategory = null;

products.forEach((product) => {
if (
product.name.toLowerCase().indexOf(
filterText.toLowerCase()
) === -1
) {
return;
}
if (inStockOnly && !product.stocked) {
return;
}
if (product.category !== lastCategory) {
rows.push(
<ProductCategoryRow
category={product.category}
key={product.category} />
);
}
rows.push(
<ProductRow
product={product}
key={product.name} />
);
lastCategory = product.category;
});

return (
<table>
<thead>
<tr>
<th>Name</th>
<th>Price</th>
</tr>
</thead>
```

```
<tbody>{rows}</tbody>
</table>
);
}

function SearchBar({ filterText, inStockOnly }) {
return (
<form>
<input
type="text"
value={filterText}
placeholder="Search..."/>
<label>
<input
type="checkbox"
checked={inStockOnly} />
{' '}
Only show products in stock
</label>
</form>
);
}

const PRODUCTS = [
{category: "Fruits", price: "$1", stocked: true, name: "Apple"},
{category: "Fruits", price: "$1", stocked: true, name: "Dragonfruit"},
{category: "Fruits", price: "$2", stocked: false, name: "Passionfruit"},
{category: "Vegetables", price: "$2", stocked: true, name: "Spinach"},
{category: "Vegetables", price: "$4", stocked: false, name: "Pumpkin"},
{category: "Vegetables", price: "$1", stocked: true, name: "Peas"}
];

export default function App() {
return <FilterableProductTable products={PRODUCTS} />;
}
```

```css
```

```
body {
padding: 5px
}
label {
display: block;
margin-top: 5px;
margin-bottom: 5px;
}
th {
padding-top: 5px;
}
td {
padding: 2px;
}
```

</Sandpack>

Notice that editing the form doesn't work yet. There is a console error in the sandbox above explaining why:

<ConsoleBlock level="error">

You provided a \`value\` prop to a form field without an \`onChange\` handler. This will render a read-only field.

</ConsoleBlock>

In the sandbox above, `ProductTable` and `SearchBar` read the `filterText` and `inStockOnly` props to render the table, the input, and the checkbox. For example, here is how `SearchBar` populates the input value:

```js {1,6}
function SearchBar({ filterText, inStockOnly }) {
return (
<form>
<input
type="text"
value={filterText}
placeholder="Search..."/>
```

However, you haven't added any code to respond to the user actions like typing yet. This will be your final step.

Step 5: Add inverse data flow {/*step-5-add-inverse-data-flow*/}

Currently your app renders correctly with props and state flowing down the hierarchy. But to change the state according to user input, you will need 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, but it requires a little more typing than two-way data binding. If you try to type or check the box in the example above, you'll see that React ignores your input. This is intentional. By writing `<input value={filterText} />`, you've set the `value` prop of the `input` to always be equal to the `filterText` state passed in from `FilterableProductTable`. Since `filterText` state is never set, the input never changes.

You want to make it so whenever the user changes the form inputs, the state updates to reflect those changes. The state is owned by `FilterableProductTable`, so only it can call `setFilterText` and `setInStockOnly`. To let `SearchBar` update the `FilterableProductTable`'s state, you need to pass these functions down to `SearchBar`:

```js {2,3,10,11}
function FilterableProductTable({ products }) {
const [filterText, setFilterText] = useState('');
const [inStockOnly, setInStockOnly] = useState(false);

return (
<div>
<SearchBar
filterText={filterText}
inStockOnly={inStockOnly}
onFilterTextChange={setFilterText}
onInStockOnlyChange={setInStockOnly} />
```

Inside the `SearchBar`, you will add the `onChange` event handlers and set the parent state from them:

```js {5}
<input
type="text"
value={filterText}
placeholder="Search..."
onChange={(e) => onFilterTextChange(e.target.value)} />
```

Now the application fully works!

<Sandpack>

```jsx App.js
import { useState } from 'react';

function FilterableProductTable({ products }) {
const [filterText, setFilterText] = useState('');
const [inStockOnly, setInStockOnly] = useState(false);

return (
<div>
<SearchBar
filterText={filterText}
inStockOnly={inStockOnly}
onFilterTextChange={setFilterText}
onInStockOnlyChange={setInStockOnly} />
<ProductTable
products={products}
filterText={filterText}
inStockOnly={inStockOnly} />
</div>
);
}

function ProductCategoryRow({ category }) {
return (
<tr>
<th colSpan="2">
{category}
</th>
</tr>
);
}

function ProductRow({ product }) {
const name = product.stocked ? product.name :
<span style={{ color: 'red' }}>
```

```
{product.name}
</span>;

return (
<tr>
<td>{name}</td>
<td>{product.price}</td>
</tr>
);
}

function ProductTable({ products, filterText, inStockOnly }) {
const rows = [];
let lastCategory = null;

products.forEach((product) => {
if (
product.name.toLowerCase().indexOf(
filterText.toLowerCase()
) === -1
) {
return;
}
if (inStockOnly && !product.stocked) {
return;
}
if (product.category !== lastCategory) {
rows.push(
<ProductCategoryRow
category={product.category}
key={product.category} />
);
}
rows.push(
<ProductRow
product={product}
key={product.name} />
```

```
);
lastCategory = product.category;
});

return (
<table>
<thead>
<tr>
<th>Name</th>
<th>Price</th>
</tr>
</thead>
<tbody>{rows}</tbody>
</table>
);
}

function SearchBar({
filterText,
inStockOnly,
onFilterTextChange,
onInStockOnlyChange
}) {
return (
<form>
<input
type="text"
value={filterText} placeholder="Search..."
onChange={(e) => onFilterTextChange(e.target.value)} />
<label>
<input
type="checkbox"
checked={inStockOnly}
onChange={(e) => onInStockOnlyChange(e.target.checked)} />
{' '}
Only show products in stock
</label>
```

```
</form>
);
}

const PRODUCTS = [
{category: "Fruits", price: "$1", stocked: true, name: "Apple"},
{category: "Fruits", price: "$1", stocked: true, name: "Dragonfruit"},
{category: "Fruits", price: "$2", stocked: false, name: "Passionfruit"},
{category: "Vegetables", price: "$2", stocked: true, name: "Spinach"},
{category: "Vegetables", price: "$4", stocked: false, name: "Pumpkin"},
{category: "Vegetables", price: "$1", stocked: true, name: "Peas"}
];

export default function App() {
return <FilterableProductTable products={PRODUCTS} />;
}
```

```css
body {
padding: 5px
}
label {
display: block;
margin-top: 5px;
margin-bottom: 5px;
}
th {
padding: 4px;
}
td {
padding: 2px;
}
```

</Sandpack>

You can learn all about handling events and updating state in the [Adding Interactivity](/learn/adding-interactivity) section.

Where to go from here {/*where-to-go-from-here*/}

This was a very brief introduction to how to think about building components and applications with React. You can [start a React project](/learn/installation) right now or [dive deeper on all the syntax](/learn/describing-the-ui) used in this tutorial.

---

title: Writing Markup with JSX

<Intro>

JSX is a syntax extension for JavaScript that lets you write HTML-like markup inside a JavaScript file. Although there are other ways to write components, most React developers prefer the conciseness of JSX, and most codebases use it.

</Intro>

<YouWillLearn>

* Why React mixes markup with rendering logic

* How JSX is different from HTML

* How to display information with JSX

</YouWillLearn>

JSX: Putting markup into JavaScript {/*jsx-putting-markup-into-javascript*/}

The Web has been built on HTML, CSS, and JavaScript. For many years, web developers kept content in HTML, design in CSS, and logic in JavaScript—often in separate files! Content was marked up inside HTML while the page's logic lived separately in JavaScript:

<DiagramGroup>

<Diagram name="writing_jsx_html" height={237} width={325} alt="HTML markup with purple background and a div with two child tags: p and form. ">

HTML

</Diagram>

<Diagram name="writing_jsx_js" height={237} width={325} alt="Three JavaScript handlers with yellow background: onSubmit, onLogin, and onClick.">

JavaScript

</Diagram>

</DiagramGroup>

But as the Web became more interactive, logic increasingly determined content. JavaScript was in charge of the HTML! This is why **in React, rendering logic and markup live together in the same place—components.**

<DiagramGroup>

<Diagram name="writing_jsx_sidebar" height={330} width={325} alt="React component with HTML and JavaScript from previous examples mixed. Function name is Sidebar which calls the function isLoggedIn, highlighted in yellow. Nested inside the function highlighted in purple is the p tag from before, and a Form tag referencing the component shown in the next diagram.">

`Sidebar.js` React component

</Diagram>

<Diagram name="writing_jsx_form" height={330} width={325} alt="React component with HTML and JavaScript from previous examples mixed. Function name is Form containing two handlers onClick and onSubmit highlighted in yellow. Following the handlers is HTML highlighted in purple. The HTML contains a form element with a nested input element, each with an onClick prop.">

`Form.js` React component

</Diagram>

</DiagramGroup>

Keeping a button's rendering logic and markup together ensures that they stay in sync with each other on every edit. Conversely, details that are unrelated, such as the button's markup and a sidebar's markup, are isolated from each other, making it safer to change either of them on their own.

Each React component is a JavaScript function that may contain some markup that React renders into the browser. React components use a syntax extension called JSX to represent that markup. JSX looks a lot like HTML, but it is a bit stricter and can display dynamic information. The best way to understand this is to convert some HTML markup to JSX markup.

<Note>

JSX and React are two separate things. They're often used together, but you *can* [use them independently](https://reactjs.org/blog/2020/09/22/introducing-the-new-jsx-transform.html#whats-a-jsx-transform) of each other. JSX is a syntax extension, while React is a JavaScript library.

</Note>

Converting HTML to JSX {/*converting-html-to-jsx*/}

Suppose that you have some (perfectly valid) HTML:

```html
<h1>Hedy Lamarr's Todos</h1>
<img
src="https://i.imgur.com/yXOvdOSs.jpg"
```

alt="Hedy Lamarr"
class="photo"
>

Invent new traffic lights
Rehearse a movie scene
Improve the spectrum technology

```

And you want to put it into your component:

```js
export default function TodoList() {
return (
// ???
)
}
```

If you copy and paste it as is, it will not work:

<Sandpack>

```js
export default function TodoList() {
return (
// This doesn't quite work!
<h1>Hedy Lamarr's Todos</h1>
<img
src="https://i.imgur.com/yXOvdOSs.jpg"
alt="Hedy Lamarr"
class="photo"
>
<ul>
<li>Invent new traffic lights
<li>Rehearse a movie scene
<li>Improve the spectrum technology

````

);
}
```

```css
img { height: 90px }
```

</Sandpack>

This is because JSX is stricter and has a few more rules than HTML! If you read the error messages above, they'll guide you to fix the markup, or you can follow the guide below.

<Note>

Most of the time, React's on-screen error messages will help you find where the problem is. Give them a read if you get stuck!

</Note>

The Rules of JSX {/*the-rules-of-jsx*/}

1. Return a single root element {/*1-return-a-single-root-element*/}

To return multiple elements from a component, **wrap them with a single parent tag.**

For example, you can use a `<div>`:

```js {1,11}
<div>
<h1>Hedy Lamarr's Todos</h1>
<img
src="https://i.imgur.com/yXOvdOSs.jpg"
alt="Hedy Lamarr"
class="photo"
>
<ul>
...
</ul>
</div>
```

If you don't want to add an extra `<div>` to your markup, you can write `<>` and `</>` instead:

```js {1,11}
<>
<h1>Hedy Lamarr's Todos</h1>
<img
src="https://i.imgur.com/yXOvdOSs.jpg"
alt="Hedy Lamarr"
class="photo"
>
<ul>
...
</ul>
</>
```

This empty tag is called a *[Fragment.](/reference/react/Fragment)* Fragments let you group things without leaving any trace in the browser HTML tree.

<DeepDive>

Why do multiple JSX tags need to be wrapped?
{/*why-do-multiple-jsx-tags-need-to-be-wrapped*/}

JSX looks like HTML, but under the hood it is transformed into plain JavaScript objects. You can't return two objects from a function without wrapping them into an array. This explains why you also can't return two JSX tags without wrapping them into another tag or a Fragment.

</DeepDive>

2. Close all the tags {/*2-close-all-the-tags*/}

JSX requires tags to be explicitly closed: self-closing tags like `<img>` must become `<img />`, and wrapping tags like `<li>oranges` must be written as `<li>oranges</li>`.

This is how Hedy Lamarr's image and list items look closed:

```js {2-6,8-10}
<>
<img
src="https://i.imgur.com/yXOvdOSs.jpg"
alt="Hedy Lamarr"
class="photo"
/>
<ul>
```

Invent new traffic lights

Rehearse a movie scene

Improve the spectrum technology

</>

```

### 3. camelCase <s>all</s> most of the things! {/*3-camelcase-salls-most-of-the-things*/}

JSX turns into JavaScript and attributes written in JSX become keys of JavaScript objects. In your own components, you will often want to read those attributes into variables. But JavaScript has limitations on variable names. For example, their names can't contain dashes or be reserved words like `class`.

This is why, in React, many HTML and SVG attributes are written in camelCase. For example, instead of `stroke-width` you use `strokeWidth`. Since `class` is a reserved word, in React you write `className` instead, named after the [corresponding DOM property](https://developer.mozilla.org/en-US/docs/Web/API/Element/className):

```js {4}

<img

src="https://i.imgur.com/yXOvdOSs.jpg"

alt="Hedy Lamarr"

className="photo"

/>

```

You can [find all these attributes in the list of DOM component props.](/reference/react-dom/components/common) If you get one wrong, don't worry—React will print a message with a possible correction to the [browser console.](https://developer.mozilla.org/docs/Tools/Browser_Console)

<Pitfall>

For historical reasons, [`aria-*`](https://developer.mozilla.org/docs/Web/Accessibility/ARIA) and [`data-*`](https://developer.mozilla.org/docs/Learn/HTML/Howto/Use_data_attributes) attributes are written as in HTML with dashes.

</Pitfall>

Pro-tip: Use a JSX Converter {/*pro-tip-use-a-jsx-converter*/}

Converting all these attributes in existing markup can be tedious! We recommend using a [converter](https://transform.tools/html-to-jsx) to translate your existing HTML and SVG to JSX. Converters are very useful in practice, but it's still worth understanding what is going on so that you can comfortably write JSX on your own.

Here is your final result:
```

<Sandpack>

```js
export default function TodoList() {
return (
<>
<h1>Hedy Lamarr's Todos</h1>
<img
src="https://i.imgur.com/yXOvdOSs.jpg"
alt="Hedy Lamarr"
className="photo"
/>
<ul>
<li>Invent new traffic lights</li>
<li>Rehearse a movie scene</li>
<li>Improve the spectrum technology</li>
</ul>
</>
);
}
```

```css
img { height: 90px }
```

</Sandpack>

<Recap>

Now you know why JSX exists and how to use it in components:

* React components group rendering logic together with markup because they are related.
* JSX is similar to HTML, with a few differences. You can use a [converter](https://transform.tools/html-to-jsx) if you need to.
* Error messages will often point you in the right direction to fixing your markup.

</Recap>

<Challenges>

Convert some HTML to JSX {/*convert-some-html-to-jsx*/}

This HTML was pasted into a component, but it's not valid JSX. Fix it:

<Sandpack>

```js
export default function Bio() {
return (
<div class="intro">
<h1>Welcome to my website!</h1>
</div>
<p class="summary">
You can find my thoughts here.
<br><br>
<b>And <i>pictures</b></i> of scientists!
</p>
);
}
```

```css
.intro {
background-image: linear-gradient(to left, violet, indigo, blue, green, yellow, orange, red);
background-clip: text;
color: transparent;
-webkit-background-clip: text;
-webkit-text-fill-color: transparent;
}

.summary {
padding: 20px;
border: 10px solid gold;
}
```

</Sandpack>

Whether to do it by hand or using the converter is up to you!

<Solution>

<Sandpack>

```js
export default function Bio() {
return (
<div>
<div className="intro">
<h1>Welcome to my website!</h1>
</div>
<p className="summary">
You can find my thoughts here.
<br /><br />
<b>And <i>pictures</i></b> of scientists!
</p>
</div>
);
}
```

```css
.intro {
background-image: linear-gradient(to left, violet, indigo, blue, green, yellow, orange, red);
background-clip: text;
color: transparent;
-webkit-background-clip: text;
-webkit-text-fill-color: transparent;
}

.summary {
padding: 20px;
border: 10px solid gold;
}
```

</Sandpack>

</Solution>

</Challenges>

---

title: Add React to an Existing Project

<Intro>

If you want to add some interactivity to your existing project, you don't have to rewrite it in React. Add React to your existing stack, and render interactive React components anywhere.

</Intro>

<Note>

You need to install [Node.js](https://nodejs.org/en/) for local development. Although you can [try React](/learn/installation#try-react) online or with a simple HTML page, realistically most JavaScript tooling you'll want to use for development requires Node.js.

</Note>

Using React for an entire subroute of your existing website {/*using-react-for-an-entire-subroute-of-your-existing-website*/}

Let's say you have an existing web app at `example.com` built with another server technology (like Rails), and you want to implement all routes starting with `example.com/some-app/` fully with React.

Here's how we recommend to set it up:

1. **Build the React part of your app** using one of the [React-based frameworks](/learn/start-a-new-react-project).
2. **Specify `/some-app` as the *base path*** in your framework's configuration (here's how: [Next.js](https://nextjs.org/docs/api-reference/next.config.js/basepath), [Gatsby](https://www.gatsbyjs.com/docs/how-to/previews-deploys-hosting/path-prefix/)).
3. **Configure your server or a proxy** so that all requests under `/some-app/` are handled by your React app.

This ensures the React part of your app can [benefit from the best practices](/learn/start-a-new-react-project#can-i-use-react-without-a-framework) baked into those frameworks.

Many React-based frameworks are full-stack and let your React app take advantage of the server. However, you can use the same approach even if you can't or don't want to run JavaScript on the server. In that case, serve the HTML/CSS/JS export ([`next export` output](https://nextjs.org/docs/advanced-features/static-html-export) for Next.js, default for Gatsby) at `/some-app/` instead.

Using React for a part of your existing page {/*using-react-for-a-part-of-your-existing-page*/}

Let's say you have an existing page built with another technology (either a server one like Rails, or a client one like Backbone), and you want to render interactive React components somewhere on that page. That's a common way to integrate React--in fact, it's how most React usage looked at Meta for many years!

You can do this in two steps:

1. **Set up a JavaScript environment** that lets you use the [JSX syntax](/learn/writing-markup-with-jsx), split your code into modules with the [`import`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import) / [`export`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export) syntax, and use packages (for example, React) from the [npm](https://www.npmjs.com/) package registry.
2. **Render your React components** where you want to see them on the page.

The exact approach depends on your existing page setup, so let's walk through some details.

Step 1: Set up a modular JavaScript environment {/*step-1-set-up-a-modular-javascript-environment*/}

A modular JavaScript environment lets you write your React components in individual files, as opposed to writing all of your code in a single file. It also lets you use all the wonderful packages published by other developers on the [npm](https://www.npmjs.com/) registry--including React itself! How you do this depends on your existing setup:

* **If your app is already split into files that use `import` statements,** try to use the setup you already have. Check whether writing `<div />` in your JS code causes a syntax error. If it causes a syntax error, you might need to [transform your JavaScript code with Babel](https://babeljs.io/setup), and enable the [Babel React preset](https://babeljs.io/docs/babel-preset-react) to use JSX.

* **If your app doesn't have an existing setup for compiling JavaScript modules,** set it up with [Vite](https://vitejs.dev/). The Vite community maintains [many integrations with backend frameworks](https://github.com/vitejs/awesome-vite#integrations-with-backends), including Rails, Django, and Laravel. If your backend framework is not listed, [follow this guide](https://vitejs.dev/guide/backend-integration.html) to manually integrate Vite builds with your backend.

To check whether your setup works, run this command in your project folder:

<TerminalBlock>

npm install react react-dom

</TerminalBlock>

Then add these lines of code at the top of your main JavaScript file (it might be called `index.js` or `main.js`):

<Sandpack>

```html index.html hidden
<!DOCTYPE html>
<html>
<head><title>My app</title></head>
<body>
<!-- Your existing page content (in this example, it gets replaced) -->
```

</body>

</html>

```

```js index.js active
import { createRoot } from 'react-dom/client';

// Clear the existing HTML content

document.body.innerHTML = '<div id="app"></div>';

// Render your React component instead

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

root.render(<h1>Hello, world</h1>);

```

</Sandpack>

If the entire content of your page was replaced by a "Hello, world!", everything worked! Keep reading.

<Note>

Integrating a modular JavaScript environment into an existing project for the first time can feel intimidating, but it's worth it! If you get stuck, try our [community resources](/community) or the [Vite Chat](https://chat.vitejs.dev/).

</Note>

Step 2: Render React components anywhere on the page
{/*step-2-render-react-components-anywhere-on-the-page*/}

In the previous step, you put this code at the top of your main file:

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

// Clear the existing HTML content

document.body.innerHTML = '<div id="app"></div>';

// Render your React component instead

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

root.render(<h1>Hello, world</h1>);

```

Of course, you don't actually want to clear the existing HTML content!

Delete this code.
```

Instead, you probably want to render your React components in specific places in your HTML. Open your HTML page (or the server templates that generate it) and add a unique [`id`](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/id) attribute to any tag, for example:

```html
<!-- ... somewhere in your html ... -->
<nav id="navigation"></nav>
<!-- ... more html ... -->
```

This lets you find that HTML element with [`document.getElementById`](https://developer.mozilla.org/en-US/docs/Web/API/Document/getElementById) and pass it to [`createRoot`](/reference/react-dom/client/createRoot) so that you can render your own React component inside:

<Sandpack>

```html index.html
<!DOCTYPE html>
<html>
<head><title>My app</title></head>
<body>
<p>This paragraph is a part of HTML.</p>
<nav id="navigation"></nav>
<p>This paragraph is also a part of HTML.</p>
</body>
</html>
```

```js index.js active
import { createRoot } from 'react-dom/client';

function NavigationBar() {
// TODO: Actually implement a navigation bar
return <h1>Hello from React!</h1>;
}

const domNode = document.getElementById('navigation');
const root = createRoot(domNode);
root.render(<NavigationBar />);
```

</Sandpack>

Notice how the original HTML content from `index.html` is preserved, but your own `NavigationBar` React component now appears inside the `<nav id="navigation">` from your HTML. Read the [`createRoot` usage documentation](/reference/react-dom/client/createRoot#rendering-a-page-partially-built-with-react) to learn more about rendering React components inside an existing HTML page.

When you adopt React in an existing project, it's common to start with small interactive components (like buttons), and then gradually keep "moving upwards" until eventually your entire page is built with React. If you ever reach that point, we recommend migrating to [a React framework](/learn/start-a-new-react-project) right after to get the most out of React.

Using React Native in an existing native mobile app {/*using-react-native-in-an-existing-native-mobile-app*/}

[React Native](https://reactnative.dev/) can also be integrated into existing native apps incrementally. If you have an existing native app for Android (Java or Kotlin) or iOS (Objective-C or Swift), [follow this guide](https://reactnative.dev/docs/integration-with-existing-apps) to add a React Native screen to it.

---

title: Managing State

<Intro>

As your application grows, it helps to be more intentional about how your state is organized and how the data flows between your components. Redundant or duplicate state is a common source of bugs. In this chapter, you'll learn how to structure your state well, how to keep your state update logic maintainable, and how to share state between distant components.

</Intro>

<YouWillLearn isChapter={true}>

* [How to think about UI changes as state changes](/learn/reacting-to-input-with-state)
* [How to structure state well](/learn/choosing-the-state-structure)
* [How to "lift state up" to share it between components](/learn/sharing-state-between-components)
* [How to control whether the state gets preserved or reset](/learn/preserving-and-resetting-state)
* [How to consolidate complex state logic in a function](/learn/extracting-state-logic-into-a-reducer)
* [How to pass information without "prop drilling"](/learn/passing-data-deeply-with-context)
* [How to scale state management as your app grows](/learn/scaling-up-with-reducer-and-context)

</YouWillLearn>

Reacting to input with state {/*reacting-to-input-with-state*/}

With React, you won't modify the UI from code directly. For example, you won't write commands like "disable the button", "enable the button", "show the success message", etc. Instead, you will describe

the UI you want to see for the different visual states of your component ("initial state", "typing state", "success state"), and then trigger the state changes in response to user input. This is similar to how designers think about UI.

Here is a quiz form built using React. Note how it uses the `status` state variable to determine whether to enable or disable the submit button, and whether to show the success message instead.

<Sandpack>

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

export default function Form() {
const [answer, setAnswer] = useState('');
const [error, setError] = useState(null);
const [status, setStatus] = useState('typing');

if (status === 'success') {
return <h1>That's right!</h1>
}

async function handleSubmit(e) {
e.preventDefault();
setStatus('submitting');
try {
await submitForm(answer);
setStatus('success');
} catch (err) {
setStatus('typing');
setError(err);
}
}

function handleTextareaChange(e) {
setAnswer(e.target.value);
}

return (
<>
<h2>City quiz</h2>
<p>
In which city is there a billboard that turns air into drinkable water?
```

```
</p>
<form onSubmit={handleSubmit}>
<textarea
value={answer}
onChange={handleTextareaChange}
disabled={status === 'submitting'}
/>
<br />
<button disabled={
answer.length === 0 ||
status === 'submitting'
}>
Submit
</button>
{error !== null &&
<p className="Error">
{error.message}
</p>
}
</form>
</>
);
}

function submitForm(answer) {
// Pretend it's hitting the network.
return new Promise((resolve, reject) => {
setTimeout(() => {
let shouldError = answer.toLowerCase() !== 'lima'
if (shouldError) {
reject(new Error('Good guess but a wrong answer. Try again!'));
} else {
resolve();
}
}, 1500);
});
```

}
```

```css
.Error { color: red; }
```

</Sandpack>

<LearnMore path="/learn/reacting-to-input-with-state">

Read **[Reacting to Input with State](/learn/reacting-to-input-with-state)** to learn how to approach interactions with a state-driven mindset.

</LearnMore>

Choosing the state structure {/*choosing-the-state-structure*/}

Structuring state well can make a difference between a component that is pleasant to modify and debug, and one that is a constant source of bugs. The most important principle is that state shouldn't contain redundant or duplicated information. If there's unnecessary state, it's easy to forget to update it, and introduce bugs!

For example, this form has a **redundant** `fullName` state variable:

<Sandpack>

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

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

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

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

return (

<>
<h2>Let's check you in</h2>
<label>
First name:{' '}
<input
value={firstName}
onChange={handleFirstNameChange}
/>
</label>
<label>
Last name:{' '}
<input
value={lastName}
onChange={handleLastNameChange}
/>
</label>

Your ticket will be issued to: {fullName}

</>
);
}
```

```css
label { display: block; margin-bottom: 5px; }
```

</Sandpack>

You can remove it and simplify the code by calculating `fullName` while the component is rendering:

<Sandpack>

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

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

```
const [lastName, setLastName] = useState('');

const fullName = firstName + ' ' + lastName;

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

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

return (
<>
<h2>Let's check you in</h2>
<label>
First name:{' '}
<input
value={firstName}
onChange={handleFirstNameChange}
/>
</label>
<label>
Last name:{' '}
<input
value={lastName}
onChange={handleLastNameChange}
/>
</label>
<p>
Your ticket will be issued to: <b>{fullName}</b>
</p>
</>
);
}
```

```css
label { display: block; margin-bottom: 5px; }
```

```

</Sandpack>

This might seem like a small change, but many bugs in React apps are fixed this way.

<LearnMore path="/learn/choosing-the-state-structure">

Read **[Choosing the State Structure](/learn/choosing-the-state-structure)** to learn how to design the state shape to avoid bugs.

</LearnMore>

## Sharing state between components {/*sharing-state-between-components*/}

Sometimes, you want the state of two components to always change together. To do it, remove state from both of them, move it to their closest common parent, and then pass it down to them via props. This is known as "lifting state up", and it's one of the most common things you will do writing React code.

In this example, only one panel should be active at a time. To achieve this, instead of keeping the active state inside each individual panel, the parent component holds the state and specifies the props for its children.

<Sandpack>

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

export default function Accordion() {
const [activeIndex, setActiveIndex] = useState(0);
return (
<>
<h2>Almaty, Kazakhstan</h2>
<Panel
title="About"
isActive={activeIndex === 0}
onShow={() => setActiveIndex(0)}
>
With a population of about 2 million, Almaty is Kazakhstan's largest city. From 1929 to 1997, it was its capital city.
</Panel>
<Panel
title="Etymology"
isActive={activeIndex === 1}
```
```

```
onShow={() => setActiveIndex(1)}
>
The name comes from <span lang="kk-KZ">■■■■</span>, the Kazakh word for "apple" and is often translated as "full of apples". In fact, the region surrounding Almaty is thought to be the ancestral home of the apple, and the wild <i lang="la">Malus sieversii</i> is considered a likely candidate for the ancestor of the modern domestic apple.
</Panel>
</>
);
}

function Panel({
title,
children,
isActive,
onShow
}) {
return (
<section className="panel">
<h3>{title}</h3>
{isActive ? (
<p>{children}</p>
) : (
<button onClick={onShow}>
Show
</button>
)}
</section>
);
}
```

```css
h3, p { margin: 5px 0px; }
.panel {
padding: 10px;
border: 1px solid #aaa;
}
```

```

</Sandpack>

<LearnMore path="/learn/sharing-state-between-components">

Read **[Sharing State Between Components](/learn/sharing-state-between-components)** to learn how to lift state up and keep components in sync.

</LearnMore>

## Preserving and resetting state {/*preserving-and-resetting-state*/}

When you re-render a component, React needs to decide which parts of the tree to keep (and update), and which parts to discard or re-create from scratch. In most cases, React's automatic behavior works well enough. By default, React preserves the parts of the tree that "match up" with the previously rendered component tree.

However, sometimes this is not what you want. In this chat app, typing a message and then switching the recipient does not reset the input. This can make the user accidentally send a message to the wrong person:

<Sandpack>

```js App.js
import { useState } from 'react';
import Chat from './Chat.js';
import ContactList from './ContactList.js';

export default function Messenger() {
const [to, setTo] = useState(contacts[0]);
return (
<div>
<ContactList
contacts={contacts}
selectedContact={to}
onSelect={contact => setTo(contact)}
/>
<Chat contact={to} />
</div>
)
}

const contacts = [
{ name: 'Taylor', email: 'taylor@mail.com' },
```
```

```
{ name: 'Alice', email: 'alice@mail.com' },
{ name: 'Bob', email: 'bob@mail.com' }
];
```

```js ContactList.js
export default function ContactList({
selectedContact,
contacts,
onSelect
}) {
return (
<section className="contact-list">
<ul>
{contacts.map(contact =>
<li key={contact.email}>
<button onClick={() => {
onSelect(contact);
}}>
{contact.name}
</button>
</li>
)}
</ul>
</section>
);
}
```

```js Chat.js
import { useState } from 'react';

export default function Chat({ contact }) {
const [text, setText] = useState('');
return (
<section className="chat">
<textarea
```

value={text}
placeholder={'Chat to ' + contact.name}
onChange={e => setText(e.target.value)}
/>

<button>Send to {contact.email}</button>
</section>
);
}
```

```css
.chat, .contact-list {
float: left;
margin-bottom: 20px;
}
ul, li {
list-style: none;
margin: 0;
padding: 0;
}
li button {
width: 100px;
padding: 10px;
margin-right: 10px;
}
textarea {
height: 150px;
}
```

</Sandpack>

React lets you override the default behavior, and *force* a component to reset its state by passing it a different `key`, like `<Chat key={email} />`. This tells React that if the recipient is different, it should be considered a *different* `Chat` component that needs to be re-created from scratch with the new data (and UI like inputs). Now switching between the recipients resets the input field--even though you render the same component.
```

<Sandpack>

```js App.js
import { useState } from 'react';
import Chat from './Chat.js';
import ContactList from './ContactList.js';

export default function Messenger() {
const [to, setTo] = useState(contacts[0]);
return (
<div>
<ContactList
contacts={contacts}
selectedContact={to}
onSelect={contact => setTo(contact)}
/>
<Chat key={to.email} contact={to} />
</div>
)
}

const contacts = [
{ name: 'Taylor', email: 'taylor@mail.com' },
{ name: 'Alice', email: 'alice@mail.com' },
{ name: 'Bob', email: 'bob@mail.com' }
];
```

```js ContactList.js
export default function ContactList({
selectedContact,
contacts,
onSelect
}) {
return (
<section className="contact-list">
<ul>
{contacts.map(contact =>
```

```
<li key={contact.email}>
<button onClick={() => {
onSelect(contact);
}}>
{contact.name}
</button>
</li>
)}
</ul>
</section>
);
}
```

```js Chat.js
import { useState } from 'react';

export default function Chat({ contact }) {
const [text, setText] = useState('');
return (
<section className="chat">
<textarea
value={text}
placeholder={'Chat to ' + contact.name}
onChange={e => setText(e.target.value)}
/>
<br />
<button>Send to {contact.email}</button>
</section>
);
}
```

```css
.chat, .contact-list {
float: left;
margin-bottom: 20px;
```

```
}
ul, li {
list-style: none;
margin: 0;
padding: 0;
}
li button {
width: 100px;
padding: 10px;
margin-right: 10px;
}
textarea {
height: 150px;
}
```

</Sandpack>

<LearnMore path="/learn/preserving-and-resetting-state">

Read **[Preserving and Resetting State](/learn/preserving-and-resetting-state)** to learn the lifetime of state and how to control it.

</LearnMore>

Extracting state logic into a reducer {/*extracting-state-logic-into-a-reducer*/}

Components with many state updates spread across many event handlers can get overwhelming. For these cases, you can consolidate all the state update logic outside your component in a single function, called "reducer". Your event handlers become concise because they only specify the user "actions". At the bottom of the file, the reducer function specifies how the state should update in response to each action!

<Sandpack>

```js App.js
import { useReducer } from 'react';
import AddTask from './AddTask.js';
import TaskList from './TaskList.js';

export default function TaskApp() {
const [tasks, dispatch] = useReducer(
tasksReducer,
```

```
initialTasks
);

function handleAddTask(text) {
dispatch({
type: 'added',
id: nextId++,
text: text,
});
}

function handleChangeTask(task) {
dispatch({
type: 'changed',
task: task
});
}

function handleDeleteTask(taskId) {
dispatch({
type: 'deleted',
id: taskId
});
}

return (
<>
<h1>Prague itinerary</h1>
<AddTask
onAddTask={handleAddTask}
/>
<TaskList
tasks={tasks}
onChangeTask={handleChangeTask}
onDeleteTask={handleDeleteTask}
/>
</>
);
```

```
}

function tasksReducer(tasks, action) {
switch (action.type) {
case 'added': {
return [...tasks, {
id: action.id,
text: action.text,
done: false
}];
}
case 'changed': {
return tasks.map(t => {
if (t.id === action.task.id) {
return action.task;
} else {
return t;
}
});
}
case 'deleted': {
return tasks.filter(t => t.id !== action.id);
}
default: {
throw Error('Unknown action: ' + action.type);
}
}
}

let nextId = 3;
const initialTasks = [
{ id: 0, text: 'Visit Kafka Museum', done: true },
{ id: 1, text: 'Watch a puppet show', done: false },
{ id: 2, text: 'Lennon Wall pic', done: false }
];
```

```js AddTask.js hidden
import { useState } from 'react';

export default function AddTask({ onAddTask }) {
const [text, setText] = useState('');
return (
<>
<input
placeholder="Add task"
value={text}
onChange={e => setText(e.target.value)}
/>
<button onClick={() => {
setText('');
onAddTask(text);
}}>Add</button>
</>
)
}
```

```js TaskList.js hidden
import { useState } from 'react';

export default function TaskList({
tasks,
onChangeTask,
onDeleteTask
}) {
return (
<ul>
{tasks.map(task => (
<li key={task.id}>
<Task
task={task}
onChange={onChangeTask}
onDelete={onDeleteTask}
```

```
/>
</li>
))}
</ul>
);
}

function Task({ task, onChange, onDelete }) {
const [isEditing, setIsEditing] = useState(false);
let taskContent;
if (isEditing) {
taskContent = (
<>
<input
value={task.text}
onChange={e => {
onChange({
...task,
text: e.target.value
});
}} />
<button onClick={() => setIsEditing(false)}>
Save
</button>
</>
);
} else {
taskContent = (
<>
{task.text}
<button onClick={() => setIsEditing(true)}>
Edit
</button>
</>
);
}
```

```
return (
<label>
<input
type="checkbox"
checked={task.done}
onChange={e => {
onChange({
...task,
done: e.target.checked
});
}}
/>
{taskContent}
<button onClick={() => onDelete(task.id)}>
Delete
</button>
</label>
);
}
```

```css
button { margin: 5px; }
li { list-style-type: none; }
ul, li { margin: 0; padding: 0; }
```

</Sandpack>

<LearnMore path="/learn/extracting-state-logic-into-a-reducer">

Read **[Extracting State Logic into a Reducer](/learn/extracting-state-logic-into-a-reducer)** to learn how to consolidate logic in the reducer function.

</LearnMore>

Passing data deeply with context {/*passing-data-deeply-with-context*/}

Usually, you will pass information from a parent component to a child component via props. But passing props can become inconvenient if you need to pass some prop through many components, or if many components need the same information. Context lets the parent component make some information

available to any component in the tree below it—no matter how deep it is—without passing it explicitly through props.

Here, the `Heading` component determines its heading level by "asking" the closest `Section` for its level. Each `Section` tracks its own level by asking the parent `Section` and adding one to it. Every `Section` provides information to all components below it without passing props--it does that through context.

<Sandpack>

```js
import Heading from './Heading.js';
import Section from './Section.js';

export default function Page() {
return (
<Section>
<Heading>Title</Heading>
<Section>
<Heading>Heading</Heading>
<Heading>Heading</Heading>
<Heading>Heading</Heading>
<Section>
<Heading>Sub-heading</Heading>
<Heading>Sub-heading</Heading>
<Heading>Sub-heading</Heading>
<Section>
<Heading>Sub-sub-heading</Heading>
<Heading>Sub-sub-heading</Heading>
<Heading>Sub-sub-heading</Heading>
</Section>
</Section>
</Section>
</Section>
);
}
```

```js Section.js
import { useContext } from 'react';
```

```
import { LevelContext } from './LevelContext.js';

export default function Section({ children }) {
const level = useContext(LevelContext);
return (
<section className="section">
<LevelContext.Provider value={level + 1}>
{children}
</LevelContext.Provider>
</section>
);
}
```

```js Heading.js
import { useContext } from 'react';
import { LevelContext } from './LevelContext.js';

export default function Heading({ children }) {
const level = useContext(LevelContext);
switch (level) {
case 0:
throw Error('Heading must be inside a Section!');
case 1:
return <h1>{children}</h1>;
case 2:
return <h2>{children}</h2>;
case 3:
return <h3>{children}</h3>;
case 4:
return <h4>{children}</h4>;
case 5:
return <h5>{children}</h5>;
case 6:
return <h6>{children}</h6>;
default:
throw Error('Unknown level: ' + level);
```

}

}

```

```js LevelContext.js
import { createContext } from 'react';

export const LevelContext = createContext(0);
```

```css
.section {
padding: 10px;
margin: 5px;
border-radius: 5px;
border: 1px solid #aaa;
}
```

</Sandpack>

<LearnMore path="/learn/passing-data-deeply-with-context">

Read **[Passing Data Deeply with Context](/learn/passing-data-deeply-with-context)** to learn about using context as an alternative to passing props.

</LearnMore>

Scaling up with reducer and context {/*scaling-up-with-reducer-and-context*/}

Reducers let you consolidate a component's state update logic. Context lets you pass information deep down to other components. You can combine reducers and context together to manage state of a complex screen.

With this approach, a parent component with complex state manages it with a reducer. Other components anywhere deep in the tree can read its state via context. They can also dispatch actions to update that state.

<Sandpack>

```js App.js
import AddTask from './AddTask.js';
import TaskList from './TaskList.js';
import { TasksProvider } from './TasksContext.js';

export default function TaskApp() {

```
return (
<TasksProvider>
<h1>Day off in Kyoto</h1>
<AddTask />
<TaskList />
</TasksProvider>
);
}
```

```js TasksContext.js
import { createContext, useContext, useReducer } from 'react';

const TasksContext = createContext(null);
const TasksDispatchContext = createContext(null);

export function TasksProvider({ children }) {
const [tasks, dispatch] = useReducer(
tasksReducer,
initialTasks
);

return (
<TasksContext.Provider value={tasks}>
<TasksDispatchContext.Provider
value={dispatch}
>
{children}
</TasksDispatchContext.Provider>
</TasksContext.Provider>
);
}

export function useTasks() {
return useContext(TasksContext);
}

export function useTasksDispatch() {
return useContext(TasksDispatchContext);
```

```
}

function tasksReducer(tasks, action) {
switch (action.type) {
case 'added': {
return [...tasks, {
id: action.id,
text: action.text,
done: false
}];
}
case 'changed': {
return tasks.map(t => {
if (t.id === action.task.id) {
return action.task;
} else {
return t;
}
});
}
case 'deleted': {
return tasks.filter(t => t.id !== action.id);
}
default: {
throw Error('Unknown action: ' + action.type);
}
}
}

const initialTasks = [
{ id: 0, text: 'Philosopher’s Path', done: true },
{ id: 1, text: 'Visit the temple', done: false },
{ id: 2, text: 'Drink matcha', done: false }
];
```

```js AddTask.js
```

```
import { useState, useContext } from 'react';
import { useTasksDispatch } from './TasksContext.js';

export default function AddTask({ onAddTask }) {
const [text, setText] = useState('');
const dispatch = useTasksDispatch();
return (
<>
<input
placeholder="Add task"
value={text}
onChange={e => setText(e.target.value)}
/>
<button onClick={() => {
setText('');
dispatch({
type: 'added',
id: nextId++,
text: text,
});
}}>Add</button>
</>
);
}

let nextId = 3;
```

```js TaskList.js
import { useState, useContext } from 'react';
import { useTasks, useTasksDispatch } from './TasksContext.js';

export default function TaskList() {
const tasks = useTasks();
return (
<ul>
{tasks.map(task => (
<li key={task.id}>
```

```
<Task task={task} />
</li>
))}
</ul>
);
}

function Task({ task }) {
const [isEditing, setIsEditing] = useState(false);
const dispatch = useTasksDispatch();
let taskContent;
if (isEditing) {
taskContent = (
<>
<input
value={task.text}
onChange={e => {
dispatch({
type: 'changed',
task: {
...task,
text: e.target.value
}
});
}} />
<button onClick={() => setIsEditing(false)}>
Save
</button>
</>
);
} else {
taskContent = (
<>
{task.text}
<button onClick={() => setIsEditing(true)}>
Edit
```

```
</button>
</>
);
}
return (
<label>
<input
type="checkbox"
checked={task.done}
onChange={e => {
dispatch({
type: 'changed',
task: {
...task,
done: e.target.checked
}
});
}}
/>
{taskContent}
<button onClick={() => {
dispatch({
type: 'deleted',
id: task.id
});
}}>
Delete
</button>
</label>
);
}
```

```css
button { margin: 5px; }
li { list-style-type: none; }
```

ul, li { margin: 0; padding: 0; }
```

</Sandpack>

<LearnMore path="/learn/scaling-up-with-reducer-and-context">

Read **[Scaling Up with Reducer and Context](/learn/scaling-up-with-reducer-and-context)** to learn how state management scales in a growing app.

</LearnMore>

## What's next? {/*whats-next*/}

Head over to [Reacting to Input with State](/learn/reacting-to-input-with-state) to start reading this chapter page by page!

Or, if you're already familiar with these topics, why not read about [Escape Hatches](/learn/escape-hatches)?

---

title: Reacting to Input with State

---

<Intro>

React provides a declarative way to manipulate the UI. Instead of manipulating individual pieces of the UI directly, you describe the different states that your component can be in, and switch between them in response to the user input. This is similar to how designers think about the UI.

</Intro>

<YouWillLearn>

* How declarative UI programming differs from imperative UI programming
* How to enumerate the different visual states your component can be in
* How to trigger the changes between the different visual states from code

</YouWillLearn>

## How declarative UI compares to imperative {/*how-declarative-ui-compares-to-imperative*/}

When you design UI interactions, you probably think about how the UI *changes* in response to user actions. Consider a form that lets the user submit an answer:

* When you type something into the form, the "Submit" button **becomes enabled.**
* When you press "Submit", both the form and the button **become disabled,** and a spinner **appears.**
* If the network request succeeds, the form **gets hidden,** and the "Thank you" message **appears.**
```

* If the network request fails, an error message **appears,** and the form **becomes enabled** again.

In **imperative programming,** the above corresponds directly to how you implement interaction. You have to write the exact instructions to manipulate the UI depending on what just happened. Here's another way to think about this: imagine riding next to someone in a car and telling them turn by turn where to go.

<Illustration src="/images/docs/illustrations/i_imperative-ui-programming.png" alt="In a car driven by an anxious-looking person representing JavaScript, a passenger orders the driver to execute a sequence of complicated turn by turn navigations." />

They don't know where you want to go, they just follow your commands. (And if you get the directions wrong, you end up in the wrong place!) It's called *imperative* because you have to "command" each element, from the spinner to the button, telling the computer *how* to update the UI.

In this example of imperative UI programming, the form is built *without* React. It only uses the browser [DOM](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model):

<Sandpack>

```js index.js active
async function handleFormSubmit(e) {
e.preventDefault();
disable(textarea);
disable(button);
show(loadingMessage);
hide(errorMessage);
try {
await submitForm(textarea.value);
show(successMessage);
hide(form);
} catch (err) {
show(errorMessage);
errorMessage.textContent = err.message;
} finally {
hide(loadingMessage);
enable(textarea);
enable(button);
}
}

function handleTextareaChange() {
if (textarea.value.length === 0) {
```

```
disable(button);
} else {
enable(button);
}
}

function hide(el) {
el.style.display = 'none';
}

function show(el) {
el.style.display = '';
}

function enable(el) {
el.disabled = false;
}

function disable(el) {
el.disabled = true;
}

function submitForm(answer) {
// Pretend it's hitting the network.
return new Promise((resolve, reject) => {
setTimeout(() => {
if (answer.toLowerCase() == 'istanbul') {
resolve();
} else {
reject(new Error('Good guess but a wrong answer. Try again!'));
}
}, 1500);
});
}

let form = document.getElementById('form');
let textarea = document.getElementById('textarea');
let button = document.getElementById('button');
let loadingMessage = document.getElementById('loading');
```

```
let errorMessage = document.getElementById('error');
let successMessage = document.getElementById('success');
form.onsubmit = handleFormSubmit;
textarea.oninput = handleTextareaChange;
```

```js sandbox.config.json hidden
{
"hardReloadOnChange": true
}
```

```html public/index.html
<form id="form">
<h2>City quiz</h2>
<p>
What city is located on two continents?
</p>
<textarea id="textarea"></textarea>
<br />
<button id="button" disabled>Submit</button>
<p id="loading" style="display: none">Loading...</p>
<p id="error" style="display: none; color: red;"></p>
</form>
<h1 id="success" style="display: none">That's right!</h1>

<style>
* { box-sizing: border-box; }
body { font-family: sans-serif; margin: 20px; padding: 0; }
</style>
```

</Sandpack>

Manipulating the UI imperatively works well enough for isolated examples, but it gets exponentially more difficult to manage in more complex systems. Imagine updating a page full of different forms like this one. Adding a new UI element or a new interaction would require carefully checking all existing code to make sure you haven't introduced a bug (for example, forgetting to show or hide something).

React was built to solve this problem.

In React, you don't directly manipulate the UI--meaning you don't enable, disable, show, or hide components directly. Instead, you **declare what you want to show,** and React figures out how to update the UI. Think of getting into a taxi and telling the driver where you want to go instead of telling them exactly where to turn. It's the driver's job to get you there, and they might even know some shortcuts you haven't considered!

<Illustration src="/images/docs/illustrations/i_declarative-ui-programming.png" alt="In a car driven by React, a passenger asks to be taken to a specific place on the map. React figures out how to do that." />

Thinking about UI declaratively {/*thinking-about-ui-declaratively*/}

You've seen how to implement a form imperatively above. To better understand how to think in React, you'll walk through reimplementing this UI in React below:

1. **Identify** your component's different visual states
2. **Determine** what triggers those state changes
3. **Represent** the state in memory using `useState`
4. **Remove** any non-essential state variables
5. **Connect** the event handlers to set the state

Step 1: Identify your component's different visual states {/*step-1-identify-your-components-different-visual-states*/}

In computer science, you may hear about a ["state machine"](https://en.wikipedia.org/wiki/Finite-state_machine) being in one of several “states”. If you work with a designer, you may have seen mockups for different "visual states". React stands at the intersection of design and computer science, so both of these ideas are sources of inspiration.

First, you need to visualize all the different "states" of the UI the user might see:

* **Empty**: Form has a disabled "Submit" button.
* **Typing**: Form has an enabled "Submit" button.
* **Submitting**: Form is completely disabled. Spinner is shown.
* **Success**: "Thank you" message is shown instead of a form.
* **Error**: Same as Typing state, but with an extra error message.

Just like a designer, you'll want to "mock up" or create "mocks" for the different states before you add logic. For example, here is a mock for just the visual part of the form. This mock is controlled by a prop called `status` with a default value of `'empty'`:

<Sandpack>

```js
export default function Form({
status = 'empty'
}) {
```

```
if (status === 'success') {
return <h1>That's right!</h1>
}
return (
<>
<h2>City quiz</h2>
<p>
In which city is there a billboard that turns air into drinkable water?
</p>
<form>
<textarea />
<br />
<button>
Submit
</button>
</form>
</>
)
}
```

</Sandpack>

You could call that prop anything you like, the naming is not important. Try editing `status = 'empty'` to `status = 'success'` to see the success message appear. Mocking lets you quickly iterate on the UI before you wire up any logic. Here is a more fleshed out prototype of the same component, still "controlled" by the `status` prop:

<Sandpack>

```js
export default function Form({
// Try 'submitting', 'error', 'success':
status = 'empty'
}) {
if (status === 'success') {
return <h1>That's right!</h1>
}
return (
```

```
<>
<h2>City quiz</h2>
<p>
In which city is there a billboard that turns air into drinkable water?
</p>
<form>
<textarea disabled={
status === 'submitting'
} />
<br />
<button disabled={
status === 'empty' ||
status === 'submitting'
}>
Submit
</button>
{status === 'error' &&
<p className="Error">
Good guess but a wrong answer. Try again!
</p>
}
</form>
</>
);
}
```

```css
.Error { color: red; }
```

</Sandpack>

<DeepDive>

Displaying many visual states at once {/*displaying-many-visual-states-at-once*/}

If a component has a lot of visual states, it can be convenient to show them all on one page:

<Sandpack>

```js App.js active
import Form from './Form.js';

let statuses = [
'empty',
'typing',
'submitting',
'success',
'error',
];

export default function App() {
return (
<>
{statuses.map(status => (
<section key={status}>
<h4>Form ({status}):</h4>
<Form status={status} />
</section>
))}
</>
);
}
```

```js Form.js
export default function Form({ status }) {
if (status === 'success') {
return <h1>That's right!</h1>
}
return (
<form>
<textarea disabled={
status === 'submitting'
} />
<br />
```

<button disabled={

status === 'empty' ||

status === 'submitting'

}>

Submit

</button>

{status === 'error' &&

Good guess but a wrong answer. Try again!

}

</form>

);

}

```

```css

section { border-bottom: 1px solid #aaa; padding: 20px; }

h4 { color: #222; }

body { margin: 0; }

.Error { color: red; }

```

</Sandpack>

Pages like this are often called "living styleguides" or "storybooks".

</DeepDive>

Step 2: Determine what triggers those state changes
{/*step-2-determine-what-triggers-those-state-changes*/}

You can trigger state updates in response to two kinds of inputs:

* **Human inputs,** like clicking a button, typing in a field, navigating a link.

* **Computer inputs,** like a network response arriving, a timeout completing, an image loading.

<IllustrationBlock>

<Illustration caption="Human inputs" alt="A finger." src="/images/docs/illustrations/i_inputs1.png" />

<Illustration caption="Computer inputs" alt="Ones and zeroes."
src="/images/docs/illustrations/i_inputs2.png" />
```

</IllustrationBlock>

In both cases, **you must set [state variables](/learn/state-a-components-memory#anatomy-of-usestate) to update the UI.** For the form you're developing, you will need to change state in response to a few different inputs:

* **Changing the text input** (human) should switch it from the *Empty* state to the *Typing* state or back, depending on whether the text box is empty or not.

* **Clicking the Submit button** (human) should switch it to the *Submitting* state.

* **Successful network response** (computer) should switch it to the *Success* state.

* **Failed network response** (computer) should switch it to the *Error* state with the matching error message.

<Note>

Notice that human inputs often require [event handlers](/learn/responding-to-events)!

</Note>

To help visualize this flow, try drawing each state on paper as a labeled circle, and each change between two states as an arrow. You can sketch out many flows this way and sort out bugs long before implementation.

<DiagramGroup>

<Diagram name="responding_to_input_flow" height={350} width={688} alt="Flow chart moving left to right with 5 nodes. The first node labeled 'empty' has one edge labeled 'start typing' connected to a node labeled 'typing'. That node has one edge labeled 'press submit' connected to a node labeled 'submitting', which has two edges. The left edge is labeled 'network error' connecting to a node labeled 'error'. The right edge is labeled 'network success' connecting to a node labeled 'success'.">

Form states

</Diagram>

</DiagramGroup>

Step 3: Represent the state in memory with `useState` {/*step-3-represent-the-state-in-memory-with-usestate*/}

Next you'll need to represent the visual states of your component in memory with [`useState`.](/reference/react/useState) Simplicity is key: each piece of state is a "moving piece", and **you want as few "moving pieces" as possible.** More complexity leads to more bugs!

Start with the state that *absolutely must* be there. For example, you'll need to store the `answer` for the input, and the `error` (if it exists) to store the last error:

```js
const [answer, setAnswer] = useState('');
const [error, setError] = useState(null);
```

```

Then, you'll need a state variable representing which one of the visual states that you want to display. There's usually more than a single way to represent that in memory, so you'll need to experiment with it.

If you struggle to think of the best way immediately, start by adding enough state that you're *definitely* sure that all the possible visual states are covered:

```js
const [isEmpty, setIsEmpty] = useState(true);
const [isTyping, setIsTyping] = useState(false);
const [isSubmitting, setIsSubmitting] = useState(false);
const [isSuccess, setIsSuccess] = useState(false);
const [isError, setIsError] = useState(false);
```

Your first idea likely won't be the best, but that's ok--refactoring state is a part of the process!

Step 4: Remove any non-essential state variables
{/*step-4-remove-any-non-essential-state-variables*/}

You want to avoid duplication in the state content so you're only tracking what is essential. Spending a little time on refactoring your state structure will make your components easier to understand, reduce duplication, and avoid unintended meanings. Your goal is to **prevent the cases where the state in memory doesn't represent any valid UI that you'd want a user to see.** (For example, you never want to show an error message and disable the input at the same time, or the user won't be able to correct the error!)

Here are some questions you can ask about your state variables:

* **Does this state cause a paradox?** For example, `isTyping` and `isSubmitting` can't both be `true`. A paradox usually means that the state is not constrained enough. There are four possible combinations of two booleans, but only three correspond to valid states. To remove the "impossible" state, you can combine these into a `status` that must be one of three values: `'typing'`, `'submitting'`, or `'success'`.
* **Is the same information available in another state variable already?** Another paradox: `isEmpty` and `isTyping` can't be `true` at the same time. By making them separate state variables, you risk them going out of sync and causing bugs. Fortunately, you can remove `isEmpty` and instead check `answer.length === 0`.
* **Can you get the same information from the inverse of another state variable?** `isError` is not needed because you can check `error !== null` instead.

After this clean-up, you're left with 3 (down from 7!) *essential* state variables:

```js
const [answer, setAnswer] = useState('');
const [error, setError] = useState(null);
```
```

const [status, setStatus] = useState('typing'); // 'typing', 'submitting', or 'success'
```

You know they are essential, because you can't remove any of them without breaking the functionality.

<DeepDive>

#### Eliminating “impossible” states with a reducer {/*eliminating-impossible-states-with-a-reducer*/}

These three variables are a good enough representation of this form's state. However, there are still some intermediate states that don't fully make sense. For example, a non-null `error` doesn't make sense when `status` is `'success'`. To model the state more precisely, you can [extract it into a reducer.](/learn/extracting-state-logic-into-a-reducer) Reducers let you unify multiple state variables into a single object and consolidate all the related logic!

</DeepDive>

### Step 5: Connect the event handlers to set state {/*step-5-connect-the-event-handlers-to-set-state*/}

Lastly, create event handlers that update the state. Below is the final form, with all event handlers wired up:

<Sandpack>

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

export default function Form() {
const [answer, setAnswer] = useState('');
const [error, setError] = useState(null);
const [status, setStatus] = useState('typing');

if (status === 'success') {
return <h1>That's right!</h1>
}

async function handleSubmit(e) {
e.preventDefault();
setStatus('submitting');
try {
await submitForm(answer);
setStatus('success');
} catch (err) {
setStatus('typing');
setError(err);
```
```

```
}
}

function handleTextareaChange(e) {
setAnswer(e.target.value);
}

return (
<>
<h2>City quiz</h2>
<p>
In which city is there a billboard that turns air into drinkable water?
</p>
<form onSubmit={handleSubmit}>
<textarea
value={answer}
onChange={handleTextareaChange}
disabled={status === 'submitting'}
/>
<br />
<button disabled={
answer.length === 0 ||
status === 'submitting'
}>
Submit
</button>
{error !== null &&
<p className="Error">
{error.message}
</p>
}
</form>
</>
);
}

function submitForm(answer) {
```

```
// Pretend it's hitting the network.
return new Promise((resolve, reject) => {
setTimeout(() => {
let shouldError = answer.toLowerCase() !== 'lima'
if (shouldError) {
reject(new Error('Good guess but a wrong answer. Try again!'));
} else {
resolve();
}
}, 1500);
});
}
```

```css
.Error { color: red; }
```

</Sandpack>

Although this code is longer than the original imperative example, it is much less fragile. Expressing all interactions as state changes lets you later introduce new visual states without breaking existing ones. It also lets you change what should be displayed in each state without changing the logic of the interaction itself.

<Recap>

* Declarative programming means describing the UI for each visual state rather than micromanaging the UI (imperative).
* When developing a component:
1. Identify all its visual states.
2. Determine the human and computer triggers for state changes.
3. Model the state with `useState`.
4. Remove non-essential state to avoid bugs and paradoxes.
5. Connect the event handlers to set state.

</Recap>

<Challenges>

Add and remove a CSS class {/*add-and-remove-a-css-class*/}

Make it so that clicking on the picture *removes* the `background--active` CSS class from the outer `<div>`, but *adds* the `picture--active` class to the `<img>`. Clicking the background again should restore the original CSS classes.

Visually, you should expect that clicking on the picture removes the purple background and highlights the picture border. Clicking outside the picture highlights the background, but removes the picture border highlight.

<Sandpack>

```js
export default function Picture() {
return (
<div className="background background--active">
<img
className="picture"
alt="Rainbow houses in Kampung Pelangi, Indonesia"
src="https://i.imgur.com/5qwVYb1.jpeg"
/>
</div>
);
}
```

```css
body { margin: 0; padding: 0; height: 250px; }

.background {
width: 100vw;
height: 100vh;
display: flex;
justify-content: center;
align-items: center;
background: #eee;
}

.background--active {
background: #a6b5ff;
}

.picture {
```

width: 200px;

height: 200px;

border-radius: 10px;

}

.picture--active {

border: 5px solid #a6b5ff;

}

```

</Sandpack>

<Solution>

This component has two visual states: when the image is active, and when the image is inactive:

* When the image is active, the CSS classes are `background` and `picture picture--active`.

* When the image is inactive, the CSS classes are `background background--active` and `picture`.

A single boolean state variable is enough to remember whether the image is active. The original task was to remove or add CSS classes. However, in React you need to *describe* what you want to see rather than *manipulate* the UI elements. So you need to calculate both CSS classes based on the current state. You also need to [stop the propagation](/learn/responding-to-events#stopping-propagation) so that clicking the image doesn't register as a click on the background.

Verify that this version works by clicking the image and then outside of it:

<Sandpack>

```js

import { useState } from 'react';

export default function Picture() {

const [isActive, setIsActive] = useState(false);

let backgroundClassName = 'background';

let pictureClassName = 'picture';

if (isActive) {

pictureClassName += ' picture--active';

} else {

backgroundClassName += ' background--active';

}

return (

```
<div
className={backgroundClassName}
onClick={() => setIsActive(false)}
>
<img
onClick={e => {
e.stopPropagation();
setIsActive(true);
}}
className={pictureClassName}
alt="Rainbow houses in Kampung Pelangi, Indonesia"
src="https://i.imgur.com/5qwVYb1.jpeg"
/>
</div>
);
}
```

```css
body { margin: 0; padding: 0; height: 250px; }

.background {
width: 100vw;
height: 100vh;
display: flex;
justify-content: center;
align-items: center;
background: #eee;
}

.background--active {
background: #a6b5ff;
}

.picture {
width: 200px;
height: 200px;
border-radius: 10px;
```

````
border: 5px solid transparent;
}

.picture--active {
border: 5px solid #a6b5ff;
}
```

</Sandpack>

Alternatively, you could return two separate chunks of JSX:

<Sandpack>

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

export default function Picture() {
const [isActive, setIsActive] = useState(false);
if (isActive) {
return (
<div
className="background"
onClick={() => setIsActive(false)}
>
<img
className="picture picture--active"
alt="Rainbow houses in Kampung Pelangi, Indonesia"
src="https://i.imgur.com/5qwVYb1.jpeg"
onClick={e => e.stopPropagation()}
/>
</div>
);
}
return (
<div className="background background--active">
<img
className="picture"
alt="Rainbow houses in Kampung Pelangi, Indonesia"
````

src="https://i.imgur.com/5qwVYb1.jpeg"
onClick={() => setIsActive(true)}
/>

);
}
```

```css
body { margin: 0; padding: 0; height: 250px; }

.background {
width: 100vw;
height: 100vh;
display: flex;
justify-content: center;
align-items: center;
background: #eee;
}

.background--active {
background: #a6b5ff;
}

.picture {
width: 200px;
height: 200px;
border-radius: 10px;
border: 5px solid transparent;
}

.picture--active {
border: 5px solid #a6b5ff;
}
```

</Sandpack>

Keep in mind that if two different JSX chunks describe the same tree, their nesting (first `<div>` → first `<img>`) has to line up. Otherwise, toggling `isActive` would recreate the whole tree below and [reset its state.](/learn/preserving-and-resetting-state) This is why, if a similar JSX tree gets returned in both
```

cases, it is better to write them as a single piece of JSX.

</Solution>

Profile editor {/*profile-editor*/}

Here is a small form implemented with plain JavaScript and DOM. Play with it to understand its behavior:

<Sandpack>

```js index.js active
function handleFormSubmit(e) {
e.preventDefault();
if (editButton.textContent === 'Edit Profile') {
editButton.textContent = 'Save Profile';
hide(firstNameText);
hide(lastNameText);
show(firstNameInput);
show(lastNameInput);
} else {
editButton.textContent = 'Edit Profile';
hide(firstNameInput);
hide(lastNameInput);
show(firstNameText);
show(lastNameText);
}
}

function handleFirstNameChange() {
firstNameText.textContent = firstNameInput.value;
helloText.textContent = (
'Hello ' +
firstNameInput.value + ' ' +
lastNameInput.value + '!'
);
}

function handleLastNameChange() {
lastNameText.textContent = lastNameInput.value;
```

```
helloText.textContent = (
'Hello ' +
firstNameInput.value + ' ' +
lastNameInput.value + '!'
);
}

function hide(el) {
el.style.display = 'none';
}

function show(el) {
el.style.display = '';
}

let form = document.getElementById('form');
let editButton = document.getElementById('editButton');
let firstNameInput = document.getElementById('firstNameInput');
let firstNameText = document.getElementById('firstNameText');
let lastNameInput = document.getElementById('lastNameInput');
let lastNameText = document.getElementById('lastNameText');
let helloText = document.getElementById('helloText');
form.onsubmit = handleFormSubmit;
firstNameInput.oninput = handleFirstNameChange;
lastNameInput.oninput = handleLastNameChange;
```

```js sandbox.config.json hidden
{
"hardReloadOnChange": true
}
```

```html public/index.html
<form id="form">
<label>
First name:
<b id="firstNameText">Jane</b>
<input
```

id="firstNameInput"

value="Jane"

style="display: none">

</label>

<label>

Last name:

Jacobs

<input

id="lastNameInput"

value="Jacobs"

style="display: none">

</label>

<button type="submit" id="editButton">Edit Profile</button>

Hello, Jane Jacobs!

</form>

<style>

* { box-sizing: border-box; }

body { font-family: sans-serif; margin: 20px; padding: 0; }

label { display: block; margin-bottom: 20px; }

</style>

```

</Sandpack>

This form switches between two modes: in the editing mode, you see the inputs, and in the viewing mode, you only see the result. The button label changes between "Edit" and "Save" depending on the mode you're in. When you change the inputs, the welcome message at the bottom updates in real time.

Your task is to reimplement it in React in the sandbox below. For your convenience, the markup was already converted to JSX, but you'll need to make it show and hide the inputs like the original does.

Make sure that it updates the text at the bottom, too!

<Sandpack>

```js

export default function EditProfile() {

return (

<form>

<label>

First name:{' '}

Jane

<input />

</label>

<label>

Last name:{' '}

Jacobs

<input />

</label>

<button type="submit">

Edit Profile

</button>

Hello, Jane Jacobs!

</form>

);

}

```

```css

label { display: block; margin-bottom: 20px; }

```

</Sandpack>

<Solution>

You will need two state variables to hold the input values: `firstName` and `lastName`. You're also going to need an `isEditing` state variable that holds whether to display the inputs or not. You should _not_ need a `fullName` variable because the full name can always be calculated from the `firstName` and the `lastName`.

Finally, you should use [conditional rendering](/learn/conditional-rendering) to show or hide the inputs depending on `isEditing`.

<Sandpack>

```js

import { useState } from 'react';

export default function EditProfile() {

const [isEditing, setIsEditing] = useState(false);

const [firstName, setFirstName] = useState('Jane');

```
const [lastName, setLastName] = useState('Jacobs');

return (
<form onSubmit={e => {
e.preventDefault();
setIsEditing(!isEditing);
}}>
<label>
First name:{' '}
{isEditing ? (
<input
value={firstName}
onChange={e => {
setFirstName(e.target.value)
}}
/>
) : (
{firstName}
)}
</label>
<label>
Last name:{' '}
{isEditing ? (
<input
value={lastName}
onChange={e => {
setLastName(e.target.value)
}}
/>
) : (
{lastName}
)}
</label>
<button type="submit">
{isEditing ? 'Save' : 'Edit'} Profile
</button>
```

Hello, {firstName} {lastName}!
</form>
);
}
```

```css
label { display: block; margin-bottom: 20px; }
```

</Sandpack>

Compare this solution to the original imperative code. How are they different?

</Solution>

Refactor the imperative solution without React {/*refactor-the-imperative-solution-without-react*/}

Here is the original sandbox from the previous challenge, written imperatively without React:

<Sandpack>

```js index.js active
function handleFormSubmit(e) {
e.preventDefault();
if (editButton.textContent === 'Edit Profile') {
editButton.textContent = 'Save Profile';
hide(firstNameText);
hide(lastNameText);
show(firstNameInput);
show(lastNameInput);
} else {
editButton.textContent = 'Edit Profile';
hide(firstNameInput);
hide(lastNameInput);
show(firstNameText);
show(lastNameText);
}
}

function handleFirstNameChange() {

```
firstNameText.textContent = firstNameInput.value;
helloText.textContent = (
'Hello ' +
firstNameInput.value + ' ' +
lastNameInput.value + '!'
);
}

function handleLastNameChange() {
lastNameText.textContent = lastNameInput.value;
helloText.textContent = (
'Hello ' +
firstNameInput.value + ' ' +
lastNameInput.value + '!'
);
}

function hide(el) {
el.style.display = 'none';
}

function show(el) {
el.style.display = '';
}

let form = document.getElementById('form');
let editButton = document.getElementById('editButton');
let firstNameInput = document.getElementById('firstNameInput');
let firstNameText = document.getElementById('firstNameText');
let lastNameInput = document.getElementById('lastNameInput');
let lastNameText = document.getElementById('lastNameText');
let helloText = document.getElementById('helloText');
form.onsubmit = handleFormSubmit;
firstNameInput.oninput = handleFirstNameChange;
lastNameInput.oninput = handleLastNameChange;
```

```js sandbox.config.json hidden
{
```

"hardReloadOnChange": true

}

```

```html public/index.html

<form id="form">

<label>

First name:

<b id="firstNameText">Jane</b>

<input

id="firstNameInput"

value="Jane"

style="display: none">

</label>

<label>

Last name:

<b id="lastNameText">Jacobs</b>

<input

id="lastNameInput"

value="Jacobs"

style="display: none">

</label>

<button type="submit" id="editButton">Edit Profile</button>

<p><i id="helloText">Hello, Jane Jacobs!</i></p>

</form>

<style>

* { box-sizing: border-box; }

body { font-family: sans-serif; margin: 20px; padding: 0; }

label { display: block; margin-bottom: 20px; }

</style>

```

</Sandpack>

Imagine React didn't exist. Can you refactor this code in a way that makes the logic less fragile and more similar to the React version? What would it look like if the state was explicit, like in React?
```

If you're struggling to think where to start, the stub below already has most of the structure in place. If you start here, fill in the missing logic in the `updateDOM` function. (Refer to the original code where needed.)

<Sandpack>

```js index.js active
let firstName = 'Jane';
let lastName = 'Jacobs';
let isEditing = false;

function handleFormSubmit(e) {
e.preventDefault();
setIsEditing(!isEditing);
}

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

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

function setFirstName(value) {
firstName = value;
updateDOM();
}

function setLastName(value) {
lastName = value;
updateDOM();
}

function setIsEditing(value) {
isEditing = value;
updateDOM();
}

function updateDOM() {
if (isEditing) {
editButton.textContent = 'Save Profile';
```

```
// TODO: show inputs, hide content
} else {
editButton.textContent = 'Edit Profile';
// TODO: hide inputs, show content
}
// TODO: update text labels
}

function hide(el) {
el.style.display = 'none';
}

function show(el) {
el.style.display = '';
}

let form = document.getElementById('form');
let editButton = document.getElementById('editButton');
let firstNameInput = document.getElementById('firstNameInput');
let firstNameText = document.getElementById('firstNameText');
let lastNameInput = document.getElementById('lastNameInput');
let lastNameText = document.getElementById('lastNameText');
let helloText = document.getElementById('helloText');
form.onsubmit = handleFormSubmit;
firstNameInput.oninput = handleFirstNameChange;
lastNameInput.oninput = handleLastNameChange;
```

```js sandbox.config.json hidden
{
"hardReloadOnChange": true
}
```

```html public/index.html
<form id="form">
<label>
First name:
<b id="firstNameText">Jane</b>
```

<input
id="firstNameInput"
value="Jane"
style="display: none">
</label>
<label>
Last name:
Jacobs
<input
id="lastNameInput"
value="Jacobs"
style="display: none">
</label>
<button type="submit" id="editButton">Edit Profile</button>
Hello, Jane Jacobs!
</form>

<style>
* { box-sizing: border-box; }
body { font-family: sans-serif; margin: 20px; padding: 0; }
label { display: block; margin-bottom: 20px; }
</style>
```

</Sandpack>

<Solution>

The missing logic included toggling the display of inputs and content, and updating the labels:

<Sandpack>

```js index.js active
let firstName = 'Jane';
let lastName = 'Jacobs';
let isEditing = false;

function handleFormSubmit(e) {
e.preventDefault();
setIsEditing(!isEditing);

```
}

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

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

function setFirstName(value) {
firstName = value;
updateDOM();
}

function setLastName(value) {
lastName = value;
updateDOM();
}

function setIsEditing(value) {
isEditing = value;
updateDOM();
}

function updateDOM() {
if (isEditing) {
editButton.textContent = 'Save Profile';
hide(firstNameText);
hide(lastNameText);
show(firstNameInput);
show(lastNameInput);
} else {
editButton.textContent = 'Edit Profile';
hide(firstNameInput);
hide(lastNameInput);
show(firstNameText);
show(lastNameText);
}
```

```
firstNameText.textContent = firstName;
lastNameText.textContent = lastName;
helloText.textContent = (
'Hello ' +
firstName + ' ' +
lastName + '!'
);
}

function hide(el) {
el.style.display = 'none';
}

function show(el) {
el.style.display = '';
}

let form = document.getElementById('form');
let editButton = document.getElementById('editButton');
let firstNameInput = document.getElementById('firstNameInput');
let firstNameText = document.getElementById('firstNameText');
let lastNameInput = document.getElementById('lastNameInput');
let lastNameText = document.getElementById('lastNameText');
let helloText = document.getElementById('helloText');
form.onsubmit = handleFormSubmit;
firstNameInput.oninput = handleFirstNameChange;
lastNameInput.oninput = handleLastNameChange;
```

```js sandbox.config.json hidden
{
"hardReloadOnChange": true
}
```

```html public/index.html
<form id="form">
<label>
First name:
```

Jane

<input

id="firstNameInput"

value="Jane"

style="display: none">

</label>

<label>

Last name:

Jacobs

<input

id="lastNameInput"

value="Jacobs"

style="display: none">

</label>

<button type="submit" id="editButton">Edit Profile</button>

Hello, Jane Jacobs!

</form>

<style>

* { box-sizing: border-box; }

body { font-family: sans-serif; margin: 20px; padding: 0; }

label { display: block; margin-bottom: 20px; }

</style>

```

</Sandpack>

The `updateDOM` function you wrote shows what React does under the hood when you set the state. (However, React also avoids touching the DOM for properties that have not changed since the last time they were set.)

</Solution>

</Challenges>

---

title: Adding Interactivity

---

<Intro>
```

Some things on the screen update in response to user input. For example, clicking an image gallery switches the active image. In React, data that changes over time is called *state.* You can add state to any component, and update it as needed. In this chapter, you'll learn how to write components that handle interactions, update their state, and display different output over time.

</Intro>

<YouWillLearn isChapter={true}>

* [How to handle user-initiated events](/learn/responding-to-events)
* [How to make components "remember" information with state](/learn/state-a-components-memory)
* [How React updates the UI in two phases](/learn/render-and-commit)
* [Why state doesn't update right after you change it](/learn/state-as-a-snapshot)
* [How to queue multiple state updates](/learn/queueing-a-series-of-state-updates)
* [How to update an object in state](/learn/updating-objects-in-state)
* [How to update an array in state](/learn/updating-arrays-in-state)

</YouWillLearn>

Responding to events {/*responding-to-events*/}

React lets you add *event handlers* to your JSX. Event handlers are your own functions that will be triggered in response to user interactions like clicking, hovering, focusing on form inputs, and so on.

Built-in components like `<button>` only support built-in browser events like `onClick`. However, you can also create your own components, and give their event handler props any application-specific names that you like.

<Sandpack>

```js
export default function App() {
return (
<Toolbar
onPlayMovie={() => alert('Playing!')}
onUploadImage={() => alert('Uploading!')}
/>
);
}

function Toolbar({ onPlayMovie, onUploadImage }) {
return (
<div>
<Button onClick={onPlayMovie}>
```

Play Movie

</Button>

<Button onClick={onUploadImage}>

Upload Image

</Button>

);

}

function Button({ onClick, children }) {

return (

<button onClick={onClick}>

{children}

</button>

);

}

```

```css

button { margin-right: 10px; }

```

</Sandpack>

<LearnMore path="/learn/responding-to-events">

Read **[Responding to Events](/learn/responding-to-events)** to learn how to add event handlers.

</LearnMore>

State: a component's memory {/*state-a-components-memory*/}

Components often need to change what's on the screen as a result of an interaction. Typing into the form should update the input field, clicking "next" on an image carousel should change which image is displayed, clicking "buy" puts a product in the shopping cart. Components need to "remember" things: the current input value, the current image, the shopping cart. In React, this kind of component-specific memory is called *state.*

You can add state to a component with a [`useState`](/reference/react/useState) Hook. *Hooks* are special functions that let your components use React features (state is one of those features). The `useState` Hook lets you declare a state variable. It takes the initial state and returns a pair of values: the current state, and a state setter function that lets you update it.

```js

```
const [index, setIndex] = useState(0);
const [showMore, setShowMore] = useState(false);
```

Here is how an image gallery uses and updates state on click:

<Sandpack>

```js
import { useState } from 'react';
import { sculptureList } from './data.js';

export default function Gallery() {
const [index, setIndex] = useState(0);
const [showMore, setShowMore] = useState(false);
const hasNext = index < sculptureList.length - 1;

function handleNextClick() {
if (hasNext) {
setIndex(index + 1);
} else {
setIndex(0);
}
}

function handleMoreClick() {
setShowMore(!showMore);
}

let sculpture = sculptureList[index];
return (
<>
<button onClick={handleNextClick}>
Next
</button>
<h2>
<i>{sculpture.name} </i>
by {sculpture.artist}
</h2>
<h3>
```

({index + 1} of {sculptureList.length})

</h3>

<button onClick={handleMoreClick}>

{showMore ? 'Hide' : 'Show'} details

</button>

{showMore && {sculpture.description} }

<img

src={sculpture.url}

alt={sculpture.alt}

/>

</>

);

}

```

```js data.js

export const sculptureList = [{

name: 'Homenaje a la Neurocirugía',

artist: 'Marta Colvin Andrade',

description: 'Although Colvin is predominantly known for abstract themes that allude to pre-Hispanic symbols, this gigantic sculpture, an homage to neurosurgery, is one of her most recognizable public art pieces.',

url: 'https://i.imgur.com/Mx7dA2Y.jpg',

alt: 'A bronze statue of two crossed hands delicately holding a human brain in their fingertips.'

}, {

name: 'Floralis Genérica',

artist: 'Eduardo Catalano',

description: 'This enormous (75 ft. or 23m) silver flower is located in Buenos Aires. It is designed to move, closing its petals in the evening or when strong winds blow and opening them in the morning.',

url: 'https://i.imgur.com/ZF6s192m.jpg',

alt: 'A gigantic metallic flower sculpture with reflective mirror-like petals and strong stamens.'

}, {

name: 'Eternal Presence',

artist: 'John Woodrow Wilson',

description: 'Wilson was known for his preoccupation with equality, social justice, as well as the essential and spiritual qualities of humankind. This massive (7ft. or 2,13m) bronze represents what he described as "a symbolic Black presence infused with a sense of universal humanity."',

url: 'https://i.imgur.com/aTtVpES.jpg',

alt: 'The sculpture depicting a human head seems ever-present and solemn. It radiates calm and serenity.'

}, {

name: 'Moai',

artist: 'Unknown Artist',

description: 'Located on the Easter Island, there are 1,000 moai, or extant monumental statues, created by the early Rapa Nui people, which some believe represented deified ancestors.',

url: 'https://i.imgur.com/RCwLEoQm.jpg',

alt: 'Three monumental stone busts with the heads that are disproportionately large with somber faces.'

}, {

name: 'Blue Nana',

artist: 'Niki de Saint Phalle',

description: 'The Nanas are triumphant creatures, symbols of femininity and maternity. Initially, Saint Phalle used fabric and found objects for the Nanas, and later on introduced polyester to achieve a more vibrant effect.',

url: 'https://i.imgur.com/Sd1AgUOm.jpg',

alt: 'A large mosaic sculpture of a whimsical dancing female figure in a colorful costume emanating joy.'

}, {

name: 'Ultimate Form',

artist: 'Barbara Hepworth',

description: 'This abstract bronze sculpture is a part of The Family of Man series located at Yorkshire Sculpture Park. Hepworth chose not to create literal representations of the world but developed abstract forms inspired by people and landscapes.',

url: 'https://i.imgur.com/2heNQDcm.jpg',

alt: 'A tall sculpture made of three elements stacked on each other reminding of a human figure.'

}, {

name: 'Cavaliere',

artist: 'Lamidi Olonade Fakeye',

description: "Descended from four generations of woodcarvers, Fakeye's work blended traditional and contemporary Yoruba themes.",

url: 'https://i.imgur.com/wIdGuZwm.png',

alt: 'An intricate wood sculpture of a warrior with a focused face on a horse adorned with patterns.'

}, {

name: 'Big Bellies',

artist: 'Alina Szapocznikow',

description: "Szapocznikow is known for her sculptures of the fragmented body as a metaphor for the fragility and impermanence of youth and beauty. This sculpture depicts two very realistic large bellies stacked on top of each other, each around five feet (1,5m) tall.",

url: 'https://i.imgur.com/AlHTAdDm.jpg',

alt: 'The sculpture reminds a cascade of folds, quite different from bellies in classical sculptures.'

}, {

name: 'Terracotta Army',

artist: 'Unknown Artist',

description: 'The Terracotta Army is a collection of terracotta sculptures depicting the armies of Qin Shi Huang, the first Emperor of China. The army consisted of more than 8,000 soldiers, 130 chariots with 520 horses, and 150 cavalry horses.',

url: 'https://i.imgur.com/HMFmH6m.jpg',

alt: '12 terracotta sculptures of solemn warriors, each with a unique facial expression and armor.'

}, {

name: 'Lunar Landscape',

artist: 'Louise Nevelson',

description: 'Nevelson was known for scavenging objects from New York City debris, which she would later assemble into monumental constructions. In this one, she used disparate parts like a bedpost, juggling pin, and seat fragment, nailing and gluing them into boxes that reflect the influence of Cubism’s geometric abstraction of space and form.',

url: 'https://i.imgur.com/rN7hY6om.jpg',

alt: 'A black matte sculpture where the individual elements are initially indistinguishable.'

}, {

name: 'Aureole',

artist: 'Ranjani Shettar',

description: 'Shettar merges the traditional and the modern, the natural and the industrial. Her art focuses on the relationship between man and nature. Her work was described as compelling both abstractly and figuratively, gravity defying, and a "fine synthesis of unlikely materials."',

url: 'https://i.imgur.com/okTpbHhm.jpg',

alt: 'A pale wire-like sculpture mounted on concrete wall and descending on the floor. It appears light.'

}, {

name: 'Hippos',

artist: 'Taipei Zoo',

description: 'The Taipei Zoo commissioned a Hippo Square featuring submerged hippos at play.',

url: 'https://i.imgur.com/6o5Vuyu.jpg',

alt: 'A group of bronze hippo sculptures emerging from the sett sidewalk as if they were swimming.'

}];

```

```css

h2 { margin-top: 10px; margin-bottom: 0; }

```
h3 {
margin-top: 5px;
font-weight: normal;
font-size: 100%;
}
img { width: 120px; height: 120px; }
button {
display: block;
margin-top: 10px;
margin-bottom: 10px;
}
```

</Sandpack>

<LearnMore path="/learn/state-a-components-memory">

Read **[State: A Component's Memory](/learn/state-a-components-memory)** to learn how to remember a value and update it on interaction.

</LearnMore>

Render and commit {/*render-and-commit*/}

Before your components are displayed on the screen, they must be rendered by React. Understanding the steps in this process will help you think about how your code executes and explain its behavior.

Imagine that your components are cooks in the kitchen, assembling tasty dishes from ingredients. In this scenario, React is the waiter who puts in requests from customers and brings them their orders. This process of requesting and serving UI has three steps:

1. **Triggering** a render (delivering the diner's order to the kitchen)
2. **Rendering** the component (preparing the order in the kitchen)
3. **Committing** to the DOM (placing the order on the table)

<IllustrationBlock sequential>

<Illustration caption="Trigger" alt="React as a server in a restaurant, fetching orders from the users and delivering them to the Component Kitchen." src="/images/docs/illustrations/i_render-and-commit1.png" />

<Illustration caption="Render" alt="The Card Chef gives React a fresh Card component." src="/images/docs/illustrations/i_render-and-commit2.png" />

<Illustration caption="Commit" alt="React delivers the Card to the user at their table." src="/images/docs/illustrations/i_render-and-commit3.png" />

</IllustrationBlock>

<LearnMore path="/learn/render-and-commit">

Read **[Render and Commit](/learn/render-and-commit)** to learn the lifecycle of a UI update.

</LearnMore>

State as a snapshot {/*state-as-a-snapshot*/}

Unlike regular JavaScript variables, React state behaves more like a snapshot. Setting it does not change the state variable you already have, but instead triggers a re-render. This can be surprising at first!

```js
console.log(count); // 0
setCount(count + 1); // Request a re-render with 1
console.log(count); // Still 0!
```

This behavior help you avoid subtle bugs. Here is a little chat app. Try to guess what happens if you press "Send" first and *then* change the recipient to Bob. Whose name will appear in the `alert` five seconds later?

<Sandpack>

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

export default function Form() {
const [to, setTo] = useState('Alice');
const [message, setMessage] = useState('Hello');

function handleSubmit(e) {
e.preventDefault();
setTimeout(() => {
alert(`You said ${message} to ${to}`);
}, 5000);
}

return (
<form onSubmit={handleSubmit}>
<label>
To:{' '}
<select
value={to}
```

onChange={e => setTo(e.target.value)}>
<option value="Alice">Alice</option>
<option value="Bob">Bob</option>
</select>
</label>
<textarea
placeholder="Message"
value={message}
onChange={e => setMessage(e.target.value)}
/>
<button type="submit">Send</button>
</form>
);
}
```

```css
label, textarea { margin-bottom: 10px; display: block; }
```

</Sandpack>

<LearnMore path="/learn/state-as-a-snapshot">

Read **[State as a Snapshot](/learn/state-as-a-snapshot)** to learn why state appears "fixed" and unchanging inside the event handlers.

</LearnMore>

Queueing a series of state updates {/*queueing-a-series-of-state-updates*/}

This component is buggy: clicking "+3" increments the score only once.

<Sandpack>

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

export default function Counter() {
const [score, setScore] = useState(0);

function increment() {
setScore(score + 1);

```
}

return (
<>
<button onClick={() => increment()}>+1</button>
<button onClick={() => {
increment();
increment();
increment();
}}>+3</button>
<h1>Score: {score}</h1>
</>
)
}
```

```css
button { display: inline-block; margin: 10px; font-size: 20px; }
```

</Sandpack>

[State as a Snapshot](/learn/state-as-a-snapshot) explains why this is happening. Setting state requests a new re-render, but does not change it in the already running code. So `score` continues to be `0` right after you call `setScore(score + 1)`.

```js
console.log(score); // 0
setScore(score + 1); // setScore(0 + 1);
console.log(score); // 0
setScore(score + 1); // setScore(0 + 1);
console.log(score); // 0
setScore(score + 1); // setScore(0 + 1);
console.log(score); // 0
```

You can fix this by passing an *updater function* when setting state. Notice how replacing `setScore(score + 1)` with `setScore(s => s + 1)` fixes the "+3" button. This lets you queue multiple state updates.

<Sandpack>

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

export default function Counter() {
const [score, setScore] = useState(0);

function increment() {
setScore(s => s + 1);
}

return (
<>
<button onClick={() => increment()}>+1</button>
<button onClick={() => {
increment();
increment();
increment();
}}>+3</button>
<h1>Score: {score}</h1>
</>
)
}
```

```css
button { display: inline-block; margin: 10px; font-size: 20px; }
```

</Sandpack>

<LearnMore path="/learn/queueing-a-series-of-state-updates">

Read **[Queueing a Series of State Updates](/learn/queueing-a-series-of-state-updates)** to learn how to queue a sequence of state updates.

</LearnMore>

Updating objects in state {/*updating-objects-in-state*/}

State can hold any kind of JavaScript value, including objects. But you shouldn't change objects and arrays that you hold in the React state directly. Instead, when you want to update an object and array, you need to create a new one (or make a copy of an existing one), and then update the state to use that copy.

Usually, you will use the `...` spread syntax to copy objects and arrays that you want to change. For example, updating a nested object could look like this:

<Sandpack>

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

export default function Form() {
const [person, setPerson] = useState({
name: 'Niki de Saint Phalle',
artwork: {
title: 'Blue Nana',
city: 'Hamburg',
image: 'https://i.imgur.com/Sd1AgUOm.jpg',
}
});

function handleNameChange(e) {
setPerson({
...person,
name: e.target.value
});
}

function handleTitleChange(e) {
setPerson({
...person,
artwork: {
...person.artwork,
title: e.target.value
}
});
}

function handleCityChange(e) {
setPerson({
...person,
artwork: {
...person.artwork,
```

```
city: e.target.value
}
});
}
function handleImageChange(e) {
setPerson({
...person,
artwork: {
...person.artwork,
image: e.target.value
}
});
}
return (
<>
<label>
Name:
<input
value={person.name}
onChange={handleNameChange}
/>
</label>
<label>
Title:
<input
value={person.artwork.title}
onChange={handleTitleChange}
/>
</label>
<label>
City:
<input
value={person.artwork.city}
onChange={handleCityChange}
/>
```

</label>
<label>
Image:
<input
value={person.artwork.image}
onChange={handleImageChange}
/>
</label>

{person.artwork.title}
{' by '}
{person.name}

(located in {person.artwork.city})

<img
src={person.artwork.image}
alt={person.artwork.title}
/>
</>
);
}
```

```css
label { display: block; }
input { margin-left: 5px; margin-bottom: 5px; }
img { width: 200px; height: 200px; }
```

</Sandpack>

If copying objects in code gets tedious, you can use a library like [Immer](https://github.com/immerjs/use-immer) to reduce repetitive code:

<Sandpack>

```js
import { useImmer } from 'use-immer';

```
export default function Form() {
const [person, updatePerson] = useImmer({
name: 'Niki de Saint Phalle',
artwork: {
title: 'Blue Nana',
city: 'Hamburg',
image: 'https://i.imgur.com/Sd1AgUOm.jpg',
}
});

function handleNameChange(e) {
updatePerson(draft => {
draft.name = e.target.value;
});
}

function handleTitleChange(e) {
updatePerson(draft => {
draft.artwork.title = e.target.value;
});
}

function handleCityChange(e) {
updatePerson(draft => {
draft.artwork.city = e.target.value;
});
}

function handleImageChange(e) {
updatePerson(draft => {
draft.artwork.image = e.target.value;
});
}

return (
<>
<label>
Name:
<input
```

```
value={person.name}
onChange={handleNameChange}
/>
</label>
<label>
Title:
<input
value={person.artwork.title}
onChange={handleTitleChange}
/>
</label>
<label>
City:
<input
value={person.artwork.city}
onChange={handleCityChange}
/>
</label>
<label>
Image:
<input
value={person.artwork.image}
onChange={handleImageChange}
/>
</label>
<p>
<i>{person.artwork.title}</i>
{' by '}
{person.name}
<br />
(located in {person.artwork.city})
</p>
<img
src={person.artwork.image}
alt={person.artwork.title}
```

/>

</>

);

}

```

```json package.json

{

"dependencies": {

"immer": "1.7.3",

"react": "latest",

"react-dom": "latest",

"react-scripts": "latest",

"use-immer": "0.5.1"

},

"scripts": {

"start": "react-scripts start",

"build": "react-scripts build",

"test": "react-scripts test --env=jsdom",

"eject": "react-scripts eject"

}

}

```

```css

label { display: block; }

input { margin-left: 5px; margin-bottom: 5px; }

img { width: 200px; height: 200px; }

```

</Sandpack>

<LearnMore path="/learn/updating-objects-in-state">

Read **[Updating Objects in State](/learn/updating-objects-in-state)** to learn how to update objects correctly.

</LearnMore>

Updating arrays in state {/*updating-arrays-in-state*/}
```

Arrays are another type of mutable JavaScript objects you can store in state and should treat as read-only. Just like with objects, when you want to update an array stored in state, you need to create a new one (or make a copy of an existing one), and then set state to use the new array:

<Sandpack>

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

let nextId = 3;
const initialList = [
{ id: 0, title: 'Big Bellies', seen: false },
{ id: 1, title: 'Lunar Landscape', seen: false },
{ id: 2, title: 'Terracotta Army', seen: true },
];

export default function BucketList() {
const [list, setList] = useState(
initialList
);

function handleToggle(artworkId, nextSeen) {
setList(list.map(artwork => {
if (artwork.id === artworkId) {
return { ...artwork, seen: nextSeen };
} else {
return artwork;
}
}));
}

return (
<>
<h1>Art Bucket List</h1>
<h2>My list of art to see:</h2>
<ItemList
artworks={list}
onToggle={handleToggle} />
</>
);
```

```
}

function ItemList({ artworks, onToggle }) {
return (
<ul>
{artworks.map(artwork => (
<li key={artwork.id}>
<label>
<input
type="checkbox"
checked={artwork.seen}
onChange={e => {
onToggle(
artwork.id,
e.target.checked
);
}}
/>
{artwork.title}
</label>
</li>
))}
</ul>
);
}
```

</Sandpack>

If copying arrays in code gets tedious, you can use a library like [Immer](https://github.com/immerjs/use-immer) to reduce repetitive code:

<Sandpack>

```js
import { useState } from 'react';
import { useImmer } from 'use-immer';

let nextId = 3;
const initialList = [
```

```
{ id: 0, title: 'Big Bellies', seen: false },
{ id: 1, title: 'Lunar Landscape', seen: false },
{ id: 2, title: 'Terracotta Army', seen: true },
];

export default function BucketList() {
const [list, updateList] = useImmer(initialList);

function handleToggle(artworkId, nextSeen) {
updateList(draft => {
const artwork = draft.find(a =>
a.id === artworkId
);
artwork.seen = nextSeen;
});
}

return (
<>
<h1>Art Bucket List</h1>
<h2>My list of art to see:</h2>
<ItemList
artworks={list}
onToggle={handleToggle} />
</>
);
}

function ItemList({ artworks, onToggle }) {
return (
<ul>
{artworks.map(artwork => (
<li key={artwork.id}>
<label>
<input
type="checkbox"
checked={artwork.seen}
onChange={e => {
```

onToggle(
artwork.id,
e.target.checked
);
}}
/>
{artwork.title}
</label>

))}

);
}
```

```json package.json
{
"dependencies": {
"immer": "1.7.3",
"react": "latest",
"react-dom": "latest",
"react-scripts": "latest",
"use-immer": "0.5.1"
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
}
}
```

</Sandpack>

<LearnMore path="/learn/updating-arrays-in-state">
```

Read **[Updating Arrays in State](/learn/updating-arrays-in-state)** to learn how to update arrays correctly.

</LearnMore>

What's next? {/*whats-next*/}

Head over to [Responding to Events](/learn/responding-to-events) to start reading this chapter page by page!

Or, if you're already familiar with these topics, why not read about [Managing State](/learn/managing-state)?

---

title: Extracting State Logic into a Reducer

<Intro>

Components with many state updates spread across many event handlers can get overwhelming. For these cases, you can consolidate all the state update logic outside your component in a single function, called a _reducer._

</Intro>

<YouWillLearn>

- What a reducer function is
- How to refactor `useState` to `useReducer`
- When to use a reducer
- How to write one well

</YouWillLearn>

Consolidate state logic with a reducer {/*consolidate-state-logic-with-a-reducer*/}

As your components grow in complexity, it can get harder to see at a glance all the different ways in which a component's state gets updated. For example, the `TaskApp` component below holds an array of `tasks` in state and uses three different event handlers to add, remove, and edit tasks:

<Sandpack>

```js App.js
import { useState } from 'react';
import AddTask from './AddTask.js';
import TaskList from './TaskList.js';

export default function TaskApp() {
const [tasks, setTasks] = useState(initialTasks);
```

```
function handleAddTask(text) {
setTasks([
...tasks,
{
id: nextId++,
text: text,
done: false,
},
]);
}

function handleChangeTask(task) {
setTasks(
tasks.map((t) => {
if (t.id === task.id) {
return task;
} else {
return t;
}
})
);
}

function handleDeleteTask(taskId) {
setTasks(tasks.filter((t) => t.id !== taskId));
}

return (
<>
<h1>Prague itinerary</h1>
<AddTask onAddTask={handleAddTask} />
<TaskList
tasks={tasks}
onChangeTask={handleChangeTask}
onDeleteTask={handleDeleteTask}
/>
</>
```

```
);
}

let nextId = 3;
const initialTasks = [
{id: 0, text: 'Visit Kafka Museum', done: true},
{id: 1, text: 'Watch a puppet show', done: false},
{id: 2, text: 'Lennon Wall pic', done: false},
];
```

```js AddTask.js hidden
import { useState } from 'react';

export default function AddTask({onAddTask}) {
const [text, setText] = useState('');
return (
<>
<input
placeholder="Add task"
value={text}
onChange={(e) => setText(e.target.value)}
/>
<button
onClick={() => {
setText('');
onAddTask(text);
}}>
Add
</button>
</>
);
}
```

```js TaskList.js hidden
import { useState } from 'react';

export default function TaskList({tasks, onChangeTask, onDeleteTask}) {
```

```
return (
<ul>
{tasks.map((task) => (
<li key={task.id}>
<Task task={task} onChange={onChangeTask} onDelete={onDeleteTask} />
</li>
))}
</ul>
);
}

function Task({task, onChange, onDelete}) {
const [isEditing, setIsEditing] = useState(false);
let taskContent;
if (isEditing) {
taskContent = (
<>
<input
value={task.text}
onChange={(e) => {
onChange({
...task,
text: e.target.value,
});
}}
/>
<button onClick={() => setIsEditing(false)}>Save</button>
</>
);
} else {
taskContent = (
<>
{task.text}
<button onClick={() => setIsEditing(true)}>Edit</button>
</>
);
```

```
}
return (
<label>
<input
type="checkbox"
checked={task.done}
onChange={(e) => {
onChange({
...task,
done: e.target.checked,
});
}}
/>
{taskContent}
<button onClick={() => onDelete(task.id)}>Delete</button>
</label>
);
}
```

```css
button {
margin: 5px;
}
li {
list-style-type: none;
}
ul,
li {
margin: 0;
padding: 0;
}
```

</Sandpack>

Each of its event handlers calls `setTasks` in order to update the state. As this component grows, so does the amount of state logic sprinkled throughout it. To reduce this complexity and keep all your logic

in one easy-to-access place, you can move that state logic into a single function outside your component, **called a "reducer".**

Reducers are a different way to handle state. You can migrate from `useState` to `useReducer` in three steps:

1. **Move** from setting state to dispatching actions.
2. **Write** a reducer function.
3. **Use** the reducer from your component.

Step 1: Move from setting state to dispatching actions
{/*step-1-move-from-setting-state-to-dispatching-actions*/}

Your event handlers currently specify _what to do_ by setting state:

```js
function handleAddTask(text) {
setTasks([
...tasks,
{
id: nextId++,
text: text,
done: false,
},
]);
}

function handleChangeTask(task) {
setTasks(
tasks.map((t) => {
if (t.id === task.id) {
return task;
} else {
return t;
}
})
);
}

function handleDeleteTask(taskId) {
setTasks(tasks.filter((t) => t.id !== taskId));
```

}
```

Remove all the state setting logic. What you are left with are three event handlers:

- `handleAddTask(text)` is called when the user presses "Add".
- `handleChangeTask(task)` is called when the user toggles a task or presses "Save".
- `handleDeleteTask(taskId)` is called when the user presses "Delete".

Managing state with reducers is slightly different from directly setting state. Instead of telling React "what to do" by setting state, you specify "what the user just did" by dispatching "actions" from your event handlers. (The state update logic will live elsewhere!) So instead of "setting `tasks`" via an event handler, you're dispatching an "added/changed/deleted a task" action. This is more descriptive of the user's intent.

```js
function handleAddTask(text) {
dispatch({
type: 'added',
id: nextId++,
text: text,
});
}

function handleChangeTask(task) {
dispatch({
type: 'changed',
task: task,
});
}

function handleDeleteTask(taskId) {
dispatch({
type: 'deleted',
id: taskId,
});
}
```

The object you pass to `dispatch` is called an "action":

```js {3-7}

```
function handleDeleteTask(taskId) {
dispatch(
// "action" object:
{
type: 'deleted',
id: taskId,
}
);
}
```

It is a regular JavaScript object. You decide what to put in it, but generally it should contain the minimal information about _what happened_. (You will add the `dispatch` function itself in a later step.)

<Note>

An action object can have any shape.

By convention, it is common to give it a string `type` that describes what happened, and pass any additional information in other fields. The `type` is specific to a component, so in this example either `'added'` or `'added_task'` would be fine. Choose a name that says what happened!

```js
dispatch({
// specific to component
type: 'what_happened',
// other fields go here
});
```

</Note>

Step 2: Write a reducer function {/*step-2-write-a-reducer-function*/}

A reducer function is where you will put your state logic. It takes two arguments, the current state and the action object, and it returns the next state:

```js
function yourReducer(state, action) {
// return next state for React to set
}
```

React will set the state to what you return from the reducer.

To move your state setting logic from your event handlers to a reducer function in this example, you will:

1. Declare the current state (`tasks`) as the first argument.
2. Declare the `action` object as the second argument.
3. Return the _next_ state from the reducer (which React will set the state to).

Here is all the state setting logic migrated to a reducer function:

```js
function tasksReducer(tasks, action) {
if (action.type === 'added') {
return [
...tasks,
{
id: action.id,
text: action.text,
done: false,
},
];
} else if (action.type === 'changed') {
return tasks.map((t) => {
if (t.id === action.task.id) {
return action.task;
} else {
return t;
}
});
} else if (action.type === 'deleted') {
return tasks.filter((t) => t.id !== action.id);
} else {
throw Error('Unknown action: ' + action.type);
}
}
```

Because the reducer function takes state (`tasks`) as an argument, you can **declare it outside of your component.** This decreases the indentation level and can make your code easier to read.

<Note>

The code above uses if/else statements, but it's a convention to use [switch statements](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Statements/switch) inside reducers. The result is the same, but it can be easier to read switch statements at a glance.

We'll be using them throughout the rest of this documentation like so:

```js
function tasksReducer(tasks, action) {
switch (action.type) {
case 'added': {
return [
...tasks,
{
id: action.id,
text: action.text,
done: false,
},
];
}
case 'changed': {
return tasks.map((t) => {
if (t.id === action.task.id) {
return action.task;
} else {
return t;
}
});
}
case 'deleted': {
return tasks.filter((t) => t.id !== action.id);
}
default: {
throw Error('Unknown action: ' + action.type);
}
```

}

}

```

We recommend wrapping each `case` block into the `{` and `}` curly braces so that variables declared inside of different `case`s don't clash with each other. Also, a `case` should usually end with a `return`. If you forget to `return`, the code will "fall through" to the next `case`, which can lead to mistakes!

If you're not yet comfortable with switch statements, using if/else is completely fine.

</Note>

<DeepDive>

#### Why are reducers called this way? {/*why-are-reducers-called-this-way*/}

Although reducers can "reduce" the amount of code inside your component, they are actually named after the [`reduce()`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce) operation that you can perform on arrays.

The `reduce()` operation lets you take an array and "accumulate" a single value out of many:

```

const arr = [1, 2, 3, 4, 5];

const sum = arr.reduce(

(result, number) => result + number

); // 1 + 2 + 3 + 4 + 5

```

The function you pass to `reduce` is known as a "reducer". It takes the _result so far_ and the _current item,_ then it returns the _next result._ React reducers are an example of the same idea: they take the _state so far_ and the _action_, and return the _next state._ In this way, they accumulate actions over time into state.

You could even use the `reduce()` method with an `initialState` and an array of `actions` to calculate the final state by passing your reducer function to it:

<Sandpack>

```js index.js active

import tasksReducer from './tasksReducer.js';

let initialState = [];

let actions = [

{type: 'added', id: 1, text: 'Visit Kafka Museum'},

{type: 'added', id: 2, text: 'Watch a puppet show'},

{type: 'deleted', id: 1},

{type: 'added', id: 3, text: 'Lennon Wall pic'},
];

let finalState = actions.reduce(tasksReducer, initialState);

const output = document.getElementById('output');
output.textContent = JSON.stringify(finalState, null, 2);
```

```js tasksReducer.js
export default function tasksReducer(tasks, action) {
switch (action.type) {
case 'added': {
return [
...tasks,
{
id: action.id,
text: action.text,
done: false,
},
];
}
case 'changed': {
return tasks.map((t) => {
if (t.id === action.task.id) {
return action.task;
} else {
return t;
}
});
}
case 'deleted': {
return tasks.filter((t) => t.id !== action.id);
}
default: {
throw Error('Unknown action: ' + action.type);
}

}

}
```

```html public/index.html
<pre id="output"></pre>
```

</Sandpack>

You probably won't need to do this yourself, but this is similar to what React does!

</DeepDive>

Step 3: Use the reducer from your component {/*step-3-use-the-reducer-from-your-component*/}

Finally, you need to hook up the `tasksReducer` to your component. Import the `useReducer` Hook from React:

```js
import { useReducer } from 'react';
```

Then you can replace `useState`:

```js
const [tasks, setTasks] = useState(initialTasks);
```

with `useReducer` like so:

```js
const [tasks, dispatch] = useReducer(tasksReducer, initialTasks);
```

The `useReducer` Hook is similar to `useState`—you must pass it an initial state and it returns a stateful value and a way to set state (in this case, the dispatch function). But it's a little different.

The `useReducer` Hook takes two arguments:

1. A reducer function
2. An initial state

And it returns:

1. A stateful value
2. A dispatch function (to "dispatch" user actions to the reducer)
```

Now it's fully wired up! Here, the reducer is declared at the bottom of the component file:

<Sandpack>

```js App.js
import { useReducer } from 'react';
import AddTask from './AddTask.js';
import TaskList from './TaskList.js';

export default function TaskApp() {
const [tasks, dispatch] = useReducer(tasksReducer, initialTasks);

function handleAddTask(text) {
dispatch({
type: 'added',
id: nextId++,
text: text,
});
}

function handleChangeTask(task) {
dispatch({
type: 'changed',
task: task,
});
}

function handleDeleteTask(taskId) {
dispatch({
type: 'deleted',
id: taskId,
});
}

return (
<>
<h1>Prague itinerary</h1>
<AddTask onAddTask={handleAddTask} />
<TaskList
tasks={tasks}
```

```
onChangeTask={handleChangeTask}
onDeleteTask={handleDeleteTask}
/>
</>
);
}

function tasksReducer(tasks, action) {
switch (action.type) {
case 'added': {
return [
...tasks,
{
id: action.id,
text: action.text,
done: false,
},
];
}
case 'changed': {
return tasks.map((t) => {
if (t.id === action.task.id) {
return action.task;
} else {
return t;
}
});
}
case 'deleted': {
return tasks.filter((t) => t.id !== action.id);
}
default: {
throw Error('Unknown action: ' + action.type);
}
}
}
```

```
let nextId = 3;
const initialTasks = [
{id: 0, text: 'Visit Kafka Museum', done: true},
{id: 1, text: 'Watch a puppet show', done: false},
{id: 2, text: 'Lennon Wall pic', done: false},
];
```

```js AddTask.js hidden
import { useState } from 'react';

export default function AddTask({onAddTask}) {
const [text, setText] = useState('');
return (
<>
<input
placeholder="Add task"
value={text}
onChange={(e) => setText(e.target.value)}
/>
<button
onClick={() => {
setText('');
onAddTask(text);
}}>
Add
</button>
</>
);
}
```

```js TaskList.js hidden
import { useState } from 'react';

export default function TaskList({tasks, onChangeTask, onDeleteTask}) {
return (
<ul>
```

```
{tasks.map((task) => (
<li key={task.id}>
<Task task={task} onChange={onChangeTask} onDelete={onDeleteTask} />
</li>
))}
</ul>
);
}

function Task({task, onChange, onDelete}) {
const [isEditing, setIsEditing] = useState(false);
let taskContent;
if (isEditing) {
taskContent = (
<>
<input
value={task.text}
onChange={(e) => {
onChange({
...task,
text: e.target.value,
});
}}
/>
<button onClick={() => setIsEditing(false)}>Save</button>
</>
);
} else {
taskContent = (
<>
{task.text}
<button onClick={() => setIsEditing(true)}>Edit</button>
</>
);
}
return (
```

<label>
<input
type="checkbox"
checked={task.done}
onChange={(e) => {
onChange({
...task,
done: e.target.checked,
});
}}
/>
{taskContent}
<button onClick={() => onDelete(task.id)}>Delete</button>
</label>
);
}
```

```css
button {
margin: 5px;
}
li {
list-style-type: none;
}
ul,
li {
margin: 0;
padding: 0;
}
```

</Sandpack>

If you want, you can even move the reducer to a different file:

<Sandpack>

```js App.js

```
import { useReducer } from 'react';
import AddTask from './AddTask.js';
import TaskList from './TaskList.js';
import tasksReducer from './tasksReducer.js';

export default function TaskApp() {
const [tasks, dispatch] = useReducer(tasksReducer, initialTasks);

function handleAddTask(text) {
dispatch({
type: 'added',
id: nextId++,
text: text,
});
}

function handleChangeTask(task) {
dispatch({
type: 'changed',
task: task,
});
}

function handleDeleteTask(taskId) {
dispatch({
type: 'deleted',
id: taskId,
});
}

return (
<>
<h1>Prague itinerary</h1>
<AddTask onAddTask={handleAddTask} />
<TaskList
tasks={tasks}
onChangeTask={handleChangeTask}
onDeleteTask={handleDeleteTask}
/>
```

```
</>
);
}

let nextId = 3;
const initialTasks = [
{id: 0, text: 'Visit Kafka Museum', done: true},
{id: 1, text: 'Watch a puppet show', done: false},
{id: 2, text: 'Lennon Wall pic', done: false},
];
```

```js tasksReducer.js
export default function tasksReducer(tasks, action) {
switch (action.type) {
case 'added': {
return [
...tasks,
{
id: action.id,
text: action.text,
done: false,
},
];
}
case 'changed': {
return tasks.map((t) => {
if (t.id === action.task.id) {
return action.task;
} else {
return t;
}
});
}
case 'deleted': {
return tasks.filter((t) => t.id !== action.id);
}
```

```
default: {
throw Error('Unknown action: ' + action.type);
}
}
}
```

```js AddTask.js hidden
import { useState } from 'react';

export default function AddTask({onAddTask}) {
const [text, setText] = useState('');
return (
<>
<input
placeholder="Add task"
value={text}
onChange={(e) => setText(e.target.value)}
/>
<button
onClick={() => {
setText('');
onAddTask(text);
}}>
Add
</button>
</>
);
}
```

```js TaskList.js hidden
import { useState } from 'react';

export default function TaskList({tasks, onChangeTask, onDeleteTask}) {
return (
<ul>
{tasks.map((task) => (
```

```
<li key={task.id}>
<Task task={task} onChange={onChangeTask} onDelete={onDeleteTask} />
</li>
))}
</ul>
);
}

function Task({task, onChange, onDelete}) {
const [isEditing, setIsEditing] = useState(false);
let taskContent;
if (isEditing) {
taskContent = (
<>
<input
value={task.text}
onChange={(e) => {
onChange({
...task,
text: e.target.value,
});
}}
/>
<button onClick={() => setIsEditing(false)}>Save</button>
</>
);
} else {
taskContent = (
<>
{task.text}
<button onClick={() => setIsEditing(true)}>Edit</button>
</>
);
}
return (
<label>
```

<input
type="checkbox"
checked={task.done}
onChange={(e) => {
onChange({
...task,
done: e.target.checked,
});
}}
/>
{taskContent}
<button onClick={() => onDelete(task.id)}>Delete</button>
</label>
);
}
```

```css
button {
margin: 5px;
}
li {
list-style-type: none;
}
ul,
li {
margin: 0;
padding: 0;
}
```

</Sandpack>

Component logic can be easier to read when you separate concerns like this. Now the event handlers only specify _what happened_ by dispatching actions, and the reducer function determines _how the state updates_ in response to them.

Comparing `useState` and `useReducer` {/*comparing-usestate-and-usereducer*/}
```

Reducers are not without downsides! Here's a few ways you can compare them:

- **Code size:** Generally, with `useState` you have to write less code upfront. With `useReducer`, you have to write both a reducer function _and_ dispatch actions. However, `useReducer` can help cut down on the code if many event handlers modify state in a similar way.

- **Readability:** `useState` is very easy to read when the state updates are simple. When they get more complex, they can bloat your component's code and make it difficult to scan. In this case, `useReducer` lets you cleanly separate the _how_ of update logic from the _what happened_ of event handlers.

- **Debugging:** When you have a bug with `useState`, it can be difficult to tell _where_ the state was set incorrectly, and _why_. With `useReducer`, you can add a console log into your reducer to see every state update, and _why_ it happened (due to which `action`). If each `action` is correct, you'll know that the mistake is in the reducer logic itself. However, you have to step through more code than with `useState`.

- **Testing:** A reducer is a pure function that doesn't depend on your component. This means that you can export and test it separately in isolation. While generally it's best to test components in a more realistic environment, for complex state update logic it can be useful to assert that your reducer returns a particular state for a particular initial state and action.

- **Personal preference:** Some people like reducers, others don't. That's okay. It's a matter of preference. You can always convert between `useState` and `useReducer` back and forth: they are equivalent!

We recommend using a reducer if you often encounter bugs due to incorrect state updates in some component, and want to introduce more structure to its code. You don't have to use reducers for everything: feel free to mix and match! You can even `useState` and `useReducer` in the same component.

Writing reducers well {/*writing-reducers-well*/}

Keep these two tips in mind when writing reducers:

- **Reducers must be pure.** Similar to [state updater functions](/learn/queueing-a-series-of-state-updates), reducers run during rendering! (Actions are queued until the next render.) This means that reducers [must be pure](/learn/keeping-components-pure)—same inputs always result in the same output. They should not send requests, schedule timeouts, or perform any side effects (operations that impact things outside the component). They should update [objects](/learn/updating-objects-in-state) and [arrays](/learn/updating-arrays-in-state) without mutations.

- **Each action describes a single user interaction, even if that leads to multiple changes in the data.** For example, if a user presses "Reset" on a form with five fields managed by a reducer, it makes more sense to dispatch one `reset_form` action rather than five separate `set_field` actions. If you log every action in a reducer, that log should be clear enough for you to reconstruct what interactions or responses happened in what order. This helps with debugging!

Writing concise reducers with Immer {/*writing-concise-reducers-with-immer*/}

Just like with [updating objects](/learn/updating-objects-in-state#write-concise-update-logic-with-immer) and [arrays](/learn/updating-arrays-in-state#write-concise-update-logic-with-immer) in regular state, you can use the Immer library to make reducers more concise. Here, [`useImmerReducer`](https://github.com/immerjs/use-immer#useimmerreducer) lets you mutate the

state with `push` or `arr[i] =` assignment:

<Sandpack>

```js App.js
import { useImmerReducer } from 'use-immer';
import AddTask from './AddTask.js';
import TaskList from './TaskList.js';

function tasksReducer(draft, action) {
switch (action.type) {
case 'added': {
draft.push({
id: action.id,
text: action.text,
done: false,
});
break;
}
case 'changed': {
const index = draft.findIndex((t) => t.id === action.task.id);
draft[index] = action.task;
break;
}
case 'deleted': {
return draft.filter((t) => t.id !== action.id);
}
default: {
throw Error('Unknown action: ' + action.type);
}
}
}

export default function TaskApp() {
const [tasks, dispatch] = useImmerReducer(tasksReducer, initialTasks);

function handleAddTask(text) {
dispatch({
type: 'added',
```

```
id: nextId++,
text: text,
});
}

function handleChangeTask(task) {
dispatch({
type: 'changed',
task: task,
});
}

function handleDeleteTask(taskId) {
dispatch({
type: 'deleted',
id: taskId,
});
}

return (
<>
<h1>Prague itinerary</h1>
<AddTask onAddTask={handleAddTask} />
<TaskList
tasks={tasks}
onChangeTask={handleChangeTask}
onDeleteTask={handleDeleteTask}
/>
</>
);
}

let nextId = 3;
const initialTasks = [
{id: 0, text: 'Visit Kafka Museum', done: true},
{id: 1, text: 'Watch a puppet show', done: false},
{id: 2, text: 'Lennon Wall pic', done: false},
];
```

```

```js AddTask.js hidden
import { useState } from 'react';

export default function AddTask({onAddTask}) {
const [text, setText] = useState('');
return (
<>
<input
placeholder="Add task"
value={text}
onChange={(e) => setText(e.target.value)}
/>
<button
onClick={() => {
setText('');
onAddTask(text);
}}>
Add
</button>
</>
);
}
```

```js TaskList.js hidden
import { useState } from 'react';

export default function TaskList({tasks, onChangeTask, onDeleteTask}) {
return (
<ul>
{tasks.map((task) => (
<li key={task.id}>
<Task task={task} onChange={onChangeTask} onDelete={onDeleteTask} />
</li>
))}
</ul>

```
);
}

function Task({task, onChange, onDelete}) {
const [isEditing, setIsEditing] = useState(false);
let taskContent;
if (isEditing) {
taskContent = (
<>
<input
value={task.text}
onChange={(e) => {
onChange({
...task,
text: e.target.value,
});
}}
/>
<button onClick={() => setIsEditing(false)}>Save</button>
</>
);
} else {
taskContent = (
<>
{task.text}
<button onClick={() => setIsEditing(true)}>Edit</button>
</>
);
}
return (
<label>
<input
type="checkbox"
checked={task.done}
onChange={(e) => {
onChange({
```

```
...task,
done: e.target.checked,
});
}}
/>
{taskContent}
<button onClick={() => onDelete(task.id)}>Delete</button>
</label>
);
}
```

```css
button {
margin: 5px;
}
li {
list-style-type: none;
}
ul,
li {
margin: 0;
padding: 0;
}
```

```json package.json
{
"dependencies": {
"immer": "1.7.3",
"react": "latest",
"react-dom": "latest",
"react-scripts": "latest",
"use-immer": "0.5.1"
},
"scripts": {
"start": "react-scripts start",
```

"build": "react-scripts build",

"test": "react-scripts test --env=jsdom",

"eject": "react-scripts eject"

}

}

```

</Sandpack>

Reducers must be pure, so they shouldn't mutate state. But Immer provides you with a special `draft` object which is safe to mutate. Under the hood, Immer will create a copy of your state with the changes you made to the `draft`. This is why reducers managed by `useImmerReducer` can mutate their first argument and don't need to return state.

<Recap>

- To convert from `useState` to `useReducer`:

1. Dispatch actions from event handlers.

2. Write a reducer function that returns the next state for a given state and action.

3. Replace `useState` with `useReducer`.

- Reducers require you to write a bit more code, but they help with debugging and testing.

- Reducers must be pure.

- Each action describes a single user interaction.

- Use Immer if you want to write reducers in a mutating style.

</Recap>

<Challenges>

#### Dispatch actions from event handlers {/*dispatch-actions-from-event-handlers*/}

Currently, the event handlers in `ContactList.js` and `Chat.js` have `// TODO` comments. This is why typing into the input doesn't work, and clicking on the buttons doesn't change the selected recipient.

Replace these two `// TODO`s with the code to `dispatch` the corresponding actions. To see the expected shape and the type of the actions, check the reducer in `messengerReducer.js`. The reducer is already written so you won't need to change it. You only need to dispatch the actions in `ContactList.js` and `Chat.js`.

<Hint>

The `dispatch` function is already available in both of these components because it was passed as a prop. So you need to call `dispatch` with the corresponding action object.

To check the action object shape, you can look at the reducer and see which `action` fields it expects to see. For example, the `changed_selection` case in the reducer looks like this:
```

```js
case 'changed_selection': {
return {
...state,
selectedId: action.contactId
};
}
```

This means that your action object should have a `type: 'changed_selection'`. You also see the `action.contactId` being used, so you need to include a `contactId` property into your action.

</Hint>

<Sandpack>

```js App.js
import { useReducer } from 'react';
import Chat from './Chat.js';
import ContactList from './ContactList.js';
import { initialState, messengerReducer } from './messengerReducer';

export default function Messenger() {
const [state, dispatch] = useReducer(messengerReducer, initialState);
const message = state.message;
const contact = contacts.find((c) => c.id === state.selectedId);
return (
<div>
<ContactList
contacts={contacts}
selectedId={state.selectedId}
dispatch={dispatch}
/>
<Chat
key={contact.id}
message={message}
contact={contact}
dispatch={dispatch}
/>
```

```
</div>
);
}

const contacts = [
{id: 0, name: 'Taylor', email: 'taylor@mail.com'},
{id: 1, name: 'Alice', email: 'alice@mail.com'},
{id: 2, name: 'Bob', email: 'bob@mail.com'},
];
```

```js messengerReducer.js
export const initialState = {
selectedId: 0,
message: 'Hello',
};

export function messengerReducer(state, action) {
switch (action.type) {
case 'changed_selection': {
return {
...state,
selectedId: action.contactId,
message: '',
};
}
case 'edited_message': {
return {
...state,
message: action.message,
};
}
default: {
throw Error('Unknown action: ' + action.type);
}
}
}
```

```

```js ContactList.js
export default function ContactList({contacts, selectedId, dispatch}) {
return (
<section className="contact-list">
<ul>
{contacts.map((contact) => (
<li key={contact.id}>
<button
onClick={() => {
// TODO: dispatch changed_selection
}}>
{selectedId === contact.id ? <b>{contact.name}</b> : contact.name}
</button>
</li>
))}
</ul>
</section>
);
}
```

```js Chat.js
import { useState } from 'react';

export default function Chat({contact, message, dispatch}) {
return (
<section className="chat">
<textarea
value={message}
placeholder={'Chat to ' + contact.name}
onChange={(e) => {
// TODO: dispatch edited_message
// (Read the input value from e.target.value)
}}
/>

<br />
<button>Send to {contact.email}</button>
</section>
);
}
```

```css
.chat,
.contact-list {
float: left;
margin-bottom: 20px;
}
ul,
li {
list-style: none;
margin: 0;
padding: 0;
}
li button {
width: 100px;
padding: 10px;
margin-right: 10px;
}
textarea {
height: 150px;
}
```

</Sandpack>

<Solution>

From the reducer code, you can infer that actions need to look like this:

```js
// When the user presses "Alice"
dispatch({
type: 'changed_selection',

contactId: 1,
});

// When user types "Hello!"
dispatch({
type: 'edited_message',
message: 'Hello!',
});
```

Here is the example updated to dispatch the corresponding messages:

<Sandpack>

```js App.js
import { useReducer } from 'react';
import Chat from './Chat.js';
import ContactList from './ContactList.js';
import { initialState, messengerReducer } from './messengerReducer';

export default function Messenger() {
const [state, dispatch] = useReducer(messengerReducer, initialState);
const message = state.message;
const contact = contacts.find((c) => c.id === state.selectedId);
return (

<ContactList
contacts={contacts}
selectedId={state.selectedId}
dispatch={dispatch}
/>
<Chat
key={contact.id}
message={message}
contact={contact}
dispatch={dispatch}
/>

);

```
}

const contacts = [
{id: 0, name: 'Taylor', email: 'taylor@mail.com'},
{id: 1, name: 'Alice', email: 'alice@mail.com'},
{id: 2, name: 'Bob', email: 'bob@mail.com'},
];
```

```js messengerReducer.js
export const initialState = {
selectedId: 0,
message: 'Hello',
};

export function messengerReducer(state, action) {
switch (action.type) {
case 'changed_selection': {
return {
...state,
selectedId: action.contactId,
message: '',
};
}
case 'edited_message': {
return {
...state,
message: action.message,
};
}
default: {
throw Error('Unknown action: ' + action.type);
}
}
}
```

```js ContactList.js
```

```
export default function ContactList({contacts, selectedId, dispatch}) {
return (
<section className="contact-list">
<ul>
{contacts.map((contact) => (
<li key={contact.id}>
<button
onClick={() => {
dispatch({
type: 'changed_selection',
contactId: contact.id,
});
}}>
{selectedId === contact.id ? <b>{contact.name}</b> : contact.name}
</button>
</li>
))}
</ul>
</section>
);
}
```

```js Chat.js
import { useState } from 'react';

export default function Chat({contact, message, dispatch}) {
return (
<section className="chat">
<textarea
value={message}
placeholder={'Chat to ' + contact.name}
onChange={(e) => {
dispatch({
type: 'edited_message',
message: e.target.value,
});
```

}}
/>

<button>Send to {contact.email}</button>
</section>
);
}
```

```css
.chat,
.contact-list {
float: left;
margin-bottom: 20px;
}
ul,
li {
list-style: none;
margin: 0;
padding: 0;
}
li button {
width: 100px;
padding: 10px;
margin-right: 10px;
}
textarea {
height: 150px;
}
```

</Sandpack>

</Solution>

Clear the input on sending a message {/*clear-the-input-on-sending-a-message*/}

Currently, pressing "Send" doesn't do anything. Add an event handler to the "Send" button that will:
```

1. Show an `alert` with the recipient's email and the message.
2. Clear the message input.

<Sandpack>

```js App.js
import { useReducer } from 'react';
import Chat from './Chat.js';
import ContactList from './ContactList.js';
import { initialState, messengerReducer } from './messengerReducer';

export default function Messenger() {
const [state, dispatch] = useReducer(messengerReducer, initialState);
const message = state.message;
const contact = contacts.find((c) => c.id === state.selectedId);
return (
<div>
<ContactList
contacts={contacts}
selectedId={state.selectedId}
dispatch={dispatch}
/>
<Chat
key={contact.id}
message={message}
contact={contact}
dispatch={dispatch}
/>
</div>
);
}

const contacts = [
{id: 0, name: 'Taylor', email: 'taylor@mail.com'},
{id: 1, name: 'Alice', email: 'alice@mail.com'},
{id: 2, name: 'Bob', email: 'bob@mail.com'},
];
```

```js messengerReducer.js
export const initialState = {
selectedId: 0,
message: 'Hello',
};

export function messengerReducer(state, action) {
switch (action.type) {
case 'changed_selection': {
return {
...state,
selectedId: action.contactId,
message: '',
};
}
case 'edited_message': {
return {
...state,
message: action.message,
};
}
default: {
throw Error('Unknown action: ' + action.type);
}
}
}
```

```js ContactList.js
export default function ContactList({contacts, selectedId, dispatch}) {
return (
<section className="contact-list">
<ul>
{contacts.map((contact) => (
<li key={contact.id}>
<button
onClick={() => {
```

```
dispatch({
type: 'changed_selection',
contactId: contact.id,
});
}}>
{selectedId === contact.id ? <b>{contact.name}</b> : contact.name}
</button>
</li>
))}
</ul>
</section>
);
}
```

```js Chat.js active
import { useState } from 'react';

export default function Chat({contact, message, dispatch}) {
return (
<section className="chat">
<textarea
value={message}
placeholder={'Chat to ' + contact.name}
onChange={(e) => {
dispatch({
type: 'edited_message',
message: e.target.value,
});
}}
/>
<br />
<button>Send to {contact.email}</button>
</section>
);
}
```

```css
.chat,
.contact-list {
float: left;
margin-bottom: 20px;
}
ul,
li {
list-style: none;
margin: 0;
padding: 0;
}
li button {
width: 100px;
padding: 10px;
margin-right: 10px;
}
textarea {
height: 150px;
}
```

</Sandpack>

<Solution>

There are a couple of ways you could do it in the "Send" button event handler. One approach is to show an alert and then dispatch an `edited_message` action with an empty `message`:

<Sandpack>

```js App.js
import { useReducer } from 'react';
import Chat from './Chat.js';
import ContactList from './ContactList.js';
import { initialState, messengerReducer } from './messengerReducer';

export default function Messenger() {
const [state, dispatch] = useReducer(messengerReducer, initialState);
const message = state.message;
```

```
const contact = contacts.find((c) => c.id === state.selectedId);
return (
<div>
<ContactList
contacts={contacts}
selectedId={state.selectedId}
dispatch={dispatch}
/>
<Chat
key={contact.id}
message={message}
contact={contact}
dispatch={dispatch}
/>
</div>
);
}

const contacts = [
{id: 0, name: 'Taylor', email: 'taylor@mail.com'},
{id: 1, name: 'Alice', email: 'alice@mail.com'},
{id: 2, name: 'Bob', email: 'bob@mail.com'},
];
```

```js messengerReducer.js
export const initialState = {
selectedId: 0,
message: 'Hello',
};

export function messengerReducer(state, action) {
switch (action.type) {
case 'changed_selection': {
return {
...state,
selectedId: action.contactId,
```

```
message: '',
};
}
case 'edited_message': {
return {
...state,
message: action.message,
};
}
default: {
throw Error('Unknown action: ' + action.type);
}
}
}
```

```js ContactList.js
export default function ContactList({contacts, selectedId, dispatch}) {
return (
<section className="contact-list">
<ul>
{contacts.map((contact) => (
<li key={contact.id}>
<button
onClick={() => {
dispatch({
type: 'changed_selection',
contactId: contact.id,
});
}}>
{selectedId === contact.id ? <b>{contact.name}</b> : contact.name}
</button>
</li>
))}
</ul>
</section>
```

```
);
}
```

```js Chat.js active
import { useState } from 'react';

export default function Chat({contact, message, dispatch}) {
return (
<section className="chat">
<textarea
value={message}
placeholder={'Chat to ' + contact.name}
onChange={(e) => {
dispatch({
type: 'edited_message',
message: e.target.value,
});
}}
/>
<br />
<button
onClick={() => {
alert(`Sending "${message}" to ${contact.email}`);
dispatch({
type: 'edited_message',
message: '',
});
}}>
Send to {contact.email}
</button>
</section>
);
}
```

```css
```

.chat,

.contact-list {

float: left;

margin-bottom: 20px;

}

ul,

li {

list-style: none;

margin: 0;

padding: 0;

}

li button {

width: 100px;

padding: 10px;

margin-right: 10px;

}

textarea {

height: 150px;

}

```

</Sandpack>

This works and clears the input when you hit "Send".

However, _from the user's perspective_, sending a message is a different action than editing the field. To reflect that, you could instead create a _new_ action called `sent_message`, and handle it separately in the reducer:

<Sandpack>

```js App.js

import { useReducer } from 'react';

import Chat from './Chat.js';

import ContactList from './ContactList.js';

import { initialState, messengerReducer } from './messengerReducer';

export default function Messenger() {

const [state, dispatch] = useReducer(messengerReducer, initialState);

const message = state.message;

```
const contact = contacts.find((c) => c.id === state.selectedId);
return (
<div>
<ContactList
contacts={contacts}
selectedId={state.selectedId}
dispatch={dispatch}
/>
<Chat
key={contact.id}
message={message}
contact={contact}
dispatch={dispatch}
/>
</div>
);
}

const contacts = [
{id: 0, name: 'Taylor', email: 'taylor@mail.com'},
{id: 1, name: 'Alice', email: 'alice@mail.com'},
{id: 2, name: 'Bob', email: 'bob@mail.com'},
];
```

```js messengerReducer.js active
export const initialState = {
selectedId: 0,
message: 'Hello',
};

export function messengerReducer(state, action) {
switch (action.type) {
case 'changed_selection': {
return {
...state,
selectedId: action.contactId,
```

```
message: '',
};
}
case 'edited_message': {
return {
...state,
message: action.message,
};
}
case 'sent_message': {
return {
...state,
message: '',
};
}
default: {
throw Error('Unknown action: ' + action.type);
}
}
}
```

```js ContactList.js
export default function ContactList({contacts, selectedId, dispatch}) {
return (
<section className="contact-list">
<ul>
{contacts.map((contact) => (
<li key={contact.id}>
<button
onClick={() => {
dispatch({
type: 'changed_selection',
contactId: contact.id,
});
}}>
```

{selectedId === contact.id ? {contact.name} : contact.name}
</button>

))}

</section>
);
}
```

```js Chat.js active
import { useState } from 'react';

export default function Chat({contact, message, dispatch}) {
return (
<section className="chat">
<textarea
value={message}
placeholder={'Chat to ' + contact.name}
onChange={(e) => {
dispatch({
type: 'edited_message',
message: e.target.value,
});
}}
/>
<br />
<button
onClick={() => {
alert(`Sending "${message}" to ${contact.email}`);
dispatch({
type: 'sent_message',
});
}}>
Send to {contact.email}
</button>
</section>

);
}
```

```css
.chat,
.contact-list {
float: left;
margin-bottom: 20px;
}
ul,
li {
list-style: none;
margin: 0;
padding: 0;
}
li button {
width: 100px;
padding: 10px;
margin-right: 10px;
}
textarea {
height: 150px;
}
```

</Sandpack>

The resulting behavior is the same. But keep in mind that action types should ideally describe "what the user did" rather than "how you want the state to change". This makes it easier to later add more features.

With either solution, it's important that you **don't** place the `alert` inside a reducer. The reducer should be a pure function--it should only calculate the next state. It should not "do" anything, including displaying messages to the user. That should happen in the event handler. (To help catch mistakes like this, React will call your reducers multiple times in Strict Mode. This is why, if you put an alert in a reducer, it fires twice.)

</Solution>

Restore input values when switching between tabs
{/*restore-input-values-when-switching-between-tabs*/}
```

In this example, switching between different recipients always clears the text input:

```js
case 'changed_selection': {
return {
...state,
selectedId: action.contactId,
message: '' // Clears the input
};
```

This is because you don't want to share a single message draft between several recipients. But it would be better if your app "remembered" a draft for each contact separately, restoring them when you switch contacts.

Your task is to change the way the state is structured so that you remember a separate message draft _per contact_. You would need to make a few changes to the reducer, the initial state, and the components.

<Hint>

You can structure your state like this:

```js
export const initialState = {
selectedId: 0,
messages: {
0: 'Hello, Taylor', // Draft for contactId = 0
1: 'Hello, Alice', // Draft for contactId = 1
},
};
```

The `[key]: value` [computed property](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object_initializer#computed_property_names) syntax can help you update the `messages` object:

```js
{
...state.messages,
[id]: message
}
```

</Hint>

<Sandpack>

```js App.js
import { useReducer } from 'react';
import Chat from './Chat.js';
import ContactList from './ContactList.js';
import { initialState, messengerReducer } from './messengerReducer';

export default function Messenger() {
const [state, dispatch] = useReducer(messengerReducer, initialState);
const message = state.message;
const contact = contacts.find((c) => c.id === state.selectedId);
return (
<div>
<ContactList
contacts={contacts}
selectedId={state.selectedId}
dispatch={dispatch}
/>
<Chat
key={contact.id}
message={message}
contact={contact}
dispatch={dispatch}
/>
</div>
);
}

const contacts = [
{id: 0, name: 'Taylor', email: 'taylor@mail.com'},
{id: 1, name: 'Alice', email: 'alice@mail.com'},
{id: 2, name: 'Bob', email: 'bob@mail.com'},
];
```

```js messengerReducer.js
```

```
export const initialState = {
selectedId: 0,
message: 'Hello',
};

export function messengerReducer(state, action) {
switch (action.type) {
case 'changed_selection': {
return {
...state,
selectedId: action.contactId,
message: '',
};
}
case 'edited_message': {
return {
...state,
message: action.message,
};
}
case 'sent_message': {
return {
...state,
message: '',
};
}
default: {
throw Error('Unknown action: ' + action.type);
}
}
}
```

```js ContactList.js
export default function ContactList({contacts, selectedId, dispatch}) {
return (
<section className="contact-list">
```

```
<ul>
{contacts.map((contact) => (
<li key={contact.id}>
<button
onClick={() => {
dispatch({
type: 'changed_selection',
contactId: contact.id,
});
}}>
{selectedId === contact.id ? <b>{contact.name}</b> : contact.name}
</button>
</li>
))}
</ul>
</section>
);
}
```

```js Chat.js
import { useState } from 'react';

export default function Chat({contact, message, dispatch}) {
return (
<section className="chat">
<textarea
value={message}
placeholder={'Chat to ' + contact.name}
onChange={(e) => {
dispatch({
type: 'edited_message',
message: e.target.value,
});
}}
/>
<br />
```

<button
onClick={() => {
alert(`Sending "${message}" to ${contact.email}`);
dispatch({
type: 'sent_message',
});
}}>
Send to {contact.email}
</button>
</section>
);
}
```

```css
.chat,
.contact-list {
float: left;
margin-bottom: 20px;
}
ul,
li {
list-style: none;
margin: 0;
padding: 0;
}
li button {
width: 100px;
padding: 10px;
margin-right: 10px;
}
textarea {
height: 150px;
}
```

</Sandpack>
```

<Solution>

You'll need to update the reducer to store and update a separate message draft per contact:

```js
// When the input is edited
case 'edited_message': {
return {
// Keep other state like selection
...state,
messages: {
// Keep messages for other contacts
...state.messages,
// But change the selected contact's message
[state.selectedId]: action.message
}
};
}
```

You would also update the `Messenger` component to read the message for the currently selected contact:

```js
const message = state.messages[state.selectedId];
```

Here is the complete solution:

<Sandpack>

```js App.js
import { useReducer } from 'react';
import Chat from './Chat.js';
import ContactList from './ContactList.js';
import { initialState, messengerReducer } from './messengerReducer';

export default function Messenger() {
const [state, dispatch] = useReducer(messengerReducer, initialState);
const message = state.messages[state.selectedId];
const contact = contacts.find((c) => c.id === state.selectedId);
```

```
return (
<div>
<ContactList
contacts={contacts}
selectedId={state.selectedId}
dispatch={dispatch}
/>
<Chat
key={contact.id}
message={message}
contact={contact}
dispatch={dispatch}
/>
</div>
);
}

const contacts = [
{id: 0, name: 'Taylor', email: 'taylor@mail.com'},
{id: 1, name: 'Alice', email: 'alice@mail.com'},
{id: 2, name: 'Bob', email: 'bob@mail.com'},
];
```

```js messengerReducer.js
export const initialState = {
selectedId: 0,
messages: {
0: 'Hello, Taylor',
1: 'Hello, Alice',
2: 'Hello, Bob',
},
};

export function messengerReducer(state, action) {
switch (action.type) {
case 'changed_selection': {
```

```
return {
...state,
selectedId: action.contactId,
};
}
case 'edited_message': {
return {
...state,
messages: {
...state.messages,
[state.selectedId]: action.message,
},
};
}
case 'sent_message': {
return {
...state,
messages: {
...state.messages,
[state.selectedId]: '',
},
};
}
default: {
throw Error('Unknown action: ' + action.type);
}
}
}
```

```js ContactList.js
export default function ContactList({contacts, selectedId, dispatch}) {
return (
<section className="contact-list">
<ul>
{contacts.map((contact) => (
```

```
<li key={contact.id}>
<button
onClick={() => {
dispatch({
type: 'changed_selection',
contactId: contact.id,
});
}}>
{selectedId === contact.id ? <b>{contact.name}</b> : contact.name}
</button>
</li>
))}
</ul>
</section>
);
}
```

```js Chat.js
import { useState } from 'react';

export default function Chat({contact, message, dispatch}) {
return (
<section className="chat">
<textarea
value={message}
placeholder={'Chat to ' + contact.name}
onChange={(e) => {
dispatch({
type: 'edited_message',
message: e.target.value,
});
}}
/>
<br />
<button
onClick={() => {
```

```
alert(`Sending "${message}" to ${contact.email}`);
dispatch({
type: 'sent_message',
});
}}>
Send to {contact.email}
</button>
</section>
);
}
```

```css
.chat,
.contact-list {
float: left;
margin-bottom: 20px;
}
ul,
li {
list-style: none;
margin: 0;
padding: 0;
}
li button {
width: 100px;
padding: 10px;
margin-right: 10px;
}
textarea {
height: 150px;
}
```

</Sandpack>

Notably, you didn't need to change any of the event handlers to implement this different behavior. Without a reducer, you would have to change every event handler that updates the state.

</Solution>

Implement `useReducer` from scratch {/*implement-usereducer-from-scratch*/}

In the earlier examples, you imported the `useReducer` Hook from React. This time, you will implement _the `useReducer` Hook itself!_ Here is a stub to get you started. It shouldn't take more than 10 lines of code.

To test your changes, try typing into the input or select a contact.

<Hint>

Here is a more detailed sketch of the implementation:

```js
export function useReducer(reducer, initialState) {
const [state, setState] = useState(initialState);

function dispatch(action) {
// ???
}

return [state, dispatch];
}
```

Recall that a reducer function takes two arguments--the current state and the action object--and it returns the next state. What should your `dispatch` implementation do with it?

</Hint>

<Sandpack>

```js App.js
import { useReducer } from './MyReact.js';
import Chat from './Chat.js';
import ContactList from './ContactList.js';
import { initialState, messengerReducer } from './messengerReducer';

export default function Messenger() {
const [state, dispatch] = useReducer(messengerReducer, initialState);
const message = state.messages[state.selectedId];
const contact = contacts.find((c) => c.id === state.selectedId);
return (
<div>
```

```
<ContactList
contacts={contacts}
selectedId={state.selectedId}
dispatch={dispatch}
/>
<Chat
key={contact.id}
message={message}
contact={contact}
dispatch={dispatch}
/>
</div>
);
}

const contacts = [
{id: 0, name: 'Taylor', email: 'taylor@mail.com'},
{id: 1, name: 'Alice', email: 'alice@mail.com'},
{id: 2, name: 'Bob', email: 'bob@mail.com'},
];
```

```js messengerReducer.js
export const initialState = {
selectedId: 0,
messages: {
0: 'Hello, Taylor',
1: 'Hello, Alice',
2: 'Hello, Bob',
},
};

export function messengerReducer(state, action) {
switch (action.type) {
case 'changed_selection': {
return {
...state,
```

```
selectedId: action.contactId,
};
}
case 'edited_message': {
return {
...state,
messages: {
...state.messages,
[state.selectedId]: action.message,
},
};
}
case 'sent_message': {
return {
...state,
messages: {
...state.messages,
[state.selectedId]: '',
},
};
}
default: {
throw Error('Unknown action: ' + action.type);
}
}
}
```

```js MyReact.js active
import { useState } from 'react';

export function useReducer(reducer, initialState) {
const [state, setState] = useState(initialState);

// ???

return [state, dispatch];
}
```

```

```js ContactList.js hidden
export default function ContactList({contacts, selectedId, dispatch}) {
return (
<section className="contact-list">
<ul>
{contacts.map((contact) => (
<li key={contact.id}>
<button
onClick={() => {
dispatch({
type: 'changed_selection',
contactId: contact.id,
});
}}>
{selectedId === contact.id ? <b>{contact.name}</b> : contact.name}
</button>
</li>
))}
</ul>
</section>
);
}
```

```js Chat.js hidden
import { useState } from 'react';

export default function Chat({contact, message, dispatch}) {
return (
<section className="chat">
<textarea
value={message}
placeholder={'Chat to ' + contact.name}
onChange={(e) => {
dispatch({

```
type: 'edited_message',
message: e.target.value,
});
}}
/>
<br />
<button
onClick={() => {
alert(`Sending "${message}" to ${contact.email}`);
dispatch({
type: 'sent_message',
});
}}>
Send to {contact.email}
</button>
</section>
);
}
```

```css
.chat,
.contact-list {
float: left;
margin-bottom: 20px;
}
ul,
li {
list-style: none;
margin: 0;
padding: 0;
}
li button {
width: 100px;
padding: 10px;
margin-right: 10px;
```

}

textarea {

height: 150px;

}

```

</Sandpack>

<Solution>

Dispatching an action calls a reducer with the current state and the action, and stores the result as the next state. This is what it looks like in code:

<Sandpack>

```js App.js
import { useReducer } from './MyReact.js';

import Chat from './Chat.js';

import ContactList from './ContactList.js';

import { initialState, messengerReducer } from './messengerReducer';

export default function Messenger() {

const [state, dispatch] = useReducer(messengerReducer, initialState);

const message = state.messages[state.selectedId];

const contact = contacts.find((c) => c.id === state.selectedId);

return (

<ContactList

contacts={contacts}

selectedId={state.selectedId}

dispatch={dispatch}

/>

<Chat

key={contact.id}

message={message}

contact={contact}

dispatch={dispatch}

/>

);

```
}

const contacts = [
{id: 0, name: 'Taylor', email: 'taylor@mail.com'},
{id: 1, name: 'Alice', email: 'alice@mail.com'},
{id: 2, name: 'Bob', email: 'bob@mail.com'},
];
```

```js messengerReducer.js
export const initialState = {
selectedId: 0,
messages: {
0: 'Hello, Taylor',
1: 'Hello, Alice',
2: 'Hello, Bob',
},
};

export function messengerReducer(state, action) {
switch (action.type) {
case 'changed_selection': {
return {
...state,
selectedId: action.contactId,
};
}
case 'edited_message': {
return {
...state,
messages: {
...state.messages,
[state.selectedId]: action.message,
},
};
}
case 'sent_message': {
```

```
return {
...state,
messages: {
...state.messages,
[state.selectedId]: '',
},
};
}
default: {
throw Error('Unknown action: ' + action.type);
}
}
}
```

```js MyReact.js active
import { useState } from 'react';

export function useReducer(reducer, initialState) {
const [state, setState] = useState(initialState);

function dispatch(action) {
const nextState = reducer(state, action);
setState(nextState);
}

return [state, dispatch];
}
```

```js ContactList.js hidden
export default function ContactList({contacts, selectedId, dispatch}) {
return (
<section className="contact-list">
<ul>
{contacts.map((contact) => (
<li key={contact.id}>
<button
onClick={() => {
```

```
dispatch({
type: 'changed_selection',
contactId: contact.id,
});
}}>
{selectedId === contact.id ? <b>{contact.name}</b> : contact.name}
</button>
</li>
))}
</ul>
</section>
);
}
```

```js Chat.js hidden
import { useState } from 'react';

export default function Chat({contact, message, dispatch}) {
return (
<section className="chat">
<textarea
value={message}
placeholder={'Chat to ' + contact.name}
onChange={(e) => {
dispatch({
type: 'edited_message',
message: e.target.value,
});
}}
/>
<br />
<button
onClick={() => {
alert(`Sending "${message}" to ${contact.email}`);
dispatch({
type: 'sent_message',
```

});
}}>
Send to {contact.email}
</button>
</section>
);
}
```

```css
.chat,
.contact-list {
float: left;
margin-bottom: 20px;
}
ul,
li {
list-style: none;
margin: 0;
padding: 0;
}
li button {
width: 100px;
padding: 10px;
margin-right: 10px;
}
textarea {
height: 150px;
}
```

</Sandpack>

Though it doesn't matter in most cases, a slightly more accurate implementation looks like this:

```js
function dispatch(action) {
setState((s) => reducer(s, action));

}
```

This is because the dispatched actions are queued until the next render, [similar to the updater functions.](/learn/queueing-a-series-of-state-updates)

</Solution>

</Challenges>

---

title: Importing and Exporting Components

<Intro>

The magic of components lies in their reusability: you can create components that are composed of other components. But as you nest more and more components, it often makes sense to start splitting them into different files. This lets you keep your files easy to scan and reuse components in more places.

</Intro>

<YouWillLearn>

* What a root component file is
* How to import and export a component
* When to use default and named imports and exports
* How to import and export multiple components from one file
* How to split components into multiple files

</YouWillLearn>

The root component file {/*the-root-component-file*/}

In [Your First Component](/learn/your-first-component), you made a `Profile` component and a `Gallery` component that renders it:

<Sandpack>

```js
function Profile() {
return (
<img
src="https://i.imgur.com/MK3eW3As.jpg"
alt="Katherine Johnson"
/>

);

}

export default function Gallery() {

return (

<section>

<h1>Amazing scientists</h1>

<Profile />

<Profile />

<Profile />

</section>

);

}

```

```css

img { margin: 0 10px 10px 0; height: 90px; }

```

</Sandpack>

These currently live in a **root component file,** named `App.js` in this example. In [Create React App](https://create-react-app.dev/), your app lives in `src/App.js`. Depending on your setup, your root component could be in another file, though. If you use a framework with file-based routing, such as Next.js, your root component will be different for every page.

Exporting and importing a component {/*exporting-and-importing-a-component*/}

What if you want to change the landing screen in the future and put a list of science books there? Or place all the profiles somewhere else? It makes sense to move `Gallery` and `Profile` out of the root component file. This will make them more modular and reusable in other files. You can move a component in three steps:

1. **Make** a new JS file to put the components in.

2. **Export** your function component from that file (using either [default](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Statements/export#using_the_default_export) or [named](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Statements/export#using_named_exports) exports).

3. **Import** it in the file where you'll use the component (using the corresponding technique for importing [default](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Statements/import#importing_defaults) or [named](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Statements/import#import_a_single_export_from_a_module) exports).

Here both `Profile` and `Gallery` have been moved out of `App.js` into a new file called `Gallery.js`. Now you can change `App.js` to import `Gallery` from `Gallery.js`:
```

<Sandpack>

```js App.js
import Gallery from './Gallery.js';

export default function App() {
return (
<Gallery />
);
}
```

```js Gallery.js
function Profile() {
return (
<img
src="https://i.imgur.com/QIrZWGIs.jpg"
alt="Alan L. Hart"
/>
);
}

export default function Gallery() {
return (
<section>
<h1>Amazing scientists</h1>
<Profile />
<Profile />
<Profile />
</section>
);
}
```

```css
img { margin: 0 10px 10px 0; height: 90px; }
```

</Sandpack>

Notice how this example is broken down into two component files now:

1. `Gallery.js`:

- Defines the `Profile` component which is only used within the same file and is not exported.

- Exports the `Gallery` component as a **default export.**

2. `App.js`:

- Imports `Gallery` as a **default import** from `Gallery.js`.

- Exports the root `App` component as a **default export.**

<Note>

You may encounter files that leave off the `.js` file extension like so:

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

Either `'./Gallery.js'` or `'./Gallery'` will work with React, though the former is closer to how [native ES Modules](https://developer.mozilla.org/docs/Web/JavaScript/Guide/Modules) work.

</Note>

<DeepDive>

Default vs named exports {/*default-vs-named-exports*/}

There are two primary ways to export values with JavaScript: default exports and named exports. So far, our examples have only used default exports. But you can use one or both of them in the same file. **A file can have no more than one _default_ export, but it can have as many _named_ exports as you like.**

How you export your component dictates how you must import it. You will get an error if you try to import a default export the same way you would a named export! This chart can help you keep track:

Syntax	Export statement	Import statement
Default	`export default function Button() {}`	`import Button from './Button.js';`
Named	`export function Button() {}`	`import { Button } from './Button.js';`

When you write a _default_ import, you can put any name you want after `import`. For example, you could write `import Banana from './Button.js'` instead and it would still provide you with the same default export. In contrast, with named imports, the name has to match on both sides. That's why they are called _named_ imports!

People often use default exports if the file exports only one component, and use named exports if it exports multiple components and values. Regardless of which coding style you prefer, always give meaningful names to your component functions and the files that contain them. Components without names, like `export default () => {}`, are discouraged because they make debugging harder.

</DeepDive>

Exporting and importing multiple components from the same file {/*exporting-and-importing-multiple-components-from-the-same-file*/}

What if you want to show just one `Profile` instead of a gallery? You can export the `Profile` component, too. But `Gallery.js` already has a *default* export, and you can't have _two_ default exports. You could create a new file with a default export, or you could add a *named* export for `Profile`. **A file can only have one default export, but it can have numerous named exports!**

<Note>

To reduce the potential confusion between default and named exports, some teams choose to only stick to one style (default or named), or avoid mixing them in a single file. Do what works best for you!

</Note>

First, **export** `Profile` from `Gallery.js` using a named export (no `default` keyword):

```js
export function Profile() {
// ...
}
```

Then, **import** `Profile` from `Gallery.js` to `App.js` using a named import (with the curly braces):

```js
import { Profile } from './Gallery.js';
```

Finally, **render** `<Profile />` from the `App` component:

```js
export default function App() {
return <Profile />;
}
```

Now `Gallery.js` contains two exports: a default `Gallery` export, and a named `Profile` export. `App.js` imports both of them. Try editing `<Profile />` to `<Gallery />` and back in this example:

<Sandpack>

```js App.js
import Gallery from './Gallery.js';
import { Profile } from './Gallery.js';

export default function App() {
return (
<Profile />
);
}
```

```js Gallery.js
export function Profile() {
return (
<img
src="https://i.imgur.com/QIrZWGIs.jpg"
alt="Alan L. Hart"
/>
);
}

export default function Gallery() {
return (
<section>
<h1>Amazing scientists</h1>
<Profile />
<Profile />
<Profile />
</section>
);
}
```

```css
img { margin: 0 10px 10px 0; height: 90px; }
```

</Sandpack>

Now you're using a mix of default and named exports:

* `Gallery.js`:

- Exports the `Profile` component as a **named export called `Profile`.**

- Exports the `Gallery` component as a **default export.**

* `App.js`:

- Imports `Profile` as a **named import called `Profile`** from `Gallery.js`.

- Imports `Gallery` as a **default import** from `Gallery.js`.

- Exports the root `App` component as a **default export.**

<Recap>

On this page you learned:

* What a root component file is

* How to import and export a component

* When and how to use default and named imports and exports

* How to export multiple components from the same file

</Recap>

<Challenges>

Split the components further {/*split-the-components-further*/}

Currently, `Gallery.js` exports both `Profile` and `Gallery`, which is a bit confusing.

Move the `Profile` component to its own `Profile.js`, and then change the `App` component to render both `<Profile />` and `<Gallery />` one after another.

You may use either a default or a named export for `Profile`, but make sure that you use the corresponding import syntax in both `App.js` and `Gallery.js`! You can refer to the table from the deep dive above:

| Syntax | Export statement | Import statement |

| ---------- | ---------- | ---------- |

| Default | `export default function Button() {}` | `import Button from './Button.js';` |

| Named | `export function Button() {}` | `import { Button } from './Button.js';` |

<Hint>

Don't forget to import your components where they are called. Doesn't `Gallery` use `Profile`, too?

</Hint>

<Sandpack>

```js App.js
import Gallery from './Gallery.js';
import { Profile } from './Gallery.js';

export default function App() {
return (
<div>
<Profile />
</div>
);
}
```

```js Gallery.js active
// Move me to Profile.js!
export function Profile() {
return (
<img
src="https://i.imgur.com/QIrZWGIs.jpg"
alt="Alan L. Hart"
/>
);
}

export default function Gallery() {
return (
<section>
<h1>Amazing scientists</h1>
<Profile />
<Profile />
<Profile />
</section>
);
}
```

```js Profile.js
```

```css
img { margin: 0 10px 10px 0; height: 90px; }
```

</Sandpack>

After you get it working with one kind of exports, make it work with the other kind.

<Solution>

This is the solution with named exports:

<Sandpack>

```js App.js
import Gallery from './Gallery.js';
import { Profile } from './Profile.js';

export default function App() {
return (
<div>
<Profile />
<Gallery />
</div>
);
}
```

```js Gallery.js
import { Profile } from './Profile.js';

export default function Gallery() {
return (
<section>
<h1>Amazing scientists</h1>
<Profile />
<Profile />
<Profile />
</section>
);
}
```

```

```js Profile.js
export function Profile() {
return (
<img
src="https://i.imgur.com/QIrZWGIs.jpg"
alt="Alan L. Hart"
/>
);
}
```

```css
img { margin: 0 10px 10px 0; height: 90px; }
```

</Sandpack>

This is the solution with default exports:

<Sandpack>

```js App.js
import Gallery from './Gallery.js';
import Profile from './Profile.js';

export default function App() {
return (
<div>
<Profile />
<Gallery />
</div>
);
}
```

```js Gallery.js
import Profile from './Profile.js';

export default function Gallery() {

return (
<section>
<h1>Amazing scientists</h1>
<Profile />
<Profile />
<Profile />
</section>
);
}
```

```js Profile.js
export default function Profile() {
return (
<img
src="https://i.imgur.com/QIrZWGIs.jpg"
alt="Alan L. Hart"
/>
);
}
```

```css
img { margin: 0 10px 10px 0; height: 90px; }
```

</Sandpack>

</Solution>

</Challenges>

---

title: 'Removing Effect Dependencies'

<Intro>

When you write an Effect, the linter will verify that you've included every reactive value (like props and state) that the Effect reads in the list of your Effect's dependencies. This ensures that your Effect remains synchronized with the latest props and state of your component. Unnecessary dependencies may cause your Effect to run too often, or even create an infinite loop. Follow this guide to review and
```

remove unnecessary dependencies from your Effects.

</Intro>

<YouWillLearn>

- How to fix infinite Effect dependency loops
- What to do when you want to remove a dependency
- How to read a value from your Effect without "reacting" to it
- How and why to avoid object and function dependencies
- Why suppressing the dependency linter is dangerous, and what to do instead

</YouWillLearn>

Dependencies should match the code {/*dependencies-should-match-the-code*/}

When you write an Effect, you first specify how to [start and stop](/learn/lifecycle-of-reactive-effects#the-lifecycle-of-an-effect) whatever you want your Effect to be doing:

```js {5-7}
const serverUrl = 'https://localhost:1234';

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

Then, if you leave the Effect dependencies empty (`[]`), the linter will suggest the correct dependencies:

<Sandpack>

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

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

function ChatRoom({ roomId }) {
useEffect(() => {
const connection = createConnection(serverUrl, roomId);
```

```
connection.connect();
return () => connection.disconnect();
}, []); // <-- Fix the mistake here!
return <h1>Welcome to the {roomId} room!</h1>;
}

export default function App() {
const [roomId, setRoomId] = useState('general');
return (
<>
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
</select>
</label>
<hr />
<ChatRoom roomId={roomId} />
</>
);
}
```

```js chat.js
export function createConnection(serverUrl, roomId) {
// A real implementation would actually connect to the server
return {
connect() {
console.log('■ Connecting to "' + roomId + '" room at ' + serverUrl + '...');
},
disconnect() {
console.log('■ Disconnected from "' + roomId + '" room at ' + serverUrl);
```

```
}
};
}
```

```css
input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }
```

</Sandpack>

Fill them in according to what the linter says:

```js {6}
function ChatRoom({ roomId }) {
useEffect(() => {
const connection = createConnection(serverUrl, roomId);
connection.connect();
return () => connection.disconnect();
}, [roomId]); // ■ All dependencies declared
// ...
}
```

[Effects "react" to reactive values.](/learn/lifecycle-of-reactive-effects#effects-react-to-reactive-values) Since `roomId` is a reactive value (it can change due to a re-render), the linter verifies that you've specified it as a dependency. If `roomId` receives a different value, React will re-synchronize your Effect. This ensures that the chat stays connected to the selected room and "reacts" to the dropdown:

<Sandpack>

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

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

function ChatRoom({ roomId }) {
useEffect(() => {
const connection = createConnection(serverUrl, roomId);
connection.connect();
```

```
return () => connection.disconnect();
}, [roomId]);
return <h1>Welcome to the {roomId} room!</h1>;
}

export default function App() {
const [roomId, setRoomId] = useState('general');
return (
<>
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
</select>
</label>
<hr />
<ChatRoom roomId={roomId} />
</>
);
}
```

```js chat.js
export function createConnection(serverUrl, roomId) {
// A real implementation would actually connect to the server
return {
connect() {
console.log('■ Connecting to "' + roomId + '" room at ' + serverUrl + '...');
},
disconnect() {
console.log('■ Disconnected from "' + roomId + '" room at ' + serverUrl);
}
```

```
};
}
```

```css
input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }
```

</Sandpack>

To remove a dependency, prove that it's not a dependency
{/*to-remove-a-dependency-prove-that-its-not-a-dependency*/}

Notice that you can't "choose" the dependencies of your Effect. Every <CodeStep step={2}>reactive value</CodeStep> used by your Effect's code must be declared in your dependency list. The dependency list is determined by the surrounding code:

```js [[2, 3, "roomId"], [2, 5, "roomId"], [2, 8, "roomId"]]
const serverUrl = 'https://localhost:1234';

function ChatRoom({ roomId }) { // This is a reactive value
useEffect(() => {
const connection = createConnection(serverUrl, roomId); // This Effect reads that reactive value
connection.connect();
return () => connection.disconnect();
}, [roomId]); // ■ So you must specify that reactive value as a dependency of your Effect
// ...
}
```

[Reactive values](/learn/lifecycle-of-reactive-effects#all-variables-declared-in-the-component-body-are-reactive) include props and all variables and functions declared directly inside of your component. Since `roomId` is a reactive value, you can't remove it from the dependency list. The linter wouldn't allow it:

```js {8}
const serverUrl = 'https://localhost:1234';

function ChatRoom({ roomId }) {
useEffect(() => {
const connection = createConnection(serverUrl, roomId);
connection.connect();
```

return () => connection.disconnect();

}, []); // ■ React Hook useEffect has a missing dependency: 'roomId'

// ...

}

```

And the linter would be right! Since `roomId` may change over time, this would introduce a bug in your code.

**To remove a dependency, "prove" to the linter that it *doesn't need* to be a dependency.** For example, you can move `roomId` out of your component to prove that it's not reactive and won't change on re-renders:

```js {2,9}

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

const roomId = 'music'; // Not a reactive value anymore

function ChatRoom() {

useEffect(() => {

const connection = createConnection(serverUrl, roomId);

connection.connect();

return () => connection.disconnect();

}, []); // ■ All dependencies declared

// ...

}

```

Now that `roomId` is not a reactive value (and can't change on a re-render), it doesn't need to be a dependency:

<Sandpack>

```js

import { useState, useEffect } from 'react';

import { createConnection } from './chat.js';

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

const roomId = 'music';

export default function ChatRoom() {

useEffect(() => {

const connection = createConnection(serverUrl, roomId);

connection.connect();

```
return () => connection.disconnect();
}, []);
return <h1>Welcome to the {roomId} room!</h1>;
}
```

```js chat.js
export function createConnection(serverUrl, roomId) {
// A real implementation would actually connect to the server
return {
connect() {
console.log('■ Connecting to "' + roomId + '" room at ' + serverUrl + '...');
},
disconnect() {
console.log('■ Disconnected from "' + roomId + '" room at ' + serverUrl);
}
};
}
```

```css
input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }
```

</Sandpack>

This is why you could now specify an [empty (`[]`) dependency list.](/learn/lifecycle-of-reactive-effects#what-an-effect-with-empty-dependencies-means) Your Effect *really doesn't* depend on any reactive value anymore, so it *really doesn't* need to re-run when any of the component's props or state change.

To change the dependencies, change the code {/*to-change-the-dependencies-change-the-code*/}

You might have noticed a pattern in your workflow:

1. First, you **change the code** of your Effect or how your reactive values are declared.
2. Then, you follow the linter and adjust the dependencies to **match the code you have changed.**
3. If you're not happy with the list of dependencies, you **go back to the first step** (and change the code again).

The last part is important. **If you want to change the dependencies, change the surrounding code first.** You can think of the dependency list as [a list of all the reactive values used by your Effect's cod e.](/learn/lifecycle-of-reactive-effects#react-verifies-that-you-specified-every-reactive-value-as-a-depen dency) You don't *choose* what to put on that list. The list *describes* your code. To change the dependency list, change the code.

This might feel like solving an equation. You might start with a goal (for example, to remove a dependency), and you need to "find" the code matching that goal. Not everyone finds solving equations fun, and the same thing could be said about writing Effects! Luckily, there is a list of common recipes that you can try below.

<Pitfall>

If you have an existing codebase, you might have some Effects that suppress the linter like this:

```js {3-4}
useEffect(() => {
// ...
// ■ Avoid suppressing the linter like this:
// eslint-ignore-next-line react-hooks/exhaustive-deps
}, []);
```

When dependencies don't match the code, there is a very high risk of introducing bugs. By suppressing the linter, you "lie" to React about the values your Effect depends on.

Instead, use the techniques below.

</Pitfall>

<DeepDive>

Why is suppressing the dependency linter so dangerous?
{/*why-is-suppressing-the-dependency-linter-so-dangerous*/}

Suppressing the linter leads to very unintuitive bugs that are hard to find and fix. Here's one example:

<Sandpack>

```js
import { useState, useEffect } from 'react';

export default function Timer() {
const [count, setCount] = useState(0);
const [increment, setIncrement] = useState(1);

function onTick() {
setCount(count + increment);
```

```
}

useEffect(() => {
const id = setInterval(onTick, 1000);
return () => clearInterval(id);
// eslint-disable-next-line react-hooks/exhaustive-deps
}, []);

return (
<>
<h1>
Counter: {count}
<button onClick={() => setCount(0)}>Reset</button>
</h1>
<hr />
<p>
Every second, increment by:
<button disabled={increment === 0} onClick={() => {
setIncrement(i => i - 1);
}}>–</button>
<b>{increment}</b>
<button onClick={() => {
setIncrement(i => i + 1);
}}>+</button>
</p>
</>
);
}
```

```css
button { margin: 10px; }
```

</Sandpack>

Let's say that you wanted to run the Effect "only on mount". You've read that [empty (`[]`) dependencies](/learn/lifecycle-of-reactive-effects#what-an-effect-with-empty-dependencies-means) do that, so you've decided to ignore the linter, and forcefully specified `[]` as the dependencies.

This counter was supposed to increment every second by the amount configurable with the two buttons. However, since you "lied" to React that this Effect doesn't depend on anything, React forever keeps using the `onTick` function from the initial render. [During that render,](/learn/state-as-a-snapshot#rendering-takes-a-snapshot-in-time) `count` was `0` and `increment` was `1`. This is why `onTick` from that render always calls `setCount(0 + 1)` every second, and you always see `1`. Bugs like this are harder to fix when they're spread across multiple components.

There's always a better solution than ignoring the linter! To fix this code, you need to add `onTick` to the dependency list. (To ensure the interval is only setup once, [make `onTick` an Effect Event.](/learn/separating-events-from-effects#reading-latest-props-and-state-with-effect-events))

We recommend treating the dependency lint error as a compilation error. If you don't suppress it, you will never see bugs like this. The rest of this page documents the alternatives for this and other cases.

</DeepDive>

Removing unnecessary dependencies {/*removing-unnecessary-dependencies*/}

Every time you adjust the Effect's dependencies to reflect the code, look at the dependency list. Does it make sense for the Effect to re-run when any of these dependencies change? Sometimes, the answer is "no":

* You might want to re-execute *different parts* of your Effect under different conditions.

* You might want to only read the *latest value* of some dependency instead of "reacting" to its changes.

* A dependency may change too often *unintentionally* because it's an object or a function.

To find the right solution, you'll need to answer a few questions about your Effect. Let's walk through them.

Should this code move to an event handler? {/*should-this-code-move-to-an-event-handler*/}

The first thing you should think about is whether this code should be an Effect at all.

Imagine a form. On submit, you set the `submitted` state variable to `true`. You need to send a POST request and show a notification. You've put this logic inside an Effect that "reacts" to `submitted` being `true`:

```js {6-8}
function Form() {
const [submitted, setSubmitted] = useState(false);

useEffect(() => {
if (submitted) {
// ■ Avoid: Event-specific logic inside an Effect
post('/api/register');
showNotification('Successfully registered!');
```

```
}
}, [submitted]);

function handleSubmit() {
setSubmitted(true);
}

// ...
}
```

Later, you want to style the notification message according to the current theme, so you read the current theme. Since `theme` is declared in the component body, it is a reactive value, so you add it as a dependency:

```js {3,9,11}
function Form() {
const [submitted, setSubmitted] = useState(false);
const theme = useContext(ThemeContext);

useEffect(() => {
if (submitted) {
// ■ Avoid: Event-specific logic inside an Effect
post('/api/register');
showNotification('Successfully registered!', theme);
}
}, [submitted, theme]); // ■ All dependencies declared

function handleSubmit() {
setSubmitted(true);
}

// ...
}
```

By doing this, you've introduced a bug. Imagine you submit the form first and then switch between Dark and Light themes. The `theme` will change, the Effect will re-run, and so it will display the same notification again!

The problem here is that this shouldn't be an Effect in the first place. You want to send this POST request and show the notification in response to *submitting the form,* which is a particular interaction. To run some code in response to particular interaction, put that logic directly into the corresponding event handler:

```js {6-7}
function Form() {
const theme = useContext(ThemeContext);

function handleSubmit() {
// ■ Good: Event-specific logic is called from event handlers
post('/api/register');
showNotification('Successfully registered!', theme);
}

// ...
}
```

Now that the code is in an event handler, it's not reactive--so it will only run when the user submits the form. Read more about [choosing between event handlers and Effects](/learn/separating-events-from-effects#reactive-values-and-reactive-logic) and [how to delete unnecessary Effects.](/learn/you-might-not-need-an-effect)

Is your Effect doing several unrelated things? {/*is-your-effect-doing-several-unrelated-things*/}

The next question you should ask yourself is whether your Effect is doing several unrelated things.

Imagine you're creating a shipping form where the user needs to choose their city and area. You fetch the list of `cities` from the server according to the selected `country` to show them in a dropdown:

```js
function ShippingForm({ country }) {
const [cities, setCities] = useState(null);
const [city, setCity] = useState(null);

useEffect(() => {
let ignore = false;
fetch(`/api/cities?country=${country}`)
.then(response => response.json())
.then(json => {
if (!ignore) {
setCities(json);
}
});
return () => {
ignore = true;
```

```
};
}, [country]); // ■ All dependencies declared

// ...
```

This is a good example of [fetching data in an Effect.](/learn/you-might-not-need-an-effect#fetching-data) You are synchronizing the `cities` state with the network according to the `country` prop. You can't do this in an event handler because you need to fetch as soon as `ShippingForm` is displayed and whenever the `country` changes (no matter which interaction causes it).

Now let's say you're adding a second select box for city areas, which should fetch the `areas` for the currently selected `city`. You might start by adding a second `fetch` call for the list of areas inside the same Effect:

```js {15-24,28}
function ShippingForm({ country }) {
const [cities, setCities] = useState(null);
const [city, setCity] = useState(null);
const [areas, setAreas] = useState(null);

useEffect(() => {
let ignore = false;
fetch(`/api/cities?country=${country}`)
.then(response => response.json())
.then(json => {
if (!ignore) {
setCities(json);
}
});
// ■ Avoid: A single Effect synchronizes two independent processes
if (city) {
fetch(`/api/areas?city=${city}`)
.then(response => response.json())
.then(json => {
if (!ignore) {
setAreas(json);
}
});
}
```

```
return () => {
ignore = true;
};
}, [country, city]); // ■ All dependencies declared

// ...
```

However, since the Effect now uses the `city` state variable, you've had to add `city` to the list of dependencies. That, in turn, introduced a problem: when the user selects a different city, the Effect will re-run and call `fetchCities(country)`. As a result, you will be unnecessarily refetching the list of cities many times.

The problem with this code is that you're synchronizing two different unrelated things:

1. You want to synchronize the `cities` state to the network based on the `country` prop.
1. You want to synchronize the `areas` state to the network based on the `city` state.

Split the logic into two Effects, each of which reacts to the prop that it needs to synchronize with:

```js {19-33}
function ShippingForm({ country }) {
const [cities, setCities] = useState(null);
useEffect(() => {
let ignore = false;
fetch(`/api/cities?country=${country}`)
.then(response => response.json())
.then(json => {
if (!ignore) {
setCities(json);
}
});
return () => {
ignore = true;
};
}, [country]); // ■ All dependencies declared

const [city, setCity] = useState(null);
const [areas, setAreas] = useState(null);
useEffect(() => {
if (city) {
```

```
let ignore = false;
fetch(`/api/areas?city=${city}`)
.then(response => response.json())
.then(json => {
if (!ignore) {
setAreas(json);
}
});
return () => {
ignore = true;
};
}
}, [city]); // ■ All dependencies declared

// ...
```

Now the first Effect only re-runs if the `country` changes, while the second Effect re-runs when the `city` changes. You've separated them by purpose: two different things are synchronized by two separate Effects. Two separate Effects have two separate dependency lists, so they won't trigger each other unintentionally.

The final code is longer than the original, but splitting these Effects is still correct. [Each Effect should represent an independent synchronization process.](/learn/lifecycle-of-reactive-effects#each-effect-represents-a-separate-synchronization-process) In this example, deleting one Effect doesn't break the other Effect's logic. This means they *synchronize different things,* and it's good to split them up. If you're concerned about duplication, you can improve this code by [extracting repetitive logic into a custom Hook.](/learn/reusing-logic-with-custom-hooks#when-to-use-custom-hooks)

Are you reading some state to calculate the next state?
{/*are-you-reading-some-state-to-calculate-the-next-state*/}

This Effect updates the `messages` state variable with a newly created array every time a new message arrives:

```js {2,6-8}
function ChatRoom({ roomId }) {
const [messages, setMessages] = useState([]);
useEffect(() => {
const connection = createConnection();
connection.connect();
connection.on('message', (receivedMessage) => {
setMessages([...messages, receivedMessage]);
```

});

// ...

```

It uses the `messages` variable to [create a new array](/learn/updating-arrays-in-state) starting with all the existing messages and adds the new message at the end. However, since `messages` is a reactive value read by an Effect, it must be a dependency:

```js {7,10}
function ChatRoom({ roomId }) {
const [messages, setMessages] = useState([]);
useEffect(() => {
const connection = createConnection();
connection.connect();
connection.on('message', (receivedMessage) => {
setMessages([...messages, receivedMessage]);
});
return () => connection.disconnect();
}, [roomId, messages]); // ■ All dependencies declared
// ...
```

And making `messages` a dependency introduces a problem.

Every time you receive a message, `setMessages()` causes the component to re-render with a new `messages` array that includes the received message. However, since this Effect now depends on `messages`, this will *also* re-synchronize the Effect. So every new message will make the chat re-connect. The user would not like that!

To fix the issue, don't read `messages` inside the Effect. Instead, pass an [updater function](/reference/react/useState#updating-state-based-on-the-previous-state) to `setMessages`:

```js {7,10}
function ChatRoom({ roomId }) {
const [messages, setMessages] = useState([]);
useEffect(() => {
const connection = createConnection();
connection.connect();
connection.on('message', (receivedMessage) => {
setMessages(msgs => [...msgs, receivedMessage]);
});
```
```

return () => connection.disconnect();

}, [roomId]); // ■ All dependencies declared

// ...

```

**Notice how your Effect does not read the `messages` variable at all now.** You only need to pass an updater function like `msgs => [...msgs, receivedMessage]`. React [puts your updater function in a queue](/learn/queueing-a-series-of-state-updates) and will provide the `msgs` argument to it during the next render. This is why the Effect itself doesn't need to depend on `messages` anymore. As a result of this fix, receiving a chat message will no longer make the chat re-connect.

### Do you want to read a value without "reacting" to its changes?
{/*do-you-want-to-read-a-value-without-reacting-to-its-changes*/}

<Wip>

This section describes an **experimental API that has not yet been released** in a stable version of React.

</Wip>

Suppose that you want to play a sound when the user receives a new message unless `isMuted` is `true`:

```js {3,10-12}
function ChatRoom({ roomId }) {

const [messages, setMessages] = useState([]);

const [isMuted, setIsMuted] = useState(false);

useEffect(() => {

const connection = createConnection();

connection.connect();

connection.on('message', (receivedMessage) => {

setMessages(msgs => [...msgs, receivedMessage]);

if (!isMuted) {

playSound();

}

});

// ...

```

Since your Effect now uses `isMuted` in its code, you have to add it to the dependencies:

```js {10,15}

```
function ChatRoom({ roomId }) {
const [messages, setMessages] = useState([]);
const [isMuted, setIsMuted] = useState(false);

useEffect(() => {
const connection = createConnection();
connection.connect();
connection.on('message', (receivedMessage) => {
setMessages(msgs => [...msgs, receivedMessage]);
if (!isMuted) {
playSound();
}
});
return () => connection.disconnect();
}, [roomId, isMuted]); // ■ All dependencies declared
// ...
```

The problem is that every time `isMuted` changes (for example, when the user presses the "Muted" toggle), the Effect will re-synchronize, and reconnect to the chat. This is not the desired user experience! (In this example, even disabling the linter would not work--if you do that, `isMuted` would get "stuck" with its old value.)

To solve this problem, you need to extract the logic that shouldn't be reactive out of the Effect. You don't want this Effect to "react" to the changes in `isMuted`. [Move this non-reactive piece of logic into an Effect Event:](/learn/separating-events-from-effects#declaring-an-effect-event)

```js {1,7-12,18,21}
import { useState, useEffect, useEffectEvent } from 'react';

function ChatRoom({ roomId }) {
const [messages, setMessages] = useState([]);
const [isMuted, setIsMuted] = useState(false);

const onMessage = useEffectEvent(receivedMessage => {
setMessages(msgs => [...msgs, receivedMessage]);
if (!isMuted) {
playSound();
}
});

useEffect(() => {
```

```
const connection = createConnection();
connection.connect();
connection.on('message', (receivedMessage) => {
onMessage(receivedMessage);
});
return () => connection.disconnect();
}, [roomId]); // ■ All dependencies declared
// ...
```

Effect Events let you split an Effect into reactive parts (which should "react" to reactive values like `roomId` and their changes) and non-reactive parts (which only read their latest values, like `onMessage` reads `isMuted`). **Now that you read `isMuted` inside an Effect Event, it doesn't need to be a dependency of your Effect.** As a result, the chat won't re-connect when you toggle the "Muted" setting on and off, solving the original issue!

Wrapping an event handler from the props {/*wrapping-an-event-handler-from-the-props*/}

You might run into a similar problem when your component receives an event handler as a prop:

```js {1,8,11}
function ChatRoom({ roomId, onReceiveMessage }) {
const [messages, setMessages] = useState([]);

useEffect(() => {
const connection = createConnection();
connection.connect();
connection.on('message', (receivedMessage) => {
onReceiveMessage(receivedMessage);
});
return () => connection.disconnect();
}, [roomId, onReceiveMessage]); // ■ All dependencies declared
// ...
```

Suppose that the parent component passes a *different* `onReceiveMessage` function on every render:

```js {3-5}
<ChatRoom
roomId={roomId}
onReceiveMessage={receivedMessage => {
```

// ...

}}

/>

```

Since `onReceiveMessage` is a dependency, it would cause the Effect to re-synchronize after every parent re-render. This would make it re-connect to the chat. To solve this, wrap the call in an Effect Event:

```js {4-6,12,15}

function ChatRoom({ roomId, onReceiveMessage }) {

const [messages, setMessages] = useState([]);

const onMessage = useEffectEvent(receivedMessage => {

onReceiveMessage(receivedMessage);

});

useEffect(() => {

const connection = createConnection();

connection.connect();

connection.on('message', (receivedMessage) => {

onMessage(receivedMessage);

});

return () => connection.disconnect();

}, [roomId]); // ■ All dependencies declared

// ...

```

Effect Events aren't reactive, so you don't need to specify them as dependencies. As a result, the chat will no longer re-connect even if the parent component passes a function that's different on every re-render.

Separating reactive and non-reactive code {/*separating-reactive-and-non-reactive-code*/}

In this example, you want to log a visit every time `roomId` changes. You want to include the current `notificationCount` with every log, but you *don't* want a change to `notificationCount` to trigger a log event.

The solution is again to split out the non-reactive code into an Effect Event:

```js {2-4,7}

function Chat({ roomId, notificationCount }) {

const onVisit = useEffectEvent(visitedRoomId => {

```
logVisit(visitedRoomId, notificationCount);
});

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

You want your logic to be reactive with regards to `roomId`, so you read `roomId` inside of your Effect. However, you don't want a change to `notificationCount` to log an extra visit, so you read `notificationCount` inside of the Effect Event. [Learn more about reading the latest props and state from Effects using Effect Events.](/learn/separating-events-from-effects#reading-latest-props-and-state-with-effect-events)

Does some reactive value change unintentionally? {/*does-some-reactive-value-change-unintentionally*/}

Sometimes, you *do* want your Effect to "react" to a certain value, but that value changes more often than you'd like--and might not reflect any actual change from the user's perspective. For example, let's say that you create an `options` object in the body of your component, and then read that object from inside of your Effect:

```js {3-6,9}
function ChatRoom({ roomId }) {
// ...
const options = {
serverUrl: serverUrl,
roomId: roomId
};

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

This object is declared in the component body, so it's a [reactive value.](/learn/lifecycle-of-reactive-effects#effects-react-to-reactive-values) When you read a reactive value like this inside an Effect, you declare it as a dependency. This ensures your Effect "reacts" to its changes:

```js {3,6}
```

```
// ...
useEffect(() => {
const connection = createConnection(options);
connection.connect();
return () => connection.disconnect();
}, [options]); // ■ All dependencies declared
// ...
```

It is important to declare it as a dependency! This ensures, for example, that if the `roomId` changes, your Effect will re-connect to the chat with the new `options`. However, there is also a problem with the code above. To see it, try typing into the input in the sandbox below, and watch what happens in the console:

<Sandpack>

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

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

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

// Temporarily disable the linter to demonstrate the problem
// eslint-disable-next-line react-hooks/exhaustive-deps
const options = {
serverUrl: serverUrl,
roomId: roomId
};

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

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

```
</>
);
}

export default function App() {
const [roomId, setRoomId] = useState('general');
return (
<>
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
</select>
</label>
<hr />
<ChatRoom roomId={roomId} />
</>
);
}
```

```js chat.js
export function createConnection({ serverUrl, roomId }) {
// A real implementation would actually connect to the server
return {
connect() {
console.log('■ Connecting to "' + roomId + '" room at ' + serverUrl + '...');
},
disconnect() {
console.log('■ Disconnected from "' + roomId + '" room at ' + serverUrl);
}
};
```

}
```

```css
input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }
```

</Sandpack>

In the sandbox above, the input only updates the `message` state variable. From the user's perspective, this should not affect the chat connection. However, every time you update the `message`, your component re-renders. When your component re-renders, the code inside of it runs again from scratch.

A new `options` object is created from scratch on every re-render of the `ChatRoom` component. React sees that the `options` object is a *different object* from the `options` object created during the last render. This is why it re-synchronizes your Effect (which depends on `options`), and the chat re-connects as you type.

This problem only affects objects and functions. In JavaScript, each newly created object and function is considered distinct from all the others. It doesn't matter that the contents inside of them may be the same!

```js {7-8}
// During the first render
const options1 = { serverUrl: 'https://localhost:1234', roomId: 'music' };

// During the next render
const options2 = { serverUrl: 'https://localhost:1234', roomId: 'music' };

// These are two different objects!
console.log(Object.is(options1, options2)); // false
```

Object and function dependencies can make your Effect re-synchronize more often than you need.

This is why, whenever possible, you should try to avoid objects and functions as your Effect's dependencies. Instead, try moving them outside the component, inside the Effect, or extracting primitive values out of them.

Move static objects and functions outside your component {/*move-static-objects-and-functions-outside-your-component*/}

If the object does not depend on any props and state, you can move that object outside your component:

```js {1-4,13}

```
const options = {
serverUrl: 'https://localhost:1234',
roomId: 'music'
};

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

useEffect(() => {
const connection = createConnection(options);
connection.connect();
return () => connection.disconnect();
}, []); // ■ All dependencies declared
// ...
```

This way, you *prove* to the linter that it's not reactive. It can't change as a result of a re-render, so it doesn't need to be a dependency. Now re-rendering `ChatRoom` won't cause your Effect to re-synchronize.

This works for functions too:

```js {1-6,12}
function createOptions() {
return {
serverUrl: 'https://localhost:1234',
roomId: 'music'
};
}

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

useEffect(() => {
const options = createOptions();
const connection = createConnection();
connection.connect();
return () => connection.disconnect();
}, []); // ■ All dependencies declared
// ...
```

Since `createOptions` is declared outside your component, it's not a reactive value. This is why it doesn't need to be specified in your Effect's dependencies, and why it won't ever cause your Effect to re-synchronize.

Move dynamic objects and functions inside your Effect
{/*move-dynamic-objects-and-functions-inside-your-effect*/}

If your object depends on some reactive value that may change as a result of a re-render, like a `roomId` prop, you can't pull it *outside* your component. You can, however, move its creation *inside* of your Effect's code:

```js {7-10,11,14}
const serverUrl = 'https://localhost:1234';

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

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

Now that `options` is declared inside of your Effect, it is no longer a dependency of your Effect. Instead, the only reactive value used by your Effect is `roomId`. Since `roomId` is not an object or function, you can be sure that it won't be *unintentionally* different. In JavaScript, numbers and strings are compared by their content:

```js {7-8}
// During the first render
const roomId1 = 'music';

// During the next render
const roomId2 = 'music';

// These two strings are the same!
console.log(Object.is(roomId1, roomId2)); // true
```

Thanks to this fix, the chat no longer re-connects if you edit the input:

<Sandpack>

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

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

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

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

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

export default function App() {
const [roomId, setRoomId] = useState('general');
return (
<>
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
```

>
<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
</select>
</label>
<hr />
<ChatRoom roomId={roomId} />
</>
);
}
```

```js chat.js
export function createConnection({ serverUrl, roomId }) {
// A real implementation would actually connect to the server
return {
connect() {
console.log('■ Connecting to "' + roomId + '" room at ' + serverUrl + '...');
},
disconnect() {
console.log('■ Disconnected from "' + roomId + '" room at ' + serverUrl);
}
};
}
```

```css
input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }
```

</Sandpack>

However, it *does* re-connect when you change the `roomId` dropdown, as you would expect.

This works for functions, too:

```js {7-12,14}

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

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

useEffect(() => {
function createOptions() {
return {
serverUrl: serverUrl,
roomId: roomId
};
}

const options = createOptions();
const connection = createConnection(options);
connection.connect();
return () => connection.disconnect();
}, [roomId]); // ■ All dependencies declared
// ...
```

You can write your own functions to group pieces of logic inside your Effect. As long as you also declare them *inside* your Effect, they're not reactive values, and so they don't need to be dependencies of your Effect.

Read primitive values from objects {/*read-primitive-values-from-objects*/}

Sometimes, you may receive an object from props:

```js {1,5,8}
function ChatRoom({ options }) {
const [message, setMessage] = useState('');

useEffect(() => {
const connection = createConnection(options);
connection.connect();
return () => connection.disconnect();
}, [options]); // ■ All dependencies declared
// ...
```

The risk here is that the parent component will create the object during rendering:

```js {3-6}
<ChatRoom
roomId={roomId}
options={{
serverUrl: serverUrl,
roomId: roomId
}}
/>
```

This would cause your Effect to re-connect every time the parent component re-renders. To fix this, read information from the object *outside* the Effect, and avoid having object and function dependencies:

```js {4,7-8,12}
function ChatRoom({ options }) {
const [message, setMessage] = useState('');

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

The logic gets a little repetitive (you read some values from an object outside an Effect, and then create an object with the same values inside the Effect). But it makes it very explicit what information your Effect *actually* depends on. If an object is re-created unintentionally by the parent component, the chat would not re-connect. However, if `options.roomId` or `options.serverUrl` really are different, the chat would re-connect.

Calculate primitive values from functions {/*calculate-primitive-values-from-functions*/}

The same approach can work for functions. For example, suppose the parent component passes a function:

```js {3-8}
```

<ChatRoom

roomId={roomId}

getOptions={() => {

return {

serverUrl: serverUrl,

roomId: roomId

};

}}

/>

```

To avoid making it a dependency (and causing it to re-connect on re-renders), call it outside the Effect. This gives you the `roomId` and `serverUrl` values that aren't objects, and that you can read from inside your Effect:

```js {1,4}

function ChatRoom({ getOptions }) {

const [message, setMessage] = useState('');

const { roomId, serverUrl } = getOptions();

useEffect(() => {

const connection = createConnection({

roomId: roomId,

serverUrl: serverUrl

});

connection.connect();

return () => connection.disconnect();

}, [roomId, serverUrl]); // ■ All dependencies declared

// ...

```

This only works for [pure](/learn/keeping-components-pure) functions because they are safe to call during rendering. If your function is an event handler, but you don't want its changes to re-synchronize your Effect, [wrap it into an Effect Event instead.](#do-you-want-to-read-a-value-without-reacting-to-its-changes)

<Recap>

- Dependencies should always match the code.

- When you're not happy with your dependencies, what you need to edit is the code.

- Suppressing the linter leads to very confusing bugs, and you should always avoid it.
```

- To remove a dependency, you need to "prove" to the linter that it's not necessary.
- If some code should run in response to a specific interaction, move that code to an event handler.
- If different parts of your Effect should re-run for different reasons, split it into several Effects.
- If you want to update some state based on the previous state, pass an updater function.
- If you want to read the latest value without "reacting" it, extract an Effect Event from your Effect.
- In JavaScript, objects and functions are considered different if they were created at different times.
- Try to avoid object and function dependencies. Move them outside the component or inside the Effect.

</Recap>

<Challenges>

Fix a resetting interval {/*fix-a-resetting-interval*/}

This Effect sets up an interval that ticks every second. You've noticed something strange happening: it seems like the interval gets destroyed and re-created every time it ticks. Fix the code so that the interval doesn't get constantly re-created.

<Hint>

It seems like this Effect's code depends on `count`. Is there some way to not need this dependency? There should be a way to update the `count` state based on its previous value without adding a dependency on that value.

</Hint>

<Sandpack>

```js
import { useState, useEffect } from 'react';

export default function Timer() {
const [count, setCount] = useState(0);

useEffect(() => {
console.log('■ Creating an interval');
const id = setInterval(() => {
console.log('■ Interval tick');
setCount(count + 1);
}, 1000);
return () => {
console.log('■ Clearing an interval');
clearInterval(id);
};
```

}, [count]);

return <h1>Counter: {count}</h1>

}
```

</Sandpack>

<Solution>

You want to update the `count` state to be `count + 1` from inside the Effect. However, this makes your Effect depend on `count`, which changes with every tick, and that's why your interval gets re-created on every tick.

To solve this, use the [updater function](/reference/react/useState#updating-state-based-on-the-previous-state) and write `setCount(c => c + 1)` instead of `setCount(count + 1)`:

<Sandpack>

```js
import { useState, useEffect } from 'react';

export default function Timer() {

const [count, setCount] = useState(0);

useEffect(() => {

console.log('■ Creating an interval');

const id = setInterval(() => {

console.log('■ Interval tick');

setCount(c => c + 1);

}, 1000);

return () => {

console.log('■ Clearing an interval');

clearInterval(id);

};

}, []);

return <h1>Counter: {count}</h1>

}
```

</Sandpack>
```

Instead of reading `count` inside the Effect, you pass a `c => c + 1` instruction ("increment this number!") to React. React will apply it on the next render. And since you don't need to read the value of `count` inside your Effect anymore, so you can keep your Effect's dependencies empty (`[]`). This prevents your Effect from re-creating the interval on every tick.

</Solution>

Fix a retriggering animation {/*fix-a-retriggering-animation*/}

In this example, when you press "Show", a welcome message fades in. The animation takes a second. When you press "Remove", the welcome message immediately disappears. The logic for the fade-in animation is implemented in the `animation.js` file as plain JavaScript [animation loop.](https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame) You don't need to change that logic. You can treat it as a third-party library. Your Effect creates an instance of `FadeInAnimation` for the DOM node, and then calls `start(duration)` or `stop()` to control the animation. The `duration` is controlled by a slider. Adjust the slider and see how the animation changes.

This code already works, but there is something you want to change. Currently, when you move the slider that controls the `duration` state variable, it retriggers the animation. Change the behavior so that the Effect does not "react" to the `duration` variable. When you press "Show", the Effect should use the current `duration` on the slider. However, moving the slider itself should not by itself retrigger the animation.

<Hint>

Is there a line of code inside the Effect that should not be reactive? How can you move non-reactive code out of the Effect?

</Hint>

<Sandpack>

```json package.json hidden
{
"dependencies": {
"react": "experimental",
"react-dom": "experimental",
"react-scripts": "latest"
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
}
```

}
```

```js
import { useState, useEffect, useRef } from 'react';
import { experimental_useEffectEvent as useEffectEvent } from 'react';
import { FadeInAnimation } from './animation.js';

function Welcome({ duration }) {
const ref = useRef(null);

useEffect(() => {
const animation = new FadeInAnimation(ref.current);
animation.start(duration);
return () => {
animation.stop();
};
}, [duration]);

return (
<h1
ref={ref}
style={{
opacity: 0,
color: 'white',
padding: 50,
textAlign: 'center',
fontSize: 50,
backgroundImage: 'radial-gradient(circle, rgba(63,94,251,1) 0%, rgba(252,70,107,1) 100%)'
}}
>
Welcome
</h1>
);
}

export default function App() {
const [duration, setDuration] = useState(1000);
const [show, setShow] = useState(false);

```
return (
<>
<label>
<input
type="range"
min="100"
max="3000"
value={duration}
onChange={e => setDuration(Number(e.target.value))}
/>
<br />
Fade in duration: {duration} ms
</label>
<button onClick={() => setShow(!show)}>
{show ? 'Remove' : 'Show'}
</button>
<hr />
{show && <Welcome duration={duration} />}
</>
);
}
```

```js animation.js
export class FadeInAnimation {
constructor(node) {
this.node = node;
}
start(duration) {
this.duration = duration;
if (this.duration === 0) {
// Jump to end immediately
this.onProgress(1);
} else {
this.onProgress(0);
// Start animating
```

```
this.startTime = performance.now();
this.frameId = requestAnimationFrame(() => this.onFrame());
}
}
onFrame() {
const timePassed = performance.now() - this.startTime;
const progress = Math.min(timePassed / this.duration, 1);
this.onProgress(progress);
if (progress < 1) {
// We still have more frames to paint
this.frameId = requestAnimationFrame(() => this.onFrame());
}
}
onProgress(progress) {
this.node.style.opacity = progress;
}
stop() {
cancelAnimationFrame(this.frameId);
this.startTime = null;
this.frameId = null;
this.duration = 0;
}
}
```

```css
label, button { display: block; margin-bottom: 20px; }
html, body { min-height: 300px; }
```

</Sandpack>

<Solution>

Your Effect needs to read the latest value of `duration`, but you don't want it to "react" to changes in `duration`. You use `duration` to start the animation, but starting animation isn't reactive. Extract the non-reactive line of code into an Effect Event, and call that function from your Effect.

<Sandpack>

```json package.json hidden
{
"dependencies": {
"react": "experimental",
"react-dom": "experimental",
"react-scripts": "latest"
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
}
}
```

```js
import { useState, useEffect, useRef } from 'react';
import { FadeInAnimation } from './animation.js';
import { experimental_useEffectEvent as useEffectEvent } from 'react';

function Welcome({ duration }) {
const ref = useRef(null);

const onAppear = useEffectEvent(animation => {
animation.start(duration);
});

useEffect(() => {
const animation = new FadeInAnimation(ref.current);
onAppear(animation);
return () => {
animation.stop();
};
}, []);

return (
<h1
ref={ref}
```

```
style={{
opacity: 0,
color: 'white',
padding: 50,
textAlign: 'center',
fontSize: 50,
backgroundImage: 'radial-gradient(circle, rgba(63,94,251,1) 0%, rgba(252,70,107,1) 100%)'
}}
>
Welcome
</h1>
);
}

export default function App() {
const [duration, setDuration] = useState(1000);
const [show, setShow] = useState(false);

return (
<>
<label>
<input
type="range"
min="100"
max="3000"
value={duration}
onChange={e => setDuration(Number(e.target.value))}
/>
<br />
Fade in duration: {duration} ms
</label>
<button onClick={() => setShow(!show)}>
{show ? 'Remove' : 'Show'}
</button>
<hr />
{show && <Welcome duration={duration} />}
</>
```

```
);
}
```

```js animation.js
export class FadeInAnimation {
constructor(node) {
this.node = node;
}
start(duration) {
this.duration = duration;
this.onProgress(0);
this.startTime = performance.now();
this.frameId = requestAnimationFrame(() => this.onFrame());
}
onFrame() {
const timePassed = performance.now() - this.startTime;
const progress = Math.min(timePassed / this.duration, 1);
this.onProgress(progress);
if (progress < 1) {
// We still have more frames to paint
this.frameId = requestAnimationFrame(() => this.onFrame());
}
}
onProgress(progress) {
this.node.style.opacity = progress;
}
stop() {
cancelAnimationFrame(this.frameId);
this.startTime = null;
this.frameId = null;
this.duration = 0;
}
}
```

```css
```

label, button { display: block; margin-bottom: 20px; }

html, body { min-height: 300px; }

```

</Sandpack>

Effect Events like `onAppear` are not reactive, so you can read `duration` inside without retriggering the animation.

</Solution>

#### Fix a reconnecting chat {/*fix-a-reconnecting-chat*/}

In this example, every time you press "Toggle theme", the chat re-connects. Why does this happen? Fix the mistake so that the chat re-connects only when you edit the Server URL or choose a different chat room.

Treat `chat.js` as an external third-party library: you can consult it to check its API, but don't edit it.

<Hint>

There's more than one way to fix this, but ultimately you want to avoid having an object as your dependency.

</Hint>

<Sandpack>

```js App.js

import { useState } from 'react';

import ChatRoom from './ChatRoom.js';

export default function App() {

const [isDark, setIsDark] = useState(false);

const [roomId, setRoomId] = useState('general');

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

const options = {

serverUrl: serverUrl,

roomId: roomId

};

return (

<div className={isDark ? 'dark' : 'light'}>

<button onClick={() => setIsDark(!isDark)}>

Toggle theme

```
</button>
<label>
Server URL:{' '}
<input
value={serverUrl}
onChange={e => setServerUrl(e.target.value)}
/>
</label>
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
</select>
</label>
<hr />
<ChatRoom options={options} />
</div>
);
}
```

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

export default function ChatRoom({ options }) {
useEffect(() => {
const connection = createConnection(options);
connection.connect();
return () => connection.disconnect();
}, [options]);
```

return <h1>Welcome to the {options.roomId} room!</h1>;
}
```

```js chat.js
export function createConnection({ serverUrl, roomId }) {
// A real implementation would actually connect to the server
if (typeof serverUrl !== 'string') {
throw Error('Expected serverUrl to be a string. Received: ' + serverUrl);
}
if (typeof roomId !== 'string') {
throw Error('Expected roomId to be a string. Received: ' + roomId);
}
return {
connect() {
console.log('■ Connecting to "' + roomId + '" room at ' + serverUrl + '...');
},
disconnect() {
console.log('■ Disconnected from "' + roomId + '" room at ' + serverUrl);
}
};
}
```

```css
label, button { display: block; margin-bottom: 5px; }
.dark { background: #222; color: #eee; }
```

</Sandpack>

<Solution>

Your Effect is re-running because it depends on the `options` object. Objects can be re-created unintentionally, you should try to avoid them as dependencies of your Effects whenever possible.

The least invasive fix is to read `roomId` and `serverUrl` right outside the Effect, and then make the Effect depend on those primitive values (which can't change unintentionally). Inside the Effect, create an object and it pass to `createConnection`:

<Sandpack>
```

```js App.js
import { useState } from 'react';
import ChatRoom from './ChatRoom.js';

export default function App() {
const [isDark, setIsDark] = useState(false);
const [roomId, setRoomId] = useState('general');
const [serverUrl, setServerUrl] = useState('https://localhost:1234');

const options = {
serverUrl: serverUrl,
roomId: roomId
};

return (
<div className={isDark ? 'dark' : 'light'}>
<button onClick={() => setIsDark(!isDark)}>
Toggle theme
</button>
<label>
Server URL:{' '}
<input
value={serverUrl}
onChange={e => setServerUrl(e.target.value)}
/>
</label>
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
</select>
</label>
```

<hr />
<ChatRoom options={options} />

);
}
```

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

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

return <h1>Welcome to the {options.roomId} room!</h1>;
}
```

```js chat.js
export function createConnection({ serverUrl, roomId }) {
// A real implementation would actually connect to the server
if (typeof serverUrl !== 'string') {
throw Error('Expected serverUrl to be a string. Received: ' + serverUrl);
}
if (typeof roomId !== 'string') {
throw Error('Expected roomId to be a string. Received: ' + roomId);
}
return {
connect() {
console.log('■ Connecting to "' + roomId + '" room at ' + serverUrl + '...');

},
disconnect() {
console.log('■ Disconnected from "' + roomId + '" room at ' + serverUrl);
}
};
}
```

```css
label, button { display: block; margin-bottom: 5px; }
.dark { background: #222; color: #eee; }
```

</Sandpack>

It would be even better to replace the object `options` prop with the more specific `roomId` and `serverUrl` props:

<Sandpack>

```js App.js
import { useState } from 'react';
import ChatRoom from './ChatRoom.js';

export default function App() {
const [isDark, setIsDark] = useState(false);
const [roomId, setRoomId] = useState('general');
const [serverUrl, setServerUrl] = useState('https://localhost:1234');

return (
<div className={isDark ? 'dark' : 'light'}>
<button onClick={() => setIsDark(!isDark)}>
Toggle theme
</button>
<label>
Server URL:{' '}
<input
value={serverUrl}
onChange={e => setServerUrl(e.target.value)}
/>

```
</label>
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
</select>
</label>
<hr />
<ChatRoom
roomId={roomId}
serverUrl={serverUrl}
/>
</div>
);
}
```

```js ChatRoom.js active
import { useState, useEffect } from 'react';
import { createConnection } from './chat.js';

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

return <h1>Welcome to the {roomId} room!</h1>;
```

}
```

```js chat.js
export function createConnection({ serverUrl, roomId }) {
// A real implementation would actually connect to the server
if (typeof serverUrl !== 'string') {
throw Error('Expected serverUrl to be a string. Received: ' + serverUrl);
}
if (typeof roomId !== 'string') {
throw Error('Expected roomId to be a string. Received: ' + roomId);
}
return {
connect() {
console.log('■ Connecting to "' + roomId + '" room at ' + serverUrl + '...');
},
disconnect() {
console.log('■ Disconnected from "' + roomId + '" room at ' + serverUrl);
}
};
}
```

```css
label, button { display: block; margin-bottom: 5px; }
.dark { background: #222; color: #eee; }
```

</Sandpack>

Sticking to primitive props where possible makes it easier to optimize your components later.

</Solution>

Fix a reconnecting chat, again {/*fix-a-reconnecting-chat-again*/}

This example connects to the chat either with or without encryption. Toggle the checkbox and notice the different messages in the console when the encryption is on and off. Try changing the room. Then, try toggling the theme. When you're connected to a chat room, you will receive new messages every few seconds. Verify that their color matches the theme you've picked.
```

In this example, the chat re-connects every time you try to change the theme. Fix this. After the fix, changing the theme should not re-connect the chat, but toggling encryption settings or changing the room should re-connect.

Don't change any code in `chat.js`. Other than that, you can change any code as long as it results in the same behavior. For example, you may find it helpful to change which props are being passed down.

<Hint>

You're passing down two functions: `onMessage` and `createConnection`. Both of them are created from scratch every time `App` re-renders. They are considered to be new values every time, which is why they re-trigger your Effect.

One of these functions is an event handler. Do you know some way to call an event handler an Effect without "reacting" to the new values of the event handler function? That would come in handy!

Another of these functions only exists to pass some state to an imported API method. Is this function really necessary? What is the essential information that's being passed down? You might need to move some imports from `App.js` to `ChatRoom.js`.

</Hint>

<Sandpack>

```json package.json hidden
{
"dependencies": {
"react": "experimental",
"react-dom": "experimental",
"react-scripts": "latest",
"toastify-js": "1.12.0"
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
}
}
```

```js App.js
import { useState } from 'react';
import ChatRoom from './ChatRoom.js';
```

```
import {
createEncryptedConnection,
createUnencryptedConnection,
} from './chat.js';
import { showNotification } from './notifications.js';

export default function App() {
const [isDark, setIsDark] = useState(false);
const [roomId, setRoomId] = useState('general');
const [isEncrypted, setIsEncrypted] = useState(false);

return (
<>
<label>
<input
type="checkbox"
checked={isDark}
onChange={e => setIsDark(e.target.checked)}
/>
Use dark theme
</label>
<label>
<input
type="checkbox"
checked={isEncrypted}
onChange={e => setIsEncrypted(e.target.checked)}
/>
Enable encryption
</label>
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
<option value="general">general</option>
<option value="travel">travel</option>
```

```
<option value="music">music</option>
</select>
</label>
<hr />
<ChatRoom
roomId={roomId}
onMessage={msg => {
showNotification('New message: ' + msg, isDark ? 'dark' : 'light');
}}
createConnection={() => {
const options = {
serverUrl: 'https://localhost:1234',
roomId: roomId
};
if (isEncrypted) {
return createEncryptedConnection(options);
} else {
return createUnencryptedConnection(options);
}
}}
/>
</>
);
}
```

```js ChatRoom.js active
import { useState, useEffect } from 'react';
import { experimental_useEffectEvent as useEffectEvent } from 'react';

export default function ChatRoom({ roomId, createConnection, onMessage }) {
useEffect(() => {
const connection = createConnection();
connection.on('message', (msg) => onMessage(msg));
connection.connect();
return () => connection.disconnect();
}, [createConnection, onMessage]);
```

````
return <h1>Welcome to the {roomId} room!</h1>;
}
```

```js chat.js
export function createEncryptedConnection({ serverUrl, roomId }) {
// A real implementation would actually connect to the server
if (typeof serverUrl !== 'string') {
throw Error('Expected serverUrl to be a string. Received: ' + serverUrl);
}
if (typeof roomId !== 'string') {
throw Error('Expected roomId to be a string. Received: ' + roomId);
}
let intervalId;
let messageCallback;
return {
connect() {
console.log('■ ■ Connecting to "' + roomId + '" room... (encrypted)');
clearInterval(intervalId);
intervalId = setInterval(() => {
if (messageCallback) {
if (Math.random() > 0.5) {
messageCallback('hey')
} else {
messageCallback('lol');
}
}
}, 3000);
},
disconnect() {
clearInterval(intervalId);
messageCallback = null;
console.log('■ ■ Disconnected from "' + roomId + '" room (encrypted)');
},
on(event, callback) {
if (messageCallback) {
````

```
throw Error('Cannot add the handler twice.');
}
if (event !== 'message') {
throw Error('Only "message" event is supported.');
}
messageCallback = callback;
},
};
}

export function createUnencryptedConnection({ serverUrl, roomId }) {
// A real implementation would actually connect to the server
if (typeof serverUrl !== 'string') {
throw Error('Expected serverUrl to be a string. Received: ' + serverUrl);
}
if (typeof roomId !== 'string') {
throw Error('Expected roomId to be a string. Received: ' + roomId);
}
let intervalId;
let messageCallback;
return {
connect() {
console.log('■ Connecting to "' + roomId + '" room (unencrypted)...');
clearInterval(intervalId);
intervalId = setInterval(() => {
if (messageCallback) {
if (Math.random() > 0.5) {
messageCallback('hey')
} else {
messageCallback('lol');
}
}
}, 3000);
},
disconnect() {
clearInterval(intervalId);
```

```
messageCallback = null;
console.log('■ Disconnected from "' + roomId + '" room (unencrypted)');
},
on(event, callback) {
if (messageCallback) {
throw Error('Cannot add the handler twice.');
}
if (event !== 'message') {
throw Error('Only "message" event is supported.');
}
messageCallback = callback;
},
};
}
```

```js notifications.js
import Toastify from 'toastify-js';
import 'toastify-js/src/toastify.css';

export function showNotification(message, theme) {
Toastify({
text: message,
duration: 2000,
gravity: 'top',
position: 'right',
style: {
background: theme === 'dark' ? 'black' : 'white',
color: theme === 'dark' ? 'white' : 'black',
},
}).showToast();
}
```

```css
label, button { display: block; margin-bottom: 5px; }
```

</Sandpack>

<Solution>

There's more than one correct way to solve this, but here is one possible solution.

In the original example, toggling the theme caused different `onMessage` and `createConnection` functions to be created and passed down. Since the Effect depended on these functions, the chat would re-connect every time you toggle the theme.

To fix the problem with `onMessage`, you needed to wrap it into an Effect Event:

```js {1,2,6}
export default function ChatRoom({ roomId, createConnection, onMessage }) {
const onReceiveMessage = useEffectEvent(onMessage);

useEffect(() => {
const connection = createConnection();
connection.on('message', (msg) => onReceiveMessage(msg));
// ...
```

Unlike the `onMessage` prop, the `onReceiveMessage` Effect Event is not reactive. This is why it doesn't need to be a dependency of your Effect. As a result, changes to `onMessage` won't cause the chat to re-connect.

You can't do the same with `createConnection` because it *should* be reactive. You *want* the Effect to re-trigger if the user switches between an encrypted and an unencryption connection, or if the user switches the current room. However, because `createConnection` is a function, you can't check whether the information it reads has *actually* changed or not. To solve this, instead of passing `createConnection` down from the `App` component, pass the raw `roomId` and `isEncrypted` values:

```js {2-3}
<ChatRoom
roomId={roomId}
isEncrypted={isEncrypted}
onMessage={msg => {
showNotification('New message: ' + msg, isDark ? 'dark' : 'light');
}}
/>
```

Now you can move the `createConnection` function *inside* the Effect instead of passing it down from the `App`:

```js {1-4,6,10-20}
```

```
import {
createEncryptedConnection,
createUnencryptedConnection,
} from './chat.js';

export default function ChatRoom({ roomId, isEncrypted, onMessage }) {
const onReceiveMessage = useEffectEvent(onMessage);

useEffect(() => {
function createConnection() {
const options = {
serverUrl: 'https://localhost:1234',
roomId: roomId
};
if (isEncrypted) {
return createEncryptedConnection(options);
} else {
return createUnencryptedConnection(options);
}
}
// ...
```

After these two changes, your Effect no longer depends on any function values:

```js {1,8,10,21}
export default function ChatRoom({ roomId, isEncrypted, onMessage }) { // Reactive values
const onReceiveMessage = useEffectEvent(onMessage); // Not reactive

useEffect(() => {
function createConnection() {
const options = {
serverUrl: 'https://localhost:1234',
roomId: roomId // Reading a reactive value
};
if (isEncrypted) { // Reading a reactive value
return createEncryptedConnection(options);
} else {
return createUnencryptedConnection(options);
```

```
}
}

const connection = createConnection();
connection.on('message', (msg) => onReceiveMessage(msg));
connection.connect();
return () => connection.disconnect();
}, [roomId, isEncrypted]); // ■ All dependencies declared
```

As a result, the chat re-connects only when something meaningful (`roomId` or `isEncrypted`) changes:

<Sandpack>

```json package.json hidden
{
"dependencies": {
"react": "experimental",
"react-dom": "experimental",
"react-scripts": "latest",
"toastify-js": "1.12.0"
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
}
}
```

```js App.js
import { useState } from 'react';
import ChatRoom from './ChatRoom.js';

import { showNotification } from './notifications.js';

export default function App() {
const [isDark, setIsDark] = useState(false);
const [roomId, setRoomId] = useState('general');
```

```
const [isEncrypted, setIsEncrypted] = useState(false);

return (
<>
<label>
<input
type="checkbox"
checked={isDark}
onChange={e => setIsDark(e.target.checked)}
/>
Use dark theme
</label>
<label>
<input
type="checkbox"
checked={isEncrypted}
onChange={e => setIsEncrypted(e.target.checked)}
/>
Enable encryption
</label>
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
</select>
</label>
<hr />
<ChatRoom
roomId={roomId}
isEncrypted={isEncrypted}
onMessage={msg => {
```

```
showNotification('New message: ' + msg, isDark ? 'dark' : 'light');
}}
/>
</>
);
}
```

```js ChatRoom.js active
import { useState, useEffect } from 'react';
import { experimental_useEffectEvent as useEffectEvent } from 'react';
import {
createEncryptedConnection,
createUnencryptedConnection,
} from './chat.js';

export default function ChatRoom({ roomId, isEncrypted, onMessage }) {
const onReceiveMessage = useEffectEvent(onMessage);

useEffect(() => {
function createConnection() {
const options = {
serverUrl: 'https://localhost:1234',
roomId: roomId
};
if (isEncrypted) {
return createEncryptedConnection(options);
} else {
return createUnencryptedConnection(options);
}
}

const connection = createConnection();
connection.on('message', (msg) => onReceiveMessage(msg));
connection.connect();
return () => connection.disconnect();
}, [roomId, isEncrypted]);

return <h1>Welcome to the {roomId} room!</h1>;
```

}
```

```js chat.js
export function createEncryptedConnection({ serverUrl, roomId }) {
// A real implementation would actually connect to the server
if (typeof serverUrl !== 'string') {
throw Error('Expected serverUrl to be a string. Received: ' + serverUrl);
}
if (typeof roomId !== 'string') {
throw Error('Expected roomId to be a string. Received: ' + roomId);
}
let intervalId;
let messageCallback;
return {
connect() {
console.log('■ ■ Connecting to "' + roomId + '" room... (encrypted)');
clearInterval(intervalId);
intervalId = setInterval(() => {
if (messageCallback) {
if (Math.random() > 0.5) {
messageCallback('hey')
} else {
messageCallback('lol');
}
}
}, 3000);
},
disconnect() {
clearInterval(intervalId);
messageCallback = null;
console.log('■ ■ Disconnected from "' + roomId + '" room (encrypted)');
},
on(event, callback) {
if (messageCallback) {
throw Error('Cannot add the handler twice.');

```
}
if (event !== 'message') {
throw Error('Only "message" event is supported.');
}
messageCallback = callback;
},
};
}

export function createUnencryptedConnection({ serverUrl, roomId }) {
// A real implementation would actually connect to the server
if (typeof serverUrl !== 'string') {
throw Error('Expected serverUrl to be a string. Received: ' + serverUrl);
}
if (typeof roomId !== 'string') {
throw Error('Expected roomId to be a string. Received: ' + roomId);
}
let intervalId;
let messageCallback;
return {
connect() {
console.log('■ Connecting to "' + roomId + '" room (unencrypted)...');
clearInterval(intervalId);
intervalId = setInterval(() => {
if (messageCallback) {
if (Math.random() > 0.5) {
messageCallback('hey')
} else {
messageCallback('lol');
}
}
}, 3000);
},
disconnect() {
clearInterval(intervalId);
messageCallback = null;
```

```
console.log('■ Disconnected from "' + roomId + '" room (unencrypted)');
},
on(event, callback) {
if (messageCallback) {
throw Error('Cannot add the handler twice.');
}
if (event !== 'message') {
throw Error('Only "message" event is supported.');
}
messageCallback = callback;
},
};
}
```

```js notifications.js
import Toastify from 'toastify-js';
import 'toastify-js/src/toastify.css';

export function showNotification(message, theme) {
Toastify({
text: message,
duration: 2000,
gravity: 'top',
position: 'right',
style: {
background: theme === 'dark' ? 'black' : 'white',
color: theme === 'dark' ? 'white' : 'black',
},
}).showToast();
}
```

```css
label, button { display: block; margin-bottom: 5px; }
```

</Sandpack>

</Solution>

</Challenges>

---

title: 'Separating Events from Effects'

<Intro>

Event handlers only re-run when you perform the same interaction again. Unlike event handlers, Effects re-synchronize if some value they read, like a prop or a state variable, is different from what it was during the last render. Sometimes, you also want a mix of both behaviors: an Effect that re-runs in response to some values but not others. This page will teach you how to do that.

</Intro>

<YouWillLearn>

- How to choose between an event handler and an Effect
- Why Effects are reactive, and event handlers are not
- What to do when you want a part of your Effect's code to not be reactive
- What Effect Events are, and how to extract them from your Effects
- How to read the latest props and state from Effects using Effect Events

</YouWillLearn>

Choosing between event handlers and Effects {/*choosing-between-event-handlers-and-effects*/}

First, let's recap the difference between event handlers and Effects.

Imagine you're implementing a chat room component. Your requirements look like this:

1. Your component should automatically connect to the selected chat room.
1. When you click the "Send" button, it should send a message to the chat.

Let's say you've already implemented the code for them, but you're not sure where to put it. Should you use event handlers or Effects? Every time you need to answer this question, consider [*why* the code needs to
run.](/learn/synchronizing-with-effects#what-are-effects-and-how-are-they-different-from-events)

Event handlers run in response to specific interactions
{/*event-handlers-run-in-response-to-specific-interactions*/}

From the user's perspective, sending a message should happen *because* the particular "Send" button was clicked. The user will get rather upset if you send their message at any other time or for any other reason. This is why sending a message should be an event handler. Event handlers let you handle specific interactions:

```js {4-6}
```

```
function ChatRoom({ roomId }) {
const [message, setMessage] = useState('');
// ...
function handleSendClick() {
sendMessage(message);
}
// ...
return (
<>
<input value={message} onChange={e => setMessage(e.target.value)} />
<button onClick={handleSendClick}>Send</button>;
</>
);
}
```

With an event handler, you can be sure that `sendMessage(message)` will *only* run if the user presses the button.

Effects run whenever synchronization is needed
{/*effects-run-whenever-synchronization-is-needed*/}

Recall that you also need to keep the component connected to the chat room. Where does that code go?

The *reason* to run this code is not some particular interaction. It doesn't matter why or how the user navigated to the chat room screen. Now that they're looking at it and could interact with it, the component needs to stay connected to the selected chat server. Even if the chat room component was the initial screen of your app, and the user has not performed any interactions at all, you would *still* need to connect. This is why it's an Effect:

```js {3-9}
function ChatRoom({ roomId }) {
// ...
useEffect(() => {
const connection = createConnection(serverUrl, roomId);
connection.connect();
return () => {
connection.disconnect();
};
}, [roomId]);
```

```
// ...
}
```

With this code, you can be sure that there is always an active connection to the currently selected chat server, *regardless* of the specific interactions performed by the user. Whether the user has only opened your app, selected a different room, or navigated to another screen and back, your Effect ensures that the component will *remain synchronized* with the currently selected room, and will [re-connect whenever it's necessary.](/learn/lifecycle-of-reactive-effects#why-synchronization-may-need-to-happen-more-than-once)

<Sandpack>

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

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

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

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

function handleSendClick() {
sendMessage(message);
}

return (
<>
<h1>Welcome to the {roomId} room!</h1>
<input value={message} onChange={e => setMessage(e.target.value)} />
<button onClick={handleSendClick}>Send</button>
</>
);
}

export default function App() {
const [roomId, setRoomId] = useState('general');
```

```
const [show, setShow] = useState(false);
return (
<>
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
</select>
</label>
<button onClick={() => setShow(!show)}>
{show ? 'Close chat' : 'Open chat'}
</button>
{show && <hr />}
{show && <ChatRoom roomId={roomId} />}
</>
);
}
```

```js chat.js
export function sendMessage(message) {
console.log('■ You sent: ' + message);
}

export function createConnection(serverUrl, roomId) {
// A real implementation would actually connect to the server
return {
connect() {
console.log('■ Connecting to '' + roomId + '' room at ' + serverUrl + '...');
},
disconnect() {
console.log('■ Disconnected from '' + roomId + '' room at ' + serverUrl);
```

```
}
};
}
```

```css
input, select { margin-right: 20px; }
```

</Sandpack>

Reactive values and reactive logic {/*reactive-values-and-reactive-logic*/}

Intuitively, you could say that event handlers are always triggered "manually", for example by clicking a button. Effects, on the other hand, are "automatic": they run and re-run as often as it's needed to stay synchronized.

There is a more precise way to think about this.

Props, state, and variables declared inside your component's body are called <CodeStep step={2}>reactive values</CodeStep>. In this example, `serverUrl` is not a reactive value, but `roomId` and `message` are. They participate in the rendering data flow:

```js [[2, 3, "roomId"], [2, 4, "message"]]
const serverUrl = 'https://localhost:1234';

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

// ...
}
```

Reactive values like these can change due to a re-render. For example, the user may edit the `message` or choose a different `roomId` in a dropdown. Event handlers and Effects respond to changes differently:

- **Logic inside event handlers is *not reactive.*** It will not run again unless the user performs the same interaction (e.g. a click) again. Event handlers can read reactive values without "reacting" to their changes.
- **Logic inside Effects is *reactive.*** If your Effect reads a reactive value, [you have to specify it as a dependency.](/learn/lifecycle-of-reactive-effects#effects-react-to-reactive-values) Then, if a re-render causes that value to change, React will re-run your Effect's logic with the new value.

Let's revisit the previous example to illustrate this difference.

Logic inside event handlers is not reactive {/*logic-inside-event-handlers-is-not-reactive*/}

Take a look at this line of code. Should this logic be reactive or not?

```js [[2, 2, "message"]]
// ...
sendMessage(message);
// ...
```

From the user's perspective, **a change to the `message` does _not_ mean that they want to send a message.** It only means that the user is typing. In other words, the logic that sends a message should not be reactive. It should not run again only because the <CodeStep step={2}>reactive value</CodeStep> has changed. That's why it belongs in the event handler:

```js {2}
function handleSendClick() {
sendMessage(message);
}
```

Event handlers aren't reactive, so `sendMessage(message)` will only run when the user clicks the Send button.

Logic inside Effects is reactive {/*logic-inside-effects-is-reactive*/}

Now let's return to these lines:

```js [[2, 2, "roomId"]]
// ...
const connection = createConnection(serverUrl, roomId);
connection.connect();
// ...
```

From the user's perspective, **a change to the `roomId` *does* mean that they want to connect to a different room.** In other words, the logic for connecting to the room should be reactive. You *want* these lines of code to "keep up" with the <CodeStep step={2}>reactive value</CodeStep>, and to run again if that value is different. That's why it belongs in an Effect:

```js {2-3}
useEffect(() => {
const connection = createConnection(serverUrl, roomId);
connection.connect();
return () => {
connection.disconnect()
```

};

}, [roomId]);
```

Effects are reactive, so `createConnection(serverUrl, roomId)` and `connection.connect()` will run for every distinct value of `roomId`. Your Effect keeps the chat connection synchronized to the currently selected room.

## Extracting non-reactive logic out of Effects {/*extracting-non-reactive-logic-out-of-effects*/}

Things get more tricky when you want to mix reactive logic with non-reactive logic.

For example, imagine that you want to show a notification when the user connects to the chat. You read the current theme (dark or light) from the props so that you can show the notification in the correct color:

```js {1,4-6}
function ChatRoom({ roomId, theme }) {
useEffect(() => {
const connection = createConnection(serverUrl, roomId);
connection.on('connected', () => {
showNotification('Connected!', theme);
});
connection.connect();
// ...
```

However, `theme` is a reactive value (it can change as a result of re-rendering), and [every reactive value read by an Effect must be declared as its dependency.](/learn/lifecycle-of-reactive-effects#react-verifies-that-you-specified-every-reactive-value-as-a-dependency) Now you have to specify `theme` as a dependency of your Effect:

```js {5,11}
function ChatRoom({ roomId, theme }) {
useEffect(() => {
const connection = createConnection(serverUrl, roomId);
connection.on('connected', () => {
showNotification('Connected!', theme);
});
connection.connect();
return () => {
connection.disconnect()
```
```

};
}, [roomId, theme]); // ■ All dependencies declared
// ...
```

Play with this example and see if you can spot the problem with this user experience:

<Sandpack>

```json package.json hidden
{
"dependencies": {
"react": "latest",
"react-dom": "latest",
"react-scripts": "latest",
"toastify-js": "1.12.0"
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
}
}
```

```js
import { useState, useEffect } from 'react';
import { createConnection, sendMessage } from './chat.js';
import { showNotification } from './notifications.js';

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

function ChatRoom({ roomId, theme }) {
useEffect(() => {
const connection = createConnection(serverUrl, roomId);
connection.on('connected', () => {
showNotification('Connected!', theme);
});

```
connection.connect();
return () => connection.disconnect();
}, [roomId, theme]);

return <h1>Welcome to the {roomId} room!</h1>
}

export default function App() {
const [roomId, setRoomId] = useState('general');
const [isDark, setIsDark] = useState(false);
return (
<>
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
</select>
</label>
<label>
<input
type="checkbox"
checked={isDark}
onChange={e => setIsDark(e.target.checked)}
/>
Use dark theme
</label>
<hr />
<ChatRoom
roomId={roomId}
theme={isDark ? 'dark' : 'light'}
/>
</>
```

);
}
```

```js chat.js
export function createConnection(serverUrl, roomId) {
// A real implementation would actually connect to the server
let connectedCallback;
let timeout;
return {
connect() {
timeout = setTimeout(() => {
if (connectedCallback) {
connectedCallback();
}
}, 100);
},
on(event, callback) {
if (connectedCallback) {
throw Error('Cannot add the handler twice.');
}
if (event !== 'connected') {
throw Error('Only "connected" event is supported.');
}
connectedCallback = callback;
},
disconnect() {
clearTimeout(timeout);
}
};
}
```

```js notifications.js
import Toastify from 'toastify-js';
import 'toastify-js/src/toastify.css';

export function showNotification(message, theme) {

Toastify({

text: message,

duration: 2000,

gravity: 'top',

position: 'right',

style: {

background: theme === 'dark' ? 'black' : 'white',

color: theme === 'dark' ? 'white' : 'black',

},

}).showToast();

}

```

```css

label { display: block; margin-top: 10px; }

```

</Sandpack>

When the `roomId` changes, the chat re-connects as you would expect. But since `theme` is also a dependency, the chat *also* re-connects every time you switch between the dark and the light theme. That's not great!

In other words, you *don't* want this line to be reactive, even though it is inside an Effect (which is reactive):

```js

// ...

showNotification('Connected!', theme);

// ...

```

You need a way to separate this non-reactive logic from the reactive Effect around it.

Declaring an Effect Event {/*declaring-an-effect-event*/}

<Wip>

This section describes an **experimental API that has not yet been released** in a stable version of React.

</Wip>
```

Use a special Hook called [`useEffectEvent`](/reference/react/experimental_useEffectEvent) to extract this non-reactive logic out of your Effect:

```js {1,4-6}
import { useEffect, useEffectEvent } from 'react';

function ChatRoom({ roomId, theme }) {
const onConnected = useEffectEvent(() => {
showNotification('Connected!', theme);
});
// ...
```

Here, `onConnected` is called an *Effect Event.* It's a part of your Effect logic, but it behaves a lot more like an event handler. The logic inside it is not reactive, and it always "sees" the latest values of your props and state.

Now you can call the `onConnected` Effect Event from inside your Effect:

```js {2-4,9,13}
function ChatRoom({ roomId, theme }) {
const onConnected = useEffectEvent(() => {
showNotification('Connected!', theme);
});

useEffect(() => {
const connection = createConnection(serverUrl, roomId);
connection.on('connected', () => {
onConnected();
});
connection.connect();
return () => connection.disconnect();
}, [roomId]); // ■ All dependencies declared
// ...
```

This solves the problem. Note that you had to *remove* `onConnected` from the list of your Effect's dependencies. **Effect Events are not reactive and must be omitted from dependencies.**

Verify that the new behavior works as you would expect:

<Sandpack>

```json package.json hidden
{
"dependencies": {
"react": "experimental",
"react-dom": "experimental",
"react-scripts": "latest",
"toastify-js": "1.12.0"
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
}
}
```

```js
import { useState, useEffect } from 'react';
import { experimental_useEffectEvent as useEffectEvent } from 'react';
import { createConnection, sendMessage } from './chat.js';
import { showNotification } from './notifications.js';

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

function ChatRoom({ roomId, theme }) {
const onConnected = useEffectEvent(() => {
showNotification('Connected!', theme);
});

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

```
return <h1>Welcome to the {roomId} room!</h1>
}

export default function App() {
const [roomId, setRoomId] = useState('general');
const [isDark, setIsDark] = useState(false);
return (
<>
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
</select>
</label>
<label>
<input
type="checkbox"
checked={isDark}
onChange={e => setIsDark(e.target.checked)}
/>
Use dark theme
</label>
<hr />
<ChatRoom
roomId={roomId}
theme={isDark ? 'dark' : 'light'}
/>
</>
);
}
```

```js chat.js
export function createConnection(serverUrl, roomId) {
// A real implementation would actually connect to the server
let connectedCallback;
let timeout;
return {
connect() {
timeout = setTimeout(() => {
if (connectedCallback) {
connectedCallback();
}
}, 100);
},
on(event, callback) {
if (connectedCallback) {
throw Error('Cannot add the handler twice.');
}
if (event !== 'connected') {
throw Error('Only "connected" event is supported.');
}
connectedCallback = callback;
},
disconnect() {
clearTimeout(timeout);
}
};
}
```

```js notifications.js hidden
import Toastify from 'toastify-js';
import 'toastify-js/src/toastify.css';

export function showNotification(message, theme) {
Toastify({
text: message,
duration: 2000,
```

gravity: 'top',

position: 'right',

style: {

background: theme === 'dark' ? 'black' : 'white',

color: theme === 'dark' ? 'white' : 'black',

},

}).showToast();

}

```

```css

label { display: block; margin-top: 10px; }

```

</Sandpack>

You can think of Effect Events as being very similar to event handlers. The main difference is that event handlers run in response to a user interactions, whereas Effect Events are triggered by you from Effects. Effect Events let you "break the chain" between the reactivity of Effects and code that should not be reactive.

Reading latest props and state with Effect Events
{/*reading-latest-props-and-state-with-effect-events*/}

<Wip>

This section describes an **experimental API that has not yet been released** in a stable version of React.

</Wip>

Effect Events let you fix many patterns where you might be tempted to suppress the dependency linter.

For example, say you have an Effect to log the page visits:

```js

function Page() {

useEffect(() => {

logVisit();

}, []);

// ...

}

```
```

Later, you add multiple routes to your site. Now your `Page` component receives a `url` prop with the current path. You want to pass the `url` as a part of your `logVisit` call, but the dependency linter complains:

```js {1,3}
function Page({ url }) {
useEffect(() => {
logVisit(url);
}, []); // ■ React Hook useEffect has a missing dependency: 'url'
// ...
}
```

Think about what you want the code to do. You *want* to log a separate visit for different URLs since each URL represents a different page. In other words, this `logVisit` call *should* be reactive with respect to the `url`. This is why, in this case, it makes sense to follow the dependency linter, and add `url` as a dependency:

```js {4}
function Page({ url }) {
useEffect(() => {
logVisit(url);
}, [url]); // ■ All dependencies declared
// ...
}
```

Now let's say you want to include the number of items in the shopping cart together with every page visit:

```js {2-3,6}
function Page({ url }) {
const { items } = useContext(ShoppingCartContext);
const numberOfItems = items.length;

useEffect(() => {
logVisit(url, numberOfItems);
}, [url]); // ■ React Hook useEffect has a missing dependency: 'numberOfItems'
// ...
}
```

You used `numberOfItems` inside the Effect, so the linter asks you to add it as a dependency. However, you *don't* want the `logVisit` call to be reactive with respect to `numberOfItems`. If the user puts something into the shopping cart, and the `numberOfItems` changes, this *does not mean* that the user visited the page again. In other words, *visiting the page* is, in some sense, an "event". It happens at a precise moment in time.

Split the code in two parts:

```js {5-7,10}
function Page({ url }) {
const { items } = useContext(ShoppingCartContext);
const numberOfItems = items.length;

const onVisit = useEffectEvent(visitedUrl => {
logVisit(visitedUrl, numberOfItems);
});

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

Here, `onVisit` is an Effect Event. The code inside it isn't reactive. This is why you can use `numberOfItems` (or any other reactive value!) without worrying that it will cause the surrounding code to re-execute on changes.

On the other hand, the Effect itself remains reactive. Code inside the Effect uses the `url` prop, so the Effect will re-run after every re-render with a different `url`. This, in turn, will call the `onVisit` Effect Event.

As a result, you will call `logVisit` for every change to the `url`, and always read the latest `numberOfItems`. However, if `numberOfItems` changes on its own, this will not cause any of the code to re-run.

<Note>

You might be wondering if you could call `onVisit()` with no arguments, and read the `url` inside it:

```js {2,6}
const onVisit = useEffectEvent(() => {
logVisit(url, numberOfItems);
});

useEffect(() => {
```

onVisit();

}, [url]);
```

This would work, but it's better to pass this `url` to the Effect Event explicitly. **By passing `url` as an argument to your Effect Event, you are saying that visiting a page with a different `url` constitutes a separate "event" from the user's perspective.** The `visitedUrl` is a *part* of the "event" that happened:

```js {1-2,6}
const onVisit = useEffectEvent(visitedUrl => {

logVisit(visitedUrl, numberOfItems);

});

useEffect(() => {

onVisit(url);

}, [url]);
```

Since your Effect Event explicitly "asks" for the `visitedUrl`, now you can't accidentally remove `url` from the Effect's dependencies. If you remove the `url` dependency (causing distinct page visits to be counted as one), the linter will warn you about it. You want `onVisit` to be reactive with regards to the `url`, so instead of reading the `url` inside (where it wouldn't be reactive), you pass it *from* your Effect.

This becomes especially important if there is some asynchronous logic inside the Effect:

```js {6,8}
const onVisit = useEffectEvent(visitedUrl => {

logVisit(visitedUrl, numberOfItems);

});

useEffect(() => {

setTimeout(() => {

onVisit(url);

}, 5000); // Delay logging visits

}, [url]);
```

Here, `url` inside `onVisit` corresponds to the *latest* `url` (which could have already changed), but `visitedUrl` corresponds to the `url` that originally caused this Effect (and this `onVisit` call) to run.

</Note>

<DeepDive>
```

Is it okay to suppress the dependency linter instead?
{/*is-it-okay-to-suppress-the-dependency-linter-instead*/}

In the existing codebases, you may sometimes see the lint rule suppressed like this:

```js {7-9}
function Page({ url }) {
const { items } = useContext(ShoppingCartContext);
const numberOfItems = items.length;

useEffect(() => {
logVisit(url, numberOfItems);
// ■ Avoid suppressing the linter like this:
// eslint-disable-next-line react-hooks/exhaustive-deps
}, [url]);
// ...
}
```

After `useEffectEvent` becomes a stable part of React, we recommend **never suppressing the linter**.

The first downside of suppressing the rule is that React will no longer warn you when your Effect needs to "react" to a new reactive dependency you've introduced to your code. In the earlier example, you added `url` to the dependencies *because* React reminded you to do it. You will no longer get such reminders for any future edits to that Effect if you disable the linter. This leads to bugs.

Here is an example of a confusing bug caused by suppressing the linter. In this example, the `handleMove` function is supposed to read the current `canMove` state variable value in order to decide whether the dot should follow the cursor. However, `canMove` is always `true` inside `handleMove`.

Can you see why?

<Sandpack>

```js
import { useState, useEffect } from 'react';

export default function App() {
const [position, setPosition] = useState({ x: 0, y: 0 });
const [canMove, setCanMove] = useState(true);

function handleMove(e) {
if (canMove) {
setPosition({ x: e.clientX, y: e.clientY });
```

```
}
}

useEffect(() => {
window.addEventListener('pointermove', handleMove);
return () => window.removeEventListener('pointermove', handleMove);
// eslint-disable-next-line react-hooks/exhaustive-deps
}, []);

return (
<>
<label>
<input type="checkbox"
checked={canMove}
onChange={e => setCanMove(e.target.checked)}
/>
The dot is allowed to move
</label>
<hr />
<div style={{
position: 'absolute',
backgroundColor: 'pink',
borderRadius: '50%',
opacity: 0.6,
transform: `translate(${position.x}px, ${position.y}px)`,
pointerEvents: 'none',
left: -20,
top: -20,
width: 40,
height: 40,
}} />
</>
);
}
```

```css
```

body {

height: 200px;

}

```

</Sandpack>

The problem with this code is in suppressing the dependency linter. If you remove the suppression, you'll see that this Effect should depend on the `handleMove` function. This makes sense: `handleMove` is declared inside the component body, which makes it a reactive value. Every reactive value must be specified as a dependency, or it can potentially get stale over time!

The author of the original code has "lied" to React by saying that the Effect does not depend (`[]`) on any reactive values. This is why React did not re-synchronize the Effect after `canMove` has changed (and `handleMove` with it). Because React did not re-synchronize the Effect, the `handleMove` attached as a listener is the `handleMove` function created during the initial render. During the initial render, `canMove` was `true`, which is why `handleMove` from the initial render will forever see that value.

**If you never suppress the linter, you will never see problems with stale values.**

With `useEffectEvent`, there is no need to "lie" to the linter, and the code works as you would expect:

<Sandpack>

```json package.json hidden

{

"dependencies": {

"react": "experimental",

"react-dom": "experimental",

"react-scripts": "latest"

},

"scripts": {

"start": "react-scripts start",

"build": "react-scripts build",

"test": "react-scripts test --env=jsdom",

"eject": "react-scripts eject"

}

}

```

```js

import { useState, useEffect } from 'react';

```
import { experimental_useEffectEvent as useEffectEvent } from 'react';

export default function App() {
const [position, setPosition] = useState({ x: 0, y: 0 });
const [canMove, setCanMove] = useState(true);

const onMove = useEffectEvent(e => {
if (canMove) {
setPosition({ x: e.clientX, y: e.clientY });
}
});

useEffect(() => {
window.addEventListener('pointermove', onMove);
return () => window.removeEventListener('pointermove', onMove);
}, []);

return (
<>
<label>
<input type="checkbox"
checked={canMove}
onChange={e => setCanMove(e.target.checked)}
/>
The dot is allowed to move
</label>
<hr />
<div style={{
position: 'absolute',
backgroundColor: 'pink',
borderRadius: '50%',
opacity: 0.6,
transform: `translate(${position.x}px, ${position.y}px)`,
pointerEvents: 'none',
left: -20,
top: -20,
width: 40,
height: 40,
```

}} />

</>

);

}

```

```css

body {

height: 200px;

}

```

</Sandpack>

This doesn't mean that `useEffectEvent` is *always* the correct solution. You should only apply it to the lines of code that you don't want to be reactive. In the above sandbox, you didn't want the Effect's code to be reactive with regards to `canMove`. That's why it made sense to extract an Effect Event.

Read [Removing Effect Dependencies](/learn/removing-effect-dependencies) for other correct alternatives to suppressing the linter.

</DeepDive>

Limitations of Effect Events {/*limitations-of-effect-events*/}

<Wip>

This section describes an **experimental API that has not yet been released** in a stable version of React.

</Wip>

Effect Events are very limited in how you can use them:

* **Only call them from inside Effects.**

* **Never pass them to other components or Hooks.**

For example, don't declare and pass an Effect Event like this:

```js {4-6,8}

function Timer() {

const [count, setCount] = useState(0);

const onTick = useEffectEvent(() => {

setCount(count + 1);

});

```
useTimer(onTick, 1000); // ■ Avoid: Passing Effect Events

return <h1>{count}</h1>
}

function useTimer(callback, delay) {
useEffect(() => {
const id = setInterval(() => {
callback();
}, delay);
return () => {
clearInterval(id);
};
}, [delay, callback]); // Need to specify "callback" in dependencies
}
```

Instead, always declare Effect Events directly next to the Effects that use them:

```js {10-12,16,21}
function Timer() {
const [count, setCount] = useState(0);
useTimer(() => {
setCount(count + 1);
}, 1000);
return <h1>{count}</h1>
}

function useTimer(callback, delay) {
const onTick = useEffectEvent(() => {
callback();
});

useEffect(() => {
const id = setInterval(() => {
onTick(); // ■ Good: Only called locally inside an Effect
}, delay);
return () => {
clearInterval(id);
```

};

}, [delay]); // No need to specify "onTick" (an Effect Event) as a dependency

}

```

Effect Events are non-reactive "pieces" of your Effect code. They should be next to the Effect using them.

<Recap>

- Event handlers run in response to specific interactions.
- Effects run whenever synchronization is needed.
- Logic inside event handlers is not reactive.
- Logic inside Effects is reactive.
- You can move non-reactive logic from Effects into Effect Events.
- Only call Effect Events from inside Effects.
- Don't pass Effect Events to other components or Hooks.

</Recap>

<Challenges>

#### Fix a variable that doesn't update {/*fix-a-variable-that-doesnt-update*/}

This `Timer` component keeps a `count` state variable which increases every second. The value by which it's increasing is stored in the `increment` state variable. You can control the `increment` variable with the plus and minus buttons.

However, no matter how many times you click the plus button, the counter is still incremented by one every second. What's wrong with this code? Why is `increment` always equal to `1` inside the Effect's code? Find the mistake and fix it.

<Hint>

To fix this code, it's enough to follow the rules.

</Hint>

<Sandpack>

```js
import { useState, useEffect } from 'react';

export default function Timer() {
const [count, setCount] = useState(0);
const [increment, setIncrement] = useState(1);

useEffect(() => {
const id = setInterval(() => {
setCount(c => c + increment);
}, 1000);
return () => {
clearInterval(id);
};
// eslint-disable-next-line react-hooks/exhaustive-deps
}, []);

return (
<>
<h1>
Counter: {count}
<button onClick={() => setCount(0)}>Reset</button>
</h1>
<hr />

Every second, increment by:
<button disabled={increment === 0} onClick={() => {
setIncrement(i => i - 1);
}}>–</button>
{increment}
<button onClick={() => {
setIncrement(i => i + 1);
}}>+</button>

</>
);
}
```

```css
button { margin: 10px; }
```

</Sandpack>
```

<Solution>

As usual, when you're looking for bugs in Effects, start by searching for linter suppressions.

If you remove the suppression comment, React will tell you that this Effect's code depends on `increment`, but you "lied" to React by claiming that this Effect does not depend on any reactive values (`[]`). Add `increment` to the dependency array:

<Sandpack>

```js
import { useState, useEffect } from 'react';

export default function Timer() {
const [count, setCount] = useState(0);
const [increment, setIncrement] = useState(1);

useEffect(() => {
const id = setInterval(() => {
setCount(c => c + increment);
}, 1000);
return () => {
clearInterval(id);
};
}, [increment]);

return (
<>
<h1>
Counter: {count}
<button onClick={() => setCount(0)}>Reset</button>
</h1>
<hr />
<p>
Every second, increment by:
<button disabled={increment === 0} onClick={() => {
setIncrement(i => i - 1);
}}>–</button>
<b>{increment}</b>
<button onClick={() => {
setIncrement(i => i + 1);
```

}}>+</button>

</p>

</>

);

}

```

```css

button { margin: 10px; }

```

</Sandpack>

Now, when `increment` changes, React will re-synchronize your Effect, which will restart the interval.

</Solution>

Fix a freezing counter {/*fix-a-freezing-counter*/}

This `Timer` component keeps a `count` state variable which increases every second. The value by which it's increasing is stored in the `increment` state variable, which you can control it with the plus and minus buttons. For example, try pressing the plus button nine times, and notice that the `count` now increases each second by ten rather than by one.

There is a small issue with this user interface. You might notice that if you keep pressing the plus or minus buttons faster than once per second, the timer itself seems to pause. It only resumes after a second passes since the last time you've pressed either button. Find why this is happening, and fix the issue so that the timer ticks on *every* second without interruptions.

<Hint>

It seems like the Effect which sets up the timer "reacts" to the `increment` value. Does the line that uses the current `increment` value in order to call `setCount` really need to be reactive?

</Hint>

<Sandpack>

```json package.json hidden

{

"dependencies": {

"react": "experimental",

"react-dom": "experimental",

"react-scripts": "latest"

},

"scripts": {

"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
}
}
```

```js
import { useState, useEffect } from 'react';
import { experimental_useEffectEvent as useEffectEvent } from 'react';

export default function Timer() {
const [count, setCount] = useState(0);
const [increment, setIncrement] = useState(1);

useEffect(() => {
const id = setInterval(() => {
setCount(c => c + increment);
}, 1000);
return () => {
clearInterval(id);
};
}, [increment]);

return (
<>
<h1>
Counter: {count}
<button onClick={() => setCount(0)}>Reset</button>
</h1>
<hr />
<p>
Every second, increment by:
<button disabled={increment === 0} onClick={() => {
setIncrement(i => i - 1);
}}>–</button>
{increment}

<button onClick={() => {

setIncrement(i => i + 1);

}}>+</button>

</p>

</>

);

}

```

```css

button { margin: 10px; }

```

</Sandpack>

<Solution>

The issue is that the code inside the Effect uses the `increment` state variable. Since it's a dependency of your Effect, every change to `increment` causes the Effect to re-synchronize, which causes the interval to clear. If you keep clearing the interval every time before it has a chance to fire, it will appear as if the timer has stalled.

To solve the issue, extract an `onTick` Effect Event from the Effect:

<Sandpack>

```json package.json hidden

{

"dependencies": {

"react": "experimental",

"react-dom": "experimental",

"react-scripts": "latest"

},

"scripts": {

"start": "react-scripts start",

"build": "react-scripts build",

"test": "react-scripts test --env=jsdom",

"eject": "react-scripts eject"

}

}

```
```

```js
import { useState, useEffect } from 'react';
import { experimental_useEffectEvent as useEffectEvent } from 'react';

export default function Timer() {
const [count, setCount] = useState(0);
const [increment, setIncrement] = useState(1);

const onTick = useEffectEvent(() => {
setCount(c => c + increment);
});

useEffect(() => {
const id = setInterval(() => {
onTick();
}, 1000);
return () => {
clearInterval(id);
};
}, []);

return (
<>
<h1>
Counter: {count}
<button onClick={() => setCount(0)}>Reset</button>
</h1>
<hr />
<p>
Every second, increment by:
<button disabled={increment === 0} onClick={() => {
setIncrement(i => i - 1);
}}>–</button>
<b>{increment}</b>
<button onClick={() => {
setIncrement(i => i + 1);
}}>+</button>
</p>
```

</>

);

}

```

```css

button { margin: 10px; }

```

</Sandpack>

Since `onTick` is an Effect Event, the code inside it isn't reactive. The change to `increment` does not trigger any Effects.

</Solution>

Fix a non-adjustable delay {/*fix-a-non-adjustable-delay*/}

In this example, you can customize the interval delay. It's stored in a `delay` state variable which is updated by two buttons. However, even if you press the "plus 100 ms" button until the `delay` is 1000 milliseconds (that is, a second), you'll notice that the timer still increments very fast (every 100 ms). It's as if your changes to the `delay` are ignored. Find and fix the bug.

<Hint>

Code inside Effect Events is not reactive. Are there cases in which you would _want_ the `setInterval` call to re-run?

</Hint>

<Sandpack>

```json package.json hidden

{

"dependencies": {

"react": "experimental",

"react-dom": "experimental",

"react-scripts": "latest"

},

"scripts": {

"start": "react-scripts start",

"build": "react-scripts build",

"test": "react-scripts test --env=jsdom",

"eject": "react-scripts eject"

```
}
}
```

```js
import { useState, useEffect } from 'react';
import { experimental_useEffectEvent as useEffectEvent } from 'react';

export default function Timer() {
const [count, setCount] = useState(0);
const [increment, setIncrement] = useState(1);
const [delay, setDelay] = useState(100);

const onTick = useEffectEvent(() => {
setCount(c => c + increment);
});

const onMount = useEffectEvent(() => {
return setInterval(() => {
onTick();
}, delay);
});

useEffect(() => {
const id = onMount();
return () => {
clearInterval(id);
}
}, []);

return (
<>
<h1>
Counter: {count}
<button onClick={() => setCount(0)}>Reset</button>
</h1>
<hr />
<p>
Increment by:
```

<button disabled={increment === 0} onClick={() => {
setIncrement(i => i - 1);
}}>–</button>
{increment}
<button onClick={() => {
setIncrement(i => i + 1);
}}>+</button>

Increment delay:
<button disabled={delay === 100} onClick={() => {
setDelay(d => d - 100);
}}>–100 ms</button>
{delay} ms
<button onClick={() => {
setDelay(d => d + 100);
}}>+100 ms</button>

</>
);
}
```

```css
button { margin: 10px; }
```

</Sandpack>

<Solution>

The problem with the above example is that it extracted an Effect Event called `onMount` without considering what the code should actually be doing. You should only extract Effect Events for a specific reason: when you want to make a part of your code non-reactive. However, the `setInterval` call *should* be reactive with respect to the `delay` state variable. If the `delay` changes, you want to set up the interval from scratch! To fix this code, pull all the reactive code back inside the Effect:

<Sandpack>

```json package.json hidden

````
{
"dependencies": {
"react": "experimental",
"react-dom": "experimental",
"react-scripts": "latest"
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
}
}
```

```js
import { useState, useEffect } from 'react';
import { experimental_useEffectEvent as useEffectEvent } from 'react';

export default function Timer() {
const [count, setCount] = useState(0);
const [increment, setIncrement] = useState(1);
const [delay, setDelay] = useState(100);

const onTick = useEffectEvent(() => {
setCount(c => c + increment);
});

useEffect(() => {
const id = setInterval(() => {
onTick();
}, delay);
return () => {
clearInterval(id);
}
}, [delay]);

return (
<>
````

```
<h1>
Counter: {count}
<button onClick={() => setCount(0)}>Reset</button>
</h1>
<hr />
<p>
Increment by:
<button disabled={increment === 0} onClick={() => {
setIncrement(i => i - 1);
}}>–</button>
<b>{increment}</b>
<button onClick={() => {
setIncrement(i => i + 1);
}}>+</button>
</p>
<p>
Increment delay:
<button disabled={delay === 100} onClick={() => {
setDelay(d => d - 100);
}}>–100 ms</button>
<b>{delay} ms</b>
<button onClick={() => {
setDelay(d => d + 100);
}}>+100 ms</button>
</p>
</>
);
}
```

```css
button { margin: 10px; }
```

</Sandpack>

In general, you should be suspicious of functions like `onMount` that focus on the *timing* rather than the *purpose* of a piece of code. It may feel "more descriptive" at first but it obscures your intent. As a

rule of thumb, Effect Events should correspond to something that happens from the *user's* perspective. For example, `onMessage`, `onTick`, `onVisit`, or `onConnected` are good Effect Event names. Code inside them would likely not need to be reactive. On the other hand, `onMount`, `onUpdate`, `onUnmount`, or `onAfterRender` are so generic that it's easy to accidentally put code that *should* be reactive into them. This is why you should name your Effect Events after *what the user thinks has happened,* not when some code happened to run.

</Solution>

Fix a delayed notification {/*fix-a-delayed-notification*/}

When you join a chat room, this component shows a notification. However, it doesn't show the notification immediately. Instead, the notification is artificially delayed by two seconds so that the user has a chance to look around the UI.

This almost works, but there is a bug. Try changing the dropdown from "general" to "travel" and then to "music" very quickly. If you do it fast enough, you will see two notifications (as expected!) but they will *both* say "Welcome to music".

Fix it so that when you switch from "general" to "travel" and then to "music" very quickly, you see two notifications, the first one being "Welcome to travel" and the second one being "Welcome to music". (For an additional challenge, assuming you've *already* made the notifications show the correct rooms, change the code so that only the latter notification is displayed.)

<Hint>

Your Effect knows which room it connected to. Is there any information that you might want to pass to your Effect Event?

</Hint>

<Sandpack>

```json package.json hidden
{
"dependencies": {
"react": "experimental",
"react-dom": "experimental",
"react-scripts": "latest",
"toastify-js": "1.12.0"
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
```

```
}
}
```

```js
import { useState, useEffect } from 'react';
import { experimental_useEffectEvent as useEffectEvent } from 'react';
import { createConnection, sendMessage } from './chat.js';
import { showNotification } from './notifications.js';

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

function ChatRoom({ roomId, theme }) {
const onConnected = useEffectEvent(() => {
showNotification('Welcome to ' + roomId, theme);
});

useEffect(() => {
const connection = createConnection(serverUrl, roomId);
connection.on('connected', () => {
setTimeout(() => {
onConnected();
}, 2000);
});
connection.connect();
return () => connection.disconnect();
}, [roomId]);

return <h1>Welcome to the {roomId} room!</h1>
}

export default function App() {
const [roomId, setRoomId] = useState('general');
const [isDark, setIsDark] = useState(false);
return (
<>
<label>
Choose the chat room:{' '}
<select
```

```
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
</select>
</label>
<label>
<input
type="checkbox"
checked={isDark}
onChange={e => setIsDark(e.target.checked)}
/>
Use dark theme
</label>
<hr />
<ChatRoom
roomId={roomId}
theme={isDark ? 'dark' : 'light'}
/>
</>
);
}
```

```js chat.js
export function createConnection(serverUrl, roomId) {
// A real implementation would actually connect to the server
let connectedCallback;
let timeout;
return {
connect() {
timeout = setTimeout(() => {
if (connectedCallback) {
connectedCallback();
```

```
}
}, 100);
},
on(event, callback) {
if (connectedCallback) {
throw Error('Cannot add the handler twice.');
}
if (event !== 'connected') {
throw Error('Only "connected" event is supported.');
}
connectedCallback = callback;
},
disconnect() {
clearTimeout(timeout);
}
};
}
```

```js notifications.js hidden
import Toastify from 'toastify-js';
import 'toastify-js/src/toastify.css';

export function showNotification(message, theme) {
Toastify({
text: message,
duration: 2000,
gravity: 'top',
position: 'right',
style: {
background: theme === 'dark' ? 'black' : 'white',
color: theme === 'dark' ? 'white' : 'black',
},
}).showToast();
}
```

```css
label { display: block; margin-top: 10px; }
```

</Sandpack>

<Solution>

Inside your Effect Event, `roomId` is the value *at the time Effect Event was called.*

Your Effect Event is called with a two second delay. If you're quickly switching from the travel to the music room, by the time the travel room's notification shows, `roomId` is already `"music"`. This is why both notifications say "Welcome to music".

To fix the issue, instead of reading the *latest* `roomId` inside the Effect Event, make it a parameter of your Effect Event, like `connectedRoomId` below. Then pass `roomId` from your Effect by calling `onConnected(roomId)`:

<Sandpack>

```json package.json hidden
{
"dependencies": {
"react": "experimental",
"react-dom": "experimental",
"react-scripts": "latest",
"toastify-js": "1.12.0"
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
}
}
```

```js
import { useState, useEffect } from 'react';
import { experimental_useEffectEvent as useEffectEvent } from 'react';
import { createConnection, sendMessage } from './chat.js';
import { showNotification } from './notifications.js';
```

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

function ChatRoom({ roomId, theme }) {
const onConnected = useEffectEvent(connectedRoomId => {
showNotification('Welcome to ' + connectedRoomId, theme);
});

useEffect(() => {
const connection = createConnection(serverUrl, roomId);
connection.on('connected', () => {
setTimeout(() => {
onConnected(roomId);
}, 2000);
});
connection.connect();
return () => connection.disconnect();
}, [roomId]);

return <h1>Welcome to the {roomId} room!</h1>
}

export default function App() {
const [roomId, setRoomId] = useState('general');
const [isDark, setIsDark] = useState(false);
return (
<>
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
</select>
</label>
<label>
```

```
<input
type="checkbox"
checked={isDark}
onChange={e => setIsDark(e.target.checked)}
/>
Use dark theme
</label>
<hr />
<ChatRoom
roomId={roomId}
theme={isDark ? 'dark' : 'light'}
/>
</>
);
}
```

```js chat.js
export function createConnection(serverUrl, roomId) {
// A real implementation would actually connect to the server
let connectedCallback;
let timeout;
return {
connect() {
timeout = setTimeout(() => {
if (connectedCallback) {
connectedCallback();
}
}, 100);
},
on(event, callback) {
if (connectedCallback) {
throw Error('Cannot add the handler twice.');
}
if (event !== 'connected') {
throw Error('Only "connected" event is supported.');
```

```
}
connectedCallback = callback;
},
disconnect() {
clearTimeout(timeout);
}
};
}
```

```js notifications.js hidden
import Toastify from 'toastify-js';
import 'toastify-js/src/toastify.css';

export function showNotification(message, theme) {
Toastify({
text: message,
duration: 2000,
gravity: 'top',
position: 'right',
style: {
background: theme === 'dark' ? 'black' : 'white',
color: theme === 'dark' ? 'white' : 'black',
},
}).showToast();
}
```

```css
label { display: block; margin-top: 10px; }
```

</Sandpack>

The Effect that had `roomId` set to `"travel"` (so it connected to the `"travel"` room) will show the notification for `"travel"`. The Effect that had `roomId` set to `"music"` (so it connected to the `"music"` room) will show the notification for `"music"`. In other words, `connectedRoomId` comes from your Effect (which is reactive), while `theme` always uses the latest value.

To solve the additional challenge, save the notification timeout ID and clear it in the cleanup function of your Effect:

<Sandpack>

```json package.json hidden
{
"dependencies": {
"react": "experimental",
"react-dom": "experimental",
"react-scripts": "latest",
"toastify-js": "1.12.0"
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
}
}
```

```js
import { useState, useEffect } from 'react';
import { experimental_useEffectEvent as useEffectEvent } from 'react';
import { createConnection, sendMessage } from './chat.js';
import { showNotification } from './notifications.js';

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

function ChatRoom({ roomId, theme }) {
const onConnected = useEffectEvent(connectedRoomId => {
showNotification('Welcome to ' + connectedRoomId, theme);
});

useEffect(() => {
const connection = createConnection(serverUrl, roomId);
let notificationTimeoutId;
connection.on('connected', () => {
notificationTimeoutId = setTimeout(() => {
onConnected(roomId);
}, 2000);
```

```
});
connection.connect();
return () => {
connection.disconnect();
if (notificationTimeoutId !== undefined) {
clearTimeout(notificationTimeoutId);
}
};
}, [roomId]);

return <h1>Welcome to the {roomId} room!</h1>
}

export default function App() {
const [roomId, setRoomId] = useState('general');
const [isDark, setIsDark] = useState(false);
return (
<>
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
</select>
</label>
<label>
<input
type="checkbox"
checked={isDark}
onChange={e => setIsDark(e.target.checked)}
/>
Use dark theme
</label>
```

```
<hr />
<ChatRoom
roomId={roomId}
theme={isDark ? 'dark' : 'light'}
/>
</>
);
}
```

```js chat.js
export function createConnection(serverUrl, roomId) {
// A real implementation would actually connect to the server
let connectedCallback;
let timeout;
return {
connect() {
timeout = setTimeout(() => {
if (connectedCallback) {
connectedCallback();
}
}, 100);
},
on(event, callback) {
if (connectedCallback) {
throw Error('Cannot add the handler twice.');
}
if (event !== 'connected') {
throw Error('Only "connected" event is supported.');
}
connectedCallback = callback;
},
disconnect() {
clearTimeout(timeout);
}
};
```

}
```

```js notifications.js hidden
import Toastify from 'toastify-js';
import 'toastify-js/src/toastify.css';

export function showNotification(message, theme) {
Toastify({
text: message,
duration: 2000,
gravity: 'top',
position: 'right',
style: {
background: theme === 'dark' ? 'black' : 'white',
color: theme === 'dark' ? 'white' : 'black',
},
}).showToast();
}
```

```css
label { display: block; margin-top: 10px; }
```

</Sandpack>

This ensures that already scheduled (but not yet displayed) notifications get cancelled when you change rooms.

</Solution>

</Challenges>

---

title: State as a Snapshot

<Intro>

State variables might look like regular JavaScript variables that you can read and write to. However, state behaves more like a snapshot. Setting it does not change the state variable you already have, but instead triggers a re-render.
```

</Intro>

<YouWillLearn>

* How setting state triggers re-renders
* When and how state updates
* Why state does not update immediately after you set it
* How event handlers access a "snapshot" of the state

</YouWillLearn>

Setting state triggers renders {/*setting-state-triggers-renders*/}

You might think of your user interface as changing directly in response to the user event like a click. In React, it works a little differently from this mental model. On the previous page, you saw that [setting state requests a re-render](/learn/render-and-commit#step-1-trigger-a-render) from React. This means that for an interface to react to the event, you need to *update the state*.

In this example, when you press "send", `setIsSent(true)` tells React to re-render the UI:

<Sandpack>

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

export default function Form() {
const [isSent, setIsSent] = useState(false);
const [message, setMessage] = useState('Hi!');
if (isSent) {
return <h1>Your message is on its way!</h1>
}
return (
<form onSubmit={(e) => {
e.preventDefault();
setIsSent(true);
sendMessage(message);
}}>
<textarea
placeholder="Message"
value={message}
onChange={e => setMessage(e.target.value)}
/>
```

<button type="submit">Send</button>

</form>

);

}

function sendMessage(message) {

// ...

}

```

```css

label, textarea { margin-bottom: 10px; display: block; }

```

</Sandpack>

Here's what happens when you click the button:

1. The `onSubmit` event handler executes.

2. `setIsSent(true)` sets `isSent` to `true` and queues a new render.

3. React re-renders the component according to the new `isSent` value.

Let's take a closer look at the relationship between state and rendering.

Rendering takes a snapshot in time {/*rendering-takes-a-snapshot-in-time*/}

["Rendering"](/learn/render-and-commit#step-2-react-renders-your-components) means that React is calling your component, which is a function. The JSX you return from that function is like a snapshot of the UI in time. Its props, event handlers, and local variables were all calculated **using its state at the time of the render.**

Unlike a photograph or a movie frame, the UI "snapshot" you return is interactive. It includes logic like event handlers that specify what happens in response to inputs. React updates the screen to match this snapshot and connects the event handlers. As a result, pressing a button will trigger the click handler from your JSX.

When React re-renders a component:

1. React calls your function again.

2. Your function returns a new JSX snapshot.

3. React then updates the screen to match the snapshot you've returned.

<IllustrationBlock sequential>

<Illustration caption="React executing the function" src="/images/docs/illustrations/i_render1.png" />

<Illustration caption="Calculating the snapshot" src="/images/docs/illustrations/i_render2.png" />
```

<Illustration caption="Updating the DOM tree" src="/images/docs/illustrations/i_render3.png" />

</IllustrationBlock>

As a component's memory, state is not like a regular variable that disappears after your function returns. State actually "lives" in React itself--as if on a shelf!--outside of your function. When React calls your component, it gives you a snapshot of the state for that particular render. Your component returns a snapshot of the UI with a fresh set of props and event handlers in its JSX, all calculated **using the state values from that render!**

<IllustrationBlock sequential>

<Illustration caption="You tell React to update the state"
src="/images/docs/illustrations/i_state-snapshot1.png" />

<Illustration caption="React updates the state value"
src="/images/docs/illustrations/i_state-snapshot2.png" />

<Illustration caption="React passes a snapshot of the state value into the component"
src="/images/docs/illustrations/i_state-snapshot3.png" />

</IllustrationBlock>

Here's a little experiment to show you how this works. In this example, you might expect that clicking the "+3" button would increment the counter three times because it calls `setNumber(number + 1)` three times.

See what happens when you click the "+3" button:

<Sandpack>

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

export default function Counter() {
const [number, setNumber] = useState(0);

return (
<>
<h1>{number}</h1>
<button onClick={() => {
setNumber(number + 1);
setNumber(number + 1);
setNumber(number + 1);
}}>+3</button>
</>
)
}
```

```css
button { display: inline-block; margin: 10px; font-size: 20px; }
h1 { display: inline-block; margin: 10px; width: 30px; text-align: center; }
```

</Sandpack>

Notice that `number` only increments once per click!

Setting state only changes it for the *next* render. During the first render, `number` was `0`. This is why, in *that render's* `onClick` handler, the value of `number` is still `0` even after `setNumber(number + 1)` was called:

```js
<button onClick={() => {
setNumber(number + 1);
setNumber(number + 1);
setNumber(number + 1);
}}>+3</button>
```

Here is what this button's click handler tells React to do:

1. `setNumber(number + 1)`: `number` is `0` so `setNumber(0 + 1)`.
- React prepares to change `number` to `1` on the next render.
2. `setNumber(number + 1)`: `number` is `0` so `setNumber(0 + 1)`.
- React prepares to change `number` to `1` on the next render.
3. `setNumber(number + 1)`: `number` is `0` so `setNumber(0 + 1)`.
- React prepares to change `number` to `1` on the next render.

Even though you called `setNumber(number + 1)` three times, in *this render's* event handler `number` is always `0`, so you set the state to `1` three times. This is why, after your event handler finishes, React re-renders the component with `number` equal to `1` rather than `3`.

You can also visualize this by mentally substituting state variables with their values in your code. Since the `number` state variable is `0` for *this render*, its event handler looks like this:

```js
<button onClick={() => {
setNumber(0 + 1);
setNumber(0 + 1);
setNumber(0 + 1);
}}>+3</button>
```

```

For the next render, `number` is `1`, so *that render's* click handler looks like this:

```js
<button onClick={() => {
setNumber(1 + 1);
setNumber(1 + 1);
setNumber(1 + 1);
}}>+3</button>
```

This is why clicking the button again will set the counter to `2`, then to `3` on the next click, and so on.

State over time {/*state-over-time*/}

Well, that was fun. Try to guess what clicking this button will alert:

<Sandpack>

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

export default function Counter() {
const [number, setNumber] = useState(0);

return (
<>
<h1>{number}</h1>
<button onClick={() => {
setNumber(number + 5);
alert(number);
}}>+5</button>
</>
)
}
```

```css
button { display: inline-block; margin: 10px; font-size: 20px; }
h1 { display: inline-block; margin: 10px; width: 30px; text-align: center; }
```
```

</Sandpack>

If you use the substitution method from before, you can guess that the alert shows "0":

```js
setNumber(0 + 5);
alert(0);
```

But what if you put a timer on the alert, so it only fires _after_ the component re-rendered? Would it say "0" or "5"? Have a guess!

<Sandpack>

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

export default function Counter() {
const [number, setNumber] = useState(0);

return (
<>
<h1>{number}</h1>
<button onClick={() => {
setNumber(number + 5);
setTimeout(() => {
alert(number);
}, 3000);
}}>+5</button>
</>
)
}
```

```css
button { display: inline-block; margin: 10px; font-size: 20px; }
h1 { display: inline-block; margin: 10px; width: 30px; text-align: center; }
```

</Sandpack>

Surprised? If you use the substitution method, you can see the "snapshot" of the state passed to the alert.

```js
setNumber(0 + 5);
setTimeout(() => {
alert(0);
}, 3000);
```

The state stored in React may have changed by the time the alert runs, but it was scheduled using a snapshot of the state at the time the user interacted with it!

A state variable's value never changes within a render, even if its event handler's code is asynchronous. Inside *that render's* `onClick`, the value of `number` continues to be `0` even after `setNumber(number + 5)` was called. Its value was "fixed" when React "took the snapshot" of the UI by calling your component.

Here is an example of how that makes your event handlers less prone to timing mistakes. Below is a form that sends a message with a five-second delay. Imagine this scenario:

1. You press the "Send" button, sending "Hello" to Alice.
2. Before the five-second delay ends, you change the value of the "To" field to "Bob".

What do you expect the `alert` to display? Would it display, "You said Hello to Alice"? Or would it display, "You said Hello to Bob"? Make a guess based on what you know, and then try it:

<Sandpack>

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

export default function Form() {
const [to, setTo] = useState('Alice');
const [message, setMessage] = useState('Hello');

function handleSubmit(e) {
e.preventDefault();
setTimeout(() => {
alert(`You said ${message} to ${to}`);
}, 5000);
}

return (
<form onSubmit={handleSubmit}>
<label>
To:{' '}
```

<select

value={to}

onChange={e => setTo(e.target.value)}>

<option value="Alice">Alice</option>

<option value="Bob">Bob</option>

</select>

</label>

<textarea

placeholder="Message"

value={message}

onChange={e => setMessage(e.target.value)}

/>

<button type="submit">Send</button>

</form>

);

}

```

```css

label, textarea { margin-bottom: 10px; display: block; }

```

</Sandpack>

React keeps the state values "fixed" within one render's event handlers. You don't need to worry whether the state has changed while the code is running.

But what if you wanted to read the latest state before a re-render? You'll want to use a [state updater function](/learn/queueing-a-series-of-state-updates), covered on the next page!

<Recap>

* Setting state requests a new render.

* React stores state outside of your component, as if on a shelf.

* When you call `useState`, React gives you a snapshot of the state *for that render*.

* Variables and event handlers don't "survive" re-renders. Every render has its own event handlers.

* Every render (and functions inside it) will always "see" the snapshot of the state that React gave to *that* render.

* You can mentally substitute state in event handlers, similarly to how you think about the rendered JSX.

* Event handlers created in the past have the state values from the render in which they were created.
```

</Recap>

<Challenges>

Implement a traffic light {/*implement-a-traffic-light*/}

Here is a crosswalk light component that toggles when the button is pressed:

<Sandpack>

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

export default function TrafficLight() {
const [walk, setWalk] = useState(true);

function handleClick() {
setWalk(!walk);
}

return (
<>
<button onClick={handleClick}>
Change to {walk ? 'Stop' : 'Walk'}
</button>
<h1 style={{
color: walk ? 'darkgreen' : 'darkred'
}}>
{walk ? 'Walk' : 'Stop'}
</h1>
</>
);
}
```

```css
h1 { margin-top: 20px; }
```

</Sandpack>

Add an `alert` to the click handler. When the light is green and says "Walk", clicking the button should say "Stop is next". When the light is red and says "Stop", clicking the button should say "Walk is next".

Does it make a difference whether you put the `alert` before or after the `setWalk` call?

<Solution>

Your `alert` should look like this:

<Sandpack>

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

export default function TrafficLight() {
const [walk, setWalk] = useState(true);

function handleClick() {
setWalk(!walk);
alert(walk ? 'Stop is next' : 'Walk is next');
}

return (
<>
<button onClick={handleClick}>
Change to {walk ? 'Stop' : 'Walk'}
</button>
<h1 style={{
color: walk ? 'darkgreen' : 'darkred'
}}>
{walk ? 'Walk' : 'Stop'}
</h1>
</>
);
}
```

```css
h1 { margin-top: 20px; }
```

</Sandpack>

Whether you put it before or after the `setWalk` call makes no difference. That render's value of `walk` is fixed. Calling `setWalk` will only change it for the *next* render, but will not affect the event handler from the previous render.

This line might seem counter-intuitive at first:

```js
alert(walk ? 'Stop is next' : 'Walk is next');
```

But it makes sense if you read it as: "If the traffic light shows 'Walk now', the message should say 'Stop is next.'" The `walk` variable inside your event handler matches that render's value of `walk` and does not change.

You can verify that this is correct by applying the substitution method. When `walk` is `true`, you get:

```js
<button onClick={() => {
setWalk(false);
alert('Stop is next');
}}>
Change to Stop
</button>
<h1 style={{color: 'darkgreen'}}>
Walk
</h1>
```

So clicking "Change to Stop" queues a render with `walk` set to `false`, and alerts "Stop is next".

</Solution>

</Challenges>

---

title: Passing Data Deeply with Context

<Intro>

Usually, you will pass information from a parent component to a child component via props. But passing props can become verbose and inconvenient if you have to pass them through many components in the middle, or if many components in your app need the same information. *Context* lets the parent component make some information available to any component in the tree below it—no matter how deep—without passing it explicitly through props.

</Intro>

<YouWillLearn>

- What "prop drilling" is
- How to replace repetitive prop passing with context
- Common use cases for context
- Common alternatives to context

</YouWillLearn>

The problem with passing props {/*the-problem-with-passing-props*/}

[Passing props](/learn/passing-props-to-a-component) is a great way to explicitly pipe data through your UI tree to the components that use it.

But passing props can become verbose and inconvenient when you need to pass some prop deeply through the tree, or if many components need the same prop. The nearest common ancestor could be far removed from the components that need data, and [lifting state up](/learn/sharing-state-between-components) that high can lead to a situation called "prop drilling".

<DiagramGroup>

<Diagram name="passing_data_lifting_state" height={160} width={608} captionPosition="top" alt="Diagram with a tree of three components. The parent contains a bubble representing a value highlighted in purple. The value flows down to each of the two children, both highlighted in purple." >

Lifting state up

</Diagram>

<Diagram name="passing_data_prop_drilling" height={430} width={608} captionPosition="top" alt="Diagram with a tree of ten nodes, each node with two children or less. The root node contains a bubble representing a value highlighted in purple. The value flows down through the two children, each of which pass the value but do not contain it. The left child passes the value down to two children which are both highlighted purple. The right child of the root passes the value through to one of its two children - the right one, which is highlighted purple. That child passed the value through its single child, which passes it down to both of its two children, which are highlighted purple.">

Prop drilling

</Diagram>

</DiagramGroup>

Wouldn't it be great if there were a way to "teleport" data to the components in the tree that need it without passing props? With React's context feature, there is!

Context: an alternative to passing props {/*context-an-alternative-to-passing-props*/}

Context lets a parent component provide data to the entire tree below it. There are many uses for context. Here is one example. Consider this `Heading` component that accepts a `level` for its size:

<Sandpack>

```js
import Heading from './Heading.js';
import Section from './Section.js';

export default function Page() {
return (
<Section>
<Heading level={1}>Title</Heading>
<Heading level={2}>Heading</Heading>
<Heading level={3}>Sub-heading</Heading>
<Heading level={4}>Sub-sub-heading</Heading>
<Heading level={5}>Sub-sub-sub-heading</Heading>
<Heading level={6}>Sub-sub-sub-sub-heading</Heading>
</Section>
);
}
```

```js Section.js
export default function Section({ children }) {
return (
<section className="section">
{children}
</section>
);
}
```

```js Heading.js
export default function Heading({ level, children }) {
switch (level) {
case 1:
return <h1>{children}</h1>;
case 2:
return <h2>{children}</h2>;
case 3:
```

return <h3>{children}</h3>;
case 4:
return <h4>{children}</h4>;
case 5:
return <h5>{children}</h5>;
case 6:
return <h6>{children}</h6>;
default:
throw Error('Unknown level: ' + level);
}
}
```

```css
.section {
padding: 10px;
margin: 5px;
border-radius: 5px;
border: 1px solid #aaa;
}
```

</Sandpack>

Let's say you want multiple headings within the same `Section` to always have the same size:

<Sandpack>

```js
import Heading from './Heading.js';
import Section from './Section.js';

export default function Page() {
return (
<Section>
<Heading level={1}>Title</Heading>
<Section>
<Heading level={2}>Heading</Heading>
<Heading level={2}>Heading</Heading>

<Heading level={2}>Heading</Heading>
<Section>
<Heading level={3}>Sub-heading</Heading>
<Heading level={3}>Sub-heading</Heading>
<Heading level={3}>Sub-heading</Heading>
<Section>
<Heading level={4}>Sub-sub-heading</Heading>
<Heading level={4}>Sub-sub-heading</Heading>
<Heading level={4}>Sub-sub-heading</Heading>
</Section>
</Section>
</Section>
</Section>
);
}
```

```js Section.js
export default function Section({ children }) {
return (
<section className="section">
{children}
</section>
);
}
```

```js Heading.js
export default function Heading({ level, children }) {
switch (level) {
case 1:
return <h1>{children}</h1>;
case 2:
return <h2>{children}</h2>;
case 3:
return <h3>{children}</h3>;
case 4:

return <h4>{children}</h4>;
case 5:
return <h5>{children}</h5>;
case 6:
return <h6>{children}</h6>;
default:
throw Error('Unknown level: ' + level);
}
}
```

```css
.section {
padding: 10px;
margin: 5px;
border-radius: 5px;
border: 1px solid #aaa;
}
```

</Sandpack>

Currently, you pass the `level` prop to each `<Heading>` separately:

```js
<Section>
<Heading level={3}>About</Heading>
<Heading level={3}>Photos</Heading>
<Heading level={3}>Videos</Heading>
</Section>
```

It would be nice if you could pass the `level` prop to the `<Section>` component instead and remove it from the `<Heading>`. This way you could enforce that all headings in the same section have the same size:

```js
<Section level={3}>
<Heading>About</Heading>
<Heading>Photos</Heading>

<Heading>Videos</Heading>

</Section>

```

But how can the `<Heading>` component know the level of its closest `<Section>`? **That would require some way for a child to "ask" for data from somewhere above in the tree.**

You can't do it with props alone. This is where context comes into play. You will do it in three steps:

1. **Create** a context. (You can call it `LevelContext`, since it's for the heading level.)

2. **Use** that context from the component that needs the data. (`Heading` will use `LevelContext`.)

3. **Provide** that context from the component that specifies the data. (`Section` will provide `LevelContext`.)

Context lets a parent--even a distant one!--provide some data to the entire tree inside of it.

<DiagramGroup>

<Diagram name="passing_data_context_close" height={160} width={608} captionPosition="top" alt="Diagram with a tree of three components. The parent contains a bubble representing a value highlighted in orange which projects down to the two children, each highlighted in orange." >

Using context in close children

</Diagram>

<Diagram name="passing_data_context_far" height={430} width={608} captionPosition="top" alt="Diagram with a tree of ten nodes, each node with two children or less. The root parent node contains a bubble representing a value highlighted in orange. The value projects down directly to four leaves and one intermediate component in the tree, which are all highlighted in orange. None of the other intermediate components are highlighted.">

Using context in distant children

</Diagram>

</DiagramGroup>

Step 1: Create the context {/*step-1-create-the-context*/}

First, you need to create the context. You'll need to **export it from a file** so that your components can use it:

<Sandpack>

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

import Section from './Section.js';

export default function Page() {

```
return (
<Section>
<Heading level={1}>Title</Heading>
<Section>
<Heading level={2}>Heading</Heading>
<Heading level={2}>Heading</Heading>
<Heading level={2}>Heading</Heading>
<Section>
<Heading level={3}>Sub-heading</Heading>
<Heading level={3}>Sub-heading</Heading>
<Heading level={3}>Sub-heading</Heading>
<Section>
<Heading level={4}>Sub-sub-heading</Heading>
<Heading level={4}>Sub-sub-heading</Heading>
<Heading level={4}>Sub-sub-heading</Heading>
</Section>
</Section>
</Section>
</Section>
);
}
```

```js Section.js
export default function Section({ children }) {
return (
<section className="section">
{children}
</section>
);
}
```

```js Heading.js
export default function Heading({ level, children }) {
switch (level) {
case 1:
```

```
return <h1>{children}</h1>;
case 2:
return <h2>{children}</h2>;
case 3:
return <h3>{children}</h3>;
case 4:
return <h4>{children}</h4>;
case 5:
return <h5>{children}</h5>;
case 6:
return <h6>{children}</h6>;
default:
throw Error('Unknown level: ' + level);
}
}
```

```js LevelContext.js active
import { createContext } from 'react';

export const LevelContext = createContext(1);
```

```css
.section {
padding: 10px;
margin: 5px;
border-radius: 5px;
border: 1px solid #aaa;
}
```

</Sandpack>

The only argument to `createContext` is the _default_ value. Here, `1` refers to the biggest heading level, but you could pass any kind of value (even an object). You will see the significance of the default value in the next step.

Step 2: Use the context {/*step-2-use-the-context*/}

Import the `useContext` Hook from React and your context:

```js
import { useContext } from 'react';
import { LevelContext } from './LevelContext.js';
```

Currently, the `Heading` component reads `level` from props:

```js
export default function Heading({ level, children }) {
// ...
}
```

Instead, remove the `level` prop and read the value from the context you just imported, `LevelContext`:

```js {2}
export default function Heading({ children }) {
const level = useContext(LevelContext);
// ...
}
```

`useContext` is a Hook. Just like `useState` and `useReducer`, you can only call a Hook immediately inside a React component (not inside loops or conditions). **`useContext` tells React that the `Heading` component wants to read the `LevelContext`.**

Now that the `Heading` component doesn't have a `level` prop, you don't need to pass the level prop to `Heading` in your JSX like this anymore:

```js
<Section>
<Heading level={4}>Sub-sub-heading</Heading>
<Heading level={4}>Sub-sub-heading</Heading>
<Heading level={4}>Sub-sub-heading</Heading>
</Section>
```

Update the JSX so that it's the `Section` that receives it instead:

```jsx
<Section level={4}>
<Heading>Sub-sub-heading</Heading>
```

<Heading>Sub-sub-heading</Heading>
<Heading>Sub-sub-heading</Heading>
</Section>
```

As a reminder, this is the markup that you were trying to get working:

<Sandpack>

```js
import Heading from './Heading.js';
import Section from './Section.js';

export default function Page() {
return (
<Section level={1}>
<Heading>Title</Heading>
<Section level={2}>
<Heading>Heading</Heading>
<Heading>Heading</Heading>
<Heading>Heading</Heading>
<Section level={3}>
<Heading>Sub-heading</Heading>
<Heading>Sub-heading</Heading>
<Heading>Sub-heading</Heading>
<Section level={4}>
<Heading>Sub-sub-heading</Heading>
<Heading>Sub-sub-heading</Heading>
<Heading>Sub-sub-heading</Heading>
</Section>
</Section>
</Section>
</Section>
);
}
```

```js Section.js
export default function Section({ children }) {

```
return (
<section className="section">
{children}
</section>
);
}
```

```js Heading.js
import { useContext } from 'react';
import { LevelContext } from './LevelContext.js';

export default function Heading({ children }) {
const level = useContext(LevelContext);
switch (level) {
case 1:
return <h1>{children}</h1>;
case 2:
return <h2>{children}</h2>;
case 3:
return <h3>{children}</h3>;
case 4:
return <h4>{children}</h4>;
case 5:
return <h5>{children}</h5>;
case 6:
return <h6>{children}</h6>;
default:
throw Error('Unknown level: ' + level);
}
}
```

```js LevelContext.js
import { createContext } from 'react';

export const LevelContext = createContext(1);
```

```css
.section {
padding: 10px;
margin: 5px;
border-radius: 5px;
border: 1px solid #aaa;
}
```

</Sandpack>

Notice this example doesn't quite work, yet! All the headings have the same size because **even though you're *using* the context, you have not *provided* it yet.** React doesn't know where to get it!

If you don't provide the context, React will use the default value you've specified in the previous step. In this example, you specified `1` as the argument to `createContext`, so `useContext(LevelContext)` returns `1`, setting all those headings to `<h1>`. Let's fix this problem by having each `Section` provide its own context.

Step 3: Provide the context {/*step-3-provide-the-context*/}

The `Section` component currently renders its children:

```js
export default function Section({ children }) {
return (
<section className="section">
{children}
</section>
);
}
```

Wrap them with a context provider to provide the `LevelContext` to them:

```js {1,6,8}
import { LevelContext } from './LevelContext.js';

export default function Section({ level, children }) {
return (
<section className="section">
<LevelContext.Provider value={level}>
{children}
```

</LevelContext.Provider>

</section>

);

}

```

This tells React: "if any component inside this `<Section>` asks for `LevelContext`, give them this `level`." The component will use the value of the nearest `<LevelContext.Provider>` in the UI tree above it.

<Sandpack>

```js

import Heading from './Heading.js';

import Section from './Section.js';

export default function Page() {

return (

<Section level={1}>

<Heading>Title</Heading>

<Section level={2}>

<Heading>Heading</Heading>

<Heading>Heading</Heading>

<Heading>Heading</Heading>

<Section level={3}>

<Heading>Sub-heading</Heading>

<Heading>Sub-heading</Heading>

<Heading>Sub-heading</Heading>

<Section level={4}>

<Heading>Sub-sub-heading</Heading>

<Heading>Sub-sub-heading</Heading>

<Heading>Sub-sub-heading</Heading>

</Section>

</Section>

</Section>

</Section>

);

}

```
```

```js Section.js
import { LevelContext } from './LevelContext.js';

export default function Section({ level, children }) {
return (
<section className="section">
<LevelContext.Provider value={level}>
{children}
</LevelContext.Provider>
</section>
);
}
```

```js Heading.js
import { useContext } from 'react';
import { LevelContext } from './LevelContext.js';

export default function Heading({ children }) {
const level = useContext(LevelContext);
switch (level) {
case 1:
return <h1>{children}</h1>;
case 2:
return <h2>{children}</h2>;
case 3:
return <h3>{children}</h3>;
case 4:
return <h4>{children}</h4>;
case 5:
return <h5>{children}</h5>;
case 6:
return <h6>{children}</h6>;
default:
throw Error('Unknown level: ' + level);
}
}
```

```

```js LevelContext.js
import { createContext } from 'react';

export const LevelContext = createContext(1);
```

```css
.section {
padding: 10px;
margin: 5px;
border-radius: 5px;
border: 1px solid #aaa;
}
```

</Sandpack>

It's the same result as the original code, but you did not need to pass the `level` prop to each `Heading` component! Instead, it "figures out" its heading level by asking the closest `Section` above:

1. You pass a `level` prop to the `<Section>`.
2. `Section` wraps its children into `<LevelContext.Provider value={level}>`.
3. `Heading` asks the closest value of `LevelContext` above with `useContext(LevelContext)`.

Using and providing context from the same component
{/*using-and-providing-context-from-the-same-component*/}

Currently, you still have to specify each section's `level` manually:

```js
export default function Page() {
return (
<Section level={1}>
...
<Section level={2}>
...
<Section level={3}>
...
```
```

Since context lets you read information from a component above, each `Section` could read the `level` from the `Section` above, and pass `level + 1` down automatically. Here is how you could do it:

```js Section.js {5,8}
import { useContext } from 'react';
import { LevelContext } from './LevelContext.js';

export default function Section({ children }) {
const level = useContext(LevelContext);
return (
<section className="section">
<LevelContext.Provider value={level + 1}>
{children}
</LevelContext.Provider>
</section>
);
}
```

With this change, you don't need to pass the `level` prop *either* to the `<Section>` or to the `<Heading>`:

<Sandpack>

```js
import Heading from './Heading.js';
import Section from './Section.js';

export default function Page() {
return (
<Section>
<Heading>Title</Heading>
<Section>
<Heading>Heading</Heading>
<Heading>Heading</Heading>
<Heading>Heading</Heading>
<Section>
<Heading>Sub-heading</Heading>
<Heading>Sub-heading</Heading>
<Heading>Sub-heading</Heading>
```

<Section>
<Heading>Sub-sub-heading</Heading>
<Heading>Sub-sub-heading</Heading>
<Heading>Sub-sub-heading</Heading>
</Section>
</Section>
</Section>
</Section>
);
}
```

```js Section.js
import { useContext } from 'react';
import { LevelContext } from './LevelContext.js';

export default function Section({ children }) {
const level = useContext(LevelContext);
return (
<section className="section">
<LevelContext.Provider value={level + 1}>
{children}
</LevelContext.Provider>
</section>
);
}
```

```js Heading.js
import { useContext } from 'react';
import { LevelContext } from './LevelContext.js';

export default function Heading({ children }) {
const level = useContext(LevelContext);
switch (level) {
case 0:
throw Error('Heading must be inside a Section!');
case 1:

```
return <h1>{children}</h1>;
case 2:
return <h2>{children}</h2>;
case 3:
return <h3>{children}</h3>;
case 4:
return <h4>{children}</h4>;
case 5:
return <h5>{children}</h5>;
case 6:
return <h6>{children}</h6>;
default:
throw Error('Unknown level: ' + level);
}
}
```

```js LevelContext.js
import { createContext } from 'react';

export const LevelContext = createContext(0);
```

```css
.section {
padding: 10px;
margin: 5px;
border-radius: 5px;
border: 1px solid #aaa;
}
```

</Sandpack>

Now both `Heading` and `Section` read the `LevelContext` to figure out how "deep" they are. And the `Section` wraps its children into the `LevelContext` to specify that anything inside of it is at a "deeper" level.

<Note>

This example uses heading levels because they show visually how nested components can override context. But context is useful for many other use cases too. You can pass down any information needed by the entire subtree: the current color theme, the currently logged in user, and so on.

</Note>

Context passes through intermediate components
{/*context-passes-through-intermediate-components*/}

You can insert as many components as you like between the component that provides context and the one that uses it. This includes both built-in components like `<div>` and components you might build yourself.

In this example, the same `Post` component (with a dashed border) is rendered at two different nesting levels. Notice that the `<Heading>` inside of it gets its level automatically from the closest `<Section>`:

<Sandpack>

```js
import Heading from './Heading.js';
import Section from './Section.js';

export default function ProfilePage() {
return (
<Section>
<Heading>My Profile</Heading>
<Post
title="Hello traveller!"
body="Read about my adventures."
/>
<AllPosts />
</Section>
);
}

function AllPosts() {
return (
<Section>
<Heading>Posts</Heading>
<RecentPosts />
</Section>
);
}
```

```
function RecentPosts() {
return (
<Section>
<Heading>Recent Posts</Heading>
<Post
title="Flavors of Lisbon"
body="...those pastéis de nata!"
/>
<Post
title="Buenos Aires in the rhythm of tango"
body="I loved it!"
/>
</Section>
);
}

function Post({ title, body }) {
return (
<Section isFancy={true}>
<Heading>
{title}
</Heading>
<p><i>{body}</i></p>
</Section>
);
}
```

```js Section.js
import { useContext } from 'react';
import { LevelContext } from './LevelContext.js';

export default function Section({ children, isFancy }) {
const level = useContext(LevelContext);
return (
<section className={
'section ' +
```

```
(isFancy ? 'fancy' : '')
}>
<LevelContext.Provider value={level + 1}>
{children}
</LevelContext.Provider>
</section>
);
}
```

```js Heading.js
import { useContext } from 'react';
import { LevelContext } from './LevelContext.js';

export default function Heading({ children }) {
const level = useContext(LevelContext);
switch (level) {
case 0:
throw Error('Heading must be inside a Section!');
case 1:
return <h1>{children}</h1>;
case 2:
return <h2>{children}</h2>;
case 3:
return <h3>{children}</h3>;
case 4:
return <h4>{children}</h4>;
case 5:
return <h5>{children}</h5>;
case 6:
return <h6>{children}</h6>;
default:
throw Error('Unknown level: ' + level);
}
}
```

```js LevelContext.js
import { createContext } from 'react';

export const LevelContext = createContext(0);
```

```css
.section {
padding: 10px;
margin: 5px;
border-radius: 5px;
border: 1px solid #aaa;
}

.fancy {
border: 4px dashed pink;
}
```

</Sandpack>

You didn't do anything special for this to work. A `Section` specifies the context for the tree inside it, so you can insert a `<Heading>` anywhere, and it will have the correct size. Try it in the sandbox above!

Context lets you write components that "adapt to their surroundings" and display themselves differently depending on _where_ (or, in other words, _in which context_) they are being rendered.

How context works might remind you of [CSS property inheritance.](https://developer.mozilla.org/en-US/docs/Web/CSS/inheritance) In CSS, you can specify `color: blue` for a `<div>`, and any DOM node inside of it, no matter how deep, will inherit that color unless some other DOM node in the middle overrides it with `color: green`. Similarly, in React, the only way to override some context coming from above is to wrap children into a context provider with a different value.

In CSS, different properties like `color` and `background-color` don't override each other. You can set all `<div>`'s `color` to red without impacting `background-color`. Similarly, **different React contexts don't override each other.** Each context that you make with `createContext()` is completely separate from other ones, and ties together components using and providing *that particular* context. One component may use or provide many different contexts without a problem.

Before you use context {/*before-you-use-context*/}

Context is very tempting to use! However, this also means it's too easy to overuse it. **Just because you need to pass some props several levels deep doesn't mean you should put that information into context.**

Here's a few alternatives you should consider before using context:

1. **Start by [passing props.](/learn/passing-props-to-a-component)** If your components are not trivial, it's not unusual to pass a dozen props down through a dozen components. It may feel like a slog, but it makes it very clear which components use which data! The person maintaining your code will be glad you've made the data flow explicit with props.

2. **Extract components and [pass JSX as `children`](/learn/passing-props-to-a-component#passing-jsx-as-children) to them.** If you pass some data through many layers of intermediate components that don't use that data (and only pass it further down), this often means that you forgot to extract some components along the way. For example, maybe you pass data props like `posts` to visual components that don't use them directly, like `<Layout posts={posts} />`. Instead, make `Layout` take `children` as a prop, and render `<Layout><Posts posts={posts} /></Layout>`. This reduces the number of layers between the component specifying the data and the one that needs it.

If neither of these approaches works well for you, consider context.

Use cases for context {/*use-cases-for-context*/}

* **Theming:** If your app lets the user change its appearance (e.g. dark mode), you can put a context provider at the top of your app, and use that context in components that need to adjust their visual look.

* **Current account:** Many components might need to know the currently logged in user. Putting it in context makes it convenient to read it anywhere in the tree. Some apps also let you operate multiple accounts at the same time (e.g. to leave a comment as a different user). In those cases, it can be convenient to wrap a part of the UI into a nested provider with a different current account value.

* **Routing:** Most routing solutions use context internally to hold the current route. This is how every link "knows" whether it's active or not. If you build your own router, you might want to do it too.

* **Managing state:** As your app grows, you might end up with a lot of state closer to the top of your app. Many distant components below may want to change it. It is common to [use a reducer together with context](/learn/scaling-up-with-reducer-and-context) to manage complex state and pass it down to distant components without too much hassle.

Context is not limited to static values. If you pass a different value on the next render, React will update all the components reading it below! This is why context is often used in combination with state.

In general, if some information is needed by distant components in different parts of the tree, it's a good indication that context will help you.

<Recap>

* Context lets a component provide some information to the entire tree below it.

* To pass context:

1. Create and export it with `export const MyContext = createContext(defaultValue)`.

2. Pass it to the `useContext(MyContext)` Hook to read it in any child component, no matter how deep.

3. Wrap children into `<MyContext.Provider value={...}>` to provide it from a parent.

* Context passes through any components in the middle.

* Context lets you write components that "adapt to their surroundings".

* Before you use context, try passing props or passing JSX as `children`.

</Recap>

<Challenges>

Replace prop drilling with context {/*replace-prop-drilling-with-context*/}

In this example, toggling the checkbox changes the `imageSize` prop passed to each `<PlaceImage>`. The checkbox state is held in the top-level `App` component, but each `<PlaceImage>` needs to be aware of it.

Currently, `App` passes `imageSize` to `List`, which passes it to each `Place`, which passes it to the `PlaceImage`. Remove the `imageSize` prop, and instead pass it from the `App` component directly to `PlaceImage`.

You can declare context in `Context.js`.

<Sandpack>

```js App.js
import { useState } from 'react';
import { places } from './data.js';
import { getImageUrl } from './utils.js';

export default function App() {
const [isLarge, setIsLarge] = useState(false);
const imageSize = isLarge ? 150 : 100;
return (
<>
<label>
<input
type="checkbox"
checked={isLarge}
onChange={e => {
setIsLarge(e.target.checked);
}}
/>
Use large images
</label>
<hr />
<List imageSize={imageSize} />
</>
)
```

```
}

function List({ imageSize }) {
const listItems = places.map(place =>
<li key={place.id}>
<Place
place={place}
imageSize={imageSize}
/>
</li>
);
return <ul>{listItems}</ul>;
}

function Place({ place, imageSize }) {
return (
<>
<PlaceImage
place={place}
imageSize={imageSize}
/>
<p>
<b>{place.name}</b>
{': ' + place.description}
</p>
</>
);
}

function PlaceImage({ place, imageSize }) {
return (
<img
src={getImageUrl(place)}
alt={place.name}
width={imageSize}
height={imageSize}
/>
```

);

}

```

```js Context.js

```

```js data.js
export const places = [{

id: 0,

name: 'Bo-Kaap in Cape Town, South Africa',

description: 'The tradition of choosing bright colors for houses began in the late 20th century.',

imageId: 'K9HVAGH'

}, {

id: 1,

name: 'Rainbow Village in Taichung, Taiwan',

description: 'To save the houses from demolition, Huang Yung-Fu, a local resident, painted all 1,200 of them in 1924.',

imageId: '9EAYZrt'

}, {

id: 2,

name: 'Macromural de Pachuca, Mexico',

description: 'One of the largest murals in the world covering homes in a hillside neighborhood.',

imageId: 'DgXHVwu'

}, {

id: 3,

name: 'Selarón Staircase in Rio de Janeiro, Brazil',

description: 'This landmark was created by Jorge Selarón, a Chilean-born artist, as a "tribute to the Brazilian people."',

imageId: 'aeO3rpI'

}, {

id: 4,

name: 'Burano, Italy',

description: 'The houses are painted following a specific color system dating back to 16th century.',

imageId: 'kxsph5C'

}, {

id: 5,

name: 'Chefchaouen, Marocco',

description: 'There are a few theories on why the houses are painted blue, including that the color repells mosquitos or that it symbolizes sky and heaven.',

imageId: 'rTqKo46'

}, {

id: 6,

name: 'Gamcheon Culture Village in Busan, South Korea',

description: 'In 2009, the village was converted into a cultural hub by painting the houses and featuring exhibitions and art installations.',

imageId: 'ZfQOOzf'

}];

```

```js utils.js

export function getImageUrl(place) {

return (

'https://i.imgur.com/' +

place.imageId +

'l.jpg'

);

}

```

```css

ul { list-style-type: none; padding: 0px 10px; }

li {

margin-bottom: 10px;

display: grid;

grid-template-columns: auto 1fr;

gap: 20px;

align-items: center;

}

```

</Sandpack>

<Solution>

Remove `imageSize` prop from all the components.
```

Create and export `ImageSizeContext` from `Context.js`. Then wrap the List into `<ImageSizeContext.Provider value={imageSize}>` to pass the value down, and `useContext(ImageSizeContext)` to read it in the `PlaceImage`:

<Sandpack>

```js App.js
import { useState, useContext } from 'react';
import { places } from './data.js';
import { getImageUrl } from './utils.js';
import { ImageSizeContext } from './Context.js';

export default function App() {
const [isLarge, setIsLarge] = useState(false);
const imageSize = isLarge ? 150 : 100;
return (
<ImageSizeContext.Provider
value={imageSize}
>
<label>
<input
type="checkbox"
checked={isLarge}
onChange={e => {
setIsLarge(e.target.checked);
}}
/>
Use large images
</label>
<hr />
<List />
</ImageSizeContext.Provider>
)
}

function List() {
const listItems = places.map(place =>
<li key={place.id}>
<Place place={place} />
```

```
</li>
);
return <ul>{listItems}</ul>;
}

function Place({ place }) {
return (
<>
<PlaceImage place={place} />
<p>
<b>{place.name}</b>
{': ' + place.description}
</p>
</>
);
}

function PlaceImage({ place }) {
const imageSize = useContext(ImageSizeContext);
return (
<img
src={getImageUrl(place)}
alt={place.name}
width={imageSize}
height={imageSize}
/>
);
}
```

```js Context.js
import { createContext } from 'react';

export const ImageSizeContext = createContext(500);
```

```js data.js
export const places = [{
id: 0,
```

name: 'Bo-Kaap in Cape Town, South Africa',

description: 'The tradition of choosing bright colors for houses began in the late 20th century.',

imageId: 'K9HVAGH'

}, {

id: 1,

name: 'Rainbow Village in Taichung, Taiwan',

description: 'To save the houses from demolition, Huang Yung-Fu, a local resident, painted all 1,200 of them in 1924.',

imageId: '9EAYZrt'

}, {

id: 2,

name: 'Macromural de Pachuca, Mexico',

description: 'One of the largest murals in the world covering homes in a hillside neighborhood.',

imageId: 'DgXHVwu'

}, {

id: 3,

name: 'Selarón Staircase in Rio de Janeiro, Brazil',

description: 'This landmark was created by Jorge Selarón, a Chilean-born artist, as a "tribute to the Brazilian people".',

imageId: 'aeO3rpI'

}, {

id: 4,

name: 'Burano, Italy',

description: 'The houses are painted following a specific color system dating back to 16th century.',

imageId: 'kxsph5C'

}, {

id: 5,

name: 'Chefchaouen, Marocco',

description: 'There are a few theories on why the houses are painted blue, including that the color repells mosquitos or that it symbolizes sky and heaven.',

imageId: 'rTqKo46'

}, {

id: 6,

name: 'Gamcheon Culture Village in Busan, South Korea',

description: 'In 2009, the village was converted into a cultural hub by painting the houses and featuring exhibitions and art installations.',

imageId: 'ZfQOOzf'

}];
```

```js utils.js
export function getImageUrl(place) {
return (
'https://i.imgur.com/' +
place.imageId +
'l.jpg'
);
}
```

```css
ul { list-style-type: none; padding: 0px 10px; }
li {
margin-bottom: 10px;
display: grid;
grid-template-columns: auto 1fr;
gap: 20px;
align-items: center;
}
```

</Sandpack>

Note how components in the middle don't need to pass `imageSize` anymore.

</Solution>

</Challenges>

---

title: Installation

<Intro>

React has been designed from the start for gradual adoption. 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 an HTML page, or start a complex React-powered app, this section will help you get started.

</Intro>
```

<YouWillLearn isChapter={true}>

* [How to start a new React project](/learn/start-a-new-react-project)
* [How to add React to an existing project](/learn/add-react-to-an-existing-project)
* [How to set up your editor](/learn/editor-setup)
* [How to install React Developer Tools](/learn/react-developer-tools)

</YouWillLearn>

Try React {/*try-react*/}

You don't need to install anything to play with React. Try editing this sandbox!

<Sandpack>

```js
function Greeting({ name }) {
return <h1>Hello, {name}</h1>;
}

export default function App() {
return <Greeting name="world" />
}
```

</Sandpack>

You can edit it directly or open it in a new tab by pressing the "Fork" button in the upper right corner.

Most pages in the React documentation contain sandboxes like this. Outside of the React documentation, there are many online sandboxes that support React: for example, [CodeSandbox](https://codesandbox.io/s/new), [StackBlitz](https://stackblitz.com/fork/react), or [CodePen.](https://codepen.io/pen?&editors=0010&layout=left&prefill_data_id=3f4569d1-1b11-4bce-bd46-89090eed5ddb)

Try React locally {/*try-react-locally*/}

To try React locally on your computer, [download this HTML page.](https://gist.githubusercontent.com/gaearon/0275b1e1518599bbeafcde4722e79ed1/raw/db72dcbf3384ee1708c4a07d3be79860db04bff0/example.html) Open it in your editor and in your browser!

Start a new React project {/*start-a-new-react-project*/}

If you want to build an app or a website fully with React, [start a new React project.](/learn/start-a-new-react-project)

Add React to an existing project {/*add-react-to-an-existing-project*/}

If want to try using React in your existing app or a website, [add React to an existing project.](/learn/add-react-to-an-existing-project)

Next steps {/*next-steps*/}

Head to the [Quick Start](/learn) guide for a tour of the most important React concepts you will encounter every day.

---

title: Start a New React Project

<Intro>

If you want to build a new app or a new website fully with React, we recommend picking one of the React-powered frameworks popular in the community. Frameworks provide features that most apps and sites eventually need, including routing, data fetching, and generating HTML.

</Intro>

<Note>

You need to install [Node.js](https://nodejs.org/en/) for local development. You can *also* choose to use Node.js in production, but you don't have to. Many React frameworks support export to a static HTML/CSS/JS folder.

</Note>

Production-grade React frameworks {/*production-grade-react-frameworks*/}

Next.js {/*nextjs*/}

[Next.js](https://nextjs.org/) is a full-stack React framework. It's versatile and lets you create React apps of any size--from a mostly static blog to a complex dynamic application. To create a new Next.js project, run in your terminal:

<TerminalBlock>

npx create-next-app

</TerminalBlock>

If you're new to Next.js, check out the [Next.js tutorial.](https://nextjs.org/learn/foundations/about-nextjs)

Next.js is maintained by [Vercel](https://vercel.com/). You can [deploy a Next.js app](https://nextjs.org/docs/deployment) to any Node.js or serverless hosting, or to your own server. [Fully static Next.js apps](https://nextjs.org/docs/advanced-features/static-html-export) can be deployed to any static hosting.

Remix {/*remix*/}

[Remix](https://remix.run/) is a full-stack React framework with nested routing. It lets you break your app into nested parts that can load data in parallel and refresh in response to the user actions. To

create a new Remix project, run:

<TerminalBlock>

npx create-remix

</TerminalBlock>

If you're new to Remix, check out the Remix [blog tutorial](https://remix.run/docs/en/main/tutorials/blog) (short) and [app tutorial](https://remix.run/docs/en/main/tutorials/jokes) (long).

Remix is maintained by [Shopify](https://www.shopify.com/). When you create a Remix project, you need to [pick your deployment target](https://remix.run/docs/en/main/guides/deployment). You can deploy a Remix app to any Node.js or serverless hosting by using or writing an [adapter](https://remix.run/docs/en/main/other-api/adapter).

Gatsby {/*gatsby*/}

[Gatsby](https://www.gatsbyjs.com/) is a React framework for fast CMS-backed websites. Its rich plugin ecosystem and its GraphQL data layer simplify integrating content, APIs, and services into one website. To create a new Gatsby project, run:

<TerminalBlock>

npx create-gatsby

</TerminalBlock>

If you're new to Gatsby, check out the [Gatsby tutorial.](https://www.gatsbyjs.com/docs/tutorial/)

Gatsby is maintained by [Netlify](https://www.netlify.com/). You can [deploy a fully static Gatsby site](https://www.gatsbyjs.com/docs/how-to/previews-deploys-hosting) to any static hosting. If you opt into using server-only features, make sure your hosting provider supports them for Gatsby.

Expo (for native apps) {/*expo*/}

[Expo](https://expo.dev/) is a React framework that lets you create universal Android, iOS, and web apps with truly native UIs. It provides an SDK for [React Native](https://reactnative.dev/) that makes the native parts easier to use. To create a new Expo project, run:

<TerminalBlock>

npx create-expo-app

</TerminalBlock>

If you're new to Expo, check out the [Expo tutorial](https://docs.expo.dev/tutorial/introduction/).

Expo is maintained by [Expo (the company)](https://expo.dev/about). Building apps with Expo is free, and you can submit them to the Google and Apple app stores without restrictions. Expo additionally provides opt-in paid cloud services.

<DeepDive>

Can I use React without a framework? {/*can-i-use-react-without-a-framework*/}

You can definitely use React without a framework--that's how you'd [use React for a part of your page.](/learn/add-react-to-an-existing-project#using-react-for-a-part-of-your-existing-page) **However, if you're building a new app or a site fully with React, we recommend using a framework.**

Here's why.

Even if you don't need routing or data fetching at first, you'll likely want to add some libraries for them. As your JavaScript bundle grows with every new feature, you might have to figure out how to split code for every route individually. As your data fetching needs get more complex, you are likely to encounter server-client network waterfalls that make your app feel very slow. As your audience includes more users with poor network conditions and low-end devices, you might need to generate HTML from your components to display content early--either on the server, or during the build time. Changing your setup to run some of your code on the server or during the build can be very tricky.

These problems are not React-specific. This is why Svelte has SvelteKit, Vue has Nuxt, and so on. To solve these problems on your own, you'll need to integrate your bundler with your router and with your data fetching library. It's not hard to get an initial setup working, but there are a lot of subtleties involved in making an app that loads quickly even as it grows over time. You'll want to send down the minimal amount of app code but do so in a single client–server roundtrip, in parallel with any data required for the page. You'll likely want the page to be interactive before your JavaScript code even runs, to support progressive enhancement. You may want to generate a folder of fully static HTML files for your marketing pages that can be hosted anywhere and still work with JavaScript disabled. Building these capabilities yourself takes real work.

React frameworks on this page solve problems like these by default, with no extra work from your side. They let you start very lean and then scale your app with your needs. Each React framework has a community, so finding answers to questions and upgrading tooling is easier. Frameworks also give structure to your code, helping you and others retain context and skills between different projects. Conversely, with a custom setup it's easier to get stuck on unsupported dependency versions, and you'll essentially end up creating your own framework—albeit one with no community or upgrade path (and if it's anything like the ones we've made in the past, more haphazardly designed).

If you're still not convinced, or your app has unusual constraints not served well by these frameworks and you'd like to roll your own custom setup, we can't stop you--go for it! Grab `react` and `react-dom` from npm, set up your custom build process with a bundler like [Vite](https://vitejs.dev/) or [Parcel](https://parceljs.org/), and add other tools as you need them for routing, static generation or server-side rendering, and more.

</DeepDive>

Bleeding-edge React frameworks {/*bleeding-edge-react-frameworks*/}

As we've explored how to continue improving React, we realized that integrating React more closely with frameworks (specifically, with routing, bundling, and server technologies) is our biggest opportunity to help React users build better apps. The Next.js team has agreed to collaborate with us in researching, developing, integrating, and testing framework-agnostic bleeding-edge React features like [React Server Components.](/blog/2023/03/22/react-labs-what-we-have-been-working-on-march-2023#react-server-components)

These features are getting closer to being production-ready every day, and we've been in talks with other bundler and framework developers about integrating them. Our hope is that in a year or two, all frameworks listed on this page will have full support for these features. (If you're a framework author interested in partnering with us to experiment with these features, please let us know!)

Next.js (App Router) {/*nextjs-app-router*/}

[Next.js's App Router](https://beta.nextjs.org/docs/getting-started) is a redesign of the Next.js APIs aiming to fulfill the React team's full-stack architecture vision. It lets you fetch data in asynchronous components that run on the server or even during the build.

Next.js is maintained by [Vercel](https://vercel.com/). You can [deploy a Next.js app](https://nextjs.org/docs/deployment) to any Node.js or serverless hosting, or to your own server. Next.js also supports [static export](https://beta.nextjs.org/docs/configuring/static-export) which doesn't require a server.

<Pitfall>

Next.js's App Router is **currently in beta and is not yet recommended for production** (as of Mar 2023). To experiment with it in an existing Next.js project, [follow this incremental migration guide](https://beta.nextjs.org/docs/upgrade-guide#migrating-from-pages-to-app).

</Pitfall>

<DeepDive>

Which features make up the React team's full-stack architecture vision? {/*which-features-make-up-the-react-teams-full-stack-architecture-vision*/}

Next.js's App Router bundler fully implements the official [React Server Components specification](https://github.com/reactjs/rfcs/blob/main/text/0188-server-components.md). This lets you mix build-time, server-only, and interactive components in a single React tree.

For example, you can write a server-only React component as an `async` function that reads from a database or from a file. Then you can pass data down from it to your interactive components:

```js
// This component runs *only* on the server (or during the build).
async function Talks({ confId }) {
// 1. You're on the server, so you can talk to your data layer. API endpoint not required.
const talks = await db.Talks.findAll({ confId });

// 2. Add any amount of rendering logic. It won't make your JavaScript bundle larger.
const videos = talks.map(talk => talk.video);

// 3. Pass the data down to the components that will run in the browser.
return <SearchableVideoList videos={videos} />;
}
```

Next.js's App Router also integrates [data fetching with Suspense](/blog/2022/03/29/react-v18#suspense-in-data-frameworks). This lets you specify a loading state (like a skeleton placeholder) for different parts of your user interface directly in your React tree:

```js
<Suspense fallback={<TalksLoading />}>
<Talks confId={conf.id} />
</Suspense>
```

Server Components and Suspense are React features rather than Next.js features. However, adopting them at the framework level requires buy-in and non-trivial implementation work. At the moment, the Next.js App Router is the most complete implementation. The React team is working with bundler developers to make these features easier to implement in the next generation of frameworks.

</DeepDive>

---

title: 'Tutorial: Tic-Tac-Toe'

<Intro>

You will build a small tic-tac-toe game during this tutorial. This tutorial does not assume any existing React knowledge. The techniques you'll learn in the tutorial are fundamental to building any React app, and fully understanding it will give you a deep understanding of React.

</Intro>

<Note>

This tutorial is designed for people who prefer to **learn by doing** and want to quickly try making something tangible. If you prefer learning each concept step by step, start with [Describing the UI.](/learn/describing-the-ui)

</Note>

The tutorial is divided into several sections:

- [Setup for the tutorial](#setup-for-the-tutorial) will give you **a starting point** to follow the tutorial.
- [Overview](#overview) will teach you **the fundamentals** of React: components, props, and state.
- [Completing the game](#completing-the-game) will teach you **the most common techniques** in React development.
- [Adding time travel](#adding-time-travel) will give you **a deeper insight** into the unique strengths of React.

What are you building? {/*what-are-you-building*/}

In this tutorial, you'll build an interactive tic-tac-toe game with React.

You can see what it will look like when you're finished here:

<Sandpack>

```js App.js
import { useState } from 'react';

function Square({ value, onSquareClick }) {
return (
<button className="square" onClick={onSquareClick}>
{value}
</button>
);
}

function Board({ xIsNext, squares, onPlay }) {
function handleClick(i) {
if (calculateWinner(squares) || squares[i]) {
return;
}
const nextSquares = squares.slice();
if (xIsNext) {
nextSquares[i] = 'X';
} else {
nextSquares[i] = 'O';
}
onPlay(nextSquares);
}

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

return (
<>
<div className="status">{status}</div>
<div className="board-row">
<Square value={squares[0]} onSquareClick={() => handleClick(0)} />
```

```
<Square value={squares[1]} onSquareClick={() => handleClick(1)} />
<Square value={squares[2]} onSquareClick={() => handleClick(2)} />
</div>
<div className="board-row">
<Square value={squares[3]} onSquareClick={() => handleClick(3)} />
<Square value={squares[4]} onSquareClick={() => handleClick(4)} />
<Square value={squares[5]} onSquareClick={() => handleClick(5)} />
</div>
<div className="board-row">
<Square value={squares[6]} onSquareClick={() => handleClick(6)} />
<Square value={squares[7]} onSquareClick={() => handleClick(7)} />
<Square value={squares[8]} onSquareClick={() => handleClick(8)} />
</div>
</>
);
}

export default function Game() {
const [history, setHistory] = useState([Array(9).fill(null)]);
const [currentMove, setCurrentMove] = useState(0);
const xIsNext = currentMove % 2 === 0;
const currentSquares = history[currentMove];

function handlePlay(nextSquares) {
const nextHistory = [...history.slice(0, currentMove + 1), nextSquares];
setHistory(nextHistory);
setCurrentMove(nextHistory.length - 1);
}

function jumpTo(nextMove) {
setCurrentMove(nextMove);
}

const moves = history.map((squares, move) => {
let description;
if (move > 0) {
description = 'Go to move #' + move;
} else {
```

```
description = 'Go to game start';
}
return (
<li key={move}>
<button onClick={() => jumpTo(move)}>{description}</button>
</li>
);
});

return (
<div className="game">
<div className="game-board">
<Board xIsNext={xIsNext} squares={currentSquares} onPlay={handlePlay} />
</div>
<div className="game-info">
<ol>{moves}</ol>
</div>
</div>
);
}

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;
}
```

```css styles.css
* {
box-sizing: border-box;
}

body {
font-family: sans-serif;
margin: 20px;
padding: 0;
}

.square {
background: #fff;
border: 1px solid #999;
float: left;
font-size: 24px;
font-weight: bold;
line-height: 34px;
height: 34px;
margin-right: -1px;
margin-top: -1px;
padding: 0;
text-align: center;
width: 34px;
}

.board-row:after {
clear: both;
content: '';
display: table;
}

.status {
margin-bottom: 10px;
```

```
}
.game {
display: flex;
flex-direction: row;
}
.game-info {
margin-left: 20px;
}
```

</Sandpack>

If the code doesn't make sense to you yet, 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 above 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 it is updated as the game progresses.

Once you've played around with the finished tic-tac-toe game, keep scrolling. You'll start with a simpler template in this tutorial. Our next step is to set you up so that you can start building the game.

Setup for the tutorial {/*setup-for-the-tutorial*/}

In the live code editor below, click **Fork** in the top-right corner to open the editor in a new tab using the website CodeSandbox. CodeSandbox lets you write code in your browser and preview how your users will see the app you've created. The new tab should display an empty square and the starter code for this tutorial.

<Sandpack>

```js App.js
export default function Square() {
return <button className="square">X</button>;
}
```

```css styles.css
* {
box-sizing: border-box;
}
body {
```

```
font-family: sans-serif;
margin: 20px;
padding: 0;
}

.square {
background: #fff;
border: 1px solid #999;
float: left;
font-size: 24px;
font-weight: bold;
line-height: 34px;
height: 34px;
margin-right: -1px;
margin-top: -1px;
padding: 0;
text-align: center;
width: 34px;
}

.board-row:after {
clear: both;
content: '';
display: table;
}

.status {
margin-bottom: 10px;
}
.game {
display: flex;
flex-direction: row;
}

.game-info {
margin-left: 20px;
}
```

</Sandpack>

<Note>

You can also follow this tutorial using your local development environment. To do this, you need to:

1. Install [Node.js](https://nodejs.org/en/)
1. In the CodeSandbox tab you opened earlier, press the top-left corner button to open the menu, and then choose **File > Export to ZIP** in that menu to download an archive of the files locally
1. Unzip the archive, then open a terminal and `cd` to the directory you unzipped
1. Install the dependencies with `npm install`
1. Run `npm start` to start a local server and follow the prompts to view the code running in a browser

If you get stuck, don't let this stop you! Follow along online instead and try a local setup again later.

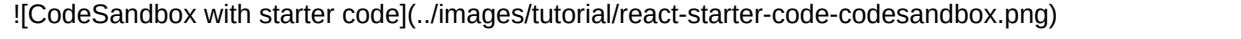
</Note>

Overview {/*overview*/}

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

Inspecting the starter code {/*inspecting-the-starter-code*/}

In CodeSandbox you'll see three main sections:

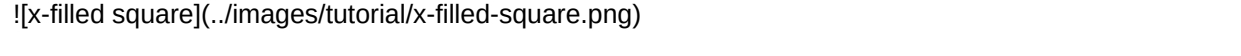


1. The _Files_ section with a list of files like `App.js`, `index.js`, `styles.css` and a folder called `public`
1. The _code editor_ where you'll see the source code of your selected file
1. The _browser_ section where you'll see how the code you've written will be displayed

The `App.js` file should be selected in the _Files_ section. The contents of that file in the _code editor_ should be:

```jsx
export default function Square() {
return <button className="square">X</button>;
}
```

The _browser_ section should be displaying a square with a X in it like this:



Now let's have a look at the files in the starter code.

`App.js` {/*appjs*/}

The code in `App.js` creates a _component_. In React, a component is a piece of reusable code that represents a part of a user interface. Components are used to render, manage, and update the UI elements in your application. Let's look at the component line by line to see what's going on:

```js {1}
export default function Square() {
return <button className="square">X</button>;
}
```

The first line defines a function called `Square`. The `export` JavaScript keyword makes this function accessible outside of this file. The `default` keyword tells other files using your code that it's the main function in your file.

```js {2}
export default function Square() {
return <button className="square">X</button>;
}
```

The second line returns a button. The `return` JavaScript keyword means whatever comes after is returned as a value to the caller of the function. `<button>` is a *JSX element*. A JSX element is a combination of JavaScript code and HTML tags that describes what you'd like to display. `className="square"` is a button property or *prop* that tells CSS how to style the button. `X` is the text displayed inside of the button and `</button>` closes the JSX element to indicate that any following content shouldn't be placed inside the button.

`styles.css` {/*stylescss*/}

Click on the file labeled `styles.css` in the _Files_ section of CodeSandbox. This file defines the styles for your React app. The first two _CSS selectors_ (`*` and `body`) define the style of large parts of your app while the `.square` selector defines the style of any component where the `className` property is set to `square`. In your code, that would match the button from your Square component in the `App.js` file.

`index.js` {/*indexjs*/}

Click on the file labeled `index.js` in the _Files_ section of CodeSandbox. You won't be editing this file during the tutorial but it is the bridge between the component you created in the `App.js` file and the web browser.

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

import App from './App';
```

```

Lines 1-5 brings all the necessary pieces together:

* React

* React's library to talk to web browsers (React DOM)

* the styles for your components

* the component you created in `App.js`.

The remainder of the file brings all the pieces together and injects the final product into `index.html` in the `public` folder.

### Building the board {/*building-the-board*/}

Let's get back to `App.js`. This is where you'll spend the rest of the tutorial.

Currently the board is only a single square, but you need nine! If you just try and copy paste your square to make two squares like this:

```js {2}
export default function Square() {
return <button className="square">X</button><button className="square">X</button>;
}
```

You'll get this error:

<ConsoleBlock level="error">

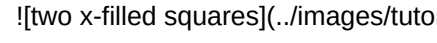
/src/App.js: Adjacent JSX elements must be wrapped in an enclosing tag. Did you want a JSX fragment `<>...</>`?

</ConsoleBlock>

React components need to return a single JSX element and not multiple adjacent JSX elements like two buttons. To fix this you can use *fragments* (`<>` and `</>`) to wrap multiple adjacent JSX elements like this:

```js {3-6}
export default function Square() {
return (
<>
<button className="square">X</button>
<button className="square">X</button>
</>
);
```
```

}
```

Now you should see:




Great! Now you just need to copy-paste a few times to add nine squares and...


Oh no! The squares are all in a single line, not in a grid like you need for our board. To fix this you'll need to group your squares into rows with `div`s and add some CSS classes. While you're at it, you'll give each square a number to make sure you know where each square is displayed.

In the `App.js` file, update the `Square` component to look like this:

```js {3-19}
export default function Square() {
return (
<>
<div className="board-row">
<button className="square">1</button>
<button className="square">2</button>
<button className="square">3</button>
</div>
<div className="board-row">
<button className="square">4</button>
<button className="square">5</button>
<button className="square">6</button>
</div>
<div className="board-row">
<button className="square">7</button>
<button className="square">8</button>
<button className="square">9</button>
</div>
</>
);
}
```
```

The CSS defined in `styles.css` styles the divs with the `className` of `board-row`. Now that you've grouped your components into rows with the styled `div`s you have your tic-tac-toe board:

But you now have a problem. Your component named `Square`, really isn't a square anymore. Let's fix that by changing the name to `Board`:

```js {1}
export default function Board() {
//...
}
```

At this point your code should look something like this:

<Sandpack>

```js
export default function Board() {
return (
<>
<div className="board-row">
<button className="square">1</button>
<button className="square">2</button>
<button className="square">3</button>
</div>
<div className="board-row">
<button className="square">4</button>
<button className="square">5</button>
<button className="square">6</button>
</div>
<div className="board-row">
<button className="square">7</button>
<button className="square">8</button>
<button className="square">9</button>
</div>
</>
);
}
```

```

```css styles.css
* {
box-sizing: border-box;
}

body {
font-family: sans-serif;
margin: 20px;
padding: 0;
}

.square {
background: #fff;
border: 1px solid #999;
float: left;
font-size: 24px;
font-weight: bold;
line-height: 34px;
height: 34px;
margin-right: -1px;
margin-top: -1px;
padding: 0;
text-align: center;
width: 34px;
}

.board-row:after {
clear: both;
content: '';
display: table;
}

.status {
margin-bottom: 10px;
}
.game {
display: flex;

flex-direction: row;

}

.game-info {

margin-left: 20px;

}

```

</Sandpack>

<Note>

Pssst... That's a lot to type! It's okay to copy and paste code from this page. However, if you're up for a little challenge, we recommend only copying code that you've manually typed at least once yourself.

</Note>

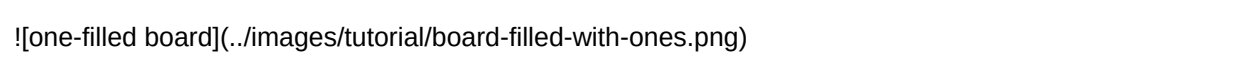
Passing data through props {/*passing-data-through-props*/}

Next, you'll want to change the value of a square from empty to "X" when the user clicks on the square. With how you've built the board so far you would need to copy-paste the code that updates the square nine times (once for each square you have)! Instead of copy-pasting, React's component architecture allows you to create a reusable component to avoid messy, duplicated code.

First, you are going to copy the line defining your first square (`<button className="square">1</button>`) from your `Board` component into a new `Square` component:

```js {1-3}

function Square() {

return <button className="square">1</button>;

}

export default function Board() {

// ...

}

```

Then you'll update the Board component to render that `Square` component using JSX syntax:

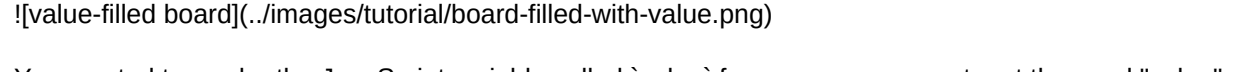
```js {5-19}

// ...

export default function Board() {

return (

<>

```
<Square />
<Square />
<Square />
</div>
<div className="board-row">
<Square />
<Square />
<Square />
</div>
<div className="board-row">
<Square />
<Square />
<Square />
</div>
</>
);
}
```

Note how unlike the browser `div`s, your own components `Board` and `Square` must start with a capital letter.

Let's take a look:



Oh no! You lost the numbered squares you had before. Now each square says "1". To fix this, you will use *props* to pass the value each square should have from the parent component (`Board`) to its child (`Square`).

Update the `Square` component to read the `value` prop that you'll pass from the `Board`:

```
```js {1}
function Square({ value }) {
return <button className="square">1</button>;
}
```
```

`function Square({ value })` indicates the Square component can be passed a prop called `value`.

Now you want to display that `value` instead of `1` inside every square. Try doing it like this:

```js {2}
function Square({ value }) {
return <button className="square">value</button>;
}
```

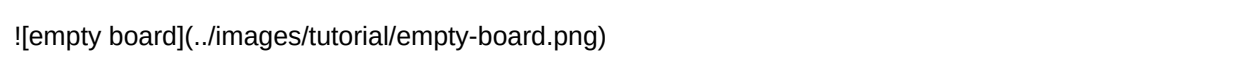
Oops, this is not what you wanted:



You wanted to render the JavaScript variable called `value` from your component, not the word "value". To "escape into JavaScript" from JSX, you need curly braces. Add curly braces around `value` in JSX like so:

```js {2}
function Square({ value }) {
return <button className="square">{value}</button>;
}
```

For now, you should see an empty board:



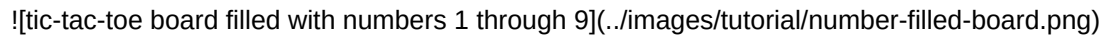
This is because the `Board` component hasn't passed the `value` prop to each `Square` component it renders yet. To fix it you'll add the `value` prop to each `Square` component rendered by the `Board` component:

```js {5-7,10-12,15-17}
export default function Board() {
return (
<>
<div className="board-row">
<Square value="1" />
<Square value="2" />
<Square value="3" />
</div>
<div className="board-row">
<Square value="4" />
<Square value="5" />
<Square value="6" />
</div>
```

<Square value="7" />
<Square value="8" />
<Square value="9" />

</>
);
}
```

Now you should see a grid of numbers again:




Your updated code should look like this:

<Sandpack>

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

export default function Board() {
return (
<>

<Square value="1" />
<Square value="2" />
<Square value="3" />

<Square value="4" />
<Square value="5" />
<Square value="6" />

<Square value="7" />
<Square value="8" />

```
<Square value="9" />
</div>
</>
);
}
```

```css styles.css
* {
box-sizing: border-box;
}

body {
font-family: sans-serif;
margin: 20px;
padding: 0;
}

.square {
background: #fff;
border: 1px solid #999;
float: left;
font-size: 24px;
font-weight: bold;
line-height: 34px;
height: 34px;
margin-right: -1px;
margin-top: -1px;
padding: 0;
text-align: center;
width: 34px;
}

.board-row:after {
clear: both;
content: '';
display: table;
}
```

```
.status {
margin-bottom: 10px;
}
.game {
display: flex;
flex-direction: row;
}
.game-info {
margin-left: 20px;
}
```

</Sandpack>

Making an interactive component {/*making-an-interactive-component*/}

Let's fill the `Square` component with an `X` when you click it. Declare a function called `handleClick` inside of the `Square`. Then, add `onClick` to the props of the button JSX element returned from the `Square`:

```js {2-4,9}
function Square({ value }) {
function handleClick() {
console.log('clicked!');
}
return (
<button
className="square"
onClick={handleClick}
>
{value}
</button>
);
}
```

If you click on a square now, you should see a log saying `"clicked!"` in the _Console_ tab at the bottom of the _Browser_ section in CodeSandbox. Clicking the square more than once will log `"clicked!"` again. Repeated console logs with the same message will not create more lines in the console. Instead, you will see an incrementing counter next to your first `"clicked!"` log.

<Note>

If you are following this tutorial using your local development environment, you need to open your browser's Console. For example, if you use the Chrome browser, you can view the Console with the keyboard shortcut **Shift + Ctrl + J** (on Windows/Linux) or **Option + ■ + J** (on macOS).

</Note>

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

React provides a special function called `useState` that you can call from your component to let it "remember" things. Let's store the current value of the `Square` in state, and change it when the `Square` is clicked.

Import `useState` at the top of the file. Remove the `value` prop from the `Square` component. Instead, add a new line at the start of the `Square` that calls `useState`. Have it return a state variable called `value`:

```js {1,3,4}
import { useState } from 'react';

function Square() {
const [value, setValue] = useState(null);

function handleClick() {
//...
```

`value` stores the value and `setValue` is a function that can be used to change the value. The `null` passed to `useState` is used as the initial value for this state variable, so `value` here starts off equal to `null`.

Since the `Square` component no longer accepts props anymore, you'll remove the `value` prop from all nine of the Square components created by the Board component:

```js {6-8,11-13,16-18}
// ...
export default function Board() {
return (
<>
<div className="board-row">
<Square />
<Square />
<Square />
</div>
```

```
<div className="board-row">
<Square />
<Square />
<Square />
</div>
<div className="board-row">
<Square />
<Square />
<Square />
</div>
</>
);
}
```

Now you'll change `Square` to display an "X" when clicked. Replace the `console.log("clicked!");` event handler with `setValue('X');`. Now your `Square` component looks like this:

```js {5}
function Square() {
const [value, setValue] = useState(null);

function handleClick() {
setValue('X');
}

return (
<button
className="square"
onClick={handleClick}
>
{value}
</button>
);
}
```

By calling this `set` function from an `onClick` handler, you're telling React to re-render that `Square` whenever its `<button>` is clicked. After the update, the `Square`'s `value` will be `'X'`, so you'll see the "X" on the game board. Click on any Square, and "X" should show up:

![adding xes to board](../images/tutorial/tictac-adding-x-s.gif)

Each Square has its own state: the `value` stored in each Square is completely independent of the others. When you call a `set` function in a component, React automatically updates the child components inside too.

After you've made the above changes, your code will look like this:

<Sandpack>

```js App.js
import { useState } from 'react';

function Square() {
const [value, setValue] = useState(null);

function handleClick() {
setValue('X');
}

return (
<button
className="square"
onClick={handleClick}
>
{value}
</button>
);
}

export default function Board() {
return (
<>
<div className="board-row">
<Square />
<Square />
<Square />
</div>
<div className="board-row">
<Square />
<Square />
```

```
<Square />
</div>
<div className="board-row">
<Square />
<Square />
<Square />
</div>
</>
);
}
```

```css styles.css
* {
box-sizing: border-box;
}

body {
font-family: sans-serif;
margin: 20px;
padding: 0;
}

.square {
background: #fff;
border: 1px solid #999;
float: left;
font-size: 24px;
font-weight: bold;
line-height: 34px;
height: 34px;
margin-right: -1px;
margin-top: -1px;
padding: 0;
text-align: center;
width: 34px;
}
```

```
.board-row:after {
clear: both;
content: '';
display: table;
}

.status {
margin-bottom: 10px;
}
.game {
display: flex;
flex-direction: row;
}

.game-info {
margin-left: 20px;
}
```

</Sandpack>

React Developer Tools {/*react-developer-tools*/}

React DevTools let you check the props and the state of your React components. You can find the React DevTools tab at the bottom of the _browser_ section in CodeSandbox:

![React DevTools in CodeSandbox](../images/tutorial/codesandbox-devtools.png)

To inspect a particular component on the screen, use the button in the top left corner of React DevTools:

![Selecting components on the page with React DevTools](../images/tutorial/devtools-select.gif)

<Note>

For local development, React DevTools is available as a [Chrome](https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi?hl=en), [Firefox](https://addons.mozilla.org/en-US/firefox/addon/react-devtools/), and [Edge](https://microsoftedge.microsoft.com/addons/detail/react-developer-tools/gpphkfbcpidddadnkolkpfckpihlkkil) browser extension. Install it, and the *Components* tab will appear in your browser Developer Tools for sites using React.

</Note>

Completing the game {/*completing-the-game*/}

By this point, you have all the basic building blocks for your tic-tac-toe game. To have a complete game, you now need to alternate placing "X"s and "O"s on the board, and you need a way to determine a winner.

Lifting state up {/*lifting-state-up*/}

Currently, each `Square` component maintains a part of the game's state. To check for a winner in a tic-tac-toe game, the `Board` would need to somehow know the state of each of the 9 `Square` components.

How would you approach that? At first, you might guess that the `Board` needs to "ask" each `Square` for that `Square`'s state. Although this approach is technically 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, like you did when you passed a number to each Square.

To collect data from multiple children, or to have two child components communicate with each other, declare the shared state in their parent component instead. The parent component can pass that state back down to the children via props. This keeps the child components in sync with each other and with their parent.

Lifting state into a parent component is common when React components are refactored.

Let's take this opportunity to try it out. Edit the `Board` component so that it declares a state variable named `squares` that defaults to an array of 9 nulls corresponding to the 9 squares:

```js {3}
// ...
export default function Board() {
const [squares, setSquares] = useState(Array(9).fill(null));
return (
// ...
);
}
```

`Array(9).fill(null)` creates an array with nine elements and sets each of them to `null`. The `useState()` call around it declares a `squares` state variable that's initially set to that array. Each entry in the array corresponds to the value of a square. When you fill the board in later, the `squares` array will look like this:

```jsx
['O', null, 'X', 'X', 'X', 'O', 'O', null, null]
```

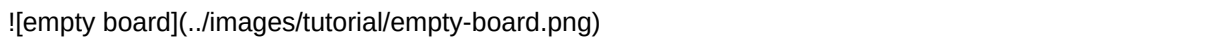
Now your `Board` component needs to pass the `value` prop down to each `Square` that it renders:

```js {6-8,11-13,16-18}
export default function Board() {
const [squares, setSquares] = useState(Array(9).fill(null));
return (
<>
<div className="board-row">
<Square value={squares[0]} />
<Square value={squares[1]} />
<Square value={squares[2]} />
</div>
<div className="board-row">
<Square value={squares[3]} />
<Square value={squares[4]} />
<Square value={squares[5]} />
</div>
<div className="board-row">
<Square value={squares[6]} />
<Square value={squares[7]} />
<Square value={squares[8]} />
</div>
</>
);
}
```

Next, you'll edit the `Square` component to receive the `value` prop from the Board component. This will require removing the Square component's own stateful tracking of `value` and the button's `onClick` prop:

```js {1,2}
function Square({value}) {
return <button className="square">{value}</button>;
}
```

At this point you should see an empty tic-tac-toe board:



And your code should look like this:

<Sandpack>

```js App.js
import { useState } from 'react';

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

export default function Board() {
const [squares, setSquares] = useState(Array(9).fill(null));
return (
<>
<div className="board-row">
<Square value={squares[0]} />
<Square value={squares[1]} />
<Square value={squares[2]} />
</div>
<div className="board-row">
<Square value={squares[3]} />
<Square value={squares[4]} />
<Square value={squares[5]} />
</div>
<div className="board-row">
<Square value={squares[6]} />
<Square value={squares[7]} />
<Square value={squares[8]} />
</div>
</>
);
}
```

```css styles.css
* {
box-sizing: border-box;
}
```

```
body {
font-family: sans-serif;
margin: 20px;
padding: 0;
}

.square {
background: #fff;
border: 1px solid #999;
float: left;
font-size: 24px;
font-weight: bold;
line-height: 34px;
height: 34px;
margin-right: -1px;
margin-top: -1px;
padding: 0;
text-align: center;
width: 34px;
}

.board-row:after {
clear: both;
content: '';
display: table;
}

.status {
margin-bottom: 10px;
}
.game {
display: flex;
flex-direction: row;
}

.game-info {
margin-left: 20px;
}
```

```

</Sandpack>

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

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

Instead, you'll pass down a function from the `Board` component to the `Square` component, and you'll have `Square` call that function when a square is clicked. You'll start with the function that the `Square` component will call when it is clicked. You'll call that function `onSquareClick`:

```js {3}
function Square({ value }) {
return (
<button className="square" onClick={onSquareClick}>
{value}
</button>
);
}
```

Next, you'll add the `onSquareClick` function to the `Square` component's props:

```js {1}
function Square({ value, onSquareClick }) {
return (
<button className="square" onClick={onSquareClick}>
{value}
</button>
);
}
```

Now you'll connect the `onSquareClick` prop to a function in the `Board` component that you'll name `handleClick`. To connect `onSquareClick` to `handleClick` you'll pass a function to the `onSquareClick` prop of the first `Square` component:

```js {7}
export default function Board() {
const [squares, setSquares] = useState(Array(9).fill(null));
```
```

```
return (
<>
<div className="board-row">
<Square value={squares[0]} onSquareClick={handleClick} />
//...
);
}
```

Lastly, you will define the `handleClick` function inside the Board component to update the `squares` array holding your board's state:

```js {4-8}
export default function Board() {
const [squares, setSquares] = useState(Array(9).fill(null));

function handleClick() {
const nextSquares = squares.slice();
nextSquares[0] = "X";
setSquares(nextSquares);
}

return (
// ...
)
}
```

The `handleClick` function creates a copy of the `squares` array (`nextSquares`) with the JavaScript `slice()` Array method. Then, `handleClick` updates the `nextSquares` array to add `X` to the first (`[0]` index) square.

Calling the `setSquares` function lets React know the state of the component has changed. This will trigger a re-render of the components that use the `squares` state (`Board`) as well as its child components (the `Square` components that make up the board).

<Note>

JavaScript supports [closures](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures) which means an inner function (e.g. `handleClick`) has access to variables and functions defined in a outer function (e.g. `Board`). The `handleClick` function can read the `squares` state and call the `setSquares` method because they are both defined inside of the `Board` function.

</Note>

Now you can add X's to the board... but only to the upper left square. Your `handleClick` function is hardcoded to update the index for the upper left square (`0`). Let's update `handleClick` to be able to update any square. Add an argument `i` to the `handleClick` function that takes the index of the square to update:

```js {4,6}
export default function Board() {
const [squares, setSquares] = useState(Array(9).fill(null));

function handleClick(i) {
const nextSquares = squares.slice();
nextSquares[i] = "X";
setSquares(nextSquares);
}

return (
// ...
)
}
```

Next, you will need to pass that `i` to `handleClick`. You could try to set the `onSquareClick` prop of square to be `handleClick(0)` directly in the JSX like this, but it won't work:

```jsx
<Square value={squares[0]} onSquareClick={handleClick(0)} />
```

Here is why this doesn't work. The `handleClick(0)` call will be a part of rendering the board component. Because `handleClick(0)` alters the state of the board component by calling `setSquares`, your entire board component will be re-rendered again. But this runs `handleClick(0)` again, leading to an infinite loop:

<ConsoleBlock level="error">

Too many re-renders. React limits the number of renders to prevent an infinite loop.

</ConsoleBlock>

Why didn't this problem happen earlier?

When you were passing `onSquareClick={handleClick}`, you were passing the `handleClick` function down as a prop. You were not calling it! But now you are *calling* that function right away--notice the parentheses in `handleClick(0)`--and that's why it runs too early. You don't *want* to call `handleClick` until the user clicks!

You could fix by creating a function like `handleFirstSquareClick` that calls `handleClick(0)`, a function like `handleSecondSquareClick` that calls `handleClick(1)`, and so on. You would pass (rather than call) these functions down as props like `onSquareClick={handleFirstSquareClick}`. This would solve the infinite loop.

However, defining nine different functions and giving each of them a name is too verbose. Instead, let's do this:

```js {6}
export default function Board() {
// ...
return (
<>
<div className="board-row">
<Square value={squares[0]} onSquareClick={() => handleClick(0)} />
// ...
);
}
```

Notice the new `() =>` syntax. Here, `() => handleClick(0)` is an *arrow function,* which is a shorter way to define functions. When the square is clicked, the code after the `=>` "arrow" will run, calling `handleClick(0)`.

Now you need to update the other eight squares to call `handleClick` from the arrow functions you pass. Make sure that the argument for each call of the `handleClick` corresponds to the index of the correct square:

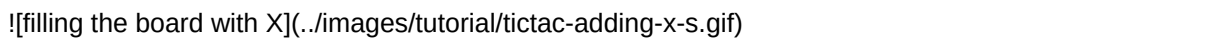
```js {6-8,11-13,16-18}
export default function Board() {
// ...
return (
<>
<div className="board-row">
<Square value={squares[0]} onSquareClick={() => handleClick(0)} />
<Square value={squares[1]} onSquareClick={() => handleClick(1)} />
<Square value={squares[2]} onSquareClick={() => handleClick(2)} />
</div>
<div className="board-row">
<Square value={squares[3]} onSquareClick={() => handleClick(3)} />
<Square value={squares[4]} onSquareClick={() => handleClick(4)} />
```

<Square value={squares[5]} onSquareClick={() => handleClick(5)} />

<Square value={squares[6]} onSquareClick={() => handleClick(6)} />
<Square value={squares[7]} onSquareClick={() => handleClick(7)} />
<Square value={squares[8]} onSquareClick={() => handleClick(8)} />

</>
);
};
```

Now you can again add X's to any square on the board by clicking on them:



But this time all the state management is handled by the `Board` component!

This is what your code should look like:

<Sandpack>

```js App.js
import { useState } from 'react';

function Square({ value, onSquareClick }) {
return (
<button className="square" onClick={onSquareClick}>
{value}
</button>
);
}

export default function Board() {
const [squares, setSquares] = useState(Array(9).fill(null));

function handleClick(i) {
const nextSquares = squares.slice();
nextSquares[i] = 'X';
setSquares(nextSquares);
}

```
return (
<>
<div className="board-row">
<Square value={squares[0]} onSquareClick={() => handleClick(0)} />
<Square value={squares[1]} onSquareClick={() => handleClick(1)} />
<Square value={squares[2]} onSquareClick={() => handleClick(2)} />
</div>
<div className="board-row">
<Square value={squares[3]} onSquareClick={() => handleClick(3)} />
<Square value={squares[4]} onSquareClick={() => handleClick(4)} />
<Square value={squares[5]} onSquareClick={() => handleClick(5)} />
</div>
<div className="board-row">
<Square value={squares[6]} onSquareClick={() => handleClick(6)} />
<Square value={squares[7]} onSquareClick={() => handleClick(7)} />
<Square value={squares[8]} onSquareClick={() => handleClick(8)} />
</div>
</>
);
}
```

```css styles.css
* {
box-sizing: border-box;
}

body {
font-family: sans-serif;
margin: 20px;
padding: 0;
}

.square {
background: #fff;
border: 1px solid #999;
float: left;
```

font-size: 24px;
font-weight: bold;
line-height: 34px;
height: 34px;
margin-right: -1px;
margin-top: -1px;
padding: 0;
text-align: center;
width: 34px;
}

.board-row:after {
clear: both;
content: '';
display: table;
}

.status {
margin-bottom: 10px;
}
.game {
display: flex;
flex-direction: row;
}

.game-info {
margin-left: 20px;
}
```

</Sandpack>

Now that your state handling is in the `Board` component, the parent `Board` component passes props to the child `Square` components so that they can be displayed correctly. When clicking on a `Square`, the child `Square` component now asks the parent `Board` component to update the state of the board. When the `Board`'s state changes, both the `Board` component and every child `Square` re-renders automatically. Keeping the state of all squares in the `Board` component will allow it to determine the winner in the future.

Let's recap what happens when a user clicks the top left square on your board to add an `X` to it:
```

1. Clicking on the upper left square runs the function that the `button` received as its `onClick` prop from the `Square`. The `Square` component received that function as its `onSquareClick` prop from the `Board`. The `Board` component defined that function directly in the JSX. It calls `handleClick` with an argument of `0`.

1. `handleClick` uses the argument (`0`) to update the first element of the `squares` array from `null` to `X`.

1. The `squares` state of the `Board` component was updated, so the `Board` and all of its children re-render. This causes the `value` prop of the `Square` component with index `0` to change from `null` to `X`.

In the end the user sees that the upper left square has changed from empty to having a `X` after clicking it.

<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. You could give any name to the `Square`'s `onSquareClick` prop or `Board`'s `handleClick` function, and the code would work the same. In React, it's conventional to use `onSomething` names for props which represent events and `handleSomething` for the function definitions which handle those events.

</Note>

Why immutability is important {/*why-immutability-is-important*/}

Note how in `handleClick`, you call `.slice()` to create a copy of the `squares` array instead of modifying the existing array. To explain why, we need to 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. Here is what it would look like if you mutated the `squares` array:

```jsx
const squares = [null, null, null, null, null, null, null, null, null];
squares[0] = 'X';
// Now `squares` is ["X", null, null, null, null, null, null, null, null];
```

And here is what it would look like if you changed data without mutating the `squares` array:

```jsx
const squares = [null, null, null, null, null, null, null, null, null];
const nextSquares = ['X', null, null, null, null, null, null, null, null];
// Now `squares` is unchanged, but `nextSquares` first element is 'X' rather than `null`
```

The result is the same but by not mutating (changing the underlying data) directly, you gain several benefits.

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

There is also another benefit of immutability. By default, all child components re-render automatically when the state of a parent component changes. This includes even the child components that weren't affected by the change. Although re-rendering is not by itself noticeable to the user (you shouldn't actively try to avoid it!), you might want to skip re-rendering a part of the tree that clearly wasn't affected by it for performance reasons. Immutability makes it very cheap for components to compare whether their data has changed or not. You can learn more about how React chooses when to re-render a component in [the `memo` API reference](/reference/react/memo).

Taking turns {/*taking-turns*/}

It's now time to fix a major defect in this tic-tac-toe game: the "O"s cannot be marked on the board.

You'll set the first move to be "X" by default. Let's keep track of this by adding another piece of state to the Board component:

```js {2}
function Board() {
const [xIsNext, setXIsNext] = useState(true);
const [squares, setSquares] = useState(Array(9).fill(null));

// ...
}
```

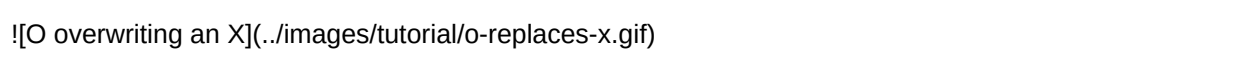
Each time a player moves, `xIsNext` (a boolean) will be flipped to determine which player goes next and the game's state will be saved. You'll update the `Board`'s `handleClick` function to flip the value of `xIsNext`:

```js {7,8,9,10,11,13}
export default function Board() {
const [xIsNext, setXIsNext] = useState(true);
const [squares, setSquares] = useState(Array(9).fill(null));

function handleClick(i) {
const nextSquares = squares.slice();
if (xIsNext) {
nextSquares[i] = "X";
} else {
```

```
nextSquares[i] = "O";
}
setSquares(nextSquares);
setXIsNext(!xIsNext);
}
return (
//...
);
}
```

Now, as you click on different squares, they will alternate between `X` and `O`, as they should!

But wait, there's a problem. Try clicking on the same square multiple times:



The `X` is overwritten by an `O`! While this would add a very interesting twist to the game, we're going to stick to the original rules for now.

When you mark a square with a `X` or an `O` you aren't first checking to see if the square already has a `X` or `O` value. You can fix this by *returning early*. You'll check to see if the square already has a `X` or an `O`. If the square is already filled, you will `return` in the `handleClick` function early--before it tries to update the board state.

```js {2,3,4}
function handleClick(i) {
if (squares[i]) {
return;
}
const nextSquares = squares.slice();
//...
}
```

Now you can only add `X`'s or `O`'s to empty squares! Here is what your code should look like at this point:

<Sandpack>

```js App.js
import { useState } from 'react';
```

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

export default function Board() {
const [xIsNext, setXIsNext] = useState(true);
const [squares, setSquares] = useState(Array(9).fill(null));

function handleClick(i) {
if (squares[i]) {
return;
}
const nextSquares = squares.slice();
if (xIsNext) {
nextSquares[i] = 'X';
} else {
nextSquares[i] = 'O';
}
setSquares(nextSquares);
setXIsNext(!xIsNext);
}

return (
<>
<div className="board-row">
<Square value={squares[0]} onSquareClick={() => handleClick(0)} />
<Square value={squares[1]} onSquareClick={() => handleClick(1)} />
<Square value={squares[2]} onSquareClick={() => handleClick(2)} />
</div>
<div className="board-row">
<Square value={squares[3]} onSquareClick={() => handleClick(3)} />
<Square value={squares[4]} onSquareClick={() => handleClick(4)} />
<Square value={squares[5]} onSquareClick={() => handleClick(5)} />
```

```
</div>
<div className="board-row">
<Square value={squares[6]} onSquareClick={() => handleClick(6)} />
<Square value={squares[7]} onSquareClick={() => handleClick(7)} />
<Square value={squares[8]} onSquareClick={() => handleClick(8)} />
</div>
</>
);
}
```

```css styles.css
* {
box-sizing: border-box;
}

body {
font-family: sans-serif;
margin: 20px;
padding: 0;
}

.square {
background: #fff;
border: 1px solid #999;
float: left;
font-size: 24px;
font-weight: bold;
line-height: 34px;
height: 34px;
margin-right: -1px;
margin-top: -1px;
padding: 0;
text-align: center;
width: 34px;
}

.board-row:after {
```

clear: both;

content: '';

display: table;

}

.status {

margin-bottom: 10px;

}

.game {

display: flex;

flex-direction: row;

}

.game-info {

margin-left: 20px;

}

```

</Sandpack>

### Declaring a winner {/*declaring-a-winner*/}

Now that the players can take turns, you'll want to show when the game is won and there are no more turns to make. To do this you'll add a helper function called `calculateWinner` that takes an array of 9 squares, checks for a winner and returns `'X'`, `'O'`, or `null` as appropriate. Don't worry too much about the `calculateWinner` function; it's not specific to React:

```js App.js

export default function Board() {

//...

}

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;
}
```

<Note>

It does not matter whether you define `calculateWinner` before or after the `Board`. Let's put it at the end so that you don't have to scroll past it every time you edit your components.

</Note>

You will call `calculateWinner(squares)` in the `Board` component's `handleClick` function to check if a player has won. You can perform this check at the same time you check if a user has clicked a square that already has a `X` or and `O`. We'd like to return early in both cases:

```js {2}
function handleClick(i) {
if (squares[i] || calculateWinner(squares)) {
return;
}
const nextSquares = squares.slice();
//...
}
```

To let the players know when the game is over, you can display text such as "Winner: X" or "Winner: O". To do that you'll add a `status` section to the `Board` component. The status will display the winner if the game is over and if the game is ongoing you'll display which player's turn is next:

```js {3-9,13}
export default function Board() {
// ...
const winner = calculateWinner(squares);
let status;
```

```
if (winner) {
status = "Winner: " + winner;
} else {
status = "Next player: " + (xIsNext ? "X" : "O");
}

return (
<>
<div className="status">{status}</div>
<div className="board-row">
// ...
)
}
```

Congratulations! You now have a working tic-tac-toe game. And you've just learned the basics of React too. So _you_ are the real winner here. Here is what the code should look like:

<Sandpack>

```js App.js
import { useState } from 'react';

function Square({value, onSquareClick}) {
return (
<button className="square" onClick={onSquareClick}>
{value}
</button>
);
}

export default function Board() {
const [xIsNext, setXIsNext] = useState(true);
const [squares, setSquares] = useState(Array(9).fill(null));

function handleClick(i) {
if (calculateWinner(squares) || squares[i]) {
return;
}
const nextSquares = squares.slice();
```

```
if (xIsNext) {
nextSquares[i] = 'X';
} else {
nextSquares[i] = 'O';
}
setSquares(nextSquares);
setXIsNext(!xIsNext);
}

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

return (
<>
<div className="status">{status}</div>
<div className="board-row">
<Square value={squares[0]} onSquareClick={() => handleClick(0)} />
<Square value={squares[1]} onSquareClick={() => handleClick(1)} />
<Square value={squares[2]} onSquareClick={() => handleClick(2)} />
</div>
<div className="board-row">
<Square value={squares[3]} onSquareClick={() => handleClick(3)} />
<Square value={squares[4]} onSquareClick={() => handleClick(4)} />
<Square value={squares[5]} onSquareClick={() => handleClick(5)} />
</div>
<div className="board-row">
<Square value={squares[6]} onSquareClick={() => handleClick(6)} />
<Square value={squares[7]} onSquareClick={() => handleClick(7)} />
<Square value={squares[8]} onSquareClick={() => handleClick(8)} />
</div>
</>
);
```

```
}

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;
}
```

```css styles.css
* {
box-sizing: border-box;
}

body {
font-family: sans-serif;
margin: 20px;
padding: 0;
}

.square {
background: #fff;
border: 1px solid #999;
float: left;
```

```
font-size: 24px;
font-weight: bold;
line-height: 34px;
height: 34px;
margin-right: -1px;
margin-top: -1px;
padding: 0;
text-align: center;
width: 34px;
}

.board-row:after {
clear: both;
content: '';
display: table;
}

.status {
margin-bottom: 10px;
}
.game {
display: flex;
flex-direction: row;
}

.game-info {
margin-left: 20px;
}
```

</Sandpack>

Adding time travel {/*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 {/*storing-a-history-of-moves*/}

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

However, you used `slice()` to create a new copy of the `squares` array after every move, and treated it as immutable. This will allow you to store every past version of the `squares` array, and navigate

between the turns that have already happened.

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

```jsx
[
// Before first move
[null, null, null, null, null, null, null, null, null],
// After first move
[null, null, null, null, 'X', null, null, null, null],
// After second move
[null, null, null, null, 'X', null, null, null, 'O'],
// ...
]
```

Lifting state up, again {/*lifting-state-up-again*/}

You will now write a new top-level component called `Game` to display a list of past moves. That's where you will place the `history` state that contains the entire game history.

Placing the `history` state into the `Game` component will let you remove the `squares` state from its child `Board` component. Just like you "lifted state up" from the `Square` component into the `Board` component, you will now lift 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, add a `Game` component with `export default`. Have it render the `Board` component and some markup:

```js {1,5-16}
function Board() {
// ...
}

export default function Game() {
return (
<div className="game">
<div className="game-board">
<Board />
</div>
```

{/*TODO*/}

);

}

```

Note that you are removing the `export default` keywords before the `function Board() {` declaration and adding them before the `function Game() {` declaration. This tells your `index.js` file to use the `Game` component as the top-level component instead of your `Board` component. The additional `div`s returned by the `Game` component are making room for the game information you'll add to the board later.

Add some state to the `Game` component to track which player is next and the history of moves:

```js {2-3}

export default function Game() {

const [xIsNext, setXIsNext] = useState(true);

const [history, setHistory] = useState([Array(9).fill(null)]);

// ...

```

Notice how `[Array(9).fill(null)]` is an array with a single item, which itself is an array of 9 `null`s.

To render the squares for the current move, you'll want to read the last squares array from the `history`. You don't need `useState` for this--you already have enough information to calculate it during rendering:

```js {4}

export default function Game() {

const [xIsNext, setXIsNext] = useState(true);

const [history, setHistory] = useState([Array(9).fill(null)]);

const currentSquares = history[history.length - 1];

// ...

```

Next, create a `handlePlay` function inside the `Game` component that will be called by the `Board` component to update the game. Pass `xIsNext`, `currentSquares` and `handlePlay` as props to the `Board` component:

```js {6-8,13}

export default function Game() {

```
const [xIsNext, setXIsNext] = useState(true);
const [history, setHistory] = useState([Array(9).fill(null)]);
const currentSquares = history[history.length - 1];

function handlePlay(nextSquares) {
// TODO
}

return (
<div className="game">
<div className="game-board">
<Board xIsNext={xIsNext} squares={currentSquares} onPlay={handlePlay} />
//...
)
}
```

Let's make the `Board` component fully controlled by the props it receives. Change the `Board` component to take three props: `xIsNext`, `squares`, and a new `onPlay` function that `Board` can call with the updated squares array when a player makes a move. Next, remove the first two lines of the `Board` function that call `useState`:

```js {1}
function Board({ xIsNext, squares, onPlay }) {
function handleClick(i) {
//...
}
// ...
}
```

Now replace the `setSquares` and `setXIsNext` calls in `handleClick` in the `Board` component with a single call to your new `onPlay` function so the `Game` component can update the `Board` when the user clicks a square:

```js {12}
function Board({ xIsNext, squares, onPlay }) {
function handleClick(i) {
if (calculateWinner(squares) || squares[i]) {
return;
}
```

```
const nextSquares = squares.slice();
if (xIsNext) {
nextSquares[i] = "X";
} else {
nextSquares[i] = "O";
}
onPlay(nextSquares);
}
//...
}
```

The `Board` component is fully controlled by the props passed to it by the `Game` component. You need to implement the `handlePlay` function in the `Game` component to get the game working again.

What should `handlePlay` do when called? Remember that Board used to call `setSquares` with an updated array; now it passes the updated `squares` array to `onPlay`.

The `handlePlay` function needs to update `Game`'s state to trigger a re-render, but you don't have a `setSquares` function that you can call any more--you're now using the `history` state variable to store this information. You'll want to update `history` by appending the updated `squares` array as a new history entry. You also want to toggle `xIsNext`, just as Board used to do:

```js {4-5}
export default function Game() {
//...
function handlePlay(nextSquares) {
setHistory([...history, nextSquares]);
setXIsNext(!xIsNext);
}
//...
}
```

Here, `[...history, nextSquares]` creates a new array that contains all the items in `history`, followed by `nextSquares`. (You can read the `...history` [*spread syntax*](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax) as "enumerate all the items in `history`".)

For example, if `history` is `[[null,null,null], ["X",null,null]]` and `nextSquares` is `["X",null,"O"]`, then the new `[...history, nextSquares]` array will be `[[null,null,null], ["X",null,null], ["X",null,"O"]]`.

At this point, you've moved the state to live in the `Game` component, and the UI should be fully working, just as it was before the refactor. Here is what the code should look like at this point:

<Sandpack>

```js App.js
import { useState } from 'react';

function Square({ value, onSquareClick }) {
return (
<button className="square" onClick={onSquareClick}>
{value}
</button>
);
}

function Board({ xIsNext, squares, onPlay }) {
function handleClick(i) {
if (calculateWinner(squares) || squares[i]) {
return;
}
const nextSquares = squares.slice();
if (xIsNext) {
nextSquares[i] = 'X';
} else {
nextSquares[i] = 'O';
}
onPlay(nextSquares);
}

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

return (
<>
<div className="status">{status}</div>
<div className="board-row">
```

```
<Square value={squares[0]} onSquareClick={() => handleClick(0)} />
<Square value={squares[1]} onSquareClick={() => handleClick(1)} />
<Square value={squares[2]} onSquareClick={() => handleClick(2)} />
</div>
<div className="board-row">
<Square value={squares[3]} onSquareClick={() => handleClick(3)} />
<Square value={squares[4]} onSquareClick={() => handleClick(4)} />
<Square value={squares[5]} onSquareClick={() => handleClick(5)} />
</div>
<div className="board-row">
<Square value={squares[6]} onSquareClick={() => handleClick(6)} />
<Square value={squares[7]} onSquareClick={() => handleClick(7)} />
<Square value={squares[8]} onSquareClick={() => handleClick(8)} />
</div>
</>
);
}

export default function Game() {
const [xIsNext, setXIsNext] = useState(true);
const [history, setHistory] = useState([Array(9).fill(null)]);
const currentSquares = history[history.length - 1];

function handlePlay(nextSquares) {
setHistory([...history, nextSquares]);
setXIsNext(!xIsNext);
}

return (
<div className="game">
<div className="game-board">
<Board xIsNext={xIsNext} squares={currentSquares} onPlay={handlePlay} />
</div>
<div className="game-info">
<ol>{/*TODO*/}</ol>
</div>
</div>
```

```
);
}

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;
}
```

```css styles.css
* {
box-sizing: border-box;
}

body {
font-family: sans-serif;
margin: 20px;
padding: 0;
}

.square {
background: #fff;
border: 1px solid #999;
```

```
float: left;
font-size: 24px;
font-weight: bold;
line-height: 34px;
height: 34px;
margin-right: -1px;
margin-top: -1px;
padding: 0;
text-align: center;
width: 34px;
}

.board-row:after {
clear: both;
content: '';
display: table;
}

.status {
margin-bottom: 10px;
}
.game {
display: flex;
flex-direction: row;
}

.game-info {
margin-left: 20px;
}
```

</Sandpack>

Showing the past moves {/*showing-the-past-moves*/}

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

React elements like `<button>` are regular JavaScript objects; you can pass them around in your application. To render multiple items in React, you can use an array of React elements.

You already have an array of `history` moves in state, so now you need to transform it to an array of React elements. In JavaScript, to transform one array into another, you can use the [array `map` meth od:](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map)

```jsx
[1, 2, 3].map((x) => x * 2) // [2, 4, 6]
```

You'll use `map` to transform your `history` of moves into 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 component:

```js {11-13,15-27,35}
export default function Game() {
const [xIsNext, setXIsNext] = useState(true);
const [history, setHistory] = useState([Array(9).fill(null)]);
const currentSquares = history[history.length - 1];

function handlePlay(nextSquares) {
setHistory([...history, nextSquares]);
setXIsNext(!xIsNext);
}

function jumpTo(nextMove) {
// TODO
}

const moves = history.map((squares, move) => {
let description;
if (move > 0) {
description = 'Go to move #' + move;
} else {
description = 'Go to game start';
}
return (
<li>
<button onClick={() => jumpTo(move)}>{description}</button>
</li>
);
});
```

```
return (
<div className="game">
<div className="game-board">
<Board xIsNext={xIsNext} squares={currentSquares} onPlay={handlePlay} />
</div>
<div className="game-info">
<ol>{moves}</ol>
</div>
</div>
);
}
```

You can see what your code should look like below. Note that you should see an error in the developer tools console that says: ``Warning: Each child in an array or iterator should have a unique "key" prop. Check the render method of `Game`.`` You'll fix this error in the next section.

<Sandpack>

```js App.js
import { useState } from 'react';

function Square({ value, onSquareClick }) {
return (
<button className="square" onClick={onSquareClick}>
{value}
</button>
);
}

function Board({ xIsNext, squares, onPlay }) {
function handleClick(i) {
if (calculateWinner(squares) || squares[i]) {
return;
}
const nextSquares = squares.slice();
if (xIsNext) {
nextSquares[i] = 'X';
} else {
```

```
nextSquares[i] = 'O';
}
onPlay(nextSquares);
}

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

return (
<>
<div className="status">{status}</div>
<div className="board-row">
<Square value={squares[0]} onSquareClick={() => handleClick(0)} />
<Square value={squares[1]} onSquareClick={() => handleClick(1)} />
<Square value={squares[2]} onSquareClick={() => handleClick(2)} />
</div>
<div className="board-row">
<Square value={squares[3]} onSquareClick={() => handleClick(3)} />
<Square value={squares[4]} onSquareClick={() => handleClick(4)} />
<Square value={squares[5]} onSquareClick={() => handleClick(5)} />
</div>
<div className="board-row">
<Square value={squares[6]} onSquareClick={() => handleClick(6)} />
<Square value={squares[7]} onSquareClick={() => handleClick(7)} />
<Square value={squares[8]} onSquareClick={() => handleClick(8)} />
</div>
</>
);
}

export default function Game() {
const [xIsNext, setXIsNext] = useState(true);
```

```
const [history, setHistory] = useState([Array(9).fill(null)]);
const currentSquares = history[history.length - 1];

function handlePlay(nextSquares) {
setHistory([...history, nextSquares]);
setXIsNext(!xIsNext);
}

function jumpTo(nextMove) {
// TODO
}

const moves = history.map((squares, move) => {
let description;
if (move > 0) {
description = 'Go to move #' + move;
} else {
description = 'Go to game start';
}
return (
<li>
<button onClick={() => jumpTo(move)}>{description}</button>
</li>
);
});

return (
<div className="game">
<div className="game-board">
<Board xIsNext={xIsNext} squares={currentSquares} onPlay={handlePlay} />
</div>
<div className="game-info">
<ol>{moves}</ol>
</div>
</div>
);
}

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;
}
```

```css styles.css
* {
box-sizing: border-box;
}

body {
font-family: sans-serif;
margin: 20px;
padding: 0;
}

.square {
background: #fff;
border: 1px solid #999;
float: left;
font-size: 24px;
font-weight: bold;
```

```
line-height: 34px;
height: 34px;
margin-right: -1px;
margin-top: -1px;
padding: 0;
text-align: center;
width: 34px;
}

.board-row:after {
clear: both;
content: '';
display: table;
}

.status {
margin-bottom: 10px;
}

.game {
display: flex;
flex-direction: row;
}

.game-info {
margin-left: 20px;
}
```

</Sandpack>

As you iterate through `history` array inside the function you passed to `map`, the `squares` argument goes through each element of `history`, and the `move` argument goes through each array index: `0`, `1`, `2`, …. (In most cases, you'd need the actual array elements, but to render a list of moves you will only need indexes.)

For each move in the tic-tac-toe game's history, you create a list item `<li>` which contains a button `<button>`. The button has an `onClick` handler which calls a function called `jumpTo` (that you haven't implemented yet).

For now, you should see a list of the moves that occurred in the game and an error in the developer tools console. Let's discuss what the "key" error means.

Picking a key {/*picking-a-key*/}

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

Imagine transitioning from

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

to

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

In addition to the updated counts, a human reading this would probably say that you swapped Alexa and Ben's ordering and inserted Claudia between Alexa and Ben. However, React is a computer program and can't know what you intended, so you need to specify a _key_ property for each list item to differentiate each list item from its siblings. If your data was from a database, Alexa, Ben, and Claudia's database IDs could be used as keys.

```js {1}
<li key={user.id}>
{user.name}: {user.taskCount} tasks left
</li>
```

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 didn't exist before, React creates a component. If the current list is missing a key that existed in the previous list, React destroys the previous component. If two keys match, the corresponding component is moved.

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.

`key` is a special and reserved property in React. 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 is passed as props, React automatically uses `key` to decide which components to update. There's no way for a component to ask what `key` its parent specified.

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 report an error 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 error but has the same problems as array indices and is not recommended in most cases.

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

Implementing time travel {/*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. Moves will never be re-ordered, deleted, or inserted in the middle, so it's safe to use the move index as a key.

In the `Game` function, you can add the key as `<li key={move}>`, and if you reload the rendered game, React's "key" error should disappear:

```js {4}
const moves = history.map((squares, move) => {
//...
return (
<li key={move}>
<button onClick={() => jumpTo(move)}>{description}</button>
</li>
);
});
```

<Sandpack>

```js App.js
import { useState } from 'react';

function Square({ value, onSquareClick }) {
return (
<button className="square" onClick={onSquareClick}>
{value}
</button>
);
}

function Board({ xIsNext, squares, onPlay }) {
```

```
function handleClick(i) {
if (calculateWinner(squares) || squares[i]) {
return;
}
const nextSquares = squares.slice();
if (xIsNext) {
nextSquares[i] = 'X';
} else {
nextSquares[i] = 'O';
}
onPlay(nextSquares);
}

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

return (
<>
<div className="status">{status}</div>
<div className="board-row">
<Square value={squares[0]} onSquareClick={() => handleClick(0)} />
<Square value={squares[1]} onSquareClick={() => handleClick(1)} />
<Square value={squares[2]} onSquareClick={() => handleClick(2)} />
</div>
<div className="board-row">
<Square value={squares[3]} onSquareClick={() => handleClick(3)} />
<Square value={squares[4]} onSquareClick={() => handleClick(4)} />
<Square value={squares[5]} onSquareClick={() => handleClick(5)} />
</div>
<div className="board-row">
<Square value={squares[6]} onSquareClick={() => handleClick(6)} />
<Square value={squares[7]} onSquareClick={() => handleClick(7)} />
```

```
<Square value={squares[8]} onSquareClick={() => handleClick(8)} />
</div>
</>
);
}

export default function Game() {
const [xIsNext, setXIsNext] = useState(true);
const [history, setHistory] = useState([Array(9).fill(null)]);
const currentSquares = history[history.length - 1];

function handlePlay(nextSquares) {
setHistory([...history, nextSquares]);
setXIsNext(!xIsNext);
}

function jumpTo(nextMove) {
// TODO
}

const moves = history.map((squares, move) => {
let description;
if (move > 0) {
description = 'Go to move #' + move;
} else {
description = 'Go to game start';
}
return (
<li key={move}>
<button onClick={() => jumpTo(move)}>{description}</button>
</li>
);
});

return (
<div className="game">
<div className="game-board">
<Board xIsNext={xIsNext} squares={currentSquares} onPlay={handlePlay} />
</div>
```

```
<div className="game-info">
<ol>{moves}</ol>
</div>
</div>
);
}

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;
}
```

```css styles.css
* {
box-sizing: border-box;
}

body {
font-family: sans-serif;
margin: 20px;
padding: 0;
```

```
}

.square {
background: #fff;
border: 1px solid #999;
float: left;
font-size: 24px;
font-weight: bold;
line-height: 34px;
height: 34px;
margin-right: -1px;
margin-top: -1px;
padding: 0;
text-align: center;
width: 34px;
}

.board-row:after {
clear: both;
content: '';
display: table;
}

.status {
margin-bottom: 10px;
}

.game {
display: flex;
flex-direction: row;
}

.game-info {
margin-left: 20px;
}
```

</Sandpack>

Before you can implement `jumpTo`, you need the `Game` component to keep track of which step the user is currently viewing. To do this, define a new state variable called `currentMove`, defaulting to `0`:

```js {4}
export default function Game() {
const [xIsNext, setXIsNext] = useState(true);
const [history, setHistory] = useState([Array(9).fill(null)]);
const [currentMove, setCurrentMove] = useState(0);
const currentSquares = history[history.length - 1];
//...
}
```

Next, update the `jumpTo` function inside `Game` to update that `currentMove`. You'll also set `xIsNext` to `true` if the number that you're changing `currentMove` to is even.

```js {4-5}
export default function Game() {
// ...
function jumpTo(nextMove) {
setCurrentMove(nextMove);
setXIsNext(nextMove % 2 === 0);
}
//...
}
```

You will now make two changes to the `Game`'s `handlePlay` function which is called when you click on a square.

- If you "go back in time" and then make a new move from that point, you only want to keep the history up to that point. Instead of adding `nextSquares` after all items (`...` spread syntax) in `history`, you'll add it after all items in `history.slice(0, currentMove + 1)` so that you're only keeping that portion of the old history.
- Each time a move is made, you need to update `currentMove` to point to the latest history entry.

```js {2-4}
function handlePlay(nextSquares) {
const nextHistory = [...history.slice(0, currentMove + 1), nextSquares];
setHistory(nextHistory);
setCurrentMove(nextHistory.length - 1);
```

setXIsNext(!xIsNext);

}
```

Finally, you will modify the `Game` component to render the currently selected move, instead of always rendering the final move:

```js {5}
export default function Game() {

const [xIsNext, setXIsNext] = useState(true);

const [history, setHistory] = useState([Array(9).fill(null)]);

const [currentMove, setCurrentMove] = useState(0);

const currentSquares = history[currentMove];

// ...

}
```

If you 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.

<Sandpack>

```js App.js
import { useState } from 'react';

function Square({value, onSquareClick}) {

return (

<button className="square" onClick={onSquareClick}>

{value}

</button>

);

}

function Board({ xIsNext, squares, onPlay }) {

function handleClick(i) {

if (calculateWinner(squares) || squares[i]) {

return;

}

const nextSquares = squares.slice();

if (xIsNext) {

```
nextSquares[i] = 'X';
} else {
nextSquares[i] = 'O';
}
onPlay(nextSquares);
}

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

return (
<>
<div className="status">{status}</div>
<div className="board-row">
<Square value={squares[0]} onSquareClick={() => handleClick(0)} />
<Square value={squares[1]} onSquareClick={() => handleClick(1)} />
<Square value={squares[2]} onSquareClick={() => handleClick(2)} />
</div>
<div className="board-row">
<Square value={squares[3]} onSquareClick={() => handleClick(3)} />
<Square value={squares[4]} onSquareClick={() => handleClick(4)} />
<Square value={squares[5]} onSquareClick={() => handleClick(5)} />
</div>
<div className="board-row">
<Square value={squares[6]} onSquareClick={() => handleClick(6)} />
<Square value={squares[7]} onSquareClick={() => handleClick(7)} />
<Square value={squares[8]} onSquareClick={() => handleClick(8)} />
</div>
</>
);
}
```

```
export default function Game() {
const [xIsNext, setXIsNext] = useState(true);
const [history, setHistory] = useState([Array(9).fill(null)]);
const [currentMove, setCurrentMove] = useState(0);
const currentSquares = history[currentMove];

function handlePlay(nextSquares) {
const nextHistory = [...history.slice(0, currentMove + 1), nextSquares];
setHistory(nextHistory);
setCurrentMove(nextHistory.length - 1);
setXIsNext(!xIsNext);
}

function jumpTo(nextMove) {
setCurrentMove(nextMove);
setXIsNext(nextMove % 2 === 0);
}

const moves = history.map((squares, move) => {
let description;
if (move > 0) {
description = 'Go to move #' + move;
} else {
description = 'Go to game start';
}
return (
<li key={move}>
<button onClick={() => jumpTo(move)}>{description}</button>
</li>
);
});

return (
<div className="game">
<div className="game-board">
<Board xIsNext={xIsNext} squares={currentSquares} onPlay={handlePlay} />
</div>
<div className="game-info">
```

{moves}

);
}

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;
}
```

```css styles.css
* {
box-sizing: border-box;
}

body {
font-family: sans-serif;
margin: 20px;
padding: 0;
}

```
.square {
background: #fff;
border: 1px solid #999;
float: left;
font-size: 24px;
font-weight: bold;
line-height: 34px;
height: 34px;
margin-right: -1px;
margin-top: -1px;
padding: 0;
text-align: center;
width: 34px;
}

.board-row:after {
clear: both;
content: '';
display: table;
}

.status {
margin-bottom: 10px;
}
.game {
display: flex;
flex-direction: row;
}

.game-info {
margin-left: 20px;
}
```

</Sandpack>

Final cleanup {/*final-cleanup*/}

If you look at the code very closely, you may notice that `xIsNext === true` when `currentMove` is even and `xIsNext === false` when `currentMove` is odd. In other words, if you know the value of

`currentMove`, then you can always figure out what `xIsNext` should be.

There's no reason for you to store both of these in state. In fact, always try to avoid redundant state. Simplifying what you store in state reduces bugs and makes your code easier to understand. Change `Game` so that it doesn't store `xIsNext` as a separate state variable and instead figures it out based on the `currentMove`:

```js {4,11,15}
export default function Game() {
const [history, setHistory] = useState([Array(9).fill(null)]);
const [currentMove, setCurrentMove] = useState(0);
const xIsNext = currentMove % 2 === 0;
const currentSquares = history[currentMove];

function handlePlay(nextSquares) {
const nextHistory = [...history.slice(0, currentMove + 1), nextSquares];
setHistory(nextHistory);
setCurrentMove(nextHistory.length - 1);
}

function jumpTo(nextMove) {
setCurrentMove(nextMove);
}
// ...
}
```

You no longer need the `xIsNext` state declaration or the calls to `setXIsNext`. Now, there's no chance for `xIsNext` to get out of sync with `currentMove`, even if you make a mistake while coding the components.

Wrapping up {/*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 of how React works.

Check out the final result here:

<Sandpack>

```js App.js
import { useState } from 'react';

function Square({ value, onSquareClick }) {
return (
<button className="square" onClick={onSquareClick}>
{value}
</button>
);
}

function Board({ xIsNext, squares, onPlay }) {
function handleClick(i) {
if (calculateWinner(squares) || squares[i]) {
return;
}
const nextSquares = squares.slice();
if (xIsNext) {
nextSquares[i] = 'X';
} else {
nextSquares[i] = 'O';
}
onPlay(nextSquares);
}

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

return (
<>
<div className="status">{status}</div>
<div className="board-row">
```

```
<Square value={squares[0]} onSquareClick={() => handleClick(0)} />
<Square value={squares[1]} onSquareClick={() => handleClick(1)} />
<Square value={squares[2]} onSquareClick={() => handleClick(2)} />
</div>
<div className="board-row">
<Square value={squares[3]} onSquareClick={() => handleClick(3)} />
<Square value={squares[4]} onSquareClick={() => handleClick(4)} />
<Square value={squares[5]} onSquareClick={() => handleClick(5)} />
</div>
<div className="board-row">
<Square value={squares[6]} onSquareClick={() => handleClick(6)} />
<Square value={squares[7]} onSquareClick={() => handleClick(7)} />
<Square value={squares[8]} onSquareClick={() => handleClick(8)} />
</div>
</>
);
}

export default function Game() {
const [history, setHistory] = useState([Array(9).fill(null)]);
const [currentMove, setCurrentMove] = useState(0);
const xIsNext = currentMove % 2 === 0;
const currentSquares = history[currentMove];

function handlePlay(nextSquares) {
const nextHistory = [...history.slice(0, currentMove + 1), nextSquares];
setHistory(nextHistory);
setCurrentMove(nextHistory.length - 1);
}

function jumpTo(nextMove) {
setCurrentMove(nextMove);
}

const moves = history.map((squares, move) => {
let description;
if (move > 0) {
description = 'Go to move #' + move;
```

```
} else {
description = 'Go to game start';
}
return (
<li key={move}>
<button onClick={() => jumpTo(move)}>{description}</button>
</li>
);
});

return (
<div className="game">
<div className="game-board">
<Board xIsNext={xIsNext} squares={currentSquares} onPlay={handlePlay} />
</div>
<div className="game-info">
<ol>{moves}</ol>
</div>
</div>
);
}

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;
}
```

```css styles.css
* {
box-sizing: border-box;
}

body {
font-family: sans-serif;
margin: 20px;
padding: 0;
}

.square {
background: #fff;
border: 1px solid #999;
float: left;
font-size: 24px;
font-weight: bold;
line-height: 34px;
height: 34px;
margin-right: -1px;
margin-top: -1px;
padding: 0;
text-align: center;
width: 34px;
}

.board-row:after {
clear: both;
content: '';
display: table;
}

.status {
```

```
margin-bottom: 10px;
}
.game {
display: flex;
flex-direction: row;
}
.game-info {
margin-left: 20px;
}
```

</Sandpack>

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, listed in order of increasing difficulty:

1. For the current move only, show "You are at move #..." instead of a button.
1. Rewrite `Board` to use two loops to make the squares instead of hardcoding them.
1. Add a toggle button that lets you sort the moves in either ascending or descending order.
1. When someone wins, highlight the three squares that caused the win (and when no one wins, display a message about the result being a draw).
1. Display the location for each move in the format (row, col) in the move history list.

Throughout this tutorial, you've touched on React concepts including elements, components, props, and state. Now that you've seen how these concepts work when building a game, check out [Thinking in React](/learn/thinking-in-react) to see how the same React concepts work when build an app's UI.

---

title: Rendering Lists

<Intro>

You will often want to display multiple similar components from a collection of data. You can use the [JavaScript array methods](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Array#) to manipulate an array of data. On this page, you'll use [`filter()`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Array/filter) and [`map()`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Array/map) with React to filter and transform your array of data into an array of components.

</Intro>

<YouWillLearn>

* How to render components from an array using JavaScript's `map()`

* How to render only specific components using JavaScript's `filter()`

* When and why to use React keys

</YouWillLearn>

Rendering data from arrays {/*rendering-data-from-arrays*/}

Say that you have a list of content.

```js
<ul>
<li>Creola Katherine Johnson: mathematician</li>
<li>Mario José Molina-Pasquel Henríquez: chemist</li>
<li>Mohammad Abdus Salam: physicist</li>
<li>Percy Lavon Julian: chemist</li>
<li>Subrahmanyan Chandrasekhar: astrophysicist</li>
</ul>
```

The only difference among those list items is their contents, their data. You will often need to show several instances of the same component using different data when building interfaces: from lists of comments to galleries of profile images. In these situations, you can store that data in JavaScript objects and arrays and use methods like [`map()`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map) and [`filter()`](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/Array/filter) to render lists of components from them.

Here's a short example of how to generate a list of items from an array:

1. **Move** the data into an array:

```js
const people = [
'Creola Katherine Johnson: mathematician',
'Mario José Molina-Pasquel Henríquez: chemist',
'Mohammad Abdus Salam: physicist',
'Percy Lavon Julian: chemist',
'Subrahmanyan Chandrasekhar: astrophysicist'
];
```

2. **Map** the `people` members into a new array of JSX nodes, `listItems`:

```js
const listItems = people.map(person => <li>{person}</li>);
```

3. **Return** `listItems` from your component wrapped in a `<ul>`:

```js
return <ul>{listItems}</ul>;
```

Here is the result:

<Sandpack>

```js
const people = [
'Creola Katherine Johnson: mathematician',
'Mario José Molina-Pasquel Henríquez: chemist',
'Mohammad Abdus Salam: physicist',
'Percy Lavon Julian: chemist',
'Subrahmanyan Chandrasekhar: astrophysicist'
];

export default function List() {
const listItems = people.map(person =>
<li>{person}</li>
);
return <ul>{listItems}</ul>;
}
```

```css
li { margin-bottom: 10px; }
```

</Sandpack>

Notice the sandbox above displays a console error:

<ConsoleBlock level="error">

Warning: Each child in a list should have a unique "key" prop.

</ConsoleBlock>

You'll learn how to fix this error later on this page. Before we get to that, let's add some structure to your data.

Filtering arrays of items {/*filtering-arrays-of-items*/}

This data can be structured even more.

```js
const people = [{
id: 0,
name: 'Creola Katherine Johnson',
profession: 'mathematician',
}, {
id: 1,
name: 'Mario José Molina-Pasquel Henríquez',
profession: 'chemist',
}, {
id: 2,
name: 'Mohammad Abdus Salam',
profession: 'physicist',
}, {
name: 'Percy Lavon Julian',
profession: 'chemist',
}, {
name: 'Subrahmanyan Chandrasekhar',
profession: 'astrophysicist',
}];
```

Let's say you want a way to only show people whose profession is `'chemist'`. You can use JavaScript's `filter()` method to return just those people. This method takes an array of items, passes them through a “test” (a function that returns `true` or `false`), and returns a new array of only those items that passed the test (returned `true`).

You only want the items where `profession` is `'chemist'`. The "test" function for this looks like `(person) => person.profession === 'chemist'`. Here's how to put it together:

1. **Create** a new array of just “chemist” people, `chemists`, by calling `filter()` on the `people` filtering by `person.profession === 'chemist'`:

```js
```

```
const chemists = people.filter(person =>
person.profession === 'chemist'
);
```

2. Now **map** over `chemists`:

```js {1,13}
const listItems = chemists.map(person =>
<li>
<img
src={getImageUrl(person)}
alt={person.name}
/>
<p>
<b>{person.name}:</b>
{' ' + person.profession + ' '}
known for {person.accomplishment}
</p>
</li>
);
```

3. Lastly, **return** the `listItems` from your component:

```js
return <ul>{listItems}</ul>;
```

<Sandpack>

```js App.js
import { people } from './data.js';
import { getImageUrl } from './utils.js';

export default function List() {
const chemists = people.filter(person =>
person.profession === 'chemist'
);
const listItems = chemists.map(person =>
```

```
<li>
<img
src={getImageUrl(person)}
alt={person.name}
/>
<p>
<b>{person.name}:</b>
{' ' + person.profession + ' '}
known for {person.accomplishment}
</p>
</li>
);
return <ul>{listItems}</ul>;
}
```

```js data.js
export const people = [{
id: 0,
name: 'Creola Katherine Johnson',
profession: 'mathematician',
accomplishment: 'spaceflight calculations',
imageId: 'MK3eW3A'
}, {
id: 1,
name: 'Mario José Molina-Pasquel Henríquez',
profession: 'chemist',
accomplishment: 'discovery of Arctic ozone hole',
imageId: 'mynHUSa'
}, {
id: 2,
name: 'Mohammad Abdus Salam',
profession: 'physicist',
accomplishment: 'electromagnetism theory',
imageId: 'bE7W1ji'
}, {
```

id: 3,
name: 'Percy Lavon Julian',
profession: 'chemist',
accomplishment: 'pioneering cortisone drugs, steroids and birth control pills',
imageId: 'IOjWm71'
}, {
id: 4,
name: 'Subrahmanyan Chandrasekhar',
profession: 'astrophysicist',
accomplishment: 'white dwarf star mass calculations',
imageId: 'lrWQx8l'
}];
```

```js utils.js
export function getImageUrl(person) {
return (
'https://i.imgur.com/' +
person.imageId +
's.jpg'
);
}
```

```css
ul { list-style-type: none; padding: 0px 10px; }
li {
margin-bottom: 10px;
display: grid;
grid-template-columns: auto 1fr;
gap: 20px;
align-items: center;
}
img { width: 100px; height: 100px; border-radius: 50%; }
```

</Sandpack>
```

<Pitfall>

Arrow functions implicitly return the expression right after `=>`, so you didn't need a `return` statement:

```js
const listItems = chemists.map(person =>
<li>...</li> // Implicit return!
);
```

However, **you must write `return` explicitly if your `=>` is followed by a `{` curly brace!**

```js
const listItems = chemists.map(person => { // Curly brace
return <li>...</li>;
});
```

Arrow functions containing `=> {` are said to have a ["block body".](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions#function_body) They let you write more than a single line of code, but you *have to* write a `return` statement yourself. If you forget it, nothing gets returned!

</Pitfall>

Keeping list items in order with `key` {/*keeping-list-items-in-order-with-key*/}

Notice that all the sandboxes above show an error in the console:

<ConsoleBlock level="error">

Warning: Each child in a list should have a unique "key" prop.

</ConsoleBlock>

You need to give each array item a `key` -- a string or a number that uniquely identifies it among other items in that array:

```js
<li key={person.id}>...</li>
```

<Note>

JSX elements directly inside a `map()` call always need keys!

</Note>

Keys tell React which array item each component corresponds to, so that it can match them up later. This becomes important if your array items can move (e.g. due to sorting), get inserted, or get deleted. A well-chosen `key` helps React infer what exactly has happened, and make the correct updates to the DOM tree.

Rather than generating keys on the fly, you should include them in your data:

<Sandpack>

```js App.js
import { people } from './data.js';
import { getImageUrl } from './utils.js';

export default function List() {
const listItems = people.map(person =>
<li key={person.id}>
<img
src={getImageUrl(person)}
alt={person.name}
/>
<p>
<b>{person.name}</b>
{' ' + person.profession + ' '}
known for {person.accomplishment}
</p>
</li>
);
return <ul>{listItems}</ul>;
}
```

```js data.js active
export const people = [{
id: 0, // Used in JSX as a key
name: 'Creola Katherine Johnson',
profession: 'mathematician',
accomplishment: 'spaceflight calculations',
imageId: 'MK3eW3A'
}, {
id: 1, // Used in JSX as a key
```

```
name: 'Mario José Molina-Pasquel Henríquez',
profession: 'chemist',
accomplishment: 'discovery of Arctic ozone hole',
imageId: 'mynHUSa'
}, {
id: 2, // Used in JSX as a key
name: 'Mohammad Abdus Salam',
profession: 'physicist',
accomplishment: 'electromagnetism theory',
imageId: 'bE7W1ji'
}, {
id: 3, // Used in JSX as a key
name: 'Percy Lavon Julian',
profession: 'chemist',
accomplishment: 'pioneering cortisone drugs, steroids and birth control pills',
imageId: 'IOjWm71'
}, {
id: 4, // Used in JSX as a key
name: 'Subrahmanyan Chandrasekhar',
profession: 'astrophysicist',
accomplishment: 'white dwarf star mass calculations',
imageId: 'lrWQx8l'
}];
```

```js utils.js
export function getImageUrl(person) {
return (
'https://i.imgur.com/' +
person.imageId +
's.jpg'
);
}
```

```css
ul { list-style-type: none; padding: 0px 10px; }
```

li {

margin-bottom: 10px;

display: grid;

grid-template-columns: auto 1fr;

gap: 20px;

align-items: center;

}

img { width: 100px; height: 100px; border-radius: 50%; }

```

</Sandpack>

<DeepDive>

#### Displaying several DOM nodes for each list item
{/*displaying-several-dom-nodes-for-each-list-item*/}

What do you do when each item needs to render not one, but several DOM nodes?

The short [`<>...</>` Fragment](/reference/react/Fragment) syntax won't let you pass a key, so you need to either group them into a single `<div>`, or use the slightly longer and [more explicit `<Fragment>` syntax:](/reference/react/Fragment#rendering-a-list-of-fragments)

```js

import { Fragment } from 'react';

// ...

const listItems = people.map(person =>

<Fragment key={person.id}>

<h1>{person.name}</h1>

<p>{person.bio}</p>

</Fragment>

);

```

Fragments disappear from the DOM, so this will produce a flat list of `<h1>`, `<p>`, `<h1>`, `<p>`, and so on.

</DeepDive>

Where to get your `key` {/*where-to-get-your-key*/}

Different sources of data provide different sources of keys:
```

* **Data from a database:** If your data is coming from a database, you can use the database keys/IDs, which are unique by nature.

* **Locally generated data:** If your data is generated and persisted locally (e.g. notes in a note-taking app), use an incrementing counter, [`crypto.randomUUID()`](https://developer.mozilla.org/en-US/docs/Web/API/Crypto/randomUUID) or a package like [`uuid`](https://www.npmjs.com/package/uuid) when creating items.

Rules of keys {/*rules-of-keys*/}

* **Keys must be unique among siblings.** However, it's okay to use the same keys for JSX nodes in _different_ arrays.

* **Keys must not change** or that defeats their purpose! Don't generate them while rendering.

Why does React need keys? {/*why-does-react-need-keys*/}

Imagine that files on your desktop didn't have names. Instead, you'd refer to them by their order -- the first file, the second file, and so on. You could get used to it, but once you delete a file, it would get confusing. The second file would become the first file, the third file would be the second file, and so on.

File names in a folder and JSX keys in an array serve a similar purpose. They let us uniquely identify an item between its siblings. A well-chosen key provides more information than the position within the array. Even if the _position_ changes due to reordering, the `key` lets React identify the item throughout its lifetime.

<Pitfall>

You might be tempted to use an item's index in the array as its key. In fact, that's what React will use if you don't specify a `key` at all. But the order in which you render items will change over time if an item is inserted, deleted, or if the array gets reordered. Index as a key often leads to subtle and confusing bugs.

Similarly, do not generate keys on the fly, e.g. with `key={Math.random()}`. This will cause keys to never match up between renders, leading to all your components and DOM being recreated every time. Not only is this slow, but it will also lose any user input inside the list items. Instead, use a stable ID based on the data.

Note that your components won't receive `key` as a prop. It's only used as a hint by React itself. If your component needs an ID, you have to pass it as a separate prop: `<Profile key={id} userId={id} />`.

</Pitfall>

<Recap>

On this page you learned:

* How to move data out of components and into data structures like arrays and objects.

* How to generate sets of similar components with JavaScript's `map()`.

* How to create arrays of filtered items with JavaScript's `filter()`.

* Why and how to set `key` on each component in a collection so React can keep track of each of them even if their position or data changes.

</Recap>

<Challenges>

Splitting a list in two {/*splitting-a-list-in-two*/}

This example shows a list of all people.

Change it to show two separate lists one after another: **Chemists** and **Everyone Else.** Like previously, you can determine whether a person is a chemist by checking if `person.profession === 'chemist'`.

<Sandpack>

```js App.js
import { people } from './data.js';
import { getImageUrl } from './utils.js';

export default function List() {
const listItems = people.map(person =>
<li key={person.id}>
<img
src={getImageUrl(person)}
alt={person.name}
/>
<p>
<b>{person.name}:</b>
{' ' + person.profession + ' '}
known for {person.accomplishment}
</p>
</li>
);
return (
<article>
<h1>Scientists</h1>
<ul>{listItems}</ul>
</article>
);
}
```

```js data.js
export const people = [{
id: 0,
name: 'Creola Katherine Johnson',
profession: 'mathematician',
accomplishment: 'spaceflight calculations',
imageId: 'MK3eW3A'
}, {
id: 1,
name: 'Mario José Molina-Pasquel Henríquez',
profession: 'chemist',
accomplishment: 'discovery of Arctic ozone hole',
imageId: 'mynHUSa'
}, {
id: 2,
name: 'Mohammad Abdus Salam',
profession: 'physicist',
accomplishment: 'electromagnetism theory',
imageId: 'bE7W1ji'
}, {
id: 3,
name: 'Percy Lavon Julian',
profession: 'chemist',
accomplishment: 'pioneering cortisone drugs, steroids and birth control pills',
imageId: 'IOjWm71'
}, {
id: 4,
name: 'Subrahmanyan Chandrasekhar',
profession: 'astrophysicist',
accomplishment: 'white dwarf star mass calculations',
imageId: 'lrWQx8l'
}];
```

```js utils.js
export function getImageUrl(person) {
```

```
return (
'https://i.imgur.com/' +
person.imageId +
's.jpg'
);
}
```

```css
ul { list-style-type: none; padding: 0px 10px; }
li {
margin-bottom: 10px;
display: grid;
grid-template-columns: auto 1fr;
gap: 20px;
align-items: center;
}
img { width: 100px; height: 100px; border-radius: 50%; }
```

</Sandpack>

<Solution>

You could use `filter()` twice, creating two separate arrays, and then `map` over both of them:

<Sandpack>

```js App.js
import { people } from './data.js';
import { getImageUrl } from './utils.js';

export default function List() {
const chemists = people.filter(person =>
person.profession === 'chemist'
);
const everyoneElse = people.filter(person =>
person.profession !== 'chemist'
);
return (
```

```
<article>
<h1>Scientists</h1>
<h2>Chemists</h2>
<ul>
{chemists.map(person =>
<li key={person.id}>
<img
src={getImageUrl(person)}
alt={person.name}
/>
<p>
<b>{person.name}:</b>
{' ' + person.profession + ' '}
known for {person.accomplishment}
</p>
</li>
)}
</ul>
<h2>Everyone Else</h2>
<ul>
{everyoneElse.map(person =>
<li key={person.id}>
<img
src={getImageUrl(person)}
alt={person.name}
/>
<p>
<b>{person.name}:</b>
{' ' + person.profession + ' '}
known for {person.accomplishment}
</p>
</li>
)}
</ul>
</article>
```

);
}
```

```js data.js
export const people = [{
id: 0,
name: 'Creola Katherine Johnson',
profession: 'mathematician',
accomplishment: 'spaceflight calculations',
imageId: 'MK3eW3A'
}, {
id: 1,
name: 'Mario José Molina-Pasquel Henríquez',
profession: 'chemist',
accomplishment: 'discovery of Arctic ozone hole',
imageId: 'mynHUSa'
}, {
id: 2,
name: 'Mohammad Abdus Salam',
profession: 'physicist',
accomplishment: 'electromagnetism theory',
imageId: 'bE7W1ji'
}, {
id: 3,
name: 'Percy Lavon Julian',
profession: 'chemist',
accomplishment: 'pioneering cortisone drugs, steroids and birth control pills',
imageId: 'IOjWm71'
}, {
id: 4,
name: 'Subrahmanyan Chandrasekhar',
profession: 'astrophysicist',
accomplishment: 'white dwarf star mass calculations',
imageId: 'lrWQx8l'
}];

```

```js utils.js
export function getImageUrl(person) {
return (
'https://i.imgur.com/' +
person.imageId +
's.jpg'
);
}
```

```css
ul { list-style-type: none; padding: 0px 10px; }
li {
margin-bottom: 10px;
display: grid;
grid-template-columns: auto 1fr;
gap: 20px;
align-items: center;
}
img { width: 100px; height: 100px; border-radius: 50%; }
```

</Sandpack>

In this solution, the `map` calls are placed directly inline into the parent `<ul>` elements, but you could introduce variables for them if you find that more readable.

There is still a bit duplication between the rendered lists. You can go further and extract the repetitive parts into a `<ListSection>` component:

<Sandpack>

```js App.js
import { people } from './data.js';
import { getImageUrl } from './utils.js';

function ListSection({ title, people }) {
return (
<>

```
<h2>{title}</h2>
<ul>
{people.map(person =>
<li key={person.id}>
<img
src={getImageUrl(person)}
alt={person.name}
/>
<p>
<b>{person.name}:</b>
{' ' + person.profession + ' '}
known for {person.accomplishment}
</p>
</li>
)}
</ul>
</>
);
}

export default function List() {
const chemists = people.filter(person =>
person.profession === 'chemist'
);
const everyoneElse = people.filter(person =>
person.profession !== 'chemist'
);
return (
<article>
<h1>Scientists</h1>
<ListSection
title="Chemists"
people={chemists}
/>
<ListSection
title="Everyone Else"
```

people={everyoneElse}
/>
</article>
);
}
```

```js data.js
export const people = [{
id: 0,
name: 'Creola Katherine Johnson',
profession: 'mathematician',
accomplishment: 'spaceflight calculations',
imageId: 'MK3eW3A'
}, {
id: 1,
name: 'Mario José Molina-Pasquel Henríquez',
profession: 'chemist',
accomplishment: 'discovery of Arctic ozone hole',
imageId: 'mynHUSa'
}, {
id: 2,
name: 'Mohammad Abdus Salam',
profession: 'physicist',
accomplishment: 'electromagnetism theory',
imageId: 'bE7W1ji'
}, {
id: 3,
name: 'Percy Lavon Julian',
profession: 'chemist',
accomplishment: 'pioneering cortisone drugs, steroids and birth control pills',
imageId: 'IOjWm71'
}, {
id: 4,
name: 'Subrahmanyan Chandrasekhar',
profession: 'astrophysicist',

accomplishment: 'white dwarf star mass calculations',
imageId: 'lrWQx8l'
}];
```

```js utils.js
export function getImageUrl(person) {
return (
'https://i.imgur.com/' +
person.imageId +
's.jpg'
);
}
```

```css
ul { list-style-type: none; padding: 0px 10px; }
li {
margin-bottom: 10px;
display: grid;
grid-template-columns: auto 1fr;
gap: 20px;
align-items: center;
}
img { width: 100px; height: 100px; border-radius: 50%; }
```

</Sandpack>

A very attentive reader might notice that with two `filter` calls, we check each person's profession twice. Checking a property is very fast, so in this example it's fine. If your logic was more expensive than that, you could replace the `filter` calls with a loop that manually constructs the arrays and checks each person once.

In fact, if `people` never change, you could move this code out of your component. From React's perspective, all that matters is that you give it an array of JSX nodes in the end. It doesn't care how you produce that array:

<Sandpack>

```js App.js
import { people } from './data.js';

```
import { getImageUrl } from './utils.js';

let chemists = [];
let everyoneElse = [];
people.forEach(person => {
if (person.profession === 'chemist') {
chemists.push(person);
} else {
everyoneElse.push(person);
}
});

function ListSection({ title, people }) {
return (
<>
<h2>{title}</h2>
<ul>
{people.map(person =>
<li key={person.id}>
<img
src={getImageUrl(person)}
alt={person.name}
/>
<p>
<b>{person.name}:</b>
{' ' + person.profession + ' '}
known for {person.accomplishment}
</p>
</li>
)}
</ul>
</>
);
}

export default function List() {
return (
```

```
<article>
<h1>Scientists</h1>
<ListSection
title="Chemists"
people={chemists}
/>
<ListSection
title="Everyone Else"
people={everyoneElse}
/>
</article>
);
}
```

```js data.js
export const people = [{
id: 0,
name: 'Creola Katherine Johnson',
profession: 'mathematician',
accomplishment: 'spaceflight calculations',
imageId: 'MK3eW3A'
}, {
id: 1,
name: 'Mario José Molina-Pasquel Henríquez',
profession: 'chemist',
accomplishment: 'discovery of Arctic ozone hole',
imageId: 'mynHUSa'
}, {
id: 2,
name: 'Mohammad Abdus Salam',
profession: 'physicist',
accomplishment: 'electromagnetism theory',
imageId: 'bE7W1ji'
}, {
id: 3,
```

name: 'Percy Lavon Julian',
profession: 'chemist',
accomplishment: 'pioneering cortisone drugs, steroids and birth control pills',
imageId: 'IOjWm71'
}, {
id: 4,
name: 'Subrahmanyan Chandrasekhar',
profession: 'astrophysicist',
accomplishment: 'white dwarf star mass calculations',
imageId: 'lrWQx8l'
}];
```

```js utils.js
export function getImageUrl(person) {
return (
'https://i.imgur.com/' +
person.imageId +
's.jpg'
);
}
```

```css
ul { list-style-type: none; padding: 0px 10px; }
li {
margin-bottom: 10px;
display: grid;
grid-template-columns: auto 1fr;
gap: 20px;
align-items: center;
}
img { width: 100px; height: 100px; border-radius: 50%; }
```

</Sandpack>

</Solution>
```

Nested lists in one component {/*nested-lists-in-one-component*/}

Make a list of recipes from this array! For each recipe in the array, display its name as an `<h2>` and list its ingredients in a `<ul>`.

<Hint>

This will require nesting two different `map` calls.

</Hint>

<Sandpack>

```js App.js
import { recipes } from './data.js';

export default function RecipeList() {
return (
<div>
<h1>Recipes</h1>
</div>
);
}
```

```js data.js
export const recipes = [{
id: 'greek-salad',
name: 'Greek Salad',
ingredients: ['tomatoes', 'cucumber', 'onion', 'olives', 'feta']
}, {
id: 'hawaiian-pizza',
name: 'Hawaiian Pizza',
ingredients: ['pizza crust', 'pizza sauce', 'mozzarella', 'ham', 'pineapple']
}, {
id: 'hummus',
name: 'Hummus',
ingredients: ['chickpeas', 'olive oil', 'garlic cloves', 'lemon', 'tahini']
}];
```

</Sandpack>

<Solution>

Here is one way you could go about it:

<Sandpack>

```js App.js
import { recipes } from './data.js';

export default function RecipeList() {
return (
<div>
<h1>Recipes</h1>
{recipes.map(recipe =>
<div key={recipe.id}>
<h2>{recipe.name}</h2>
<ul>
{recipe.ingredients.map(ingredient =>
<li key={ingredient}>
{ingredient}
</li>
)}
</ul>
</div>
)}
</div>
);
}
```

```js data.js
export const recipes = [{
id: 'greek-salad',
name: 'Greek Salad',
ingredients: ['tomatoes', 'cucumber', 'onion', 'olives', 'feta']
}, {
id: 'hawaiian-pizza',
name: 'Hawaiian Pizza',
ingredients: ['pizza crust', 'pizza sauce', 'mozzarella', 'ham', 'pineapple']
```

}, {

id: 'hummus',

name: 'Hummus',

ingredients: ['chickpeas', 'olive oil', 'garlic cloves', 'lemon', 'tahini']

}];

```

</Sandpack>

Each of the `recipes` already includes an `id` field, so that's what the outer loop uses for its `key`. There is no ID you could use to loop over ingredients. However, it's reasonable to assume that the same ingredient won't be listed twice within the same recipe, so its name can serve as a `key`. Alternatively, you could change the data structure to add IDs, or use index as a `key` (with the caveat that you can't safely reorder ingredients).

</Solution>

#### Extracting a list item component {/*extracting-a-list-item-component*/}

This `RecipeList` component contains two nested `map` calls. To simplify it, extract a `Recipe` component from it which will accept `id`, `name`, and `ingredients` props. Where do you place the outer `key` and why?

<Sandpack>

```js App.js
import { recipes } from './data.js';

export default function RecipeList() {

return (

<h1>Recipes</h1>

{recipes.map(recipe =>

<h2>{recipe.name}</h2>

<ul>

{recipe.ingredients.map(ingredient =>

<li key={ingredient}>

{ingredient}

</li>

)}

</ul>

)}

);

}

```

```js data.js

export const recipes = [{

id: 'greek-salad',

name: 'Greek Salad',

ingredients: ['tomatoes', 'cucumber', 'onion', 'olives', 'feta']

}, {

id: 'hawaiian-pizza',

name: 'Hawaiian Pizza',

ingredients: ['pizza crust', 'pizza sauce', 'mozzarella', 'ham', 'pineapple']

}, {

id: 'hummus',

name: 'Hummus',

ingredients: ['chickpeas', 'olive oil', 'garlic cloves', 'lemon', 'tahini']

}];

```

</Sandpack>

<Solution>

You can copy-paste the JSX from the outer `map` into a new `Recipe` component and return that JSX. Then you can change `recipe.name` to `name`, `recipe.id` to `id`, and so on, and pass them as props to the `Recipe`:

<Sandpack>

```js

import { recipes } from './data.js';

function Recipe({ id, name, ingredients }) {

return (

<h2>{name}</h2>

<ul>

{ingredients.map(ingredient =>

```
<li key={ingredient}>
{ingredient}

)}

</div>
);
}

export default function RecipeList() {
return (
<div>
<h1>Recipes</h1>
{recipes.map(recipe =>
<Recipe {...recipe} key={recipe.id} />
)}
</div>
);
}
```

```js data.js
export const recipes = [{
id: 'greek-salad',
name: 'Greek Salad',
ingredients: ['tomatoes', 'cucumber', 'onion', 'olives', 'feta']
}, {
id: 'hawaiian-pizza',
name: 'Hawaiian Pizza',
ingredients: ['pizza crust', 'pizza sauce', 'mozzarella', 'ham', 'pineapple']
}, {
id: 'hummus',
name: 'Hummus',
ingredients: ['chickpeas', 'olive oil', 'garlic cloves', 'lemon', 'tahini']
}];
```

</Sandpack>

Here, `<Recipe {...recipe} key={recipe.id} />` is a syntax shortcut saying "pass all properties of the `recipe` object as props to the `Recipe` component". You could also write each prop explicitly: `<Recipe id={recipe.id} name={recipe.name} ingredients={recipe.ingredients} key={recipe.id} />`.

Note that the `key` is specified on the `<Recipe>` itself rather than on the root `<div>` returned from `Recipe`. This is because this `key` is needed directly within the context of the surrounding array. Previously, you had an array of `<div>`s so each of them needed a `key`, but now you have an array of `<Recipe>`s. In other words, when you extract a component, don't forget to leave the `key` outside the JSX you copy and paste.

</Solution>

List with a separator {/*list-with-a-separator*/}

This example renders a famous haiku by Katsushika Hokusai, with each line wrapped in a `<p>` tag. Your job is to insert an `<hr />` separator between each paragraph. Your resulting structure should look like this:

```js
<article>
<p>I write, erase, rewrite</p>
<hr />
<p>Erase again, and then</p>
<hr />
<p>A poppy blooms.</p>
</article>
```

A haiku only contains three lines, but your solution should work with any number of lines. Note that `<hr />` elements only appear *between* the `<p>` elements, not in the beginning or the end!

<Sandpack>

```js
const poem = {
lines: [
'I write, erase, rewrite',
'Erase again, and then',
'A poppy blooms.'
]
};

export default function Poem() {
```

```
return (
<article>
{poem.lines.map((line, index) =>
<p key={index}>
{line}
</p>
)}
</article>
);
}
```

```css
body {
text-align: center;
}
p {
font-family: Georgia, serif;
font-size: 20px;
font-style: italic;
}
hr {
margin: 0 120px 0 120px;
border: 1px dashed #45c3d8;
}
```

</Sandpack>

(This is a rare case where index as a key is acceptable because a poem's lines will never reorder.)

<Hint>

You'll either need to convert `map` to a manual loop, or use a fragment.

</Hint>

<Solution>

You can write a manual loop, inserting `<hr />` and `<p>...</p>` into the output array as you go:

<Sandpack>

```js
const poem = {
lines: [
'I write, erase, rewrite',
'Erase again, and then',
'A poppy blooms.'
]
};

export default function Poem() {
let output = [];

// Fill the output array
poem.lines.forEach((line, i) => {
output.push(
<hr key={i + '-separator'} />
);
output.push(
<p key={i + '-text'}>
{line}
</p>
);
});
// Remove the first <hr />
output.shift();

return (
<article>
{output}
</article>
);
}
```

```css
body {
text-align: center;
```

```
}
p {
font-family: Georgia, serif;
font-size: 20px;
font-style: italic;
}
hr {
margin: 0 120px 0 120px;
border: 1px dashed #45c3d8;
}
```

</Sandpack>

Using the original line index as a `key` doesn't work anymore because each separator and paragraph are now in the same array. However, you can give each of them a distinct key using a suffix, e.g. `key={i + '-text'}`.

Alternatively, you could render a collection of fragments which contain `<hr />` and `<p>...</p>`. However, the `<>...</>` shorthand syntax doesn't support passing keys, so you'd have to write `<Fragment>` explicitly:

<Sandpack>

```js
import { Fragment } from 'react';

const poem = {
lines: [
'I write, erase, rewrite',
'Erase again, and then',
'A poppy blooms.'
]
};

export default function Poem() {
return (
<article>
{poem.lines.map((line, i) =>
<Fragment key={i}>
{i > 0 && <hr />}
```

{line}

</Fragment>

)}

</article>

);

}

```

```css

body {

text-align: center;

}

p {

font-family: Georgia, serif;

font-size: 20px;

font-style: italic;

}

hr {

margin: 0 120px 0 120px;

border: 1px dashed #45c3d8;

}

```

</Sandpack>

Remember, fragments (often written as `<> </>`) let you group JSX nodes without adding extra `<div>`s!

</Solution>

</Challenges>

---

title: Keeping Components Pure

<Intro>

Some JavaScript functions are *pure.* Pure functions only perform a calculation and nothing more. By strictly only writing your components as pure functions, you can avoid an entire class of baffling bugs and unpredictable behavior as your codebase grows. To get these benefits, though, there are a few rules you must follow.
```

</Intro>

<YouWillLearn>

* What purity is and how it helps you avoid bugs
* How to keep components pure by keeping changes out of the render phase
* How to use Strict Mode to find mistakes in your components

</YouWillLearn>

Purity: Components as formulas {/*purity-components-as-formulas*/}

In computer science (and especially the world of functional programming), [a pure function](https://wikipedia.org/wiki/Pure_function) is a function with the following characteristics:

* **It minds its own business.** It does not change any objects or variables that existed before it was called.
* **Same inputs, same output.** Given the same inputs, a pure function should always return the same result.

You might already be familiar with one example of pure functions: formulas in math.

Consider this math formula: <Math><MathI>y</MathI> = 2<MathI>x</MathI></Math>.

If <Math><MathI>x</MathI> = 2</Math> then <Math><MathI>y</MathI> = 4</Math>. Always.

If <Math><MathI>x</MathI> = 3</Math> then <Math><MathI>y</MathI> = 6</Math>. Always.

If <Math><MathI>x</MathI> = 3</Math>, <MathI>y</MathI> won't sometimes be 9 or $–1$ or 2.5 depending on the time of day or the state of the stock market.

If <Math><MathI>y</MathI> = 2<MathI>x</MathI></Math> and <Math><MathI>x</MathI> = 3</Math>, <MathI>y</MathI> will _always_ be 6.

If we made this into a JavaScript function, it would look like this:

```js
function double(number) {
return 2 * number;
}
```

In the above example, `double` is a **pure function.** If you pass it `3`, it will return `6`. Always.

React is designed around this concept. **React assumes that every component you write is a pure function.** This means that React components you write must always return the same JSX given the same inputs:

<Sandpack>

```js App.js
function Recipe({ drinkers }) {
return (
<ol>
<li>Boil {drinkers} cups of water.</li>
<li>Add {drinkers} spoons of tea and {0.5 * drinkers} spoons of spice.</li>
<li>Add {0.5 * drinkers} cups of milk to boil and sugar to taste.</li>
</ol>
);
}

export default function App() {
return (
<section>
<h1>Spiced Chai Recipe</h1>
<h2>For two</h2>
<Recipe drinkers={2} />
<h2>For a gathering</h2>
<Recipe drinkers={4} />
</section>
);
}
```

</Sandpack>

When you pass `drinkers={2}` to `Recipe`, it will return JSX containing `2 cups of water`. Always.

If you pass `drinkers={4}`, it will return JSX containing `4 cups of water`. Always.

Just like a math formula.

You could think of your components as recipes: if you follow them and don't introduce new ingredients during the cooking process, you will get the same dish every time. That "dish" is the JSX that the component serves to React to [render.](/learn/render-and-commit)

<Illustration src="/images/docs/illustrations/i_puritea-recipe.png" alt="A tea recipe for x people: take x cups of water, add x spoons of tea and 0.5x spoons of spices, and 0.5x cups of milk" />

Side Effects: (un)intended consequences {/*side-effects-unintended-consequences*/}

React's rendering process must always be pure. Components should only *return* their JSX, and not *change* any objects or variables that existed before rendering—that would make them impure!

Here is a component that breaks this rule:

<Sandpack>

```js
let guest = 0;

function Cup() {
// Bad: changing a preexisting variable!
guest = guest + 1;
return <h2>Tea cup for guest #{guest}</h2>;
}

export default function TeaSet() {
return (
<>
<Cup />
<Cup />
<Cup />
</>
);
}
```

</Sandpack>

This component is reading and writing a `guest` variable declared outside of it. This means that **calling this component multiple times will produce different JSX!** And what's more, if _other_ components read `guest`, they will produce different JSX, too, depending on when they were rendered! That's not predictable.

Going back to our formula <Math><MathI>y</MathI> = 2<MathI>x</MathI></Math>, now even if <Math><MathI>x</MathI> = 2</Math>, we cannot trust that <Math><MathI>y</MathI> = 4</Math>. Our tests could fail, our users would be baffled, planes would fall out of the sky—you can see how this would lead to confusing bugs!

You can fix this component by [passing `guest` as a prop instead](/learn/passing-props-to-a-component):

<Sandpack>

```js
function Cup({ guest }) {
return <h2>Tea cup for guest #{guest}</h2>;
```

```
}

export default function TeaSet() {
return (
<>
<Cup guest={1} />
<Cup guest={2} />
<Cup guest={3} />
</>
);
}
```

</Sandpack>

Now your component is pure, as the JSX it returns only depends on the `guest` prop.

In general, you should not expect your components to be rendered in any particular order. It doesn't matter if you call <Math><MathI>y</MathI> = 2<MathI>x</MathI></Math> before or after <Math><MathI>y</MathI> = 5<MathI>x</MathI></Math>: both formulas will resolve independently of each other. In the same way, each component should only "think for itself", and not attempt to coordinate with or depend upon others during rendering. Rendering is like a school exam: each component should calculate JSX on their own!

<DeepDive>

Detecting impure calculations with StrictMode {/*detecting-impure-calculations-with-strict-mode*/}

Although you might not have used them all yet, in React there are three kinds of inputs that you can read while rendering: [props](/learn/passing-props-to-a-component), [state](/learn/state-a-components-memory), and [context.](/learn/passing-data-deeply-with-context) You should always treat these inputs as read-only.

When you want to *change* something in response to user input, you should [set state](/learn/state-a-components-memory) instead of writing to a variable. You should never change preexisting variables or objects while your component is rendering.

React offers a "Strict Mode" in which it calls each component's function twice during development. **By calling the component functions twice, Strict Mode helps find components that break these rules.**

Notice how the original example displayed "Guest #2", "Guest #4", and "Guest #6" instead of "Guest #1", "Guest #2", and "Guest #3". The original function was impure, so calling it twice broke it. But the fixed pure version works even if the function is called twice every time. **Pure functions only calculate, so calling them twice won't change anything**--just like calling `double(2)` twice doesn't change what's returned, and solving <Math><MathI>y</MathI> = 2<MathI>x</MathI></Math> twice doesn't change what <MathI>y</MathI> is. Same inputs, same outputs. Always.

Strict Mode has no effect in production, so it won't slow down the app for your users. To opt into Strict Mode, you can wrap your root component into `<React.StrictMode>`. Some frameworks do this by default.

</DeepDive>

Local mutation: Your component's little secret {/*local-mutation-your-components-little-secret*/}

In the above example, the problem was that the component changed a *preexisting* variable while rendering. This is often called a **"mutation"** to make it sound a bit scarier. Pure functions don't mutate variables outside of the function's scope or objects that were created before the call—that makes them impure!

However, **it's completely fine to change variables and objects that you've *just* created while rendering.** In this example, you create an `[]` array, assign it to a `cups` variable, and then `push` a dozen cups into it:

<Sandpack>

```js
function Cup({ guest }) {
return <h2>Tea cup for guest #{guest}</h2>;
}

export default function TeaGathering() {
let cups = [];
for (let i = 1; i <= 12; i++) {
cups.push(<Cup key={i} guest={i} />);
}
return cups;
}
```

</Sandpack>

If the `cups` variable or the `[]` array were created outside the `TeaGathering` function, this would be a huge problem! You would be changing a *preexisting* object by pushing items into that array.

However, it's fine because you've created them *during the same render*, inside `TeaGathering`. No code outside of `TeaGathering` will ever know that this happened. This is called **"local mutation"**—it's like your component's little secret.

Where you _can_ cause side effects {/*where-you-_can_-cause-side-effects*/}

While functional programming relies heavily on purity, at some point, somewhere, _something_ has to change. That's kind of the point of programming! These changes—updating the screen, starting an animation, changing the data—are called **side effects.** They're things that happen _"on the side"_, not during rendering.

In React, **side effects usually belong inside [event handlers.](/learn/responding-to-events)** Event handlers are functions that React runs when you perform some action—for example, when you click a button. Even though event handlers are defined *inside* your component, they don't run *during* rendering! **So event handlers don't need to be pure.**

If you've exhausted all other options and can't find the right event handler for your side effect, you can still attach it to your returned JSX with a [`useEffect`](/reference/react/useEffect) call in your component. This tells React to execute it later, after rendering, when side effects are allowed. **However, this approach should be your last resort.**

When possible, try to express your logic with rendering alone. You'll be surprised how far this can take you!

<DeepDive>

Why does React care about purity? {/*why-does-react-care-about-purity*/}

Writing pure functions takes some habit and discipline. But it also unlocks marvelous opportunities:

* Your components could run in a different environment—for example, on the server! Since they return the same result for the same inputs, one component can serve many user requests.

* You can improve performance by [skipping rendering](/reference/react/memo) components whose inputs have not changed. This is safe because pure functions always return the same results, so they are safe to cache.

* If some data changes in the middle of rendering a deep component tree, React can restart rendering without wasting time to finish the outdated render. Purity makes it safe to stop calculating at any time.

Every new React feature we're building takes advantage of purity. From data fetching to animations to performance, keeping components pure unlocks the power of the React paradigm.

</DeepDive>

<Recap>

* A component must be pure, meaning:

* **It minds its own business.** It should not change any objects or variables that existed before rendering.

* **Same inputs, same output.** Given the same inputs, a component should always return the same JSX.

* Rendering can happen at any time, so components should not depend on each others' rendering sequence.

* You should not mutate any of the inputs that your components use for rendering. That includes props, state, and context. To update the screen, ["set" state](/learn/state-a-components-memory) instead of mutating preexisting objects.

* Strive to express your component's logic in the JSX you return. When you need to "change things", you'll usually want to do it in an event handler. As a last resort, you can `useEffect`.

* Writing pure functions takes a bit of practice, but it unlocks the power of React's paradigm.

</Recap>

<Challenges>

Fix a broken clock {/*fix-a-broken-clock*/}

This component tries to set the `<h1>`'s CSS class to `"night"` during the time from midnight to six hours in the morning, and `"day"` at all other times. However, it doesn't work. Can you fix this component?

You can verify whether your solution works by temporarily changing the computer's timezone. When the current time is between midnight and six in the morning, the clock should have inverted colors!

<Hint>

Rendering is a *calculation*, it shouldn't try to "do" things. Can you express the same idea differently?

</Hint>

<Sandpack>

```js Clock.js active
export default function Clock({ time }) {
let hours = time.getHours();
if (hours >= 0 && hours <= 6) {
document.getElementById('time').className = 'night';
} else {
document.getElementById('time').className = 'day';
}
return (
<h1 id="time">
{time.toLocaleTimeString()}
</h1>
);
}
```

```js App.js hidden
import { useState, useEffect } from 'react';
import Clock from './Clock.js';

function useTime() {
const [time, setTime] = useState(() => new Date());
useEffect(() => {
const id = setInterval(() => {
```

````
setTime(new Date());
}, 1000);
return () => clearInterval(id);
}, []);
return time;
}

export default function App() {
const time = useTime();
return (
<Clock time={time} />
);
}
```

```css
body > * {
width: 100%;
height: 100%;
}
.day {
background: #fff;
color: #222;
}
.night {
background: #222;
color: #fff;
}
```
````

</Sandpack>

<Solution>

You can fix this component by calculating the `className` and including it in the render output:

<Sandpack>

```js Clock.js active
export default function Clock({ time }) {

```
let hours = time.getHours();
let className;
if (hours >= 0 && hours <= 6) {
className = 'night';
} else {
className = 'day';
}
return (
<h1 className={className}>
{time.toLocaleTimeString()}
</h1>
);
}
```

```js App.js hidden
import { useState, useEffect } from 'react';
import Clock from './Clock.js';

function useTime() {
const [time, setTime] = useState(() => new Date());
useEffect(() => {
const id = setInterval(() => {
setTime(new Date());
}, 1000);
return () => clearInterval(id);
}, []);
return time;
}

export default function App() {
const time = useTime();
return (
<Clock time={time} />
);
}
```

```css
body > * {
width: 100%;
height: 100%;
}
.day {
background: #fff;
color: #222;
}
.night {
background: #222;
color: #fff;
}
```

</Sandpack>

In this example, the side effect (modifying the DOM) was not necessary at all. You only needed to return JSX.

</Solution>

Fix a broken profile {/*fix-a-broken-profile*/}

Two `Profile` components are rendered side by side with different data. Press "Collapse" on the first profile, and then "Expand" it. You'll notice that both profiles now show the same person. This is a bug.

Find the cause of the bug and fix it.

<Hint>

The buggy code is in `Profile.js`. Make sure you read it all from top to bottom!

</Hint>

<Sandpack>

```js Profile.js
import Panel from './Panel.js';
import { getImageUrl } from './utils.js';

let currentPerson;

export default function Profile({ person }) {
currentPerson = person;
```

```
return (
<Panel>
<Header />
<Avatar />
</Panel>
)
}

function Header() {
return <h1>{currentPerson.name}</h1>;
}

function Avatar() {
return (
<img
className="avatar"
src={getImageUrl(currentPerson)}
alt={currentPerson.name}
width={50}
height={50}
/>
);
}
```

```js Panel.js hidden
import { useState } from 'react';

export default function Panel({ children }) {
const [open, setOpen] = useState(true);
return (
<section className="panel">
<button onClick={() => setOpen(!open)}>
{open ? 'Collapse' : 'Expand'}
</button>
{open && children}
</section>
);
```

}
```

```js App.js
import Profile from './Profile.js';

export default function App() {
return (
<>
<Profile person={{
imageId: 'lrWQx8l',
name: 'Subrahmanyan Chandrasekhar',
}} />
<Profile person={{
imageId: 'MK3eW3A',
name: 'Creola Katherine Johnson',
}} />
</>
)
}
```

```js utils.js hidden
export function getImageUrl(person, size = 's') {
return (
'https://i.imgur.com/' +
person.imageId +
size +
'.jpg'
);
}
```

```css
.avatar { margin: 5px; border-radius: 50%; }
.panel {
border: 1px solid #aaa;
border-radius: 6px;
```
```

margin-top: 20px;

padding: 10px;

width: 200px;

}

h1 { margin: 5px; font-size: 18px; }

```

</Sandpack>

<Solution>

The problem is that the `Profile` component writes to a preexisting variable called `currentPerson`, and the `Header` and `Avatar` components read from it. This makes *all three of them* impure and difficult to predict.

To fix the bug, remove the `currentPerson` variable. Instead, pass all information from `Profile` to `Header` and `Avatar` via props. You'll need to add a `person` prop to both components and pass it all the way down.

<Sandpack>

```js Profile.js active

import Panel from './Panel.js';

import { getImageUrl } from './utils.js';

export default function Profile({ person }) {

return (

<Panel>

<Header person={person} />

<Avatar person={person} />

</Panel>

)

}

function Header({ person }) {

return <h1>{person.name}</h1>;

}

function Avatar({ person }) {

return (

<img

className="avatar"

src={getImageUrl(person)}

```
alt={person.name}
width={50}
height={50}
/>
);
}
```

```js Panel.js hidden
import { useState } from 'react';

export default function Panel({ children }) {
const [open, setOpen] = useState(true);
return (
<section className="panel">
<button onClick={() => setOpen(!open)}>
{open ? 'Collapse' : 'Expand'}
</button>
{open && children}
</section>
);
}
```

```js App.js
import Profile from './Profile.js';

export default function App() {
return (
<>
<Profile person={{
imageId: 'lrWQx8l',
name: 'Subrahmanyan Chandrasekhar',
}} />
<Profile person={{
imageId: 'MK3eW3A',
name: 'Creola Katherine Johnson',
}} />
```

```
</>
);
}
```

```js utils.js hidden
export function getImageUrl(person, size = 's') {
return (
'https://i.imgur.com/' +
person.imageId +
size +
'.jpg'
);
}
```

```css
.avatar { margin: 5px; border-radius: 50%; }
.panel {
border: 1px solid #aaa;
border-radius: 6px;
margin-top: 20px;
padding: 10px;
width: 200px;
}
h1 { margin: 5px; font-size: 18px; }
```

</Sandpack>

Remember that React does not guarantee that component functions will execute in any particular order, so you can't communicate between them by setting variables. All communication must happen through props.

</Solution>

Fix a broken story tray {/*fix-a-broken-story-tray*/}

The CEO of your company is asking you to add "stories" to your online clock app, and you can't say no. You've written a `StoryTray` component that accepts a list of `stories`, followed by a "Create Story" placeholder.

You implemented the "Create Story" placeholder by pushing one more fake story at the end of the `stories` array that you receive as a prop. But for some reason, "Create Story" appears more than once. Fix the issue.

<Sandpack>

```js StoryTray.js active
export default function StoryTray({ stories }) {
stories.push({
id: 'create',
label: 'Create Story'
});

return (
<ul>
{stories.map(story => (
<li key={story.id}>
{story.label}
</li>
))}
</ul>
);
}
```

```js App.js hidden
import { useState, useEffect } from 'react';
import StoryTray from './StoryTray.js';

let initialStories = [
{id: 0, label: "Ankit's Story" },
{id: 1, label: "Taylor's Story" },
];

export default function App() {
let [stories, setStories] = useState([...initialStories])
let time = useTime();

// HACK: Prevent the memory from growing forever while you read docs.
// We're breaking our own rules here.
if (stories.length > 100) {
```

```
stories.length = 100;
}

return (
<div
style={{
width: '100%',
height: '100%',
textAlign: 'center',
}}
>
<h2>It is {time.toLocaleTimeString()} now.</h2>
<StoryTray stories={stories} />
</div>
);
}

function useTime() {
const [time, setTime] = useState(() => new Date());
useEffect(() => {
const id = setInterval(() => {
setTime(new Date());
}, 1000);
return () => clearInterval(id);
}, []);
return time;
}
```

```css
ul {
margin: 0;
list-style-type: none;
}

li {
border: 1px solid #aaa;
border-radius: 6px;
```

float: left;

margin: 5px;

margin-bottom: 20px;

padding: 5px;

width: 70px;

height: 100px;

}

```

```js sandbox.config.json hidden

{

"hardReloadOnChange": true

}

```

</Sandpack>

<Solution>

Notice how whenever the clock updates, "Create Story" is added *twice*. This serves as a hint that we have a mutation during rendering--Strict Mode calls components twice to make these issues more noticeable.

`StoryTray` function is not pure. By calling `push` on the received `stories` array (a prop!), it is mutating an object that was created *before* `StoryTray` started rendering. This makes it buggy and very difficult to predict.

The simplest fix is to not touch the array at all, and render "Create Story" separately:

<Sandpack>

```js StoryTray.js active

export default function StoryTray({ stories }) {

return (

<ul>

{stories.map(story => (

<li key={story.id}>

{story.label}

</li>

))}

<li>Create Story</li>

</ul>

```
);
}
```

```js App.js hidden
import { useState, useEffect } from 'react';
import StoryTray from './StoryTray.js';

let initialStories = [
{id: 0, label: "Ankit's Story" },
{id: 1, label: "Taylor's Story" },
];

export default function App() {
let [stories, setStories] = useState([...initialStories])
let time = useTime();

// HACK: Prevent the memory from growing forever while you read docs.
// We're breaking our own rules here.
if (stories.length > 100) {
stories.length = 100;
}

return (
<div
style={{
width: '100%',
height: '100%',
textAlign: 'center',
}}
>
<h2>It is {time.toLocaleTimeString()} now.</h2>
<StoryTray stories={stories} />
</div>
);
}

function useTime() {
const [time, setTime] = useState(() => new Date());
```

```
useEffect(() => {
const id = setInterval(() => {
setTime(new Date());
}, 1000);
return () => clearInterval(id);
}, []);
return time;
}
```

```css
ul {
margin: 0;
list-style-type: none;
}

li {
border: 1px solid #aaa;
border-radius: 6px;
float: left;
margin: 5px;
margin-bottom: 20px;
padding: 5px;
width: 70px;
height: 100px;
}
```

</Sandpack>

Alternatively, you could create a _new_ array (by copying the existing one) before you push an item into it:

<Sandpack>

```js StoryTray.js active
export default function StoryTray({ stories }) {
// Copy the array!
let storiesToDisplay = stories.slice();
```

```
// Does not affect the original array:
storiesToDisplay.push({
id: 'create',
label: 'Create Story'
});

return (
<ul>
{storiesToDisplay.map(story => (
<li key={story.id}>
{story.label}
</li>
))}
</ul>
);
}
```

```js App.js hidden
import { useState, useEffect } from 'react';
import StoryTray from './StoryTray.js';

let initialStories = [
{id: 0, label: "Ankit's Story" },
{id: 1, label: "Taylor's Story" },
];

export default function App() {
let [stories, setStories] = useState([...initialStories])
let time = useTime();

// HACK: Prevent the memory from growing forever while you read docs.
// We're breaking our own rules here.
if (stories.length > 100) {
stories.length = 100;
}

return (
<div
```

```
style={{
width: '100%',
height: '100%',
textAlign: 'center',
}}
>
<h2>It is {time.toLocaleTimeString()} now.</h2>
<StoryTray stories={stories} />
</div>
);
}

function useTime() {
const [time, setTime] = useState(() => new Date());
useEffect(() => {
const id = setInterval(() => {
setTime(new Date());
}, 1000);
return () => clearInterval(id);
}, []);
return time;
}
```

```css
ul {
margin: 0;
list-style-type: none;
}

li {
border: 1px solid #aaa;
border-radius: 6px;
float: left;
margin: 5px;
margin-bottom: 20px;
padding: 5px;
```

width: 70px;

height: 100px;

}

```

</Sandpack>

This keeps your mutation local and your rendering function pure. However, you still need to be careful: for example, if you tried to change any of the array's existing items, you'd have to clone those items too.

It is useful to remember which operations on arrays mutate them, and which don't. For example, `push`, `pop`, `reverse`, and `sort` will mutate the original array, but `slice`, `filter`, and `map` will create a new one.

</Solution>

</Challenges>

---

title: Render and Commit

---

<Intro>

Before your components are displayed on screen, they must be rendered by React. Understanding the steps in this process will help you think about how your code executes and explain its behavior.

</Intro>

<YouWillLearn>

* What rendering means in React

* When and why React renders a component

* The steps involved in displaying a component on screen

* Why rendering does not always produce a DOM update

</YouWillLearn>

Imagine that your components are cooks in the kitchen, assembling tasty dishes from ingredients. In this scenario, React is the waiter who puts in requests from customers and brings them their orders. This process of requesting and serving UI has three steps:

1. **Triggering** a render (delivering the guest's order to the kitchen)

2. **Rendering** the component (preparing the order in the kitchen)

3. **Committing** to the DOM (placing the order on the table)

<IllustrationBlock sequential>
```

<Illustration caption="Trigger" alt="React as a server in a restaurant, fetching orders from the users and delivering them to the Component Kitchen." src="/images/docs/illustrations/i_render-and-commit1.png" />

<Illustration caption="Render" alt="The Card Chef gives React a fresh Card component." src="/images/docs/illustrations/i_render-and-commit2.png" />

<Illustration caption="Commit" alt="React delivers the Card to the user at their table." src="/images/docs/illustrations/i_render-and-commit3.png" />

</IllustrationBlock>

Step 1: Trigger a render {/*step-1-trigger-a-render*/}

There are two reasons for a component to render:

1. It's the component's **initial render.**

2. The component's (or one of its ancestors') **state has been updated.**

Initial render {/*initial-render*/}

When your app starts, you need to trigger the initial render. Frameworks and sandboxes sometimes hide this code, but it's done by calling [`createRoot`](/reference/react-dom/client/createRoot) with the target DOM node, and then calling its `render` method with your component:

<Sandpack>

```js index.js active
import Image from './Image.js';
import { createRoot } from 'react-dom/client';

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

```js Image.js
export default function Image() {
return (
<img
src="https://i.imgur.com/ZF6s192.jpg"
alt="'Floralis Genérica' by Eduardo Catalano: a gigantic metallic flower sculpture with reflective petals"
/>
);
}
```

</Sandpack>

Try commenting out the `root.render()` call and see the component disappear!

Re-renders when state updates {/*re-renders-when-state-updates*/}

Once the component has been initially rendered, you can trigger further renders by updating its state with the [`set` function.](/reference/react/useState#setstate) Updating your component's state automatically queues a render. (You can imagine these as a restaurant guest ordering tea, dessert, and all sorts of things after putting in their first order, depending on the state of their thirst or hunger.)

<IllustrationBlock sequential>

<Illustration caption="State update..." alt="React as a server in a restaurant, serving a Card UI to the user, represented as a patron with a cursor for their head. They patron expresses they want a pink card, not a black one!" src="/images/docs/illustrations/i_rerender1.png" />

<Illustration caption="...triggers..." alt="React returns to the Component Kitchen and tells the Card Chef they need a pink Card." src="/images/docs/illustrations/i_rerender2.png" />

<Illustration caption="...render!" alt="The Card Chef gives React the pink Card." src="/images/docs/illustrations/i_rerender3.png" />

</IllustrationBlock>

Step 2: React renders your components {/*step-2-react-renders-your-components*/}

After you trigger a render, React calls your components to figure out what to display on screen. **"Rendering" is React calling your components.**

* **On initial render,** React will call the root component.

* **For subsequent renders,** React will call the function component whose state update triggered the render.

This process is recursive: if the updated component returns some other component, React will render _that_ component next, and if that component also returns something, it will render _that_ component next, and so on. The process will continue until there are no more nested components and React knows exactly what should be displayed on screen.

In the following example, React will call `Gallery()` and `Image()` several times:

<Sandpack>

```js Gallery.js active
export default function Gallery() {
return (
<section>
<h1>Inspiring Sculptures</h1>
<Image />
<Image />
<Image />
</section>
```

```
);
}

function Image() {
return (
<img
src="https://i.imgur.com/ZF6s192.jpg"
alt="'Floralis Genérica' by Eduardo Catalano: a gigantic metallic flower sculpture with reflective petals"
/>
);
}
```

```js index.js
import Gallery from './Gallery.js';
import { createRoot } from 'react-dom/client';

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

```css
img { margin: 0 10px 10px 0; }
```

</Sandpack>

* **During the initial render,** React will [create the DOM nodes](https://developer.mozilla.org/docs/Web/API/Document/createElement) for `<section>`, `<h1>`, and three `<img>` tags.

* **During a re-render,** React will calculate which of their properties, if any, have changed since the previous render. It won't do anything with that information until the next step, the commit phase.

<Pitfall>

Rendering must always be a [pure calculation](/learn/keeping-components-pure):

* **Same inputs, same output.** Given the same inputs, a component should always return the same JSX. (When someone orders a salad with tomatoes, they should not receive a salad with onions!)

* **It minds its own business.** It should not change any objects or variables that existed before rendering. (One order should not change anyone else's order.)

Otherwise, you can encounter confusing bugs and unpredictable behavior as your codebase grows in complexity. When developing in "Strict Mode", React calls each component's function twice, which can

help surface mistakes caused by impure functions.

</Pitfall>

<DeepDive>

Optimizing performance {/*optimizing-performance*/}

The default behavior of rendering all components nested within the updated component is not optimal for performance if the updated component is very high in the tree. If you run into a performance issue, there are several opt-in ways to solve it described in the [Performance](https://reactjs.org/docs/optimizing-performance.html) section. **Don't optimize prematurely!**

</DeepDive>

Step 3: React commits changes to the DOM {/*step-3-react-commits-changes-to-the-dom*/}

After rendering (calling) your components, React will modify the DOM.

* **For the initial render,** React will use the [`appendChild()`](https://developer.mozilla.org/docs/Web/API/Node/appendChild) DOM API to put all the DOM nodes it has created on screen.

* **For re-renders,** React will apply the minimal necessary operations (calculated while rendering!) to make the DOM match the latest rendering output.

React only changes the DOM nodes if there's a difference between renders. For example, here is a component that re-renders with different props passed from its parent every second. Notice how you can add some text into the `<input>`, updating its `value`, but the text doesn't disappear when the component re-renders:

<Sandpack>

```js Clock.js active
export default function Clock({ time }) {
return (
<>
<h1>{time}</h1>
<input />
</>
);
}
```

```js App.js hidden
import { useState, useEffect } from 'react';
import Clock from './Clock.js';
```

```
function useTime() {
const [time, setTime] = useState(() => new Date());
useEffect(() => {
const id = setInterval(() => {
setTime(new Date());
}, 1000);
return () => clearInterval(id);
}, []);
return time;
}

export default function App() {
const time = useTime();
return (
<Clock time={time.toLocaleTimeString()} />
);
}
```

</Sandpack>

This works because during this last step, React only updates the content of `<h1>` with the new `time`. It sees that the `<input>` appears in the JSX in the same place as last time, so React doesn't touch the `<input>`—or its `value`!

Epilogue: Browser paint {/*epilogue-browser-paint*/}

After rendering is done and React updated the DOM, the browser will repaint the screen. Although this process is known as "browser rendering", we'll refer to it as "painting" to avoid confusion throughout the docs.

<Illustration alt="A browser painting 'still life with card element'."
src="/images/docs/illustrations/i_browser-paint.png" />

<Recap>

* Any screen update in a React app happens in three steps:

1. Trigger

2. Render

3. Commit

* You can use Strict Mode to find mistakes in your components

* React does not touch the DOM if the rendering result is the same as last time

</Recap>

---

title: useImperativeHandle

<Intro>

`useImperativeHandle` is a React Hook that lets you customize the handle exposed as a [ref.](/learn/manipulating-the-dom-with-refs)

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

</Intro>

<InlineToc />

Reference {/*reference*/}

`useImperativeHandle(ref, createHandle, dependencies?)` {/*useimperativehandle*/}

Call `useImperativeHandle` at the top level of your component to customize the ref handle it exposes:

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

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

[See more examples below.](#usage)

Parameters {/*parameters*/}

* `ref`: The `ref` you received as the second argument from the [`forwardRef` render function.](/reference/react/forwardRef#render-function)

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

* **optional** `dependencies`: The list of all reactive values referenced inside of the `createHandle` code. Reactive values include props, state, and all the variables and functions declared directly inside your component body. If your linter is [configured for React](/learn/editor-setup#linting), it will verify that every reactive value is correctly specified as a dependency. The list of dependencies must have a constant number of items and be written inline like `[dep1, dep2, dep3]`. React will compare each dependency with its previous value using the [`Object.is`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/is) comparison. If a re-render resulted in a change to some dependency, or if you omitted this argument, your `createHandle` function will re-execute, and the newly created handle will be assigned to the ref.

Returns {/*returns*/}

`useImperativeHandle` returns `undefined`.

Usage {/*usage*/}

Exposing a custom ref handle to the parent component
{/*exposing-a-custom-ref-handle-to-the-parent-component*/}

By default, components don't expose their DOM nodes to parent components. For example, if you want the parent component of `MyInput` to [have access](/learn/manipulating-the-dom-with-refs) to the `<input>` DOM node, you have to opt in with [`forwardRef`:](/reference/react/forwardRef)

```js {4}
import { forwardRef } from 'react';

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

With the code above, [a ref to `MyInput` will receive the `<input>` DOM node.](/reference/react/forwardRef#exposing-a-dom-node-to-the-parent-component) However, you can expose a custom value instead. To customize the exposed handle, call `useImperativeHandle` at the top level of your component:

```js {4-8}
import { forwardRef, useImperativeHandle } from 'react';

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

```
};
}, []);

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

Note that in the code above, the `ref` is no longer forwarded to the `<input>`.

For example, suppose you don't want to expose the entire `<input>` DOM node, but you want to expose two of its methods: `focus` and `scrollIntoView`. To do this, keep the real browser DOM in a separate ref. Then use `useImperativeHandle` to expose a handle with only the methods that you want the parent component to call:

```
```js {7-14}
import { forwardRef, useRef, useImperativeHandle } from 'react';

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

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

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

Now, if the parent component gets a ref to `MyInput`, it will be able to call the `focus` and `scrollIntoView` methods on it. However, it will not have full access to the underlying `<input>` DOM node.

<Sandpack>

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

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

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

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

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

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

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

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

export default MyInput;
```

```css
input {
margin: 5px;
}
```

</Sandpack>

Exposing your own imperative methods {/*exposing-your-own-imperative-methods*/}

The methods you expose via an imperative handle don't have to match the DOM methods exactly. For example, this `Post` component exposes a `scrollAndFocusAddComment` method via an imperative handle. This lets the parent `Page` scroll the list of comments *and* focus the input field when you click the button:

<Sandpack>

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

export default function Page() {
const postRef = useRef(null);

function handleClick() {
postRef.current.scrollAndFocusAddComment();
}

return (
<>
<button onClick={handleClick}>
Write a comment
</button>
<Post ref={postRef} />
</>
);
}
```
```

```js Post.js
import { forwardRef, useRef, useImperativeHandle } from 'react';
import CommentList from './CommentList.js';
import AddComment from './AddComment.js';

const Post = forwardRef((props, ref) => {
const commentsRef = useRef(null);
const addCommentRef = useRef(null);

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

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

export default Post;
```

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

const CommentList = forwardRef(function CommentList(props, ref) {
const divRef = useRef(null);

useImperativeHandle(ref, () => {
return {
```

```
scrollToBottom() {
const node = divRef.current;
node.scrollTop = node.scrollHeight;
}
};
}, []);

let comments = [];
for (let i = 0; i < 50; i++) {
comments.push(<p key={i}>Comment #{i}</p>);
}

return (
<div className="CommentList" ref={divRef}>
{comments}
</div>
);
});

export default CommentList;
```

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

const AddComment = forwardRef(function AddComment(props, ref) {
return <input placeholder="Add comment..." ref={ref} />;
});

export default AddComment;
```

```css
.CommentList {
height: 100px;
overflow: scroll;
border: 1px solid black;
margin-top: 20px;
margin-bottom: 20px;
}
```

```

</Sandpack>

<Pitfall>

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

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

</Pitfall>

---

title: "Built-in React Hooks"

---

<Intro>

*Hooks* let you use different React features from your components. You can either use the built-in Hooks or combine them to build your own. This page lists all built-in Hooks in React.

</Intro>

---

## State Hooks {/*state-hooks*/}

*State* lets a component ["remember" information like user input.](/learn/state-a-components-memory) For example, a form component can use state to store the input value, while an image gallery component can use state to store the selected image index.

To add state to a component, use one of these Hooks:

* [`useState`](/reference/react/useState) declares a state variable that you can update directly.

* [`useReducer`](/reference/react/useReducer) declares a state variable with the update logic inside a [reducer function.](/learn/extracting-state-logic-into-a-reducer)

```js

function ImageGallery() {

const [index, setIndex] = useState(0);

// ...
```

```

Context Hooks {/*context-hooks*/}

Context lets a component [receive information from distant parents without passing it as props.](/learn/passing-props-to-a-component) For example, your app's top-level component can pass the current UI theme to all components below, no matter how deep.

* [`useContext`](/reference/react/useContext) reads and subscribes to a context.

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

Ref Hooks {/*ref-hooks*/}

Refs let a component [hold some information that isn't used for rendering,](/learn/referencing-values-with-refs) like a DOM node or a timeout ID. Unlike with state, updating a ref does not re-render your component. Refs are an "escape hatch" from the React paradigm. They are useful when you need to work with non-React systems, such as the built-in browser APIs.

* [`useRef`](/reference/react/useRef) declares a ref. You can hold any value in it, but most often it's used to hold a DOM node.
* [`useImperativeHandle`](/reference/react/useImperativeHandle) lets you customize the ref exposed by your component. This is rarely used.

```js
function Form() {
const inputRef = useRef(null);
// ...
```

Effect Hooks {/*effect-hooks*/}

Effects let a component [connect to and synchronize with external systems.](/learn/synchronizing-with-effects) This includes dealing with network, browser DOM, animations, widgets written using a different UI library, and other non-React code.

* [`useEffect`](/reference/react/useEffect) connects a component to an external system.

```js
function ChatRoom({ roomId }) {
```

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

Effects are an "escape hatch" from the React paradigm. Don't use Effects to orchestrate the data flow of your application. If you're not interacting with an external system, [you might not need an Effect.](/learn/you-might-not-need-an-effect)

There are two rarely used variations of `useEffect` with differences in timing:

* [`useLayoutEffect`](/reference/react/useLayoutEffect) fires before the browser repaints the screen. You can measure layout here.
* [`useInsertionEffect`](/reference/react/useInsertionEffect) fires before React makes changes to the DOM. Libraries can insert dynamic CSS here.

Performance Hooks {/*performance-hooks*/}

A common way to optimize re-rendering performance is to skip unnecessary work. For example, you can tell React to reuse a cached calculation or to skip a re-render if the data has not changed since the previous render.

To skip calculations and unnecessary re-rendering, use one of these Hooks:

- [`useMemo`](/reference/react/useMemo) lets you cache the result of an expensive calculation.
- [`useCallback`](/reference/react/useCallback) lets you cache a function definition before passing it down to an optimized component.

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

Sometimes, you can't skip re-rendering because the screen actually needs to update. In that case, you can improve performance by separating blocking updates that must be synchronous (like typing into an input) from non-blocking updates which don't need to block the user interface (like updating a chart).

To prioritize rendering, use one of these Hooks:

- [`useTransition`](/reference/react/useTransition) lets you mark a state transition as non-blocking and allow other updates to interrupt it.

- [`useDeferredValue`](/reference/react/useDeferredValue) lets you defer updating a non-critical part of the UI and let other parts update first.

Other Hooks {/*other-hooks*/}

These Hooks are mostly useful to library authors and aren't commonly used in the application code.

- [`useDebugValue`](/reference/react/useDebugValue) lets you customize the label React DevTools displays for your custom Hook.

- [`useId`](/reference/react/useId) lets a component associate a unique ID with itself. Typically used with accessibility APIs.

- [`useSyncExternalStore`](/reference/react/useSyncExternalStore) lets a component subscribe to an external store.

Your own Hooks {/*your-own-hooks*/}

You can also [define your own custom Hooks](/learn/reusing-logic-with-custom-hooks#extracting-your-own-custom-hook-from-a-component) as JavaScript functions.

---

title: useRef

<Intro>

`useRef` is a React Hook that lets you reference a value that's not needed for rendering.

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

</Intro>

<InlineToc />

Reference {/*reference*/}

`useRef(initialValue)` {/*useref*/}

Call `useRef` at the top level of your component to declare a [ref.](/learn/referencing-values-with-refs)

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

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

[See more examples below.](#usage)

Parameters {/*parameters*/}

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

Returns {/*returns*/}

`useRef` returns an object with a single property:

* `current`: Initially, it's set to the `initialValue` you have passed. You can later set it to something else. If you pass the ref object to React as a `ref` attribute to a JSX node, React will set its `current` property.

On the next renders, `useRef` will return the same object.

Caveats {/*caveats*/}

* You can mutate the `ref.current` property. Unlike state, it is mutable. However, if it holds an object that is used for rendering (for example, a piece of your state), then you shouldn't mutate that object.

* When you change the `ref.current` property, React does not re-render your component. React is not aware of when you change it because a ref is a plain JavaScript object.

* Do not write _or read_ `ref.current` during rendering, except for [initialization.](#avoiding-recreating-the-ref-contents) This makes your component's behavior unpredictable.

* In Strict Mode, React will **call your component function twice** in order to [help you find accidental impurities.](#my-initializer-or-updater-function-runs-twice) This is development-only behavior and does not affect production. Each ref object will be created twice, but one of the versions will be discarded. If your component function is pure (as it should be), this should not affect the behavior.

Usage {/*usage*/}

Referencing a value with a ref {/*referencing-a-value-with-a-ref*/}

Call `useRef` at the top level of your component to declare one or more [refs.](/learn/referencing-values-with-refs)

```js [[1, 4, "intervalRef"], [3, 4, "0"]]
```

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

function Stopwatch() {

const intervalRef = useRef(0);

// ...
```

`useRef` returns a <CodeStep step={1}>ref object</CodeStep> with a single <CodeStep step={2}>`current` property</CodeStep> initially set to the <CodeStep step={3}>initial value</CodeStep> you provided.

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

Changing a ref does not trigger a re-render. This means refs are perfect for storing information that doesn't affect the visual output of your component. For example, if you need to store an [interval ID](https://developer.mozilla.org/en-US/docs/Web/API/setInterval) and retrieve it later, you can put it in a ref. To update the value inside the ref, you need to manually change its <CodeStep step={2}>`current` property</CodeStep>:

```js [[2, 5, "intervalRef.current"]]

function handleStartClick() {

const intervalId = setInterval(() => {

// ...

}, 1000);

intervalRef.current = intervalId;

}
```

Later, you can read that interval ID from the ref so that you can call [clear that interval](https://developer.mozilla.org/en-US/docs/Web/API/clearInterval):

```js [[2, 2, "intervalRef.current"]]

function handleStopClick() {

const intervalId = intervalRef.current;

clearInterval(intervalId);

}
```

By using a ref, you ensure that:

- You can **store information** between re-renders (unlike regular variables, which reset on every render).

- Changing it **does not trigger a re-render** (unlike state variables, which trigger a re-render).

- The **information is local** to each copy of your component (unlike the variables outside, which are shared).

Changing a ref does not trigger a re-render, so refs are not appropriate for storing information you want to display on the screen. Use state for that instead. Read more about [choosing between `useRef` and `useState`.](/learn/referencing-values-with-refs#differences-between-refs-and-state)

<Recipes titleText="Examples of referencing a value with useRef" titleId="examples-value">

Click counter {/*click-counter*/}

This component uses a ref to keep track of how many times the button was clicked. Note that it's okay to use a ref instead of state here because the click count is only read and written in an event handler.

<Sandpack>

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

export default function Counter() {
let ref = useRef(0);

function handleClick() {
ref.current = ref.current + 1;
alert('You clicked ' + ref.current + ' times!');
}

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

</Sandpack>

If you show `{ref.current}` in the JSX, the number won't update on click. This is because setting `ref.current` does not trigger a re-render. Information that's used for rendering should be state instead.

<Solution />

A stopwatch {/*a-stopwatch*/}

This example uses a combination of state and refs. Both `startTime` and `now` are state variables because they are used for rendering. But we also need to hold an [interval ID](https://developer.mozilla.org/en-US/docs/Web/API/setInterval) so that we can stop the interval on

button press. Since the interval ID is not used for rendering, it's appropriate to keep it in a ref, and manually update it.

<Sandpack>

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

export default function Stopwatch() {
const [startTime, setStartTime] = useState(null);
const [now, setNow] = useState(null);
const intervalRef = useRef(null);

function handleStart() {
setStartTime(Date.now());
setNow(Date.now());

clearInterval(intervalRef.current);
intervalRef.current = setInterval(() => {
setNow(Date.now());
}, 10);
}

function handleStop() {
clearInterval(intervalRef.current);
}

let secondsPassed = 0;
if (startTime != null && now != null) {
secondsPassed = (now - startTime) / 1000;
}

return (
<>
<h1>Time passed: {secondsPassed.toFixed(3)}</h1>
<button onClick={handleStart}>
Start
</button>
<button onClick={handleStop}>
Stop
</button>
```

</>

);

}

```

</Sandpack>

<Solution />

</Recipes>

<Pitfall>

**Do not write _or read_ `ref.current` during rendering.**

React expects that the body of your component [behaves like a pure function](/learn/keeping-components-pure):

- If the inputs ([props](/learn/passing-props-to-a-component), [state](/learn/state-a-components-memory), and [context](/learn/passing-data-deeply-with-context)) are the same, it should return exactly the same JSX.
- Calling it in a different order or with different arguments should not affect the results of other calls.

Reading or writing a ref **during rendering** breaks these expectations.

```js {3-4,6-7}
function MyComponent() {
// ...
// ■ Don't write a ref during rendering
myRef.current = 123;
// ...
// ■ Don't read a ref during rendering
return <h1>{myOtherRef.current}</h1>;
}
```

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

```js {4-5,9-10}
function MyComponent() {
// ...
useEffect(() => {
// ■ You can read or write refs in effects
myRef.current = 123;

});

// ...

function handleClick() {

// ■ You can read or write refs in event handlers

doSomething(myOtherRef.current);

}

// ...

}

```

If you *have to* read [or write](/reference/react/useState#storing-information-from-previous-renders) something during rendering, [use state](/reference/react/useState) instead.

When you break these rules, your component might still work, but most of the newer features we're adding to React will rely on these expectations. Read more about [keeping your components pure.](/learn/keeping-components-pure#where-you-_can_-cause-side-effects)

</Pitfall>

Manipulating the DOM with a ref {/*manipulating-the-dom-with-a-ref*/}

It's particularly common to use a ref to manipulate the [DOM.](https://developer.mozilla.org/en-US/docs/Web/API/HTML_DOM_API) React has built-in support for this.

First, declare a <CodeStep step={1}>ref object</CodeStep> with an <CodeStep step={3}>initial value</CodeStep> of `null`:

```js [[1, 4, "inputRef"], [3, 4, "null"]]

import { useRef } from 'react';

function MyComponent() {

const inputRef = useRef(null);

// ...

```

Then pass your ref object as the `ref` attribute to the JSX of the DOM node you want to manipulate:

```js [[1, 2, "inputRef"]]

// ...

return <input ref={inputRef} />;

```
```

After React creates the DOM node and puts it on the screen, React will set the <CodeStep step={2}>`current` property</CodeStep> of your ref object to that DOM node. Now you can access the `<input>`'s DOM node and call methods like [`focus()`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/focus):

```js [[2, 2, "inputRef.current"]]
function handleClick() {
inputRef.current.focus();
}
```

React will set the `current` property back to `null` when the node is removed from the screen.

Read more about [manipulating the DOM with refs.](/learn/manipulating-the-dom-with-refs)

<Recipes titleText="Examples of manipulating the DOM with useRef" titleId="examples-dom">

Focusing a text input {/*focusing-a-text-input*/}

In this example, clicking the button will focus the input:

<Sandpack>

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

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

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

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

</Sandpack>

<Solution />

Scrolling an image into view {/*scrolling-an-image-into-view*/}

In this example, clicking the button will scroll an image into view. It uses a ref to the list DOM node, and then calls DOM [`querySelectorAll`](https://developer.mozilla.org/en-US/docs/Web/API/Document/querySelectorAll) API to find the image we want to scroll to.

<Sandpack>

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

export default function CatFriends() {
const listRef = useRef(null);

function scrollToIndex(index) {
const listNode = listRef.current;
// This line assumes a particular DOM structure:
const imgNode = listNode.querySelectorAll('li > img')[index];
imgNode.scrollIntoView({
behavior: 'smooth',
block: 'nearest',
inline: 'center'
});
}

return (
<>
<nav>
<button onClick={() => scrollToIndex(0)}>
Tom
</button>
<button onClick={() => scrollToIndex(1)}>
Maru
</button>
<button onClick={() => scrollToIndex(2)}>
Jellylorum
</button>
```

```
</nav>
<div>
<ul ref={listRef}>
<li>
<img
src="https://placekitten.com/g/200/200"
alt="Tom"
/>
</li>
<li>
<img
src="https://placekitten.com/g/300/200"
alt="Maru"
/>
</li>
<li>
<img
src="https://placekitten.com/g/250/200"
alt="Jellylorum"
/>
</li>
</ul>
</div>
</>
);
}
```

```css
div {
width: 100%;
overflow: hidden;
}

nav {
text-align: center;
}
```

```
button {
margin: .25rem;
}

ul,
li {
list-style: none;
white-space: nowrap;
}

li {
display: inline;
padding: 0.5rem;
}
```

</Sandpack>

<Solution />

Playing and pausing a video {/*playing-and-pausing-a-video*/}

This example uses a ref to call [`play()`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement/play) and [`pause()`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement/pause) on a `<video>` DOM node.

<Sandpack>

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

export default function VideoPlayer() {
const [isPlaying, setIsPlaying] = useState(false);
const ref = useRef(null);

function handleClick() {
const nextIsPlaying = !isPlaying;
setIsPlaying(nextIsPlaying);

if (nextIsPlaying) {
ref.current.play();
} else {
ref.current.pause();
```

```
}
}

return (
<>
<button onClick={handleClick}>
{isPlaying ? 'Pause' : 'Play'}
</button>
<video
width="250"
ref={ref}
onPlay={() => setIsPlaying(true)}
onPause={() => setIsPlaying(false)}
>
<source
src="https://interactive-examples.mdn.mozilla.net/media/cc0-videos/flower.mp4"
type="video/mp4"
/>
</video>
</>
);
}
```

```css
button { display: block; margin-bottom: 20px; }
```

</Sandpack>

<Solution />

Exposing a ref to your own component {/*exposing-a-ref-to-your-own-component*/}

Sometimes, you may want to let the parent component manipulate the DOM inside of your component. For example, maybe you're writing a `MyInput` component, but you want the parent to be able to focus the input (which the parent has no access to). You can use a combination of `useRef` to hold the input and [`forwardRef`](/reference/react/forwardRef) to expose it to the parent component. Read a [detailed walkthrough](/learn/manipulating-the-dom-with-refs#accessing-another-components-dom-nodes) here.

<Sandpack>

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

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

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

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

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

</Sandpack>

<Solution />

</Recipes>

Avoiding recreating the ref contents {/*avoiding-recreating-the-ref-contents*/}

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

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

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

To solve it, you may initialize the ref like this instead:

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

Normally, writing or reading `ref.current` during render is not allowed. However, it's fine in this case because the result is always the same, and the condition only executes during initialization so it's fully predictable.

<DeepDive>

How to avoid null checks when initializing useRef later
{/*how-to-avoid-null-checks-when-initializing-use-ref-later*/}

If you use a type checker and don't want to always check for `null`, you can try a pattern like this instead:

```js
function Video() {
const playerRef = useRef(null);

function getPlayer() {
if (playerRef.current !== null) {
return playerRef.current;
}
const player = new VideoPlayer();
playerRef.current = player;
return player;
}

// ...
```

Here, the `playerRef` itself is nullable. However, you should be able to convince your type checker that there is no case in which `getPlayer()` returns `null`. Then use `getPlayer()` in your event handlers.

</DeepDive>

Troubleshooting {/*troubleshooting*/}

I can't get a ref to a custom component {/*i-cant-get-a-ref-to-a-custom-component*/}

If you try to pass a `ref` to your own component like this:

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

return <MyInput ref={inputRef} />;
```

You might get an error in the console:

<ConsoleBlock level="error">

Warning: Function components cannot be given refs. Attempts to access this ref will fail. Did you mean to use React.forwardRef()?

</ConsoleBlock>

By default, your own components don't expose refs to the DOM nodes inside them.

To fix this, find the component that you want to get a ref to:

```js
export default function MyInput({ value, onChange }) {
return (
<input
value={value}
onChange={onChange}
/>
);
}
```

And then wrap it in [`forwardRef`](/reference/react/forwardRef) like this:

```js {3,8}
import { forwardRef } from 'react';

const MyInput = forwardRef(({ value, onChange }, ref) => {
```

return (

<input

value={value}

onChange={onChange}

ref={ref}

/>

);

});

export default MyInput;

```

Then the parent component can get a ref to it.

Read more about [accessing another component's DOM nodes.](/learn/manipulating-the-dom-with-refs#accessing-another-components-dom-nodes)

---

title: PureComponent

---

<Pitfall>

We recommend defining components as functions instead of classes. [See how to migrate.](#alternatives)

</Pitfall>

<Intro>

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

```js

class Greeting extends PureComponent {

render() {

return <h1>Hello, {this.props.name}!</h1>;

}

}

```

</Intro>

<InlineToc />
```

Reference {/*reference*/}

`PureComponent` {/*purecomponent*/}

To skip re-rendering a class component for same props and state, extend `PureComponent` instead of [`Component`:](/reference/react/Component)

```js
import { PureComponent } from 'react';

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

`PureComponent` is a subclass of `Component` and supports [all the `Component` APIs.](/reference/react/Component#reference) Extending `PureComponent` is equivalent to defining a custom [`shouldComponentUpdate`](/reference/react/Component#shouldcomponentupdate) method that shallowly compares props and state.

[See more examples below.](#usage)

Usage {/*usage*/}

Skipping unnecessary re-renders for class components {/*skipping-unnecessary-re-renders-for-class-components*/}

React normally re-renders a component whenever its parent re-renders. As an optimization, you can create a component that React will not re-render when its parent re-renders so long as its new props and state are the same as the old props and state. [Class components](/reference/react/Component) can opt into this behavior by extending `PureComponent`:

```js {1}
class Greeting extends PureComponent {
render() {
return <h1>Hello, {this.props.name}!</h1>;
}
}
```

A React component should always have [pure rendering logic.](/learn/keeping-components-pure) This means that it must return the same output if its props, state, and context haven't changed. By using `PureComponent`, you are telling React that your component complies with this requirement, so React doesn't need to re-render as long as its props and state haven't changed. However, your component will still re-render if a context that it's using changes.

In this example, notice that the `Greeting` component re-renders whenever `name` is changed (because that's one of its props), but not when `address` is changed (because it's not passed to `Greeting` as a prop):

<Sandpack>

```js
import { PureComponent, useState } from 'react';

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

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

```css
label {
display: block;
margin-bottom: 16px;
}
```

</Sandpack>

<Pitfall>

We recommend defining components as functions instead of classes. [See how to migrate.](#alternatives)

</Pitfall>

Alternatives {/*alternatives*/}

Migrating from a `PureComponent` class component to a function {/*migrating-from-a-purecomponent-class-component-to-a-function*/}

We recommend using function components instead of [class components](/reference/react/Component) in new code. If you have some existing class components using `PureComponent`, here is how you can convert them. This is the original code:

<Sandpack>

```js
import { PureComponent, useState } from 'react';

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

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

Name{': '}

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

</label>

<label>

Address{': '}

<input value={address} onChange={e => setAddress(e.target.value)} />

</label>

<Greeting name={name} />

</>

);

}

```

```css

label {

display: block;

margin-bottom: 16px;

}

```

</Sandpack>

When you [convert this component from a class to a function,](/reference/react/Component#alternatives) wrap it in [`memo`:](/reference/react/memo)

<Sandpack>

```js

import { memo, useState } from 'react';

const Greeting = memo(function Greeting({ name }) {

console.log("Greeting was rendered at", new Date().toLocaleTimeString());

return <h3>Hello{name && ', '}{name}!</h3>;

});

export default function MyApp() {

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

const [address, setAddress] = useState('');

return (

<>

<label>

Name{': '}

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

</label>

<label>

Address{': '}

<input value={address} onChange={e => setAddress(e.target.value)} />

</label>

<Greeting name={name} />

</>

);

}

```

```css

label {

display: block;

margin-bottom: 16px;

}

```

</Sandpack>

<Note>

Unlike `PureComponent`, [`memo`](/reference/react/memo) does not compare the new and the old state. In function components, calling the [`set` function](/reference/react/useState#setstate) with the same state [already prevents re-renders by default,](/reference/react/memo#updating-a-memoized-component-using-state) even without `memo`.

</Note>

---

title: "Built-in React Components"

<Intro>

React exposes a few built-in components that you can use in your JSX.

</Intro>

```

Built-in components {/*built-in-components*/}

* [`<Fragment>`](/reference/react/Fragment), alternatively written as `<>...</>`, lets you group multiple JSX nodes together.
* [`<Profiler>`](/reference/react/Profiler) lets you measure rendering performance of a React tree programmatically.
* [`<Suspense>`](/reference/react/Suspense) lets you display a fallback while the child components are loading.
* [`<StrictMode>`](/reference/react/StrictMode) enables extra development-only checks that help you find bugs early.

Your own components {/*your-own-components*/}

You can also [define your own components](/learn/your-first-component) as JavaScript functions.

---

title: useCallback

<Intro>

`useCallback` is a React Hook that lets you cache a function definition between re-renders.

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

</Intro>

<InlineToc />

Reference {/*reference*/}

`useCallback(fn, dependencies)` {/*usecallback*/}

Call `useCallback` at the top level of your component to cache a function definition between re-renders:

```js {4,9}
import { useCallback } from 'react';

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

orderDetails,

});

}, [productId, referrer]);

```

[See more examples below.](#usage)

#### Parameters {/*parameters*/}

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

* `dependencies`: The list of all reactive values referenced inside of the `fn` code. Reactive values include props, state, and all the variables and functions declared directly inside your component body. If your linter is [configured for React](/learn/editor-setup#linting), it will verify that every reactive value is correctly specified as a dependency. The list of dependencies must have a constant number of items and be written inline like `[dep1, dep2, dep3]`. React will compare each dependency with its previous value using the [`Object.is`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/is) comparison algorithm.

#### Returns {/*returns*/}

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

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

#### Caveats {/*caveats*/}

* `useCallback` is a Hook, so you can only call it **at the top level of your component** or your own Hooks. You can't call it inside loops or conditions. If you need that, extract a new component and move the state into it.

* React **will not throw away the cached function unless there is a specific reason to do that.** For example, in development, React throws away the cache when you edit the file of your component. Both in development and in production, React will throw away the cache if your component suspends during the initial mount. In the future, React may add more features that take advantage of throwing away the cache--for example, if React adds built-in support for virtualized lists in the future, it would make sense to throw away the cache for items that scroll out of the virtualized table viewport. This should match your expectations if you rely on `useCallback` as a performance optimization. Otherwise, a [state variable](/reference/react/useState#im-trying-to-set-state-to-a-function-but-it-gets-called-instead) or a [ref](/reference/react/useRef#avoiding-recreating-the-ref-contents) may be more appropriate.

---

## Usage {/*usage*/}

### Skipping re-rendering of components {/*skipping-re-rendering-of-components*/}
```

When you optimize rendering performance, you will sometimes need to cache the functions that you pass to child components. Let's first look at the syntax for how to do this, and then see in which cases it's useful.

To cache a function between re-renders of your component, wrap its definition into the `useCallback` Hook:

```js [[3, 4, "handleSubmit"], [2, 9, "[productId, referrer]"]]
import { useCallback } from 'react';

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

You need to pass two things to `useCallback`:

1. A function definition that you want to cache between re-renders.
2. A <CodeStep step={2}>list of dependencies</CodeStep> including every value within your component that's used inside your function.

On the initial render, the <CodeStep step={3}>returned function</CodeStep> you'll get from `useCallback` will be the function you passed.

On the following renders, React will compare the <CodeStep step={2}>dependencies</CodeStep> with the dependencies you passed during the previous render. If none of the dependencies have changed (compared with [`Object.is`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/is)), `useCallback` will return the same function as before. Otherwise, `useCallback` will return the function you passed on *this* render.

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

Let's walk through an example to see when this is useful.

Say you're passing a `handleSubmit` function down from the `ProductPage` to the `ShippingForm` component:

```js {5}
function ProductPage({ productId, referrer, theme }) {
// ...
return (
```

<ShippingForm onSubmit={handleSubmit} />

);
```

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

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

```js {3,5}

import { memo } from 'react';

const ShippingForm = memo(function ShippingForm({ onSubmit }) {

// ...

});
```

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

```js {2,3,8,12-13}

function ProductPage({ productId, referrer, theme }) {

// Every time the theme changes, this will be a different function...

function handleSubmit(orderDetails) {

post('/product/' + productId + '/buy', {

referrer,

orderDetails,

});

}

return (

{/* ... so ShippingForm's props will never be the same, and it will re-render every time */}

<ShippingForm onSubmit={handleSubmit} />

);

}

```

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

```js {2,3,8,12-13}

function ProductPage({ productId, referrer, theme }) {

// Tell React to cache your function between re-renders...

const handleSubmit = useCallback((orderDetails) => {

post('/product/' + productId + '/buy', {

referrer,

orderDetails,

});

}, [productId, referrer]); // ...so as long as these dependencies don't change...

return (

<div className={theme}>

{/* ...ShippingForm will receive the same props and can skip re-rendering */}

<ShippingForm onSubmit={handleSubmit} />

</div>

);

}

```

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

<Note>

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

</Note>

<DeepDive>

How is useCallback related to useMemo? {/*how-is-usecallback-related-to-usememo*/}
```

You will often see [`useMemo`](/reference/react/useMemo) alongside `useCallback`. They are both useful when you're trying to optimize a child component. They let you [memoize](https://en.wikipedia.org/wiki/Memoization) (or, in other words, cache) something you're passing down:

```js {6-8,10-15,19}
import { useMemo, useCallback } from 'react';

function ProductPage({ productId, referrer }) {
const product = useData('/product/' + productId);

const requirements = useMemo(() => { // Calls your function and caches its result
return computeRequirements(product);
}, [product]);

const handleSubmit = useCallback((orderDetails) => { // Caches your function itself
post('/product/' + productId + '/buy', {
referrer,
orderDetails,
});
}, [productId, referrer]);

return (
<div className={theme}>
<ShippingForm requirements={requirements} onSubmit={handleSubmit} />
</div>
);
}
```

The difference is in *what* they're letting you cache:

* **[`useMemo`](/reference/react/useMemo) caches the *result* of calling your function.** In this example, it caches the result of calling `computeRequirements(product)` so that it doesn't change unless `product` has changed. This lets you pass the `requirements` object down without unnecessarily re-rendering `ShippingForm`. When necessary, React will call the function you've passed during rendering to calculate the result.

* **`useCallback` caches *the function itself.*** Unlike `useMemo`, it does not call the function you provide. Instead, it caches the function you provided so that `handleSubmit` *itself* doesn't change unless `productId` or `referrer` has changed. This lets you pass the `handleSubmit` function down without unnecessarily re-rendering `ShippingForm`. Your code won't run until the user submits the form.

If you're already familiar with [`useMemo`,](/reference/react/useMemo) you might find it helpful to think of `useCallback` as this:

```js
// Simplified implementation (inside React)
function useCallback(fn, dependencies) {
return useMemo(() => fn, dependencies);
}
```

[Read more about the difference between `useMemo` and `useCallback`.](/reference/react/useMemo#memoizing-a-function)

</DeepDive>

<DeepDive>

Should you add useCallback everywhere? {/*should-you-add-usecallback-everywhere*/}

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

Caching a function with `useCallback` is only valuable in a few cases:

- You pass it as a prop to a component wrapped in [`memo`.](/reference/react/memo) You want to skip re-rendering if the value hasn't changed. Memoization lets your component re-render only if dependencies changed.
- The function you're passing is later used as a dependency of some Hook. For example, another function wrapped in `useCallback` depends on it, or you depend on this function from [`useEffect.`](/reference/react/useEffect)

There is no benefit to wrapping a function in `useCallback` in other cases. There is no significant harm to doing that either, so some teams choose to not think about individual cases, and memoize as much as possible. The downside is that code becomes less readable. Also, not all memoization is effective: a single value that's "always new" is enough to break memoization for an entire component.

Note that `useCallback` does not prevent *creating* the function. You're always creating a function (and that's fine!), but React ignores it and gives you back a cached function if nothing changed.

In practice, you can make a lot of memoization unnecessary by following a few principles:

1. When a component visually wraps other components, let it [accept JSX as children.](/learn/passing-props-to-a-component#passing-jsx-as-children) Then, if the wrapper component updates its own state, React knows that its children don't need to re-render.
1. Prefer local state and don't [lift state up](/learn/sharing-state-between-components) any further than necessary. Don't keep transient state like forms and whether an item is hovered at the top of your tree or in a global state library.
1. Keep your [rendering logic pure.](/learn/keeping-components-pure) If re-rendering a component causes a problem or produces some noticeable visual artifact, it's a bug in your component! Fix the bug instead of adding memoization.

1. Avoid [unnecessary Effects that update state.](/learn/you-might-not-need-an-effect) Most performance problems in React apps are caused by chains of updates originating from Effects that cause your components to render over and over.

1. Try to [remove unnecessary dependencies from your Effects.](/learn/removing-effect-dependencies) For example, instead of memoization, it's often simpler to move some object or a function inside an Effect or outside the component.

If a specific interaction still feels laggy, [use the React Developer Tools profiler](https://legacy.reactjs.org/blog/2018/09/10/introducing-the-react-profiler.html) to see which components benefit the most from memoization, and add memoization where needed. These principles make your components easier to debug and understand, so it's good to follow them in any case. In long term, we're researching [doing memoization automatically](https://www.youtube.com/watch?v=lGEMwh32soc) to solve this once and for all.

</DeepDive>

<Recipes titleText="The difference between useCallback and declaring a function directly" titleId="examples-rerendering">

Skipping re-rendering with `useCallback` and `memo`
{/*skipping-re-rendering-with-usecallback-and-memo*/}

In this example, the `ShippingForm` component is **artificially slowed down** so that you can see what happens when a React component you're rendering is genuinely slow. Try incrementing the counter and toggling the theme.

Incrementing the counter feels slow because it forces the slowed down `ShippingForm` to re-render. That's expected because the counter has changed, and so you need to reflect the user's new choice on the screen.

Next, try toggling the theme. **Thanks to `useCallback` together with [`memo`](/reference/react/memo), it’s fast despite the artificial slowdown!** `ShippingForm` skipped re-rendering because the `handleSubmit` function has not changed. The `handleSubmit` function has not changed because both `productId` and `referrer` (your `useCallback` dependencies) haven't changed since last render.

<Sandpack>

```js App.js
import { useState } from 'react';
import ProductPage from './ProductPage.js';

export default function App() {
const [isDark, setIsDark] = useState(false);
return (
<>
<label>
<input
type="checkbox"
```

````
checked={isDark}
onChange={e => setIsDark(e.target.checked)}
/>
Dark mode
</label>
<hr />
<ProductPage
referrerId="wizard_of_oz"
productId={123}
theme={isDark ? 'dark' : 'light'}
/>
</>
);
}
```

```js ProductPage.js active
import { useCallback } from 'react';
import ShippingForm from './ShippingForm.js';

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

return (
<div className={theme}>
<ShippingForm onSubmit={handleSubmit} />
</div>
);
}

function post(url, data) {
// Imagine this sends a request...
console.log('POST /' + url);
````

````
console.log(data);
}
```

```js ShippingForm.js
import { memo, useState } from 'react';

const ShippingForm = memo(function ShippingForm({ onSubmit }) {
const [count, setCount] = useState(1);

console.log('[ARTIFICIALLY SLOW] Rendering <ShippingForm />');
let startTime = performance.now();
while (performance.now() - startTime < 500) {
// Do nothing for 500 ms to emulate extremely slow code
}

function handleSubmit(e) {
e.preventDefault();
const formData = new FormData(e.target);
const orderDetails = {
...Object.fromEntries(formData),
count
};
onSubmit(orderDetails);
}

return (
<form onSubmit={handleSubmit}>
<p><b>Note: <code>ShippingForm</code> is artificially slowed down!</b></p>
<label>
Number of items:
<button type="button" onClick={() => setCount(count - 1)}>–</button>
{count}
<button type="button" onClick={() => setCount(count + 1)}>+</button>
</label>
<label>
Street:
<input name="street" />
</label>
````

<label>
City:
<input name="city" />
</label>
<label>
Postal code:
<input name="zipCode" />
</label>
<button type="submit">Submit</button>
</form>
);
});

export default ShippingForm;
```

```css
label {
display: block; margin-top: 10px;
}

input {
margin-left: 5px;
}

button[type="button"] {
margin: 5px;
}

.dark {
background-color: black;
color: white;
}

.light {
background-color: white;
color: black;
}
```
```

</Sandpack>

<Solution />

Always re-rendering a component {/*always-re-rendering-a-component*/}

In this example, the `ShippingForm` implementation is also **artificially slowed down** so that you can see what happens when some React component you're rendering is genuinely slow. Try incrementing the counter and toggling the theme.

Unlike in the previous example, toggling the theme is also slow now! This is because **there is no `useCallback` call in this version,** so `handleSubmit` is always a new function, and the slowed down `ShippingForm` component can't skip re-rendering.

<Sandpack>

```js App.js
import { useState } from 'react';
import ProductPage from './ProductPage.js';

export default function App() {
const [isDark, setIsDark] = useState(false);
return (
<>
<label>
<input
type="checkbox"
checked={isDark}
onChange={e => setIsDark(e.target.checked)}
/>
Dark mode
</label>
<hr />
<ProductPage
referrerId="wizard_of_oz"
productId={123}
theme={isDark ? 'dark' : 'light'}
/>
</>
);
}
```

```

```js ProductPage.js active
import ShippingForm from './ShippingForm.js';

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

return (
<div className={theme}>
<ShippingForm onSubmit={handleSubmit} />
</div>
);
}

function post(url, data) {
// Imagine this sends a request...
console.log('POST /' + url);
console.log(data);
}
```

```js ShippingForm.js
import { memo, useState } from 'react';

const ShippingForm = memo(function ShippingForm({ onSubmit }) {
const [count, setCount] = useState(1);

console.log('[ARTIFICIALLY SLOW] Rendering <ShippingForm />');
let startTime = performance.now();
while (performance.now() - startTime < 500) {
// Do nothing for 500 ms to emulate extremely slow code
}

function handleSubmit(e) {
e.preventDefault();

```
const formData = new FormData(e.target);
const orderDetails = {
...Object.fromEntries(formData),
count
};
onSubmit(orderDetails);
}

return (
<form onSubmit={handleSubmit}>
<p><b>Note: <code>ShippingForm</code> is artificially slowed down!</b></p>
<label>
Number of items:
<button type="button" onClick={() => setCount(count - 1)}>–</button>
{count}
<button type="button" onClick={() => setCount(count + 1)}>+</button>
</label>
<label>
Street:
<input name="street" />
</label>
<label>
City:
<input name="city" />
</label>
<label>
Postal code:
<input name="zipCode" />
</label>
<button type="submit">Submit</button>
</form>
);
});

export default ShippingForm;
```

```css
label {
display: block; margin-top: 10px;
}

input {
margin-left: 5px;
}

button[type="button"] {
margin: 5px;
}

.dark {
background-color: black;
color: white;
}

.light {
background-color: white;
color: black;
}
```

</Sandpack>

However, here is the same code **with the artificial slowdown removed.** Does the lack of `useCallback` feel noticeable or not?

<Sandpack>

```js App.js
import { useState } from 'react';
import ProductPage from './ProductPage.js';

export default function App() {
const [isDark, setIsDark] = useState(false);
return (
<>
<label>
<input
```

type="checkbox"
checked={isDark}
onChange={e => setIsDark(e.target.checked)}
/>
Dark mode
</label>
<hr />
<ProductPage
referrerId="wizard_of_oz"
productId={123}
theme={isDark ? 'dark' : 'light'}
/>
</>
);
}
```

```js ProductPage.js active
import ShippingForm from './ShippingForm.js';

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

return (

<ShippingForm onSubmit={handleSubmit} />

);
}

function post(url, data) {
// Imagine this sends a request...
console.log('POST /' + url);

```
console.log(data);
}
```

```js ShippingForm.js
import { memo, useState } from 'react';

const ShippingForm = memo(function ShippingForm({ onSubmit }) {
const [count, setCount] = useState(1);

console.log('Rendering <ShippingForm />');

function handleSubmit(e) {
e.preventDefault();
const formData = new FormData(e.target);
const orderDetails = {
...Object.fromEntries(formData),
count
};
onSubmit(orderDetails);
}

return (
<form onSubmit={handleSubmit}>
<label>
Number of items:
<button type="button" onClick={() => setCount(count - 1)}>–</button>
{count}
<button type="button" onClick={() => setCount(count + 1)}>+</button>
</label>
<label>
Street:
<input name="street" />
</label>
<label>
City:
<input name="city" />
</label>
<label>
```

Postal code:

<input name="zipCode" />

</label>

<button type="submit">Submit</button>

</form>

);

});

export default ShippingForm;

```

```css

label {

display: block; margin-top: 10px;

}

input {

margin-left: 5px;

}

button[type="button"] {

margin: 5px;

}

.dark {

background-color: black;

color: white;

}

.light {

background-color: white;

color: black;

}

```

</Sandpack>

Quite often, code without memoization works fine. If your interactions are fast enough, you don't need memoization.

Keep in mind that you need to run React in production mode, disable [React Developer Tools](/learn/react-developer-tools), and use devices similar to the ones your app's users have in order
```

to get a realistic sense of what's actually slowing down your app.

<Solution />

</Recipes>

Updating state from a memoized callback {/*updating-state-from-a-memoized-callback*/}

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

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

```js {6,7}
function TodoList() {
const [todos, setTodos] = useState([]);

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

You'll usually want memoized functions to have as few dependencies as possible. When you read some state only to calculate the next state, you can remove that dependency by passing an [updater function](/reference/react/useState#updating-state-based-on-the-previous-state) instead:

```js {6,7}
function TodoList() {
const [todos, setTodos] = useState([]);

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

Here, instead of making `todos` a dependency and reading it inside, you pass an instruction about *how* to update the state (`todos => [...todos, newTodo]`) to React. [Read more about updater functions.](/reference/react/useState#updating-state-based-on-the-previous-state)

Preventing an Effect from firing too often {/*preventing-an-effect-from-firing-too-often*/}

Sometimes, you might want to call a function from inside an [Effect:](/learn/synchronizing-with-effects)

```js {4-9,12}
function ChatRoom({ roomId }) {
const [message, setMessage] = useState('');

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

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

This creates a problem. [Every reactive value must be declared as a dependency of your Effect.](/learn/lifecycle-of-reactive-effects#react-verifies-that-you-specified-every-reactive-value-as-a-dependency) However, if you declare `createOptions` as a dependency, it will cause your Effect to constantly reconnect to the chat room:

```js {6}
useEffect(() => {
const options = createOptions();
const connection = createConnection();
connection.connect();
return () => connection.disconnect();
}, [createOptions]); // ■ Problem: This dependency changes on every render
// ...
```

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

```js {4-9,16}
function ChatRoom({ roomId }) {
```

```
const [message, setMessage] = useState('');

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

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

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

```
```js {5-10,16}
function ChatRoom({ roomId }) {
const [message, setMessage] = useState('');

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

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

```

Now your code is simpler and doesn't need `useCallback`. [Learn more about removing Effect dependencies.](/learn/removing-effect-dependencies#move-dynamic-objects-and-functions-inside-your-effect)

---

### Optimizing a custom Hook {/*optimizing-a-custom-hook*/}

If you're writing a [custom Hook,](/learn/reusing-logic-with-custom-hooks) it's recommended to wrap any functions that it returns into `useCallback`:

```js {4-6,8-10}
function useRouter() {
const { dispatch } = useContext(RouterStateContext);

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

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

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

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

Troubleshooting {/*troubleshooting*/}

Every time my component renders, `useCallback` returns a different function {/*every-time-my-component-renders-usecallback-returns-a-different-function*/}

Make sure you've specified the dependency array as a second argument!

If you forget the dependency array, `useCallback` will return a new function every time:

```js {7}
function ProductPage({ productId, referrer }) {
```
```

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

This is the corrected version passing the dependency array as a second argument:

```js {7}
function ProductPage({ productId, referrer }) {
const handleSubmit = useCallback((orderDetails) => {
post('/product/' + productId + '/buy', {
referrer,
orderDetails,
});
}, [productId, referrer]); // ■ Does not return a new function unnecessarily
// ...
```

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

```js {5}
const handleSubmit = useCallback((orderDetails) => {
// ..
}, [productId, referrer]);

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

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

```js
Object.is(temp1[0], temp2[0]); // Is the first dependency the same between the arrays?
Object.is(temp1[1], temp2[1]); // Is the second dependency the same between the arrays?
```

Object.is(temp1[2], temp2[2]); // ... and so on for every dependency ...
```

When you find which dependency is breaking memoization, either find a way to remove it, or [memoize it as well.](/reference/react/useMemo#memoizing-a-dependency-of-another-hook)

---

### I need to call `useCallback` for each list item in a loop, but it's not allowed {/*i-need-to-call-usememo-for-each-list-item-in-a-loop-but-its-not-allowed*/}

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

```js {5-14}
function ReportList({ items }) {
return (
<article>
{items.map(item => {
// ■ You can't call useCallback in a loop like this:
const handleClick = useCallback(() => {
sendReport(item)
}, [item]);
return (
<figure key={item.id}>
<Chart onClick={handleClick} />
</figure>
);
})}
</article>
);
}
```

Instead, extract a component for an individual item, and put `useCallback` there:

```js {5,12-21}
function ReportList({ items }) {
return (
<article>

{items.map(item =>

<Report key={item.id} item={item} />

)}

</article>

);

}

function Report({ item }) {

// ■ Call useCallback at the top level:

const handleClick = useCallback(() => {

sendReport(item)

}, [item]);

return (

<figure>

<Chart onClick={handleClick} />

</figure>

);

}

```

Alternatively, you could remove `useCallback` in the last snippet and instead wrap `Report` itself in [`memo`.](/reference/react/memo) If the `item` prop does not change, `Report` will skip re-rendering, so `Chart` will skip re-rendering too:

```js {5,6-8,15}

function ReportList({ items }) {

// ...

}

const Report = memo(function Report({ item }) {

function handleClick() {

sendReport(item);

}

return (

<figure>

<Chart onClick={handleClick} />

</figure>

);

});
```

---

title: Component

<Pitfall>

We recommend defining components as functions instead of classes. [See how to migrate.](#alternatives)

</Pitfall>

<Intro>

`Component` is the base class for the React components defined as [JavaScript classes.](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes) Class components are still supported by React, but we don't recommend using them in new code.

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

</Intro>

<InlineToc />

Reference {/*reference*/}

`Component` {/*component*/}

To define a React component as a class, extend the built-in `Component` class and define a [`render` method:](#render)

```js
import { Component } from 'react';

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

}

}

```

Only the `render` method is required, other methods are optional.

[See more examples below.](#usage)

`context` {/*context*/}

The [context](/learn/passing-data-deeply-with-context) of a class component is available as `this.context`. It is only available if you specify *which* context you want to receive using [`static contextType`](#static-contexttype) (modern) or [`static contextTypes`](#static-contexttypes) (deprecated).

A class component can only read one context at a time.

```js {2,5}

class Button extends Component {

static contextType = ThemeContext;

render() {

const theme = this.context;

const className = 'button-' + theme;

return (

<button className={className}>

{this.props.children}

</button>

);

}

}

```

<Note>

Reading `this.context` in class components is equivalent to [`useContext`](/reference/react/useContext) in function components.

[See how to migrate.](#migrating-a-component-with-context-from-a-class-to-a-function)

</Note>

```

`props` {/*props*/}

The props passed to a class component are available as `this.props`.

```js {3}
class Greeting extends Component {
render() {
return <h1>Hello, {this.props.name}!</h1>;
}
}

<Greeting name="Taylor" />
```

<Note>

Reading `this.props` in class components is equivalent to [declaring props](/learn/passing-props-to-a-component#step-2-read-props-inside-the-child-component) in function components.

[See how to migrate.](#migrating-a-simple-component-from-a-class-to-a-function)

</Note>

`refs` {/*refs*/}

<Deprecated>

This API will be removed in a future major version of React. [Use `createRef` instead.](/reference/react/createRef)

</Deprecated>

Lets you access [legacy string refs](https://reactjs.org/docs/refs-and-the-dom.html#legacy-api-string-refs) for this component.

`state` {/*state*/}

The state of a class component is available as `this.state`. The `state` field must be an object. Do not mutate the state directly. If you wish to change the state, call `setState` with the new state.

```js {2-4,7-9,18}
class Counter extends Component {
state = {
```

```
age: 42,
};

handleAgeChange = () => {
this.setState({
age: this.state.age + 1
});
};

render() {
return (
<>
<button onClick={this.handleAgeChange}>
Increment age
</button>
<p>You are {this.state.age}.</p>
</>
);
}
}
```

<Note>

Defining `state` in class components is equivalent to calling [`useState`](/reference/react/useState) in function components.

[See how to migrate.](#migrating-a-component-with-state-from-a-class-to-a-function)

</Note>

`constructor(props)` {/*constructor*/}

The [constructor](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/constructor) runs before your class component *mounts* (gets added to the screen). Typically, a constructor is only used for two purposes in React. It lets you declare state and [bind](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_objects/Function/bind) your class methods to the class instance:

```js {2-6}
class Counter extends Component {
```

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

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

If you use modern JavaScript syntax, constructors are rarely needed. Instead, you can rewrite this code above using the [public class field syntax](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/Public_class_fields) which is supported both by modern browsers and tools like [Babel:](https://babeljs.io/)

```js {2,4}
class Counter extends Component {
state = { counter: 0 };

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

A constructor should not contain any side effects or subscriptions.

Parameters {/*constructor-parameters*/}

* `props`: The component's initial props.

Returns {/*constructor-returns*/}

`constructor` should not return anything.

Caveats {/*constructor-caveats*/}

* Do not run any side effects or subscriptions in the constructor. Instead, use [`componentDidMount`](#componentdidmount) for that.

* Inside a constructor, you need to call `super(props)` before any other statement. If you don't do that, `this.props` will be `undefined` while the constructor runs, which can be confusing and cause bugs.

* Constructor is the only place where you can assign [`this.state`](#state) directly. In all other methods, you need to use [`this.setState()`](#setstate) instead. Do not call `setState` in the constructor.

* When you use [server rendering,](/reference/react-dom/server) the constructor will run on the server too, followed by the [`render`](#render) method. However, lifecycle methods like `componentDidMount` or `componentWillUnmount` will not run on the server.

* When [Strict Mode](/reference/react/StrictMode) is on, React will call `constructor` twice in development and then throw away one of the instances. This helps you notice the accidental side effects that need to be moved out of the `constructor`.

<Note>

There is no exact equivalent for `constructor` in function components. To declare state in a function component, call [`useState`.](/reference/react/useState) To avoid recalculating the initial state, [pass a function to `useState`.](/reference/react/useState#avoiding-recreating-the-initial-state)

</Note>

`componentDidCatch(error, info)` {/*componentdidcatch*/}

If you define `componentDidCatch`, React will call it when some child component (including distant children) throws an error during rendering. This lets you log that error to an error reporting service in production.

Typically, it is used together with [`static getDerivedStateFromError`](#static-getderivedstatefromerror) which lets you update state in response to an error and display an error message to the user. A component with these methods is called an *error boundary.*

[See an example.](#catching-rendering-errors-with-an-error-boundary)

Parameters {/*componentdidcatch-parameters*/}

* `error`: The error that was thrown. In practice, it will usually be an instance of [`Error`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error) but this is not guaranteed because JavaScript allows to [`throw`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/throw) any value, including strings or even `null`.

* `info`: An object containing additional information about the error. Its `componentStack` field contains a stack trace with the component that threw, as well as the names and source locations of all its parent components. In production, the component names will be minified. If you set up production error reporting, you can decode the component stack using sourcemaps the same way as you would do for regular JavaScript error stacks.

Returns {/*componentdidcatch-returns*/}

`componentDidCatch` should not return anything.

Caveats {/*componentdidcatch-caveats*/}

* In the past, it was common to call `setState` inside `componentDidCatch` in order to update the UI and display the fallback error message. This is deprecated in favor of defining [`static getDerivedStateFromError`.](#static-getderivedstatefromerror)

* Production and development builds of React slightly differ in the way `componentDidCatch` handles errors. In development, the errors will bubble up to `window`, which means that any `window.onerror` or `window.addEventListener('error', callback)` will intercept the errors that have been caught by `componentDidCatch`. In production, instead, the errors will not bubble up, which means any ancestor error handler will only receive errors not explicitly caught by `componentDidCatch`.

<Note>

There is no direct equivalent for `componentDidCatch` in function components yet. If you'd like to avoid creating class components, write a single `ErrorBoundary` component like above and use it throughout your app. Alternatively, you can use the [`react-error-boundary`](https://github.com/bvaughn/react-error-boundary) package which does that for you.

</Note>

`componentDidMount()` {/*componentdidmount*/}

If you define the `componentDidMount` method, React will call it when your component is added *(mounted)* to the screen. This is a common place to start data fetching, set up subscriptions, or manipulate the DOM nodes.

If you implement `componentDidMount`, you usually need to implement other lifecycle methods to avoid bugs. For example, if `componentDidMount` reads some state or props, you also have to implement [`componentDidUpdate`](#componentdidupdate) to handle their changes, and [`componentWillUnmount`](#componentwillunmount) to clean up whatever `componentDidMount` was doing.

```js {6-8}
class ChatRoom extends Component {
state = {
serverUrl: 'https://localhost:1234'
};

componentDidMount() {
this.setupConnection();
}

componentDidUpdate(prevProps, prevState) {
if (
this.props.roomId !== prevProps.roomId ||
this.state.serverUrl !== prevState.serverUrl
) {
this.destroyConnection();
this.setupConnection();
```

```
}
}

componentWillUnmount() {
this.destroyConnection();
}

// ...
}
```

[See more examples.](#adding-lifecycle-methods-to-a-class-component)

Parameters {/*componentdidmount-parameters*/}

`componentDidMount` does not take any parameters.

Returns {/*componentdidmount-returns*/}

`componentDidMount` should not return anything.

Caveats {/*componentdidmount-caveats*/}

- When [Strict Mode](/reference/react/StrictMode) is on, in development React will call `componentDidMount`, then immediately call [`componentWillUnmount`,](#componentwillunmount) and then call `componentDidMount` again. This helps you notice if you forgot to implement `componentWillUnmount` or if its logic doesn't fully "mirror" what `componentDidMount` does.

- Although you may call [`setState`](#setstate) immediately in `componentDidMount`, it's best to avoid that when you can. It will trigger an extra rendering, but it will happen before the browser updates the screen. This guarantees that even though the [`render`](#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`](#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.

<Note>

For many use cases, defining `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` together in class components is equivalent to calling [`useEffect`](/reference/react/useEffect) in function components. In the rare cases where it's important for the code to run before browser paint, [`useLayoutEffect`](/reference/react/useLayoutEffect) is a closer match.

[See how to migrate.](#migrating-a-component-with-lifecycle-methods-from-a-class-to-a-function)

</Note>

`componentDidUpdate(prevProps, prevState, snapshot?)` {/*componentdidupdate*/}

If you define the `componentDidUpdate` method, React will call it immediately after your component has been re-rendered with updated props or state. This method is not called for the initial render.

You can use it to manipulate the DOM after an update. This is also a common 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). Typically, you'd use it together with [`componentDidMount`](#componentdidmount) and [`componentWillUnmount`:](#componentwillunmount)

```js {10-18}
class ChatRoom extends Component {
state = {
serverUrl: 'https://localhost:1234'
};

componentDidMount() {
this.setupConnection();
}

componentDidUpdate(prevProps, prevState) {
if (
this.props.roomId !== prevProps.roomId ||
this.state.serverUrl !== prevState.serverUrl
) {
this.destroyConnection();
this.setupConnection();
}
}

componentWillUnmount() {
this.destroyConnection();
}

// ...
}
```

[See more examples.](#adding-lifecycle-methods-to-a-class-component)

Parameters {/*componentdidupdate-parameters*/}

* `prevProps`: Props before the update. Compare `prevProps` to [`this.props`](#props) to determine what changed.

* `prevState`: State before the update. Compare `prevState` to [`this.state`](#state) to determine what changed.

* `snapshot`: If you implemented [`getSnapshotBeforeUpdate`](#getsnapshotbeforeupdate), `snapshot` will contain the value you returned from that method. Otherwise, it will be `undefined`.

Returns {/*componentdidupdate-returns*/}

`componentDidUpdate` should not return anything.

Caveats {/*componentdidupdate-caveats*/}

- `componentDidUpdate` will not get called if [`shouldComponentUpdate`](#shouldcomponentupdate) is defined and returns `false`.

- The logic inside `componentDidUpdate` should usually be wrapped in conditions comparing `this.props` with `prevProps`, and `this.state` with `prevState`. Otherwise, there's a risk of creating infinite loops.

- Although you may call [`setState`](#setstate) immediately in `componentDidUpdate`, it's best to avoid that when you can. It will trigger an extra rendering, but it will happen before the browser updates the screen. This guarantees that even though the [`render`](#render) will be called twice in this case, the user won't see the intermediate state. This pattern often causes performance issues, but it may be necessary for rare cases like modals and tooltips when you need to measure a DOM node before rendering something that depends on its size or position.

<Note>

For many use cases, defining `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` together in class components is equivalent to calling [`useEffect`](/reference/react/useEffect) in function components. In the rare cases where it's important for the code to run before browser paint, [`useLayoutEffect`](/reference/react/useLayoutEffect) is a closer match.

[See how to migrate.](#migrating-a-component-with-lifecycle-methods-from-a-class-to-a-function)

</Note>

`componentWillMount()` {/*componentwillmount*/}

<Deprecated>

This API has been renamed from `componentWillMount` to [`UNSAFE_componentWillMount`.](#unsafe_componentwillmount) The old name has been deprecated. In a future major version of React, only the new name will work.

Run the [`rename-unsafe-lifecycles` codemod](https://github.com/reactjs/react-codemod#rename-unsafe-lifecycles) to automatically update

your components.

</Deprecated>

`componentWillReceiveProps(nextProps)` {/*componentwillreceiveprops*/}

<Deprecated>

This API has been renamed from `componentWillReceiveProps` to [`UNSAFE_componentWillReceiveProps`.](#unsafe_componentwillreceiveprops) The old name has been deprecated. In a future major version of React, only the new name will work.

Run the [`rename-unsafe-lifecycles` codemod](https://github.com/reactjs/react-codemod#rename-unsafe-lifecycles) to automatically update your components.

</Deprecated>

`componentWillUpdate(nextProps, nextState)` {/*componentwillupdate*/}

<Deprecated>

This API has been renamed from `componentWillUpdate` to [`UNSAFE_componentWillUpdate`.](#unsafe_componentwillupdate) The old name has been deprecated. In a future major version of React, only the new name will work.

Run the [`rename-unsafe-lifecycles` codemod](https://github.com/reactjs/react-codemod#rename-unsafe-lifecycles) to automatically update your components.

</Deprecated>

`componentWillUnmount()` {/*componentwillunmount*/}

If you define the `componentWillUnmount` method, React will call it before your component is removed *(unmounted)* from the screen. This is a common place to cancel data fetching or remove subscriptions.

The logic inside `componentWillUnmount` should "mirror" the logic inside [`componentDidMount`.](#componentdidmount) For example, if `componentDidMount` sets up a subscription, `componentWillUnmount` should clean up that subscription. If the cleanup logic in your `componentWillUnmount` reads some props or state, you will usually also need to implement [`componentDidUpdate`](#componentdidupdate) to clean up resources (such as subscriptions) corresponding to the old props and state.

```js {20-22}
```

```
class ChatRoom extends Component {
state = {
serverUrl: 'https://localhost:1234'
};

componentDidMount() {
this.setupConnection();
}

componentDidUpdate(prevProps, prevState) {
if (
this.props.roomId !== prevProps.roomId ||
this.state.serverUrl !== prevState.serverUrl
) {
this.destroyConnection();
this.setupConnection();
}
}

componentWillUnmount() {
this.destroyConnection();
}

// ...
}
```

[See more examples.](#adding-lifecycle-methods-to-a-class-component)

Parameters {/*componentwillunmount-parameters*/}

`componentWillUnmount` does not take any parameters.

Returns {/*componentwillunmount-returns*/}

`componentWillUnmount` should not return anything.

Caveats {/*componentwillunmount-caveats*/}

- When [Strict Mode](/reference/react/StrictMode) is on, in development React will call [`componentDidMount`,](#componentdidmount) then immediately call `componentWillUnmount`, and then call `componentDidMount` again. This helps you notice if you forgot to implement `componentWillUnmount` or if its logic doesn't fully "mirror" what `componentDidMount` does.

<Note>

For many use cases, defining `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` together in class components is equivalent to calling [`useEffect`](/reference/react/useEffect) in function components. In the rare cases where it's important for the code to run before browser paint, [`useLayoutEffect`](/reference/react/useLayoutEffect) is a closer match.

[See how to migrate.](#migrating-a-component-with-lifecycle-methods-from-a-class-to-a-function)

</Note>

`forceUpdate(callback?)` {/*forceupdate*/}

Forces a component to re-render.

Usually, this is not necessary. If your component's [`render`](#render) method only reads from [`this.props`](#props), [`this.state`](#state), or [`this.context`,](#context) it will re-render automatically when you call [`setState`](#setstate) inside your component or one of its parents. However, if your component's `render` method reads directly from an external data source, you have to tell React to update the user interface when that data source changes. That's what `forceUpdate` lets you do.

Try to avoid all uses of `forceUpdate` and only read from `this.props` and `this.state` in `render`.

Parameters {/*forceupdate-parameters*/}

* **optional** `callback` If specified, React will call the `callback` you've provided after the update is committed.

Returns {/*forceupdate-returns*/}

`forceUpdate` does not return anything.

Caveats {/*forceupdate-caveats*/}

- If you call `forceUpdate`, React will re-render without calling [`shouldComponentUpdate`.](#shouldcomponentupdate)

<Note>

Reading an external data source and forcing class components to re-render in response to its changes with `forceUpdate` has been superseded by [`useSyncExternalStore`](/reference/react/useSyncExternalStore) in function components.

</Note>

`getChildContext()` {/*getchildcontext*/}

<Deprecated>

This API will be removed in a future major version of React. [Use `Context.Provider` instead.](/reference/react/createContext#provider)

</Deprecated>

Lets you specify the values for the [legacy context](https://reactjs.org/docs/legacy-context.html) is provided by this component.

`getSnapshotBeforeUpdate(prevProps, prevState)` {/*getsnapshotbeforeupdate*/}

If you implement `getSnapshotBeforeUpdate`, React will call it immediately before React updates the DOM. It enables your component to capture some information from the DOM (e.g. scroll position) before it is potentially changed. Any value returned by this lifecycle method will be passed as a parameter to [`componentDidUpdate`.](#componentdidupdate)

For example, you can use it in a UI like a chat thread that needs to preserve its scroll position during updates:

```js {7-15,17}
class ScrollingList extends React.Component {
constructor(props) {
super(props);
this.listRef = React.createRef();
}

getSnapshotBeforeUpdate(prevProps, prevState) {
// Are we adding new items to the list?
// Capture the scroll position so we can adjust scroll later.
if (prevProps.list.length < this.props.list.length) {
const list = this.listRef.current;
return list.scrollHeight - list.scrollTop;
}
return null;
}

componentDidUpdate(prevProps, prevState, snapshot) {
// If we have a snapshot value, we've just added new items.
// Adjust scroll so these new items don't push the old ones out of view.
// (snapshot here is the value returned from getSnapshotBeforeUpdate)
if (snapshot !== null) {
const list = this.listRef.current;
```

```
list.scrollTop = list.scrollHeight - snapshot;
}
}

render() {
return (
<div ref={this.listRef}>{/* ...contents... */}</div>
);
}
}
```

In the above example, it is important to read the `scrollHeight` property directly in `getSnapshotBeforeUpdate`. It is not safe to read it in [`render`](#render), [`UNSAFE_componentWillReceiveProps`](#unsafe_componentwillreceiveprops), or [`UNSAFE_componentWillUpdate`](#unsafe_componentwillupdate) because there is a potential time gap between these methods getting called and React updating the DOM.

Parameters {/*getsnapshotbeforeupdate-parameters*/}

* `prevProps`: Props before the update. Compare `prevProps` to [`this.props`](#props) to determine what changed.

* `prevState`: State before the update. Compare `prevState` to [`this.state`](#state) to determine what changed.

Returns {/*getsnapshotbeforeupdate-returns*/}

You should return a snapshot value of any type that you'd like, or `null`. The value you returned will be passed as the third argument to [`componentDidUpdate`.](#componentdidupdate)

Caveats {/*getsnapshotbeforeupdate-caveats*/}

- `getSnapshotBeforeUpdate` will not get called if [`shouldComponentUpdate`](#shouldcomponentupdate) is defined and returns `false`.

<Note>

At the moment, there is no equivalent to `getSnapshotBeforeUpdate` for function components. This use case is very uncommon, but if you have the need for it, for now you'll have to write a class component.

</Note>

`render()` {/*render*/}

The `render` method is the only required method in a class component.

The `render` method should specify what you want to appear on the screen, for example:

```js {4-6}
import { Component } from 'react';

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

React may call `render` at any moment, so you shouldn't assume that it runs at a particular time. Usually, the `render` method should return a piece of [JSX](/learn/writing-markup-with-jsx), but a few [other return types](#render-returns) (like strings) are supported. To calculate the returned JSX, the `render` method can read [`this.props`](#props), [`this.state`](#state), and [`this.context`](#context).

You should write the `render` method as a pure function, meaning that it should return the same result if props, state, and context are the same. It also shouldn't contain side effects (like setting up subscriptions) or interact with the browser APIs. Side effects should happen either in event handlers or methods like [`componentDidMount`.](#componentdidmount)

Parameters {/*render-parameters*/}

* `prevProps`: Props before the update. Compare `prevProps` to [`this.props`](#props) to determine what changed.

* `prevState`: State before the update. Compare `prevState` to [`this.state`](#state) to determine what changed.

Returns {/*render-returns*/}

`render` can return any valid React node. This includes React elements such as `<div />`, strings, numbers, [portals](/reference/react-dom/createPortal), empty nodes (`null`, `undefined`, `true`, and `false`), and arrays of React nodes.

Caveats {/*render-caveats*/}

- `render` should be written as a pure function of props, state, and context. It should not have side effects.

- `render` will not get called if [`shouldComponentUpdate`](#shouldcomponentupdate) is defined and returns `false`.

- When [Strict Mode](/reference/react/StrictMode) is on, React will call `render` twice in development and then throw away one of the results. This helps you notice the accidental side effects that need to be moved out of the `render` method.

- There is no one-to-one correspondence between the `render` call and the subsequent `componentDidMount` or `componentDidUpdate` call. Some of the `render` call results may be

discarded by React when it's beneficial.

`setState(nextState, callback?)` {/*setstate*/}

Call `setState` to update the state of your React component.

```js {8-10}
class Form extends Component {
state = {
name: 'Taylor',
};

handleNameChange = (e) => {
const newName = e.target.value;
this.setState({
name: newName
});
}

render() {
return (
<>
<input value={this.state.name} onChange={this.handleNameChange} />
<p>Hello, {this.state.name}.
</>
);
}
}
```

`setState` enqueues changes to the component state. It tells React that this component and its children need to re-render with the new state. This is the main way you'll update the user interface in response to interactions.

<Pitfall>

Calling `setState` **does not** change the current state in the already executing code:

```js {6}
function handleClick() {
console.log(this.state.name); // "Taylor"
```

```
this.setState({
name: 'Robin'
});
console.log(this.state.name); // Still "Taylor"!
}
```

It only affects what `this.state` will return starting from the *next* render.

</Pitfall>

You can also pass a function to `setState`. It lets you update state based on the previous state:

```js {2-6}
handleIncreaseAge = () => {
this.setState(prevState => {
return {
age: prevState.age + 1
};
});
}
```

You don't have to do this, but it's handy if you want to update state multiple times during the same event.

Parameters {/*setstate-parameters*/}

* `nextState`: Either an object or a function.

* If you pass an object as `nextState`, it will be shallowly merged into `this.state`.

* If you pass a function as `nextState`, it will be treated as an _updater function_. It must be pure, should take the pending state and props as arguments, and should return the object to be shallowly merged into `this.state`. React will put your updater function in a queue and re-render your component. During the next render, React will calculate the next state by applying all of the queued updaters to the previous state.

* **optional** `callback`: If specified, React will call the `callback` you've provided after the update is committed.

Returns {/*setstate-returns*/}

`setState` does not return anything.

Caveats {/*setstate-caveats*/}

- Think of `setState` as a *request* rather than an immediate command to update the component. When multiple components update their state in response to an event, React will batch their updates and re-render them together in a single pass at the end of the event. In the rare case that you need to force a particular state update to be applied synchronously, you may wrap it in [`flushSync`,](/reference/react-dom/flushSync) but this may hurt performance.

- `setState` does not update `this.state` immediately. This makes reading `this.state` right after calling `setState` a potential pitfall. Instead, use [`componentDidUpdate`](#componentdidupdate) or the setState `callback` argument, 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, you can pass a function to `nextState` as described above.

<Note>

Calling `setState` in class components is similar to calling a [`set` function](/reference/react/useState#setstate) in function components.

[See how to migrate.](#migrating-a-component-with-state-from-a-class-to-a-function)

</Note>

`shouldComponentUpdate(nextProps, nextState, nextContext)` {/*shouldcomponentupdate*/}

If you define `shouldComponentUpdate`, React will call it to determine whether a re-render can be skipped.

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.

```js {6-18}
class Rectangle extends Component {
state = {
isHovered: false
};

shouldComponentUpdate(nextProps, nextState) {
if (
nextProps.position.x === this.props.position.x &&
nextProps.position.y === this.props.position.y &&
nextProps.size.width === this.props.size.width &&
nextProps.size.height === this.props.size.height &&
nextState.isHovered === this.state.isHovered
) {
// Nothing has changed, so a re-render is unnecessary
```

return false;

}

return true;

}

// ...

}

```

React calls `shouldComponentUpdate` 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`](#forceupdate) is used.

#### Parameters {/*shouldcomponentupdate-parameters*/}

- `nextProps`: The next props that the component is about to render with. Compare `nextProps` to [`this.props`](#props) to determine what changed.

- `nextState`: The next state that the component is about to render with. Compare `nextState` to [`this.state`](#props) to determine what changed.

- `nextContext`: The next context that the component is about to render with. Compare `nextContext` to [`this.context`](#context) to determine what changed. Only available if you specify [`static contextType`](#static-contexttype) (modern) or [`static contextTypes`](#static-contexttypes) (legacy).

#### Returns {/*shouldcomponentupdate-returns*/}

Return `true` if you want the component to re-render. That's the default behavior.

Return `false` to tell React that re-rendering can be skipped.

#### Caveats {/*shouldcomponentupdate-caveats*/}

- This method *only* exists as a performance optimization. If your component breaks without it, fix that first.

- Consider using [`PureComponent`](/reference/react/PureComponent) instead of writing `shouldComponentUpdate` by hand. `PureComponent` shallowly compares props and state, and reduces the chance that you'll skip a necessary update.

- We do not recommend doing deep equality checks or using `JSON.stringify` in `shouldComponentUpdate`. It makes performance unpredictable and dependent on the data structure of every prop and state. In the best case, you risk introducing multi-second stalls to your application, and in the worst case you risk crashing it.

- Returning `false` does not prevent child components from re-rendering when *their* state changes.

- Returning `false` does not *guarantee* that the component will not re-render. React will use the return value as a hint but it may still choose to re-render your component if it makes sense to do for other reasons.
```

<Note>

Optimizing class components with `shouldComponentUpdate` is similar to optimizing function components with [`memo`.](/reference/react/memo) Function components also offer more granular optimization with [`useMemo`.](/reference/react/useMemo)

</Note>

`UNSAFE_componentWillMount()` {/*unsafe_componentwillmount*/}

If you define `UNSAFE_componentWillMount`, React will call it immediately after the [`constructor`.](#constructor) It only exists for historical reasons and should not be used in any new code. Instead, use one of the alternatives:

- To initialize state, declare [`state`](#state) as a class field or set `this.state` inside the [`constructor`.](#constructor)

- If you need to run a side effect or set up a subscription, move that logic to [`componentDidMount`](#componentdidmount) instead.

[See examples of migrating away from unsafe lifecycles.](https://legacy.reactjs.org/blog/2018/03/27/update-on-async-rendering.html#examples)

Parameters {/*unsafe_componentwillmount-parameters*/}

`UNSAFE_componentWillMount` does not take any parameters.

Returns {/*unsafe_componentwillmount-returns*/}

`UNSAFE_componentWillMount` should not return anything.

Caveats {/*unsafe_componentwillmount-caveats*/}

- `UNSAFE_componentWillMount` will not get called if the component implements [`static getDerivedStateFromProps`](#static-getderivedstatefromprops) or [`getSnapshotBeforeUpdate`.](#getsnapshotbeforeupdate)

- Despite its naming, `UNSAFE_componentWillMount` does not guarantee that the component *will* get mounted if your app uses modern React features like [`Suspense`.](/reference/react/Suspense) If a render attempt is suspended (for example, because the code for some child component has not loaded yet), React will throw the in-progress tree away and attempt to construct the component from scratch during the next attempt. This is why this method is "unsafe". Code that relies on mounting (like adding a subscription) should go into [`componentDidMount`.](#componentdidmount)

- `UNSAFE_componentWillMount` is the only lifecycle method that runs during [server rendering.](/reference/react-dom/server) For all practical purposes, it is identical to [`constructor`,](#constructor) so you should use the `constructor` for this type of logic instead.

<Note>

Calling [`setState`](#setstate) inside `UNSAFE_componentWillMount` in a class component to initialize state is equivalent to passing that state as the initial state to [`useState`](/reference/react/useState) in a function component.

</Note>

`UNSAFE_componentWillReceiveProps(nextProps, nextContext)` {/*unsafe_componentwillreceiveprops*/}

If you define `UNSAFE_componentWillReceiveProps`, React will call it when the component receives new props. It only exists for historical reasons and should not be used in any new code. Instead, use one of the alternatives:

- If you need to **run a side effect** (for example, fetch data, run an animation, or reinitialize a subscription) in response to prop changes, move that logic to [`componentDidUpdate`](#componentdidupdate) instead.

- If you need to **avoid re-computing some data only when a prop changes,** use a [memoization helper](https://legacy.reactjs.org/blog/2018/06/07/you-probably-dont-need-derived-state.html#what-about-memoization) instead.

- If you need to **"reset" some state when a prop changes,** consider either making a component [fully controlled](https://legacy.reactjs.org/blog/2018/06/07/you-probably-dont-need-derived-state.html#recommendation-fully-controlled-component) or [fully uncontrolled with a key](https://legacy.reactjs.org/blog/2018/06/07/you-probably-dont-need-derived-state.html#recommendation-fully-uncontrolled-component-with-a-key) instead.

- If you need to **"adjust" some state when a prop changes,** check whether you can compute all the necessary information from props alone during rendering. If you can't, use [`static getDerivedStateFromProps`](/reference/react/Component#static-getderivedstatefromprops) instead.

[See examples of migrating away from unsafe lifecycles.](https://legacy.reactjs.org/blog/2018/03/27/update-on-async-rendering.html#updating-state-based-on-props)

Parameters {/*unsafe_componentwillreceiveprops-parameters*/}

- `nextProps`: The next props that the component is about to receive from its parent component. Compare `nextProps` to [`this.props`](#props) to determine what changed.

- `nextContext`: The next props that the component is about to receive from the closest provider. Compare `nextContext` to [`this.context`](#context) to determine what changed. Only available if you specify [`static contextType`](#static-contexttype) (modern) or [`static contextTypes`](#static-contexttypes) (legacy).

Returns {/*unsafe_componentwillreceiveprops-returns*/}

`UNSAFE_componentWillReceiveProps` should not return anything.

Caveats {/*unsafe_componentwillreceiveprops-caveats*/}

- `UNSAFE_componentWillReceiveProps` will not get called if the component implements [`static getDerivedStateFromProps`](#static-getderivedstatefromprops) or [`getSnapshotBeforeUpdate`.](#getsnapshotbeforeupdate)

- Despite its naming, `UNSAFE_componentWillReceiveProps` does not guarantee that the component *will* receive those props if your app uses modern React features like [`Suspense`.](/reference/react/Suspense) If a render attempt is suspended (for example, because the code for some child component has not loaded yet), React will throw the in-progress tree away and attempt to construct the component from scratch during the next attempt. By the time of the next render attempt, the props might be different. This is why this method is "unsafe". Code that should run only for committed updates (like resetting a subscription) should go into [`componentDidUpdate`.](#componentdidupdate)

- `UNSAFE_componentWillReceiveProps` does not mean that the component has received *different* props than the last time. You need to compare `nextProps` and `this.props` yourself to check if something changed.

- React doesn't call `UNSAFE_componentWillReceiveProps` with initial props during mounting. It only calls this method if some of component's props are going to be updated. For example, calling [`setState`](#setstate) doesn't generally trigger `UNSAFE_componentWillReceiveProps` inside the same component.

<Note>

Calling [`setState`](#setstate) inside `UNSAFE_componentWillReceiveProps` in a class component to "adjust" state is equivalent to [calling the `set` function from `useState` during rendering](/reference/react/useState#storing-information-from-previous-renders) in a function component.

</Note>

`UNSAFE_componentWillUpdate(nextProps, nextState)` {/*unsafe_componentwillupdate*/}

If you define `UNSAFE_componentWillUpdate`, React will call it before rendering with the new props or state. It only exists for historical reasons and should not be used in any new code. Instead, use one of the alternatives:

- If you need to run a side effect (for example, fetch data, run an animation, or reinitialize a subscription) in response to prop or state changes, move that logic to [`componentDidUpdate`](#componentdidupdate) instead.

- If you need to read some information from the DOM (for example, to save the current scroll position) so that you can use it in [`componentDidUpdate`](#componentdidupdate) later, read it inside [`getSnapshotBeforeUpdate`](#getsnapshotbeforeupdate) instead.

[See examples of migrating away from unsafe lifecycles.](https://legacy.reactjs.org/blog/2018/03/27/update-on-async-rendering.html#examples)

Parameters {/*unsafe_componentwillupdate-parameters*/}

- `nextProps`: The next props that the component is about to render with. Compare `nextProps` to [`this.props`](#props) to determine what changed.

- `nextState`: The next state that the component is about to render with. Compare `nextState` to [`this.state`](#state) to determine what changed.

Returns {/*unsafe_componentwillupdate-returns*/}

`UNSAFE_componentWillUpdate` should not return anything.

Caveats {/*unsafe_componentwillupdate-caveats*/}

- `UNSAFE_componentWillUpdate` will not get called if [`shouldComponentUpdate`](#shouldcomponentupdate) is defined and returns `false`.

- `UNSAFE_componentWillUpdate` will not get called if the component implements [`static getDerivedStateFromProps`](#static-getderivedstatefromprops) or [`getSnapshotBeforeUpdate`.](#getsnapshotbeforeupdate)

- It's not supported to call [`setState`](#setstate) (or any method that leads to `setState` being called, like dispatching a Redux action) during `componentWillUpdate`.

- Despite its naming, `UNSAFE_componentWillUpdate` does not guarantee that the component *will* update if your app uses modern React features like [`Suspense`.](/reference/react/Suspense) If a render attempt is suspended (for example, because the code for some child component has not loaded yet), React will throw the in-progress tree away and attempt to construct the component from scratch during the next attempt. By the time of the next render attempt, the props and state might be different. This is why this method is "unsafe". Code that should run only for committed updates (like resetting a subscription) should go into [`componentDidUpdate`.](#componentdidupdate)

- `UNSAFE_componentWillUpdate` does not mean that the component has received *different* props or state than the last time. You need to compare `nextProps` with `this.props` and `nextState` with `this.state` yourself to check if something changed.

- React doesn't call `UNSAFE_componentWillUpdate` with initial props and state during mounting.

<Note>

There is no direct equivalent to `UNSAFE_componentWillUpdate` in function components.

</Note>

`static childContextTypes` {/*static-childcontexttypes*/}

<Deprecated>

This API will be removed in a future major version of React. [Use `static contextType` instead.](#static-contexttype)

</Deprecated>

Lets you specify which [legacy context](https://reactjs.org/docs/legacy-context.html) is provided by this component.

`static contextTypes` {/*static-contexttypes*/}

<Deprecated>

This API will be removed in a future major version of React. [Use `static contextType` instead.](#static-contexttype)

</Deprecated>

Lets you specify which [legacy context](https://reactjs.org/docs/legacy-context.html) is consumed by this component.

`static contextType` {/*static-contexttype*/}

If you want to read [`this.context`](#context-instance-field) from your class component, you must specify which context it needs to read. The context you specify as the `static contextType` must be a value previously created by [`createContext`.](/reference/react/createContext)

```js {2}
class Button extends Component {
static contextType = ThemeContext;

render() {
const theme = this.context;
const className = 'button-' + theme;
return (
<button className={className}>
{this.props.children}
</button>
);
}
}
```

<Note>

Reading `this.context` in class components is equivalent to [`useContext`](/reference/react/useContext) in function components.

[See how to migrate.](#migrating-a-component-with-context-from-a-class-to-a-function)

</Note>

`static defaultProps` {/*static-defaultprops*/}

You can define `static defaultProps` to set the default props for the class. They will be used for `undefined` and missing props, but not for `null` props.

For example, here is how you define that the `color` prop should default to `'blue'`:

```js {2-4}
class Button extends Component {
static defaultProps = {
color: 'blue'
};
render() {
return <button className={this.props.color}>click me</button>;
}
}
```

If the `color` prop is not provided or is `undefined`, it will be set by default to `'blue'`:

```js
<>
{/* this.props.color is "blue" */}
<Button />
{/* this.props.color is "blue" */}
<Button color={undefined} />
{/* this.props.color is null */}
<Button color={null} />
{/* this.props.color is "red" */}
<Button color="red" />
</>
```

<Note>

Defining `defaultProps` in class components is similar to using [default values](/learn/passing-props-to-a-component#specifying-a-default-value-for-a-prop) in function components.

</Note>

`static propTypes` {/*static-proptypes*/}

You can define `static propTypes` together with the [`prop-types`](https://www.npmjs.com/package/prop-types) library to declare the types of the props accepted by your component. These types will be checked during rendering and in development only.

```js
import PropTypes from 'prop-types';

class Greeting extends React.Component {
static propTypes = {
name: PropTypes.string
};

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

<Note>

We recommend using [TypeScript](https://www.typescriptlang.org/) instead of checking prop types at runtime.

</Note>

`static getDerivedStateFromError(error)` {/*static-getderivedstatefromerror*/}

If you define `static getDerivedStateFromError`, React will call it when a child component (including distant children) throws an error during rendering. This lets you display an error message instead of clearing the UI.

Typically, it is used together with [`componentDidCatch`](#componentdidcatch) which lets you send the error report to some analytics service. A component with these methods is called an *error boundary.*

[See an example.](#catching-rendering-errors-with-an-error-boundary)

Parameters {/*static-getderivedstatefromerror-parameters*/}

* `error`: The error that was thrown. In practice, it will usually be an instance of [`Error`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error) but this is not guaranteed because JavaScript allows to [`throw`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/throw) any

value, including strings or even `null`.

Returns {/*static-getderivedstatefromerror-returns*/}

`static getDerivedStateFromError` should return the state telling the component to display the error message.

Caveats {/*static-getderivedstatefromerror-caveats*/}

* `static getDerivedStateFromError` should be a pure function. If you want to perform a side effect (for example, to call an analytics service), you need to also implement [`componentDidCatch`.](#componentdidcatch)

<Note>

There is no direct equivalent for `static getDerivedStateFromError` in function components yet. If you'd like to avoid creating class components, write a single `ErrorBoundary` component like above and use it throughout your app. Alternatively, use the [`react-error-boundary`](https://github.com/bvaughn/react-error-boundary) package which does that.

</Note>

`static getDerivedStateFromProps(props, state)` {/*static-getderivedstatefromprops*/}

If you define `static getDerivedStateFromProps`, React will call it right before calling [`render`,](#render) 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](https://legacy.reactjs.org/blog/2018/06/07/you-probably-dont-need-derived-state.html#when-to-use-derived-state) where the state depends on changes in props over time. For example, this `Form` component resets the `email` state when the `userID` prop changes:

```js {7-18}
class Form extends Component {
state = {
email: this.props.defaultEmail,
prevUserID: 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.prevUserID) {
return {
```

prevUserID: props.userID,

email: props.defaultEmail

};

}

return null;

}

// ...

}

```

Note that this pattern requires you to keep a previous value of the prop (like `userID`) in state (like `prevUserID`).

<Pitfall>

Deriving state leads to verbose code and makes your components difficult to think about. [Make sure you're familiar with simpler alternatives:](https://legacy.reactjs.org/blog/2018/06/07/you-probably-dont-need-derived-state.html)

- If you need to **perform a side effect** (for example, data fetching or an animation) in response to a change in props, use [`componentDidUpdate`](#componentdidupdate) method instead.

- If you want to **re-compute some data only when a prop changes,** [use a memoization helper instead.](https://legacy.reactjs.org/blog/2018/06/07/you-probably-dont-need-derived-state.html#what-about-memoization)

- If you want to **"reset" some state when a prop changes,** consider either making a component [fully controlled](https://legacy.reactjs.org/blog/2018/06/07/you-probably-dont-need-derived-state.html#recommendation-fully-controlled-component) or [fully uncontrolled with a key](https://legacy.reactjs.org/blog/2018/06/07/you-probably-dont-need-derived-state.html#recommendation-fully-uncontrolled-component-with-a-key) instead.

</Pitfall>

#### Parameters {/*static-getderivedstatefromprops-parameters*/}

- `props`: The next props that the component is about to render with.

- `state`: The next state that the component is about to render with.

#### Returns {/*static-getderivedstatefromprops-returns*/}

`static getDerivedStateFromProps` return an object to update the state, or `null` to update nothing.

#### Caveats {/*static-getderivedstatefromprops-caveats*/}

- This method is fired on *every* render, regardless of the cause. This is different from [`UNSAFE_componentWillReceiveProps`](#unsafe_cmoponentwillreceiveprops), which only fires when the parent causes a re-render and not as a result of a local `setState`.
```

- This method doesn't have access to the component instance. If you'd like, you can reuse some code between `static getDerivedStateFromProps` and the other class methods by extracting pure functions of the component props and state outside the class definition.

<Note>

Implementing `static getDerivedStateFromProps` in a class component is equivalent to [calling the `set` function from `useState` during rendering](/reference/react/useState#storing-information-from-previous-renders) in a function component.

</Note>

Usage {/*usage*/}

Defining a class component {/*defining-a-class-component*/}

To define a React component as a class, extend the built-in `Component` class and define a [`render` method:](#render)

```js
import { Component } from 'react';

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

React will call your [`render`](#render) method whenever it needs to figure out what to display on the screen. Usually, you will return some [JSX](/learn/writing-markup-with-jsx) from it. Your `render` method should be a [pure function:](https://en.wikipedia.org/wiki/Pure_function) it should only calculate the JSX.

Similarly to [function components,](/learn/your-first-component#defining-a-component) a class component can [receive information by props](/learn/your-first-component#defining-a-component) from its parent component. However, the syntax for reading props is different. For example, if the parent component renders `<Greeting name="Taylor" />`, then you can read the `name` prop from [`this.props`](#props), like `this.props.name`:

<Sandpack>

```js
import { Component } from 'react';

class Greeting extends Component {
```

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

export default function App() {
return (
<>
<Greeting name="Sara" />
<Greeting name="Cahal" />
<Greeting name="Edite" />
</>
);
}
```

</Sandpack>

Note that Hooks (functions starting with `use`, like [`useState`](/reference/react/useState)) are not supported inside class components.

<Pitfall>

We recommend defining components as functions instead of classes. [See how to migrate.](#migrating-a-simple-component-from-a-class-to-a-function)

</Pitfall>

Adding state to a class component {/*adding-state-to-a-class-component*/}

To add [state](/learn/state-a-components-memory) to a class, assign an object to a property called [`state`](#state). To update state, call [`this.setState`](#setstate).

<Sandpack>

```js
import { Component } from 'react';

export default class Counter extends Component {
state = {
name: 'Taylor',
age: 42,
```

```
};

handleNameChange = (e) => {
this.setState({
name: e.target.value
});
}

handleAgeChange = () => {
this.setState({
age: this.state.age + 1
});
};

render() {
return (
<>
<input
value={this.state.name}
onChange={this.handleNameChange}
/>
<button onClick={this.handleAgeChange}>
Increment age
</button>
<p>Hello, {this.state.name}. You are {this.state.age}.</p>
</>
);
}
}
```

```css
button { display: block; margin-top: 10px; }
```

</Sandpack>

<Pitfall>

We recommend defining components as functions instead of classes. [See how to migrate.](#migrating-a-component-with-state-from-a-class-to-a-function)

</Pitfall>

Adding lifecycle methods to a class component {/*adding-lifecycle-methods-to-a-class-component*/}

There are a few special methods you can define on your class.

If you define the [`componentDidMount`](#componentdidmount) method, React will call it when your component is added *(mounted)* to the screen. React will call [`componentDidUpdate`](#componentdidupdate) after your component re-renders due to changed props or state. React will call [`componentWillUnmount`](#componentwillunmount) after your component has been removed *(unmounted)* from the screen.

If you implement `componentDidMount`, you usually need to implement all three lifecycles to avoid bugs. For example, if `componentDidMount` reads some state or props, you also have to implement `componentDidUpdate` to handle their changes, and `componentWillUnmount` to clean up whatever `componentDidMount` was doing.

For example, this `ChatRoom` component keeps a chat connection synchronized with props and state:

<Sandpack>

```js App.js
import { useState } from 'react';
import ChatRoom from './ChatRoom.js';

export default function App() {
const [roomId, setRoomId] = useState('general');
const [show, setShow] = useState(false);
return (
<>
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
```

```
</select>
</label>
<button onClick={() => setShow(!show)}>
{show ? 'Close chat' : 'Open chat'}
</button>
{show && <hr />}
{show && <ChatRoom roomId={roomId} />}
</>
);
}
```

```js ChatRoom.js active
import { Component } from 'react';
import { createConnection } from './chat.js';

export default class ChatRoom extends Component {
state = {
serverUrl: 'https://localhost:1234'
};

componentDidMount() {
this.setupConnection();
}

componentDidUpdate(prevProps, prevState) {
if (
this.props.roomId !== prevProps.roomId ||
this.state.serverUrl !== prevState.serverUrl
) {
this.destroyConnection();
this.setupConnection();
}
}

componentWillUnmount() {
this.destroyConnection();
}
```

````
setupConnection() {
this.connection = createConnection(
this.state.serverUrl,
this.props.roomId
);
this.connection.connect();
}

destroyConnection() {
this.connection.disconnect();
this.connection = null;
}

render() {
return (
<>
<label>
Server URL:{' '}
<input
value={this.state.serverUrl}
onChange={e => {
this.setState({
serverUrl: e.target.value
});
}}
/>
</label>
<h1>Welcome to the {this.props.roomId} room!</h1>
</>
);
}
}
```

```js chat.js
export function createConnection(serverUrl, roomId) {
// A real implementation would actually connect to the server
````

```
return {
connect() {
console.log('■ Connecting to "' + roomId + '" room at ' + serverUrl + '...');
},
disconnect() {
console.log('■ Disconnected from "' + roomId + '" room at ' + serverUrl);
}
};
}
```

```css
input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }
```

</Sandpack>

Note that in development when [Strict Mode](/reference/react/StrictMode) is on, React will call `componentDidMount`, immediately call `componentWillUnmount`, and then call `componentDidMount` again. This helps you notice if you forgot to implement `componentWillUnmount` or if its logic doesn't fully "mirror" what `componentDidMount` does.

<Pitfall>

We recommend defining components as functions instead of classes. [See how to migrate.](#migrating-a-component-with-lifecycle-methods-from-a-class-to-a-function)

</Pitfall>

Catching rendering errors with an error boundary
{/*catching-rendering-errors-with-an-error-boundary*/}

By default, if your application throws an error during rendering, React will remove its UI from the screen. To prevent this, you can wrap a part of your UI into an *error boundary*. An error boundary is a special component that lets you display some fallback UI instead of the part that crashed--for example, an error message.

To implement an error boundary component, you need to provide [`static getDerivedStateFromError`](#static-getderivedstatefromerror) which lets you update state in response to an error and display an error message to the user. You can also optionally implement [`componentDidCatch`](#componentdidcatch) to add some extra logic, for example, to log the error to an analytics service.

```js {7-10,12-19}
```

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

static getDerivedStateFromError(error) {
// Update state so the next render will show the fallback UI.
return { hasError: true };
}

componentDidCatch(error, info) {
// Example "componentStack":
// in ComponentThatThrows (created by App)
// in ErrorBoundary (created by App)
// in div (created by App)
// in App
logErrorToMyService(error, info.componentStack);
}

render() {
if (this.state.hasError) {
// You can render any custom fallback UI
return this.props.fallback;
}

return this.props.children;
}
}
```

Then you can wrap a part of your component tree with it:

```js {1,3}
<ErrorBoundary fallback={<p>Something went wrong</p>}>
<Profile />
</ErrorBoundary>
```

If `Profile` or its child component throws an error, `ErrorBoundary` will "catch" that error, display a fallback UI with the error message you've provided, and send a production error report to your error reporting service.

You don't need to wrap every component into a separate error boundary. When you think about the [granularity of error boundaries,](https://www.brandondail.com/posts/fault-tolerance-react) consider where it makes sense to display an error message. For example, in a messaging app, it makes sense to place an error boundary around the list of conversations. It also makes sense to place one around every individual message. However, it wouldn't make sense to place a boundary around every avatar.

<Note>

There is currently no way to write an error boundary as a function component. However, you don't have to write the error boundary class yourself. For example, you can use [`react-error-boundary`](https://github.com/bvaughn/react-error-boundary) instead.

</Note>

Alternatives {/*alternatives*/}

Migrating a simple component from a class to a function {/*migrating-a-simple-component-from-a-class-to-a-function*/}

Typically, you will [define components as functions](/learn/your-first-component#defining-a-component) instead.

For example, suppose you're converting this `Greeting` class component to a function:

<Sandpack>

```js
import { Component } from 'react';

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

export default function App() {
return (
<>
<Greeting name="Sara" />
<Greeting name="Cahal" />
<Greeting name="Edite" />
```

</>

);

}

```

</Sandpack>

Define a function called `Greeting`. This is where you will move the body of your `render` function.

```js

function Greeting() {

// ... move the code from the render method here ...

}

```

Instead of `this.props.name`, define the `name` prop [using the destructuring syntax](/learn/passing-props-to-a-component) and read it directly:

```js

function Greeting({ name }) {

return <h1>Hello, {name}!</h1>;

}

```

Here is a complete example:

<Sandpack>

```js

function Greeting({ name }) {

return <h1>Hello, {name}!</h1>;

}

export default function App() {

return (

<>

<Greeting name="Sara" />

<Greeting name="Cahal" />

<Greeting name="Edite" />

</>

);

}

```

</Sandpack>

Migrating a component with state from a class to a function
{/*migrating-a-component-with-state-from-a-class-to-a-function*/}

Suppose you're converting this `Counter` class component to a function:

<Sandpack>

```js
import { Component } from 'react';

export default class Counter extends Component {
state = {
name: 'Taylor',
age: 42,
};

handleNameChange = (e) => {
this.setState({
name: e.target.value
});
}

handleAgeChange = (e) => {
this.setState({
age: this.state.age + 1
});
};

render() {
return (
<>
<input
value={this.state.name}
onChange={this.handleNameChange}
/>
<button onClick={this.handleAgeChange}>
```
```

Increment age

</button>

Hello, {this.state.name}. You are {this.state.age}.

</>

);

}

}

```

```css

button { display: block; margin-top: 10px; }

```

</Sandpack>

Start by declaring a function with the necessary [state variables:](/reference/react/useState#adding-state-to-a-component)

```js {4-5}

import { useState } from 'react';

function Counter() {

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

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

// ...

```

Next, convert the event handlers:

```js {5-7,9-11}

function Counter() {

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

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

function handleNameChange(e) {

setName(e.target.value);

}

function handleAgeChange() {

setAge(age + 1);

}

// ...

```

Finally, replace all references starting with `this` with the variables and functions you defined in your component. For example, replace `this.state.age` with `age`, and replace `this.handleNameChange` with `handleNameChange`.

Here is a fully converted component:

<Sandpack>

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

export default function Counter() {
const [name, setName] = useState('Taylor');
const [age, setAge] = useState(42);

function handleNameChange(e) {
setName(e.target.value);
}

function handleAgeChange() {
setAge(age + 1);
}

return (
<>
<input
value={name}
onChange={handleNameChange}
/>
<button onClick={handleAgeChange}>
Increment age
</button>
<p>Hello, {name}. You are {age}.</p>
</>
)
}
```

```css
button { display: block; margin-top: 10px; }
```
```

```

</Sandpack>

---

### Migrating a component with lifecycle methods from a class to a function
{/*migrating-a-component-with-lifecycle-methods-from-a-class-to-a-function*/}

Suppose you're converting this `ChatRoom` class component with lifecycle methods to a function:

<Sandpack>

```js App.js
import { useState } from 'react';
import ChatRoom from './ChatRoom.js';

export default function App() {
const [roomId, setRoomId] = useState('general');
const [show, setShow] = useState(false);
return (
<>
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
</select>
</label>
<button onClick={() => setShow(!show)}>
{show ? 'Close chat' : 'Open chat'}
</button>
{show && <hr />}
{show && <ChatRoom roomId={roomId} />}
</>
);
```
```

```
}
```

```js ChatRoom.js active
import { Component } from 'react';
import { createConnection } from './chat.js';

export default class ChatRoom extends Component {
state = {
serverUrl: 'https://localhost:1234'
};

componentDidMount() {
this.setupConnection();
}

componentDidUpdate(prevProps, prevState) {
if (
this.props.roomId !== prevProps.roomId ||
this.state.serverUrl !== prevState.serverUrl
) {
this.destroyConnection();
this.setupConnection();
}
}

componentWillUnmount() {
this.destroyConnection();
}

setupConnection() {
this.connection = createConnection(
this.state.serverUrl,
this.props.roomId
);
this.connection.connect();
}

destroyConnection() {
this.connection.disconnect();
```

```
this.connection = null;
}
render() {
return (
<>
<label>
Server URL:{' '}
<input
value={this.state.serverUrl}
onChange={e => {
this.setState({
serverUrl: e.target.value
});
}}
/>
</label>
<h1>Welcome to the {this.props.roomId} room!</h1>
</>
);
}
}
```

```js chat.js
export function createConnection(serverUrl, roomId) {
// A real implementation would actually connect to the server
return {
connect() {
console.log('■ Connecting to "' + roomId + '" room at ' + serverUrl + '...');
},
disconnect() {
console.log('■ Disconnected from "' + roomId + '" room at ' + serverUrl);
}
};
}
```

```css
input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }
```

</Sandpack>

First, verify that your [`componentWillUnmount`](#componentwillunmount) does the opposite of [`componentDidMount`.](#componentdidmount) In the above example, that's true: it disconnects the connection that `componentDidMount` sets up. If such logic is missing, add it first.

Next, verify that your [`componentDidUpdate`](#componentdidupdate) method handles changes to any props and state you're using in `componentDidMount`. In the above example, `componentDidMount` calls `setupConnection` which reads `this.state.serverUrl` and `this.props.roomId`. This is why `componentDidUpdate` checks whether `this.state.serverUrl` and `this.props.roomId` have changed, and resets the connection if they did. If your `componentDidUpdate` logic is missing or doesn't handle changes to all relevant props and state, fix that first.

In the above example, the logic inside the lifecycle methods connects the component to a system outside of React (a chat server). To connect a component to an external system, [describe this logic as a single Effect:](/reference/react/useEffect#connecting-to-an-external-system)

```js {6-12}
import { useState, useEffect } from 'react';

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

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

// ...
}
```

This [`useEffect`](/reference/react/useEffect) call is equivalent to the logic in the lifecycle methods above. If your lifecycle methods do multiple unrelated things, [split them into multiple independent Effects.](/learn/removing-effect-dependencies#is-your-effect-doing-several-unrelated-things) Here is a complete example you can play with:

<Sandpack>

```js App.js
import { useState } from 'react';
import ChatRoom from './ChatRoom.js';

export default function App() {
const [roomId, setRoomId] = useState('general');
const [show, setShow] = useState(false);
return (
<>
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
</select>
</label>
<button onClick={() => setShow(!show)}>
{show ? 'Close chat' : 'Open chat'}
</button>
{show && <hr />}
{show && <ChatRoom roomId={roomId} />}
</>
);
}
```

```js ChatRoom.js active
import { useState, useEffect } from 'react';
import { createConnection } from './chat.js';

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

useEffect(() => {
```

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

return (
<>
<label>
Server URL:{' '}
<input
value={serverUrl}
onChange={e => setServerUrl(e.target.value)}
/>
</label>
<h1>Welcome to the {roomId} room!</h1>
</>
);
}
```

```js chat.js
export function createConnection(serverUrl, roomId) {
// A real implementation would actually connect to the server
return {
connect() {
console.log('■ Connecting to "' + roomId + '" room at ' + serverUrl + '...');
},
disconnect() {
console.log('■ Disconnected from "' + roomId + '" room at ' + serverUrl);
}
};
}
```

```css
```

input { display: block; margin-bottom: 20px; }

button { margin-left: 10px; }

```

</Sandpack>

<Note>

If your component does not synchronize with any external systems, [you might not need an Effect.](/learn/you-might-not-need-an-effect)

</Note>

---

### Migrating a component with context from a class to a function {/*migrating-a-component-with-context-from-a-class-to-a-function*/}

In this example, the `Panel` and `Button` class components read [context](/learn/passing-data-deeply-with-context) from [`this.context`:](#context)

<Sandpack>

```js
import { createContext, Component } from 'react';

const ThemeContext = createContext(null);

class Panel extends Component {

static contextType = ThemeContext;

render() {

const theme = this.context;

const className = 'panel-' + theme;

return (

<section className={className}>

<h1>{this.props.title}</h1>

{this.props.children}

</section>

);

}

}

class Button extends Component {

static contextType = ThemeContext;

```
render() {
const theme = this.context;
const className = 'button-' + theme;
return (
<button className={className}>
{this.props.children}
</button>
);
}
}

function Form() {
return (
<Panel title="Welcome">
<Button>Sign up</Button>
<Button>Log in</Button>
</Panel>
);
}

export default function MyApp() {
return (
<ThemeContext.Provider value="dark">
<Form />
</ThemeContext.Provider>
)
}
```

```css
.panel-light,
.panel-dark {
border: 1px solid black;
border-radius: 4px;
padding: 20px;
}
.panel-light {
```

color: #222;
background: #fff;
}

.panel-dark {
color: #fff;
background: rgb(23, 32, 42);
}

.button-light,
.button-dark {
border: 1px solid #777;
padding: 5px;
margin-right: 10px;
margin-top: 10px;
}

.button-dark {
background: #222;
color: #fff;
}

.button-light {
background: #fff;
color: #222;
}
```

</Sandpack>

When you convert them to function components, replace `this.context` with [`useContext`](/reference/react/useContext) calls:

<Sandpack>

```js
import { createContext, useContext } from 'react';

const ThemeContext = createContext(null);

function Panel({ title, children }) {
const theme = useContext(ThemeContext);

```
const className = 'panel-' + theme;
return (
<section className={className}>
<h1>{title}</h1>
{children}
</section>
)
}

function Button({ children }) {
const theme = useContext(ThemeContext);
const className = 'button-' + theme;
return (
<button className={className}>
{children}
</button>
);
}

function Form() {
return (
<Panel title="Welcome">
<Button>Sign up</Button>
<Button>Log in</Button>
</Panel>
);
}

export default function MyApp() {
return (
<ThemeContext.Provider value="dark">
<Form />
</ThemeContext.Provider>
)
}
```

```css
```

```
.panel-light,
.panel-dark {
border: 1px solid black;
border-radius: 4px;
padding: 20px;
}
.panel-light {
color: #222;
background: #fff;
}

.panel-dark {
color: #fff;
background: rgb(23, 32, 42);
}

.button-light,
.button-dark {
border: 1px solid #777;
padding: 5px;
margin-right: 10px;
margin-top: 10px;
}

.button-dark {
background: #222;
color: #fff;
}

.button-light {
background: #fff;
color: #222;
}
```

</Sandpack>

---
title: useTransition

<Intro>

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

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

</Intro>

<InlineToc />

Reference {/*reference*/}

`useTransition()` {/*usetransition*/}

Call `useTransition` at the top level of your component to mark some state updates as transitions.

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

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

[See more examples below.](#usage)

Parameters {/*parameters*/}

`useTransition` does not take any parameters.

Returns {/*returns*/}

`useTransition` returns an array with exactly two items:

1. The `isPending` flag that tells you whether there is a pending transition.
2. The [`startTransition` function](#starttransition) that lets you mark a state update as a transition.

`startTransition` function {/*starttransition*/}

The `startTransition` function returned by `useTransition` lets you mark a state update as a transition.

```js {6,8}
```

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

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

Parameters {/*starttransition-parameters*/}

* `scope`: A function that updates some state by calling one or more [`set` functions.](/reference/react/useState#setstate) React immediately calls `scope` with no parameters and marks all state updates scheduled synchronously during the `scope` function call as transitions. They will be [non-blocking](#marking-a-state-update-as-a-non-blocking-transition) and [will not display unwanted loading indicators.](#preventing-unwanted-loading-indicators)

Returns {/*starttransition-returns*/}

`startTransition` does not return anything.

Caveats {/*starttransition-caveats*/}

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

* You can wrap an update into a transition only if you have access to the `set` function of that state. If you want to start a transition in response to some prop or a custom Hook value, try [`useDeferredValue`](/reference/react/useDeferredValue) instead.

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

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

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

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

Usage {/*usage*/}

Marking a state update as a non-blocking transition
{/*marking-a-state-update-as-a-non-blocking-transition*/}

Call `useTransition` at the top level of your component to mark state updates as non-blocking *transitions*.

```js [[1, 4, "isPending"], [2, 4, "startTransition"]]
import { useState, useTransition } from 'react';

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

`useTransition` returns an array with exactly two items:

1. The <CodeStep step={1}>`isPending` flag</CodeStep> that tells you whether there is a pending transition.
2. The <CodeStep step={2}>`startTransition` function</CodeStep> that lets you mark a state update as a transition.

You can then mark a state update as a transition like this:

```js {6,8}
function TabContainer() {
const [isPending, startTransition] = useTransition();
const [tab, setTab] = useState('about');

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

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

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

<Recipes titleText="The difference between useTransition and regular state updates" titleId="examples">

Updating the current tab in a transition {/*updating-the-current-tab-in-a-transition*/}

In this example, the "Posts" tab is **artificially slowed down** so that it takes at least a second to render.

Click "Posts" and then immediately click "Contact". Notice that this interrupts the slow render of "Posts". The "Contact" tab shows immediately. Because this state update is marked as a transition, a slow re-render did not freeze the user interface.

<Sandpack>

```js
import { useState, useTransition } from 'react';
import TabButton from './TabButton.js';
import AboutTab from './AboutTab.js';
import PostsTab from './PostsTab.js';
import ContactTab from './ContactTab.js';

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

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

return (
<>
<TabButton
isActive={tab === 'about'}
onClick={() => selectTab('about')}
>
About
</TabButton>
```

<TabButton
isActive={tab === 'posts'}
onClick={() => selectTab('posts')}
>
Posts (slow)
</TabButton>
<TabButton
isActive={tab === 'contact'}
onClick={() => selectTab('contact')}
>
Contact
</TabButton>
<hr />
{tab === 'about' && <AboutTab />}
{tab === 'posts' && <PostsTab />}
{tab === 'contact' && <ContactTab />}
</>
);
}
```

```js TabButton.js
import { useTransition } from 'react';

export default function TabButton({ children, isActive, onClick }) {
if (isActive) {
return {children}
}
return (
<button onClick={() => {
onClick();
}}>
{children}
</button>
)
}

```

```js AboutTab.js
export default function AboutTab() {
return (
<p>Welcome to my profile!</p>
);
}
```

```js PostsTab.js
import { memo } from 'react';

const PostsTab = memo(function PostsTab() {
// Log once. The actual slowdown is inside SlowPost.
console.log('[ARTIFICIALLY SLOW] Rendering 500 <SlowPost />');

let items = [];
for (let i = 0; i < 500; i++) {
items.push(<SlowPost key={i} index={i} />);
}
return (
<ul className="items">
{items}
</ul>
);
});

function SlowPost({ index }) {
let startTime = performance.now();
while (performance.now() - startTime < 1) {
// Do nothing for 1 ms per item to emulate extremely slow code
}

return (
<li className="item">
Post #{index + 1}
</li>
);
```
```

}

export default PostsTab;
```

```js ContactTab.js
export default function ContactTab() {
return (
<>
<p>
You can find me online here:
</p>
<ul>
<li>admin@mysite.com</li>
<li>+123456789</li>
</ul>
</>
);
}
```

```css
button { margin-right: 10px }
b { display: inline-block; margin-right: 10px; }
```

</Sandpack>

<Solution />

Updating the current tab without a transition {/*updating-the-current-tab-without-a-transition*/}

In this example, the "Posts" tab is also **artificially slowed down** so that it takes at least a second to render. Unlike in the previous example, this state update is **not a transition.**

Click "Posts" and then immediately click "Contact". Notice that the app freezes while rendering the slowed down tab, and the UI becomes unresponsive. This state update is not a transition, so a slow re-render freezed the user interface.

<Sandpack>

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

```
import TabButton from './TabButton.js';
import AboutTab from './AboutTab.js';
import PostsTab from './PostsTab.js';
import ContactTab from './ContactTab.js';

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

function selectTab(nextTab) {
setTab(nextTab);
}

return (
<>
<TabButton
isActive={tab === 'about'}
onClick={() => selectTab('about')}
>
About
</TabButton>
<TabButton
isActive={tab === 'posts'}
onClick={() => selectTab('posts')}
>
Posts (slow)
</TabButton>
<TabButton
isActive={tab === 'contact'}
onClick={() => selectTab('contact')}
>
Contact
</TabButton>
<hr />
{tab === 'about' && <AboutTab />}
{tab === 'posts' && <PostsTab />}
{tab === 'contact' && <ContactTab />}
</>
```

```
);
}
```

```js TabButton.js
import { useTransition } from 'react';

export default function TabButton({ children, isActive, onClick }) {
if (isActive) {
return <b>{children}</b>
}
return (
<button onClick={() => {
onClick();
}}>
{children}
</button>
)
}
```

```js AboutTab.js
export default function AboutTab() {
return (
<p>Welcome to my profile!</p>
);
}
```

```js PostsTab.js
import { memo } from 'react';

const PostsTab = memo(function PostsTab() {
// Log once. The actual slowdown is inside SlowPost.
console.log('[ARTIFICIALLY SLOW] Rendering 500 <SlowPost />');

let items = [];
for (let i = 0; i < 500; i++) {
items.push(<SlowPost key={i} index={i} />);
```

```
}
return (
<ul className="items">
{items}
</ul>
);
});

function SlowPost({ index }) {
let startTime = performance.now();
while (performance.now() - startTime < 1) {
// Do nothing for 1 ms per item to emulate extremely slow code
}

return (
<li className="item">
Post #{index + 1}
</li>
);
}

export default PostsTab;
```

```js ContactTab.js
export default function ContactTab() {
return (
<>
<p>
You can find me online here:
</p>
<ul>
<li>admin@mysite.com</li>
<li>+123456789</li>
</ul>
</>
);
}
```

```

```css
button { margin-right: 10px }
b { display: inline-block; margin-right: 10px; }
```

</Sandpack>

<Solution />

</Recipes>

Updating the parent component in a transition {/*updating-the-parent-component-in-a-transition*/}

You can update a parent component's state from the `useTransition` call, too. For example, this `TabButton` component wraps its `onClick` logic in a transition:

```js {8-10}
export default function TabButton({ children, isActive, onClick }) {
const [isPending, startTransition] = useTransition();
if (isActive) {
return <b>{children}</b>
}
return (
<button onClick={() => {
startTransition(() => {
onClick();
});
}}>
{children}
</button>
);
}
```

Because the parent component updates its state inside the `onClick` event handler, that state update gets marked as a transition. This is why, like in the earlier example, you can click on "Posts" and then immediately click "Contact". Updating the selected tab is marked as a transition, so it does not block user interactions.
```

<Sandpack>

```js
import { useState } from 'react';
import TabButton from './TabButton.js';
import AboutTab from './AboutTab.js';
import PostsTab from './PostsTab.js';
import ContactTab from './ContactTab.js';

export default function TabContainer() {
const [tab, setTab] = useState('about');
return (
<>
<TabButton
isActive={tab === 'about'}
onClick={() => setTab('about')}
>
About
</TabButton>
<TabButton
isActive={tab === 'posts'}
onClick={() => setTab('posts')}
>
Posts (slow)
</TabButton>
<TabButton
isActive={tab === 'contact'}
onClick={() => setTab('contact')}
>
Contact
</TabButton>
<hr />
{tab === 'about' && <AboutTab />}
{tab === 'posts' && <PostsTab />}
{tab === 'contact' && <ContactTab />}
</>
);
```

}
```

```js TabButton.js active
import { useTransition } from 'react';

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

```js AboutTab.js
export default function AboutTab() {
return (
<p>Welcome to my profile!</p>
);
}
```

```js PostsTab.js
import { memo } from 'react';

const PostsTab = memo(function PostsTab() {
// Log once. The actual slowdown is inside SlowPost.
console.log('[ARTIFICIALLY SLOW] Rendering 500 <SlowPost />');

let items = [];

```
for (let i = 0; i < 500; i++) {
items.push(<SlowPost key={i} index={i} />);
}
return (
<ul className="items">
{items}
</ul>
);
});

function SlowPost({ index }) {
let startTime = performance.now();
while (performance.now() - startTime < 1) {
// Do nothing for 1 ms per item to emulate extremely slow code
}

return (
<li className="item">
Post #{index + 1}
</li>
);
}

export default PostsTab;
```

```js ContactTab.js
export default function ContactTab() {
return (
<>
<p>
You can find me online here:
</p>
<ul>
<li>admin@mysite.com</li>
<li>+123456789</li>
</ul>
</>
```

);

}
```

```css
button { margin-right: 10px }

b { display: inline-block; margin-right: 10px; }
```

</Sandpack>

Displaying a pending visual state during the transition
{/*displaying-a-pending-visual-state-during-the-transition*/}

You can use the `isPending` boolean value returned by `useTransition` to indicate to the user that a transition is in progress. For example, the tab button can have a special "pending" visual state:

```js {4-6}
function TabButton({ children, isActive, onClick }) {

const [isPending, startTransition] = useTransition();

// ...

if (isPending) {

return <b className="pending">{children}</b>;

}

// ...
```

Notice how clicking "Posts" now feels more responsive because the tab button itself updates right away:

<Sandpack>

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

import TabButton from './TabButton.js';

import AboutTab from './AboutTab.js';

import PostsTab from './PostsTab.js';

import ContactTab from './ContactTab.js';

export default function TabContainer() {

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

```
return (
<>
<TabButton
isActive={tab === 'about'}
onClick={() => setTab('about')}
>
About
</TabButton>
<TabButton
isActive={tab === 'posts'}
onClick={() => setTab('posts')}
>
Posts (slow)
</TabButton>
<TabButton
isActive={tab === 'contact'}
onClick={() => setTab('contact')}
>
Contact
</TabButton>
<hr />
{tab === 'about' && <AboutTab />}
{tab === 'posts' && <PostsTab />}
{tab === 'contact' && <ContactTab />}
</>
);
}
```

```js TabButton.js active
import { useTransition } from 'react';

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

````
if (isPending) {
return <b className="pending">{children}</b>;
}
return (
<button onClick={() => {
startTransition(() => {
onClick();
});
}}>
{children}
</button>
);
}
```

```js AboutTab.js
export default function AboutTab() {
return (
<p>Welcome to my profile!</p>
);
}
```

```js PostsTab.js
import { memo } from 'react';

const PostsTab = memo(function PostsTab() {
// Log once. The actual slowdown is inside SlowPost.
console.log('[ARTIFICIALLY SLOW] Rendering 500 <SlowPost />');

let items = [];
for (let i = 0; i < 500; i++) {
items.push(<SlowPost key={i} index={i} />);
}
return (
<ul className="items">
{items}
</ul>
````

```
);
});

function SlowPost({ index }) {
let startTime = performance.now();
while (performance.now() - startTime < 1) {
// Do nothing for 1 ms per item to emulate extremely slow code
}

return (
<li className="item">
Post #{index + 1}
</li>
);
}

export default PostsTab;
```

```js ContactTab.js
export default function ContactTab() {
return (
<>
<p>
You can find me online here:
</p>
<ul>
<li>admin@mysite.com</li>
<li>+123456789</li>
</ul>
</>
);
}
```

```css
button { margin-right: 10px }
b { display: inline-block; margin-right: 10px; }
.pending { color: #777; }
```

```

</Sandpack>

---

### Preventing unwanted loading indicators {/*preventing-unwanted-loading-indicators*/}

In this example, the `PostsTab` component fetches some data using a [Suspense-enabled](/reference/react/Suspense) data source. When you click the "Posts" tab, the `PostsTab` component *suspends*, causing the closest loading fallback to appear:

<Sandpack>

```js
import { Suspense, useState } from 'react';
import TabButton from './TabButton.js';
import AboutTab from './AboutTab.js';
import PostsTab from './PostsTab.js';
import ContactTab from './ContactTab.js';

export default function TabContainer() {
const [tab, setTab] = useState('about');
return (
<Suspense fallback={<h1>■ Loading...</h1>}>
<TabButton
isActive={tab === 'about'}
onClick={() => setTab('about')}
>
About
</TabButton>
<TabButton
isActive={tab === 'posts'}
onClick={() => setTab('posts')}
>
Posts
</TabButton>
<TabButton
isActive={tab === 'contact'}
onClick={() => setTab('contact')}
>
```
```

Contact
</TabButton>
<hr />
{tab === 'about' && <AboutTab />}
{tab === 'posts' && <PostsTab />}
{tab === 'contact' && <ContactTab />}
</Suspense>
);
}
```

```js TabButton.js
export default function TabButton({ children, isActive, onClick }) {
if (isActive) {
return <b>{children}</b>
}
return (
<button onClick={() => {
onClick();
}}>
{children}
</button>
);
}
```

```js AboutTab.js hidden
export default function AboutTab() {
return (
<p>Welcome to my profile!</p>
);
}
```

```js PostsTab.js hidden
import { fetchData } from './data.js';

// Note: this component is written using an experimental API

```
// that's not yet available in stable versions of React.

// For a realistic example you can follow today, try a framework
// that's integrated with Suspense, like Relay or Next.js.

function PostsTab() {
const posts = use(fetchData('/posts'));
return (
<ul className="items">
{posts.map(post =>
<Post key={post.id} title={post.title} />
)}
</ul>
);
}

function Post({ title }) {
return (
<li className="item">
{title}
</li>
);
}

export default PostsTab;

// This is a workaround for a bug to get the demo running.
// TODO: replace with real implementation when the bug is fixed.
function use(promise) {
if (promise.status === 'fulfilled') {
return promise.value;
} else if (promise.status === 'rejected') {
throw promise.reason;
} else if (promise.status === 'pending') {
throw promise;
} else {
promise.status = 'pending';
promise.then(
result => {
```

```
promise.status = 'fulfilled';
promise.value = result;
},
reason => {
promise.status = 'rejected';
promise.reason = reason;
},
);
throw promise;
}
}
```

```js ContactTab.js hidden
export default function ContactTab() {
return (
<>
<p>
You can find me online here:
</p>
<ul>
<li>admin@mysite.com</li>
<li>+123456789</li>
</ul>
</>
);
}
```

```js data.js hidden
// Note: the way you would do data fetching depends on
// the framework that you use together with Suspense.
// Normally, the caching logic would be inside a framework.

let cache = new Map();

export function fetchData(url) {
if (!cache.has(url)) {
```

```
cache.set(url, getData(url));
}
return cache.get(url);
}

async function getData(url) {
if (url.startsWith('/posts')) {
return await getPosts();
} else {
throw Error('Not implemented');
}
}

async function getPosts() {
// Add a fake delay to make waiting noticeable.
await new Promise(resolve => {
setTimeout(resolve, 1000);
});
let posts = [];
for (let i = 0; i < 500; i++) {
posts.push({
id: i,
title: 'Post #' + (i + 1)
});
}
return posts;
}
```

```css
button { margin-right: 10px }
b { display: inline-block; margin-right: 10px; }
.pending { color: #777; }
```

</Sandpack>

Hiding the entire tab container to show a loading indicator leads to a jarring user experience. If you add `useTransition` to `TabButton`, you can instead indicate display the pending state in the tab button

instead.

Notice that clicking "Posts" no longer replaces the entire tab container with a spinner:

<Sandpack>

```js
import { Suspense, useState } from 'react';
import TabButton from './TabButton.js';
import AboutTab from './AboutTab.js';
import PostsTab from './PostsTab.js';
import ContactTab from './ContactTab.js';

export default function TabContainer() {
const [tab, setTab] = useState('about');
return (
<Suspense fallback={<h1>■ Loading...</h1>}>
<TabButton
isActive={tab === 'about'}
onClick={() => setTab('about')}
>
About
</TabButton>
<TabButton
isActive={tab === 'posts'}
onClick={() => setTab('posts')}
>
Posts
</TabButton>
<TabButton
isActive={tab === 'contact'}
onClick={() => setTab('contact')}
>
Contact
</TabButton>
<hr />
{tab === 'about' && <AboutTab />}
{tab === 'posts' && <PostsTab />}
```

```
{tab === 'contact' && <ContactTab />}
</Suspense>
);
}
```

```js TabButton.js active
import { useTransition } from 'react';

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

```js AboutTab.js hidden
export default function AboutTab() {
return (
<p>Welcome to my profile!</p>
);
}
```

```js PostsTab.js hidden
```

```
import { fetchData } from './data.js';

// Note: this component is written using an experimental API
// that's not yet available in stable versions of React.

// For a realistic example you can follow today, try a framework
// that's integrated with Suspense, like Relay or Next.js.

function PostsTab() {
const posts = use(fetchData('/posts'));
return (
<ul className="items">
{posts.map(post =>
<Post key={post.id} title={post.title} />
)}
</ul>
);
}

function Post({ title }) {
return (
<li className="item">
{title}
</li>
);
}

export default PostsTab;

// This is a workaround for a bug to get the demo running.
// TODO: replace with real implementation when the bug is fixed.
function use(promise) {
if (promise.status === 'fulfilled') {
return promise.value;
} else if (promise.status === 'rejected') {
throw promise.reason;
} else if (promise.status === 'pending') {
throw promise;
} else {
```

```
promise.status = 'pending';
promise.then(
result => {
promise.status = 'fulfilled';
promise.value = result;
},
reason => {
promise.status = 'rejected';
promise.reason = reason;
},
);
throw promise;
}
}
```

```js ContactTab.js hidden
export default function ContactTab() {
return (
<>
<p>
You can find me online here:
</p>
<ul>
<li>admin@mysite.com</li>
<li>+123456789</li>
</ul>
</>
);
}
```

```js data.js hidden
// Note: the way you would do data fetching depends on
// the framework that you use together with Suspense.
// Normally, the caching logic would be inside a framework.
```

```
let cache = new Map();

export function fetchData(url) {
if (!cache.has(url)) {
cache.set(url, getData(url));
}
return cache.get(url);
}

async function getData(url) {
if (url.startsWith('/posts')) {
return await getPosts();
} else {
throw Error('Not implemented');
}
}

async function getPosts() {
// Add a fake delay to make waiting noticeable.
await new Promise(resolve => {
setTimeout(resolve, 1000);
});
let posts = [];
for (let i = 0; i < 500; i++) {
posts.push({
id: i,
title: 'Post #' + (i + 1)
});
}
return posts;
}
```

```css
button { margin-right: 10px }
b { display: inline-block; margin-right: 10px; }
.pending { color: #777; }
```

</Sandpack>

[Read more about using transitions with Suspense.](/reference/react/Suspense#preventing-already-revealed-content-from-hiding)

<Note>

Transitions will only "wait" long enough to avoid hiding *already revealed* content (like the tab container). If the Posts tab had a [nested `<Suspense>` boundary,](/reference/react/Suspense#revealing-nested-content-as-it-loads) the transition would not "wait" for it.

</Note>

Building a Suspense-enabled router {/*building-a-suspense-enabled-router*/}

If you're building a React framework or a router, we recommend marking page navigations as transitions.

```js {3,6,8}
function Router() {
const [page, setPage] = useState('/');
const [isPending, startTransition] = useTransition();

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

This is recommended for two reasons:

- [Transitions are interruptible,](#marking-a-state-update-as-a-non-blocking-transition) which lets the user click away without waiting for the re-render to complete.
- [Transitions prevent unwanted loading indicators,](#preventing-unwanted-loading-indicators) which lets the user avoid jarring jumps on navigation.

Here is a tiny simplified router example using transitions for navigations.

<Sandpack>

```json package.json hidden
{
```

```
"dependencies": {
"react": "experimental",
"react-dom": "experimental"
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
}
}
```

```js App.js
import { Suspense, useState, useTransition } from 'react';
import IndexPage from './IndexPage.js';
import ArtistPage from './ArtistPage.js';
import Layout from './Layout.js';

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

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

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

let content;
if (page === '/') {
```

```
content = (
<IndexPage navigate={navigate} />
);
} else if (page === '/the-beatles') {
content = (
<ArtistPage
artist={{
id: 'the-beatles',
name: 'The Beatles',
}}
/>
);
}
return (
<Layout isPending={isPending}>
{content}
</Layout>
);
}

function BigSpinner() {
return <h2>■ Loading...</h2>;
}
```

```js Layout.js
export default function Layout({ children, isPending }) {
return (
<div className="layout">
<section className="header" style={{
opacity: isPending ? 0.7 : 1
}}>
Music Browser
</section>
<main>
{children}
</main>
```

```
</div>
);
}
```

```js IndexPage.js
export default function IndexPage({ navigate }) {
return (
<button onClick={() => navigate('/the-beatles')}>
Open The Beatles artist page
</button>
);
}
```

```js ArtistPage.js
import { Suspense } from 'react';
import Albums from './Albums.js';
import Biography from './Biography.js';
import Panel from './Panel.js';

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

function AlbumsGlimmer() {
return (
<div className="glimmer-panel">
```

```
<div className="glimmer-line" />
<div className="glimmer-line" />
<div className="glimmer-line" />
</div>
);
}
```

```js Albums.js hidden
import { fetchData } from './data.js';

// Note: this component is written using an experimental API
// that's not yet available in stable versions of React.

// For a realistic example you can follow today, try a framework
// that's integrated with Suspense, like Relay or Next.js.

export default function Albums({ artistId }) {
const albums = use(fetchData(`/${artistId}/albums`));
return (
<ul>
{albums.map(album => (
<li key={album.id}>
{album.title} ({album.year})
</li>
))}
</ul>
);
}

// This is a workaround for a bug to get the demo running.
// TODO: replace with real implementation when the bug is fixed.
function use(promise) {
if (promise.status === 'fulfilled') {
return promise.value;
} else if (promise.status === 'rejected') {
throw promise.reason;
} else if (promise.status === 'pending') {
throw promise;
```

```
} else {
promise.status = 'pending';
promise.then(
result => {
promise.status = 'fulfilled';
promise.value = result;
},
reason => {
promise.status = 'rejected';
promise.reason = reason;
},
);
throw promise;
}
}
```

```js Biography.js hidden
import { fetchData } from './data.js';

// Note: this component is written using an experimental API
// that's not yet available in stable versions of React.

// For a realistic example you can follow today, try a framework
// that's integrated with Suspense, like Relay or Next.js.

export default function Biography({ artistId }) {
const bio = use(fetchData(`/${artistId}/bio`));
return (
<section>
<p className="bio">{bio}</p>
</section>
);
}

// This is a workaround for a bug to get the demo running.
// TODO: replace with real implementation when the bug is fixed.
function use(promise) {
if (promise.status === 'fulfilled') {
```

````
return promise.value;
} else if (promise.status === 'rejected') {
throw promise.reason;
} else if (promise.status === 'pending') {
throw promise;
} else {
promise.status = 'pending';
promise.then(
result => {
promise.status = 'fulfilled';
promise.value = result;
},
reason => {
promise.status = 'rejected';
promise.reason = reason;
},
);
throw promise;
}
}
```

```js Panel.js hidden
export default function Panel({ children }) {
return (
<section className="panel">
{children}
</section>
);
}
```

```js data.js hidden
// Note: the way you would do data fetching depends on
// the framework that you use together with Suspense.
// Normally, the caching logic would be inside a framework.
````

```
let cache = new Map();

export function fetchData(url) {
if (!cache.has(url)) {
cache.set(url, getData(url));
}
return cache.get(url);
}

async function getData(url) {
if (url === '/the-beatles/albums') {
return await getAlbums();
} else if (url === '/the-beatles/bio') {
return await getBio();
} else {
throw Error('Not implemented');
}
}

async function getBio() {
// Add a fake delay to make waiting noticeable.
await new Promise(resolve => {
setTimeout(resolve, 500);
});

return `The Beatles were an English rock band,
formed in Liverpool in 1960, that comprised
John Lennon, Paul McCartney, George Harrison
and Ringo Starr.`;
}

async function getAlbums() {
// Add a fake delay to make waiting noticeable.
await new Promise(resolve => {
setTimeout(resolve, 3000);
});

return [{
id: 13,
```

title: 'Let It Be',
year: 1970
}, {
id: 12,
title: 'Abbey Road',
year: 1969
}, {
id: 11,
title: 'Yellow Submarine',
year: 1969
}, {
id: 10,
title: 'The Beatles',
year: 1968
}, {
id: 9,
title: 'Magical Mystery Tour',
year: 1967
}, {
id: 8,
title: 'Sgt. Pepper\'s Lonely Hearts Club Band',
year: 1967
}, {
id: 7,
title: 'Revolver',
year: 1966
}, {
id: 6,
title: 'Rubber Soul',
year: 1965
}, {
id: 5,
title: 'Help!',
year: 1965
}, {

id: 4,
title: 'Beatles For Sale',
year: 1964
}, {
id: 3,
title: 'A Hard Day\'s Night',
year: 1964
}, {
id: 2,
title: 'With The Beatles',
year: 1963
}, {
id: 1,
title: 'Please Please Me',
year: 1963
}];
}
```

```css
main {
min-height: 200px;
padding: 10px;
}

.layout {
border: 1px solid black;
}

.header {
background: #222;
padding: 10px;
text-align: center;
color: white;
}

.bio { font-style: italic; }

.panel {

```
border: 1px solid #aaa;
border-radius: 6px;
margin-top: 20px;
padding: 10px;
}

.glimmer-panel {
border: 1px dashed #aaa;
background: linear-gradient(90deg, rgba(221,221,221,1) 0%, rgba(255,255,255,1) 100%);
border-radius: 6px;
margin-top: 20px;
padding: 10px;
}

.glimmer-line {
display: block;
width: 60%;
height: 20px;
margin: 10px;
border-radius: 4px;
background: #f0f0f0;
}
```

</Sandpack>

<Note>

[Suspense-enabled](/reference/react/Suspense) routers are expected to wrap the navigation updates into transitions by default.

</Note>

Troubleshooting {/*troubleshooting*/}

Updating an input in a transition doesn't work {/*updating-an-input-in-a-transition-doesnt-work*/}

You can't use a transition for a state variable that controls an input:

```js {4,10}
const [text, setText] = useState('');
```

```
// ...
function handleChange(e) {
// ■ Can't use transitions for controlled input state
startTransition(() => {
setText(e.target.value);
});
}
// ...
return <input value={text} onChange={handleChange} />;
```

This is because transitions are non-blocking, but updating an input in response to the change event should happen synchronously. If you want to run a transition in response to typing, you have two options:

1. You can declare two separate state variables: one for the input state (which always updates synchronously), and one that you will update in a transition. This lets you control the input using the synchronous state, and pass the transition state variable (which will "lag behind" the input) to the rest of your rendering logic.
2. Alternatively, you can have one state variable, and add [`useDeferredValue`](/reference/react/useDeferredValue) which will "lag behind" the real value. It will trigger non-blocking re-renders to "catch up" with the new value automatically.

React doesn't treat my state update as a transition
{/*react-doesnt-treat-my-state-update-as-a-transition*/}

When you wrap a state update in a transition, make sure that it happens *during* the `startTransition` call:

```js
startTransition(() => {
// ■ Setting state *during* startTransition call
setPage('/about');
});
```

The function you pass to `startTransition` must be synchronous.

You can't mark an update as a transition like this:

```js
startTransition(() => {
```

```
// ■ Setting state *after* startTransition call
setTimeout(() => {
setPage('/about');
}, 1000);
});
```

Instead, you could do this:

```js
setTimeout(() => {
startTransition(() => {
// ■ Setting state *during* startTransition call
setPage('/about');
});
}, 1000);
```

Similarly, you can't mark an update as a transition like this:

```js
startTransition(async () => {
await someAsyncFunction();
// ■ Setting state *after* startTransition call
setPage('/about');
});
```

However, this works instead:

```js
await someAsyncFunction();
startTransition(() => {
// ■ Setting state *during* startTransition call
setPage('/about');
});
```

I want to call `useTransition` from outside a component
{/*i-want-to-call-usetransition-from-outside-a-component*/}

You can't call `useTransition` outside a component because it's a Hook. In this case, use the standalone [`startTransition`](/reference/react/startTransition) method instead. It works the same way, but it doesn't provide the `isPending` indicator.

The function I pass to `startTransition` executes immediately
{/*the-function-i-pass-to-starttransition-executes-immediately*/}

If you run this code, it will print 1, 2, 3:

```js {1,3,6}
console.log(1);
startTransition(() => {
console.log(2);
setPage('/about');
});
console.log(3);
```

It is expected to print 1, 2, 3. The function you pass to `startTransition` does not get delayed. Unlike with the browser `setTimeout`, it does not run the callback later. React executes your function immediately, but any state updates scheduled *while it is running* are marked as transitions. You can imagine that it works like this:

```js
// A simplified version of how React works

let isInsideTransition = false;

function startTransition(scope) {
isInsideTransition = true;
scope();
isInsideTransition = false;
}

function setState() {
if (isInsideTransition) {
// ... schedule a transition state update ...
} else {
// ... schedule an urgent state update ...
```

}

}

```

---

title: <Profiler>

---

<Intro>

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

```js

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

<App />

</Profiler>

```

</Intro>

<InlineToc />

Reference {/*reference*/}

`<Profiler>` {/*profiler*/}

Wrap a component tree in a `<Profiler>` to measure its rendering performance.

```js

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

<App />

</Profiler>

```

Props {/*props*/}

* `id`: A string identifying the part of the UI you are measuring.

* `onRender`: An [`onRender` callback](#onrender-callback) that React calls every time components within the profiled tree update. It receives information about what was rendered and how much time it took.

Caveats {/*caveats*/}

* Profiling adds some additional overhead, so **it is disabled in the production build by default.** To opt into production profiling, you need to enable a [special production build with profiling
```

enabled.](https://fb.me/react-profiling)

`onRender` callback {/*onrender-callback*/}

React will call your `onRender` callback with information about what was rendered.

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

Parameters {/*onrender-parameters*/}

* `id`: The string `id` prop of the `<Profiler>` tree that has just committed. This lets you identify which part of the tree was committed if you are using multiple profilers.

* `phase`: `"mount"`, `"update"` or `"nested-update"`. This lets you know whether the tree has just been mounted for the first time or re-rendered due to a change in props, state, or hooks.

* `actualDuration`: The number of milliseconds spent rendering the `<Profiler>` and its descendants for the current update. This indicates how well the subtree makes use of memoization (e.g. [`memo`](/reference/react/memo) and [`useMemo`](/reference/react/useMemo)). Ideally this value should decrease significantly after the initial mount as many of the descendants will only need to re-render if their specific props change.

* `baseDuration`: The number of milliseconds estimating how much time it would take to re-render the entire `<Profiler>` subtree without any optimizations. It is calculated by summing up the most recent render durations of each component in the tree. This value estimates a worst-case cost of rendering (e.g. the initial mount or a tree with no memoization). Compare `actualDuration` against it to see if memoization is working.

* `startTime`: A numeric timestamp for when React began rendering the current update.

* `endTime`: A numeric timestamp for when React committed the current update. This value is shared between all profilers in a commit, enabling them to be grouped if desirable.

Usage {/*usage*/}

Measuring rendering performance programmatically
{/*measuring-rendering-performance-programmatically*/}

Wrap the `<Profiler>` component around a React tree to measure its rendering performance.

```js {2,4}
<App>
<Profiler id="Sidebar" onRender={onRender}>
<Sidebar />
```

</Profiler>

<PageContent />

</App>
```

It requires two props: an `id` (string) and an `onRender` callback (function) which React calls any time a component within the tree "commits" an update.

<Pitfall>

Profiling adds some additional overhead, so **it is disabled in the production build by default.** To opt into production profiling, you need to enable a [special production build with profiling enabled.](https://fb.me/react-profiling)

</Pitfall>

<Note>

`<Profiler>` lets you gather measurements programmatically. If you're looking for an interactive profiler, try the Profiler tab in [React Developer Tools](/learn/react-developer-tools). It exposes similar functionality as a browser extension.

</Note>

---

### Measuring different parts of the application {/*measuring-different-parts-of-the-application*/}

You can use multiple `<Profiler>` components to measure different parts of your application:

```js {5,7}
<App>
<Profiler id="Sidebar" onRender={onRender}>
<Sidebar />
</Profiler>
<Profiler id="Content" onRender={onRender}>
<Content />
</Profiler>
</App>
```

You can also nest `<Profiler>` components:

```js {5,7,9,12}
<App>
<Profiler id="Sidebar" onRender={onRender}>

<Sidebar />

</Profiler>

<Profiler id="Content" onRender={onRender}>

<Content>

<Profiler id="Editor" onRender={onRender}>

<Editor />

</Profiler>

<Preview />

</Content>

</Profiler>

</App>

```

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

---

title: useEffect

<Intro>

`useEffect` is a React Hook that lets you [synchronize a component with an external system.](/learn/synchronizing-with-effects)

```js

useEffect(setup, dependencies?)

```

</Intro>

<InlineToc />

Reference {/*reference*/}

`useEffect(setup, dependencies?)` {/*useeffect*/}

Call `useEffect` at the top level of your component to declare an Effect:

```js

import { useEffect } from 'react';

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

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

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

[See more examples below.](#usage)

Parameters {/*parameters*/}

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

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

Returns {/*returns*/}

`useEffect` returns `undefined`.

Caveats {/*caveats*/}

* `useEffect` is a Hook, so you can only call it **at the top level of your component** or your own Hooks. You can't call it inside loops or conditions. If you need that, extract a new component and move the state into it.

* If you're **not trying to synchronize with some external system,** [you probably don't need an Effect.](/learn/you-might-not-need-an-effect)

* When Strict Mode is on, React will **run one extra development-only setup+cleanup cycle** before the first real setup. This is a stress-test that ensures that your cleanup logic "mirrors" your setup logic and that it stops or undoes whatever the setup is doing. If this causes a problem, [implement the cleanup
function.](/learn/synchronizing-with-effects#how-to-handle-the-effect-firing-twice-in-development)

* If some of your dependencies are objects or functions defined inside the component, there is a risk that they will **cause the Effect to re-run more often than needed.** To fix this, remove unnecessary [object](#removing-unnecessary-object-dependencies) and [function](#removing-unnecessary-function-dependencies) dependencies. You can also [extract state updates](#updating-state-based-on-previous-state-from-an-effect) and [non-reactive logic](#reading-the-latest-props-and-state-from-an-effect) outside of your Effect.

* If your Effect wasn't caused by an interaction (like a click), React will let the browser **paint the updated screen first before running your Effect.** If your Effect is doing something visual (for example, positioning a tooltip), and the delay is noticeable (for example, it flickers), replace `useEffect` with [`useLayoutEffect`.](/reference/react/useLayoutEffect)

* Even if your Effect was caused by an interaction (like a click), **the browser may repaint the screen before processing the state updates inside your Effect.** Usually, that's what you want. However, if you must block the browser from repainting the screen, you need to replace `useEffect` with [`useLayoutEffect`.](/reference/react/useLayoutEffect)

* Effects **only run on the client.** They don't run during server rendering.

Usage {/*usage*/}

Connecting to an external system {/*connecting-to-an-external-system*/}

Some components need to stay connected to the network, some browser API, or a third-party library, while they are displayed on the page. These systems aren't controlled by React, so they are called *external.*

To [connect your component to some external system,](/learn/synchronizing-with-effects) call `useEffect` at the top level of your component:

```js [[1, 8, "const connection = createConnection(serverUrl, roomId);"], [1, 9, "connection.connect();"], [2, 11, "connection.disconnect();"], [3, 13, "[serverUrl, roomId]"]]
import { useEffect } from 'react';
import { createConnection } from './chat.js';

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

useEffect(() => {
const connection = createConnection(serverUrl, roomId);
connection.connect();
return () => {
```

```
connection.disconnect();
};
}, [serverUrl, roomId]);
// ...
}
```

You need to pass two arguments to `useEffect`:

1. A *setup function* with <CodeStep step={1}>setup code</CodeStep> that connects to that system.
- It should return a *cleanup function* with <CodeStep step={2}>cleanup code</CodeStep> that disconnects from that system.
2. A <CodeStep step={3}>list of dependencies</CodeStep> including every value from your component used inside of those functions.

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

1. Your <CodeStep step={1}>setup code</CodeStep> runs when your component is added to the page *(mounts)*.
2. After every re-render of your component where the <CodeStep step={3}>dependencies</CodeStep> have changed:
- First, your <CodeStep step={2}>cleanup code</CodeStep> runs with the old props and state.
- Then, your <CodeStep step={1}>setup code</CodeStep> runs with the new props and state.
3. Your <CodeStep step={2}>cleanup code</CodeStep> runs one final time after your component is removed from the page *(unmounts).*

Let's illustrate this sequence for the example above.

When the `ChatRoom` component above gets added to the page, it will connect to the chat room with the initial `serverUrl` and `roomId`. If either `serverUrl` or `roomId` change as a result of a re-render (say, if the user picks a different chat room in a dropdown), your Effect will *disconnect from the previous room, and connect to the next one.* When the `ChatRoom` component is removed from the page, your Effect will disconnect one last time.

To [help you find bugs,](/learn/synchronizing-with-effects#step-3-add-cleanup-if-needed) in development React runs <CodeStep step={1}>setup</CodeStep> and <CodeStep step={2}>cleanup</CodeStep> one extra time before the <CodeStep step={1}>setup</CodeStep>. This is a stress-test that verifies your Effect's logic is implemented correctly. If this causes visible issues, your cleanup function is missing some logic. The cleanup function should stop or undo whatever the setup function was doing. The rule of thumb is that the user shouldn't be able to distinguish between the setup being called once (as in production) and a *setup* → *cleanup* → *setup* sequence (as in development). [See common solutions.](/learn/synchronizing-with-effects#how-to-handle-the-effect-firing-twice-in-development)

**Try to [write every Effect as an independent process](/learn/lifecycle-of-reactive-effects#each-effect-represents-a-separate-synchronization-process) and [think about a single setup/cleanup cycle at a

time.](/learn/lifecycle-of-reactive-effects#thinking-from-the-effects-perspective)** It shouldn't matter whether your component is mounting, updating, or unmounting. When your cleanup logic correctly "mirrors" the setup logic, your Effect is resilient to running setup and cleanup as often as needed.

<Note>

An Effect lets you [keep your component synchronized](/learn/synchronizing-with-effects) with some external system (like a chat service). Here, *external system* means any piece of code that's not controlled by React, such as:

* A timer managed with <CodeStep step={1}>[`setInterval()`](https://developer.mozilla.org/en-US/docs/Web/API/setInterval)</CodeStep> and <CodeStep step={2}>[`clearInterval()`](https://developer.mozilla.org/en-US/docs/Web/API/clearInterval)</CodeStep>.

* An event subscription using <CodeStep step={1}>[`window.addEventListener()`](https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener)</CodeStep> and <CodeStep step={2}>[`window.removeEventListener()`](https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/removeEventListener)</CodeStep>.

* A third-party animation library with an API like <CodeStep step={1}>`animation.start()`</CodeStep> and <CodeStep step={2}>`animation.reset()`</CodeStep>.

If you're not connecting to any external system, [you probably don't need an Effect.](/learn/you-might-not-need-an-effect)

</Note>

<Recipes titleText="Examples of connecting to an external system" titleId="examples-connecting">

Connecting to a chat server {/*connecting-to-a-chat-server*/}

In this example, the `ChatRoom` component uses an Effect to stay connected to an external system defined in `chat.js`. Press "Open chat" to make the `ChatRoom` component appear. This sandbox runs in development mode, so there is an extra connect-and-disconnect cycle, as [explained here.](/learn/synchronizing-with-effects#step-3-add-cleanup-if-needed) Try changing the `roomId` and `serverUrl` using the dropdown and the input, and see how the Effect re-connects to the chat. Press "Close chat" to see the Effect disconnect one last time.

<Sandpack>

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

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

useEffect(() => {
const connection = createConnection(serverUrl, roomId);
connection.connect();
```

```
return () => {
connection.disconnect();
};
}, [roomId, serverUrl]);

return (
<>
<label>
Server URL:{' '}
<input
value={serverUrl}
onChange={e => setServerUrl(e.target.value)}
/>
</label>
<h1>Welcome to the {roomId} room!</h1>
</>
);
}

export default function App() {
const [roomId, setRoomId] = useState('general');
const [show, setShow] = useState(false);
return (
<>
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
</select>
</label>
<button onClick={() => setShow(!show)}>
{show ? 'Close chat' : 'Open chat'}
```

</button>
{show && <hr />}
{show && <ChatRoom roomId={roomId} />}
</>
);
}
```

```js chat.js
export function createConnection(serverUrl, roomId) {
// A real implementation would actually connect to the server
return {
connect() {
console.log('■ Connecting to "' + roomId + '" room at ' + serverUrl + '...');
},
disconnect() {
console.log('■ Disconnected from "' + roomId + '" room at ' + serverUrl);
}
};
}
```

```css
input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }
```

</Sandpack>

<Solution />

Listening to a global browser event {/*listening-to-a-global-browser-event*/}

In this example, the external system is the browser DOM itself. Normally, you'd specify event listeners with JSX, but you can't listen to the global [`window`](https://developer.mozilla.org/en-US/docs/Web/API/Window) object this way. An Effect lets you connect to the `window` object and listen to its events. Listening to the `pointermove` event lets you track the cursor (or finger) position and update the red dot to move with it.

<Sandpack>

```js

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

export default function App() {
const [position, setPosition] = useState({ x: 0, y: 0 });

useEffect(() => {
function handleMove(e) {
setPosition({ x: e.clientX, y: e.clientY });
}
window.addEventListener('pointermove', handleMove);
return () => {
window.removeEventListener('pointermove', handleMove);
};
}, []);

return (
<div style={{
position: 'absolute',
backgroundColor: 'pink',
borderRadius: '50%',
opacity: 0.6,
transform: `translate(${position.x}px, ${position.y}px)`,
pointerEvents: 'none',
left: -20,
top: -20,
width: 40,
height: 40,
}} />
);
}
```

```css
body {
min-height: 300px;
}
```

</Sandpack>

<Solution />

Triggering an animation {/*triggering-an-animation*/}

In this example, the external system is the animation library in `animation.js`. It provides a JavaScript class called `FadeInAnimation` that takes a DOM node as an argument and exposes `start()` and `stop()` methods to control the animation. This component [uses a ref](/learn/manipulating-the-dom-with-refs) to access the underlying DOM node. The Effect reads the DOM node from the ref and automatically starts the animation for that node when the component appears.

<Sandpack>

```js
import { useState, useEffect, useRef } from 'react';
import { FadeInAnimation } from './animation.js';

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

useEffect(() => {
const animation = new FadeInAnimation(ref.current);
animation.start(1000);
return () => {
animation.stop();
};
}, []);

return (
<h1
ref={ref}
style={{
opacity: 0,
color: 'white',
padding: 50,
textAlign: 'center',
fontSize: 50,
backgroundImage: 'radial-gradient(circle, rgba(63,94,251,1) 0%, rgba(252,70,107,1) 100%)'
}}
>
Welcome
</h1>
```

```
);
}

export default function App() {
const [show, setShow] = useState(false);
return (
<>
<button onClick={() => setShow(!show)}>
{show ? 'Remove' : 'Show'}
</button>
<hr />
{show && <Welcome />}
</>
);
}
```

```js animation.js
export class FadeInAnimation {
constructor(node) {
this.node = node;
}
start(duration) {
this.duration = duration;
if (this.duration === 0) {
// Jump to end immediately
this.onProgress(1);
} else {
this.onProgress(0);
// Start animating
this.startTime = performance.now();
this.frameId = requestAnimationFrame(() => this.onFrame());
}
}
onFrame() {
const timePassed = performance.now() - this.startTime;
const progress = Math.min(timePassed / this.duration, 1);
```

```
this.onProgress(progress);
if (progress < 1) {
// We still have more frames to paint
this.frameId = requestAnimationFrame(() => this.onFrame());
}
}
onProgress(progress) {
this.node.style.opacity = progress;
}
stop() {
cancelAnimationFrame(this.frameId);
this.startTime = null;
this.frameId = null;
this.duration = 0;
}
}
```

```css
label, button { display: block; margin-bottom: 20px; }
html, body { min-height: 300px; }
```

</Sandpack>

<Solution />

Controlling a modal dialog {/*controlling-a-modal-dialog*/}

In this example, the external system is the browser DOM. The `ModalDialog` component renders a [`<dialog>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/dialog) element. It uses an Effect to synchronize the `isOpen` prop to the [`showModal()`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLDialogElement/showModal) and [`close()`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLDialogElement/close) method calls.

<Sandpack>

```js
import { useState } from 'react';
import ModalDialog from './ModalDialog.js';
```

````
export default function App() {
const [show, setShow] = useState(false);
return (
<>
<button onClick={() => setShow(true)}>
Open dialog
</button>
<ModalDialog isOpen={show}>
Hello there!
<br />
<button onClick={() => {
setShow(false);
}}>Close</button>
</ModalDialog>
</>
);
}
```

```js ModalDialog.js active
import { useEffect, useRef } from 'react';

export default function ModalDialog({ isOpen, children }) {
const ref = useRef();

useEffect(() => {
if (!isOpen) {
return;
}
const dialog = ref.current;
dialog.showModal();
return () => {
dialog.close();
};
}, [isOpen]);

return <dialog ref={ref}>{children}</dialog>;
}
````

```

```css
body {
min-height: 300px;
}
```

</Sandpack>

<Solution />

Tracking element visibility {/*tracking-element-visibility*/}

In this example, the external system is again the browser DOM. The `App` component displays a long list, then a `Box` component, and then another long list. Scroll the list down. Notice that when the `Box` component appears in the viewport, the background color changes to black. To implement this, the `Box` component uses an Effect to manage an [`IntersectionObserver`](https://developer.mozilla.org/en-US/docs/Web/API/Intersection_Observer_API). This browser API notifies you when the DOM element is visible in the viewport.

<Sandpack>

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

export default function App() {
return (
<>
<LongSection />
<Box />
<LongSection />
<Box />
<LongSection />
</>
);
}

function LongSection() {
const items = [];
for (let i = 0; i < 50; i++) {
items.push(<li key={i}>Item #{i} (keep scrolling)</li>);
}
```
```

```
return <ul>{items}</ul>
}
```

```js Box.js active
import { useRef, useEffect } from 'react';

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

useEffect(() => {
const div = ref.current;
const observer = new IntersectionObserver(entries => {
const entry = entries[0];
if (entry.isIntersecting) {
document.body.style.backgroundColor = 'black';
document.body.style.color = 'white';
} else {
document.body.style.backgroundColor = 'white';
document.body.style.color = 'black';
}
});
observer.observe(div, {
threshold: 1.0
});
return () => {
observer.disconnect();
}
}, []);

return (
<div ref={ref} style={{
margin: 20,
height: 100,
width: 100,
border: '2px solid black',
backgroundColor: 'blue'
}} />
```

);

}

```

</Sandpack>

<Solution />

</Recipes>

---

### Wrapping Effects in custom Hooks {/*wrapping-effects-in-custom-hooks*/}

Effects are an ["escape hatch":](/learn/escape-hatches) you use them when you need to "step outside React" and when there is no better built-in solution for your use case. If you find yourself often needing to manually write Effects, it's usually a sign that you need to extract some [custom Hooks](/learn/reusing-logic-with-custom-hooks) for common behaviors your components rely on.

For example, this `useChatRoom` custom Hook "hides" the logic of your Effect behind a more declarative API:

```js {1,11}
function useChatRoom({ serverUrl, roomId }) {
useEffect(() => {
const options = {
serverUrl: serverUrl,
roomId: roomId
};
const connection = createConnection(options);
connection.connect();
return () => connection.disconnect();
}, [roomId, serverUrl]);
}
```

Then you can use it from any component like this:

```js {4-7}
function ChatRoom({ roomId }) {
const [serverUrl, setServerUrl] = useState('https://localhost:1234');

useChatRoom({
roomId: roomId,

```
serverUrl: serverUrl
});
// ...
```

There are also many excellent custom Hooks for every purpose available in the React ecosystem.

[Learn more about wrapping Effects in custom Hooks.](/learn/reusing-logic-with-custom-hooks)

<Recipes titleText="Examples of wrapping Effects in custom Hooks" titleId="examples-custom-hooks">

Custom `useChatRoom` Hook {/*custom-usechatroom-hook*/}

This example is identical to one of the [earlier examples,](#examples-connecting) but the logic is extracted to a custom Hook.

<Sandpack>

```js
import { useState } from 'react';
import { useChatRoom } from './useChatRoom.js';

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

useChatRoom({
roomId: roomId,
serverUrl: serverUrl
});

return (
<>
<label>
Server URL:{' '}
<input
value={serverUrl}
onChange={e => setServerUrl(e.target.value)}
/>
</label>
<h1>Welcome to the {roomId} room!</h1>
</>
);
```

```
}

export default function App() {
const [roomId, setRoomId] = useState('general');
const [show, setShow] = useState(false);
return (
<>
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
</select>
</label>
<button onClick={() => setShow(!show)}>
{show ? 'Close chat' : 'Open chat'}
</button>
{show && <hr />}
{show && <ChatRoom roomId={roomId} />}
</>
);
}
```

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

export function useChatRoom({ serverUrl, roomId }) {
useEffect(() => {
const connection = createConnection(serverUrl, roomId);
connection.connect();
return () => {
```

```
connection.disconnect();
};
}, [roomId, serverUrl]);
}
```

```js chat.js
export function createConnection(serverUrl, roomId) {
// A real implementation would actually connect to the server
return {
connect() {
console.log('■ Connecting to "' + roomId + '" room at ' + serverUrl + '...');
},
disconnect() {
console.log('■ Disconnected from "' + roomId + '" room at ' + serverUrl);
}
};
}
```

```css
input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }
```

</Sandpack>

<Solution />

Custom `useWindowListener` Hook {/*custom-usewindowlistener-hook*/}

This example is identical to one of the [earlier examples,](#examples-connecting) but the logic is extracted to a custom Hook.

<Sandpack>

```js
import { useState } from 'react';
import { useWindowListener } from './useWindowListener.js';

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

useWindowListener('pointermove', (e) => {
setPosition({ x: e.clientX, y: e.clientY });
});

return (
<div style={{
position: 'absolute',
backgroundColor: 'pink',
borderRadius: '50%',
opacity: 0.6,
transform: `translate(${position.x}px, ${position.y}px)`,
pointerEvents: 'none',
left: -20,
top: -20,
width: 40,
height: 40,
}} />
);
}
```

```js useWindowListener.js
import { useState, useEffect } from 'react';

export function useWindowListener(eventType, listener) {
useEffect(() => {
window.addEventListener(eventType, listener);
return () => {
window.removeEventListener(eventType, listener);
};
}, [eventType, listener]);
}
```

```css
body {
min-height: 300px;
}

```

</Sandpack>

<Solution />

Custom `useIntersectionObserver` Hook {/*custom-useintersectionobserver-hook*/}

This example is identical to one of the [earlier examples,](#examples-connecting) but the logic is partially extracted to a custom Hook.

<Sandpack>

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

export default function App() {
return (
<>
<LongSection />
<Box />
<LongSection />
<Box />
<LongSection />
</>
);
}

function LongSection() {
const items = [];
for (let i = 0; i < 50; i++) {
items.push(<li key={i}>Item #{i} (keep scrolling)</li>);
}
return <ul>{items}</ul>
}
```

```js Box.js active
import { useRef, useEffect } from 'react';
import { useIntersectionObserver } from './useIntersectionObserver.js';

export default function Box() {

```
const ref = useRef(null);
const isIntersecting = useIntersectionObserver(ref);

useEffect(() => {
if (isIntersecting) {
document.body.style.backgroundColor = 'black';
document.body.style.color = 'white';
} else {
document.body.style.backgroundColor = 'white';
document.body.style.color = 'black';
}
}, [isIntersecting]);

return (
<div ref={ref} style={{
margin: 20,
height: 100,
width: 100,
border: '2px solid black',
backgroundColor: 'blue'
}} />
);
}
```

```js useIntersectionObserver.js
import { useState, useEffect } from 'react';

export function useIntersectionObserver(ref) {
const [isIntersecting, setIsIntersecting] = useState(false);

useEffect(() => {
const div = ref.current;
const observer = new IntersectionObserver(entries => {
const entry = entries[0];
setIsIntersecting(entry.isIntersecting);
});
observer.observe(div, {
threshold: 1.0
```

});

return () => {

observer.disconnect();

}

}, [ref]);

return isIntersecting;

}

```

</Sandpack>

<Solution />

</Recipes>

---

### Controlling a non-React widget {/*controlling-a-non-react-widget*/}

Sometimes, you want to keep an external system synchronized to some prop or state of your component.

For example, if you have a third-party map widget or a video player component written without React, you can use an Effect to call methods on it that make its state match the current state of your React component. This Effect creates an instance of a `MapWidget` class defined in `map-widget.js`. When you change the `zoomLevel` prop of the `Map` component, the Effect calls the `setZoom()` on the class instance to keep it synchronized:

<Sandpack>

```json package.json hidden

{

"dependencies": {

"leaflet": "1.9.1",

"react": "latest",

"react-dom": "latest",

"react-scripts": "latest",

"remarkable": "2.0.1"

},

"scripts": {

"start": "react-scripts start",

"build": "react-scripts build",

"test": "react-scripts test --env=jsdom",

"eject": "react-scripts eject"
}
}
```

```js App.js
import { useState } from 'react';
import Map from './Map.js';

export default function App() {
const [zoomLevel, setZoomLevel] = useState(0);
return (
<>
Zoom level: {zoomLevel}x
<button onClick={() => setZoomLevel(zoomLevel + 1)}>+</button>
<button onClick={() => setZoomLevel(zoomLevel - 1)}>-</button>
<hr />
<Map zoomLevel={zoomLevel} />
</>
);
}
```

```js Map.js active
import { useRef, useEffect } from 'react';
import { MapWidget } from './map-widget.js';

export default function Map({ zoomLevel }) {
const containerRef = useRef(null);
const mapRef = useRef(null);

useEffect(() => {
if (mapRef.current === null) {
mapRef.current = new MapWidget(containerRef.current);
}

const map = mapRef.current;
map.setZoom(zoomLevel);
}, [zoomLevel]);

```
return (
<div
style={{ width: 200, height: 200 }}
ref={containerRef}
/>
);
}
```

```js map-widget.js
import 'leaflet/dist/leaflet.css';
import * as L from 'leaflet';

export class MapWidget {
constructor(domNode) {
this.map = L.map(domNode, {
zoomControl: false,
doubleClickZoom: false,
boxZoom: false,
keyboard: false,
scrollWheelZoom: false,
zoomAnimation: false,
touchZoom: false,
zoomSnap: 0.1
});
L.tileLayer('https://tile.openstreetmap.org/{z}/{x}/{y}.png', {
maxZoom: 19,
attribution: '© OpenStreetMap'
}).addTo(this.map);
this.map.setView([0, 0], 0);
}
setZoom(level) {
this.map.setZoom(level);
}
}
```

```css
button { margin: 5px; }
```

</Sandpack>

In this example, a cleanup function is not needed because the `MapWidget` class manages only the DOM node that was passed to it. After the `Map` React component is removed from the tree, both the DOM node and the `MapWidget` class instance will be automatically garbage-collected by the browser JavaScript engine.

Fetching data with Effects {/*fetching-data-with-effects*/}

You can use an Effect to fetch data for your component. Note that [if you use a framework,](/learn/start-a-new-react-project#production-grade-react-frameworks) using your framework's data fetching mechanism will be a lot more efficient than writing Effects manually.

If you want to fetch data from an Effect manually, your code might look like this:

```js
import { useState, useEffect } from 'react';
import { fetchBio } from './api.js';

export default function Page() {
const [person, setPerson] = useState('Alice');
const [bio, setBio] = useState(null);

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

// ...
```

Note the `ignore` variable which is initialized to `false`, and is set to `true` during cleanup. This ensures [your code doesn't suffer from "race conditions":](https://maxrozen.com/race-conditions-fetching-data-react-with-useeffect) network responses may arrive in a different order than you sent them.

<Sandpack>

```js App.js
import { useState, useEffect } from 'react';
import { fetchBio } from './api.js';

export default function Page() {
const [person, setPerson] = useState('Alice');
const [bio, setBio] = useState(null);
useEffect(() => {
let ignore = false;
setBio(null);
fetchBio(person).then(result => {
if (!ignore) {
setBio(result);
}
});
return () => {
ignore = true;
}
}, [person]);

return (
<>
<select value={person} onChange={e => {
setPerson(e.target.value);
}}>
<option value="Alice">Alice</option>
<option value="Bob">Bob</option>
<option value="Taylor">Taylor</option>
</select>
<hr />
<p><i>{bio ?? 'Loading...'}</i></p>
</>
```

);
}
```

```js api.js hidden
export async function fetchBio(person) {
const delay = person === 'Bob' ? 2000 : 200;
return new Promise(resolve => {
setTimeout(() => {
resolve('This is ' + person + ''s bio.');
}, delay);
})
}
```

</Sandpack>

You can also rewrite using the [`async` / `await`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function) syntax, but you still need to provide a cleanup function:

<Sandpack>

```js App.js
import { useState, useEffect } from 'react';
import { fetchBio } from './api.js';

export default function Page() {
const [person, setPerson] = useState('Alice');
const [bio, setBio] = useState(null);
useEffect(() => {
async function startFetching() {
setBio(null);
const result = await fetchBio(person);
if (!ignore) {
setBio(result);
}
}

let ignore = false;
startFetching();

return () => {
ignore = true;
}
}, [person]);

return (
<>
<select value={person} onChange={e => {
setPerson(e.target.value);
}}>
<option value="Alice">Alice</option>
<option value="Bob">Bob</option>
<option value="Taylor">Taylor</option>
</select>
<hr />
{bio ?? 'Loading...'}
</>
);
}
```

```js api.js hidden
export async function fetchBio(person) {
const delay = person === 'Bob' ? 2000 : 200;
return new Promise(resolve => {
setTimeout(() => {
resolve('This is ' + person + ''s bio.');
}, delay);
})
}
```

</Sandpack>

Writing data fetching directly in Effects gets repetitive and makes it difficult to add optimizations like caching and server rendering later. [It's easier to use a custom Hook--either your own or maintained by the community.](/learn/reusing-logic-with-custom-hooks#when-to-use-custom-hooks)

<DeepDive>
```

What are good alternatives to data fetching in Effects? {/*what-are-good-alternatives-to-data-fetching-in-effects*/}

Writing `fetch` calls inside Effects is a [popular way to fetch data](https://www.robinwieruch.de/react-hooks-fetch-data/), especially in fully client-side apps. This is, however, a very manual approach and it has significant downsides:

- **Effects don't run on the server.** This means that the initial server-rendered HTML will only include a loading state with no data. The client computer will have to download all JavaScript and render your app only to discover that now it needs to load the data. This is not very efficient.

- **Fetching directly in Effects makes it easy to create "network waterfalls".** You render the parent component, it fetches some data, renders the child components, and then they start fetching their data. If the network is not very fast, this is significantly slower than fetching all data in parallel.

- **Fetching directly in Effects usually means you don't preload or cache data.** For example, if the component unmounts and then mounts again, it would have to fetch the data again.

- **It's not very ergonomic.** There's quite a bit of boilerplate code involved when writing `fetch` calls in a way that doesn't suffer from bugs like [race conditions.](https://maxrozen.com/race-conditions-fetching-data-react-with-useeffect)

This list of downsides is not specific to React. It applies to fetching data on mount with any library. Like with routing, data fetching is not trivial to do well, so we recommend the following approaches:

- **If you use a [framework](/learn/start-a-new-react-project#production-grade-react-frameworks), use its built-in data fetching mechanism.** Modern React frameworks have integrated data fetching mechanisms that are efficient and don't suffer from the above pitfalls.

- **Otherwise, consider using or building a client-side cache.** Popular open source solutions include [React Query](https://react-query.tanstack.com/), [useSWR](https://swr.vercel.app/), and [React Router 6.4+.](https://beta.reactrouter.com/en/main/start/overview) You can build your own solution too, in which case you would use Effects under the hood but also add logic for deduplicating requests, caching responses, and avoiding network waterfalls (by preloading data or hoisting data requirements to routes).

You can continue fetching data directly in Effects if neither of these approaches suit you.

</DeepDive>

Specifying reactive dependencies {/*specifying-reactive-dependencies*/}

Notice that you can't "choose" the dependencies of your Effect. Every <CodeStep step={2}>reactive value</CodeStep> used by your Effect's code must be declared as a dependency. Your Effect's dependency list is determined by the surrounding code:

```js [[2, 1, "roomId"], [2, 2, "serverUrl"], [2, 5, "serverUrl"], [2, 5, "roomId"], [2, 8, "serverUrl"], [2, 8, "roomId"]]
function ChatRoom({ roomId }) { // This is a reactive value
  const [serverUrl, setServerUrl] = useState('https://localhost:1234'); // This is a reactive value too

  useEffect(() => {
```

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

If either `serverUrl` or `roomId` change, your Effect will reconnect to the chat using the new values.

[Reactive values](/learn/lifecycle-of-reactive-effects#effects-react-to-reactive-values) include props and all variables and functions declared directly inside of your component. Since `roomId` and `serverUrl` are reactive values, you can't remove them from the dependencies. If you try to omit them and [your linter is correctly configured for React,](/learn/editor-setup#linting) the linter will flag this as a mistake you need to fix:

```js {8}
function ChatRoom({ roomId }) {
const [serverUrl, setServerUrl] = useState('https://localhost:1234');

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

To remove a dependency, you need to ["prove" to the linter that it *doesn't need* to be a dependency.](/learn/removing-effect-dependencies#removing-unnecessary-dependencies) For example, you can move `serverUrl` out of your component to prove that it's not reactive and won't change on re-renders:

```js {1,8}
const serverUrl = 'https://localhost:1234'; // Not a reactive value anymore

function ChatRoom({ roomId }) {
useEffect(() => {
const connection = createConnection(serverUrl, roomId);
connection.connect();
return () => connection.disconnect();
```

}, [roomId]); // ■ All dependencies declared

// ...

}

```

Now that `serverUrl` is not a reactive value (and can't change on a re-render), it doesn't need to be a dependency. **If your Effect's code doesn't use any reactive values, its dependency list should be empty (`[]`):**

```js {1,2,9}

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

const roomId = 'music'; // Not a reactive value anymore

function ChatRoom() {

useEffect(() => {

const connection = createConnection(serverUrl, roomId);

connection.connect();

return () => connection.disconnect();

}, []); // ■ All dependencies declared

// ...

}

```

[An Effect with empty dependencies](/learn/lifecycle-of-reactive-effects#what-an-effect-with-empty-dependencies-means) doesn't re-run when any of your component's props or state change.

<Pitfall>

If you have an existing codebase, you might have some Effects that suppress the linter like this:

```js {3-4}

useEffect(() => {

// ...

// ■ Avoid suppressing the linter like this:

// eslint-ignore-next-line react-hooks/exhaustive-deps

}, []);

```

When dependencies don't match the code, there is a high risk of introducing bugs. By suppressing the linter, you "lie" to React about the values your Effect depends on. [Instead, prove they're unnecessary.](/learn/removing-effect-dependencies#removing-unnecessary-dependencies)
```

</Pitfall>

<Recipes titleText="Examples of passing reactive dependencies" titleId="examples-dependencies">

Passing a dependency array {/*passing-a-dependency-array*/}

If you specify the dependencies, your Effect runs **after the initial render _and_ after re-renders with changed dependencies.**

```js {3}
useEffect(() => {
// ...
}, [a, b]); // Runs again if a or b are different
```

In the below example, `serverUrl` and `roomId` are [reactive values,](/learn/lifecycle-of-reactive-effects#effects-react-to-reactive-values) so they both must be specified as dependencies. As a result, selecting a different room in the dropdown or editing the server URL input causes the chat to re-connect. However, since `message` isn't used in the Effect (and so it isn't a dependency), editing the message doesn't re-connect to the chat.

<Sandpack>

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

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

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

return (
<>
<label>
Server URL:{' '}
<input
```

```
value={serverUrl}
onChange={e => setServerUrl(e.target.value)}
/>
</label>
<h1>Welcome to the {roomId} room!</h1>
<label>
Your message:{' '}
<input value={message} onChange={e => setMessage(e.target.value)} />
</label>
</>
);
}

export default function App() {
const [show, setShow] = useState(false);
const [roomId, setRoomId] = useState('general');
return (
<>
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
</select>
<button onClick={() => setShow(!show)}>
{show ? 'Close chat' : 'Open chat'}
</button>
</label>
{show && <hr />}
{show && <ChatRoom roomId={roomId}/>}
</>
);
```

}
```

```js chat.js
export function createConnection(serverUrl, roomId) {
// A real implementation would actually connect to the server
return {
connect() {
console.log('■ Connecting to "' + roomId + '" room at ' + serverUrl + '...');
},
disconnect() {
console.log('■ Disconnected from "' + roomId + '" room at ' + serverUrl);
}
};
}
```

```css
input { margin-bottom: 10px; }
button { margin-left: 5px; }
```

</Sandpack>

<Solution />

Passing an empty dependency array {/*passing-an-empty-dependency-array*/}

If your Effect truly doesn't use any reactive values, it will only run **after the initial render.**

```js {3}
useEffect(() => {
// ...
}, []); // Does not run again (except once in development)
```

Even with empty dependencies, setup and cleanup will [run one extra time in development](/learn/synchronizing-with-effects#how-to-handle-the-effect-firing-twice-in-development) to help you find bugs.

In this example, both `serverUrl` and `roomId` are hardcoded. Since they're declared outside the component, they are not reactive values, and so they aren't dependencies. The dependency list is
```

empty, so the Effect doesn't re-run on re-renders.

<Sandpack>

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

const serverUrl = 'https://localhost:1234';
const roomId = 'music';

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

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

return (
<>
<h1>Welcome to the {roomId} room!</h1>
<label>
Your message:{' '}
<input value={message} onChange={e => setMessage(e.target.value)} />
</label>
</>
);
}

export default function App() {
const [show, setShow] = useState(false);
return (
<>
<button onClick={() => setShow(!show)}>
{show ? 'Close chat' : 'Open chat'}
</button>
{show && <hr />}
{show && <ChatRoom />}
```

</>

);

}

```

```js chat.js

export function createConnection(serverUrl, roomId) {

// A real implementation would actually connect to the server

return {

connect() {

console.log('■ Connecting to "' + roomId + '" room at ' + serverUrl + '...');

},

disconnect() {

console.log('■ Disconnected from "' + roomId + '" room at ' + serverUrl);

}

};

}

```

</Sandpack>

<Solution />

Passing no dependency array at all {/*passing-no-dependency-array-at-all*/}

If you pass no dependency array at all, your Effect runs **after every single render (and re-render)** of your component.

```js {3}

useEffect(() => {

// ...

}); // Always runs again

```

In this example, the Effect re-runs when you change `serverUrl` and `roomId`, which is sensible. However, it *also* re-runs when you change the `message`, which is probably undesirable. This is why usually you'll specify the dependency array.

<Sandpack>

```js

import { useState, useEffect } from 'react';

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

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

useEffect(() => {
const connection = createConnection(serverUrl, roomId);
connection.connect();
return () => {
connection.disconnect();
};
}); // No dependency array at all

return (
<>
<label>
Server URL:{' '}
<input
value={serverUrl}
onChange={e => setServerUrl(e.target.value)}
/>
</label>
<h1>Welcome to the {roomId} room!</h1>
<label>
Your message:{' '}
<input value={message} onChange={e => setMessage(e.target.value)} />
</label>
</>
);
}

export default function App() {
const [show, setShow] = useState(false);
const [roomId, setRoomId] = useState('general');
return (
<>
<label>
```

Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
</select>
<button onClick={() => setShow(!show)}>
{show ? 'Close chat' : 'Open chat'}
</button>
</label>
{show && <hr />}
{show && <ChatRoom roomId={roomId}/>}
</>
);
}
```

```js chat.js
export function createConnection(serverUrl, roomId) {
// A real implementation would actually connect to the server
return {
connect() {
console.log('■ Connecting to "' + roomId + '" room at ' + serverUrl + '...');
},
disconnect() {
console.log('■ Disconnected from "' + roomId + '" room at ' + serverUrl);
}
};
}
```

```css
input { margin-bottom: 10px; }
button { margin-left: 5px; }

```

</Sandpack>

<Solution />

</Recipes>

Updating state based on previous state from an Effect
{/*updating-state-based-on-previous-state-from-an-effect*/}

When you want to update state based on previous state from an Effect, you might run into a problem:

```js {6,9}
function Counter() {
const [count, setCount] = useState(0);

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

Since `count` is a reactive value, it must be specified in the list of dependencies. However, that causes the Effect to cleanup and setup again every time the `count` changes. This is not ideal.

To fix this, [pass the `c => c + 1` state updater](/reference/react/useState#updating-state-based-on-the-previous-state) to `setCount`:

<Sandpack>

```js
import { useState, useEffect } from 'react';

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

useEffect(() => {
const intervalId = setInterval(() => {
setCount(c => c + 1); // ■ Pass a state updater
```
```

```
}, 1000);
return () => clearInterval(intervalId);
}, []); // ■ Now count is not a dependency

return <h1>{count}</h1>;
}
```

```css
label {
display: block;
margin-top: 20px;
margin-bottom: 20px;
}

body {
min-height: 150px;
}
```

</Sandpack>

Now that you're passing `c => c + 1` instead of `count + 1`, [your Effect no longer needs to depend on `count`.](/learn/removing-effect-dependencies#are-you-reading-some-state-to-calculate-the-next-state) As a result of this fix, it won't need to cleanup and setup the interval again every time the `count` changes.

Removing unnecessary object dependencies {/*removing-unnecessary-object-dependencies*/}

If your Effect depends on an object or a function created during rendering, it might run too often. For example, this Effect re-connects after every render because the `options` object is [different for every render:](/learn/removing-effect-dependencies#does-some-reactive-value-change-unintentionally)

```js {6-9,12,15}
const serverUrl = 'https://localhost:1234';

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

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

```
};

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

Avoid using an object created during rendering as a dependency. Instead, create the object inside the Effect:

<Sandpack>

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

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

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

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

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

```
export default function App() {
const [roomId, setRoomId] = useState('general');
return (
<>
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
</select>
</label>
<hr />
<ChatRoom roomId={roomId} />
</>
);
}
```

```js chat.js
export function createConnection({ serverUrl, roomId }) {
// A real implementation would actually connect to the server
return {
connect() {
console.log('■ Connecting to "' + roomId + '" room at ' + serverUrl + '...');
},
disconnect() {
console.log('■ Disconnected from "' + roomId + '" room at ' + serverUrl);
}
};
}
```

```css
```

input { display: block; margin-bottom: 20px; }

button { margin-left: 10px; }

```

</Sandpack>

Now that you create the `options` object inside the Effect, the Effect itself only depends on the `roomId` string.

With this fix, typing into the input doesn't reconnect the chat. Unlike an object which gets re-created, a string like `roomId` doesn't change unless you set it to another value. [Read more about removing dependencies.](/learn/removing-effect-dependencies)

---

### Removing unnecessary function dependencies {/*removing-unnecessary-function-dependencies*/}

If your Effect depends on an object or a function created during rendering, it might run too often. For example, this Effect re-connects after every render because the `createOptions` function is [different for every render:](/learn/removing-effect-dependencies#does-some-reactive-value-change-unintentionally)

```js {4-9,12,16}

function ChatRoom({ roomId }) {

const [message, setMessage] = useState('');

function createOptions() { // ■ This function is created from scratch on every re-render

return {

serverUrl: serverUrl,

roomId: roomId

};

}

useEffect(() => {

const options = createOptions(); // It's used inside the Effect

const connection = createConnection();

connection.connect();

return () => connection.disconnect();

}, [createOptions]); // ■ As a result, these dependencies are always different on a re-render

// ...

```

By itself, creating a function from scratch on every re-render is not a problem. You don't need to optimize that. However, if you use it as a dependency of your Effect, it will cause your Effect to re-run after every re-render.
```

Avoid using a function created during rendering as a dependency. Instead, declare it inside the Effect:

<Sandpack>

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

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

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

useEffect(() => {
function createOptions() {
return {
serverUrl: serverUrl,
roomId: roomId
};
}

const options = createOptions();
const connection = createConnection(options);
connection.connect();
return () => connection.disconnect();
}, [roomId]);

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

export default function App() {
const [roomId, setRoomId] = useState('general');
return (
<>
<label>
Choose the chat room:{' '}
```

<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
</select>
</label>
<hr />
<ChatRoom roomId={roomId} />
</>
);
}
```

```js chat.js
export function createConnection({ serverUrl, roomId }) {
// A real implementation would actually connect to the server
return {
connect() {
console.log('■ Connecting to "' + roomId + '" room at ' + serverUrl + '...');
},
disconnect() {
console.log('■ Disconnected from "' + roomId + '" room at ' + serverUrl);
}
};
}
```

```css
input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }
```

</Sandpack>
```

Now that you define the `createOptions` function inside the Effect, the Effect itself only depends on the `roomId` string. With this fix, typing into the input doesn't reconnect the chat. Unlike a function which gets re-created, a string like `roomId` doesn't change unless you set it to another value. [Read more about removing dependencies.](/learn/removing-effect-dependencies)

Reading the latest props and state from an Effect
{/*reading-the-latest-props-and-state-from-an-effect*/}

<Wip>

This section describes an **experimental API that has not yet been released** in a stable version of React.

</Wip>

By default, when you read a reactive value from an Effect, you have to add it as a dependency. This ensures that your Effect "reacts" to every change of that value. For most dependencies, that's the behavior you want.

However, sometimes you'll want to read the *latest* props and state from an Effect without "reacting" to them. For example, imagine you want to log the number of the items in the shopping cart for every page visit:

```js {3}
function Page({ url, shoppingCart }) {
useEffect(() => {
logVisit(url, shoppingCart.length);
}, [url, shoppingCart]); // ■ All dependencies declared
// ...
}
```

What if you want to log a new page visit after every `url` change, but *not* if only the `shoppingCart` changes? You can't exclude `shoppingCart` from dependencies without breaking the [reactivity rules.](#specifying-reactive-dependencies) However, you can express that you *don't want* a piece of code to "react" to changes even though it is called from inside an Effect. [Declare an *Effect Event*](/learn/separating-events-from-effects#declaring-an-effect-event) with the [`useEffectEvent`](/reference/react/experimental_useEffectEvent) Hook, and move the code reading `shoppingCart` inside of it:

```js {2-4,7,8}
function Page({ url, shoppingCart }) {
const onVisit = useEffectEvent(visitedUrl => {
logVisit(visitedUrl, shoppingCart.length)
});
```

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

Effect Events are not reactive and must always be omitted from dependencies of your Effect. This is what lets you put non-reactive code (where you can read the latest value of some props and state) inside of them. By reading `shoppingCart` inside of `onVisit`, you ensure that `shoppingCart` won't re-run your Effect.

[Read more about how Effect Events let you separate reactive and non-reactive code.](/learn/separating-events-from-effects#reading-latest-props-and-state-with-effect-events)

Displaying different content on the server and the client {/*displaying-different-content-on-the-server-and-the-client*/}

If your app uses server rendering (either [directly](/reference/react-dom/server) or via a [framework](/learn/start-a-new-react-project#production-grade-react-frameworks)), your component will render in two different environments. On the server, it will render to produce the initial HTML. On the client, React will run the rendering code again so that it can attach your event handlers to that HTML. This is why, for [hydration](/reference/react-dom/client/hydrateRoot#hydrating-server-rendered-html) to work, your initial render output must be identical on the client and the server.

In rare cases, you might need to display different content on the client. For example, if your app reads some data from [`localStorage`](https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage), it can't possibly do that on the server. Here is how you could implement this:

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

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

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

}
```

While the app is loading, the user will see the initial render output. Then, when it's loaded and hydrated, your Effect will run and set `didMount` to `true`, triggering a re-render. This will switch to the client-only render output. Effects don't run on the server, so this is why `didMount` was `false` during the initial server render.

Use this pattern sparingly. Keep in mind that users with a slow connection will see the initial content for quite a bit of time--potentially, many seconds--so you don't want to make jarring changes to your component's appearance. In many cases, you can avoid the need for this by conditionally showing different things with CSS.

---

## Troubleshooting {/*troubleshooting*/}

### My Effect runs twice when the component mounts
{/*my-effect-runs-twice-when-the-component-mounts*/}

When Strict Mode is on, in development, React runs setup and cleanup one extra time before the actual setup.

This is a stress-test that verifies your Effect's logic is implemented correctly. If this causes visible issues, your cleanup function is missing some logic. The cleanup function should stop or undo whatever the setup function was doing. The rule of thumb is that the user shouldn't be able to distinguish between the setup being called once (as in production) and a setup → cleanup → setup sequence (as in development).

Read more about [how this helps find bugs](/learn/synchronizing-with-effects#step-3-add-cleanup-if-needed) and [how to fix your logic.](/learn/synchronizing-with-effects#how-to-handle-the-effect-firing-twice-in-development)

---

### My Effect runs after every re-render {/*my-effect-runs-after-every-re-render*/}

First, check that you haven't forgotten to specify the dependency array:

```js {3}
useEffect(() => {
// ...
}); // ■ No dependency array: re-runs after every render!
```

If you've specified the dependency array but your Effect still re-runs in a loop, it's because one of your dependencies is different on every re-render.

You can debug this problem by manually logging your dependencies to the console:
```

```js {5}
useEffect(() => {
// ..
}, [serverUrl, roomId]);

console.log([serverUrl, roomId]);
```

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

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

When you find the dependency that is different on every re-render, you can usually fix it in one of these ways:

- [Updating state based on previous state from an Effect](#updating-state-based-on-previous-state-from-an-effect)
- [Removing unnecessary object dependencies](#removing-unnecessary-object-dependencies)
- [Removing unnecessary function dependencies](#removing-unnecessary-function-dependencies)
- [Reading the latest props and state from an Effect](#reading-the-latest-props-and-state-from-an-effect)

As a last resort (if these methods didn't help), wrap its creation with [`useMemo`](/reference/react/useMemo#memoizing-a-dependency-of-another-hook) or [`useCallback`](/reference/react/useCallback#preventing-an-effect-from-firing-too-often) (for functions).

My Effect keeps re-running in an infinite cycle {/*my-effect-keeps-re-running-in-an-infinite-cycle*/}

If your Effect runs in an infinite cycle, these two things must be true:

- Your Effect is updating some state.
- That state leads to a re-render, which causes the Effect's dependencies to change.

Before you start fixing the problem, ask yourself whether your Effect is connecting to some external system (like DOM, network, a third-party widget, and so on). Why does your Effect need to set state? Does it synchronize with that external system? Or are you trying to manage your application's data flow with it?

If there is no external system, consider whether [removing the Effect altogether](/learn/you-might-not-need-an-effect) would simplify your logic.

If you're genuinely synchronizing with some external system, think about why and under what conditions your Effect should update the state. Has something changed that affects your component's visual output? If you need to keep track of some data that isn't used by rendering, a [ref](/reference/react/useRef#referencing-a-value-with-a-ref) (which doesn't trigger re-renders) might be more appropriate. Verify your Effect doesn't update the state (and trigger re-renders) more than needed.

Finally, if your Effect is updating the state at the right time, but there is still a loop, it's because that state update leads to one of the Effect's dependencies changing. [Read how to debug dependency changes.](/reference/react/useEffect#my-effect-runs-after-every-re-render)

My cleanup logic runs even though my component didn't unmount {/*my-cleanup-logic-runs-even-though-my-component-didnt-unmount*/}

The cleanup function runs not only during unmount, but before every re-render with changed dependencies. Additionally, in development, React [runs setup+cleanup one extra time immediately after component mounts.](#my-effect-runs-twice-when-the-component-mounts)

If you have cleanup code without corresponding setup code, it's usually a code smell:

```js {2-5}
useEffect(() => {
// ■ Avoid: Cleanup logic without corresponding setup logic
return () => {
doSomething();
};
}, []);
```

Your cleanup logic should be "symmetrical" to the setup logic, and should stop or undo whatever setup did:

```js {2-3,5}
useEffect(() => {
const connection = createConnection(serverUrl, roomId);
connection.connect();
return () => {
connection.disconnect();
};
}, [serverUrl, roomId]);
```

```

[Learn how the Effect lifecycle is different from the component's lifecycle.](/learn/lifecycle-of-reactive-effects#the-lifecycle-of-an-effect)

---

### My Effect does something visual, and I see a flicker before it runs {/*my-effect-does-something-visual-and-i-see-a-flicker-before-it-runs*/}

If your Effect must block the browser from [painting the screen,](/learn/render-and-commit#epilogue-browser-paint) replace `useEffect` with [`useLayoutEffect`](/reference/react/useLayoutEffect). Note that **this shouldn't be needed for the vast majority of Effects.** You'll only need this if it's crucial to run your Effect before the browser paint: for example, to measure and position a tooltip before the user sees it.

---

title: useContext

---

<Intro>

`useContext` is a React Hook that lets you read and subscribe to [context](/learn/passing-data-deeply-with-context) from your component.

```js
const value = useContext(SomeContext)
```

</Intro>

<InlineToc />

Reference {/*reference*/}

`useContext(SomeContext)` {/*usecontext*/}

Call `useContext` at the top level of your component to read and subscribe to [context.](/learn/passing-data-deeply-with-context)

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

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

[See more examples below.](#usage)

Parameters {/*parameters*/}

* `SomeContext`: The context that you've previously created with [`createContext`](/reference/react/createContext). The context itself does not hold the information, it only represents the kind of information you can provide or read from components.

Returns {/*returns*/}

`useContext` returns the context value for the calling component. It is determined as the `value` passed to the closest `SomeContext.Provider` above the calling component in the tree. If there is no such provider, then the returned value will be the `defaultValue` you have passed to [`createContext`](/reference/react/createContext) for that context. The returned value is always up-to-date. React automatically re-renders components that read some context if it changes.

Caveats {/*caveats*/}

* `useContext()` call in a component is not affected by providers returned from the *same* component. The corresponding `<Context.Provider>` **needs to be *above*** the component doing the `useContext()` call.

* React **automatically re-renders** all the children that use a particular context starting from the provider that receives a different `value`. The previous and the next values are compared with the [`Object.is`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/is) comparison. Skipping re-renders with [`memo`](/reference/react/memo) does not prevent the children receiving fresh context values.

* If your build system produces duplicates modules in the output (which can happen with symlinks), this can break context. Passing something via context only works if `SomeContext` that you use to provide context and `SomeContext` that you use to read it are ***exactly* the same object**, as determined by a `===` comparison.

Usage {/*usage*/}

Passing data deeply into the tree {/*passing-data-deeply-into-the-tree*/}

Call `useContext` at the top level of your component to read and subscribe to [context.](/learn/passing-data-deeply-with-context)

```js [[2, 4, "theme"], [1, 4, "ThemeContext"]]
import { useContext } from 'react';

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

`useContext` returns the <CodeStep step={2}>context value</CodeStep> for the <CodeStep step={1}>context</CodeStep> you passed. To determine the context value, React searches the component tree and finds **the closest context provider above** for that particular context.

To pass context to a `Button`, wrap it or one of its parent components into the corresponding context provider:

```js [[1, 3, "ThemeContext"], [2, 3, "\\"dark\\""], [1, 5, "ThemeContext"]]
function MyPage() {
return (
<ThemeContext.Provider value="dark">
<Form />
</ThemeContext.Provider>
);
}

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

It doesn't matter how many layers of components there are between the provider and the `Button`. When a `Button` *anywhere* inside of `Form` calls `useContext(ThemeContext)`, it will receive `"dark"` as the value.

<Pitfall>

`useContext()` always looks for the closest provider *above* the component that calls it. It searches upwards and **does not** consider providers in the component from which you're calling `useContext()`.

</Pitfall>

<Sandpack>

```js
import { createContext, useContext } from 'react';

const ThemeContext = createContext(null);

export default function MyApp() {
return (
<ThemeContext.Provider value="dark">
<Form />
</ThemeContext.Provider>
```

```
)
}

function Form() {
return (
<Panel title="Welcome">
<Button>Sign up</Button>
<Button>Log in</Button>
</Panel>
);
}

function Panel({ title, children }) {
const theme = useContext(ThemeContext);
const className = 'panel-' + theme;
return (
<section className={className}>
<h1>{title}</h1>
{children}
</section>
)
}

function Button({ children }) {
const theme = useContext(ThemeContext);
const className = 'button-' + theme;
return (
<button className={className}>
{children}
</button>
);
}
```

```css
.panel-light,
.panel-dark {
border: 1px solid black;
```

```
border-radius: 4px;
padding: 20px;
}
.panel-light {
color: #222;
background: #fff;
}

.panel-dark {
color: #fff;
background: rgb(23, 32, 42);
}

.button-light,
.button-dark {
border: 1px solid #777;
padding: 5px;
margin-right: 10px;
margin-top: 10px;
}

.button-dark {
background: #222;
color: #fff;
}

.button-light {
background: #fff;
color: #222;
}
```

</Sandpack>

Updating data passed via context {/*updating-data-passed-via-context*/}

Often, you'll want the context to change over time. To update context, combine it with [state.](/reference/react/useState) Declare a state variable in the parent component, and pass the current state down as the <CodeStep step={2}>context value</CodeStep> to the provider.

```js {2} [[1, 4, "ThemeContext"], [2, 4, "theme"], [1, 11, "ThemeContext"]]
function MyPage() {
const [theme, setTheme] = useState('dark');
return (
<ThemeContext.Provider value={theme}>
<Form />
<Button onClick={() => {
setTheme('light');
}}>
Switch to light theme
</Button>
</ThemeContext.Provider>
);
}
```

Now any `Button` inside of the provider will receive the current `theme` value. If you call `setTheme` to update the `theme` value that you pass to the provider, all `Button` components will re-render with the new `'light'` value.

<Recipes titleText="Examples of updating context" titleId="examples-basic">

Updating a value via context {/*updating-a-value-via-context*/}

In this example, the `MyApp` component holds a state variable which is then passed to the `ThemeContext` provider. Checking the "Dark mode" checkbox updates the state. Changing the provided value re-renders all the components using that context.

<Sandpack>

```js
import { createContext, useContext, useState } from 'react';

const ThemeContext = createContext(null);

export default function MyApp() {
const [theme, setTheme] = useState('light');
return (
<ThemeContext.Provider value={theme}>
<Form />
<label>
<input
```

```
type="checkbox"
checked={theme === 'dark'}
onChange={(e) => {
setTheme(e.target.checked ? 'dark' : 'light')
}}
/>
Use dark mode
</label>
</ThemeContext.Provider>
)
}

function Form({ children }) {
return (
<Panel title="Welcome">
<Button>Sign up</Button>
<Button>Log in</Button>
</Panel>
);
}

function Panel({ title, children }) {
const theme = useContext(ThemeContext);
const className = 'panel-' + theme;
return (
<section className={className}>
<h1>{title}</h1>
{children}
</section>
)
}

function Button({ children }) {
const theme = useContext(ThemeContext);
const className = 'button-' + theme;
return (
<button className={className}>
```

{children}
</button>
);
}
```

```css
.panel-light,
.panel-dark {
border: 1px solid black;
border-radius: 4px;
padding: 20px;
margin-bottom: 10px;
}
.panel-light {
color: #222;
background: #fff;
}

.panel-dark {
color: #fff;
background: rgb(23, 32, 42);
}

.button-light,
.button-dark {
border: 1px solid #777;
padding: 5px;
margin-right: 10px;
margin-top: 10px;
}

.button-dark {
background: #222;
color: #fff;
}

.button-light {
background: #fff;

color: #222;

}

```

</Sandpack>

Note that `value="dark"` passes the `"dark"` string, but `value={theme}` passes the value of the JavaScript `theme` variable with [JSX curly braces.](/learn/javascript-in-jsx-with-curly-braces) Curly braces also let you pass context values that aren't strings.

<Solution />

Updating an object via context {/*updating-an-object-via-context*/}

In this example, there is a `currentUser` state variable which holds an object. You combine `{ currentUser, setCurrentUser }` into a single object and pass it down through the context inside the `value={}`. This lets any component below, such as `LoginButton`, read both `currentUser` and `setCurrentUser`, and then call `setCurrentUser` when needed.

<Sandpack>

```js
import { createContext, useContext, useState } from 'react';

const CurrentUserContext = createContext(null);

export default function MyApp() {
const [currentUser, setCurrentUser] = useState(null);
return (
<CurrentUserContext.Provider
value={{
currentUser,
setCurrentUser
}}
>
<Form />
</CurrentUserContext.Provider>
);
}

function Form({ children }) {
return (
<Panel title="Welcome">
<LoginButton />

```
</Panel>
);
}

function LoginButton() {
const {
currentUser,
setCurrentUser
} = useContext(CurrentUserContext);

if (currentUser !== null) {
return <p>You logged in as {currentUser.name}.</p>;
}

return (
<Button onClick={() => {
setCurrentUser({ name: 'Advika' })
}}>Log in as Advika</Button>
);
}

function Panel({ title, children }) {
return (
<section className="panel">
<h1>{title}</h1>
{children}
</section>
)
}

function Button({ children, onClick }) {
return (
<button className="button" onClick={onClick}>
{children}
</button>
);
}
```

```css
label {
display: block;
}

.panel {
border: 1px solid black;
border-radius: 4px;
padding: 20px;
margin-bottom: 10px;
}

.button {
border: 1px solid #777;
padding: 5px;
margin-right: 10px;
margin-top: 10px;
}
```

</Sandpack>

<Solution />

Multiple contexts {/*multiple-contexts*/}

In this example, there are two independent contexts. `ThemeContext` provides the current theme, which is a string, while `CurrentUserContext` holds the object representing the current user.

<Sandpack>

```js
import { createContext, useContext, useState } from 'react';

const ThemeContext = createContext(null);
const CurrentUserContext = createContext(null);

export default function MyApp() {
const [theme, setTheme] = useState('light');
const [currentUser, setCurrentUser] = useState(null);
return (
<ThemeContext.Provider value={theme}>
```

```
<CurrentUserContext.Provider
value={{
currentUser,
setCurrentUser
}}
>
<WelcomePanel />
<label>
<input
type="checkbox"
checked={theme === 'dark'}
onChange={(e) => {
setTheme(e.target.checked ? 'dark' : 'light')
}}
/>
Use dark mode
</label>
</CurrentUserContext.Provider>
</ThemeContext.Provider>
)
}

function WelcomePanel({ children }) {
const {currentUser} = useContext(CurrentUserContext);
return (
<Panel title="Welcome">
{currentUser !== null ?
<Greeting /> :
<LoginForm />
}
</Panel>
);
}

function Greeting() {
const {currentUser} = useContext(CurrentUserContext);
return (
```

```
<p>You logged in as {currentUser.name}.</p>
)
}

function LoginForm() {
const {setCurrentUser} = useContext(CurrentUserContext);
const [firstName, setFirstName] = useState('');
const [lastName, setLastName] = useState('');
const canLogin = firstName !== '' && lastName !== '';
return (
<>
<label>
First name{': '}
<input
required
value={firstName}
onChange={e => setFirstName(e.target.value)}
/>
</label>
<label>
Last name{': '}
<input
required
value={lastName}
onChange={e => setLastName(e.target.value)}
/>
</label>
<Button
disabled={!canLogin}
onClick={() => {
setCurrentUser({
name: firstName + ' ' + lastName
});
}}
>
Log in
```

```
</Button>
{!canLogin && <i>Fill in both fields.</i>}
</>
);
}

function Panel({ title, children }) {
const theme = useContext(ThemeContext);
const className = 'panel-' + theme;
return (
<section className={className}>
<h1>{title}</h1>
{children}
</section>
)
}

function Button({ children, disabled, onClick }) {
const theme = useContext(ThemeContext);
const className = 'button-' + theme;
return (
<button
className={className}
disabled={disabled}
onClick={onClick}
>
{children}
</button>
);
}
```

```css
label {
display: block;
}

.panel-light,
```

```
.panel-dark {
border: 1px solid black;
border-radius: 4px;
padding: 20px;
margin-bottom: 10px;
}
.panel-light {
color: #222;
background: #fff;
}

.panel-dark {
color: #fff;
background: rgb(23, 32, 42);
}

.button-light,
.button-dark {
border: 1px solid #777;
padding: 5px;
margin-right: 10px;
margin-top: 10px;
}

.button-dark {
background: #222;
color: #fff;
}

.button-light {
background: #fff;
color: #222;
}
```

</Sandpack>

<Solution />

Extracting providers to a component {/*extracting-providers-to-a-component*/}

As your app grows, it is expected that you'll have a "pyramid" of contexts closer to the root of your app. There is nothing wrong with that. However, if you dislike the nesting aesthetically, you can extract the providers into a single component. In this example, `MyProviders` hides the "plumbing" and renders the children passed to it inside the necessary providers. Note that the `theme` and `setTheme` state is needed in `MyApp` itself, so `MyApp` still owns that piece of the state.

<Sandpack>

```js
import { createContext, useContext, useState } from 'react';

const ThemeContext = createContext(null);
const CurrentUserContext = createContext(null);

export default function MyApp() {
const [theme, setTheme] = useState('light');
return (
<MyProviders theme={theme} setTheme={setTheme}>
<WelcomePanel />
<label>
<input
type="checkbox"
checked={theme === 'dark'}
onChange={(e) => {
setTheme(e.target.checked ? 'dark' : 'light')
}}
/>
Use dark mode
</label>
</MyProviders>
);
}

function MyProviders({ children, theme, setTheme }) {
const [currentUser, setCurrentUser] = useState(null);
return (
<ThemeContext.Provider value={theme}>
<CurrentUserContext.Provider
value={{
currentUser,
```

```
setCurrentUser
}}
>
{children}
</CurrentUserContext.Provider>
</ThemeContext.Provider>
);
}

function WelcomePanel({ children }) {
const {currentUser} = useContext(CurrentUserContext);
return (
<Panel title="Welcome">
{currentUser !== null ?
<Greeting /> :
<LoginForm />
}
</Panel>
);
}

function Greeting() {
const {currentUser} = useContext(CurrentUserContext);
return (
<p>You logged in as {currentUser.name}.</p>
)
}

function LoginForm() {
const {setCurrentUser} = useContext(CurrentUserContext);
const [firstName, setFirstName] = useState('');
const [lastName, setLastName] = useState('');
const canLogin = firstName !== '' && lastName !== '';
return (
<>
<label>
First name{': '}
```

```
<input
required
value={firstName}
onChange={e => setFirstName(e.target.value)}
/>
</label>
<label>
Last name{': '}
<input
required
value={lastName}
onChange={e => setLastName(e.target.value)}
/>
</label>
<Button
disabled={!canLogin}
onClick={() => {
setCurrentUser({
name: firstName + ' ' + lastName
});
}}
>
Log in
</Button>
{!canLogin && <i>Fill in both fields.</i>}
</>
);
}

function Panel({ title, children }) {
const theme = useContext(ThemeContext);
const className = 'panel-' + theme;
return (
<section className={className}>
<h1>{title}</h1>
{children}
```

```
</section>
)
}

function Button({ children, disabled, onClick }) {
const theme = useContext(ThemeContext);
const className = 'button-' + theme;
return (
<button
className={className}
disabled={disabled}
onClick={onClick}
>
{children}
</button>
);
}
```

```css
label {
display: block;
}

.panel-light,
.panel-dark {
border: 1px solid black;
border-radius: 4px;
padding: 20px;
margin-bottom: 10px;
}
.panel-light {
color: #222;
background: #fff;
}

.panel-dark {
color: #fff;
```

background: rgb(23, 32, 42);

}

.button-light,

.button-dark {

border: 1px solid #777;

padding: 5px;

margin-right: 10px;

margin-top: 10px;

}

.button-dark {

background: #222;

color: #fff;

}

.button-light {

background: #fff;

color: #222;

}

```

</Sandpack>

<Solution />

#### Scaling up with context and a reducer {/*scaling-up-with-context-and-a-reducer*/}

In larger apps, it is common to combine context with a [reducer](/reference/react/useReducer) to extract the logic related to some state out of components. In this example, all the "wiring" is hidden in the `TasksContext.js`, which contains a reducer and two separate contexts.

Read a [full walkthrough](/learn/scaling-up-with-reducer-and-context) of this example.

<Sandpack>

```js App.js

import AddTask from './AddTask.js';

import TaskList from './TaskList.js';

import { TasksProvider } from './TasksContext.js';

export default function TaskApp() {

return (

```
<TasksProvider>
<h1>Day off in Kyoto</h1>
<AddTask />
<TaskList />
</TasksProvider>
);
}
```

```js TasksContext.js
import { createContext, useContext, useReducer } from 'react';

const TasksContext = createContext(null);

const TasksDispatchContext = createContext(null);

export function TasksProvider({ children }) {
const [tasks, dispatch] = useReducer(
tasksReducer,
initialTasks
);

return (
<TasksContext.Provider value={tasks}>
<TasksDispatchContext.Provider value={dispatch}>
{children}
</TasksDispatchContext.Provider>
</TasksContext.Provider>
);
}

export function useTasks() {
return useContext(TasksContext);
}

export function useTasksDispatch() {
return useContext(TasksDispatchContext);
}

function tasksReducer(tasks, action) {
switch (action.type) {
```

```
case 'added': {
return [...tasks, {
id: action.id,
text: action.text,
done: false
}];
}
case 'changed': {
return tasks.map(t => {
if (t.id === action.task.id) {
return action.task;
} else {
return t;
}
});
}
case 'deleted': {
return tasks.filter(t => t.id !== action.id);
}
default: {
throw Error('Unknown action: ' + action.type);
}
}
}

const initialTasks = [
{ id: 0, text: 'Philosopher’s Path', done: true },
{ id: 1, text: 'Visit the temple', done: false },
{ id: 2, text: 'Drink matcha', done: false }
];
```

```js AddTask.js
import { useState, useContext } from 'react';
import { useTasksDispatch } from './TasksContext.js';

export default function AddTask() {
```

```
const [text, setText] = useState('');
const dispatch = useTasksDispatch();
return (
<>
<input
placeholder="Add task"
value={text}
onChange={e => setText(e.target.value)}
/>
<button onClick={() => {
setText('');
dispatch({
type: 'added',
id: nextId++,
text: text,
});
}}>Add</button>
</>
);
}

let nextId = 3;
```

```js TaskList.js
import { useState, useContext } from 'react';
import { useTasks, useTasksDispatch } from './TasksContext.js';

export default function TaskList() {
const tasks = useTasks();
return (
<ul>
{tasks.map(task => (
<li key={task.id}>
<Task task={task} />
</li>
))}
```

```
</ul>
);
}

function Task({ task }) {
const [isEditing, setIsEditing] = useState(false);
const dispatch = useTasksDispatch();
let taskContent;
if (isEditing) {
taskContent = (
<>
<input
value={task.text}
onChange={e => {
dispatch({
type: 'changed',
task: {
...task,
text: e.target.value
}
});
}} />
<button onClick={() => setIsEditing(false)}>
Save
</button>
</>
);
} else {
taskContent = (
<>
{task.text}
<button onClick={() => setIsEditing(true)}>
Edit
</button>
</>
);
```

```
}
return (
<label>
<input
type="checkbox"
checked={task.done}
onChange={e => {
dispatch({
type: 'changed',
task: {
...task,
done: e.target.checked
}
});
}}
/>
{taskContent}
<button onClick={() => {
dispatch({
type: 'deleted',
id: task.id
});
}}>
Delete
</button>
</label>
);
}
```

```css
button { margin: 5px; }
li { list-style-type: none; }
ul, li { margin: 0; padding: 0; }
```

</Sandpack>

<Solution />

</Recipes>

Specifying a fallback default value {/*specifying-a-fallback-default-value*/}

If React can't find any providers of that particular <CodeStep step={1}>context</CodeStep> in the parent tree, the context value returned by `useContext()` will be equal to the <CodeStep step={3}>default value</CodeStep> that you specified when you [created that context](/reference/react/createContext):

```js [[1, 1, "ThemeContext"], [3, 1, "null"]]
const ThemeContext = createContext(null);
```

The default value **never changes**. If you want to update context, use it with state as [described above.](#updating-data-passed-via-context)

Often, instead of `null`, there is some more meaningful value you can use as a default, for example:

```js [[1, 1, "ThemeContext"], [3, 1, "light"]]
const ThemeContext = createContext('light');
```

This way, if you accidentally render some component without a corresponding provider, it won't break. This also helps your components work well in a test environment without setting up a lot of providers in the tests.

In the example below, the "Toggle theme" button is always light because it's **outside any theme context provider** and the default context theme value is `'light'`. Try editing the default theme to be `'dark'`.

<Sandpack>

```js
import { createContext, useContext, useState } from 'react';

const ThemeContext = createContext('light');

export default function MyApp() {
const [theme, setTheme] = useState('light');
return (
<>
<ThemeContext.Provider value={theme}>
<Form />
```

```
</ThemeContext.Provider>
<Button onClick={() => {
setTheme(theme === 'dark' ? 'light' : 'dark');
}}>
Toggle theme
</Button>
</>
)
}

function Form({ children }) {
return (
<Panel title="Welcome">
<Button>Sign up</Button>
<Button>Log in</Button>
</Panel>
);
}

function Panel({ title, children }) {
const theme = useContext(ThemeContext);
const className = 'panel-' + theme;
return (
<section className={className}>
<h1>{title}</h1>
{children}
</section>
)
}

function Button({ children, onClick }) {
const theme = useContext(ThemeContext);
const className = 'button-' + theme;
return (
<button className={className} onClick={onClick}>
{children}
</button>
```

);
}
```

```css
.panel-light,
.panel-dark {
border: 1px solid black;
border-radius: 4px;
padding: 20px;
margin-bottom: 10px;
}
.panel-light {
color: #222;
background: #fff;
}

.panel-dark {
color: #fff;
background: rgb(23, 32, 42);
}

.button-light,
.button-dark {
border: 1px solid #777;
padding: 5px;
margin-right: 10px;
margin-top: 10px;
}

.button-dark {
background: #222;
color: #fff;
}

.button-light {
background: #fff;
color: #222;
}

```

</Sandpack>

Overriding context for a part of the tree {/*overriding-context-for-a-part-of-the-tree*/}

You can override the context for a part of the tree by wrapping that part in a provider with a different value.

```js {3,5}
<ThemeContext.Provider value="dark">
...
<ThemeContext.Provider value="light">
<Footer />
</ThemeContext.Provider>
...
</ThemeContext.Provider>
```

You can nest and override providers as many times as you need.

<Recipes title="Examples of overriding context">

Overriding a theme {/*overriding-a-theme*/}

Here, the button *inside* the `Footer` receives a different context value (`"light"`) than the buttons outside (`"dark"`).

<Sandpack>

```js
import { createContext, useContext } from 'react';

const ThemeContext = createContext(null);

export default function MyApp() {
return (
<ThemeContext.Provider value="dark">
<Form />
</ThemeContext.Provider>
)
}
```
```

```
function Form() {
return (
<Panel title="Welcome">
<Button>Sign up</Button>
<Button>Log in</Button>
<ThemeContext.Provider value="light">
<Footer />
</ThemeContext.Provider>
</Panel>
);
}

function Footer() {
return (
<footer>
<Button>Settings</Button>
</footer>
);
}

function Panel({ title, children }) {
const theme = useContext(ThemeContext);
const className = 'panel-' + theme;
return (
<section className={className}>
{title && <h1>{title}</h1>}
{children}
</section>
)
}

function Button({ children }) {
const theme = useContext(ThemeContext);
const className = 'button-' + theme;
return (
<button className={className}>
{children}
```

</button>
);
}
```

```css
footer {
margin-top: 20px;
border-top: 1px solid #aaa;
}

.panel-light,
.panel-dark {
border: 1px solid black;
border-radius: 4px;
padding: 20px;
}
.panel-light {
color: #222;
background: #fff;
}

.panel-dark {
color: #fff;
background: rgb(23, 32, 42);
}

.button-light,
.button-dark {
border: 1px solid #777;
padding: 5px;
margin-right: 10px;
margin-top: 10px;
}

.button-dark {
background: #222;
color: #fff;
}

.button-light {

background: #fff;

color: #222;

}

```

</Sandpack>

<Solution />

Automatically nested headings {/*automatically-nested-headings*/}

You can "accumulate" information when you nest context providers. In this example, the `Section` component keeps track of the `LevelContext` which specifies the depth of the section nesting. It reads the `LevelContext` from the parent section, and provides the `LevelContext` number increased by one to its children. As a result, the `Heading` component can automatically decide which of the `<h1>`, `<h2>`, `<h3>`, ..., tags to use based on how many `Section` components it is nested inside of.

Read a [detailed walkthrough](/learn/passing-data-deeply-with-context) of this example.

<Sandpack>

```js

import Heading from './Heading.js';

import Section from './Section.js';

export default function Page() {

return (

<Section>

<Heading>Title</Heading>

<Section>

<Heading>Heading</Heading>

<Heading>Heading</Heading>

<Heading>Heading</Heading>

<Section>

<Heading>Sub-heading</Heading>

<Heading>Sub-heading</Heading>

<Heading>Sub-heading</Heading>

<Section>

<Heading>Sub-sub-heading</Heading>

<Heading>Sub-sub-heading</Heading>

<Heading>Sub-sub-heading</Heading>

```
</Section>
</Section>
</Section>
</Section>
);
}
```

```js Section.js
import { useContext } from 'react';
import { LevelContext } from './LevelContext.js';

export default function Section({ children }) {
const level = useContext(LevelContext);
return (
<section className="section">
<LevelContext.Provider value={level + 1}>
{children}
</LevelContext.Provider>
</section>
);
}
```

```js Heading.js
import { useContext } from 'react';
import { LevelContext } from './LevelContext.js';

export default function Heading({ children }) {
const level = useContext(LevelContext);
switch (level) {
case 0:
throw Error('Heading must be inside a Section!');
case 1:
return <h1>{children}</h1>;
case 2:
return <h2>{children}</h2>;
case 3:
```

return <h3>{children}</h3>;
case 4:
return <h4>{children}</h4>;
case 5:
return <h5>{children}</h5>;
case 6:
return <h6>{children}</h6>;
default:
throw Error('Unknown level: ' + level);
}
}
```

```js LevelContext.js
import { createContext } from 'react';

export const LevelContext = createContext(0);
```

```css
.section {
padding: 10px;
margin: 5px;
border-radius: 5px;
border: 1px solid #aaa;
}
```

</Sandpack>

<Solution />

</Recipes>

Optimizing re-renders when passing objects and functions
{/*optimizing-re-renders-when-passing-objects-and-functions*/}

You can pass any values via context, including objects and functions.

```js [[2, 10, "{ currentUser, login }"]]

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

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

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

Here, the <CodeStep step={2}>context value</CodeStep> is a JavaScript object with two properties, one of which is a function. Whenever `MyApp` re-renders (for example, on a route update), this will be a *different* object pointing at a *different* function, so React will also have to re-render all components deep in the tree that call `useContext(AuthContext)`.

In smaller apps, this is not a problem. However, there is no need to re-render them if the underlying data, like `currentUser`, has not changed. To help React take advantage of that fact, you may wrap the `login` function with [`useCallback`](/reference/react/useCallback) and wrap the object creation into [`useMemo`](/reference/react/useMemo). This is a performance optimization:

```js {6,9,11,14,17}
import { useCallback, useMemo } from 'react';

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

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

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

return (
```

<AuthContext.Provider value={contextValue}>

<Page />

</AuthContext.Provider>

);

}

```

As a result of this change, even if `MyApp` needs to re-render, the components calling `useContext(AuthContext)` won't need to re-render unless `currentUser` has changed.

Read more about [`useMemo`](/reference/react/useMemo#skipping-re-rendering-of-components) and [`useCallback`.](/reference/react/useCallback#skipping-re-rendering-of-components)

---

## Troubleshooting {/*troubleshooting*/}

### My component doesn't see the value from my provider {/*my-component-doesnt-see-the-value-from-my-provider*/}

There are a few common ways that this can happen:

1. You're rendering `<SomeContext.Provider>` in the same component (or below) as where you're calling `useContext()`. Move `<SomeContext.Provider>` *above and outside* the component calling `useContext()`.

2. You may have forgotten to wrap your component with `<SomeContext.Provider>`, or you might have put it in a different part of the tree than you thought. Check whether the hierarchy is right using [React DevTools.](/learn/react-developer-tools)

3. You might be running into some build issue with your tooling that causes `SomeContext` as seen from the providing component and `SomeContext` as seen by the reading component to be two different objects. This can happen if you use symlinks, for example. You can verify this by assigning them to globals like `window.SomeContext1` and `window.SomeContext2` and then checking whether `window.SomeContext1 === window.SomeContext2` in the console. If they're not the same, fix that issue on the build tool level.

### I am always getting `undefined` from my context although the default value is different {/*i-am-always-getting-undefined-from-my-context-although-the-default-value-is-different*/}

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

```js {1,2}

// ■ Doesn't work: no value prop

<ThemeContext.Provider>

<Button />

</ThemeContext.Provider>

```
```

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

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

```js {1,2}
// ■ Doesn't work: prop should be called "value"
<ThemeContext.Provider theme={theme}>
<Button />
</ThemeContext.Provider>
```

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

```js {1,2}
// ■ Passing the value prop
<ThemeContext.Provider value={theme}>
<Button />
</ThemeContext.Provider>
```

Note that the [default value from your `createContext(defaultValue)`
call](#specifying-a-fallback-default-value) is only used **if there is no matching provider above at all.** If there is a `<SomeContext.Provider value={undefined}>` component somewhere in the parent tree, the component calling `useContext(SomeContext)` *will* receive `undefined` as the context value.

---

title: useMemo

<Intro>

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

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

</Intro>

<InlineToc />

Reference {/*reference*/}

`useMemo(calculateValue, dependencies)` {/*usememo*/}

Call `useMemo` at the top level of your component to cache a calculation between re-renders:

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

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

[See more examples below.](#usage)

Parameters {/*parameters*/}

* `calculateValue`: The function calculating the value that you want to cache. It should be pure, should take no arguments, and should return a value of any type. React will call your function during the initial render. On next renders, React will return the same value again if the `dependencies` have not changed since the last render. Otherwise, it will call `calculateValue`, return its result, and store it so it can be reused later.

* `dependencies`: The list of all reactive values referenced inside of the `calculateValue` code. Reactive values include props, state, and all the variables and functions declared directly inside your component body. If your linter is [configured for React](/learn/editor-setup#linting), it will verify that every reactive value is correctly specified as a dependency. The list of dependencies must have a constant number of items and be written inline like `[dep1, dep2, dep3]`. React will compare each dependency with its previous value using the [`Object.is`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/is) comparison.

Returns {/*returns*/}

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

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

Caveats {/*caveats*/}

* `useMemo` is a Hook, so you can only call it **at the top level of your component** or your own Hooks. You can't call it inside loops or conditions. If you need that, extract a new component and move the state into it.

* In Strict Mode, React will **call your calculation function twice** in order to [help you find accidental impurities.](#my-calculation-runs-twice-on-every-re-render) This is development-only behavior and does not affect production. If your calculation function is pure (as it should be), this should not affect your logic. The result from one of the calls will be ignored.

* React **will not throw away the cached value unless there is a specific reason to do that.** For example, in development, React throws away the cache when you edit the file of your component. Both in development and in production, React will throw away the cache if your component suspends during the initial mount. In the future, React may add more features that take advantage of throwing away the cache--for example, if React adds built-in support for virtualized lists in the future, it would make sense to throw away the cache for items that scroll out of the virtualized table viewport. This should be fine if you rely on `useMemo` solely as a performance optimization. Otherwise, a [state variable](/reference/react/useState#avoiding-recreating-the-initial-state) or a [ref](/reference/react/useRef#avoiding-recreating-the-ref-contents) may be more appropriate.

<Note>

Caching return values like this is also known as [*memoization*,](https://en.wikipedia.org/wiki/Memoization) which is why this Hook is called `useMemo`.

</Note>

Usage {/*usage*/}

Skipping expensive recalculations {/*skipping-expensive-recalculations*/}

To cache a calculation between re-renders, wrap it in a `useMemo` call at the top level of your component:

```js [[3, 4, "visibleTodos"], [1, 4, "() => filterTodos(todos, tab)"], [2, 4, "[todos, tab]"]]
import { useMemo } from 'react';

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

You need to pass two things to `useMemo`:

1. A <CodeStep step={1}>calculation function</CodeStep> that takes no arguments, like `() =>`, and returns what you wanted to calculate.
2. A <CodeStep step={2}>list of dependencies</CodeStep> including every value within your component that's used inside your calculation.

On the initial render, the <CodeStep step={3}>value</CodeStep> you'll get from `useMemo` will be the result of calling your <CodeStep step={1}>calculation</CodeStep>.

On every subsequent render, React will compare the <CodeStep step={2}>dependencies</CodeStep> with the dependencies you passed during the last render. If none of the dependencies have changed (compared with [`Object.is`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/is)), `useMemo` will return the value you already calculated before. Otherwise, React will re-run your calculation and return the new value.

In other words, `useMemo` caches a calculation result between re-renders until its dependencies change.

Let's walk through an example to see when this is useful.

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

```js {2}
function TodoList({ todos, tab, theme }) {
const visibleTodos = filterTodos(todos, tab);
// ...
}
```

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

This type of caching is called *[memoization.](https://en.wikipedia.org/wiki/Memoization)*

<Note>

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

</Note>

<DeepDive>

How to tell if a calculation is expensive? {/*how-to-tell-if-a-calculation-is-expensive*/}

In general, unless you're creating or looping over thousands of objects, it's probably not expensive. If you want to get more confidence, you can add a console log to measure the time spent in a piece of code:

```js {1,3}
console.time('filter array');
const visibleTodos = filterTodos(todos, tab);
```

console.timeEnd('filter array');
```

Perform the interaction you're measuring (for example, typing into the input). You will then see logs like `filter array: 0.15ms` in your console. If the overall logged time adds up to a significant amount (say, `1ms` or more), it might make sense to memoize that calculation. As an experiment, you can then wrap the calculation in `useMemo` to verify whether the total logged time has decreased for that interaction or not:

```js
console.time('filter array');
const visibleTodos = useMemo(() => {
return filterTodos(todos, tab); // Skipped if todos and tab haven't changed
}, [todos, tab]);
console.timeEnd('filter array');
```

`useMemo` won't make the *first* render faster. It only helps you skip unnecessary work on updates.

Keep in mind that your machine is probably faster than your users' so it's a good idea to test the performance with an artificial slowdown. For example, Chrome offers a [CPU Throttling](https://developer.chrome.com/blog/new-in-devtools-61/#throttling) option for this.

Also note that measuring performance in development will not give you the most accurate results. (For example, when [Strict Mode](/reference/react/StrictMode) is on, you will see each component render twice rather than once.) To get the most accurate timings, build your app for production and test it on a device like your users have.

</DeepDive>

<DeepDive>

Should you add useMemo everywhere? {/*should-you-add-usememo-everywhere*/}

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

Optimizing with `useMemo` is only valuable in a few cases:

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

There is no benefit to wrapping a calculation in `useMemo` in other cases. There is no significant harm to doing that either, so some teams choose to not think about individual cases, and memoize as much as possible. The downside of this approach is that code becomes less readable. Also, not all memoization is effective: a single value that's "always new" is enough to break memoization for an entire component.

In practice, you can make a lot of memoization unnecessary by following a few principles:

1. When a component visually wraps other components, let it [accept JSX as children.](/learn/passing-props-to-a-component#passing-jsx-as-children) This way, when the wrapper component updates its own state, React knows that its children don't need to re-render.

1. Prefer local state and don't [lift state up](/learn/sharing-state-between-components) any further than necessary. For example, don't keep transient state like forms and whether an item is hovered at the top of your tree or in a global state library.

1. Keep your [rendering logic pure.](/learn/keeping-components-pure) If re-rendering a component causes a problem or produces some noticeable visual artifact, it's a bug in your component! Fix the bug instead of adding memoization.

1. Avoid [unnecessary Effects that update state.](/learn/you-might-not-need-an-effect) Most performance problems in React apps are caused by chains of updates originating from Effects that cause your components to render over and over.

1. Try to [remove unnecessary dependencies from your Effects.](/learn/removing-effect-dependencies) For example, instead of memoization, it's often simpler to move some object or a function inside an Effect or outside the component.

If a specific interaction still feels laggy, [use the React Developer Tools profiler](https://legacy.reactjs.org/blog/2018/09/10/introducing-the-react-profiler.html) to see which components would benefit the most from memoization, and add memoization where needed. These principles make your components easier to debug and understand, so it's good to follow them in any case. In the long term, we're researching [doing granular memoization automatically](https://www.youtube.com/watch?v=lGEMwh32soc) to solve this once and for all.

</DeepDive>

<Recipes titleText="The difference between useMemo and calculating a value directly" titleId="examples-recalculation">

Skipping recalculation with `useMemo` {/*skipping-recalculation-with-usememo*/}

In this example, the `filterTodos` implementation is **artificially slowed down** so that you can see what happens when some JavaScript function you're calling during rendering is genuinely slow. Try switching the tabs and toggling the theme.

Switching the tabs feels slow because it forces the slowed down `filterTodos` to re-execute. That's expected because the `tab` has changed, and so the entire calculation *needs* to re-run. (If you're curious why it runs twice, it's explained [here.](#my-calculation-runs-twice-on-every-re-render))

Toggle the theme. **Thanks to `useMemo`, it's fast despite the artificial slowdown!** The slow `filterTodos` call was skipped because both `todos` and `tab` (which you pass as dependencies to `useMemo`) haven't changed since the last render.

<Sandpack>

```js App.js
import { useState } from 'react';
import { createTodos } from './utils.js';
import TodoList from './TodoList.js';

const todos = createTodos();

export default function App() {
const [tab, setTab] = useState('all');
const [isDark, setIsDark] = useState(false);
return (
<>
<button onClick={() => setTab('all')}>
All
</button>
<button onClick={() => setTab('active')}>
Active
</button>
<button onClick={() => setTab('completed')}>
Completed
</button>
<br />
<label>
<input
type="checkbox"
checked={isDark}
onChange={e => setIsDark(e.target.checked)}
/>
Dark mode
</label>
<hr />
<TodoList
todos={todos}
tab={tab}
theme={isDark ? 'dark' : 'light'}
/>
</>
```

```
);
}
```

```js TodoList.js active
import { useMemo } from 'react';
import { filterTodos } from './utils.js'

export default function TodoList({ todos, theme, tab }) {
const visibleTodos = useMemo(
() => filterTodos(todos, tab),
[todos, tab]
);
return (
<div className={theme}>
<p><b>Note: <code>filterTodos</code> is artificially slowed down!</b></p>
<ul>
{visibleTodos.map(todo => (
<li key={todo.id}>
{todo.completed ?
<s>{todo.text}</s> :
todo.text
}
</li>
))}
</ul>
</div>
);
}
```

```js utils.js
export function createTodos() {
const todos = [];
for (let i = 0; i < 50; i++) {
todos.push({
id: i,
```

```
text: "Todo " + (i + 1),
completed: Math.random() > 0.5
});
}
return todos;
}

export function filterTodos(todos, tab) {
console.log('[ARTIFICIALLY SLOW] Filtering ' + todos.length + ' todos for "' + tab + '" tab.');
let startTime = performance.now();
while (performance.now() - startTime < 500) {
// Do nothing for 500 ms to emulate extremely slow code
}

return todos.filter(todo => {
if (tab === 'all') {
return true;
} else if (tab === 'active') {
return !todo.completed;
} else if (tab === 'completed') {
return todo.completed;
}
});
}
```

```css
label {
display: block;
margin-top: 10px;
}

.dark {
background-color: black;
color: white;
}

.light {
background-color: white;
```

color: black;

}

```

</Sandpack>

<Solution />

#### Always recalculating a value {/*always-recalculating-a-value*/}

In this example, the `filterTodos` implementation is also **artificially slowed down** so that you can see what happens when some JavaScript function you're calling during rendering is genuinely slow. Try switching the tabs and toggling the theme.

Unlike in the previous example, toggling the theme is also slow now! This is because **there is no `useMemo` call in this version,** so the artificially slowed down `filterTodos` gets called on every re-render. It is called even if only `theme` has changed.

<Sandpack>

```js App.js
import { useState } from 'react';

import { createTodos } from './utils.js';

import TodoList from './TodoList.js';

const todos = createTodos();

export default function App() {

const [tab, setTab] = useState('all');

const [isDark, setIsDark] = useState(false);

return (

<>

<button onClick={() => setTab('all')}>

All

</button>

<button onClick={() => setTab('active')}>

Active

</button>

<button onClick={() => setTab('completed')}>

Completed

</button>

<br />

<label>

```
<input
type="checkbox"
checked={isDark}
onChange={e => setIsDark(e.target.checked)}
/>
Dark mode
</label>
<hr />
<TodoList
todos={todos}
tab={tab}
theme={isDark ? 'dark' : 'light'}
/>
</>
);
}
```

```js TodoList.js active
import { filterTodos } from './utils.js'

export default function TodoList({ todos, theme, tab }) {
const visibleTodos = filterTodos(todos, tab);
return (
<div className={theme}>
<ul>
<p><b>Note: <code>filterTodos</code> is artificially slowed down!</b></p>
{visibleTodos.map(todo => (
<li key={todo.id}>
{todo.completed ?
<s>{todo.text}</s> :
todo.text
}
</li>
))}
</ul>
```

```
</div>
);
}
```

```js utils.js
export function createTodos() {
const todos = [];
for (let i = 0; i < 50; i++) {
todos.push({
id: i,
text: "Todo " + (i + 1),
completed: Math.random() > 0.5
});
}
return todos;
}

export function filterTodos(todos, tab) {
console.log('[ARTIFICIALLY SLOW] Filtering ' + todos.length + ' todos for "' + tab + '" tab.');
let startTime = performance.now();
while (performance.now() - startTime < 500) {
// Do nothing for 500 ms to emulate extremely slow code
}

return todos.filter(todo => {
if (tab === 'all') {
return true;
} else if (tab === 'active') {
return !todo.completed;
} else if (tab === 'completed') {
return todo.completed;
}
});
}
```

```css
```

label {

display: block;

margin-top: 10px;

}

.dark {

background-color: black;

color: white;

}

.light {

background-color: white;

color: black;

}

```

</Sandpack>

However, here is the same code **with the artificial slowdown removed.** Does the lack of `useMemo` feel noticeable or not?

<Sandpack>

```js App.js

import { useState } from 'react';

import { createTodos } from './utils.js';

import TodoList from './TodoList.js';

const todos = createTodos();

export default function App() {

const [tab, setTab] = useState('all');

const [isDark, setIsDark] = useState(false);

return (

<>

<button onClick={() => setTab('all')}>

All

</button>

<button onClick={() => setTab('active')}>

Active

</button>

<button onClick={() => setTab('completed')}>
Completed
</button>
<br />
<label>
<input
type="checkbox"
checked={isDark}
onChange={e => setIsDark(e.target.checked)}
/>
Dark mode
</label>
<hr />
<TodoList
todos={todos}
tab={tab}
theme={isDark ? 'dark' : 'light'}
/>
</>
);
}
```

```js TodoList.js active
import { filterTodos } from './utils.js'

export default function TodoList({ todos, theme, tab }) {
const visibleTodos = filterTodos(todos, tab);
return (

<ul>
{visibleTodos.map(todo => (
<li key={todo.id}>
{todo.completed ?
<s>{todo.text}</s> :
todo.text

```
}
</li>
))}
</ul>
</div>
);
}
```

```js utils.js
export function createTodos() {
const todos = [];
for (let i = 0; i < 50; i++) {
todos.push({
id: i,
text: "Todo " + (i + 1),
completed: Math.random() > 0.5
});
}
return todos;
}

export function filterTodos(todos, tab) {
console.log('Filtering ' + todos.length + ' todos for "' + tab + '" tab.');

return todos.filter(todo => {
if (tab === 'all') {
return true;
} else if (tab === 'active') {
return !todo.completed;
} else if (tab === 'completed') {
return todo.completed;
}
});
}
```

```css
```

label {

display: block;

margin-top: 10px;

}

.dark {

background-color: black;

color: white;

}

.light {

background-color: white;

color: black;

}

```

</Sandpack>

Quite often, code without memoization works fine. If your interactions are fast enough, you might not need memoization.

You can try increasing the number of todo items in `utils.js` and see how the behavior changes. This particular calculation wasn't very expensive to begin with, but if the number of todos grows significantly, most of the overhead will be in re-rendering rather than in the filtering. Keep reading below to see how you can optimize re-rendering with `useMemo`.

<Solution />

</Recipes>

---

### Skipping re-rendering of components {/*skipping-re-rendering-of-components*/}

In some cases, `useMemo` can also help you optimize performance of re-rendering child components. To illustrate this, let's say this `TodoList` component passes the `visibleTodos` as a prop to the child `List` component:

```js {5}

export default function TodoList({ todos, tab, theme }) {

// ...

return (

<List items={visibleTodos} />

);

}
```

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

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

```js {3,5}
import { memo } from 'react';

const List = memo(function List({ items }) {

// ...

});
```

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

```js {2-3,6-7}
export default function TodoList({ todos, tab, theme }) {

// Every time the theme changes, this will be a different array...

const visibleTodos = filterTodos(todos, tab);

return (

<div className={theme}>

{/* ... so List's props will never be the same, and it will re-render every time */}

<List items={visibleTodos} />

</div>

);

}
```

In the above example, the `filterTodos` function always creates a *different* array, similar to how the `{}` object literal always creates a new object. Normally, this wouldn't be a problem, but it means that `List` props will never be the same, and your [`memo`](/reference/react/memo) optimization won't work. This is where `useMemo` comes in handy:

```js {2-3,5,9-10}

```
export default function TodoList({ todos, tab, theme }) {
// Tell React to cache your calculation between re-renders...
const visibleTodos = useMemo(
() => filterTodos(todos, tab),
[todos, tab] // ...so as long as these dependencies don't change...
);
return (
<div className={theme}>
{/* ...List will receive the same props and can skip re-rendering */}
<List items={visibleTodos} />
</div>
);
}
```

By wrapping the `visibleTodos` calculation in `useMemo`, you ensure that it has the *same* value between the re-renders (until dependencies change). You don't *have to* wrap a calculation in `useMemo` unless you do it for some specific reason. In this example, the reason is that you pass it to a component wrapped in [`memo`,](/reference/react/memo) and this lets it skip re-rendering. There are a few other reasons to add `useMemo` which are described further on this page.

<DeepDive>

Memoizing individual JSX nodes {/*memoizing-individual-jsx-nodes*/}

Instead of wrapping `List` in [`memo`](/reference/react/memo), you could wrap the `<List />` JSX node itself in `useMemo`:

```js {3,6}
export default function TodoList({ todos, tab, theme }) {
const visibleTodos = useMemo(() => filterTodos(todos, tab), [todos, tab]);
const children = useMemo(() => <List items={visibleTodos} />, [visibleTodos]);
return (
<div className={theme}>
{children}
</div>
);
}
```

The behavior would be the same. If the `visibleTodos` haven't changed, `List` won't be re-rendered.

A JSX node like `<List items={visibleTodos} />` is an object like `{ type: List, props: { items: visibleTodos } }`. Creating this object is very cheap, but React doesn't know whether its contents is the same as last time or not. This is why by default, React will re-render the `List` component.

However, if React sees the same exact JSX as during the previous render, it won't try to re-render your component. This is because JSX nodes are [immutable.](https://en.wikipedia.org/wiki/Immutable_object) A JSX node object could not have changed over time, so React knows it's safe to skip a re-render. However, for this to work, the node has to *actually be the same object*, not merely look the same in code. This is what `useMemo` does in this example.

Manually wrapping JSX nodes into `useMemo` is not convenient. For example, you can't do this conditionally. This is usually why you would wrap components with [`memo`](/reference/react/memo) instead of wrapping JSX nodes.

</DeepDive>

<Recipes titleText="The difference between skipping re-renders and always re-rendering" titleId="examples-rerendering">

Skipping re-rendering with `useMemo` and `memo`
{/*skipping-re-rendering-with-usememo-and-memo*/}

In this example, the `List` component is **artificially slowed down** so that you can see what happens when a React component you're rendering is genuinely slow. Try switching the tabs and toggling the theme.

Switching the tabs feels slow because it forces the slowed down `List` to re-render. That's expected because the `tab` has changed, and so you need to reflect the user's new choice on the screen.

Next, try toggling the theme. **Thanks to `useMemo` together with [`memo`](/reference/react/memo), it's fast despite the artificial slowdown!** The `List` skipped re-rendering because the `visibleItems` array has not changed since the last render. The `visibleItems` array has not changed because both `todos` and `tab` (which you pass as dependencies to `useMemo`) haven't changed since the last render.

<Sandpack>

```js App.js
import { useState } from 'react';
import { createTodos } from './utils.js';
import TodoList from './TodoList.js';

const todos = createTodos();

export default function App() {
const [tab, setTab] = useState('all');
const [isDark, setIsDark] = useState(false);
return (
```

```
<>
<button onClick={() => setTab('all')}>
All
</button>
<button onClick={() => setTab('active')}>
Active
</button>
<button onClick={() => setTab('completed')}>
Completed
</button>
<br />
<label>
<input
type="checkbox"
checked={isDark}
onChange={e => setIsDark(e.target.checked)}
/>
Dark mode
</label>
<hr />
<TodoList
todos={todos}
tab={tab}
theme={isDark ? 'dark' : 'light'}
/>
</>
);
}
```

```js TodoList.js active
import { useMemo } from 'react';
import List from './List.js';
import { filterTodos } from './utils.js'

export default function TodoList({ todos, theme, tab }) {
const visibleTodos = useMemo(
```

```
() => filterTodos(todos, tab),
[todos, tab]
);
return (
<div className={theme}>
<p><b>Note: <code>List</code> is artificially slowed down!</b></p>
<List items={visibleTodos} />
</div>
);
}
```

```js List.js
import { memo } from 'react';

const List = memo(function List({ items }) {
console.log('[ARTIFICIALLY SLOW] Rendering <List /> with ' + items.length + ' items');
let startTime = performance.now();
while (performance.now() - startTime < 500) {
// Do nothing for 500 ms to emulate extremely slow code
}

return (
<ul>
{items.map(item => (
<li key={item.id}>
{item.completed ?
<s>{item.text}</s> :
item.text
}
</li>
))}
</ul>
);
});

export default List;
```

```js utils.js
export function createTodos() {
const todos = [];
for (let i = 0; i < 50; i++) {
todos.push({
id: i,
text: "Todo " + (i + 1),
completed: Math.random() > 0.5
});
}
return todos;
}

export function filterTodos(todos, tab) {
return todos.filter(todo => {
if (tab === 'all') {
return true;
} else if (tab === 'active') {
return !todo.completed;
} else if (tab === 'completed') {
return todo.completed;
}
});
}
```

```css
label {
display: block;
margin-top: 10px;
}

.dark {
background-color: black;
color: white;
}

.light {
```

background-color: white;

color: black;

}

```

</Sandpack>

<Solution />

#### Always re-rendering a component {/*always-re-rendering-a-component*/}

In this example, the `List` implementation is also **artificially slowed down** so that you can see what happens when some React component you're rendering is genuinely slow. Try switching the tabs and toggling the theme.

Unlike in the previous example, toggling the theme is also slow now! This is because **there is no `useMemo` call in this version,** so the `visibleTodos` is always a different array, and the slowed down `List` component can't skip re-rendering.

<Sandpack>

```js App.js
import { useState } from 'react';

import { createTodos } from './utils.js';

import TodoList from './TodoList.js';

const todos = createTodos();

export default function App() {

const [tab, setTab] = useState('all');

const [isDark, setIsDark] = useState(false);

return (

<>

<button onClick={() => setTab('all')}>

All

</button>

<button onClick={() => setTab('active')}>

Active

</button>

<button onClick={() => setTab('completed')}>

Completed

</button>

<br />

```
<label>
<input
type="checkbox"
checked={isDark}
onChange={e => setIsDark(e.target.checked)}
/>
Dark mode
</label>
<hr />
<TodoList
todos={todos}
tab={tab}
theme={isDark ? 'dark' : 'light'}
/>
</>
);
}
```

```js TodoList.js active
import List from './List.js';
import { filterTodos } from './utils.js'

export default function TodoList({ todos, theme, tab }) {
const visibleTodos = filterTodos(todos, tab);
return (
<div className={theme}>
<p><b>Note: <code>List</code> is artificially slowed down!</b></p>
<List items={visibleTodos} />
</div>
);
}
```

```js List.js
import { memo } from 'react';

const List = memo(function List({ items }) {
```

````
console.log('[ARTIFICIALLY SLOW] Rendering <List /> with ' + items.length + ' items');
let startTime = performance.now();
while (performance.now() - startTime < 500) {
// Do nothing for 500 ms to emulate extremely slow code
}

return (
<ul>
{items.map(item => (
<li key={item.id}>
{item.completed ?
<s>{item.text}</s> :
item.text
}
</li>
))}
</ul>
);
});

export default List;
```

```js utils.js
export function createTodos() {
const todos = [];
for (let i = 0; i < 50; i++) {
todos.push({
id: i,
text: "Todo " + (i + 1),
completed: Math.random() > 0.5
});
}
return todos;
}

export function filterTodos(todos, tab) {
return todos.filter(todo => {
````

````
if (tab === 'all') {
return true;
} else if (tab === 'active') {
return !todo.completed;
} else if (tab === 'completed') {
return todo.completed;
}
});
}
```

```css
label {
display: block;
margin-top: 10px;
}

.dark {
background-color: black;
color: white;
}

.light {
background-color: white;
color: black;
}
```

</Sandpack>

However, here is the same code **with the artificial slowdown removed.** Does the lack of `useMemo` feel noticeable or not?

<Sandpack>

```js App.js
import { useState } from 'react';
import { createTodos } from './utils.js';
import TodoList from './TodoList.js';

const todos = createTodos();
````

```
export default function App() {
const [tab, setTab] = useState('all');
const [isDark, setIsDark] = useState(false);
return (
<>
<button onClick={() => setTab('all')}>
All
</button>
<button onClick={() => setTab('active')}>
Active
</button>
<button onClick={() => setTab('completed')}>
Completed
</button>
<br />
<label>
<input
type="checkbox"
checked={isDark}
onChange={e => setIsDark(e.target.checked)}
/>
Dark mode
</label>
<hr />
<TodoList
todos={todos}
tab={tab}
theme={isDark ? 'dark' : 'light'}
/>
</>
);
}
```

```js TodoList.js active
import List from './List.js';
```

```
import { filterTodos } from './utils.js'

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

```js List.js
import { memo } from 'react';

function List({ items }) {
return (
<ul>
{items.map(item => (
<li key={item.id}>
{item.completed ?
<s>{item.text}</s> :
item.text
}
</li>
))}
</ul>
);
}

export default memo(List);
```

```js utils.js
export function createTodos() {
const todos = [];
for (let i = 0; i < 50; i++) {
todos.push({
id: i,
```

text: "Todo " + (i + 1),
completed: Math.random() > 0.5
});
}
return todos;
}

export function filterTodos(todos, tab) {
return todos.filter(todo => {
if (tab === 'all') {
return true;
} else if (tab === 'active') {
return !todo.completed;
} else if (tab === 'completed') {
return todo.completed;
}
});
}
```

```css
label {
display: block;
margin-top: 10px;
}

.dark {
background-color: black;
color: white;
}

.light {
background-color: white;
color: black;
}
```

</Sandpack>
```

Quite often, code without memoization works fine. If your interactions are fast enough, you don't need memoization.

Keep in mind that you need to run React in production mode, disable [React Developer Tools](/learn/react-developer-tools), and use devices similar to the ones your app's users have in order to get a realistic sense of what's actually slowing down your app.

<Solution />

</Recipes>

Memoizing a dependency of another Hook {/*memoizing-a-dependency-of-another-hook*/}

Suppose you have a calculation that depends on an object created directly in the component body:

```js {2}
function Dropdown({ allItems, text }) {
const searchOptions = { matchMode: 'whole-word', text };

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

Depending on an object like this defeats the point of memoization. When a component re-renders, all of the code directly inside the component body runs again. **The lines of code creating the `searchOptions` object will also run on every re-render.** Since `searchOptions` is a dependency of your `useMemo` call, and it's different every time, React knows the dependencies are different, and recalculate `searchItems` every time.

To fix this, you could memoize the `searchOptions` object *itself* before passing it as a dependency:

```js {2-4}
function Dropdown({ allItems, text }) {
const searchOptions = useMemo(() => {
return { matchMode: 'whole-word', text };
}, [text]); // ■ Only changes when text changes

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

In the example above, if the `text` did not change, the `searchOptions` object also won't change. However, an even better fix is to move the `searchOptions` object declaration *inside* of the `useMemo` calculation function:

```js {3}
function Dropdown({ allItems, text }) {
const visibleItems = useMemo(() => {
const searchOptions = { matchMode: 'whole-word', text };
return searchItems(allItems, searchOptions);
}, [allItems, text]); // ■ Only changes when allItems or text changes
// ...
```

Now your calculation depends on `text` directly (which is a string and can't "accidentally" become different).

Memoizing a function {/*memoizing-a-function*/}

Suppose the `Form` component is wrapped in [`memo`.](/reference/react/memo) You want to pass a function to it as a prop:

```js {2-7}
export default function ProductPage({ productId, referrer }) {
function handleSubmit(orderDetails) {
post('/product/' + productId + '/buy', {
referrer,
orderDetails
});
}
return <Form onSubmit={handleSubmit} />;
}
```

Just as `{}` creates a different object, function declarations like `function() {}` and expressions like `() => {}` produce a *different* function on every re-render. By itself, creating a new function is not a problem. This is not something to avoid! However, if the `Form` component is memoized, presumably you want to skip re-rendering it when no props have changed. A prop that is *always* different would defeat the point of memoization.

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

```js {2-3,8-9}
export default function Page({ productId, referrer }) {
const handleSubmit = useMemo(() => {
return (orderDetails) => {
post('/product/' + productId + '/buy', {
referrer,
orderDetails
});
};
}, [productId, referrer]);

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

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

```js {2,7}
export default function Page({ productId, referrer }) {
const handleSubmit = useCallback((orderDetails) => {
post('/product/' + productId + '/buy', {
referrer,
orderDetails
});
}, [productId, referrer]);

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

The two examples above are completely equivalent. The only benefit to `useCallback` is that it lets you avoid writing an extra nested function inside. It doesn't do anything else. [Read more about `useCallback`.](/reference/react/useCallback)

Troubleshooting {/*troubleshooting*/}

My calculation runs twice on every re-render {/*my-calculation-runs-twice-on-every-re-render*/}

In [Strict Mode](/reference/react/StrictMode), React will call some of your functions twice instead of once:

```js {2,5,6}
function TodoList({ todos, tab }) {
// This component function will run twice for every render.

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

// ...
```

This is expected and shouldn't break your code.

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

For example, this impure calculation function mutates an array you received as a prop:

```js {2-3}
const visibleTodos = useMemo(() => {
// ■ Mistake: mutating a prop
todos.push({ id: 'last', text: 'Go for a walk!' });
const filtered = filterTodos(todos, tab);
return filtered;
}, [todos, tab]);
```

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

```js {3,4}
const visibleTodos = useMemo(() => {
const filtered = filterTodos(todos, tab);
// ■ Correct: mutating an object you created during the calculation
filtered.push({ id: 'last', text: 'Go for a walk!' });
return filtered;
```

}, [todos, tab]);
```

Read [keeping components pure](/learn/keeping-components-pure) to learn more about purity.

Also, check out the guides on [updating objects](/learn/updating-objects-in-state) and [updating arrays](/learn/updating-arrays-in-state) without mutation.

---

### My `useMemo` call is supposed to return an object, but returns undefined {/*my-usememo-call-is-supposed-to-return-an-object-but-returns-undefined*/}

This code doesn't work:

```js {1-2,5}
// ■ You can't return an object from an arrow function with () => {
const searchOptions = useMemo(() => {
matchMode: 'whole-word',
text: text
}, [text]);
```

In JavaScript, `() => {` starts the arrow function body, so the `{` brace is not a part of your object. This is why it doesn't return an object, and leads to mistakes. You could fix it by adding parentheses like `({` and `})`:

```js {1-2,5}
// This works, but is easy for someone to break again
const searchOptions = useMemo(() => ({
matchMode: 'whole-word',
text: text
}), [text]);
```

However, this is still confusing and too easy for someone to break by removing the parentheses.

To avoid this mistake, write a `return` statement explicitly:

```js {1-3,6-7}
// ■ This works and is explicit
const searchOptions = useMemo(() => {
return {
matchMode: 'whole-word',
```
```

text: text

};

}, [text]);

```

---

### Every time my component renders, the calculation in `useMemo` re-runs
{/*every-time-my-component-renders-the-calculation-in-usememo-re-runs*/}

Make sure you've specified the dependency array as a second argument!

If you forget the dependency array, `useMemo` will re-run the calculation every time:

```js {2-3}
function TodoList({ todos, tab }) {
// ■ Recalculates every time: no dependency array
const visibleTodos = useMemo(() => filterTodos(todos, tab));
// ...
```

This is the corrected version passing the dependency array as a second argument:

```js {2-3}
function TodoList({ todos, tab }) {
// ■ Does not recalculate unnecessarily
const visibleTodos = useMemo(() => filterTodos(todos, tab), [todos, tab]);
// ...
```

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

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

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

```js

Object.is(temp1[0], temp2[0]); // Is the first dependency the same between the arrays?

Object.is(temp1[1], temp2[1]); // Is the second dependency the same between the arrays?

Object.is(temp1[2], temp2[2]); // ... and so on for every dependency ...

```

When you find which dependency breaks memoization, either find a way to remove it, or [memoize it as well.](#memoizing-a-dependency-of-another-hook)

I need to call `useMemo` for each list item in a loop, but it's not allowed {/*i-need-to-call-usememo-for-each-list-item-in-a-loop-but-its-not-allowed*/}

Suppose the `Chart` component is wrapped in [`memo`](/reference/react/memo). You want to skip re-rendering every `Chart` in the list when the `ReportList` component re-renders. However, you can't call `useMemo` in a loop:

```js {5-11}

function ReportList({ items }) {

return (

<article>

{items.map(item => {

// ■ You can't call useMemo in a loop like this:

const data = useMemo(() => calculateReport(item), [item]);

return (

<figure key={item.id}>

<Chart data={data} />

</figure>

);

})}

</article>

);

}

```

Instead, extract a component for each item and memoize data for individual items:

```js {5,12-18}

function ReportList({ items }) {

return (

<article>

{items.map(item =>

<Report key={item.id} item={item} />

)}

</article>

);

}

function Report({ item }) {

// ■ Call useMemo at the top level:

const data = useMemo(() => calculateReport(item), [item]);

return (

<figure>

<Chart data={data} />

</figure>

);

}

```

Alternatively, you could remove `useMemo` and instead wrap `Report` itself in [`memo`.](/reference/react/memo) If the `item` prop does not change, `Report` will skip re-rendering, so `Chart` will skip re-rendering too:

```js {5,6,12}

function ReportList({ items }) {

// ...

}

const Report = memo(function Report({ item }) {

const data = calculateReport(item);

return (

<figure>

<Chart data={data} />

</figure>

);

});

```

---

title: lazy

```

<Intro>

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

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

</Intro>

<InlineToc />

Reference {/*reference*/}

`lazy(load)` {/*lazy*/}

Call `lazy` outside your components to declare a lazy-loaded React component:

```js
import { lazy } from 'react';

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

[See more examples below.](#usage)

Parameters {/*parameters*/}

* `load`: A function that returns a [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) or another *thenable* (a Promise-like object with a `then` method). React will not call `load` until the first time you attempt to render the returned component. After React first calls `load`, it will wait for it to resolve, and then render the resolved value as a React component. Both the returned Promise and the Promise's resolved value will be cached, so React will not call `load` more than once. If the Promise rejects, React will `throw` the rejection reason for the nearest Error Boundary to handle.

Returns {/*returns*/}

`lazy` returns a React component you can render in your tree. While the code for the lazy component is still loading, attempting to render it will *suspend.* Use [`<Suspense>`](/reference/react/Suspense) to display a loading indicator while it's loading.

`load` function {/*load*/}

Parameters {/*load-parameters*/}

`load` receives no parameters.

Returns {/*load-returns*/}

You need to return a [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) or some other *thenable* (a Promise-like object with a `then` method). It needs to eventually resolve to a valid React component type, such as a function, [`memo`](/reference/react/memo), or a [`forwardRef`](/reference/react/forwardRef) component.

Usage {/*usage*/}

Lazy-loading components with Suspense {/*suspense-for-code-splitting*/}

Usually, you import components with the static [`import`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import) declaration:

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

To defer loading this component's code until it's rendered for the first time, replace this import with:

```js
import { lazy } from 'react';

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

This code relies on [dynamic `import()`,](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/import) which might require support from your bundler or framework.

Now that your component's code loads on demand, you also need to specify what should be displayed while it is loading. You can do this by wrapping the lazy component or any of its parents into a [`<Suspense>`](/reference/react/Suspense) boundary:

```js {1,4}
<Suspense fallback={<Loading />}>
<h2>Preview</h2>
<MarkdownPreview />
</Suspense>
```

In this example, the code for `MarkdownPreview` won't be loaded until you attempt to render it. If `MarkdownPreview` hasn't loaded yet, `Loading` will be shown in its place. Try ticking the checkbox:

<Sandpack>

```js App.js
import { useState, Suspense, lazy } from 'react';
import Loading from './Loading.js';

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

export default function MarkdownEditor() {
const [showPreview, setShowPreview] = useState(false);
const [markdown, setMarkdown] = useState('Hello, **world**!');
return (
<>
<textarea value={markdown} onChange={e => setMarkdown(e.target.value)} />
<label>
<input type="checkbox" checked={showPreview} onChange={e => setShowPreview(e.target.checked)}
/>
Show preview
</label>
<hr />
{showPreview && (
<Suspense fallback={<Loading />}>
<h2>Preview</h2>
<MarkdownPreview markdown={markdown} />
</Suspense>
)}
</>
);
}

// Add a fixed delay so you can see the loading state
function delayForDemo(promise) {
return new Promise(resolve => {
setTimeout(resolve, 2000);
}).then(() => promise);
}
```

```js Loading.js
export default function Loading() {
```

return Loading... ;
}
```

```js MarkdownPreview.js
import { Remarkable } from 'remarkable';

const md = new Remarkable();

export default function MarkdownPreview({ markdown }) {
return (
<div
className="content"
dangerouslySetInnerHTML={{__html: md.render(markdown)}}
/>
);
}
```

```json package.json hidden
{
"dependencies": {
"immer": "1.7.3",
"react": "latest",
"react-dom": "latest",
"react-scripts": "latest",
"remarkable": "2.0.1"
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
}
}
```

```css
label {

```
display: block;
}

input, textarea {
margin-bottom: 10px;
}

body {
min-height: 200px;
}
```

</Sandpack>

This demo loads with an artificial delay. The next time you untick and tick the checkbox, `Preview` will be cached, so there will be no loading state. To see the loading state again, click "Reset" on the sandbox.

[Learn more about managing loading states with Suspense.](/reference/react/Suspense)

Troubleshooting {/*troubleshooting*/}

My `lazy` component's state gets reset unexpectedly {/*my-lazy-components-state-gets-reset-unexpectedly*/}

Do not declare `lazy` components *inside* other components:

```js {4-5}
import { lazy } from 'react';

function Editor() {
// ■ Bad: This will cause all state to be reset on re-renders
const MarkdownPreview = lazy(() => import('./MarkdownPreview.js'));
// ...
}
```

Instead, always declare them at the top level of your module:

```js {3-4}
import { lazy } from 'react';

// ■ Good: Declare lazy components outside of your components
const MarkdownPreview = lazy(() => import('./MarkdownPreview.js'));
```

function Editor() {
// ...
}
```

---
title: useId
---

<Intro>

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

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

</Intro>

<InlineToc />

Reference {/*reference*/}

`useId()` {/*useid*/}

Call `useId` at the top level of your component to generate a unique ID:

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

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

[See more examples below.](#usage)

Parameters {/*parameters*/}

`useId` does not take any parameters.

Returns {/*returns*/}

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

Caveats {/*caveats*/}

* `useId` is a Hook, so you can only call it **at the top level of your component** or your own Hooks. You can't call it inside loops or conditions. If you need that, extract a new component and move the state into it.

* `useId` **should not be used to generate keys** in a list. [Keys should be generated from your data.](/learn/rendering-lists#where-to-get-your-key)

Usage {/*usage*/}

<Pitfall>

Do not call `useId` to generate keys in a list. [Keys should be generated from your data.](/learn/rendering-lists#where-to-get-your-key)

</Pitfall>

Generating unique IDs for accessibility attributes
{/*generating-unique-ids-for-accessibility-attributes*/}

Call `useId` at the top level of your component to generate a unique ID:

```js [[1, 4, "passwordHintId"]]
import { useId } from 'react';

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

You can then pass the <CodeStep step={1}>generated ID</CodeStep> to different attributes:

```js [[1, 2, "passwordHintId"], [1, 3, "passwordHintId"]]
<>
<input type="password" aria-describedby={passwordHintId} />
<p id={passwordHintId}>
</>
```

Let's walk through an example to see when this is useful.

[HTML accessibility attributes](https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA) like [`aria-describedby`](https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/Attributes/aria-describedby) let you specify that two tags are related to each other. For example, you can specify that an element (like an input) is described by another element (like a paragraph).

In regular HTML, you would write it like this:

```html {5,8}
<label>
Password:
<input
type="password"
aria-describedby="password-hint"
/>
</label>
<p id="password-hint">
The password should contain at least 18 characters
</p>
```

However, hardcoding IDs like this is not a good practice in React. A component may be rendered more than once on the page--but IDs have to be unique! Instead of hardcoding an ID, generate a unique ID with `useId`:

```js {4,11,14}
import { useId } from 'react';

function PasswordField() {
const passwordHintId = useId();
return (
<>
<label>
Password:
<input
type="password"
aria-describedby={passwordHintId}
/>
</label>
<p id={passwordHintId}>
The password should contain at least 18 characters
</p>
</>
);
}
```

```

Now, even if `PasswordField` appears multiple times on the screen, the generated IDs won't clash.

<Sandpack>

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

function PasswordField() {
const passwordHintId = useId();
return (
<>
<label>
Password:
<input
type="password"
aria-describedby={passwordHintId}
/>
</label>
<p id={passwordHintId}>
The password should contain at least 18 characters
</p>
</>
);
}

export default function App() {
return (
<>
<h2>Choose password</h2>
<PasswordField />
<h2>Confirm password</h2>
<PasswordField />
</>
);
}
```
```

```css
input { margin: 5px; }
```

</Sandpack>

[Watch this video](https://www.youtube.com/watch?v=0dNzNcuEuOo) to see the difference in the user experience with assistive technologies.

<Pitfall>

With [server rendering](/reference/react-dom/server), **`useId` requires an identical component tree on the server and the client**. If the trees you render on the server and the client don't match exactly, the generated IDs won't match.

</Pitfall>

<DeepDive>

Why is useId better than an incrementing counter?
{/*why-is-useid-better-than-an-incrementing-counter*/}

You might be wondering why `useId` is better than incrementing a global variable like `nextId++`.

The primary benefit of `useId` is that React ensures that it works with [server rendering.](/reference/react-dom/server) During server rendering, your components generate HTML output. Later, on the client, [hydration](/reference/react-dom/client/hydrateRoot) attaches your event handlers to the generated HTML. For hydration to work, the client output must match the server HTML.

This is very difficult to guarantee with an incrementing counter because the order in which the client components are hydrated may not match the order in which the server HTML was emitted. By calling `useId`, you ensure that hydration will work, and the output will match between the server and the client.

Inside React, `useId` is generated from the "parent path" of the calling component. This is why, if the client and the server tree are the same, the "parent path" will match up regardless of rendering order.

</DeepDive>

Generating IDs for several related elements {/*generating-ids-for-several-related-elements*/}

If you need to give IDs to multiple related elements, you can call `useId` to generate a shared prefix for them:

<Sandpack>

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

export default function Form() {

const id = useId();

return (

<form>

<label htmlFor={id + '-firstName'}>First Name:</label>

<input id={id + '-firstName'} type="text" />

<hr />

<label htmlFor={id + '-lastName'}>Last Name:</label>

<input id={id + '-lastName'} type="text" />

</form>

);

}

```

```css

input { margin: 5px; }

```

</Sandpack>

This lets you avoid calling `useId` for every single element that needs a unique ID.

Specifying a shared prefix for all generated IDs
{/*specifying-a-shared-prefix-for-all-generated-ids*/}

If you render multiple independent React applications on a single page, pass `identifierPrefix` as an option to your [`createRoot`](/reference/react-dom/client/createRoot#parameters) or [`hydrateRoot`](/reference/react-dom/client/hydrateRoot) calls. This ensures that the IDs generated by the two different apps never clash because every identifier generated with `useId` will start with the distinct prefix you've specified.

<Sandpack>

```html index.html

<!DOCTYPE html>

<html>

<head><title>My app</title></head>

<body>

```
</body>
</html>
```

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

function PasswordField() {
const passwordHintId = useId();
console.log('Generated identifier:', passwordHintId)
return (
<>
<label>
Password:
<input
type="password"
aria-describedby={passwordHintId}
/>
</label>
<p id={passwordHintId}>
The password should contain at least 18 characters
</p>
</>
);
}

export default function App() {
return (
<>
<h2>Choose password</h2>
<PasswordField />
</>
);
}
```

```js index.js active
import { createRoot } from 'react-dom/client';
```

```
import App from './App.js';
import './styles.css';

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

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

```css
#root1 {
border: 5px solid blue;
padding: 10px;
margin: 5px;
}

#root2 {
border: 5px solid green;
padding: 10px;
margin: 5px;
}

input { margin: 5px; }
```

</Sandpack>

---
title: experimental_useEffectEvent

<Wip>

This API is experimental and is not available in a stable version of React yet.

You can try it by upgrading React packages to the most recent experimental version:

- `react@experimental`

- `react-dom@experimental`
- `eslint-plugin-react-hooks@experimental`

Experimental versions of React may contain bugs. Don't use them in production.

</Wip>

<Intro>

`useEffectEvent` is a React Hook that lets you extract non-reactive logic into an [Effect Event.](/learn/separating-events-from-effects#declaring-an-effect-event)

```js
const onSomething = useEffectEvent(callback)
```

</Intro>

<InlineToc />

---

title: useDeferredValue

<Intro>

`useDeferredValue` is a React Hook that lets you defer updating a part of the UI.

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

</Intro>

<InlineToc />

Reference {/*reference*/}

`useDeferredValue(value)` {/*usedeferredvalue*/}

Call `useDeferredValue` at the top level of your component to get a deferred version of that value.

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

function SearchPage() {
```

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

[See more examples below.](#usage)

Parameters {/*parameters*/}

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

Returns {/*returns*/}

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

Caveats {/*caveats*/}

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

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

- `useDeferredValue` is integrated with [`<Suspense>`.](/reference/react/Suspense) If the background update caused by a new value suspends the UI, the user will not see the fallback. They will see the old deferred value until the data loads.

- `useDeferredValue` does not by itself prevent extra network requests.

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

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

Usage {/*usage*/}

Showing stale content while fresh content is loading
{/*showing-stale-content-while-fresh-content-is-loading*/}

Call `useDeferredValue` at the top level of your component to defer updating some part of your UI.

```js [[1, 5, "query"], [2, 5, "deferredQuery"]]
import { useState, useDeferredValue } from 'react';

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

During the initial render, the <CodeStep step={2}>deferred value</CodeStep> will be the same as the <CodeStep step={1}>value</CodeStep> you provided.

During updates, the <CodeStep step={2}>deferred value</CodeStep> will "lag behind" the latest <CodeStep step={1}>value</CodeStep>. In particular, React will first re-render *without* updating the deferred value, and then try to re-render with the newly received value in background.

Let's walk through an example to see when this is useful.

<Note>

This example assumes you use one of Suspense-enabled data sources:

- Data fetching with Suspense-enabled frameworks like [Relay](https://relay.dev/docs/guided-tour/rendering/loading-states/) and [Next.js](https://nextjs.org/docs/getting-started/react-essentials)
- Lazy-loading component code with [`lazy`](/reference/react/lazy)

[Learn more about Suspense and its limitations.](/reference/react/Suspense)

</Note>

In this example, the `SearchResults` component [suspends](/reference/react/Suspense#displaying-a-fallback-while-content-is-loading) while fetching the search results. Try typing `"a"`, waiting for the results, and then editing it to `"ab"`. The results for `"a"` get replaced by the loading fallback.

<Sandpack>

```json package.json hidden
{
"dependencies": {
```

```
"react": "experimental",
"react-dom": "experimental"
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
}
}
```

```js App.js
import { Suspense, useState } from 'react';
import SearchResults from './SearchResults.js';

export default function App() {
const [query, setQuery] = useState('');
return (
<>
<label>
Search albums:
<input value={query} onChange={e => setQuery(e.target.value)} />
</label>
<Suspense fallback={<h2>Loading...</h2>}>
<SearchResults query={query} />
</Suspense>
</>
);
}
```

```js SearchResults.js hidden
import { fetchData } from './data.js';

// Note: this component is written using an experimental API
// that's not yet available in stable versions of React.

// For a realistic example you can follow today, try a framework
```

```
// that's integrated with Suspense, like Relay or Next.js.

export default function SearchResults({ query }) {
if (query === '') {
return null;
}
const albums = use(fetchData(`/search?q=${query}`));
if (albums.length === 0) {
return <p>No matches for <i>"{query}"</i></p>;
}
return (
<ul>
{albums.map(album => (
<li key={album.id}>
{album.title} ({album.year})
</li>
))}
</ul>
);
}

// This is a workaround for a bug to get the demo running.
// TODO: replace with real implementation when the bug is fixed.
function use(promise) {
if (promise.status === 'fulfilled') {
return promise.value;
} else if (promise.status === 'rejected') {
throw promise.reason;
} else if (promise.status === 'pending') {
throw promise;
} else {
promise.status = 'pending';
promise.then(
result => {
promise.status = 'fulfilled';
promise.value = result;
},
```

```
reason => {
promise.status = 'rejected';
promise.reason = reason;
},
);
throw promise;
}
}
```

```js data.js hidden
// Note: the way you would do data fetching depends on
// the framework that you use together with Suspense.
// Normally, the caching logic would be inside a framework.

let cache = new Map();

export function fetchData(url) {
if (!cache.has(url)) {
cache.set(url, getData(url));
}
return cache.get(url);
}

async function getData(url) {
if (url.startsWith('/search?q=')) {
return await getSearchResults(url.slice('/search?q='.length));
} else {
throw Error('Not implemented');
}
}

async function getSearchResults(query) {
// Add a fake delay to make waiting noticeable.
await new Promise(resolve => {
setTimeout(resolve, 500);
});

const allAlbums = [{
```

id: 13,
title: 'Let It Be',
year: 1970
}, {
id: 12,
title: 'Abbey Road',
year: 1969
}, {
id: 11,
title: 'Yellow Submarine',
year: 1969
}, {
id: 10,
title: 'The Beatles',
year: 1968
}, {
id: 9,
title: 'Magical Mystery Tour',
year: 1967
}, {
id: 8,
title: 'Sgt. Pepper\'s Lonely Hearts Club Band',
year: 1967
}, {
id: 7,
title: 'Revolver',
year: 1966
}, {
id: 6,
title: 'Rubber Soul',
year: 1965
}, {
id: 5,
title: 'Help!',
year: 1965

```
}, {
id: 4,
title: 'Beatles For Sale',
year: 1964
}, {
id: 3,
title: 'A Hard Day\'s Night',
year: 1964
}, {
id: 2,
title: 'With The Beatles',
year: 1963
}, {
id: 1,
title: 'Please Please Me',
year: 1963
}];

const lowerQuery = query.trim().toLowerCase();
return allAlbums.filter(album => {
const lowerTitle = album.title.toLowerCase();
return (
lowerTitle.startsWith(lowerQuery) ||
lowerTitle.indexOf(' ' + lowerQuery) !== -1
)
});
}
```

```css
input { margin: 10px; }
```

</Sandpack>

A common alternative UI pattern is to *defer* updating the list of results and to keep showing the previous results until the new results are ready. Call `useDeferredValue` to pass a deferred version of the query down:

```js {3,11}
export default function App() {
const [query, setQuery] = useState('');
const deferredQuery = useDeferredValue(query);
return (
<>
<label>
Search albums:
<input value={query} onChange={e => setQuery(e.target.value)} />
</label>
<Suspense fallback={<h2>Loading...</h2>}>
<SearchResults query={deferredQuery} />
</Suspense>
</>
);
}
```

The `query` will update immediately, so the input will display the new value. However, the `deferredQuery` will keep its previous value until the data has loaded, so `SearchResults` will show the stale results for a bit.

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

<Sandpack>

```json package.json hidden
{
"dependencies": {
"react": "experimental",
"react-dom": "experimental"
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
```

```
}
}
```

```js App.js
import { Suspense, useState, useDeferredValue } from 'react';
import SearchResults from './SearchResults.js';

export default function App() {
const [query, setQuery] = useState('');
const deferredQuery = useDeferredValue(query);
return (
<>
<label>
Search albums:
<input value={query} onChange={e => setQuery(e.target.value)} />
</label>
<Suspense fallback={<h2>Loading...</h2>}>
<SearchResults query={deferredQuery} />
</Suspense>
</>
);
}
```

```js SearchResults.js hidden
import { fetchData } from './data.js';

// Note: this component is written using an experimental API
// that's not yet available in stable versions of React.

// For a realistic example you can follow today, try a framework
// that's integrated with Suspense, like Relay or Next.js.

export default function SearchResults({ query }) {
if (query === '') {
return null;
}
const albums = use(fetchData(`/search?q=${query}`));
```

```
if (albums.length === 0) {
return <p>No matches for <i>"{query}"</i></p>;
}
return (
<ul>
{albums.map(album => (
<li key={album.id}>
{album.title} ({album.year})
</li>
))}
</ul>
);
}

// This is a workaround for a bug to get the demo running.
// TODO: replace with real implementation when the bug is fixed.
function use(promise) {
if (promise.status === 'fulfilled') {
return promise.value;
} else if (promise.status === 'rejected') {
throw promise.reason;
} else if (promise.status === 'pending') {
throw promise;
} else {
promise.status = 'pending';
promise.then(
result => {
promise.status = 'fulfilled';
promise.value = result;
},
reason => {
promise.status = 'rejected';
promise.reason = reason;
},
);
throw promise;
```

```
}
}
```

```js data.js hidden
// Note: the way you would do data fetching depends on
// the framework that you use together with Suspense.
// Normally, the caching logic would be inside a framework.

let cache = new Map();

export function fetchData(url) {
if (!cache.has(url)) {
cache.set(url, getData(url));
}
return cache.get(url);
}

async function getData(url) {
if (url.startsWith('/search?q=')) {
return await getSearchResults(url.slice('/search?q='.length));
} else {
throw Error('Not implemented');
}
}

async function getSearchResults(query) {
// Add a fake delay to make waiting noticeable.
await new Promise(resolve => {
setTimeout(resolve, 500);
});

const allAlbums = [{
id: 13,
title: 'Let It Be',
year: 1970
}, {
id: 12,
title: 'Abbey Road',
```

year: 1969
}, {
id: 11,
title: 'Yellow Submarine',
year: 1969
}, {
id: 10,
title: 'The Beatles',
year: 1968
}, {
id: 9,
title: 'Magical Mystery Tour',
year: 1967
}, {
id: 8,
title: 'Sgt. Pepper\'s Lonely Hearts Club Band',
year: 1967
}, {
id: 7,
title: 'Revolver',
year: 1966
}, {
id: 6,
title: 'Rubber Soul',
year: 1965
}, {
id: 5,
title: 'Help!',
year: 1965
}, {
id: 4,
title: 'Beatles For Sale',
year: 1964
}, {
id: 3,

title: 'A Hard Day\'s Night',

year: 1964

}, {

id: 2,

title: 'With The Beatles',

year: 1963

}, {

id: 1,

title: 'Please Please Me',

year: 1963

}];

const lowerQuery = query.trim().toLowerCase();

return allAlbums.filter(album => {

const lowerTitle = album.title.toLowerCase();

return (

lowerTitle.startsWith(lowerQuery) ||

lowerTitle.indexOf(' ' + lowerQuery) !== -1

)

});

}

```

```css

input { margin: 10px; }

```

</Sandpack>

<DeepDive>

How does deferring a value work under the hood?
{/*how-does-deferring-a-value-work-under-the-hood*/}

You can think of it as happening in two steps:

1. **First, React re-renders with the new `query` (`"ab"`) but with the old `deferredQuery` (still `"a"`).** The `deferredQuery` value, which you pass to the result list, is *deferred:* it "lags behind" the `query` value.

2. **In background, React tries to re-render with *both* `query` and `deferredQuery` updated to `"ab"`.** If this re-render completes, React will show it on the screen. However, if it suspends (the results for
```

`"ab"` have not loaded yet), React will abandon this rendering attempt, and retry this re-render again after the data has loaded. The user will keep seeing the stale deferred value until the data is ready.

The deferred "background" rendering is interruptible. For example, if you type into the input again, React will abandon it and restart with the new value. React will always use the latest provided value.

Note that there is still a network request per each keystroke. What's being deferred here is displaying results (until they're ready), not the network requests themselves. Even if the user continues typing, responses for each keystroke get cached, so pressing Backspace is instant and doesn't fetch again.

</DeepDive>

Indicating that the content is stale {/*indicating-that-the-content-is-stale*/}

In the example above, there is no indication that the result list for the latest query is still loading. This can be confusing to the user if the new results take a while to load. To make it more obvious to the user that the result list does not match the latest query, you can add a visual indication when the stale result list is displayed:

```js {2}
<div style={{
opacity: query !== deferredQuery ? 0.5 : 1,
}}>
<SearchResults query={deferredQuery} />
</div>
```

With this change, as soon as you start typing, the stale result list gets slightly dimmed until the new result list loads. You can also add a CSS transition to delay dimming so that it feels gradual, like in the example below:

<Sandpack>

```json package.json hidden
{
"dependencies": {
"react": "experimental",
"react-dom": "experimental"
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
```

"eject": "react-scripts eject"
}
}
```

```js App.js
import { Suspense, useState, useDeferredValue } from 'react';
import SearchResults from './SearchResults.js';

export default function App() {
const [query, setQuery] = useState('');
const deferredQuery = useDeferredValue(query);
const isStale = query !== deferredQuery;
return (
<>
<label>
Search albums:
<input value={query} onChange={e => setQuery(e.target.value)} />
</label>
<Suspense fallback={<h2>Loading...</h2>}>
<div style={{
opacity: isStale ? 0.5 : 1,
transition: isStale ? 'opacity 0.2s 0.2s linear' : 'opacity 0s 0s linear'
}}>
<SearchResults query={deferredQuery} />
</div>
</Suspense>
</>
);
}
```

```js SearchResults.js hidden
import { fetchData } from './data.js';

// Note: this component is written using an experimental API
// that's not yet available in stable versions of React.

// For a realistic example you can follow today, try a framework

```
// that's integrated with Suspense, like Relay or Next.js.

export default function SearchResults({ query }) {
if (query === '') {
return null;
}
const albums = use(fetchData(`/search?q=${query}`));
if (albums.length === 0) {
return <p>No matches for <i>"{query}"</i></p>;
}
return (
<ul>
{albums.map(album => (
<li key={album.id}>
{album.title} ({album.year})
</li>
))}
</ul>
);
}

// This is a workaround for a bug to get the demo running.
// TODO: replace with real implementation when the bug is fixed.
function use(promise) {
if (promise.status === 'fulfilled') {
return promise.value;
} else if (promise.status === 'rejected') {
throw promise.reason;
} else if (promise.status === 'pending') {
throw promise;
} else {
promise.status = 'pending';
promise.then(
result => {
promise.status = 'fulfilled';
promise.value = result;
},
```

```
reason => {
promise.status = 'rejected';
promise.reason = reason;
},
);
throw promise;
}
}
```

```js data.js hidden
// Note: the way you would do data fetching depends on
// the framework that you use together with Suspense.
// Normally, the caching logic would be inside a framework.

let cache = new Map();

export function fetchData(url) {
if (!cache.has(url)) {
cache.set(url, getData(url));
}
return cache.get(url);
}

async function getData(url) {
if (url.startsWith('/search?q=')) {
return await getSearchResults(url.slice('/search?q='.length));
} else {
throw Error('Not implemented');
}
}

async function getSearchResults(query) {
// Add a fake delay to make waiting noticeable.
await new Promise(resolve => {
setTimeout(resolve, 500);
});

const allAlbums = [{
```

id: 13,
title: 'Let It Be',
year: 1970
}, {
id: 12,
title: 'Abbey Road',
year: 1969
}, {
id: 11,
title: 'Yellow Submarine',
year: 1969
}, {
id: 10,
title: 'The Beatles',
year: 1968
}, {
id: 9,
title: 'Magical Mystery Tour',
year: 1967
}, {
id: 8,
title: 'Sgt. Pepper\'s Lonely Hearts Club Band',
year: 1967
}, {
id: 7,
title: 'Revolver',
year: 1966
}, {
id: 6,
title: 'Rubber Soul',
year: 1965
}, {
id: 5,
title: 'Help!',
year: 1965

```
}, {
id: 4,
title: 'Beatles For Sale',
year: 1964
}, {
id: 3,
title: 'A Hard Day\'s Night',
year: 1964
}, {
id: 2,
title: 'With The Beatles',
year: 1963
}, {
id: 1,
title: 'Please Please Me',
year: 1963
}];

const lowerQuery = query.trim().toLowerCase();
return allAlbums.filter(album => {
const lowerTitle = album.title.toLowerCase();
return (
lowerTitle.startsWith(lowerQuery) ||
lowerTitle.indexOf(' ' + lowerQuery) !== -1
)
});
}
```

```css
input { margin: 10px; }
```

</Sandpack>

Deferring re-rendering for a part of the UI {/*deferring-re-rendering-for-a-part-of-the-ui*/}

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

Imagine you have a text field and a component (like a chart or a long list) that re-renders on every keystroke:

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

First, optimize `SlowList` to skip re-rendering when its props are the same. To do this, [wrap it in `memo`:](/reference/react/memo#skipping-re-rendering-when-props-are-unchanged)

```js {1,3}
const SlowList = memo(function SlowList({ text }) {
// ...
});
```

However, this only helps if the `SlowList` props are *the same* as during the previous render. The problem you're facing now is that it's slow when they're *different,* and when you actually need to show different visual output.

Concretely, the main performance problem is that whenever you type into the input, the `SlowList` receives new props, and re-rendering its entire tree makes the typing feel janky. In this case, `useDeferredValue` lets you prioritize updating the input (which must be fast) over updating the result list (which is allowed to be slower):

```js {3,7}
function App() {
const [text, setText] = useState('');
const deferredText = useDeferredValue(text);
return (
<>
```

<input value={text} onChange={e => setText(e.target.value)} />

<SlowList text={deferredText} />

</>

);

}

```

This does not make re-rendering of the `SlowList` faster. However, it tells React that re-rendering the list can be deprioritized so that it doesn't block the keystrokes. The list will "lag behind" the input and then "catch up". Like before, React will attempt to update the list as soon as possible, but will not block the user from typing.

<Recipes titleText="The difference between useDeferredValue and unoptimized re-rendering" titleId="examples">

#### Deferred re-rendering of the list {/*deferred-re-rendering-of-the-list*/}

In this example, each item in the `SlowList` component is **artificially slowed down** so that you can see how `useDeferredValue` lets you keep the input responsive. Type into the input and notice that typing feels snappy while the list "lags behind" it.

<Sandpack>

```js

import { useState, useDeferredValue } from 'react';

import SlowList from './SlowList.js';

export default function App() {

const [text, setText] = useState('');

const deferredText = useDeferredValue(text);

return (

<>

<input value={text} onChange={e => setText(e.target.value)} />

<SlowList text={deferredText} />

</>

);

}

```

```js SlowList.js

import { memo } from 'react';

const SlowList = memo(function SlowList({ text }) {

```
// Log once. The actual slowdown is inside SlowItem.
console.log('[ARTIFICIALLY SLOW] Rendering 250 <SlowItem />');

let items = [];
for (let i = 0; i < 250; i++) {
items.push(<SlowItem key={i} text={text} />);
}
return (
<ul className="items">
{items}

);
});

function SlowItem({ text }) {
let startTime = performance.now();
while (performance.now() - startTime < 1) {
// Do nothing for 1 ms per item to emulate extremely slow code
}

return (
<li className="item">
Text: {text}

)
}

export default SlowList;
```

```css
.items {
padding: 0;
}

.item {
list-style: none;
display: block;
height: 40px;
```

padding: 5px;

margin-top: 10px;

border-radius: 4px;

border: 1px solid #aaa;

}

```

</Sandpack>

<Solution />

#### Unoptimized re-rendering of the list {/*unoptimized-re-rendering-of-the-list*/}

In this example, each item in the `SlowList` component is **artificially slowed down**, but there is no `useDeferredValue`.

Notice how typing into the input feels very janky. This is because without `useDeferredValue`, each keystroke forces the entire list to re-render immediately in a non-interruptible way.

<Sandpack>

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

import SlowList from './SlowList.js';

export default function App() {

const [text, setText] = useState('');

return (

<>

<input value={text} onChange={e => setText(e.target.value)} />

<SlowList text={text} />

</>

);

}

```

```js SlowList.js
import { memo } from 'react';

const SlowList = memo(function SlowList({ text }) {

// Log once. The actual slowdown is inside SlowItem.

console.log('[ARTIFICIALLY SLOW] Rendering 250 <SlowItem />');

```
let items = [];
for (let i = 0; i < 250; i++) {
items.push(<SlowItem key={i} text={text} />);
}
return (
<ul className="items">
{items}
</ul>
);
});

function SlowItem({ text }) {
let startTime = performance.now();
while (performance.now() - startTime < 1) {
// Do nothing for 1 ms per item to emulate extremely slow code
}

return (
<li className="item">
Text: {text}
</li>
)
}

export default SlowList;
```

```css
.items {
padding: 0;
}

.item {
list-style: none;
display: block;
height: 40px;
padding: 5px;
margin-top: 10px;
border-radius: 4px;
```

border: 1px solid #aaa;

}
```

</Sandpack>

<Solution />

</Recipes>

<Pitfall>

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

</Pitfall>

<DeepDive>

#### How is deferring a value different from debouncing and throttling?
{/*how-is-deferring-a-value-different-from-debouncing-and-throttling*/}

There are two common optimization techniques you might have used before in this scenario:

- *Debouncing* means you'd wait for the user to stop typing (e.g. for a second) before updating the list.
- *Throttling* means you'd update the list every once in a while (e.g. at most once a second).

While these techniques are helpful in some cases, `useDeferredValue` is better suited to optimizing rendering because it is deeply integrated with React itself and adapts to the user's device.

Unlike debouncing or throttling, it doesn't require choosing any fixed delay. If the user's device is fast (e.g. powerful laptop), the deferred re-render would happen almost immediately and wouldn't be noticeable. If the user's device is slow, the list would "lag behind" the input proportionally to how slow the device is.

Also, unlike with debouncing or throttling, deferred re-renders done by `useDeferredValue` are interruptible by default. This means that if React is in the middle of re-rendering a large list, but the user makes another keystroke, React will abandon that re-render, handle the keystroke, and then start rendering in background again. By contrast, debouncing and throttling still produce a janky experience because they're *blocking:* they merely postpone the moment when rendering blocks the keystroke.

If the work you're optimizing doesn't happen during rendering, debouncing and throttling are still useful. For example, they can let you fire fewer network requests. You can also use these techniques together.

</DeepDive>

---

title: <StrictMode>
```

<Intro>

`<StrictMode>` lets you find common bugs in your components early during development.

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

</Intro>

<InlineToc />

Reference {/*reference*/}

`<StrictMode>` {/*strictmode*/}

Use `StrictMode` to enable additional development behaviors and warnings for the component tree inside:

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

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

[See more examples below.](#usage)

Strict Mode enables the following development-only behaviors:

- Your components will [re-render an extra time](#fixing-bugs-found-by-double-rendering-in-development) to find bugs caused by impure rendering.

- Your components will [re-run Effects an extra time](#fixing-bugs-found-by-re-running-effects-in-development) to find bugs caused by missing Effect cleanup.
- Your components will [be checked for usage of deprecated APIs.](#fixing-deprecation-warnings-enabled-by-strict-mode)

Props {/*props*/}

`StrictMode` accepts no props.

Caveats {/*caveats*/}

* There is no way to opt out of Strict Mode inside a tree wrapped in `<StrictMode>`. This gives you confidence that all components inside `<StrictMode>` are checked. If two teams working on a product disagree whether they find the checks valuable, they need to either reach consensus or move `<StrictMode>` down in the tree.

Usage {/*usage*/}

Enabling Strict Mode for entire app {/*enabling-strict-mode-for-entire-app*/}

Strict Mode enables extra development-only checks for the entire component tree inside the `<StrictMode>` component. These checks help you find common bugs in your components early in the development process.

To enable Strict Mode for your entire app, wrap your root component with `<StrictMode>` when you render it:

```js {6,8}
import { StrictMode } from 'react';
import { createRoot } from 'react-dom/client';

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

We recommend wrapping your entire app in Strict Mode, especially for newly created apps. If you use a framework that calls [`createRoot`](/reference/react-dom/client/createRoot) for you, check its documentation for how to enable Strict Mode.

Although the Strict Mode checks **only run in development,** they help you find bugs that already exist in your code but can be tricky to reliably reproduce in production. Strict Mode lets you fix bugs before

your users report them.

<Note>

Strict Mode enables the following checks in development:

- Your components will [re-render an extra time](#fixing-bugs-found-by-double-rendering-in-development) to find bugs caused by impure rendering.
- Your components will [re-run Effects an extra time](#fixing-bugs-found-by-re-running-effects-in-development) to find bugs caused by missing Effect cleanup.
- Your components will [be checked for usage of deprecated APIs.](#fixing-deprecation-warnings-enabled-by-strict-mode)

All of these checks are development-only and do not impact the production build.

</Note>

Enabling strict mode for a part of the app {/*enabling-strict-mode-for-a-part-of-the-app*/}

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

```js {7,12}
import { StrictMode } from 'react';

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

In this example, Strict Mode checks will not run against the `Header` and `Footer` components. However, they will run on `Sidebar` and `Content`, as well as all of the components inside them, no matter how deep.

Fixing bugs found by double rendering in development
{/*fixing-bugs-found-by-double-rendering-in-development*/}

[React assumes that every component you write is a pure function.](/learn/keeping-components-pure) This means that React components you write must always return the same JSX given the same inputs (props, state, and context).

Components breaking this rule behave unpredictably and cause bugs. To help you find accidentally impure code, Strict Mode calls some of your functions (only the ones that should be pure) **twice in development.** This includes:

- Your component function body (only top-level logic, so this doesn't include code inside event handlers)

- Functions that you pass to [`useState`](/reference/react/useState), [`set` functions](/reference/react/useState#setstate), [`useMemo`](/reference/react/useMemo), or [`useReducer`](/reference/react/useReducer)

- Some class component methods like [`constructor`](/reference/react/Component#constructor), [`render`](/reference/react/Component#render), [`shouldComponentUpdate`](/reference/react/Component#shouldcomponentupdate) ([see the whole list](https://reactjs.org/docs/strict-mode.html#detecting-unexpected-side-effects))

If a function is pure, running it twice does not change its behavior because a pure function produces the same result every time. However, if a function is impure (for example, it mutates the data it receives), running it twice tends to be noticeable (that's what makes it impure!) This helps you spot and fix the bug early.

Here is an example to illustrate how double rendering in Strict Mode helps you find bugs early.

This `StoryTray` component takes an array of `stories` and adds one last "Create Story" item at the end:

<Sandpack>

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

import App from './App';

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

```js App.js
```

```
import { useState } from 'react';
import StoryTray from './StoryTray.js';

let initialStories = [
{id: 0, label: "Ankit's Story" },
{id: 1, label: "Taylor's Story" },
];

export default function App() {
let [stories, setStories] = useState(initialStories)
return (
<div
style={{
width: '100%',
height: '100%',
textAlign: 'center',
}}
>
<StoryTray stories={stories} />
</div>
);
}
```

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

}
```

```css
ul {
margin: 0;
list-style-type: none;
height: 100%;
}

li {
border: 1px solid #aaa;
border-radius: 6px;
float: left;
margin: 5px;
margin-bottom: 20px;
padding: 5px;
width: 70px;
height: 100px;
}
```

</Sandpack>

There is a mistake in the code above. However, it is easy to miss because the initial output appears correct.

This mistake will become more noticeable if the `StoryTray` component re-renders multiple times. For example, let's make the `StoryTray` re-render with a different background color whenever you hover over it:

<Sandpack>

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

import App from './App';

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

```js App.js
import { useState } from 'react';
import StoryTray from './StoryTray.js';

let initialStories = [
{id: 0, label: "Ankit's Story" },
{id: 1, label: "Taylor's Story" },
];

export default function App() {
let [stories, setStories] = useState(initialStories)
return (
<div
style={{
width: '100%',
height: '100%',
textAlign: 'center',
}}
>
<StoryTray stories={stories} />
</div>
);
}
```

```js StoryTray.js active
import { useState } from 'react';

export default function StoryTray({ stories }) {
const [isHover, setIsHover] = useState(false);
const items = stories;
items.push({ id: 'create', label: 'Create Story' });
return (
<ul
onPointerEnter={() => setIsHover(true)}
onPointerLeave={() => setIsHover(false)}
style={{
backgroundColor: isHover ? '#ddd' : '#fff'
```

```
}}
>
{items.map(story => (
<li key={story.id}>
{story.label}
</li>
))}
</ul>
);
}
```

```css
ul {
margin: 0;
list-style-type: none;
height: 100%;
}

li {
border: 1px solid #aaa;
border-radius: 6px;
float: left;
margin: 5px;
margin-bottom: 20px;
padding: 5px;
width: 70px;
height: 100px;
}
```

</Sandpack>

Notice how every time you hover over the `StoryTray` component, "Create Story" gets added to the list again. The intention of the code was to add it once at the end. But `StoryTray` directly modifies the `stories` array from the props. Every time `StoryTray` renders, it adds "Create Story" again at the end of the same array. In other words, `StoryTray` is not a pure function--running it multiple times produces different results.

To fix this problem, you can make a copy of the array, and modify that copy instead of the original one:

```js {2}
export default function StoryTray({ stories }) {
const items = stories.slice(); // Clone the array
// ■ Good: Pushing into a new array
items.push({ id: 'create', label: 'Create Story' });
```

This would [make the `StoryTray` function pure.](/learn/keeping-components-pure) Each time it is called, it would only modify a new copy of the array, and would not affect any external objects or variables. This solves the bug, but you had to make the component re-render more often before it became obvious that something is wrong with its behavior.

In the original example, the bug wasn't obvious. Now let's wrap the original (buggy) code in `<StrictMode>`:

<Sandpack>

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

import App from './App';

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

```js App.js
import { useState } from 'react';
import StoryTray from './StoryTray.js';

let initialStories = [
{id: 0, label: "Ankit's Story" },
{id: 1, label: "Taylor's Story" },
];

export default function App() {
let [stories, setStories] = useState(initialStories)
```

```
return (
<div
style={{
width: '100%',
height: '100%',
textAlign: 'center',
}}
>
<StoryTray stories={stories} />
</div>
);
}
```

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

```css
ul {
margin: 0;
list-style-type: none;
height: 100%;
}
```

li {

border: 1px solid #aaa;

border-radius: 6px;

float: left;

margin: 5px;

margin-bottom: 20px;

padding: 5px;

width: 70px;

height: 100px;

}

```

</Sandpack>

**Strict Mode *always* calls your rendering function twice, so you can see the mistake right away** ("Create Story" appears twice). This lets you notice such mistakes early in the process. When you fix your component to render in Strict Mode, you *also* fix many possible future production bugs like the hover functionality from before:

<Sandpack>

```js index.js

import { StrictMode } from 'react';

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

import './styles.css';

import App from './App';

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

root.render(

<StrictMode>

<App />

</StrictMode>

);

```

```js App.js

import { useState } from 'react';

import StoryTray from './StoryTray.js';

let initialStories = [

{id: 0, label: "Ankit's Story" },

```
{id: 1, label: "Taylor's Story" },
];

export default function App() {
let [stories, setStories] = useState(initialStories)
return (
<div
style={{
width: '100%',
height: '100%',
textAlign: 'center',
}}
>
<StoryTray stories={stories} />
</div>
);
}
```

```js StoryTray.js active
import { useState } from 'react';

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


))}

);
}
```

```css
ul {
margin: 0;
list-style-type: none;
height: 100%;
}

li {
border: 1px solid #aaa;
border-radius: 6px;
float: left;
margin: 5px;
margin-bottom: 20px;
padding: 5px;
width: 70px;
height: 100px;
}
```

</Sandpack>

Without Strict Mode, it was easy to miss the bug until you added more re-renders. Strict Mode made the same bug appear right away. Strict Mode helps you find bugs before you push them to your team and to your users.

[Read more about keeping components pure.](/learn/keeping-components-pure)

<Note>

If you have [React DevTools](/learn/react-developer-tools) installed, any `console.log` calls during the second render call will appear slightly dimmed. React DevTools also offers a setting (off by default) to suppress them completely.

</Note>

```

Fixing bugs found by re-running Effects in development
{/*fixing-bugs-found-by-re-running-effects-in-development*/}

Strict Mode can also help find bugs in [Effects.](/learn/synchronizing-with-effects)

Every Effect has some setup code and may have some cleanup code. Normally, React calls setup when the component *mounts* (is added to the screen) and calls cleanup when the component *unmounts* (is removed from the screen). React then calls cleanup and setup again if its dependencies changed since the last render.

When Strict Mode is on, React will also run **one extra setup+cleanup cycle in development for every Effect.** This may feel surprising, but it helps reveal subtle bugs that are hard to catch manually.

Here is an example to illustrate how re-running Effects in Strict Mode helps you find bugs early.

Consider this example that connects a component to a chat:

<Sandpack>

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

import App from './App';

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

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

const serverUrl = 'https://localhost:1234';
const roomId = 'general';

export default function ChatRoom() {
useEffect(() => {
const connection = createConnection(serverUrl, roomId);
connection.connect();
}, []);
return <h1>Welcome to the {roomId} room!</h1>;
}
```

```js chat.js
```

```
let connections = 0;

export function createConnection(serverUrl, roomId) {
// A real implementation would actually connect to the server
return {
connect() {
console.log('■ Connecting to "' + roomId + '" room at ' + serverUrl + '...');
connections++;
console.log('Active connections: ' + connections);
},
disconnect() {
console.log('■ Disconnected from "' + roomId + '" room at ' + serverUrl);
connections--;
console.log('Active connections: ' + connections);
}
};
}
```

```css
input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }
```

</Sandpack>

There is an issue with this code, but it might not be immediately clear.

To make the issue more obvious, let's implement a feature. In the example below, `roomId` is not hardcoded. Instead, the user can select the `roomId` that they want to connect to from a dropdown. Click "Open chat" and then select different chat rooms one by one. Keep track of the number of active connections in the console:

<Sandpack>

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

import App from './App';

const root = createRoot(document.getElementById("root"));
```

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

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

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

function ChatRoom({ roomId }) {
useEffect(() => {
const connection = createConnection(serverUrl, roomId);
connection.connect();
}, [roomId]);

return <h1>Welcome to the {roomId} room!</h1>;
}

export default function App() {
const [roomId, setRoomId] = useState('general');
const [show, setShow] = useState(false);
return (
<>
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
</select>
</label>
<button onClick={() => setShow(!show)}>
{show ? 'Close chat' : 'Open chat'}
</button>
{show && <hr />}
{show && <ChatRoom roomId={roomId} />}
```

```
</>
);
}
```

```js chat.js
let connections = 0;

export function createConnection(serverUrl, roomId) {
// A real implementation would actually connect to the server
return {
connect() {
console.log('■ Connecting to "' + roomId + '" room at ' + serverUrl + '...');
connections++;
console.log('Active connections: ' + connections);
},
disconnect() {
console.log('■ Disconnected from "' + roomId + '" room at ' + serverUrl);
connections--;
console.log('Active connections: ' + connections);
}
};
}
```

```css
input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }
```

</Sandpack>

You'll notice that the number of open connections always keeps growing. In a real app, this would cause performance and network problems. The issue is that [your Effect is missing a cleanup function:](/learn/synchronizing-with-effects#step-3-add-cleanup-if-needed)

```js {4}
useEffect(() => {
const connection = createConnection(serverUrl, roomId);
connection.connect();
```

return () => connection.disconnect();

}, [roomId]);

```

Now that your Effect "cleans up" after itself and destroys the outdated connections, the leak is solved. However, notice that the problem did not become visible until you've added more features (the select box).

**In the original example, the bug wasn't obvious. Now let's wrap the original (buggy) code in `<StrictMode>`:**

<Sandpack>

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

import App from './App';

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

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

const serverUrl = 'https://localhost:1234';
const roomId = 'general';

export default function ChatRoom() {
useEffect(() => {
const connection = createConnection(serverUrl, roomId);
connection.connect();
}, []);
return <h1>Welcome to the {roomId} room!</h1>;
}

```

```js chat.js
let connections = 0;

export function createConnection(serverUrl, roomId) {
// A real implementation would actually connect to the server
return {
connect() {
console.log('■ Connecting to "' + roomId + '" room at ' + serverUrl + '...');
connections++;
console.log('Active connections: ' + connections);
},
disconnect() {
console.log('■ Disconnected from "' + roomId + '" room at ' + serverUrl);
connections--;
console.log('Active connections: ' + connections);
}
};
}
```

```css
input { display: block; margin-bottom: 20px; }
button { margin-left: 10px; }
```

</Sandpack>

With Strict Mode, you immediately see that there is a problem (the number of active connections jumps to 2). Strict Mode runs an extra setup+cleanup cycle for every Effect. This Effect has no cleanup logic, so it creates an extra connection but doesn't destroy it. This is a hint that you're missing a cleanup function.

Strict Mode lets you notice such mistakes early in the process. When you fix your Effect by adding a cleanup function in Strict Mode, you *also* fix many possible future production bugs like the select box from before:

<Sandpack>

```js index.js
import { StrictMode } from 'react';
```
```

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

import App from './App';

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

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

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

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

return <h1>Welcome to the {roomId} room!</h1>;
}

export default function App() {
const [roomId, setRoomId] = useState('general');
const [show, setShow] = useState(false);
return (
<>
<label>
Choose the chat room:{' '}
<select
value={roomId}
onChange={e => setRoomId(e.target.value)}
>
````

<option value="general">general</option>
<option value="travel">travel</option>
<option value="music">music</option>
</select>
</label>
<button onClick={() => setShow(!show)}>
{show ? 'Close chat' : 'Open chat'}
</button>
{show && <hr />}
{show && <ChatRoom roomId={roomId} />}
</>
);
}
```

```js chat.js
let connections = 0;

export function createConnection(serverUrl, roomId) {
// A real implementation would actually connect to the server
return {
connect() {
console.log('■ Connecting to "' + roomId + '" room at ' + serverUrl + '...');
connections++;
console.log('Active connections: ' + connections);
},
disconnect() {
console.log('■ Disconnected from "' + roomId + '" room at ' + serverUrl);
connections--;
console.log('Active connections: ' + connections);
}
};
}
```

```css
input { display: block; margin-bottom: 20px; }

button { margin-left: 10px; }
```

</Sandpack>

Notice how the active connection count in the console doesn't keep growing anymore.

Without Strict Mode, it was easy to miss that your Effect needed cleanup. By running *setup → cleanup → setup* instead of *setup* for your Effect in development, Strict Mode made the missing cleanup logic more noticeable.

[Read more about implementing Effect cleanup.](/learn/synchronizing-with-effects#how-to-handle-the-effect-firing-twice-in-development)

Fixing deprecation warnings enabled by Strict Mode {/*fixing-deprecation-warnings-enabled-by-strict-mode*/}

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

* [`findDOMNode`](/reference/react-dom/findDOMNode). [See alternatives.](https://reactjs.org/docs/strict-mode.html#warning-about-deprecated-finddomnode-usage)
* `UNSAFE_` class lifecycle methods like [`UNSAFE_componentWillMount`](/reference/react/Component#unsafe_componentwillmount). [See alternatives.](https://reactjs.org/blog/2018/03/27/update-on-async-rendering.html#migrating-from-legacy-lifecycles)
* Legacy context ([`childContextTypes`](/reference/react/Component#static-childcontexttypes), [`contextTypes`](/reference/react/Component#static-contexttypes), and [`getChildContext`](/reference/react/Component#getchildcontext)). [See alternatives.](/reference/react/createContext)
* Legacy string refs ([`this.refs`](/reference/react/Component#refs)). [See alternatives.](https://reactjs.org/docs/strict-mode.html#warning-about-legacy-string-ref-api-usage)

These APIs are primarily used in older [class components](/reference/react/Component) so they rarely appear in modern apps.

---

title: isValidElement

<Intro>

`isValidElement` checks whether a value is a React element.

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

</Intro>

<InlineToc />

Reference {/*reference*/}

`isValidElement(value)` {/*isvalidelement*/}

Call `isValidElement(value)` to check whether `value` is a React element.

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

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

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

[See more examples below.](#usage)

Parameters {/*parameters*/}

* `value`: The `value` you want to check. It can be any a value of any type.

Returns {/*returns*/}

`isValidElement` returns `true` if the `value` is a React element. Otherwise, it returns `false`.

Caveats {/*caveats*/}

* **Only [JSX tags](/learn/writing-markup-with-jsx) and objects returned by [`createElement`](/reference/react/createElement) are considered to be React elements.** For example, even though a number like `42` is a valid React *node* (and can be returned from a component), it is not a valid React element. Arrays and portals created with [`createPortal`](/reference/react-dom/createPortal) are also *not* considered to be React elements.

Usage {/*usage*/}

Checking if something is a React element {/*checking-if-something-is-a-react-element*/}

Call `isValidElement` to check if some value is a *React element.*

React elements are:

- Values produced by writing a [JSX tag](/learn/writing-markup-with-jsx)
- Values produced by calling [`createElement`](/reference/react/createElement)

For React elements, `isValidElement` returns `true`:

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

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

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

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

For them, `isValidElement` returns `false`:

```js
// ■ These are *not* React elements
console.log(isValidElement(null)); // false
console.log(isValidElement(25)); // false
console.log(isValidElement('Hello')); // false
console.log(isValidElement({ age: 42 })); // false
console.log(isValidElement([<div />, <div />])); // false
console.log(isValidElement(MyComponent)); // false
```

It is very uncommon to need `isValidElement`. It's mostly useful if you're calling another API that *only* accepts elements (like [`cloneElement`](/reference/react/cloneElement) does) and you want to avoid an error when your argument is not a React element.

Unless you have some very specific reason to add an `isValidElement` check, you probably don't need it.

<DeepDive>

React elements vs React nodes {/*react-elements-vs-react-nodes*/}

When you write a component, you can return any kind of *React node* from it:

```js
function MyComponent() {
// ... you can return any React node ...
}
```

A React node can be:

- A React element created like `<div />` or `createElement('div')`
- A portal created with [`createPortal`](/reference/react-dom/createPortal)
- A string
- A number
- `true`, `false`, `null`, or `undefined` (which are not displayed)
- An array of other React nodes

Note `isValidElement` checks whether the argument is a *React element,* not whether it's a React node. For example, `42` is not a valid React element. However, it is a perfectly valid React node:

```js
function MyComponent() {
return 42; // It's ok to return a number from component
}
```

This is why you shouldn't use `isValidElement` as a way to check whether something can be rendered.

</DeepDive>

---

title: forwardRef

<Intro>

`forwardRef` lets your component expose a DOM node to parent component with a [ref.](/learn/manipulating-the-dom-with-refs)

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

</Intro>

<InlineToc />

Reference {/*reference*/}

`forwardRef(render)` {/*forwardref*/}

Call `forwardRef()` to let your component receive a ref and forward it to a child component:

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

const MyInput = forwardRef(function MyInput(props, ref) {
// ...
});
```

[See more examples below.](#usage)

Parameters {/*parameters*/}

* `render`: The render function for your component. React calls this function with the props and `ref` that your component received from its parent. The JSX you return will be the output of your component.

Returns {/*returns*/}

`forwardRef` returns a React component that you can render in JSX. Unlike React components defined as plain functions, a component returned by `forwardRef` is also able to receive a `ref` prop.

Caveats {/*caveats*/}

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

`render` function {/*render-function*/}

`forwardRef` accepts a render function as an argument. React calls this function with `props` and `ref`:

```js
const MyInput = forwardRef(function MyInput(props, ref) {
return (
<label>
{props.label}
```

<input ref={ref} />

</label>

);

});

```

#### Parameters {/*render-parameters*/}

* `props`: The props passed by the parent component.

* `ref`: The `ref` attribute passed by the parent component. The `ref` can be an object or a function. If the parent component has not passed a ref, it will be `null`. You should either pass the `ref` you receive to another component, or pass it to [`useImperativeHandle`.](/reference/react/useImperativeHandle)

#### Returns {/*render-returns*/}

`forwardRef` returns a React component that you can render in JSX. Unlike React components defined as plain functions, the component returned by `forwardRef` is able to take a `ref` prop.

---

## Usage {/*usage*/}

### Exposing a DOM node to the parent component
{/*exposing-a-dom-node-to-the-parent-component*/}

By default, each component's DOM nodes are private. However, sometimes it's useful to expose a DOM node to the parent--for example, to allow focusing it. To opt in, wrap your component definition into `forwardRef()`:

```js {3,11}
import { forwardRef } from 'react';

const MyInput = forwardRef(function MyInput(props, ref) {

const { label, ...otherProps } = props;

return (

<label>

{label}

<input {...otherProps} />

</label>

);

});

```

You will receive a <CodeStep step={1}>ref</CodeStep> as the second argument after props. Pass it to the DOM node that you want to expose:
```

```js {8} [[1, 3, "ref"], [1, 8, "ref", 30]]
import { forwardRef } from 'react';

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

This lets the parent `Form` component access the <CodeStep step={2}>`<input>` DOM node</CodeStep> exposed by `MyInput`:

```js [[1, 2, "ref"], [1, 10, "ref", 41], [2, 5, "ref.current"]]
function Form() {
const ref = useRef(null);

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

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

This `Form` component [passes a ref](/reference/react/useRef#manipulating-the-dom-with-a-ref) to `MyInput`. The `MyInput` component *forwards* that ref to the `<input>` browser tag. As a result, the `Form` component can access that `<input>` DOM node and call [`focus()`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/focus) on it.

Keep in mind that exposing a ref to the DOM node inside your component makes it harder to change your component's internals later. You will typically expose DOM nodes from reusable low-level components like buttons or text inputs, but you won't do it for application-level components like an avatar or a comment.

<Recipes title="Examples of forwarding a ref">

Focusing a text input {/*focusing-a-text-input*/}

Clicking the button will focus the input. The `Form` component defines a ref and passes it to the `MyInput` component. The `MyInput` component forwards that ref to the browser `<input>`. This lets the `Form` component focus the `<input>`.

<Sandpack>

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

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

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

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

```js MyInput.js
import { forwardRef } from 'react';

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

{label}

<input {...otherProps} ref={ref} />

</label>

);

});

export default MyInput;

```

```css

input {

margin: 5px;

}

```

</Sandpack>

<Solution />

Playing and pausing a video {/*playing-and-pausing-a-video*/}

Clicking the button will call [`play()`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement/play) and [`pause()`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement/pause) on a `<video>` DOM node. The `App` component defines a ref and passes it to the `MyVideoPlayer` component. The `MyVideoPlayer` component forwards that ref to the browser `<video>` node. This lets the `App` component play and pause the `<video>`.

<Sandpack>

```js

import { useRef } from 'react';

import MyVideoPlayer from './MyVideoPlayer.js';

export default function App() {

const ref = useRef(null);

return (

<>

<button onClick={() => ref.current.play()}>

Play

</button>

<button onClick={() => ref.current.pause()}>

Pause

</button>
<br />
<MyVideoPlayer
ref={ref}
src="https://interactive-examples.mdn.mozilla.net/media/cc0-videos/flower.mp4"
type="video/mp4"
width="250"
/>
</>
);
}
```

```js MyVideoPlayer.js
import { forwardRef } from 'react';

const VideoPlayer = forwardRef(function VideoPlayer({ src, type, width }, ref) {
return (
<video width={width} ref={ref}>
<source
src={src}
type={type}
/>
</video>
);
});

export default VideoPlayer;
```

```css
button { margin-bottom: 10px; margin-right: 10px; }
```

</Sandpack>

<Solution />

</Recipes>

```

Forwarding a ref through multiple components {/*forwarding-a-ref-through-multiple-components*/}

Instead of forwarding a `ref` to a DOM node, you can forward it to your own component like `MyInput`:

```js {1,5}
const FormField = forwardRef(function FormField(props, ref) {
// ...
return (
<>
<MyInput ref={ref} />
...
</>
);
});
```

If that `MyInput` component forwards a ref to its `<input>`, a ref to `FormField` will give you that `<input>`:

```js {2,5,10}
function Form() {
const ref = useRef(null);

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

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

The `Form` component defines a ref and passes it to `FormField`. The `FormField` component forwards that ref to `MyInput`, which forwards it to a browser `<input>` DOM node. This is how `Form` accesses that DOM node.

<Sandpack>

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

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

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

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

```js FormField.js
import { forwardRef, useState } from 'react';
import MyInput from './MyInput.js';

const FormField = forwardRef(function FormField({ label, isRequired }, ref) {
const [value, setValue] = useState('');
return (
<>
<MyInput
ref={ref}
label={label}
value={value}
onChange={e => setValue(e.target.value)}
/>
{(isRequired && value === '') &&
```

Required

}

</>

);

});

export default FormField;

```

```js MyInput.js
import { forwardRef } from 'react';

const MyInput = forwardRef((props, ref) => {

const { label, ...otherProps } = props;

return (

<label>

{label}

<input {...otherProps} ref={ref} />

</label>

);

});

export default MyInput;

```

```css
input, button {

margin: 5px;

}

```

</Sandpack>

Exposing an imperative handle instead of a DOM node
{/*exposing-an-imperative-handle-instead-of-a-dom-node*/}

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

```js {2,6}

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

// ...

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

Pass the `ref` you received to [`useImperativeHandle`](/reference/react/useImperativeHandle) and specify the value you want to expose to the `ref`:

```js {6-15}
import { forwardRef, useRef, useImperativeHandle } from 'react';

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

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

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

If some component gets a ref to `MyInput`, it will only receive your `{ focus, scrollIntoView }` object instead of the DOM node. This lets you limit the information you expose about your DOM node to the minimum.

<Sandpack>

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

export default function Form() {
```

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

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

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

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

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

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

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

export default MyInput;
````

```

```css
input {
margin: 5px;
}
```

</Sandpack>

[Read more about using imperative handles.](/reference/react/useImperativeHandle)

<Pitfall>

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

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

</Pitfall>

Troubleshooting {/*troubleshooting*/}

My component is wrapped in `forwardRef`, but the `ref` to it is always `null` {/*my-component-is-wrapped-in-forwardref-but-the-ref-to-it-is-always-null*/}

This usually means that you forgot to actually use the `ref` that you received.

For example, this component doesn't do anything with its `ref`:

```js {1}
const MyInput = forwardRef(function MyInput({ label }, ref) {
return (
<label>
{label}
<input />
</label>
);
});
```
```

To fix it, pass the `ref` down to a DOM node or another component that can accept a ref:

```js {1,5}
const MyInput = forwardRef(function MyInput({ label }, ref) {
return (
<label>
{label}
<input ref={ref} />
</label>
);
});
```

The `ref` to `MyInput` could also be `null` if some of the logic is conditional:

```js {1,5}
const MyInput = forwardRef(function MyInput({ label, showInput }, ref) {
return (
<label>
{label}
{showInput && <input ref={ref} />}
</label>
);
});
```

If `showInput` is `false`, then the ref won't be forwarded to any node, and a ref to `MyInput` will remain empty. This is particularly easy to miss if the condition is hidden inside another component, like `Panel` in this example:

```js {5,7}
const MyInput = forwardRef(function MyInput({ label, showInput }, ref) {
return (
<label>
{label}
<Panel isExpanded={showInput}>
<input ref={ref} />
</Panel>
</label>
```

);

});

```

---

title: useInsertionEffect

---

<Pitfall>

`useInsertionEffect` is for CSS-in-JS library authors. Unless you are working on a CSS-in-JS library and need a place to inject the styles, you probably want [`useEffect`](/reference/react/useEffect) or [`useLayoutEffect`](/reference/react/useLayoutEffect) instead.

</Pitfall>

<Intro>

`useInsertionEffect` allows inserting elements into the DOM before any layout effects fire.

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

</Intro>

<InlineToc />

Reference {/*reference*/}

`useInsertionEffect(setup, dependencies?)` {/*useinsertioneffect*/}

Call `useInsertionEffect` to insert styles before any effects fire that may need to read layout:

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

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

```

[See more examples below.](#usage)

Parameters {/*parameters*/}

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

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

Returns {/*returns*/}

`useInsertionEffect` returns `undefined`.

Caveats {/*caveats*/}

* Effects only run on the client. They don't run during server rendering.

* You can't update state from inside `useInsertionEffect`.

* By the time `useInsertionEffect` runs, refs are not attached yet.

* `useInsertionEffect` may run either before or after the DOM has been updated. You shouldn't rely on the DOM being updated at any particular time.

* Unlike other types of Effects, which fire cleanup for every Effect and then setup for every Effect, `useInsertionEffect` will fire both cleanup and setup one component at a time. This results in an "interleaving" of the cleanup and setup functions.

Usage {/*usage*/}

Injecting dynamic styles from CSS-in-JS libraries
{/*injecting-dynamic-styles-from-css-in-js-libraries*/}

Traditionally, you would style React components using plain CSS.

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

// In your CSS file:
```
```

.success { color: green; }
```

Some teams prefer to author styles directly in JavaScript code instead of writing CSS files. This usually requires using a CSS-in-JS library or a tool. There are three common approaches to CSS-in-JS:

1. Static extraction to CSS files with a compiler
2. Inline styles, e.g. `&lt;div style={{ opacity: 1 }}&gt;`
3. Runtime injection of `&lt;style&gt;` tags

If you use CSS-in-JS, we recommend a combination of the first two approaches (CSS files for static styles, inline styles for dynamic styles). **We don't recommend runtime `&lt;style&gt;` tag injection for two reasons:**

1. Runtime injection forces the browser to recalculate the styles a lot more often.
2. Runtime injection can be very slow if it happens at the wrong time in the React lifecycle.

The first problem is not solvable, but `useInsertionEffect` helps you solve the second problem.

Call `useInsertionEffect` to insert the styles before any layout effects fire:

```js {4-11}
// Inside your CSS-in-JS library
let isInserted = new Set();
function useCSS(rule) {
useInsertionEffect(() => {
// As explained earlier, we don't recommend runtime injection of <style> tags.
// But if you have to do it, then it's important to do in useInsertionEffect.
if (!isInserted.has(rule)) {
isInserted.add(rule);
document.head.appendChild(getStyleForRule(rule));
}
});
return rule;
}

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

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

```js {1,4-6}
let collectedRulesSet = new Set();

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

[Read more about upgrading CSS-in-JS libraries with runtime injection to `useInsertionEffect`.](https://github.com/reactwg/react-18/discussions/110)

<DeepDive>

How is this better than injecting styles during rendering or useLayoutEffect? {/*how-is-this-better-than-injecting-styles-during-rendering-or-uselayouteffect*/}

If you insert styles during rendering and React is processing a [non-blocking update,](/reference/react/useTransition#marking-a-state-update-as-a-non-blocking-transition) the browser will recalculate the styles every single frame while rendering a component tree, which can be **extremely slow.**

`useInsertionEffect` is better than inserting styles during [`useLayoutEffect`](/reference/react/useLayoutEffect) or [`useEffect`](/reference/react/useEffect) because it ensures that by the time other Effects run in your components, the `<style>` tags have already been inserted. Otherwise, layout calculations in regular Effects would be wrong due to outdated styles.

</DeepDive>

---

title: memo

<Intro>

`memo` lets you skip re-rendering a component when its props are unchanged.

```
```

const MemoizedComponent = memo(SomeComponent, arePropsEqual?)
```

</Intro>

<InlineToc />

---

## Reference {/*reference*/}

### `memo(Component, arePropsEqual?)` {/*memo*/}

Wrap a component in `memo` to get a *memoized* version of that component. This memoized version of your component will usually not be re-rendered when its parent component is re-rendered as long as its props have not changed. But React may still re-render it: memoization is a performance optimization, not a guarantee.

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

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

[See more examples below.](#usage)

Parameters {/*parameters*/}

* `Component`: The component that you want to memoize. The `memo` does not modify this component, but returns a new, memoized component instead. Any valid React component, including functions and [`forwardRef`](/reference/react/forwardRef) components, is accepted.

* **optional** `arePropsEqual`: A function that accepts two arguments: the component's previous props, and its new props. It should return `true` if the old and new props are equal: that is, if the component will render the same output and behave in the same way with the new props as with the old. Otherwise it should return `false`. Usually, you will not specify this function. By default, React will compare each prop with [`Object.is`.](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/is)

Returns {/*returns*/}

`memo` returns a new React component. It behaves the same as the component provided to `memo` except that React will not always re-render it when its parent is being re-rendered unless its props have changed.

Usage {/*usage*/}
```

Skipping re-rendering when props are unchanged
{/*skipping-re-rendering-when-props-are-unchanged*/}

React normally re-renders a component whenever its parent re-renders. With `memo`, you can create a component that React will not re-render when its parent re-renders so long as its new props are the same as the old props. Such a component is said to be *memoized*.

To memoize a component, wrap it in `memo` and use the value that it returns in place of your original component:

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

export default Greeting;
```

A React component should always have [pure rendering logic.](/learn/keeping-components-pure) This means that it must return the same output if its props, state, and context haven't changed. By using `memo`, you are telling React that your component complies with this requirement, so React doesn't need to re-render as long as its props haven't changed. Even with `memo`, your component will re-render if its own state changes or if a context that it's using changes.

In this example, notice that the `Greeting` component re-renders whenever `name` is changed (because that's one of its props), but not when `address` is changed (because it's not passed to `Greeting` as a prop):

<Sandpack>

```js
import { memo, useState } from 'react';

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

</label>

<Greeting name={name} />

</>

);

}

const Greeting = memo(function Greeting({ name }) {

console.log("Greeting was rendered at", new Date().toLocaleTimeString());

return <h3>Hello{name && ', '}{name}!</h3>;

});

```

```css

label {

display: block;

margin-bottom: 16px;

}

```

</Sandpack>

<Note>

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

</Note>

<DeepDive>

Should you add memo everywhere? {/*should-you-add-memo-everywhere*/}

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

Optimizing with `memo` is only valuable when your component re-renders often with the same exact props, and its re-rendering logic is expensive. If there is no perceptible lag when your component re-renders, `memo` is unnecessary. Keep in mind that `memo` is completely useless if the props passed to your component are *always different,* such as if you pass an object or a plain function defined during rendering. This is why you will often need [`useMemo`](/reference/react/useMemo#skipping-re-rendering-of-components) and [`useCallback`](/reference/react/useCallback#skipping-re-rendering-of-components) together with `memo`.
```

There is no benefit to wrapping a component in `memo` in other cases. There is no significant harm to doing that either, so some teams choose to not think about individual cases, and memoize as much as possible. The downside of this approach is that code becomes less readable. Also, not all memoization is effective: a single value that's "always new" is enough to break memoization for an entire component.

In practice, you can make a lot of memoization unnecessary by following a few principles:

1. When a component visually wraps other components, let it [accept JSX as children.](/learn/passing-props-to-a-component#passing-jsx-as-children) This way, when the wrapper component updates its own state, React knows that its children don't need to re-render.

1. Prefer local state and don't [lift state up](/learn/sharing-state-between-components) any further than necessary. For example, don't keep transient state like forms and whether an item is hovered at the top of your tree or in a global state library.

1. Keep your [rendering logic pure.](/learn/keeping-components-pure) If re-rendering a component causes a problem or produces some noticeable visual artifact, it's a bug in your component! Fix the bug instead of adding memoization.

1. Avoid [unnecessary Effects that update state.](/learn/you-might-not-need-an-effect) Most performance problems in React apps are caused by chains of updates originating from Effects that cause your components to render over and over.

1. Try to [remove unnecessary dependencies from your Effects.](/learn/removing-effect-dependencies) For example, instead of memoization, it's often simpler to move some object or a function inside an Effect or outside the component.

If a specific interaction still feels laggy, [use the React Developer Tools profiler](https://legacy.reactjs.org/blog/2018/09/10/introducing-the-react-profiler.html) to see which components would benefit the most from memoization, and add memoization where needed. These principles make your components easier to debug and understand, so it's good to follow them in any case. In the long term, we're researching [doing granular memoization automatically](https://www.youtube.com/watch?v=lGEMwh32soc) to solve this once and for all.

</DeepDive>

Updating a memoized component using state {/*updating-a-memoized-component-using-state*/}

Even when a component is memoized, it will still re-render when its own state changes. Memoization only has to do with props that are passed to the component from its parent.

<Sandpack>

```js
import { memo, useState } from 'react';

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

```
<>
<label>
Name{': '}
<input value={name} onChange={e => setName(e.target.value)} />
</label>
<label>
Address{': '}
<input value={address} onChange={e => setAddress(e.target.value)} />
</label>
<Greeting name={name} />
</>
);
}

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

function GreetingSelector({ value, onChange }) {
return (
<>
<label>
<input
type="radio"
checked={value === 'Hello'}
onChange={e => onChange('Hello')}
/>
Regular greeting
</label>
<label>
```

<input

type="radio"

checked={value === 'Hello and welcome'}

onChange={e => onChange('Hello and welcome')}

/>

Enthusiastic greeting

</label>

</>

);

}

```

```css

label {

display: block;

margin-bottom: 16px;

}

```

</Sandpack>

If you set a state variable to its current value, React will skip re-rendering your component even without `memo`. You may still see your component function being called an extra time, but the result will be discarded.

Updating a memoized component using a context
{/*updating-a-memoized-component-using-a-context*/}

Even when a component is memoized, it will still re-render when a context that it's using changes. Memoization only has to do with props that are passed to the component from its parent.

<Sandpack>

```js

import { createContext, memo, useContext, useState } from 'react';

const ThemeContext = createContext(null);

export default function MyApp() {

const [theme, setTheme] = useState('dark');

function handleClick() {

```
setTheme(theme === 'dark' ? 'light' : 'dark');
}

return (
<ThemeContext.Provider value={theme}>
<button onClick={handleClick}>
Switch theme
</button>
<Greeting name="Taylor" />
</ThemeContext.Provider>
);
}

const Greeting = memo(function Greeting({ name }) {
console.log("Greeting was rendered at", new Date().toLocaleTimeString());
const theme = useContext(ThemeContext);
return (
<h3 className={theme}>Hello, {name}!</h3>
);
});
```

```css
label {
display: block;
margin-bottom: 16px;
}

.light {
color: black;
background-color: white;
}

.dark {
color: white;
background-color: black;
}
```

</Sandpack>

To make your component re-render only when a _part_ of some context changes, split your component in two. Read what you need from the context in the outer component, and pass it down to a memoized child as a prop.

Minimizing props changes {/*minimizing-props-changes*/}

When you use `memo`, your component re-renders whenever any prop is not *shallowly equal* to what it was previously. This means that React compares every prop in your component with its previous value using the [`Object.is`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/is) comparison. Note that `Object.is(3, 3)` is `true`, but `Object.is({}, {})` is `false`.

To get the most out of `memo`, minimize the times that the props change. For example, if the prop is an object, prevent the parent component from re-creating that object every time by using [`useMemo`:](/reference/react/useMemo)

```js {5-8}
function Page() {
const [name, setName] = useState('Taylor');
const [age, setAge] = useState(42);

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

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

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

A better way to minimize props changes is to make sure the component accepts the minimum necessary information in its props. For example, it could accept individual values instead of a whole object:

```js {4,7}
function Page() {
const [name, setName] = useState('Taylor');
const [age, setAge] = useState(42);
```

return <Profile name={name} age={age} />;

}

const Profile = memo(function Profile({ name, age }) {

// ...

});
```

Even individual values can sometimes be projected to ones that change less frequently. For example, here a component accepts a boolean indicating the presence of a value rather than the value itself:

```js {3}
function GroupsLanding({ person }) {

const hasGroups = person.groups !== null;

return <CallToAction hasGroups={hasGroups} />;

}

const CallToAction = memo(function CallToAction({ hasGroups }) {

// ...

});
```

When you need to pass a function to memoized component, either declare it outside your component so that it never changes, or [`useCallback`](/reference/react/useCallback#skipping-re-rendering-of-components) to cache its definition between re-renders.

Specifying a custom comparison function {/*specifying-a-custom-comparison-function*/}

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

```js {3}
const Chart = memo(function Chart({ dataPoints }) {

// ...

}, arePropsEqual);

function arePropsEqual(oldProps, newProps) {

return (

oldProps.dataPoints.length === newProps.dataPoints.length &&

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

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

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

<Pitfall>

If you provide a custom `arePropsEqual` implementation, **you must compare every prop, including functions.** Functions often [close over](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures) the props and state of parent components. If you return `true` when `oldProps.onClick !== newProps.onClick`, your component will keep "seeing" the props and state from a previous render inside its `onClick` handler, leading to very confusing bugs.

Avoid doing deep equality checks inside `arePropsEqual` unless you are 100% sure that the data structure you're working with has a known limited depth. **Deep equality checks can become incredibly slow** and can freeze your app for many seconds if someone changes the data structure later.

</Pitfall>

Troubleshooting {/*troubleshooting*/}

My component re-renders when a prop is an object, array, or function {/*my-component-rerenders-when-a-prop-is-an-object-or-array*/}

React compares old and new props by shallow equality: that is, it considers whether each new prop is reference-equal to the old prop. If you create a new object or array each time the parent is re-rendered, even if the individual elements are each the same, React will still consider it to be changed. Similarly, if you create a new function when rendering the parent component, React will consider it to have changed even if the function has the same definition. To avoid this, [simplify props or memoize props in the parent component](#minimizing-props-changes).

---

title: createFactory

<Deprecated>

This API will be removed in a future major version of React. [See the alternatives.](#alternatives)

</Deprecated>

<Intro>

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

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

</Intro>

<InlineToc />

Reference {/*reference*/}

`createFactory(type)` {/*createfactory*/}

Call `createFactory(type)` to create a factory function which produces React elements of a given `type`.

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

const button = createFactory('button');
```

Then you can use it to create React elements without JSX:

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

[See more examples below.](#usage)

Parameters {/*parameters*/}

* `type`: The `type` argument must be a valid React component type. For example, it could be a tag name string (such as `'div'` or `'span'`), or a React component (a function, a class, or a special component like [`Fragment`](/reference/react/Fragment)).

Returns {/*returns*/}

Returns a factory function. That factory function receives a `props` object as the first argument, followed by a list of `...children` arguments, and returns a React element with the given `type`, `props` and `children`.

Usage {/*usage*/}

Creating React elements with a factory {/*creating-react-elements-with-a-factory*/}

Although most React projects use [JSX](/learn/writing-markup-with-jsx) to describe the user interface, JSX is not required. In the past, `createFactory` used to be one of the ways you could describe the user interface without JSX.

Call `createFactory` to create a *factory function* for a specific element type like `'button'`:

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

const button = createFactory('button');
```

Calling that factory function will produce React elements with the props and children you have provided:

<Sandpack>

```js App.js
import { createFactory } from 'react';

const button = createFactory('button');

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

</Sandpack>

This is how `createFactory` was used as an alternative to JSX. However, `createFactory` is deprecated, and you should not call `createFactory` in any new code. See how to migrate away from `createFactory` below.

Alternatives {/*alternatives*/}

Copying `createFactory` into your project {/*copying-createfactory-into-your-project*/}

If your project has many `createFactory` calls, copy this `createFactory.js` implementation into your project:

<Sandpack>

```js App.js
import { createFactory } from './createFactory.js';

const button = createFactory('button');

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

```js createFactory.js
import { createElement } from 'react';

export function createFactory(type) {
return createElement.bind(null, type);
}
```

</Sandpack>

This lets you keep all of your code unchanged except the imports.

Replacing `createFactory` with `createElement` {/*replacing-createfactory-with-createelement*/}

If you have a few `createFactory` calls that you don't mind porting manually, and you don't want to use JSX, you can replace every call a factory function with a [`createElement`](/reference/react/createElement) call. For example, you can replace this code:

```js {1,3,6}
```

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

const button = createFactory('button');

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

with this code:

```js {1,4}
import { createElement } from 'react';

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

Here is a complete example of using React without JSX:

<Sandpack>

```js App.js
import { createElement } from 'react';

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

```

</Sandpack>

---

### Replacing `createFactory` with JSX {/*replacing-createfactory-with-jsx*/}

Finally, you can use JSX instead of `createFactory`. This is the most common way to use React:

<Sandpack>

```js App.js
export default function App() {
return (
<button onClick={() => {
alert('Clicked!');
}}>
Click me
</button>
);
};
```

</Sandpack>

<Pitfall>

Sometimes, your existing code might pass some variable as a `type` instead of a constant like `'button'`:

```js {3}
function Heading({ isSubheading, ...props }) {
const type = isSubheading ? 'h2' : 'h1';
const factory = createFactory(type);
return factory(props);
}
```

To do the same in JSX, you need to rename your variable to start with an uppercase letter like `Type`:

```js {2,3}
function Heading({ isSubheading, ...props }) {
const Type = isSubheading ? 'h2' : 'h1';
```
```

return <Type {...props} />;

}
```

Otherwise React will interpret `<type>` as a built-in HTML tag because it is lowercase.

</Pitfall>

---

title: "Directives"

---

<Intro>

React uses two directives to provide instructions to [bundlers compatible with React Server Components](/learn/start-a-new-react-project#bleeding-edge-react-frameworks).

</Intro>

---

## Source code directives {/*source-code-directives*/}

* [`'use client'`](/reference/react/use-client) marks source files whose components execute on the client.

* [`'use server'`](/reference/react/use-server) marks server-side functions that can be called from client-side code.

---

title: createContext

---

<Intro>

`createContext` lets you create a [context](/learn/passing-data-deeply-with-context) that components can provide or read.

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

</Intro>

<InlineToc />

Reference {/*reference*/}

`createContext(defaultValue)` {/*createcontext*/}
```

Call `createContext` outside of any components to create a context.

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

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

[See more examples below.](#usage)

Parameters {/*parameters*/}

* `defaultValue`: The value that you want the context to have when there is no matching context provider in the tree above the component that reads context. If you don't have any meaningful default value, specify `null`. The default value is meant as a "last resort" fallback. It is static and never changes over time.

Returns {/*returns*/}

`createContext` returns a context object.

The context object itself does not hold any information. It represents _which_ context other components read or provide. Typically, you will use [`SomeContext.Provider`](#provider) in components above to specify the context value, and call [`useContext(SomeContext)`](/reference/react/useContext) in components below to read it. The context object has a few properties:

* `SomeContext.Provider` lets you provide the context value to components.
* `SomeContext.Consumer` is an alternative and rarely used way to read the context value.

`SomeContext.Provider` {/*provider*/}

Wrap your components into a context provider to specify the value of this context for all components inside:

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

```

#### Props {/*provider-props*/}

* `value`: The value that you want to pass to all the components reading this context inside this provider, no matter how deep. The context value can be of any type. A component calling [`useContext(SomeContext)`](/reference/react/useContext) inside of the provider receives the `value` of the innermost corresponding context provider above it.

---

### `SomeContext.Consumer` {/*consumer*/}

Before `useContext` existed, there was an older way to read context:

```js
function Button() {
// ■ Legacy way (not recommended)
return (
<ThemeContext.Consumer>
{theme => (
<button className={theme} />
)}
</ThemeContext.Consumer>
);
}
```

Although this older way still works, but **newly written code should read context with [`useContext()`](/reference/react/useContext) instead:**

```js
function Button() {
// ■ Recommended way
const theme = useContext(ThemeContext);
return <button className={theme} />;
}
```

Props {/*consumer-props*/}

* `children`: A function. React will call the function you pass with the current context value determined by the same algorithm as [`useContext()`](/reference/react/useContext) does, and render the result you return from this function. React will also re-run this function and update the UI whenever the context
```

from the parent components changes.

Usage {/*usage*/}

Creating context {/*creating-context*/}

Context lets components [pass information deep down](/learn/passing-data-deeply-with-context) without explicitly passing props.

Call `createContext` outside any components to create one or more contexts.

```js [[1, 3, "ThemeContext"], [1, 4, "AuthContext"], [3, 3, "'light'"], [3, 4, "null"]]
import { createContext } from 'react';

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

`createContext` returns a <CodeStep step={1}>context object</CodeStep>. Components can read context by passing it to [`useContext()`](/reference/react/useContext):

```js [[1, 2, "ThemeContext"], [1, 7, "AuthContext"]]
function Button() {
const theme = useContext(ThemeContext);
// ...
}

function Profile() {
const currentUser = useContext(AuthContext);
// ...
}
```

By default, the values they receive will be the <CodeStep step={3}>default values</CodeStep> you have specified when creating the contexts. However, by itself this isn't useful because the default values never change.

Context is useful because you can **provide other, dynamic values from your components:**

```js {8-9,11-12}
function App() {
const [theme, setTheme] = useState('dark');
const [currentUser, setCurrentUser] = useState({ name: 'Taylor' });
```

```
// ...

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

Now the `Page` component and any components inside it, no matter how deep, will "see" the passed context values. If the passed context values change, React will re-render the components reading the context as well.

[Read more about reading and providing context and see examples.](/reference/react/useContext)

Importing and exporting context from a file {/*importing-and-exporting-context-from-a-file*/}

Often, components in different files will need access to the same context. This is why it's common to declare contexts in a separate file. Then you can use the [`export` statement](https://developer.mozilla.org/en-US/docs/web/javascript/reference/statements/export) to make context available for other files:

```js {4-5}
// Contexts.js
import { createContext } from 'react';

export const ThemeContext = createContext('light');
export const AuthContext = createContext(null);
```

Components declared in other files can then use the [`import`](https://developer.mozilla.org/en-US/docs/web/javascript/reference/statements/import) statement to read or provide this context:

```js {2}
// Button.js
import { ThemeContext } from './Contexts.js';

function Button() {
const theme = useContext(ThemeContext);
```

```
// ...
}
```

```js {2}
// App.js
import { ThemeContext, AuthContext } from './Contexts.js';

function App() {
// ...
return (
<ThemeContext.Provider value={theme}>
<AuthContext.Provider value={currentUser}>
<Page />
</AuthContext.Provider>
</ThemeContext.Provider>
);
}
```

This works similar to [importing and exporting components.](/learn/importing-and-exporting-components)

Troubleshooting {/*troubleshooting*/}

I can't find a way to change the context value {/*i-cant-find-a-way-to-change-the-context-value*/}

Code like this specifies the *default* context value:

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

This value never changes. React only uses this value as a fallback if it can't find a matching provider above.

To make context change over time, [add state and wrap components in a context provider.](/reference/react/useContext#updating-data-passed-via-context)

---

title: startTransition

<Intro>

`startTransition` lets you update the state without blocking the UI.

```js
startTransition(scope)
```

</Intro>

<InlineToc />

Reference {/*reference*/}

`startTransition(scope)` {/*starttransitionscope*/}

The `startTransition` function lets you mark a state update as a transition.

```js {7,9}
import { startTransition } from 'react';

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

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

[See more examples below.](#usage)

Parameters {/*parameters*/}

* `scope`: A function that updates some state by calling one or more [`set` functions.](/reference/react/useState#setstate) React immediately calls `scope` with no parameters and marks all state updates scheduled synchronously during the `scope` function call as transitions. They will be [non-blocking](/reference/react/useTransition#marking-a-state-update-as-a-non-blocking-transition) and [will not display unwanted loading

indicators.](/reference/react/useTransition#preventing-unwanted-loading-indicators)

Returns {/*returns*/}

`startTransition` does not return anything.

Caveats {/*caveats*/}

* `startTransition` does not provide a way to track whether a transition is pending. To show a pending indicator while the transition is ongoing, you need [`useTransition`](/reference/react/useTransition) instead.

* You can wrap an update into a transition only if you have access to the `set` function of that state. If you want to start a transition in response to some prop or a custom Hook return value, try [`useDeferredValue`](/reference/react/useDeferredValue) instead.

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

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

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

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

Usage {/*usage*/}

Marking a state update as a non-blocking transition
{/*marking-a-state-update-as-a-non-blocking-transition*/}

You can mark a state update as a *transition* by wrapping it in a `startTransition` call:

```js {7,9}
import { startTransition } from 'react';

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

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

// ...

}
```

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

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

<Note>

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

[Learn about transitions and see examples on the `useTransition` page.](/reference/react/useTransition)

</Note>

---

title: useSyncExternalStore

---

<Intro>

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

```js
const snapshot = useSyncExternalStore(subscribe, getSnapshot, getServerSnapshot?)
```

</Intro>

<InlineToc />

Reference {/*reference*/}

`useSyncExternalStore(subscribe, getSnapshot, getServerSnapshot?)` {/*usesyncexternalstore*/}

Call `useSyncExternalStore` at the top level of your component to read a value from an external data store.

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

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

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

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

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

[See more examples below.](#usage)

Parameters {/*parameters*/}

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

* `getSnapshot`: A function that returns a snapshot of the data in the store that's needed by the component. While the store has not changed, repeated calls to `getSnapshot` must return the same value. If the store changes and the returned value is different (as compared by [`Object.is`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/is)), React re-renders the component.

* **optional** `getServerSnapshot`: A function that returns the initial snapshot of the data in the store. It will be used only during server rendering and during hydration of server-rendered content on the client. The server snapshot must be the same between the client and the server, and is usually serialized and passed from the server to the client. If you omit this argument, rendering the component on the server will throw an error.

Returns {/*returns*/}

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

Caveats {/*caveats*/}

* The store snapshot returned by `getSnapshot` must be immutable. If the underlying store has mutable data, return a new immutable snapshot if the data has changed. Otherwise, return a cached last snapshot.

* If a different `subscribe` function is passed during a re-render, React will re-subscribe to the store using the newly passed `subscribe` function. You can prevent this by declaring `subscribe` outside the component.

Usage {/*usage*/}

Subscribing to an external store {/*subscribing-to-an-external-store*/}

Most of your React components will only read data from their [props,](/learn/passing-props-to-a-component) [state,](/reference/react/useState) and [context.](/reference/react/useContext) However, sometimes a component needs to read some data from some store outside of React that changes over time. This includes:

* Third-party state management libraries that hold state outside of React.

* Browser APIs that expose a mutable value and events to subscribe to its changes.

Call `useSyncExternalStore` at the top level of your component to read a value from an external data store.

```js [[1, 5, "todosStore.subscribe"], [2, 5, "todosStore.getSnapshot"], [3, 5, "todos", 0]]

import { useSyncExternalStore } from 'react';

import { todosStore } from './todoStore.js';

function TodosApp() {

const todos = useSyncExternalStore(todosStore.subscribe, todosStore.getSnapshot);

// ...

}
```

It returns the <CodeStep step={3}>snapshot</CodeStep> of the data in the store. You need to pass two functions as arguments:

1. The <CodeStep step={1}>`subscribe` function</CodeStep> should subscribe to the store and return a function that unsubscribes.

2. The <CodeStep step={2}>`getSnapshot` function</CodeStep> should read a snapshot of the data from the store.

React will use these functions to keep your component subscribed to the store and re-render it on changes.

For example, in the sandbox below, `todosStore` is implemented as an external store that stores data outside of React. The `TodosApp` component connects to that external store with the `useSyncExternalStore` Hook.

<Sandpack>

```js

import { useSyncExternalStore } from 'react';

import { todosStore } from './todoStore.js';

export default function TodosApp() {

const todos = useSyncExternalStore(todosStore.subscribe, todosStore.getSnapshot);

return (
```

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

```js todoStore.js
// This is an example of a third-party store
// that you might need to integrate with React.

// If your app is fully built with React,
// we recommend using React state instead.

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

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

```
};

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

</Sandpack>

<Note>

When possible, we recommend using built-in React state with [`useState`](/reference/react/useState) and [`useReducer`](/reference/react/useReducer) instead. The `useSyncExternalStore` API is mostly useful if you need to integrate with existing non-React code.

</Note>

Subscribing to a browser API {/*subscribing-to-a-browser-api*/}

Another reason to add `useSyncExternalStore` is when you want to subscribe to some value exposed by the browser that changes over time. For example, suppose that you want your component to display whether the network connection is active. The browser exposes this information via a property called [`navigator.onLine`.](https://developer.mozilla.org/en-US/docs/Web/API/Navigator/onLine)

This value can change without React's knowledge, so you should read it with `useSyncExternalStore`.

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

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

To implement the `getSnapshot` function, read the current value from the browser API:

```js
function getSnapshot() {
return navigator.onLine;
}
```

Next, you need to implement the `subscribe` function. For example, when `navigator.onLine` changes, the browser fires the [`online`](https://developer.mozilla.org/en-US/docs/Web/API/Window/online_event) and [`offline`](https://developer.mozilla.org/en-US/docs/Web/API/Window/offline_event) events on the `window` object. You need to subscribe the `callback` argument to the corresponding events, and then return a function that cleans up the subscriptions:

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

Now React knows how to read the value from the external `navigator.onLine` API and how to subscribe to its changes. Disconnect your device from the network and notice that the component re-renders in response:

<Sandpack>

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

export default function ChatIndicator() {
const isOnline = useSyncExternalStore(subscribe, getSnapshot);
return <h1>{isOnline ? '■ Online' : '■ Disconnected'}</h1>;
}

function getSnapshot() {
return navigator.onLine;
}

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

```
};
}
```

</Sandpack>

Extracting the logic to a custom Hook {/*extracting-the-logic-to-a-custom-hook*/}

Usually you won't write `useSyncExternalStore` directly in your components. Instead, you'll typically call it from your own custom Hook. This lets you use the same external store from different components.

For example, this custom `useOnlineStatus` Hook tracks whether the network is online:

```js {3,6}
import { useSyncExternalStore } from 'react';

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

function getSnapshot() {
// ...
}

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

Now different components can call `useOnlineStatus` without repeating the underlying implementation:

<Sandpack>

```js
import { useOnlineStatus } from './useOnlineStatus.js';

function StatusBar() {
const isOnline = useOnlineStatus();
return <h1>{isOnline ? '■ Online' : '■ Disconnected'}</h1>;
}

function SaveButton() {
```

````
const isOnline = useOnlineStatus();

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

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

export default function App() {
return (
<>
<SaveButton />
<StatusBar />
</>
);
}
```

```js useOnlineStatus.js
import { useSyncExternalStore } from 'react';

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

function getSnapshot() {
return navigator.onLine;
}

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

window.removeEventListener('offline', callback);

};

}

```

</Sandpack>

---

### Adding support for server rendering {/*adding-support-for-server-rendering*/}

If your React app uses [server rendering,](/reference/react-dom/server) your React components will also run outside the browser environment to generate the initial HTML. This creates a few challenges when connecting to an external store:

- If you're connecting to a browser-only API, it won't work because it does not exist on the server.

- If you're connecting to a third-party data store, you'll need its data to match between the server and client.

To solve these issues, pass a `getServerSnapshot` function as the third argument to `useSyncExternalStore`:

```js {4,12-14}

import { useSyncExternalStore } from 'react';

export function useOnlineStatus() {

const isOnline = useSyncExternalStore(subscribe, getSnapshot, getServerSnapshot);

return isOnline;

}

function getSnapshot() {

return navigator.onLine;

}

function getServerSnapshot() {

return true; // Always show "Online" for server-generated HTML

}

function subscribe(callback) {

// ...

}

```

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

- It runs on the server when generating the HTML.

- It runs on the client during [hydration](/reference/react-dom/client/hydrateRoot), i.e. when React takes the server HTML and makes it interactive.

This lets you provide the initial snapshot value which will be used before the app becomes interactive. If there is no meaningful initial value for the server rendering, omit this argument to [force rendering on the client.](/reference/react/Suspense#providing-a-fallback-for-server-errors-and-server-only-content)

<Note>

Make sure that `getServerSnapshot` returns the same exact data on the initial client render as it returned on the server. For example, if `getServerSnapshot` returned some prepopulated store content on the server, you need to transfer this content to the client. One way to do this is to emit a `<script>` tag during server rendering that sets a global like `window.MY_STORE_DATA`, and read from that global on the client in `getServerSnapshot`. Your external store should provide instructions on how to do that.

</Note>

Troubleshooting {/*troubleshooting*/}

I'm getting an error: "The result of `getSnapshot` should be cached" {/*im-getting-an-error-the-result-of-getsnapshot-should-be-cached*/}

This error means your `getSnapshot` function returns a new object every time it's called, for example:

```js {2-5}
function getSnapshot() {
// ■ Do not return always different objects from getSnapshot
return {
todos: myStore.todos
};
}
```

React will re-render the component if `getSnapshot` return value is different from the last time. This is why, if you always return a different value, you will enter an infinite loop and get this error.

Your `getSnapshot` object should only return a different object if something has actually changed. If your store contains immutable data, you can return that data directly:

```js {2-3}
function getSnapshot() {
// ■ You can return immutable data
return myStore.todos;
```

}
```

If your store data is mutable, your `getSnapshot` function should return an immutable snapshot of it. This means it *does* need to create new objects, but it shouldn't do this for every single call. Instead, it should store the last calculated snapshot, and return the same snapshot as the last time if the data in the store has not changed. How you determine whether mutable data has changed depends on your mutable store.

---

### My `subscribe` function gets called after every re-render
{/*my-subscribe-function-gets-called-after-every-re-render*/}

This `subscribe` function is defined *inside* a component so it is different on every re-render:

```js {4-7}
function ChatIndicator() {
const isOnline = useSyncExternalStore(subscribe, getSnapshot);

// ■ Always a different function, so React will resubscribe on every re-render
function subscribe() {
// ...
}

// ...
}
```

React will resubscribe to your store if you pass a different `subscribe` function between re-renders. If this causes performance issues and you'd like to avoid resubscribing, move the `subscribe` function outside:

```js {6-9}
function ChatIndicator() {
const isOnline = useSyncExternalStore(subscribe, getSnapshot);
// ...
}

// ■ Always the same function, so React won't need to resubscribe
function subscribe() {
// ...
}
```
```

Alternatively, wrap `subscribe` into [`useCallback`](/reference/react/useCallback) to only resubscribe when some argument changes:

```js {4-8}
function ChatIndicator({ userId }) {
const isOnline = useSyncExternalStore(subscribe, getSnapshot);

// ■ Same function as long as userId doesn't change
const subscribe = useCallback(() => {
// ...
}, [userId]);

// ...
}
```

---

title: useLayoutEffect

<Pitfall>

`useLayoutEffect` can hurt performance. Prefer [`useEffect`](/reference/react/useEffect) when possible.

</Pitfall>

<Intro>

`useLayoutEffect` is a version of [`useEffect`](/reference/react/useEffect) that fires before the browser repaints the screen.

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

</Intro>

<InlineToc />

Reference {/*reference*/}

`useLayoutEffect(setup, dependencies?)` {/*useinsertioneffect*/}

Call `useLayoutEffect` to perform the layout measurements before the browser repaints the screen:

```js
```

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

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

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

[See more examples below.](#usage)

Parameters {/*parameters*/}

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

* **optional** `dependencies`: The list of all reactive values referenced inside of the `setup` code. Reactive values include props, state, and all the variables and functions declared directly inside your component body. If your linter is [configured for React](/learn/editor-setup#linting), it will verify that every reactive value is correctly specified as a dependency. The list of dependencies must have a constant number of items and be written inline like `[dep1, dep2, dep3]`. React will compare each dependency with its previous value using the [`Object.is`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/is) comparison. If you omit this argument, your Effect will re-run after every re-render of the component.

Returns {/*returns*/}

`useLayoutEffect` returns `undefined`.

Caveats {/*caveats*/}

* `useLayoutEffect` is a Hook, so you can only call it **at the top level of your component** or your own Hooks. You can't call it inside loops or conditions. If you need that, extract a component and move the Effect there.

* When Strict Mode is on, React will **run one extra development-only setup+cleanup cycle** before the first real setup. This is a stress-test that ensures that your cleanup logic "mirrors" your setup logic and that it stops or undoes whatever the setup is doing. If this causes a problem, [implement the cleanup function.](/learn/synchronizing-with-effects#how-to-handle-the-effect-firing-twice-in-development)

* If some of your dependencies are objects or functions defined inside the component, there is a risk that they will **cause the Effect to re-run more often than needed.** To fix this, remove unnecessary [object](/reference/react/useEffect#removing-unnecessary-object-dependencies) and [function](/reference/react/useEffect#removing-unnecessary-function-dependencies) dependencies. You can also [extract state updates](/reference/react/useEffect#updating-state-based-on-previous-state-from-an-effect) and [non-reactive logic](/reference/react/useEffect#reading-the-latest-props-and-state-from-an-effect) outside of your Effect.

* Effects **only run on the client.** They don't run during server rendering.

* The code inside `useLayoutEffect` and all state updates scheduled from it **block the browser from repainting the screen.** When used excessively, this makes your app slow. When possible, prefer [`useEffect`.](/reference/react/useEffect)

Usage {/*usage*/}

Measuring layout before the browser repaints the screen {/*measuring-layout-before-the-browser-repaints-the-screen*/}

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

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

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

1. Render the tooltip anywhere (even with a wrong position).
2. Measure its height and decide where to place the tooltip.
3. Render the tooltip *again* in the correct place.

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

```js {5-8}
function Tooltip() {
const ref = useRef(null);
const [tooltipHeight, setTooltipHeight] = useState(0); // You don't know real height yet

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

}, []);

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

}

```

Here's how this works step by step:

1. `Tooltip` renders with the initial `tooltipHeight = 0` (so the tooltip may be wrongly positioned).

2. React places it in the DOM and runs the code in `useLayoutEffect`.

3. Your `useLayoutEffect` [measures the height](https://developer.mozilla.org/en-US/docs/Web/API/Element/getBoundingClientRect) of the tooltip content and triggers an immediate re-render.

4. `Tooltip` renders again with the real `tooltipHeight` (so the tooltip is correctly positioned).

5. React updates it in the DOM, and the browser finally displays the tooltip.

Hover over the buttons below and see how the tooltip adjusts its position depending on whether it fits:

<Sandpack>

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

export default function App() {

return (

<ButtonWithTooltip

tooltipContent={

This tooltip does not fit above the button.

<br />

This is why it's displayed below instead!

}

>

Hover over me (tooltip above)

</ButtonWithTooltip>

<div style={{ height: 50 }} />

<ButtonWithTooltip

tooltipContent={

This tooltip fits above the button

```
}
>
Hover over me (tooltip below)
</ButtonWithTooltip>
<div style={{ height: 50 }} />
<ButtonWithTooltip
tooltipContent={
<div>This tooltip fits above the button</div>
}
>
Hover over me (tooltip below)
</ButtonWithTooltip>
</div>
);
}
```

```js ButtonWithTooltip.js
import { useState, useRef } from 'react';
import Tooltip from './Tooltip.js';

export default function ButtonWithTooltip({ tooltipContent, ...rest }) {
const [targetRect, setTargetRect] = useState(null);
const buttonRef = useRef(null);
return (
<>
<button
{...rest}
ref={buttonRef}
onPointerEnter={() => {
const rect = buttonRef.current.getBoundingClientRect();
setTargetRect({
left: rect.left,
top: rect.top,
right: rect.right,
bottom: rect.bottom,
});
```

```
}}
onPointerLeave={() => {
setTargetRect(null);
}}
/>
{targetRect !== null && (
<Tooltip targetRect={targetRect}>
{tooltipContent}
</Tooltip>
)
}
</>
);
}
```

```js Tooltip.js active
import { useRef, useLayoutEffect, useState } from 'react';
import { createPortal } from 'react-dom';
import TooltipContainer from './TooltipContainer.js';

export default function Tooltip({ children, targetRect }) {
const ref = useRef(null);
const [tooltipHeight, setTooltipHeight] = useState(0);

useLayoutEffect(() => {
const { height } = ref.current.getBoundingClientRect();
setTooltipHeight(height);
console.log('Measured tooltip height: ' + height);
}, []);

let tooltipX = 0;
let tooltipY = 0;
if (targetRect !== null) {
tooltipX = targetRect.left;
tooltipY = targetRect.top - tooltipHeight;
if (tooltipY < 0) {
// It doesn't fit above, so place below.
```

tooltipY = targetRect.bottom;
}
}

return createPortal(
<TooltipContainer x={tooltipX} y={tooltipY} contentRef={ref}>
{children}
</TooltipContainer>,
document.body
);
}
```

```js TooltipContainer.js
export default function TooltipContainer({ children, x, y, contentRef }) {
return (
<div
style={{
position: 'absolute',
pointerEvents: 'none',
left: 0,
top: 0,
transform: `translate3d(${x}px, ${y}px, 0)`
}}
>
<div ref={contentRef} className="tooltip">
{children}
</div>
</div>
);
}
```

```css
.tooltip {
color: white;
background: #222;

border-radius: 4px;

padding: 4px;

}

```

</Sandpack>

Notice that even though the `Tooltip` component has to render in two passes (first, with `tooltipHeight` initialized to `0` and then with the real measured height), you only see the final result. This is why you need `useLayoutEffect` instead of [`useEffect`](/reference/react/useEffect) for this example. Let's look at the difference in detail below.

<Recipes titleText="useLayoutEffect vs useEffect" titleId="examples">

`useLayoutEffect` blocks the browser from repainting
{/*uselayouteffect-blocks-the-browser-from-repainting*/}

React guarantees that the code inside `useLayoutEffect` and any state updates scheduled inside it will be processed **before the browser repaints the screen.** This lets you render the tooltip, measure it, and re-render the tooltip again without the user noticing the first extra render. In other words, `useLayoutEffect` blocks the browser from painting.

<Sandpack>

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

export default function App() {

return (

<ButtonWithTooltip

tooltipContent={

This tooltip does not fit above the button.

<br />

This is why it's displayed below instead!

}

>

Hover over me (tooltip above)

</ButtonWithTooltip>

<div style={{ height: 50 }} />

<ButtonWithTooltip

````
tooltipContent={
<div>This tooltip fits above the button</div>
}
>
Hover over me (tooltip below)
</ButtonWithTooltip>
<div style={{ height: 50 }} />
<ButtonWithTooltip
tooltipContent={
<div>This tooltip fits above the button</div>
}
>
Hover over me (tooltip below)
</ButtonWithTooltip>
</div>
);
}
```

```js ButtonWithTooltip.js
import { useState, useRef } from 'react';
import Tooltip from './Tooltip.js';

export default function ButtonWithTooltip({ tooltipContent, ...rest }) {
const [targetRect, setTargetRect] = useState(null);
const buttonRef = useRef(null);
return (
<>
<button
{...rest}
ref={buttonRef}
onPointerEnter={() => {
const rect = buttonRef.current.getBoundingClientRect();
setTargetRect({
left: rect.left,
top: rect.top,
right: rect.right,
````

```
bottom: rect.bottom,
});
}}
onPointerLeave={() => {
setTargetRect(null);
}}
/>
{targetRect !== null && (
<Tooltip targetRect={targetRect}>
{tooltipContent}
</Tooltip>
)
}
</>
);
}
```

```js Tooltip.js active
import { useRef, useLayoutEffect, useState } from 'react';
import { createPortal } from 'react-dom';
import TooltipContainer from './TooltipContainer.js';

export default function Tooltip({ children, targetRect }) {
const ref = useRef(null);
const [tooltipHeight, setTooltipHeight] = useState(0);

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

let tooltipX = 0;
let tooltipY = 0;
if (targetRect !== null) {
tooltipX = targetRect.left;
tooltipY = targetRect.top - tooltipHeight;
if (tooltipY < 0) {
```

```
// It doesn't fit above, so place below.
tooltipY = targetRect.bottom;
}
}

return createPortal(
<TooltipContainer x={tooltipX} y={tooltipY} contentRef={ref}>
{children}
</TooltipContainer>,
document.body
);
}
```

```js TooltipContainer.js
export default function TooltipContainer({ children, x, y, contentRef }) {
return (
<div
style={{
position: 'absolute',
pointerEvents: 'none',
left: 0,
top: 0,
transform: `translate3d(${x}px, ${y}px, 0)`
}}
>
<div ref={contentRef} className="tooltip">
{children}
</div>
</div>
);
}
```

```css
.tooltip {
color: white;
```

background: #222;

border-radius: 4px;

padding: 4px;

}

```

</Sandpack>

<Solution />

#### `useEffect` does not block the browser {/*useeffect-does-not-block-the-browser*/}

Here is the same example, but with [`useEffect`](/reference/react/useEffect) instead of `useLayoutEffect`. If you're on a slower device, you might notice that sometimes the tooltip "flickers" and you briefly see its initial position before the corrected position.

<Sandpack>

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

export default function App() {

return (

<ButtonWithTooltip

tooltipContent={

This tooltip does not fit above the button.

<br />

This is why it's displayed below instead!

}

>

Hover over me (tooltip above)

</ButtonWithTooltip>

<div style={{ height: 50 }} />

<ButtonWithTooltip

tooltipContent={

This tooltip fits above the button

}

>

Hover over me (tooltip below)
</ButtonWithTooltip>
<div style={{ height: 50 }} />
<ButtonWithTooltip
tooltipContent={
This tooltip fits above the button
}
>
Hover over me (tooltip below)
</ButtonWithTooltip>

);
}
```

```js ButtonWithTooltip.js
import { useState, useRef } from 'react';
import Tooltip from './Tooltip.js';

export default function ButtonWithTooltip({ tooltipContent, ...rest }) {
const [targetRect, setTargetRect] = useState(null);
const buttonRef = useRef(null);
return (
<>
<button
{...rest}
ref={buttonRef}
onPointerEnter={() => {
const rect = buttonRef.current.getBoundingClientRect();
setTargetRect({
left: rect.left,
top: rect.top,
right: rect.right,
bottom: rect.bottom,
});
}}
onPointerLeave={() => {

````
setTargetRect(null);
}}
/>
{targetRect !== null && (
<Tooltip targetRect={targetRect}>
{tooltipContent}
</Tooltip>
)
}
</>
);
}
```

```js Tooltip.js active
import { useRef, useEffect, useState } from 'react';
import { createPortal } from 'react-dom';
import TooltipContainer from './TooltipContainer.js';

export default function Tooltip({ children, targetRect }) {
const ref = useRef(null);
const [tooltipHeight, setTooltipHeight] = useState(0);

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

let tooltipX = 0;
let tooltipY = 0;
if (targetRect !== null) {
tooltipX = targetRect.left;
tooltipY = targetRect.top - tooltipHeight;
if (tooltipY < 0) {
// It doesn't fit above, so place below.
tooltipY = targetRect.bottom;
}
}
````

return createPortal(
<TooltipContainer x={tooltipX} y={tooltipY} contentRef={ref}>
{children}
</TooltipContainer>,
document.body
);
}
```

```js TooltipContainer.js
export default function TooltipContainer({ children, x, y, contentRef }) {
return (
<div
style={{
position: 'absolute',
pointerEvents: 'none',
left: 0,
top: 0,
transform: `translate3d(${x}px, ${y}px, 0)`
}}
>
<div ref={contentRef} className="tooltip">
{children}
</div>
</div>
);
}
```

```css
.tooltip {
color: white;
background: #222;
border-radius: 4px;
padding: 4px;
}
```
```

</Sandpack>

To make the bug easier to reproduce, this version adds an artificial delay during rendering. React will let the browser paint the screen before it processes the state update inside `useEffect`. As a result, the tooltip flickers:

<Sandpack>

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

export default function App() {
return (
<div>
<ButtonWithTooltip
tooltipContent={
<div>
This tooltip does not fit above the button.
<br />
This is why it's displayed below instead!
</div>
}
>
Hover over me (tooltip above)
</ButtonWithTooltip>
<div style={{ height: 50 }} />
<ButtonWithTooltip
tooltipContent={
<div>This tooltip fits above the button</div>
}
>
Hover over me (tooltip below)
</ButtonWithTooltip>
<div style={{ height: 50 }} />
<ButtonWithTooltip
tooltipContent={
<div>This tooltip fits above the button</div>
}
>
```

Hover over me (tooltip below)
</ButtonWithTooltip>

);
}
```

```js ButtonWithTooltip.js
import { useState, useRef } from 'react';
import Tooltip from './Tooltip.js';

export default function ButtonWithTooltip({ tooltipContent, ...rest }) {
const [targetRect, setTargetRect] = useState(null);
const buttonRef = useRef(null);
return (
<>
<button
{...rest}
ref={buttonRef}
onPointerEnter={() => {
const rect = buttonRef.current.getBoundingClientRect();
setTargetRect({
left: rect.left,
top: rect.top,
right: rect.right,
bottom: rect.bottom,
});
}}
onPointerLeave={() => {
setTargetRect(null);
}}
/>
{targetRect !== null && (
<Tooltip targetRect={targetRect}>
{tooltipContent}
</Tooltip>
)

```
}
</>
);
}
```

```js Tooltip.js active
import { useRef, useEffect, useState } from 'react';
import { createPortal } from 'react-dom';
import TooltipContainer from './TooltipContainer.js';

export default function Tooltip({ children, targetRect }) {
const ref = useRef(null);
const [tooltipHeight, setTooltipHeight] = useState(0);

// This artificially slows down rendering
let now = performance.now();
while (performance.now() - now < 100) {
// Do nothing for a bit...
}

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

let tooltipX = 0;
let tooltipY = 0;
if (targetRect !== null) {
tooltipX = targetRect.left;
tooltipY = targetRect.top - tooltipHeight;
if (tooltipY < 0) {
// It doesn't fit above, so place below.
tooltipY = targetRect.bottom;
}
}

return createPortal(
<TooltipContainer x={tooltipX} y={tooltipY} contentRef={ref}>
```

{children}
</TooltipContainer>,
document.body
);
}
```

```js TooltipContainer.js
export default function TooltipContainer({ children, x, y, contentRef }) {
return (
<div
style={{
position: 'absolute',
pointerEvents: 'none',
left: 0,
top: 0,
transform: `translate3d(${x}px, ${y}px, 0)`
}}
>
<div ref={contentRef} className="tooltip">
{children}
</div>
</div>
);
}
```

```css
.tooltip {
color: white;
background: #222;
border-radius: 4px;
padding: 4px;
}
```

</Sandpack>
```

Edit this example to `useLayoutEffect` and observe that it blocks the paint even if rendering is slowed down.

<Solution />

</Recipes>

<Note>

Rendering in two passes and blocking the browser hurts performance. Try to avoid this when you can.

</Note>

Troubleshooting {/*troubleshooting*/}

I'm getting an error: "`useLayoutEffect` does nothing on the server" {/*im-getting-an-error-uselayouteffect-does-nothing-on-the-server*/}

The purpose of `useLayoutEffect` is to let your component [use layout information for rendering:](#measuring-layout-before-the-browser-repaints-the-screen)

1. Render the initial content.
2. Measure the layout *before the browser repaints the screen.*
3. Render the final content using the layout information you've read.

When you or your framework uses [server rendering](/reference/react-dom/server), your React app renders to HTML on the server for the initial render. This lets you show the initial HTML before the JavaScript code loads.

The problem is that on the server, there is no layout information.

In the [earlier example](#measuring-layout-before-the-browser-repaints-the-screen), the `useLayoutEffect` call in the `Tooltip` component lets it position itself correctly (either above or below content) depending on the content height. If you tried to render `Tooltip` as a part of the initial server HTML, this would be impossible to determine. On the server, there is no layout yet! So, even if you rendered it on the server, its position would "jump" on the client after the JavaScript loads and runs.

Usually, components that rely on layout information don't need to render on the server anyway. For example, it probably doesn't make sense to show a `Tooltip` during the initial render. It is triggered by a client interaction.

However, if you're running into this problem, you have a few different options:

- Replace `useLayoutEffect` with [`useEffect`.](/reference/react/useEffect) This tells React that it's okay to display the initial render result without blocking the paint (because the original HTML will become visible before your Effect runs).

- Alternatively, [mark your component as client-only.](/reference/react/Suspense#providing-a-fallback-for-server-errors-and-server-only-content) This tells React to replace its content up to the closest [`<Suspense>`](/reference/react/Suspense)

boundary with a loading fallback (for example, a spinner or a glimmer) during server rendering.

- Alternatively, you can render a component with `useLayoutEffect` only after hydration. Keep a boolean `isMounted` state that's initialized to `false`, and set it to `true` inside a `useEffect` call. Your rendering logic can then be like `return isMounted ? <RealContent /> : <FallbackContent />`. On the server and during the hydration, the user will see `FallbackContent` which should not call `useLayoutEffect`. Then React will replace it with `RealContent` which runs on the client only and can include `useLayoutEffect` calls.

- If you synchronize your component with an external data store and rely on `useLayoutEffect` for different reasons than measuring layout, consider [`useSyncExternalStore`](/reference/react/useSyncExternalStore) instead which [supports server rendering.](/reference/react/useSyncExternalStore#adding-support-for-server-rendering)

---

title: useState

<Intro>

`useState` is a React Hook that lets you add a [state variable](/learn/state-a-components-memory) to your component.

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

</Intro>

<InlineToc />

Reference {/*reference*/}

`useState(initialState)` {/*usestate*/}

Call `useState` at the top level of your component to declare a [state variable.](/learn/state-a-components-memory)

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

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

```

The convention is to name state variables like `[something, setSomething]` using [array destructuring.](https://javascript.info/destructuring-assignment)

[See more examples below.](#usage)

#### Parameters {/*parameters*/}

* `initialState`: The value you want the state to be initially. It can be a value of any type, but there is a special behavior for functions. This argument is ignored after the initial render.

* If you pass a function as `initialState`, it will be treated as an _initializer function_. It should be pure, should take no arguments, and should return a value of any type. React will call your initializer function when initializing the component, and store its return value as the initial state. [See an example below.](#avoiding-recreating-the-initial-state)

#### Returns {/*returns*/}

`useState` returns an array with exactly two values:

1. The current state. During the first render, it will match the `initialState` you have passed.

2. The [`set` function](#setstate) that lets you update the state to a different value and trigger a re-render.

#### Caveats {/*caveats*/}

* `useState` is a Hook, so you can only call it **at the top level of your component** or your own Hooks. You can't call it inside loops or conditions. If you need that, extract a new component and move the state into it.

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

---

### `set` functions, like `setSomething(nextState)` {/*setstate*/}

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

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

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

Parameters {/*setstate-parameters*/}

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

* If you pass a function as `nextState`, it will be treated as an _updater function_. It must be pure, should take the pending state as its only argument, and should return the next state. React will put your updater function in a queue and re-render your component. During the next render, React will calculate the next state by applying all of the queued updaters to the previous state. [See an example below.](#updating-state-based-on-the-previous-state)

Returns {/*setstate-returns*/}

`set` functions do not have a return value.

Caveats {/*setstate-caveats*/}

* The `set` function **only updates the state variable for the *next* render**. If you read the state variable after calling the `set` function, [you will still get the old value](#ive-updated-the-state-but-logging-gives-me-the-old-value) that was on the screen before your call.

* If the new value you provide is identical to the current `state`, as determined by an [`Object.is`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/is) comparison, React will **skip re-rendering the component and its children.** This is an optimization. Although in some cases React may still need to call your component before skipping the children, it shouldn't affect your code.

* React [batches state updates.](/learn/queueing-a-series-of-state-updates) It updates the screen **after all the event handlers have run** and have called their `set` functions. This prevents multiple re-renders during a single event. In the rare case that you need to force React to update the screen earlier, for example to access the DOM, you can use [`flushSync`.](/reference/react-dom/flushSync)

* Calling the `set` function *during rendering* is only allowed from within the currently rendering component. React will discard its output and immediately attempt to render it again with the new state. This pattern is rarely needed, but you can use it to **store information from the previous renders**. [See an example below.](#storing-information-from-previous-renders)

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

Usage {/*usage*/}

Adding state to a component {/*adding-state-to-a-component*/}

Call `useState` at the top level of your component to declare one or more [state variables.](/learn/state-a-components-memory)

```js [[1, 4, "age"], [2, 4, "setAge"], [3, 4, "42"], [1, 5, "name"], [2, 5, "setName"], [3, 5, "'Taylor'"]]
```

import { useState } from 'react';

function MyComponent() {

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

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

// ...

```

The convention is to name state variables like `[something, setSomething]` using [array destructuring.](https://javascript.info/destructuring-assignment)

`useState` returns an array with exactly two items:

1. The <CodeStep step={1}>current state</CodeStep> of this state variable, initially set to the <CodeStep step={3}>initial state</CodeStep> you provided.

2. The <CodeStep step={2}>`set` function</CodeStep> that lets you change it to any other value in response to interaction.

To update what's on the screen, call the `set` function with some next state:

```js [[2, 2, "setName"]]

function handleClick() {

setName('Robin');

}

```

React will store the next state, render your component again with the new values, and update the UI.

<Pitfall>

Calling the `set` function [**does not** change the current state in the already executing code](#ive-updated-the-state-but-logging-gives-me-the-old-value):

```js {3}

function handleClick() {

setName('Robin');

console.log(name); // Still "Taylor"!

}

```

It only affects what `useState` will return starting from the *next* render.

</Pitfall>

<Recipes titleText="Basic useState examples" titleId="examples-basic">
```

Counter (number) {/*counter-number*/}

In this example, the `count` state variable holds a number. Clicking the button increments it.

<Sandpack>

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

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

function handleClick() {
setCount(count + 1);
}

return (
<button onClick={handleClick}>
You pressed me {count} times
</button>
);
}
```

</Sandpack>

<Solution />

Text field (string) {/*text-field-string*/}

In this example, the `text` state variable holds a string. When you type, `handleChange` reads the latest input value from the browser input DOM element, and calls `setText` to update the state. This allows you to display the current `text` below.

<Sandpack>

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

export default function MyInput() {
const [text, setText] = useState('hello');

function handleChange(e) {
setText(e.target.value);
}
```

return (

<>

<input value={text} onChange={handleChange} />

You typed: {text}

<button onClick={() => setText('hello')}>

Reset

</button>

</>

);

}

```

</Sandpack>

<Solution />

#### Checkbox (boolean) {/*checkbox-boolean*/}

In this example, the `liked` state variable holds a boolean. When you click the input, `setLiked` updates the `liked` state variable with whether the browser checkbox input is checked. The `liked` variable is used to render the text below the checkbox.

<Sandpack>

```js

import { useState } from 'react';

export default function MyCheckbox() {

const [liked, setLiked] = useState(true);

function handleChange(e) {

setLiked(e.target.checked);

}

return (

<>

<label>

<input

type="checkbox"

checked={liked}

onChange={handleChange}

/>

I liked this

</label>

You {liked ? 'liked' : 'did not like'} this.

</>

);

}

```

</Sandpack>

<Solution />

Form (two variables) {/*form-two-variables*/}

You can declare more than one state variable in the same component. Each state variable is completely independent.

<Sandpack>

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

export default function Form() {

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

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

return (

<>

<input

value={name}

onChange={e => setName(e.target.value)}

/>

<button onClick={() => setAge(age + 1)}>

Increment age

</button>

<p>Hello, {name}. You are {age}.</p>

</>

);

}

```

```css

button { display: block; margin-top: 10px; }
```

</Sandpack>

<Solution />

</Recipes>

Updating state based on the previous state {/*updating-state-based-on-the-previous-state*/}

Suppose the `age` is `42`. This handler calls `setAge(age + 1)` three times:

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

However, after one click, `age` will only be `43` rather than `45`! This is because calling the `set` function [does not update](/learn/state-as-a-snapshot) the `age` state variable in the already running code. So each `setAge(age + 1)` call becomes `setAge(43)`.

To solve this problem, **you may pass an *updater function*** to `setAge` instead of the next state:

```js [[1, 2, "a", 0], [2, 2, "a + 1"], [1, 3, "a", 0], [2, 3, "a + 1"], [1, 4, "a", 0], [2, 4, "a + 1"]]
function handleClick() {
setAge(a => a + 1); // setAge(42 => 43)
setAge(a => a + 1); // setAge(43 => 44)
setAge(a => a + 1); // setAge(44 => 45)
}
```

Here, `a => a + 1` is your updater function. It takes the <CodeStep step={1}>pending state</CodeStep> and calculates the <CodeStep step={2}>next state</CodeStep> from it.

React puts your updater functions in a [queue.](/learn/queueing-a-series-of-state-updates) Then, during the next render, it will call them in the same order:

1. `a => a + 1` will receive `42` as the pending state and return `43` as the next state.
1. `a => a + 1` will receive `43` as the pending state and return `44` as the next state.
```

1. `a => a + 1` will receive `44` as the pending state and return `45` as the next state.

There are no other queued updates, so React will store `45` as the current state in the end.

By convention, it's common to name the pending state argument for the first letter of the state variable name, like `a` for `age`. However, you may also call it like `prevAge` or something else that you find clearer.

React may [call your updaters twice](#my-initializer-or-updater-function-runs-twice) in development to verify that they are [pure.](/learn/keeping-components-pure)

<DeepDive>

Is using an updater always preferred? {/*is-using-an-updater-always-preferred*/}

You might hear a recommendation to always write code like `setAge(a => a + 1)` if the state you're setting is calculated from the previous state. There is no harm in it, but it is also not always necessary.

In most cases, there is no difference between these two approaches. React always makes sure that for intentional user actions, like clicks, the `age` state variable would be updated before the next click. This means there is no risk of a click handler seeing a "stale" `age` at the beginning of the event handler.

However, if you do multiple updates within the same event, updaters can be helpful. They're also helpful if accessing the state variable itself is inconvenient (you might run into this when optimizing re-renders).

If you prefer consistency over slightly more verbose syntax, it's reasonable to always write an updater if the state you're setting is calculated from the previous state. If it's calculated from the previous state of some *other* state variable, you might want to combine them into one object and [use a reducer.](/learn/extracting-state-logic-into-a-reducer)

</DeepDive>

<Recipes titleText="The difference between passing an updater and passing the next state directly" titleId="examples-updater">

Passing the updater function {/*passing-the-updater-function*/}

This example passes the updater function, so the "+3" button works.

<Sandpack>

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

export default function Counter() {
const [age, setAge] = useState(42);

function increment() {
setAge(a => a + 1);
}
```

return (
<>
<h1>Your age: {age}</h1>
<button onClick={() => {
increment();
increment();
increment();
}}>+3</button>
<button onClick={() => {
increment();
}}>+1</button>
</>
);
}
```

```css
button { display: block; margin: 10px; font-size: 20px; }
h1 { display: block; margin: 10px; }
```

</Sandpack>

<Solution />

Passing the next state directly {/*passing-the-next-state-directly*/}

This example **does not** pass the updater function, so the "+3" button **doesn't work as intended**.

<Sandpack>

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

export default function Counter() {
const [age, setAge] = useState(42);

function increment() {
setAge(age + 1);
}

return (

<>

<h1>Your age: {age}</h1>

<button onClick={() => {

increment();

increment();

increment();

}}>+3</button>

<button onClick={() => {

increment();

}}>+1</button>

</>

);

}

```

```css

button { display: block; margin: 10px; font-size: 20px; }

h1 { display: block; margin: 10px; }

```

</Sandpack>

<Solution />

</Recipes>

Updating objects and arrays in state {/*updating-objects-and-arrays-in-state*/}

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

```js

// ■ Don't mutate an object in state like this:

form.firstName = 'Taylor';

```

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

```js

// ■ Replace state with a new object

````
setForm({
...form,
firstName: 'Taylor'
});
```

Read [updating objects in state](/learn/updating-objects-in-state) and [updating arrays in state](/learn/updating-arrays-in-state) to learn more.

<Recipes titleText="Examples of objects and arrays in state" titleId="examples-objects">

Form (object) {/*form-object*/}

In this example, the `form` state variable holds an object. Each input has a change handler that calls `setForm` with the next state of the entire form. The `{ ...form }` spread syntax ensures that the state object is replaced rather than mutated.

<Sandpack>

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

export default function Form() {
const [form, setForm] = useState({
firstName: 'Barbara',
lastName: 'Hepworth',
email: 'bhepworth@sculpture.com',
});

return (
<>
<label>
First name:
<input
value={form.firstName}
onChange={e => {
setForm({
...form,
firstName: e.target.value
});
}}
/>
````

```
</label>
<label>
Last name:
<input
value={form.lastName}
onChange={e => {
setForm({
...form,
lastName: e.target.value
});
}}
/>
</label>
<label>
Email:
<input
value={form.email}
onChange={e => {
setForm({
...form,
email: e.target.value
});
}}
/>
</label>
<p>
{form.firstName}{' '}
{form.lastName}{' '}
({form.email})
</p>
</>
);
}
```

```css
```

label { display: block; }

input { margin-left: 5px; }

```

</Sandpack>

<Solution />

#### Form (nested object) {/*form-nested-object*/}

In this example, the state is more nested. When you update nested state, you need to create a copy of the object you're updating, as well as any objects "containing" it on the way upwards. Read [updating a nested object](/learn/updating-objects-in-state#updating-a-nested-object) to learn more.

<Sandpack>

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

export default function Form() {
const [person, setPerson] = useState({
name: 'Niki de Saint Phalle',
artwork: {
title: 'Blue Nana',
city: 'Hamburg',
image: 'https://i.imgur.com/Sd1AgUOm.jpg',
}
});

function handleNameChange(e) {
setPerson({
...person,
name: e.target.value
});
}

function handleTitleChange(e) {
setPerson({
...person,
artwork: {
...person.artwork,
title: e.target.value

```
}
});
}

function handleCityChange(e) {
setPerson({
...person,
artwork: {
...person.artwork,
city: e.target.value
}
});
}

function handleImageChange(e) {
setPerson({
...person,
artwork: {
...person.artwork,
image: e.target.value
}
});
}

return (
<>
<label>
Name:
<input
value={person.name}
onChange={handleNameChange}
/>
</label>
<label>
Title:
<input
value={person.artwork.title}
```

```
onChange={handleTitleChange}
/>
</label>
<label>
City:
<input
value={person.artwork.city}
onChange={handleCityChange}
/>
</label>
<label>
Image:
<input
value={person.artwork.image}
onChange={handleImageChange}
/>
</label>
<p>
<i>{person.artwork.title}</i>
{' by '}
{person.name}
<br />
(located in {person.artwork.city})
</p>
<img
src={person.artwork.image}
alt={person.artwork.title}
/>
</>
);
}
```

```css
label { display: block; }
input { margin-left: 5px; margin-bottom: 5px; }
```

img { width: 200px; height: 200px; }
```

</Sandpack>

<Solution />

#### List (array) {/*list-array*/}

In this example, the `todos` state variable holds an array. Each button handler calls `setTodos` with the next version of that array. The `[...todos]` spread syntax, `todos.map()` and `todos.filter()` ensure the state array is replaced rather than mutated.

<Sandpack>

```js App.js
import { useState } from 'react';
import AddTodo from './AddTodo.js';
import TaskList from './TaskList.js';

let nextId = 3;
const initialTodos = [
{ id: 0, title: 'Buy milk', done: true },
{ id: 1, title: 'Eat tacos', done: false },
{ id: 2, title: 'Brew tea', done: false },
];

export default function TaskApp() {
const [todos, setTodos] = useState(initialTodos);

function handleAddTodo(title) {
setTodos([
...todos,
{
id: nextId++,
title: title,
done: false
}
]);
}

function handleChangeTodo(nextTodo) {
setTodos(todos.map(t => {

```
if (t.id === nextTodo.id) {
return nextTodo;
} else {
return t;
}
}));
}

function handleDeleteTodo(todoId) {
setTodos(
todos.filter(t => t.id !== todoId)
);
}

return (
<>
<AddTodo
onAddTodo={handleAddTodo}
/>
<TaskList
todos={todos}
onChangeTodo={handleChangeTodo}
onDeleteTodo={handleDeleteTodo}
/>
</>
);
}
```

```js AddTodo.js
import { useState } from 'react';

export default function AddTodo({ onAddTodo }) {
const [title, setTitle] = useState('');
return (
<>
<input
placeholder="Add todo"
```

```
value={title}
onChange={e => setTitle(e.target.value)}
/>
<button onClick={() => {
setTitle('');
onAddTodo(title);
}}>Add</button>
</>
)
}
```

```js TaskList.js
import { useState } from 'react';

export default function TaskList({
todos,
onChangeTodo,
onDeleteTodo
}) {
return (
<ul>
{todos.map(todo => (
<li key={todo.id}>
<Task
todo={todo}
onChange={onChangeTodo}
onDelete={onDeleteTodo}
/>
</li>
))}
</ul>
);
}

function Task({ todo, onChange, onDelete }) {
const [isEditing, setIsEditing] = useState(false);
```

```
let todoContent;
if (isEditing) {
todoContent = (
<>
<input
value={todo.title}
onChange={e => {
onChange({
...todo,
title: e.target.value
});
}} />
<button onClick={() => setIsEditing(false)}>
Save
</button>
</>
);
} else {
todoContent = (
<>
{todo.title}
<button onClick={() => setIsEditing(true)}>
Edit
</button>
</>
);
}
return (
<label>
<input
type="checkbox"
checked={todo.done}
onChange={e => {
onChange({
...todo,
```

done: e.target.checked

});

}}

/>

{todoContent}

<button onClick={() => onDelete(todo.id)}>

Delete

</button>

</label>

);

}

```

```css

button { margin: 5px; }

li { list-style-type: none; }

ul, li { margin: 0; padding: 0; }

```

</Sandpack>

<Solution />

Writing concise update logic with Immer {/*writing-concise-update-logic-with-immer*/}

If updating arrays and objects without mutation feels tedious, you can use a library like [Immer](https://github.com/immerjs/use-immer) to reduce repetitive code. Immer lets you write concise code as if you were mutating objects, but under the hood it performs immutable updates:

<Sandpack>

```js

import { useState } from 'react';

import { useImmer } from 'use-immer';

let nextId = 3;

const initialList = [

{ id: 0, title: 'Big Bellies', seen: false },

{ id: 1, title: 'Lunar Landscape', seen: false },

{ id: 2, title: 'Terracotta Army', seen: true },

];

```
export default function BucketList() {
const [list, updateList] = useImmer(initialList);

function handleToggle(artworkId, nextSeen) {
updateList(draft => {
const artwork = draft.find(a =>
a.id === artworkId
);
artwork.seen = nextSeen;
});
}

return (
<>
<h1>Art Bucket List</h1>
<h2>My list of art to see:</h2>
<ItemList
artworks={list}
onToggle={handleToggle} />
</>
);
}

function ItemList({ artworks, onToggle }) {
return (

{artworks.map(artwork => (
<li key={artwork.id}>
<label>
<input
type="checkbox"
checked={artwork.seen}
onChange={e => {
onToggle(
artwork.id,
e.target.checked
);
```

}}
/>
{artwork.title}
</label>

))}

);
}
```

```json package.json
{
"dependencies": {
"immer": "1.7.3",
"react": "latest",
"react-dom": "latest",
"react-scripts": "latest",
"use-immer": "0.5.1"
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
}
}
```

</Sandpack>

<Solution />

</Recipes>

Avoiding recreating the initial state {/*avoiding-recreating-the-initial-state*/}

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

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

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

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

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

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

React may [call your initializers twice](#my-initializer-or-updater-function-runs-twice) in development to verify that they are [pure.](/learn/keeping-components-pure)

<Recipes titleText="The difference between passing an initializer and passing the initial state directly" titleId="examples-initializer">

Passing the initializer function {/*passing-the-initializer-function*/}

This example passes the initializer function, so the `createInitialTodos` function only runs during initialization. It does not run when component re-renders, such as when you type into the input.

<Sandpack>

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

function createInitialTodos() {
const initialTodos = [];
for (let i = 0; i < 50; i++) {
initialTodos.push({
id: i,
text: 'Item ' + (i + 1)
});
```

```
}
return initialTodos;
}

export default function TodoList() {
const [todos, setTodos] = useState(createInitialTodos);
const [text, setText] = useState('');

return (
<>
<input
value={text}
onChange={e => setText(e.target.value)}
/>
<button onClick={() => {
setText('');
setTodos([{
id: todos.length,
text: text
}, ...todos]);
}}>Add</button>
<ul>
{todos.map(item => (
<li key={item.id}>
{item.text}
</li>
))}
</ul>
</>
);
}
```

</Sandpack>

<Solution />

Passing the initial state directly {/*passing-the-initial-state-directly*/}

This example **does not** pass the initializer function, so the `createInitialTodos` function runs on every render, such as when you type into the input. There is no observable difference in behavior, but this code is less efficient.

<Sandpack>

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

function createInitialTodos() {
const initialTodos = [];
for (let i = 0; i < 50; i++) {
initialTodos.push({
id: i,
text: 'Item ' + (i + 1)
});
}
return initialTodos;
}

export default function TodoList() {
const [todos, setTodos] = useState(createInitialTodos());
const [text, setText] = useState('');

return (
<>
<input
value={text}
onChange={e => setText(e.target.value)}
/>
<button onClick={() => {
setText('');
setTodos([{
id: todos.length,
text: text
}, ...todos]);
}}>Add</button>
<ul>
{todos.map(item => (
```

<li key={item.id}>

{item.text}

))}

</>

);

}

```

</Sandpack>

<Solution />

</Recipes>

---

### Resetting state with a key {/*resetting-state-with-a-key*/}

You'll often encounter the `key` attribute when [rendering lists.](/learn/rendering-lists) However, it also serves another purpose.

You can **reset a component's state by passing a different `key` to a component.** In this example, the Reset button changes the `version` state variable, which we pass as a `key` to the `Form`. When the `key` changes, React re-creates the `Form` component (and all of its children) from scratch, so its state gets reset.

Read [preserving and resetting state](/learn/preserving-and-resetting-state) to learn more.

<Sandpack>

```js App.js
import { useState } from 'react';

export default function App() {

const [version, setVersion] = useState(0);

function handleReset() {

setVersion(version + 1);

}

return (

<>

<button onClick={handleReset}>Reset</button>

<Form key={version} />

</>

);

}

function Form() {

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

return (

<>

<input

value={name}

onChange={e => setName(e.target.value)}

/>

Hello, {name}.

</>

);

}

```

```css

button { display: block; margin-bottom: 20px; }

```

</Sandpack>

Storing information from previous renders {/*storing-information-from-previous-renders*/}

Usually, you will update state in event handlers. However, in rare cases you might want to adjust state in response to rendering -- for example, you might want to change a state variable when a prop changes.

In most cases, you don't need this:

* **If the value you need can be computed entirely from the current props or other state, [remove that redundant state altogether.](/learn/choosing-the-state-structure#avoid-redundant-state)** If you're worried about recomputing too often, the [`useMemo` Hook](/reference/react/useMemo) can help.

* If you want to reset the entire component tree's state, [pass a different `key` to your component.](#resetting-state-with-a-key)

* If you can, update all the relevant state in the event handlers.
```

In the rare case that none of these apply, there is a pattern you can use to update state based on the values that have been rendered so far, by calling a `set` function while your component is rendering.

Here's an example. This `CountLabel` component displays the `count` prop passed to it:

```js CountLabel.js
export default function CountLabel({ count }) {
return <h1>{count}</h1>
}
```

Say you want to show whether the counter has *increased or decreased* since the last change. The `count` prop doesn't tell you this -- you need to keep track of its previous value. Add the `prevCount` state variable to track it. Add another state variable called `trend` to hold whether the count has increased or decreased. Compare `prevCount` with `count`, and if they're not equal, update both `prevCount` and `trend`. Now you can show both the current count prop and *how it has changed since the last render*.

<Sandpack>

```js App.js
import { useState } from 'react';
import CountLabel from './CountLabel.js';

export default function App() {
const [count, setCount] = useState(0);
return (
<>
<button onClick={() => setCount(count + 1)}>
Increment
</button>
<button onClick={() => setCount(count - 1)}>
Decrement
</button>
<CountLabel count={count} />
</>
);
}
```

```js CountLabel.js active
import { useState } from 'react';
```

```
export default function CountLabel({ count }) {
const [prevCount, setPrevCount] = useState(count);
const [trend, setTrend] = useState(null);
if (prevCount !== count) {
setPrevCount(count);
setTrend(count > prevCount ? 'increasing' : 'decreasing');
}
return (
<>
<h1>{count}</h1>
{trend && <p>The count is {trend}</p>}
</>
);
}
```

```css
button { margin-bottom: 10px; }
```

</Sandpack>

Note that if you call a `set` function while rendering, it must be inside a condition like `prevCount !== count`, and there must be a call like `setPrevCount(count)` inside of the condition. Otherwise, your component would re-render in a loop until it crashes. Also, you can only update the state of the *currently rendering* component like this. Calling the `set` function of *another* component during rendering is an error. Finally, your `set` call should still [update state without mutation](#updating-objects-and-arrays-in-state) -- this doesn't mean you can break other rules of [pure functions.](/learn/keeping-components-pure)

This pattern can be hard to understand and is usually best avoided. However, it's better than updating state in an effect. When you call the `set` function during render, React will re-render that component immediately after your component exits with a `return` statement, and before rendering the children. This way, children don't need to render twice. The rest of your component function will still execute (and the result will be thrown away). If your condition is below all the Hook calls, you may add an early `return;` to restart rendering earlier.

Troubleshooting {/*troubleshooting*/}

I've updated the state, but logging gives me the old value {/*ive-updated-the-state-but-logging-gives-me-the-old-value*/}

Calling the `set` function **does not change state in the running code**:

```js {4,5,8}
function handleClick() {
console.log(count); // 0

setCount(count + 1); // Request a re-render with 1
console.log(count); // Still 0!

setTimeout(() => {
console.log(count); // Also 0!
}, 5000);
}
```

This is because [states behaves like a snapshot.](/learn/state-as-a-snapshot) Updating state requests another render with the new state value, but does not affect the `count` JavaScript variable in your already-running event handler.

If you need to use the next state, you can save it in a variable before passing it to the `set` function:

```js
const nextCount = count + 1;
setCount(nextCount);

console.log(count); // 0
console.log(nextCount); // 1
```

I've updated the state, but the screen doesn't update
{/*ive-updated-the-state-but-the-screen-doesnt-update*/}

React will **ignore your update if the next state is equal to the previous state,** as determined by an [`Object.is`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/is) comparison. This usually happens when you change an object or an array in state directly:

```js
obj.x = 10; // ■ Wrong: mutating existing object
setObj(obj); // ■ Doesn't do anything
```

You mutated an existing `obj` object and passed it back to `setObj`, so React ignored the update. To fix this, you need to ensure that you're always [_replacing_ objects and arrays in state instead of _mutating_ them](#updating-objects-and-arrays-in-state):

```js
```

```
// ■ Correct: creating a new object
setObj({
...obj,
x: 10
});
```

I'm getting an error: "Too many re-renders" {/*im-getting-an-error-too-many-re-renders*/}

You might get an error that says: `Too many re-renders. React limits the number of renders to prevent an infinite loop.` Typically, this means that you're unconditionally setting state *during render*, so your component enters a loop: render, set state (which causes a render), render, set state (which causes a render), and so on. Very often, this is caused by a mistake in specifying an event handler:

```js {1-2}
// ■ Wrong: calls the handler during render
return <button onClick={handleClick()}>Click me</button>

// ■ Correct: passes down the event handler
return <button onClick={handleClick}>Click me</button>

// ■ Correct: passes down an inline function
return <button onClick={(e) => handleClick(e)}>Click me</button>
```

If you can't find the cause of this error, click on the arrow next to the error in the console and look through the JavaScript stack to find the specific `set` function call responsible for the error.

My initializer or updater function runs twice {/*my-initializer-or-updater-function-runs-twice*/}

In [Strict Mode](/reference/react/StrictMode), React will call some of your functions twice instead of once:

```js {2,5-6,11-12}
function TodoList() {
// This component function will run twice for every render.

const [todos, setTodos] = useState(() => {
// This initializer function will run twice during initialization.
return createTodos();
});
```

```
function handleClick() {
setTodos(prevTodos => {
// This updater function will run twice for every click.
return [...prevTodos, createTodo()];
});
}
// ...
```

This is expected and shouldn't break your code.

This **development-only** behavior helps you [keep components pure.](/learn/keeping-components-pure) React uses the result of one of the calls, and ignores the result of the other call. As long as your component, initializer, and updater functions are pure, this shouldn't affect your logic. However, if they are accidentally impure, this helps you notice the mistakes.

For example, this impure updater function mutates an array in state:

```js {2,3}
setTodos(prevTodos => {
// ■ Mistake: mutating state
prevTodos.push(createTodo());
});
```

Because React calls your updater function twice, you'll see the todo was added twice, so you'll know that there is a mistake. In this example, you can fix the mistake by [replacing the array instead of mutating it](#updating-objects-and-arrays-in-state):

```js {2,3}
setTodos(prevTodos => {
// ■ Correct: replacing with new state
return [...prevTodos, createTodo()];
});
```

Now that this updater function is pure, calling it an extra time doesn't make a difference in behavior. This is why React calling it twice helps you find mistakes. **Only component, initializer, and updater functions need to be pure.** Event handlers don't need to be pure, so React will never call your event handlers twice.

Read [keeping components pure](/learn/keeping-components-pure) to learn more.

I'm trying to set state to a function, but it gets called instead
{/*im-trying-to-set-state-to-a-function-but-it-gets-called-instead*/}

You can't put a function into state like this:

```js
const [fn, setFn] = useState(someFunction);

function handleClick() {
setFn(someOtherFunction);
}
```

Because you're passing a function, React assumes that `someFunction` is an [initializer function](#avoiding-recreating-the-initial-state), and that `someOtherFunction` is an [updater function](#updating-state-based-on-the-previous-state), so it tries to call them and store the result. To actually *store* a function, you have to put `() =>` before them in both cases. Then React will store the functions you pass.

```js {1,4}
const [fn, setFn] = useState(() => someFunction);

function handleClick() {
setFn(() => someOtherFunction);
}
```

---

title: "Legacy React APIs"

<Intro>

These APIs are exported from the `react` package, but they are not recommended for use in newly written code. See the linked individual API pages for the suggested alternatives.

</Intro>

Legacy APIs {/*legacy-apis*/}

* [`Children`](/reference/react/Children) lets you manipulate and transform the JSX received as the `children` prop. [See alternatives.](/reference/react/Children#alternatives)

* [`cloneElement`](/reference/react/cloneElement) lets you create a React element using another element as a starting point. [See alternatives.](/reference/react/cloneElement#alternatives)

* [`Component`](/reference/react/Component) lets you define a React component as a JavaScript class. [See alternatives.](/reference/react/Component#alternatives)

* [`createElement`](/reference/react/createElement) lets you create a React element. Typically, you'll use JSX instead.

* [`createRef`](/reference/react/createRef) creates a ref object which can contain arbitrary value. [See alternatives.](/reference/react/createRef#alternatives)

* [`isValidElement`](/reference/react/isValidElement) checks whether a value is a React element. Typically used with [`cloneElement`.](/reference/react/cloneElement)

* [`PureComponent`](/reference/react/PureComponent) is similar to [`Component`,](/reference/react/Component) but it skip re-renders with same props. [See alternatives.](/reference/react/PureComponent#alternatives)

Deprecated APIs {/*deprecated-apis*/}

<Deprecated>

These APIs will be removed in a future major version of React.

</Deprecated>

* [`createFactory`](/reference/react/createFactory) lets you create a function that produces React elements of a certain type.

---

title: "'use client'"

<Note>

These directives are needed only if you're [using React Server Components](/learn/start-a-new-react-project#bleeding-edge-react-frameworks) or building a library compatible with them.

</Note>

<Intro>

`'use client'` marks source files whose components execute on the client.

</Intro>

<InlineToc />

Reference {/*reference*/}

`'use client'` {/*use-client*/}

Add `'use client';` at the very top of a file to mark that the file (including any child components it uses) executes on the client, regardless of where it's imported.

```js
'use client';

import { useState } from 'react';

export default function RichTextEditor(props) {
// ...
```

When a file marked `'use client'` is imported from a server component, [compatible bundlers](/learn/start-a-new-react-project#bleeding-edge-react-frameworks) will treat the import as the "cut-off point" between server-only code and client code. Components at or below this point in the module graph can use client-only React features like [`useState`](/reference/react/useState).

Caveats {/*caveats*/}

* It's not necessary to add `'use client'` to every file that uses client-only React features, only the files that are imported from server component files. `'use client'` denotes the _boundary_ between server-only and client code; any components further down the tree will automatically be executed on the client. In order to be rendered from server components, components exported from `'use client'` files must have serializable props.

* When a `'use client'` file is imported from a server file, the imported values can be rendered as a React component or passed via props to a client component. Any other use will throw an exception.

* When a `'use client'` file is imported from another client file, the directive has no effect. This allows you to write client-only components that are simultaneously usable from server and client components.

* All the code in `'use client'` file as well as any modules it imports (directly or indirectly) will become a part of the client module graph and must be sent to and executed by the client in order to be rendered by the browser. To reduce client bundle size and take full advantage of the server, move state (and the `'use client'` directives) lower in the tree when possible, and pass rendered server components [as children](/learn/passing-props-to-a-component#passing-jsx-as-children) to client components.

* Because props are serialized across the server–client boundary, note that the placement of these directives can affect the amount of data sent to the client; avoid data structures that are larger than necessary.

* Components like a `<MarkdownRenderer>` that use neither server-only nor client-only features should generally not be marked with `'use client'`. That way, they can render exclusively on the server when used from a server component, but they'll be added to the client bundle when used from a client component.

* Libraries published to npm should include `'use client'` on exported React components that can be rendered with serializable props that use client-only React features, to allow those components to be imported and rendered by server components. Otherwise, users will need to wrap library components in their own `'use client'` files which can be cumbersome and prevents the library from moving logic to the server later. When publishing prebundled files to npm, ensure that `'use client'` source files end up in a bundle marked with `'use client'`, separate from any bundle containing exports that can be used

directly on the server.

* Client components will still run as part of server-side rendering (SSR) or build-time static site generation (SSG), which act as clients to transform React components' initial render output to HTML that can be rendered before JavaScript bundles are downloaded. But they can't use server-only features like reading directly from a database.

* Directives like `'use client'` must be at the very beginning of a file, above any imports or other code (comments above directives are OK). They must be written with single or double quotes, not backticks. (The `'use xyz'` directive format somewhat resembles the `useXyz()` Hook naming convention, but the similarity is coincidental.)

Usage {/*usage*/}

<Wip>

This section is incomplete. See also the [Next.js documentation for Server Components](https://beta.nextjs.org/docs/rendering/server-and-client-components).

</Wip>

---

title: Children

<Pitfall>

Using `Children` is uncommon and can lead to fragile code. [See common alternatives.](#alternatives)

</Pitfall>

<Intro>

`Children` lets you manipulate and transform the JSX you received as the [`children` prop.](/learn/passing-props-to-a-component#passing-jsx-as-children)

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

</Intro>

<InlineToc />

Reference {/*reference*/}

`Children.count(children)` {/*children-count*/}

Call `Children.count(children)` to count the number of children in the `children` data structure.

```js RowList.js active
import { Children } from 'react';

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

[See more examples below.](#counting-children)

Parameters {/*children-count-parameters*/}

* `children`: The value of the [`children` prop](/learn/passing-props-to-a-component#passing-jsx-as-children) received by your component.

Returns {/*children-count-returns*/}

The number of nodes inside these `children`.

Caveats {/*children-count-caveats*/}

- Empty nodes (`null`, `undefined`, and Booleans), strings, numbers, and [React elements](/reference/react/createElement) count as individual nodes. Arrays don't count as individual nodes, but their children do. **The traversal does not go deeper than React elements:** they don't get rendered, and their children aren't traversed. [Fragments](/reference/react/Fragment) don't get traversed.

`Children.forEach(children, fn, thisArg?)` {/*children-foreach*/}

Call `Children.forEach(children, fn, thisArg?)` to run some code for each child in the `children` data structure.

```js RowList.js active
import { Children } from 'react';
```

function SeparatorList({ children }) {

const result = [];

Children.forEach(children, (child, index) => {

result.push(child);

result.push(<hr key={index} />);

});

// ...

```

[See more examples below.](#running-some-code-for-each-child)

#### Parameters {/*children-foreach-parameters*/}

* `children`: The value of the [`children` prop](/learn/passing-props-to-a-component#passing-jsx-as-children) received by your component.

* `fn`: The function you want to run for each child, similar to the [array `forEach` method](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach) callback. It will be called with the child as the first argument and its index as the second argument. The index starts at `0` and increments on each call.

* **optional** `thisArg`: The [`this` value](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this) with which the `fn` function should be called. If omitted, it's `undefined`.

#### Returns {/*children-foreach-returns*/}

`Children.forEach` returns `undefined`.

#### Caveats {/*children-foreach-caveats*/}

- Empty nodes (`null`, `undefined`, and Booleans), strings, numbers, and [React elements](/reference/react/createElement) count as individual nodes. Arrays don't count as individual nodes, but their children do. **The traversal does not go deeper than React elements:** they don't get rendered, and their children aren't traversed. [Fragments](/reference/react/Fragment) don't get traversed.

---

### `Children.map(children, fn, thisArg?)` {/*children-map*/}

Call `Children.map(children, fn, thisArg?)` to map or transform each child in the `children` data structure.

```js RowList.js active

import { Children } from 'react';

function RowList({ children }) {

return (

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

[See more examples below.](#transforming-children)

Parameters {/*children-map-parameters*/}

* `children`: The value of the [`children` prop](/learn/passing-props-to-a-component#passing-jsx-as-children) received by your component.
* `fn`: The mapping function, similar to the [array `map` method](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map) callback. It will be called with the child as the first argument and its index as the second argument. The index starts at `0` and increments on each call. You need to return a React node from this function. This may be an empty node (`null`, `undefined`, or a Boolean), a string, a number, a React element, or an array of other React nodes.
* **optional** `thisArg`: The [`this` value](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this) with which the `fn` function should be called. If omitted, it's `undefined`.

Returns {/*children-map-returns*/}

If `children` is `null` or `undefined`, returns the same value.

Otherwise, returns a flat array consisting of the nodes you've returned from the `fn` function. The returned array will contain all nodes you returned except for `null` and `undefined`.

Caveats {/*children-map-caveats*/}

- Empty nodes (`null`, `undefined`, and Booleans), strings, numbers, and [React elements](/reference/react/createElement) count as individual nodes. Arrays don't count as individual nodes, but their children do. **The traversal does not go deeper than React elements:** they don't get rendered, and their children aren't traversed. [Fragments](/reference/react/Fragment) don't get traversed.

- If you return an element or an array of elements with keys from `fn`, **the returned elements' keys will be automatically combined with the key of the corresponding original item from `children`.** When you return multiple elements from `fn` in an array, their keys only need to be unique locally amongst each other.

`Children.only(children)` {/*children-only*/}

Call `Children.only(children)` to assert that `children` represent a single React element.

```js
function Box({ children }) {
const element = Children.only(children);
// ...
```

Parameters {/*children-only-parameters*/}

* `children`: The value of the [`children` prop](/learn/passing-props-to-a-component#passing-jsx-as-children) received by your component.

Returns {/*children-only-returns*/}

If `children` [is a valid element,](/reference/react/isValidElement) returns that element.

Otherwise, throws an error.

Caveats {/*children-only-caveats*/}

- This method always **throws if you pass an array (such as the return value of `Children.map`) as `children`.** In other words, it enforces that `children` is a single React element, not that it's an array with a single element.

`Children.toArray(children)` {/*children-toarray*/}

Call `Children.toArray(children)` to create an array out of the `children` data structure.

```js ReversedList.js active
import { Children } from 'react';

export default function ReversedList({ children }) {
const result = Children.toArray(children);
result.reverse();
// ...
```

Parameters {/*children-toarray-parameters*/}

* `children`: The value of the [`children` prop](/learn/passing-props-to-a-component#passing-jsx-as-children) received by your component.

Returns {/*children-toarray-returns*/}

Returns a flat array of elements in `children`.

Caveats {/*children-toarray-caveats*/}

- Empty nodes (`null`, `undefined`, and Booleans) will be omitted in the returned array. **The returned elements' keys will be calculated from the original elements' keys and their level of nesting and position.** This ensures that flattening the array does not introduce changes in behavior.

Usage {/*usage*/}

Transforming children {/*transforming-children*/}

To transform the children JSX that your component [receives as the `children` prop,](/learn/passing-props-to-a-component#passing-jsx-as-children) call `Children.map`:

```js {6,10}
import { Children } from 'react';

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

In the example above, the `RowList` wraps every child it receives into a `<div className="Row">` container. For example, let's say the parent component passes three `<p>` tags as the `children` prop to `RowList`:

```js
<RowList>
<p>This is the first item.</p>
<p>This is the second item.</p>
<p>This is the third item.</p>
</RowList>
```

Then, with the `RowList` implementation above, the final rendered result will look like this:

```js
<div className="RowList">
<div className="Row">
<p>This is the first item.</p>
</div>
<div className="Row">
<p>This is the second item.</p>
</div>
<div className="Row">
<p>This is the third item.</p>
</div>
</div>
```

`Children.map` is similar to [to transforming arrays with `map()`.](/learn/rendering-lists) The difference is that the `children` data structure is considered *opaque.* This means that even if it's sometimes an array, you should not assume it's an array or any other particular data type. This is why you should use `Children.map` if you need to transform it.

<Sandpack>

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

export default function App() {
return (
<RowList>
<p>This is the first item.</p>
<p>This is the second item.</p>
<p>This is the third item.</p>
</RowList>
);
}
```

```js RowList.js active
import { Children } from 'react';

export default function RowList({ children }) {
```

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

```css
.RowList {
display: flex;
flex-direction: column;
border: 2px solid grey;
padding: 5px;
}

.Row {
border: 2px dashed black;
padding: 5px;
margin: 5px;
}
```

</Sandpack>

<DeepDive>

Why is the children prop not always an array? {/*why-is-the-children-prop-not-always-an-array*/}

In React, the `children` prop is considered an *opaque* data structure. This means that you shouldn't rely on how it is structured. To transform, filter, or count children, you should use the `Children` methods.

In practice, the `children` data structure is often represented as an array internally. However, if there is only a single child, then React won't create an extra array since this would lead to unnecessary memory overhead. As long as you use the `Children` methods instead of directly introspecting the `children` prop, your code will not break even if React changes how the data structure is actually implemented.

Even when `children` is an array, `Children.map` has useful special behavior. For example, `Children.map` combines the [keys](/learn/rendering-lists#keeping-list-items-in-order-with-key) on the returned elements with the keys on the `children` you've passed to it. This ensures the original JSX children don't "lose" keys even if they get wrapped like in the example above.

</DeepDive>

<Pitfall>

The `children` data structure **does not include rendered output** of the components you pass as JSX. In the example below, the `children` received by the `RowList` only contains two items rather than three:

1. `<p>This is the first item.</p>`
2. `<MoreRows />`

This is why only two row wrappers are generated in this example:

<Sandpack>

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

export default function App() {
return (
<RowList>
<p>This is the first item.</p>
<MoreRows />
</RowList>
);
}

function MoreRows() {
return (
<>
<p>This is the second item.</p>
<p>This is the third item.</p>
</>
);
}
```

```js RowList.js
import { Children } from 'react';
```

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

```css
.RowList {
display: flex;
flex-direction: column;
border: 2px solid grey;
padding: 5px;
}

.Row {
border: 2px dashed black;
padding: 5px;
margin: 5px;
}
```

</Sandpack>

There is no way to get the rendered output of an inner component like `<MoreRows />` when manipulating `children`. This is why [it's usually better to use one of the alternative solutions.](#alternatives)

</Pitfall>

Running some code for each child {/*running-some-code-for-each-child*/}

Call `Children.forEach` to iterate over each child in the `children` data structure. It does not return any value and is similar to the [array `forEach` method.](https://developer.mozilla.org/en-US/docs/Web/Jav

aScript/Reference/Global_Objects/Array/forEach) You can use it to run custom logic like constructing your own array.

<Sandpack>

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

export default function App() {
return (
<SeparatorList>
<p>This is the first item.</p>
<p>This is the second item.</p>
<p>This is the third item.</p>
</SeparatorList>
);
}
```

```js SeparatorList.js active
import { Children } from 'react';

export default function SeparatorList({ children }) {
const result = [];
Children.forEach(children, (child, index) => {
result.push(child);
result.push(<hr key={index} />);
});
result.pop(); // Remove the last separator
return result;
}
```

</Sandpack>

<Pitfall>

As mentioned earlier, there is no way to get the rendered output of an inner component when manipulating `children`. This is why [it's usually better to use one of the alternative solutions.](#alternatives)

</Pitfall>

Counting children {/*counting-children*/}

Call `Children.count(children)` to calculate the number of children.

<Sandpack>

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

export default function App() {
return (
<RowList>
<p>This is the first item.</p>
<p>This is the second item.</p>
<p>This is the third item.</p>
</RowList>
);
}
```

```js RowList.js active
import { Children } from 'react';

export default function RowList({ children }) {
return (
<div className="RowList">
<h1 className="RowListHeader">
Total rows: {Children.count(children)}
</h1>
{Children.map(children, child =>
<div className="Row">
{child}
</div>
)}
</div>
);
}
```

```css
.RowList {
display: flex;
flex-direction: column;
border: 2px solid grey;
padding: 5px;
}

.RowListHeader {
padding-top: 5px;
font-size: 25px;
font-weight: bold;
text-align: center;
}

.Row {
border: 2px dashed black;
padding: 5px;
margin: 5px;
}
```

</Sandpack>

<Pitfall>

As mentioned earlier, there is no way to get the rendered output of an inner component when manipulating `children`. This is why [it's usually better to use one of the alternative solutions.](#alternatives)

</Pitfall>

Converting children to an array {/*converting-children-to-an-array*/}

Call `Children.toArray(children)` to turn the `children` data structure into a regular JavaScript array. This lets you manipulate the array with built-in array methods like [`filter`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter), [`sort`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/sort), or [`reverse`.](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reverse)

<Sandpack>

```js
```

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

export default function App() {
return (
<ReversedList>
<p>This is the first item.</p>
<p>This is the second item.</p>
<p>This is the third item.</p>
</ReversedList>
);
}
```

```js ReversedList.js active
import { Children } from 'react';

export default function ReversedList({ children }) {
const result = Children.toArray(children);
result.reverse();
return result;
}
```

</Sandpack>

<Pitfall>

As mentioned earlier, there is no way to get the rendered output of an inner component when manipulating `children`. This is why [it's usually better to use one of the alternative solutions.](#alternatives)

</Pitfall>

---

## Alternatives {/*alternatives*/}

<Note>

This section describes alternatives to the `Children` API (with capital `C`) that's imported like this:

```js
import { Children } from 'react';
```
````
````

Don't confuse it with [using the `children` prop](/learn/passing-props-to-a-component#passing-jsx-as-children) (lowercase `c`), which is good and encouraged.

</Note>

Exposing multiple components {/*exposing-multiple-components*/}

Manipulating children with the `Children` methods often leads to fragile code. When you pass children to a component in JSX, you don't usually expect the component to manipulate or transform the individual children.

When you can, try to avoid using the `Children` methods. For example, if you want every child of `RowList` to be wrapped in `<div className="Row">`, export a `Row` component, and manually wrap every row into it like this:

<Sandpack>

```js
import { RowList, Row } from './RowList.js';

export default function App() {
return (
<RowList>
<Row>
<p>This is the first item.</p>
</Row>
<Row>
<p>This is the second item.</p>
</Row>
<Row>
<p>This is the third item.</p>
</Row>
</RowList>
);
}
```

```js RowList.js
export function RowList({ children }) {
return (
<div className="RowList">
{children}
```

```
</div>
);
}

export function Row({ children }) {
return (
<div className="Row">
{children}
</div>
);
}
```

```css
.RowList {
display: flex;
flex-direction: column;
border: 2px solid grey;
padding: 5px;
}

.Row {
border: 2px dashed black;
padding: 5px;
margin: 5px;
}
```

</Sandpack>

Unlike using `Children.map`, this approach does not wrap every child automatically. **However, this approach has a significant benefit compared to the [earlier example with `Children.map`](#transforming-children) because it works even if you keep extracting more components.** For example, it still works if you extract your own `MoreRows` component:

<Sandpack>

```js
import { RowList, Row } from './RowList.js';

export default function App() {
return (
```

```
<RowList>
<Row>
<p>This is the first item.</p>
</Row>
<MoreRows />
</RowList>
);
}

function MoreRows() {
return (
<>
<Row>
<p>This is the second item.</p>
</Row>
<Row>
<p>This is the third item.</p>
</Row>
</>
);
}
```

```js RowList.js
export function RowList({ children }) {
return (
<div className="RowList">
{children}
</div>
);
}

export function Row({ children }) {
return (
<div className="Row">
{children}
</div>
```

);
}
```

```css
.RowList {
display: flex;
flex-direction: column;
border: 2px solid grey;
padding: 5px;
}

.Row {
border: 2px dashed black;
padding: 5px;
margin: 5px;
}
```

</Sandpack>

This wouldn't work with `Children.map` because it would "see" `<MoreRows />` as a single child (and a single row).

Accepting an array of objects as a prop {/*accepting-an-array-of-objects-as-a-prop*/}

You can also explicitly pass an array as a prop. For example, this `RowList` accepts a `rows` array as a prop:

<Sandpack>

```js
import { RowList, Row } from './RowList.js';

export default function App() {
return (
<RowList rows={[
{ id: 'first', content: This is the first item. },
{ id: 'second', content: This is the second item. },
{ id: 'third', content: This is the third item. }
]} />

);
}
```

```js RowList.js
export function RowList({ rows }) {
return (
<div className="RowList">
{rows.map(row => (
<div className="Row" key={row.id}>
{row.content}
</div>
))}
</div>
);
}
```

```css
.RowList {
display: flex;
flex-direction: column;
border: 2px solid grey;
padding: 5px;
}

.Row {
border: 2px dashed black;
padding: 5px;
margin: 5px;
}
```

</Sandpack>

Since `rows` is a regular JavaScript array, the `RowList` component can use built-in array methods like [`map`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map) on it.
```

This pattern is especially useful when you want to be able to pass more information as structured data together with children. In the below example, the `TabSwitcher` component receives an array of objects as the `tabs` prop:

<Sandpack>

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

export default function App() {
return (
<TabSwitcher tabs={[
{
id: 'first',
header: 'First',
content: <p>This is the first item.</p>
},
{
id: 'second',
header: 'Second',
content: <p>This is the second item.</p>
},
{
id: 'third',
header: 'Third',
content: <p>This is the third item.</p>
}
]} />
);
}
```

```js TabSwitcher.js
import { useState } from 'react';

export default function TabSwitcher({ tabs }) {
const [selectedId, setSelectedId] = useState(tabs[0].id);
const selectedTab = tabs.find(tab => tab.id === selectedId);
return (
```

<>

{tabs.map(tab => (

<button

key={tab.id}

onClick={() => setSelectedId(tab.id)}

>

{tab.header}

</button>

))}

<hr />

<h3>{selectedTab.header}</h3>

{selectedTab.content}

</>

);

}

```

</Sandpack>

Unlike passing the children as JSX, this approach lets you associate some extra data like `header` with each item. Because you are working with the `tabs` directly, and it is an array, you do not need the `Children` methods.

---

### Calling a render prop to customize rendering {/*calling-a-render-prop-to-customize-rendering*/}

Instead of producing JSX for every single item, you can also pass a function that returns JSX, and call that function when necessary. In this example, the `App` component passes a `renderContent` function to the `TabSwitcher` component. The `TabSwitcher` component calls `renderContent` only for the selected tab:

<Sandpack>

```js

import TabSwitcher from './TabSwitcher.js';

export default function App() {

return (

<TabSwitcher

````
tabIds={['first', 'second', 'third']}
getHeader={tabId => {
return tabId[0].toUpperCase() + tabId.slice(1);
}}
renderContent={tabId => {
return <p>This is the {tabId} item.</p>;
}}
/>
);
}
```

```js TabSwitcher.js
import { useState } from 'react';

export default function TabSwitcher({ tabIds, getHeader, renderContent }) {
const [selectedId, setSelectedId] = useState(tabIds[0]);
return (
<>
{tabIds.map((tabId) => (
<button
key={tabId}
onClick={() => setSelectedId(tabId)}
>
{getHeader(tabId)}
</button>
))}
<hr />
<div key={selectedId}>
<h3>{getHeader(selectedId)}</h3>
{renderContent(selectedId)}
</div>
</>
);
}
```
````
````

</Sandpack>

A prop like `renderContent` is called a *render prop* because it is a prop that specifies how to render a piece of the user interface. However, there is nothing special about it: it is a regular prop which happens to be a function.

Render props are functions, so you can pass information to them. For example, this `RowList` component passes the `id` and the `index` of each row to the `renderRow` render prop, which uses `index` to highlight even rows:

<Sandpack>

```js
import { RowList, Row } from './RowList.js';

export default function App() {
return (
<RowList
rowIds={['first', 'second', 'third']}
renderRow={(id, index) => {
return (
<Row isHighlighted={index % 2 === 0}>
<p>This is the {id} item.</p>
</Row>
);
}}
/>
);
}
```

```js RowList.js
import { Fragment } from 'react';

export function RowList({ rowIds, renderRow }) {
return (
<div className="RowList">
<h1 className="RowListHeader">
Total rows: {rowIds.length}
</h1>
{rowIds.map((rowId, index) =>
```

```
<Fragment key={rowId}>
{renderRow(rowId, index)}
</Fragment>
)}
</div>
);
}

export function Row({ children, isHighlighted }) {
return (
<div className={[
'Row',
isHighlighted ? 'RowHighlighted' : ''
].join(' ')}>
{children}
</div>
);
}
```

```css
.RowList {
display: flex;
flex-direction: column;
border: 2px solid grey;
padding: 5px;
}

.RowListHeader {
padding-top: 5px;
font-size: 25px;
font-weight: bold;
text-align: center;
}

.Row {
border: 2px dashed black;
padding: 5px;
```

margin: 5px;

}

.RowHighlighted {

background: #ffa;

}

```

</Sandpack>

This is another example of how parent and child components can cooperate without manipulating the children.

---

## Troubleshooting {/*troubleshooting*/}

### I pass a custom component, but the `Children` methods don't show its render result {/*i-pass-a-custom-component-but-the-children-methods-dont-show-its-render-result*/}

Suppose you pass two children to `RowList` like this:

```js

<RowList>

<p>First item</p>

<MoreRows />

</RowList>

```

If you do `Children.count(children)` inside `RowList`, you will get `2`. Even if `MoreRows` renders 10 different items, or if it returns `null`, `Children.count(children)` will still be `2`. From the `RowList`'s perspective, it only "sees" the JSX it has received. It does not "see" the internals of the `MoreRows` component.

The limitation makes it hard to extract a component. This is why [alternatives](#alternatives) are preferred to using `Children`.

---

title: <Fragment> (<>...</>)

<Intro>

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

```js

<>

<OneChild />

<AnotherChild />

</>

```

</Intro>

<InlineToc />

Reference {/*reference*/}

`<Fragment>` {/*fragment*/}

Wrap elements in `<Fragment>` to group them together in situations where you need a single element. Grouping elements in `Fragment` has no effect on the resulting DOM; it is the same as if the elements were not grouped. The empty JSX tag `<></>` is shorthand for `<Fragment></Fragment>` in most cases.

Props {/*props*/}

- **optional** `key`: Fragments declared with the explicit `<Fragment>` syntax may have [keys.](/learn/rendering-lists#keeping-list-items-in-order-with-key)

Caveats {/*caveats*/}

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

- React does not [reset state](/learn/preserving-and-resetting-state) when you go from rendering `<><Child /></>` to `[<Child />]` or back, or when you go from rendering `<><Child /></>` to `<Child />` and back. This only works a single level deep: for example, going from `<><><Child /></></>` to `<Child />` resets the state. See the precise semantics [here.](https://gist.github.com/clemmy/b3ef00f9507909429d8aa0d3ee4f986b)

Usage {/*usage*/}

Returning multiple elements {/*returning-multiple-elements*/}

Use `Fragment`, or the equivalent `<>...</>` syntax, to group multiple elements together. You can use it to put multiple elements in any place where a single element can go. For example, a component can only return one element, but by using a Fragment you can group multiple elements together and then return them as a group:

```js {3,6}

function Post() {

return (

<>

<PostTitle />

<PostBody />

</>

);

}

```

Fragments are useful because grouping elements with a Fragment has no effect on layout or styles, unlike if you wrapped the elements in another container like a DOM element. If you inspect this example with the browser tools, you'll see that all `<h1>` and `<article>` DOM nodes appear as siblings without wrappers around them:

<Sandpack>

```js

export default function Blog() {

return (

<>

<Post title="An update" body="It's been a while since I posted..." />

<Post title="My new blog" body="I am starting a new blog!" />

</>

)

}

function Post({ title, body }) {

return (

<>

<PostTitle title={title} />

<PostBody body={body} />

</>

);

}

function PostTitle({ title }) {

return <h1>{title}</h1>

}

function PostBody({ body }) {

return (

<article>

{body}

</article>

);

}

```

</Sandpack>

<DeepDive>

How to write a Fragment without the special syntax? {/*how-to-write-a-fragment-without-the-special-syntax*/}

The example above is equivalent to importing `Fragment` from React:

```js {1,5,8}

import { Fragment } from 'react';

function Post() {

return (

<Fragment>

<PostTitle />

<PostBody />

</Fragment>

);

}

```

Usually you won't need this unless you need to [pass a `key` to your `Fragment`.](#rendering-a-list-of-fragments)

</DeepDive>

Assigning multiple elements to a variable {/*assigning-multiple-elements-to-a-variable*/}

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

```js

function CloseDialog() {

const buttons = (

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

Grouping elements with text {/*grouping-elements-with-text*/}

You can use `Fragment` to group text together with components:

```js
function DateRangePicker({ start, end }) {
return (
<>
From
<DatePicker date={start} />
to
<DatePicker date={end} />
</>
);
}
```

Rendering a list of Fragments {/*rendering-a-list-of-fragments*/}

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

```js {3,6}

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

You can inspect the DOM to verify that there are no wrapper elements around the Fragment children:

<Sandpack>

```js
import { Fragment } from 'react';

const posts = [
{ id: 1, title: 'An update', body: "It's been a while since I posted..." },
{ id: 2, title: 'My new blog', body: 'I am starting a new blog!' }
];

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

function PostTitle({ title }) {
return <h1>{title}</h1>
}

function PostBody({ body }) {
return (
<article>
<p>{body}</p>
</article>
```

);

}
```

</Sandpack>

---

title: useDebugValue

---

<Intro>

`useDebugValue` is a React Hook that lets you add a label to a custom Hook in [React DevTools.](/learn/react-developer-tools)

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

</Intro>

<InlineToc />

Reference {/*reference*/}

`useDebugValue(value, format?)` {/*usedebugvalue*/}

Call `useDebugValue` at the top level of your [custom Hook](/learn/reusing-logic-with-custom-hooks) to display a readable debug value:

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

function useOnlineStatus() {
// ...
useDebugValue(isOnline ? 'Online' : 'Offline');
// ...
}
```

[See more examples below.](#usage)

Parameters {/*parameters*/}

* `value`: The value you want to display in React DevTools. It can have any type.
```

* **optional** `format`: A formatting function. When the component is inspected, React DevTools will call the formatting function with the `value` as the argument, and then display the returned formatted value (which may have any type). If you don't specify the formatting function, the original `value` itself will be displayed.

Returns {/*returns*/}

`useDebugValue` does not return anything.

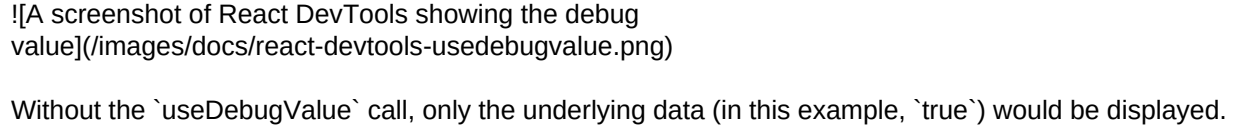
Usage {/*usage*/}

Adding a label to a custom Hook {/*adding-a-label-to-a-custom-hook*/}

Call `useDebugValue` at the top level of your [custom Hook](/learn/reusing-logic-with-custom-hooks) to display a readable <CodeStep step={1}>debug value</CodeStep> for [React DevTools.](/learn/react-developer-tools)

```js [[1, 5, "isOnline ? 'Online' : 'Offline'"]]
import { useDebugValue } from 'react';

function useOnlineStatus() {
// ...
useDebugValue(isOnline ? 'Online' : 'Offline');
// ...
}
```

This gives components calling `useOnlineStatus` a label like `OnlineStatus: "Online"` when you inspect them:



Without the `useDebugValue` call, only the underlying data (in this example, `true`) would be displayed.

<Sandpack>

```js
import { useOnlineStatus } from './useOnlineStatus.js';

function StatusBar() {
const isOnline = useOnlineStatus();
return <h1>{isOnline ? '■ Online' : '■ Disconnected'}</h1>;
}

export default function App() {
return <StatusBar />;
```

}
```

```js useOnlineStatus.js active
import { useSyncExternalStore, useDebugValue } from 'react';

export function useOnlineStatus() {
const isOnline = useSyncExternalStore(subscribe, () => navigator.onLine, () => true);
useDebugValue(isOnline ? 'Online' : 'Offline');
return isOnline;
}

function subscribe(callback) {
window.addEventListener('online', callback);
window.addEventListener('offline', callback);
return () => {
window.removeEventListener('online', callback);
window.removeEventListener('offline', callback);
};
}
```

</Sandpack>

<Note>

Don't add debug values to every custom Hook. It's most valuable for custom Hooks that are part of shared libraries and that have a complex internal data structure that's difficult to inspect.

</Note>

Deferring formatting of a debug value {/*deferring-formatting-of-a-debug-value*/}

You can also pass a formatting function as the second argument to `useDebugValue`:

```js [[1, 1, "date", 18], [2, 1, "date.toDateString()"]]
useDebugValue(date, date => date.toDateString());
```

Your formatting function will receive the <CodeStep step={1}>debug value</CodeStep> as a parameter and should return a <CodeStep step={2}>formatted display value</CodeStep>. When your component is inspected, React DevTools will call this function and display its result.
```

This lets you avoid running potentially expensive formatting logic unless the component is actually inspected. For example, if `date` is a Date value, this avoids calling `toDateString()` on it for every render.

---

title: <Suspense>

<Intro>

`<Suspense>` lets you display a fallback until its children have finished loading.

```js
<Suspense fallback={<Loading />}>
<SomeComponent />
</Suspense>
```

</Intro>

<InlineToc />

Reference {/*reference*/}

`<Suspense>` {/*suspense*/}

Props {/*props*/}

* `children`: The actual UI you intend to render. If `children` suspends while rendering, the Suspense boundary will switch to rendering `fallback`.

* `fallback`: An alternate UI to render in place of the actual UI if it has not finished loading. Any valid React node is accepted, though in practice, a fallback is a lightweight placeholder view, such as a loading spinner or skeleton. Suspense will automatically switch to `fallback` when `children` suspends, and back to `children` when the data is ready. If `fallback` suspends while rendering, it will activate the closest parent Suspense boundary.

Caveats {/*caveats*/}

- React does not preserve any state for renders that got suspended before they were able to mount for the first time. When the component has loaded, React will retry rendering the suspended tree from scratch.

- If Suspense was displaying content for the tree, but then it suspended again, the `fallback` will be shown again unless the update causing it was caused by [`startTransition`](/reference/react/startTransition) or [`useDeferredValue`](/reference/react/useDeferredValue).

- If React needs to hide the already visible content because it suspended again, it will clean up [layout Effects](/reference/react/useLayoutEffect) in the content tree. When the content is ready to be shown

again, React will fire the layout Effects again. This ensures that Effects measuring the DOM layout don't try to do this while the content is hidden.

- React includes under-the-hood optimizations like *Streaming Server Rendering* and *Selective Hydration* that are integrated with Suspense. Read [an architectural overview](https://github.com/reactwg/react-18/discussions/37) and watch [a technical talk](https://www.youtube.com/watch?v=pj5N-Khihgc) to learn more.

Usage {/*usage*/}

Displaying a fallback while content is loading {/*displaying-a-fallback-while-content-is-loading*/}

You can wrap any part of your application with a Suspense boundary:

```js [[1, 1, "<Loading />"], [2, 2, "<Albums />"]]
<Suspense fallback={<Loading />}>
<Albums />
</Suspense>
```

React will display your <CodeStep step={1}>loading fallback</CodeStep> until all the code and data needed by <CodeStep step={2}>the children</CodeStep> has been loaded.

In the example below, the `Albums` component *suspends* while fetching the list of albums. Until it's ready to render, React switches the closest Suspense boundary above to show the fallback--your `Loading` component. Then, when the data loads, React hides the `Loading` fallback and renders the `Albums` component with data.

<Sandpack>

```json package.json hidden
{
"dependencies": {
"react": "experimental",
"react-dom": "experimental"
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
}
}
```

```

```js App.js hidden
import { useState } from 'react';
import ArtistPage from './ArtistPage.js';

export default function App() {
const [show, setShow] = useState(false);
if (show) {
return (
<ArtistPage
artist={{
id: 'the-beatles',
name: 'The Beatles',
}}
/>
);
} else {
return (
<button onClick={() => setShow(true)}>
Open The Beatles artist page
</button>
);
}
}
```

```js ArtistPage.js active
import { Suspense } from 'react';
import Albums from './Albums.js';

export default function ArtistPage({ artist }) {
return (
<>
<h1>{artist.name}</h1>
<Suspense fallback={<Loading />}>
<Albums artistId={artist.id} />
</Suspense>
```
```

```
</>
);
}

function Loading() {
return <h2>■ Loading...</h2>;
}
```

```js Albums.js hidden
import { fetchData } from './data.js';

// Note: this component is written using an experimental API
// that's not yet available in stable versions of React.

// For a realistic example you can follow today, try a framework
// that's integrated with Suspense, like Relay or Next.js.

export default function Albums({ artistId }) {
const albums = use(fetchData(`/${artistId}/albums`));
return (
<ul>
{albums.map(album => (
<li key={album.id}>
{album.title} ({album.year})
</li>
))}
</ul>
);
}

// This is a workaround for a bug to get the demo running.
// TODO: replace with real implementation when the bug is fixed.
function use(promise) {
if (promise.status === 'fulfilled') {
return promise.value;
} else if (promise.status === 'rejected') {
throw promise.reason;
} else if (promise.status === 'pending') {
```

```
throw promise;
} else {
promise.status = 'pending';
promise.then(
result => {
promise.status = 'fulfilled';
promise.value = result;
},
reason => {
promise.status = 'rejected';
promise.reason = reason;
},
);
throw promise;
}
}
```

```js data.js hidden
// Note: the way you would do data fetching depends on
// the framework that you use together with Suspense.
// Normally, the caching logic would be inside a framework.

let cache = new Map();

export function fetchData(url) {
if (!cache.has(url)) {
cache.set(url, getData(url));
}
return cache.get(url);
}

async function getData(url) {
if (url === '/the-beatles/albums') {
return await getAlbums();
} else {
throw Error('Not implemented');
}
```

```
}

async function getAlbums() {
// Add a fake delay to make waiting noticeable.
await new Promise(resolve => {
setTimeout(resolve, 3000);
});

return [{
id: 13,
title: 'Let It Be',
year: 1970
}, {
id: 12,
title: 'Abbey Road',
year: 1969
}, {
id: 11,
title: 'Yellow Submarine',
year: 1969
}, {
id: 10,
title: 'The Beatles',
year: 1968
}, {
id: 9,
title: 'Magical Mystery Tour',
year: 1967
}, {
id: 8,
title: 'Sgt. Pepper\'s Lonely Hearts Club Band',
year: 1967
}, {
id: 7,
title: 'Revolver',
year: 1966
}, {
```

id: 6,

title: 'Rubber Soul',

year: 1965

}, {

id: 5,

title: 'Help!',

year: 1965

}, {

id: 4,

title: 'Beatles For Sale',

year: 1964

}, {

id: 3,

title: 'A Hard Day\'s Night',

year: 1964

}, {

id: 2,

title: 'With The Beatles',

year: 1963

}, {

id: 1,

title: 'Please Please Me',

year: 1963

}];

}

```

</Sandpack>

<Note>

**Only Suspense-enabled data sources will activate the Suspense component.** They include:

- Data fetching with Suspense-enabled frameworks like [Relay](https://relay.dev/docs/guided-tour/rendering/loading-states/) and [Next.js](https://nextjs.org/docs/getting-started/react-essentials)

- Lazy-loading component code with [`lazy`](/reference/react/lazy)

Suspense **does not** detect when data is fetched inside an Effect or event handler.
```

The exact way you would load data in the `Albums` component above depends on your framework. If you use a Suspense-enabled framework, you'll find the details in its data fetching documentation.

Suspense-enabled data fetching without the use of an opinionated framework is not yet supported. The requirements for implementing a Suspense-enabled data source are unstable and undocumented. An official API for integrating data sources with Suspense will be released in a future version of React.

</Note>

Revealing content together at once {/*revealing-content-together-at-once*/}

By default, the whole tree inside Suspense is treated as a single unit. For example, even if *only one* of these components suspends waiting for some data, *all* of them together will be replaced by the loading indicator:

```js {2-5}
<Suspense fallback={<Loading />}>
<Biography />
<Panel>
<Albums />
</Panel>
</Suspense>
```

Then, after all of them are ready to be displayed, they will all appear together at once.

In the example below, both `Biography` and `Albums` fetch some data. However, because they are grouped under a single Suspense boundary, these components always "pop in" together at the same time.

<Sandpack>

```json package.json hidden
{
"dependencies": {
"react": "experimental",
"react-dom": "experimental"
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
```

"eject": "react-scripts eject"
}
}
```

```js App.js hidden
import { useState } from 'react';
import ArtistPage from './ArtistPage.js';

export default function App() {
const [show, setShow] = useState(false);
if (show) {
return (
<ArtistPage
artist={{
id: 'the-beatles',
name: 'The Beatles',
}}
/>
);
} else {
return (
<button onClick={() => setShow(true)}>
Open The Beatles artist page
</button>
);
}
}
```

```js ArtistPage.js active
import { Suspense } from 'react';
import Albums from './Albums.js';
import Biography from './Biography.js';
import Panel from './Panel.js';

export default function ArtistPage({ artist }) {
return (

```
<>
<h1>{artist.name}</h1>
<Suspense fallback={<Loading />}>
<Biography artistId={artist.id} />
<Panel>
<Albums artistId={artist.id} />
</Panel>
</Suspense>
</>
);
}

function Loading() {
return <h2>■ Loading...</h2>;
}
```

```js Panel.js
export default function Panel({ children }) {
return (
<section className="panel">
{children}
</section>
);
}
```

```js Biography.js hidden
import { fetchData } from './data.js';

// Note: this component is written using an experimental API
// that's not yet available in stable versions of React.

// For a realistic example you can follow today, try a framework
// that's integrated with Suspense, like Relay or Next.js.

export default function Biography({ artistId }) {
const bio = use(fetchData(`/${artistId}/bio`));
return (
```

```
<section>
<p className="bio">{bio}</p>
</section>
);
}

// This is a workaround for a bug to get the demo running.
// TODO: replace with real implementation when the bug is fixed.
function use(promise) {
if (promise.status === 'fulfilled') {
return promise.value;
} else if (promise.status === 'rejected') {
throw promise.reason;
} else if (promise.status === 'pending') {
throw promise;
} else {
promise.status = 'pending';
promise.then(
result => {
promise.status = 'fulfilled';
promise.value = result;
},
reason => {
promise.status = 'rejected';
promise.reason = reason;
},
);
throw promise;
}
}
```

```js Albums.js hidden
import { fetchData } from './data.js';

// Note: this component is written using an experimental API
// that's not yet available in stable versions of React.
```

```
// For a realistic example you can follow today, try a framework
// that's integrated with Suspense, like Relay or Next.js.

export default function Albums({ artistId }) {
const albums = use(fetchData(`/${artistId}/albums`));
return (
<ul>
{albums.map(album => (
<li key={album.id}>
{album.title} ({album.year})
</li>
))}
</ul>
);
}

// This is a workaround for a bug to get the demo running.
// TODO: replace with real implementation when the bug is fixed.
function use(promise) {
if (promise.status === 'fulfilled') {
return promise.value;
} else if (promise.status === 'rejected') {
throw promise.reason;
} else if (promise.status === 'pending') {
throw promise;
} else {
promise.status = 'pending';
promise.then(
result => {
promise.status = 'fulfilled';
promise.value = result;
},
reason => {
promise.status = 'rejected';
promise.reason = reason;
},
);
```

````
throw promise;
}
}
```

```js data.js hidden
// Note: the way you would do data fetching depends on
// the framework that you use together with Suspense.
// Normally, the caching logic would be inside a framework.

let cache = new Map();

export function fetchData(url) {
if (!cache.has(url)) {
cache.set(url, getData(url));
}
return cache.get(url);
}

async function getData(url) {
if (url === '/the-beatles/albums') {
return await getAlbums();
} else if (url === '/the-beatles/bio') {
return await getBio();
} else {
throw Error('Not implemented');
}
}

async function getBio() {
// Add a fake delay to make waiting noticeable.
await new Promise(resolve => {
setTimeout(resolve, 1500);
});

return `The Beatles were an English rock band,
formed in Liverpool in 1960, that comprised
John Lennon, Paul McCartney, George Harrison
and Ringo Starr.`;
````

```
}

async function getAlbums() {
// Add a fake delay to make waiting noticeable.
await new Promise(resolve => {
setTimeout(resolve, 3000);
});

return [{
id: 13,
title: 'Let It Be',
year: 1970
}, {
id: 12,
title: 'Abbey Road',
year: 1969
}, {
id: 11,
title: 'Yellow Submarine',
year: 1969
}, {
id: 10,
title: 'The Beatles',
year: 1968
}, {
id: 9,
title: 'Magical Mystery Tour',
year: 1967
}, {
id: 8,
title: 'Sgt. Pepper\'s Lonely Hearts Club Band',
year: 1967
}, {
id: 7,
title: 'Revolver',
year: 1966
}, {
```

id: 6,
title: 'Rubber Soul',
year: 1965
}, {
id: 5,
title: 'Help!',
year: 1965
}, {
id: 4,
title: 'Beatles For Sale',
year: 1964
}, {
id: 3,
title: 'A Hard Day\'s Night',
year: 1964
}, {
id: 2,
title: 'With The Beatles',
year: 1963
}, {
id: 1,
title: 'Please Please Me',
year: 1963
}];
}
```

```css
.bio { font-style: italic; }

.panel {
border: 1px solid #aaa;
border-radius: 6px;
margin-top: 20px;
padding: 10px;
}
```
```

</Sandpack>

Components that load data don't have to be direct children of the Suspense boundary. For example, you can move `Biography` and `Albums` into a new `Details` component. This doesn't change the behavior. `Biography` and `Albums` share the same closest parent Suspense boundary, so their reveal is coordinated together.

```js {2,8-11}
<Suspense fallback={<Loading />}>
<Details artistId={artist.id} />
</Suspense>

function Details({ artistId }) {
return (
<>
<Biography artistId={artistId} />
<Panel>
<Albums artistId={artistId} />
</Panel>
</>
);
}
```

Revealing nested content as it loads {/*revealing-nested-content-as-it-loads*/}

When a component suspends, the closest parent Suspense component shows the fallback. This lets you nest multiple Suspense components to create a loading sequence. Each Suspense boundary's fallback will be filled in as the next level of content becomes available. For example, you can give the album list its own fallback:

```js {3,7}
<Suspense fallback={<BigSpinner />}>
<Biography />
<Suspense fallback={<AlbumsGlimmer />}>
<Panel>
<Albums />
</Panel>
</Suspense>
</Suspense>
```

```

With this change, displaying the `Biography` doesn't need to "wait" for the `Albums` to load.

The sequence will be:

1. If `Biography` hasn't loaded yet, `BigSpinner` is shown in place of the entire content area.
1. Once `Biography` finishes loading, `BigSpinner` is replaced by the content.
1. If `Albums` hasn't loaded yet, `AlbumsGlimmer` is shown in place of `Albums` and its parent `Panel`.
1. Finally, once `Albums` finishes loading, it replaces `AlbumsGlimmer`.

<Sandpack>

```json package.json hidden
{
"dependencies": {
"react": "experimental",
"react-dom": "experimental"
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
}
}
```

```js App.js hidden
import { useState } from 'react';
import ArtistPage from './ArtistPage.js';

export default function App() {
const [show, setShow] = useState(false);
if (show) {
return (
<ArtistPage
artist={{
id: 'the-beatles',
name: 'The Beatles',

```
}}
/>
);
} else {
return (
<button onClick={() => setShow(true)}>
Open The Beatles artist page
</button>
);
}
}
```

```js ArtistPage.js active
import { Suspense } from 'react';
import Albums from './Albums.js';
import Biography from './Biography.js';
import Panel from './Panel.js';

export default function ArtistPage({ artist }) {
return (
<>
<h1>{artist.name}</h1>
<Suspense fallback={<BigSpinner />}>
<Biography artistId={artist.id} />
<Suspense fallback={<AlbumsGlimmer />}>
<Panel>
<Albums artistId={artist.id} />
</Panel>
</Suspense>
</Suspense>
</>
);
}

function BigSpinner() {
return <h2>■ Loading...</h2>;
```

```
}

function AlbumsGlimmer() {
return (
<div className="glimmer-panel">
<div className="glimmer-line" />
<div className="glimmer-line" />
<div className="glimmer-line" />
</div>
);
}
```

```js Panel.js
export default function Panel({ children }) {
return (
<section className="panel">
{children}
</section>
);
}
```

```js Biography.js hidden
import { fetchData } from './data.js';

// Note: this component is written using an experimental API
// that's not yet available in stable versions of React.

// For a realistic example you can follow today, try a framework
// that's integrated with Suspense, like Relay or Next.js.

export default function Biography({ artistId }) {
const bio = use(fetchData(`/${artistId}/bio`));
return (
<section>
<p className="bio">{bio}</p>
</section>
);
```

```
}

// This is a workaround for a bug to get the demo running.
// TODO: replace with real implementation when the bug is fixed.
function use(promise) {
if (promise.status === 'fulfilled') {
return promise.value;
} else if (promise.status === 'rejected') {
throw promise.reason;
} else if (promise.status === 'pending') {
throw promise;
} else {
promise.status = 'pending';
promise.then(
result => {
promise.status = 'fulfilled';
promise.value = result;
},
reason => {
promise.status = 'rejected';
promise.reason = reason;
},
);
throw promise;
}
}
```

```js Albums.js hidden
import { fetchData } from './data.js';

// Note: this component is written using an experimental API
// that's not yet available in stable versions of React.

// For a realistic example you can follow today, try a framework
// that's integrated with Suspense, like Relay or Next.js.

export default function Albums({ artistId }) {
const albums = use(fetchData(`/${artistId}/albums`));
```

```
return (
<ul>
{albums.map(album => (
<li key={album.id}>
{album.title} ({album.year})
</li>
))}
</ul>
);
}

// This is a workaround for a bug to get the demo running.
// TODO: replace with real implementation when the bug is fixed.
function use(promise) {
if (promise.status === 'fulfilled') {
return promise.value;
} else if (promise.status === 'rejected') {
throw promise.reason;
} else if (promise.status === 'pending') {
throw promise;
} else {
promise.status = 'pending';
promise.then(
result => {
promise.status = 'fulfilled';
promise.value = result;
},
reason => {
promise.status = 'rejected';
promise.reason = reason;
},
);
throw promise;
}
}
```

```js data.js hidden
// Note: the way you would do data fetching depends on
// the framework that you use together with Suspense.
// Normally, the caching logic would be inside a framework.

let cache = new Map();

export function fetchData(url) {
if (!cache.has(url)) {
cache.set(url, getData(url));
}
return cache.get(url);
}

async function getData(url) {
if (url === '/the-beatles/albums') {
return await getAlbums();
} else if (url === '/the-beatles/bio') {
return await getBio();
} else {
throw Error('Not implemented');
}
}

async function getBio() {
// Add a fake delay to make waiting noticeable.
await new Promise(resolve => {
setTimeout(resolve, 500);
});

return `The Beatles were an English rock band,
formed in Liverpool in 1960, that comprised
John Lennon, Paul McCartney, George Harrison
and Ringo Starr.`;
}

async function getAlbums() {
// Add a fake delay to make waiting noticeable.
await new Promise(resolve => {
```

```
setTimeout(resolve, 3000);
});

return [{
id: 13,
title: 'Let It Be',
year: 1970
}, {
id: 12,
title: 'Abbey Road',
year: 1969
}, {
id: 11,
title: 'Yellow Submarine',
year: 1969
}, {
id: 10,
title: 'The Beatles',
year: 1968
}, {
id: 9,
title: 'Magical Mystery Tour',
year: 1967
}, {
id: 8,
title: 'Sgt. Pepper\'s Lonely Hearts Club Band',
year: 1967
}, {
id: 7,
title: 'Revolver',
year: 1966
}, {
id: 6,
title: 'Rubber Soul',
year: 1965
}, {
```

```
id: 5,
title: 'Help!',
year: 1965
}, {
id: 4,
title: 'Beatles For Sale',
year: 1964
}, {
id: 3,
title: 'A Hard Day\'s Night',
year: 1964
}, {
id: 2,
title: 'With The Beatles',
year: 1963
}, {
id: 1,
title: 'Please Please Me',
year: 1963
}];
}
```

```css
.bio { font-style: italic; }

.panel {
border: 1px solid #aaa;
border-radius: 6px;
margin-top: 20px;
padding: 10px;
}

.glimmer-panel {
border: 1px dashed #aaa;
background: linear-gradient(90deg, rgba(221,221,221,1) 0%, rgba(255,255,255,1) 100%);
border-radius: 6px;
```

margin-top: 20px;

padding: 10px;

}

.glimmer-line {

display: block;

width: 60%;

height: 20px;

margin: 10px;

border-radius: 4px;

background: #f0f0f0;

}

```

</Sandpack>

Suspense boundaries let you coordinate which parts of your UI should always "pop in" together at the same time, and which parts should progressively reveal more content in a sequence of loading states. You can add, move, or delete Suspense boundaries in any place in the tree without affecting the rest of your app's behavior.

Don't put a Suspense boundary around every component. Suspense boundaries should not be more granular than the loading sequence that you want the user to experience. If you work with a designer, ask them where the loading states should be placed--it's likely that they've already included them in their design wireframes.

---

### Showing stale content while fresh content is loading
{/*showing-stale-content-while-fresh-content-is-loading*/}

In this example, the `SearchResults` component suspends while fetching the search results. Type `"a"`, wait for the results, and then edit it to `"ab"`. The results for `"a"` will get replaced by the loading fallback.

<Sandpack>

```json package.json hidden

{

"dependencies": {

"react": "experimental",

"react-dom": "experimental"

},

"scripts": {

```
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
}
}
```

```js App.js
import { Suspense, useState } from 'react';
import SearchResults from './SearchResults.js';

export default function App() {
const [query, setQuery] = useState('');
return (
<>
<label>
Search albums:
<input value={query} onChange={e => setQuery(e.target.value)} />
</label>
<Suspense fallback={<h2>Loading...</h2>}>
<SearchResults query={query} />
</Suspense>
</>
);
}
```

```js SearchResults.js hidden
import { fetchData } from './data.js';

// Note: this component is written using an experimental API
// that's not yet available in stable versions of React.

// For a realistic example you can follow today, try a framework
// that's integrated with Suspense, like Relay or Next.js.

export default function SearchResults({ query }) {
if (query === '') {
```

```
return null;
}
const albums = use(fetchData(`/search?q=${query}`));
if (albums.length === 0) {
return <p>No matches for <i>"{query}"</i></p>;
}
return (
<ul>
{albums.map(album => (
<li key={album.id}>
{album.title} ({album.year})
</li>
))}
</ul>
);
}

// This is a workaround for a bug to get the demo running.
// TODO: replace with real implementation when the bug is fixed.
function use(promise) {
if (promise.status === 'fulfilled') {
return promise.value;
} else if (promise.status === 'rejected') {
throw promise.reason;
} else if (promise.status === 'pending') {
throw promise;
} else {
promise.status = 'pending';
promise.then(
result => {
promise.status = 'fulfilled';
promise.value = result;
},
reason => {
promise.status = 'rejected';
promise.reason = reason;
```

```
},
);
throw promise;
}
}
```

```js data.js hidden
// Note: the way you would do data fetching depends on
// the framework that you use together with Suspense.
// Normally, the caching logic would be inside a framework.

let cache = new Map();

export function fetchData(url) {
if (!cache.has(url)) {
cache.set(url, getData(url));
}
return cache.get(url);
}

async function getData(url) {
if (url.startsWith('/search?q=')) {
return await getSearchResults(url.slice('/search?q='.length));
} else {
throw Error('Not implemented');
}
}

async function getSearchResults(query) {
// Add a fake delay to make waiting noticeable.
await new Promise(resolve => {
setTimeout(resolve, 500);
});

const allAlbums = [{
id: 13,
title: 'Let It Be',
year: 1970
```

```
}, {
id: 12,
title: 'Abbey Road',
year: 1969
}, {
id: 11,
title: 'Yellow Submarine',
year: 1969
}, {
id: 10,
title: 'The Beatles',
year: 1968
}, {
id: 9,
title: 'Magical Mystery Tour',
year: 1967
}, {
id: 8,
title: 'Sgt. Pepper\'s Lonely Hearts Club Band',
year: 1967
}, {
id: 7,
title: 'Revolver',
year: 1966
}, {
id: 6,
title: 'Rubber Soul',
year: 1965
}, {
id: 5,
title: 'Help!',
year: 1965
}, {
id: 4,
title: 'Beatles For Sale',
```

year: 1964

}, {

id: 3,

title: 'A Hard Day\'s Night',

year: 1964

}, {

id: 2,

title: 'With The Beatles',

year: 1963

}, {

id: 1,

title: 'Please Please Me',

year: 1963

}];

const lowerQuery = query.trim().toLowerCase();

return allAlbums.filter(album => {

const lowerTitle = album.title.toLowerCase();

return (

lowerTitle.startsWith(lowerQuery) ||

lowerTitle.indexOf(' ' + lowerQuery) !== -1

)

});

}

```

```css

input { margin: 10px; }

```

</Sandpack>

A common alternative UI pattern is to *defer* updating the list and to keep showing the previous results until the new results are ready. The [`useDeferredValue`](/reference/react/useDeferredValue) Hook lets you pass a deferred version of the query down:

```js {3,11}

export default function App() {

const [query, setQuery] = useState('');

```
const deferredQuery = useDeferredValue(query);
return (
<>
<label>
Search albums:
<input value={query} onChange={e => setQuery(e.target.value)} />
</label>
<Suspense fallback={<h2>Loading...</h2>}>
<SearchResults query={deferredQuery} />
</Suspense>
</>
);
}
```

The `query` will update immediately, so the input will display the new value. However, the `deferredQuery` will keep its previous value until the data has loaded, so `SearchResults` will show the stale results for a bit.

To make it more obvious to the user, you can add a visual indication when the stale result list is displayed:

```js {2}
<div style={{
opacity: query !== deferredQuery ? 0.5 : 1
}}>
<SearchResults query={deferredQuery} />
</div>
```

Enter `"a"` in the example below, wait for the results to load, and then edit the input to `"ab"`. Notice how instead of the Suspense fallback, you now see the dimmed stale result list until the new results have loaded:

<Sandpack>

```json package.json hidden
{
"dependencies": {
"react": "experimental",
"react-dom": "experimental"
```

```
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
}
}
```

```js App.js
import { Suspense, useState, useDeferredValue } from 'react';
import SearchResults from './SearchResults.js';

export default function App() {
const [query, setQuery] = useState('');
const deferredQuery = useDeferredValue(query);
const isStale = query !== deferredQuery;
return (
<>
<label>
Search albums:
<input value={query} onChange={e => setQuery(e.target.value)} />
</label>
<Suspense fallback={<h2>Loading...</h2>}>
<div style={{ opacity: isStale ? 0.5 : 1 }}>
<SearchResults query={deferredQuery} />
</div>
</Suspense>
</>
);
}
```

```js SearchResults.js hidden
import { fetchData } from './data.js';

// Note: this component is written using an experimental API
```

```
// that's not yet available in stable versions of React.

// For a realistic example you can follow today, try a framework
// that's integrated with Suspense, like Relay or Next.js.

export default function SearchResults({ query }) {
if (query === '') {
return null;
}
const albums = use(fetchData(`/search?q=${query}`));
if (albums.length === 0) {
return <p>No matches for <i>"{query}"</i></p>;
}
return (
<ul>
{albums.map(album => (
<li key={album.id}>
{album.title} ({album.year})
</li>
))}
</ul>
);
}

// This is a workaround for a bug to get the demo running.
// TODO: replace with real implementation when the bug is fixed.
function use(promise) {
if (promise.status === 'fulfilled') {
return promise.value;
} else if (promise.status === 'rejected') {
throw promise.reason;
} else if (promise.status === 'pending') {
throw promise;
} else {
promise.status = 'pending';
promise.then(
result => {
```

```
promise.status = 'fulfilled';
promise.value = result;
},
reason => {
promise.status = 'rejected';
promise.reason = reason;
},
);
throw promise;
}
}
```

```js data.js hidden
// Note: the way you would do data fetching depends on
// the framework that you use together with Suspense.
// Normally, the caching logic would be inside a framework.

let cache = new Map();

export function fetchData(url) {
if (!cache.has(url)) {
cache.set(url, getData(url));
}
return cache.get(url);
}

async function getData(url) {
if (url.startsWith('/search?q=')) {
return await getSearchResults(url.slice('/search?q='.length));
} else {
throw Error('Not implemented');
}
}

async function getSearchResults(query) {
// Add a fake delay to make waiting noticeable.
await new Promise(resolve => {
setTimeout(resolve, 500);
```

```
});

const allAlbums = [{
id: 13,
title: 'Let It Be',
year: 1970
}, {
id: 12,
title: 'Abbey Road',
year: 1969
}, {
id: 11,
title: 'Yellow Submarine',
year: 1969
}, {
id: 10,
title: 'The Beatles',
year: 1968
}, {
id: 9,
title: 'Magical Mystery Tour',
year: 1967
}, {
id: 8,
title: 'Sgt. Pepper\'s Lonely Hearts Club Band',
year: 1967
}, {
id: 7,
title: 'Revolver',
year: 1966
}, {
id: 6,
title: 'Rubber Soul',
year: 1965
}, {
id: 5,
```

title: 'Help!',
year: 1965
}, {
id: 4,
title: 'Beatles For Sale',
year: 1964
}, {
id: 3,
title: 'A Hard Day\'s Night',
year: 1964
}, {
id: 2,
title: 'With The Beatles',
year: 1963
}, {
id: 1,
title: 'Please Please Me',
year: 1963
}];

const lowerQuery = query.trim().toLowerCase();
return allAlbums.filter(album => {
const lowerTitle = album.title.toLowerCase();
return (
lowerTitle.startsWith(lowerQuery) ||
lowerTitle.indexOf(' ' + lowerQuery) !== -1
)
});
}
```

```css
input { margin: 10px; }
```

</Sandpack>

<Note>
```

Both deferred values and [transitions](#preventing-already-revealed-content-from-hiding) let you avoid showing Suspense fallback in favor of inline indicators. Transitions mark the whole update as non-urgent so they are typically used by frameworks and router libraries for navigation. Deferred values, on the other hand, are mostly useful in application code where you want to mark a part of UI as non-urgent and let it "lag behind" the rest of the UI.

</Note>

Preventing already revealed content from hiding
{/*preventing-already-revealed-content-from-hiding*/}

When a component suspends, the closest parent Suspense boundary switches to showing the fallback. This can lead to a jarring user experience if it was already displaying some content. Try pressing this button:

<Sandpack>

```json package.json hidden
{
"dependencies": {
"react": "experimental",
"react-dom": "experimental"
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
}
}
```

```js App.js
import { Suspense, useState } from 'react';
import IndexPage from './IndexPage.js';
import ArtistPage from './ArtistPage.js';
import Layout from './Layout.js';

export default function App() {
return (
<Suspense fallback={<BigSpinner />}>
```

```
<Router />
</Suspense>
);
}

function Router() {
const [page, setPage] = useState('/');

function navigate(url) {
setPage(url);
}

let content;
if (page === '/') {
content = (
<IndexPage navigate={navigate} />
);
} else if (page === '/the-beatles') {
content = (
<ArtistPage
artist={{
id: 'the-beatles',
name: 'The Beatles',
}}
/>
);
}
return (
<Layout>
{content}
</Layout>
);
}

function BigSpinner() {
return <h2>■ Loading...</h2>;
}
```

```js Layout.js
export default function Layout({ children }) {
return (
<div className="layout">
<section className="header">
Music Browser
</section>
<main>
{children}
</main>
</div>
);
}
```

```js IndexPage.js
export default function IndexPage({ navigate }) {
return (
<button onClick={() => navigate('/the-beatles')}>
Open The Beatles artist page
</button>
);
}
```

```js ArtistPage.js
import { Suspense } from 'react';
import Albums from './Albums.js';
import Biography from './Biography.js';
import Panel from './Panel.js';

export default function ArtistPage({ artist }) {
return (
<>
<h1>{artist.name}</h1>
<Biography artistId={artist.id} />
<Suspense fallback={<AlbumsGlimmer />}>
```

```
<Panel>
<Albums artistId={artist.id} />
</Panel>
</Suspense>
</>
);
}

function AlbumsGlimmer() {
return (
<div className="glimmer-panel">
<div className="glimmer-line" />
<div className="glimmer-line" />
<div className="glimmer-line" />
</div>
);
}
```

```js Albums.js hidden
import { fetchData } from './data.js';

// Note: this component is written using an experimental API
// that's not yet available in stable versions of React.

// For a realistic example you can follow today, try a framework
// that's integrated with Suspense, like Relay or Next.js.

export default function Albums({ artistId }) {
const albums = use(fetchData(`/${artistId}/albums`));
return (
<ul>
{albums.map(album => (
<li key={album.id}>
{album.title} ({album.year})
</li>
))}
</ul>
);
```

```
}

// This is a workaround for a bug to get the demo running.
// TODO: replace with real implementation when the bug is fixed.
function use(promise) {
if (promise.status === 'fulfilled') {
return promise.value;
} else if (promise.status === 'rejected') {
throw promise.reason;
} else if (promise.status === 'pending') {
throw promise;
} else {
promise.status = 'pending';
promise.then(
result => {
promise.status = 'fulfilled';
promise.value = result;
},
reason => {
promise.status = 'rejected';
promise.reason = reason;
},
);
throw promise;
}
}
```

```js Biography.js hidden
import { fetchData } from './data.js';

// Note: this component is written using an experimental API
// that's not yet available in stable versions of React.

// For a realistic example you can follow today, try a framework
// that's integrated with Suspense, like Relay or Next.js.

export default function Biography({ artistId }) {
const bio = use(fetchData(`/${artistId}/bio`));
```

```
return (
<section>
<p className="bio">{bio}</p>
</section>
);
}

// This is a workaround for a bug to get the demo running.
// TODO: replace with real implementation when the bug is fixed.
function use(promise) {
if (promise.status === 'fulfilled') {
return promise.value;
} else if (promise.status === 'rejected') {
throw promise.reason;
} else if (promise.status === 'pending') {
throw promise;
} else {
promise.status = 'pending';
promise.then(
result => {
promise.status = 'fulfilled';
promise.value = result;
},
reason => {
promise.status = 'rejected';
promise.reason = reason;
},
);
throw promise;
}
}
```

```js Panel.js hidden
export default function Panel({ children }) {
return (
<section className="panel">
```

```
{children}
</section>
);
}
```

```js data.js hidden
// Note: the way you would do data fetching depends on
// the framework that you use together with Suspense.
// Normally, the caching logic would be inside a framework.

let cache = new Map();

export function fetchData(url) {
if (!cache.has(url)) {
cache.set(url, getData(url));
}
return cache.get(url);
}

async function getData(url) {
if (url === '/the-beatles/albums') {
return await getAlbums();
} else if (url === '/the-beatles/bio') {
return await getBio();
} else {
throw Error('Not implemented');
}
}

async function getBio() {
// Add a fake delay to make waiting noticeable.
await new Promise(resolve => {
setTimeout(resolve, 500);
});

return `The Beatles were an English rock band,
formed in Liverpool in 1960, that comprised
John Lennon, Paul McCartney, George Harrison
```

```
and Ringo Starr.`;
}

async function getAlbums() {
// Add a fake delay to make waiting noticeable.
await new Promise(resolve => {
setTimeout(resolve, 3000);
});

return [{
id: 13,
title: 'Let It Be',
year: 1970
}, {
id: 12,
title: 'Abbey Road',
year: 1969
}, {
id: 11,
title: 'Yellow Submarine',
year: 1969
}, {
id: 10,
title: 'The Beatles',
year: 1968
}, {
id: 9,
title: 'Magical Mystery Tour',
year: 1967
}, {
id: 8,
title: 'Sgt. Pepper\'s Lonely Hearts Club Band',
year: 1967
}, {
id: 7,
title: 'Revolver',
year: 1966
```

```
}, {
id: 6,
title: 'Rubber Soul',
year: 1965
}, {
id: 5,
title: 'Help!',
year: 1965
}, {
id: 4,
title: 'Beatles For Sale',
year: 1964
}, {
id: 3,
title: 'A Hard Day\'s Night',
year: 1964
}, {
id: 2,
title: 'With The Beatles',
year: 1963
}, {
id: 1,
title: 'Please Please Me',
year: 1963
}];
}
```

```css
main {
min-height: 200px;
padding: 10px;
}

.layout {
border: 1px solid black;
}
```

```
.header {
background: #222;
padding: 10px;
text-align: center;
color: white;
}

.bio { font-style: italic; }

.panel {
border: 1px solid #aaa;
border-radius: 6px;
margin-top: 20px;
padding: 10px;
}

.glimmer-panel {
border: 1px dashed #aaa;
background: linear-gradient(90deg, rgba(221,221,221,1) 0%, rgba(255,255,255,1) 100%);
border-radius: 6px;
margin-top: 20px;
padding: 10px;
}

.glimmer-line {
display: block;
width: 60%;
height: 20px;
margin: 10px;
border-radius: 4px;
background: #f0f0f0;
}
```

</Sandpack>

When you pressed the button, the `Router` component rendered `ArtistPage` instead of `IndexPage`. A component inside `ArtistPage` suspended, so the closest Suspense boundary started showing the fallback. The closest Suspense boundary was near the root, so the whole site layout got replaced by `BigSpinner`.

To prevent this, you can mark the navigation state update as a *transition* with [`startTransition`:](/reference/react/startTransition)

```js {5,7}
function Router() {
const [page, setPage] = useState('/');

function navigate(url) {
startTransition(() => {
setPage(url);
});
}
// ...
```

This tells React that the state transition is not urgent, and it's better to keep showing the previous page instead of hiding any already revealed content. Now clicking the button "waits" for the `Biography` to load:

<Sandpack>

```json package.json hidden
{
"dependencies": {
"react": "experimental",
"react-dom": "experimental"
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
}
}
```

```js App.js
import { Suspense, startTransition, useState } from 'react';
import IndexPage from './IndexPage.js';
import ArtistPage from './ArtistPage.js';
```

```
import Layout from './Layout.js';

export default function App() {
return (
<Suspense fallback={<BigSpinner />}>
<Router />
</Suspense>
);
}

function Router() {
const [page, setPage] = useState('/');

function navigate(url) {
startTransition(() => {
setPage(url);
});
}

let content;
if (page === '/') {
content = (
<IndexPage navigate={navigate} />
);
} else if (page === '/the-beatles') {
content = (
<ArtistPage
artist={{
id: 'the-beatles',
name: 'The Beatles',
}}
/>
);
}
return (
<Layout>
{content}
</Layout>
```

```
);
}

function BigSpinner() {
return <h2>■ Loading...</h2>;
}
```

```js Layout.js
export default function Layout({ children }) {
return (
<div className="layout">
<section className="header">
Music Browser
</section>
<main>
{children}
</main>
</div>
);
}
```

```js IndexPage.js
export default function IndexPage({ navigate }) {
return (
<button onClick={() => navigate('/the-beatles')}>
Open The Beatles artist page
</button>
);
}
```

```js ArtistPage.js
import { Suspense } from 'react';
import Albums from './Albums.js';
import Biography from './Biography.js';
import Panel from './Panel.js';
```

```
export default function ArtistPage({ artist }) {
return (
<>
<h1>{artist.name}</h1>
<Biography artistId={artist.id} />
<Suspense fallback={<AlbumsGlimmer />}>
<Panel>
<Albums artistId={artist.id} />
</Panel>
</Suspense>
</>
);
}

function AlbumsGlimmer() {
return (
<div className="glimmer-panel">
<div className="glimmer-line" />
<div className="glimmer-line" />
<div className="glimmer-line" />
</div>
);
}
```

```js Albums.js hidden
import { fetchData } from './data.js';

// Note: this component is written using an experimental API
// that's not yet available in stable versions of React.

// For a realistic example you can follow today, try a framework
// that's integrated with Suspense, like Relay or Next.js.

export default function Albums({ artistId }) {
const albums = use(fetchData(`/${artistId}/albums`));
return (
<ul>
{albums.map(album => (
```

```
<li key={album.id}>
{album.title} ({album.year})
</li>
))}
</ul>
);
}

// This is a workaround for a bug to get the demo running.
// TODO: replace with real implementation when the bug is fixed.
function use(promise) {
if (promise.status === 'fulfilled') {
return promise.value;
} else if (promise.status === 'rejected') {
throw promise.reason;
} else if (promise.status === 'pending') {
throw promise;
} else {
promise.status = 'pending';
promise.then(
result => {
promise.status = 'fulfilled';
promise.value = result;
},
reason => {
promise.status = 'rejected';
promise.reason = reason;
},
);
throw promise;
}
}
```

```js Biography.js hidden
import { fetchData } from './data.js';
```

```
// Note: this component is written using an experimental API
// that's not yet available in stable versions of React.

// For a realistic example you can follow today, try a framework
// that's integrated with Suspense, like Relay or Next.js.

export default function Biography({ artistId }) {
const bio = use(fetchData(`/${artistId}/bio`));
return (
<section>
<p className="bio">{bio}</p>
</section>
);
}

// This is a workaround for a bug to get the demo running.
// TODO: replace with real implementation when the bug is fixed.
function use(promise) {
if (promise.status === 'fulfilled') {
return promise.value;
} else if (promise.status === 'rejected') {
throw promise.reason;
} else if (promise.status === 'pending') {
throw promise;
} else {
promise.status = 'pending';
promise.then(
result => {
promise.status = 'fulfilled';
promise.value = result;
},
reason => {
promise.status = 'rejected';
promise.reason = reason;
},
);
throw promise;
```

```
}
}
```

```js Panel.js hidden
export default function Panel({ children }) {
return (
<section className="panel">
{children}
</section>
);
}
```

```js data.js hidden
// Note: the way you would do data fetching depends on
// the framework that you use together with Suspense.
// Normally, the caching logic would be inside a framework.

let cache = new Map();

export function fetchData(url) {
if (!cache.has(url)) {
cache.set(url, getData(url));
}
return cache.get(url);
}

async function getData(url) {
if (url === '/the-beatles/albums') {
return await getAlbums();
} else if (url === '/the-beatles/bio') {
return await getBio();
} else {
throw Error('Not implemented');
}
}

async function getBio() {
```

```
// Add a fake delay to make waiting noticeable.
await new Promise(resolve => {
setTimeout(resolve, 500);
});

return `The Beatles were an English rock band,
formed in Liverpool in 1960, that comprised
John Lennon, Paul McCartney, George Harrison
and Ringo Starr.`;
}

async function getAlbums() {
// Add a fake delay to make waiting noticeable.
await new Promise(resolve => {
setTimeout(resolve, 3000);
});

return [{
id: 13,
title: 'Let It Be',
year: 1970
}, {
id: 12,
title: 'Abbey Road',
year: 1969
}, {
id: 11,
title: 'Yellow Submarine',
year: 1969
}, {
id: 10,
title: 'The Beatles',
year: 1968
}, {
id: 9,
title: 'Magical Mystery Tour',
year: 1967
```

```
}, {
id: 8,
title: 'Sgt. Pepper\'s Lonely Hearts Club Band',
year: 1967
}, {
id: 7,
title: 'Revolver',
year: 1966
}, {
id: 6,
title: 'Rubber Soul',
year: 1965
}, {
id: 5,
title: 'Help!',
year: 1965
}, {
id: 4,
title: 'Beatles For Sale',
year: 1964
}, {
id: 3,
title: 'A Hard Day\'s Night',
year: 1964
}, {
id: 2,
title: 'With The Beatles',
year: 1963
}, {
id: 1,
title: 'Please Please Me',
year: 1963
}];
}
```

```css
main {
min-height: 200px;
padding: 10px;
}

.layout {
border: 1px solid black;
}

.header {
background: #222;
padding: 10px;
text-align: center;
color: white;
}

.bio { font-style: italic; }

.panel {
border: 1px solid #aaa;
border-radius: 6px;
margin-top: 20px;
padding: 10px;
}

.glimmer-panel {
border: 1px dashed #aaa;
background: linear-gradient(90deg, rgba(221,221,221,1) 0%, rgba(255,255,255,1) 100%);
border-radius: 6px;
margin-top: 20px;
padding: 10px;
}

.glimmer-line {
display: block;
width: 60%;
height: 20px;
margin: 10px;
```

```
border-radius: 4px;
background: #f0f0f0;
}
```

</Sandpack>

A transition doesn't wait for *all* content to load. It only waits long enough to avoid hiding already revealed content. For example, the website `Layout` was already revealed, so it would be bad to hide it behind a loading spinner. However, the nested `Suspense` boundary around `Albums` is new, so the transition doesn't wait for it.

<Note>

Suspense-enabled routers are expected to wrap the navigation updates into transitions by default.

</Note>

Indicating that a transition is happening {/*indicating-that-a-transition-is-happening*/}

In the above example, once you click the button, there is no visual indication that a navigation is in progress. To add an indicator, you can replace [`startTransition`](/reference/react/startTransition) with [`useTransition`](/reference/react/useTransition) which gives you a boolean `isPending` value. In the example below, it's used to change the website header styling while a transition is happening:

<Sandpack>

```json package.json hidden
{
"dependencies": {
"react": "experimental",
"react-dom": "experimental"
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
}
}
```

```js App.js
```

```
import { Suspense, useState, useTransition } from 'react';
import IndexPage from './IndexPage.js';
import ArtistPage from './ArtistPage.js';
import Layout from './Layout.js';

export default function App() {
return (
<Suspense fallback={<BigSpinner />}>
<Router />
</Suspense>
);
}

function Router() {
const [page, setPage] = useState('/');
const [isPending, startTransition] = useTransition();

function navigate(url) {
startTransition(() => {
setPage(url);
});
}

let content;
if (page === '/') {
content = (
<IndexPage navigate={navigate} />
);
} else if (page === '/the-beatles') {
content = (
<ArtistPage
artist={{
id: 'the-beatles',
name: 'The Beatles',
}}
/>
);
}
```

```
return (
<Layout isPending={isPending}>
{content}
</Layout>
);
}

function BigSpinner() {
return <h2>■ Loading...</h2>;
}
```

```js Layout.js
export default function Layout({ children, isPending }) {
return (
<div className="layout">
<section className="header" style={{
opacity: isPending ? 0.7 : 1
}}>
Music Browser
</section>
<main>
{children}
</main>
</div>
);
}
```

```js IndexPage.js
export default function IndexPage({ navigate }) {
return (
<button onClick={() => navigate('/the-beatles')}>
Open The Beatles artist page
</button>
);
}
```

```

```js ArtistPage.js
import { Suspense } from 'react';
import Albums from './Albums.js';
import Biography from './Biography.js';
import Panel from './Panel.js';

export default function ArtistPage({ artist }) {
return (
<>
<h1>{artist.name}</h1>
<Biography artistId={artist.id} />
<Suspense fallback={<AlbumsGlimmer />}>
<Panel>
<Albums artistId={artist.id} />
</Panel>
</Suspense>
</>
);
}

function AlbumsGlimmer() {
return (
<div className="glimmer-panel">
<div className="glimmer-line" />
<div className="glimmer-line" />
<div className="glimmer-line" />
</div>
);
}
```

```js Albums.js hidden
import { fetchData } from './data.js';

// Note: this component is written using an experimental API
// that's not yet available in stable versions of React.

```
// For a realistic example you can follow today, try a framework
// that's integrated with Suspense, like Relay or Next.js.

export default function Albums({ artistId }) {
const albums = use(fetchData(`/${artistId}/albums`));
return (
<ul>
{albums.map(album => (
<li key={album.id}>
{album.title} ({album.year})
</li>
))}
</ul>
);
}

// This is a workaround for a bug to get the demo running.
// TODO: replace with real implementation when the bug is fixed.
function use(promise) {
if (promise.status === 'fulfilled') {
return promise.value;
} else if (promise.status === 'rejected') {
throw promise.reason;
} else if (promise.status === 'pending') {
throw promise;
} else {
promise.status = 'pending';
promise.then(
result => {
promise.status = 'fulfilled';
promise.value = result;
},
reason => {
promise.status = 'rejected';
promise.reason = reason;
},
);
```

````
throw promise;
}
}
```

```js Biography.js hidden
import { fetchData } from './data.js';

// Note: this component is written using an experimental API
// that's not yet available in stable versions of React.

// For a realistic example you can follow today, try a framework
// that's integrated with Suspense, like Relay or Next.js.

export default function Biography({ artistId }) {
const bio = use(fetchData(`/${artistId}/bio`));
return (
<section>
<p className="bio">{bio}</p>
</section>
);
}

// This is a workaround for a bug to get the demo running.
// TODO: replace with real implementation when the bug is fixed.
function use(promise) {
if (promise.status === 'fulfilled') {
return promise.value;
} else if (promise.status === 'rejected') {
throw promise.reason;
} else if (promise.status === 'pending') {
throw promise;
} else {
promise.status = 'pending';
promise.then(
result => {
promise.status = 'fulfilled';
promise.value = result;
},
````

````
reason => {
promise.status = 'rejected';
promise.reason = reason;
},
);
throw promise;
}
}
```

```js Panel.js hidden
export default function Panel({ children }) {
return (
<section className="panel">
{children}
</section>
);
}
```

```js data.js hidden
// Note: the way you would do data fetching depends on
// the framework that you use together with Suspense.
// Normally, the caching logic would be inside a framework.

let cache = new Map();

export function fetchData(url) {
if (!cache.has(url)) {
cache.set(url, getData(url));
}
return cache.get(url);
}

async function getData(url) {
if (url === '/the-beatles/albums') {
return await getAlbums();
} else if (url === '/the-beatles/bio') {
return await getBio();
````

```
} else {
throw Error('Not implemented');
}
}

async function getBio() {
// Add a fake delay to make waiting noticeable.
await new Promise(resolve => {
setTimeout(resolve, 500);
});

return `The Beatles were an English rock band,
formed in Liverpool in 1960, that comprised
John Lennon, Paul McCartney, George Harrison
and Ringo Starr.`;
}

async function getAlbums() {
// Add a fake delay to make waiting noticeable.
await new Promise(resolve => {
setTimeout(resolve, 3000);
});

return [{
id: 13,
title: 'Let It Be',
year: 1970
}, {
id: 12,
title: 'Abbey Road',
year: 1969
}, {
id: 11,
title: 'Yellow Submarine',
year: 1969
}, {
id: 10,
title: 'The Beatles',
```

year: 1968
}, {
id: 9,
title: 'Magical Mystery Tour',
year: 1967
}, {
id: 8,
title: 'Sgt. Pepper\'s Lonely Hearts Club Band',
year: 1967
}, {
id: 7,
title: 'Revolver',
year: 1966
}, {
id: 6,
title: 'Rubber Soul',
year: 1965
}, {
id: 5,
title: 'Help!',
year: 1965
}, {
id: 4,
title: 'Beatles For Sale',
year: 1964
}, {
id: 3,
title: 'A Hard Day\'s Night',
year: 1964
}, {
id: 2,
title: 'With The Beatles',
year: 1963
}, {
id: 1,

title: 'Please Please Me',
year: 1963
}];
}
```

```css
main {
min-height: 200px;
padding: 10px;
}

.layout {
border: 1px solid black;
}

.header {
background: #222;
padding: 10px;
text-align: center;
color: white;
}

.bio { font-style: italic; }

.panel {
border: 1px solid #aaa;
border-radius: 6px;
margin-top: 20px;
padding: 10px;
}

.glimmer-panel {
border: 1px dashed #aaa;
background: linear-gradient(90deg, rgba(221,221,221,1) 0%, rgba(255,255,255,1) 100%);
border-radius: 6px;
margin-top: 20px;
padding: 10px;
}

```
.glimmer-line {
display: block;
width: 60%;
height: 20px;
margin: 10px;
border-radius: 4px;
background: #f0f0f0;
}
```

</Sandpack>

Resetting Suspense boundaries on navigation {/*resetting-suspense-boundaries-on-navigation*/}

During a transition, React will avoid hiding already revealed content. However, if you navigate to a route with different parameters, you might want to tell React it is *different* content. You can express this with a `key`:

```js
<ProfilePage key={queryParams.id} />
```

Imagine you're navigating within a user's profile page, and something suspends. If that update is wrapped in a transition, it will not trigger the fallback for already visible content. That's the expected behavior.

However, now imagine you're navigating between two different user profiles. In that case, it makes sense to show the fallback. For example, one user's timeline is *different content* from another user's timeline. By specifying a `key`, you ensure that React treats different users' profiles as different components, and resets the Suspense boundaries during navigation. Suspense-integrated routers should do this automatically.

Providing a fallback for server errors and client-only content {/*providing-a-fallback-for-server-errors-and-client-only-content*/}

If you use one of the [streaming server rendering APIs](/reference/react-dom/server) (or a framework that relies on them), React will also use your `<Suspense>` boundaries to handle errors on the server. If a component throws an error on the server, React will not abort the server render. Instead, it will find the closest `<Suspense>` component above it and include its fallback (such as a spinner) into the generated server HTML. The user will see a spinner at first.

On the client, React will attempt to render the same component again. If it errors on the client too, React will throw the error and display the closest [error boundary.](/reference/react/Component#static-getderivedstatefromerror) However, if it does not error

on the client, React will not display the error to the user since the content was eventually displayed successfully.

You can use this to opt out some components from rendering on the server. To do this, throw an error in the server environment and then wrap them in a `<Suspense>` boundary to replace their HTML with fallbacks:

```js
<Suspense fallback={<Loading />}>
<Chat />
</Suspense>

function Chat() {
if (typeof window === 'undefined') {
throw Error('Chat should only render on the client.');
}
// ...
}
```

The server HTML will include the loading indicator. It will be replaced by the `Chat` component on the client.

Troubleshooting {/*troubleshooting*/}

How do I prevent the UI from being replaced by a fallback during an update? {/*preventing-unwanted-fallbacks*/}

Replacing visible UI with a fallback creates a jarring user experience. This can happen when an update causes a component to suspend, and the nearest Suspense boundary is already showing content to the user.

To prevent this from happening, [mark the update as non-urgent using `startTransition`](#preventing-already-revealed-content-from-hiding). During a transition, React will wait until enough data has loaded to prevent an unwanted fallback from appearing:

```js {2-3,5}
function handleNextPageClick() {
// If this update suspends, don't hide the already displayed content
startTransition(() => {
setCurrentPage(currentPage + 1);
});
}
```

```

This will avoid hiding existing content. However, any newly rendered `Suspense` boundaries will still immediately display fallbacks to avoid blocking the UI and let the user see the content as it becomes available.

**React will only prevent unwanted fallbacks during non-urgent updates**. It will not delay a render if it's the result of an urgent update. You must opt in with an API like [`startTransition`](/reference/react/startTransition) or [`useDeferredValue`](/reference/react/useDeferredValue).

If your router is integrated with Suspense, it should wrap its updates into [`startTransition`](/reference/react/startTransition) automatically.

---

title: useReducer

---

<Intro>

`useReducer` is a React Hook that lets you add a [reducer](/learn/extracting-state-logic-into-a-reducer) to your component.

```js
const [state, dispatch] = useReducer(reducer, initialArg, init?)
```

</Intro>

<InlineToc />

Reference {/*reference*/}

`useReducer(reducer, initialArg, init?)` {/*usereducer*/}

Call `useReducer` at the top level of your component to manage its state with a [reducer.](/learn/extracting-state-logic-into-a-reducer)

```js
import { useReducer } from 'react';

function reducer(state, action) {
// ...
}

function MyComponent() {
const [state, dispatch] = useReducer(reducer, { age: 42 });
```
```

// ...
```

[See more examples below.](#usage)

#### Parameters {/*parameters*/}

* `reducer`: The reducer function that specifies how the state gets updated. It must be pure, should take the state and action as arguments, and should return the next state. State and action can be of any types.

* `initialArg`: The value from which the initial state is calculated. It can be a value of any type. How the initial state is calculated from it depends on the next `init` argument.

* **optional** `init`: The initializer function that should return the initial state. If it's not specified, the initial state is set to `initialArg`. Otherwise, the initial state is set to the result of calling `init(initialArg)`.

#### Returns {/*returns*/}

`useReducer` returns an array with exactly two values:

1. The current state. During the first render, it's set to `init(initialArg)` or `initialArg` (if there's no `init`).

2. The [`dispatch` function](#dispatch) that lets you update the state to a different value and trigger a re-render.

#### Caveats {/*caveats*/}

* `useReducer` is a Hook, so you can only call it **at the top level of your component** or your own Hooks. You can't call it inside loops or conditions. If you need that, extract a new component and move the state into it.

* In Strict Mode, React will **call your reducer and initializer twice** in order to [help you find accidental impurities.](#my-reducer-or-initializer-function-runs-twice) This is development-only behavior and does not affect production. If your reducer and initializer are pure (as they should be), this should not affect your logic. The result from one of the calls is ignored.

---

### `dispatch` function {/*dispatch*/}

The `dispatch` function returned by `useReducer` lets you update the state to a different value and trigger a re-render. You need to pass the action as the only argument to the `dispatch` function:

```js
const [state, dispatch] = useReducer(reducer, { age: 42 });

function handleClick() {
dispatch({ type: 'incremented_age' });
// ...
```
```

React will set the next state to the result of calling the `reducer` function you've provided with the current `state` and the action you've passed to `dispatch`.

Parameters {/*dispatch-parameters*/}

* `action`: The action performed by the user. It can be a value of any type. By convention, an action is usually an object with a `type` property identifying it and, optionally, other properties with additional information.

Returns {/*dispatch-returns*/}

`dispatch` functions do not have a return value.

Caveats {/*setstate-caveats*/}

* The `dispatch` function **only updates the state variable for the *next* render**. If you read the state variable after calling the `dispatch` function, [you will still get the old value](#ive-dispatched-an-action-but-logging-gives-me-the-old-state-value) that was on the screen before your call.

* If the new value you provide is identical to the current `state`, as determined by an [`Object.is`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/is) comparison, React will **skip re-rendering the component and its children.** This is an optimization. React may still need to call your component before ignoring the result, but it shouldn't affect your code.

* React [batches state updates.](/learn/queueing-a-series-of-state-updates) It updates the screen **after all the event handlers have run** and have called their `set` functions. This prevents multiple re-renders during a single event. In the rare case that you need to force React to update the screen earlier, for example to access the DOM, you can use [`flushSync`.](/reference/react-dom/flushSync)

Usage {/*usage*/}

Adding a reducer to a component {/*adding-a-reducer-to-a-component*/}

Call `useReducer` at the top level of your component to manage state with a [reducer.](/learn/extracting-state-logic-into-a-reducer)

```js [[1, 8, "state"], [2, 8, "dispatch"], [4, 8, "reducer"], [3, 8, "{ age: 42 }"]]
import { useReducer } from 'react';

function reducer(state, action) {
// ...
}

function MyComponent() {
const [state, dispatch] = useReducer(reducer, { age: 42 });
// ...
```

`useReducer` returns an array with exactly two items:

1. The <CodeStep step={1}>current state</CodeStep> of this state variable, initially set to the <CodeStep step={3}>initial state</CodeStep> you provided.

2. The <CodeStep step={2}>`dispatch` function</CodeStep> that lets you change it in response to interaction.

To update what's on the screen, call <CodeStep step={2}>`dispatch`</CodeStep> with an object representing what the user did, called an *action*:

```js [[2, 2, "dispatch"]]
function handleClick() {
dispatch({ type: 'incremented_age' });
}
```

React will pass the current state and the action to your <CodeStep step={4}>reducer function</CodeStep>. Your reducer will calculate and return the next state. React will store that next state, render your component with it, and update the UI.

<Sandpack>

```js
import { useReducer } from 'react';

function reducer(state, action) {
if (action.type === 'incremented_age') {
return {
age: state.age + 1
};
}
throw Error('Unknown action.');
}

export default function Counter() {
const [state, dispatch] = useReducer(reducer, { age: 42 });

return (
<>
<button onClick={() => {
dispatch({ type: 'incremented_age' })
}}>
Increment age
```

</button>

Hello! You are {state.age}.

</>

);

}

```

```css

button { display: block; margin-top: 10px; }

```

</Sandpack>

`useReducer` is very similar to [`useState`](/reference/react/useState), but it lets you move the state update logic from event handlers into a single function outside of your component. Read more about [choosing between `useState` and `useReducer`.](/learn/extracting-state-logic-into-a-reducer#comparing-usestate-and-usereducer)

Writing the reducer function {/*writing-the-reducer-function*/}

A reducer function is declared like this:

```js

function reducer(state, action) {

// ...

}

```

Then you need to fill in the code that will calculate and return the next state. By convention, it is common to write it as a [`switch` statement.](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/switch) For each `case` in the `switch`, calculate and return some next state.

```js {4-7,10-13}

function reducer(state, action) {

switch (action.type) {

case 'incremented_age': {

return {

name: state.name,

age: state.age + 1

};

```
}
case 'changed_name': {
return {
name: action.nextName,
age: state.age
};
}
}
throw Error('Unknown action: ' + action.type);
}
```

Actions can have any shape. By convention, it's common to pass objects with a `type` property identifying the action. It should include the minimal necessary information that the reducer needs to compute the next state.

```js {5,9-12}
function Form() {
const [state, dispatch] = useReducer(reducer, { name: 'Taylor', age: 42 });

function handleButtonClick() {
dispatch({ type: 'incremented_age' });
}

function handleInputChange(e) {
dispatch({
type: 'changed_name',
nextName: e.target.value
});
}
// ...
```

The action type names are local to your component. [Each action describes a single interaction, even if that leads to multiple changes in data.](/learn/extracting-state-logic-into-a-reducer#writing-reducers-well) The shape of the state is arbitrary, but usually it'll be an object or an array.

Read [extracting state logic into a reducer](/learn/extracting-state-logic-into-a-reducer) to learn more.

<Pitfall>

State is read-only. Don't modify any objects or arrays in state:

```js {4,5}
function reducer(state, action) {
switch (action.type) {
case 'incremented_age': {
// ■ Don't mutate an object in state like this:
state.age = state.age + 1;
return state;
}
```

Instead, always return new objects from your reducer:

```js {4-8}
function reducer(state, action) {
switch (action.type) {
case 'incremented_age': {
// ■ Instead, return a new object
return {
...state,
age: state.age + 1
};
}
```

Read [updating objects in state](/learn/updating-objects-in-state) and [updating arrays in state](/learn/updating-arrays-in-state) to learn more.

</Pitfall>

<Recipes titleText="Basic useReducer examples" titleId="examples-basic">

Form (object) {/*form-object*/}

In this example, the reducer manages a state object with two fields: `name` and `age`.

<Sandpack>

```js
import { useReducer } from 'react';

function reducer(state, action) {
```

```
switch (action.type) {
case 'incremented_age': {
return {
name: state.name,
age: state.age + 1
};
}
case 'changed_name': {
return {
name: action.nextName,
age: state.age
};
}
}
throw Error('Unknown action: ' + action.type);
}

const initialState = { name: 'Taylor', age: 42 };

export default function Form() {
const [state, dispatch] = useReducer(reducer, initialState);

function handleButtonClick() {
dispatch({ type: 'incremented_age' });
}

function handleInputChange(e) {
dispatch({
type: 'changed_name',
nextName: e.target.value
});
}

return (
<>
<input
value={state.name}
onChange={handleInputChange}
/>
```

<button onClick={handleButtonClick}>

Increment age

</button>

Hello, {state.name}. You are {state.age}.

</>

);

}

```

```css

button { display: block; margin-top: 10px; }

```

</Sandpack>

<Solution />

Todo list (array) {/*todo-list-array*/}

In this example, the reducer manages an array of tasks. The array needs to be updated [without mutation.](/learn/updating-arrays-in-state)

<Sandpack>

```js App.js

import { useReducer } from 'react';

import AddTask from './AddTask.js';

import TaskList from './TaskList.js';

function tasksReducer(tasks, action) {

switch (action.type) {

case 'added': {

return [...tasks, {

id: action.id,

text: action.text,

done: false

}];

}

case 'changed': {

return tasks.map(t => {

if (t.id === action.task.id) {

```
return action.task;
} else {
return t;
}
});
}
case 'deleted': {
return tasks.filter(t => t.id !== action.id);
}
default: {
throw Error('Unknown action: ' + action.type);
}
}
}

export default function TaskApp() {
const [tasks, dispatch] = useReducer(
tasksReducer,
initialTasks
);

function handleAddTask(text) {
dispatch({
type: 'added',
id: nextId++,
text: text,
});
}

function handleChangeTask(task) {
dispatch({
type: 'changed',
task: task
});
}

function handleDeleteTask(taskId) {
dispatch({
```

```
type: 'deleted',
id: taskId
});
}

return (
<>
<h1>Prague itinerary</h1>
<AddTask
onAddTask={handleAddTask}
/>
<TaskList
tasks={tasks}
onChangeTask={handleChangeTask}
onDeleteTask={handleDeleteTask}
/>
</>
);
}

let nextId = 3;
const initialTasks = [
{ id: 0, text: 'Visit Kafka Museum', done: true },
{ id: 1, text: 'Watch a puppet show', done: false },
{ id: 2, text: 'Lennon Wall pic', done: false }
];
```

```js AddTask.js hidden
import { useState } from 'react';

export default function AddTask({ onAddTask }) {
const [text, setText] = useState('');
return (
<>
<input
placeholder="Add task"
value={text}
```

```
onChange={e => setText(e.target.value)}
/>
<button onClick={() => {
setText('');
onAddTask(text);
}}>Add</button>
</>
)
}
```

```js TaskList.js hidden
import { useState } from 'react';

export default function TaskList({
tasks,
onChangeTask,
onDeleteTask
}) {
return (
<ul>
{tasks.map(task => (
<li key={task.id}>
<Task
task={task}
onChange={onChangeTask}
onDelete={onDeleteTask}
/>
</li>
))}
</ul>
);
}

function Task({ task, onChange, onDelete }) {
const [isEditing, setIsEditing] = useState(false);
let taskContent;
```

```
if (isEditing) {
taskContent = (
<>
<input
value={task.text}
onChange={e => {
onChange({
...task,
text: e.target.value
});
}} />
<button onClick={() => setIsEditing(false)}>
Save
</button>
</>
);
} else {
taskContent = (
<>
{task.text}
<button onClick={() => setIsEditing(true)}>
Edit
</button>
</>
);
}
return (
<label>
<input
type="checkbox"
checked={task.done}
onChange={e => {
onChange({
...task,
done: e.target.checked
```

```
});
}}
/>
{taskContent}
<button onClick={() => onDelete(task.id)}>
Delete
</button>
</label>
);
}
```

```css
button { margin: 5px; }
li { list-style-type: none; }
ul, li { margin: 0; padding: 0; }
```

</Sandpack>

<Solution />

Writing concise update logic with Immer {/*writing-concise-update-logic-with-immer*/}

If updating arrays and objects without mutation feels tedious, you can use a library like [Immer](https://github.com/immerjs/use-immer#useimmerreducer) to reduce repetitive code. Immer lets you write concise code as if you were mutating objects, but under the hood it performs immutable updates:

<Sandpack>

```js App.js
import { useImmerReducer } from 'use-immer';
import AddTask from './AddTask.js';
import TaskList from './TaskList.js';

function tasksReducer(draft, action) {
switch (action.type) {
case 'added': {
draft.push({
id: action.id,
text: action.text,
```

```
done: false
});
break;
}
case 'changed': {
const index = draft.findIndex(t =>
t.id === action.task.id
);
draft[index] = action.task;
break;
}
case 'deleted': {
return draft.filter(t => t.id !== action.id);
}
default: {
throw Error('Unknown action: ' + action.type);
}
}
}

export default function TaskApp() {
const [tasks, dispatch] = useImmerReducer(
tasksReducer,
initialTasks
);

function handleAddTask(text) {
dispatch({
type: 'added',
id: nextId++,
text: text,
});
}

function handleChangeTask(task) {
dispatch({
type: 'changed',
```

```
task: task
});
}

function handleDeleteTask(taskId) {
dispatch({
type: 'deleted',
id: taskId
});
}

return (
<>
<h1>Prague itinerary</h1>
<AddTask
onAddTask={handleAddTask}
/>
<TaskList
tasks={tasks}
onChangeTask={handleChangeTask}
onDeleteTask={handleDeleteTask}
/>
</>
);
}

let nextId = 3;
const initialTasks = [
{ id: 0, text: 'Visit Kafka Museum', done: true },
{ id: 1, text: 'Watch a puppet show', done: false },
{ id: 2, text: 'Lennon Wall pic', done: false },
];
```

```js AddTask.js hidden
import { useState } from 'react';

export default function AddTask({ onAddTask }) {
const [text, setText] = useState('');
```

```
return (
<>
<input
placeholder="Add task"
value={text}
onChange={e => setText(e.target.value)}
/>
<button onClick={() => {
setText('');
onAddTask(text);
}}>Add</button>
</>
)
}
```

```js TaskList.js hidden
import { useState } from 'react';

export default function TaskList({
tasks,
onChangeTask,
onDeleteTask
}) {
return (
<ul>
{tasks.map(task => (
<li key={task.id}>
<Task
task={task}
onChange={onChangeTask}
onDelete={onDeleteTask}
/>
</li>
))}
</ul>
);
```

```
}

function Task({ task, onChange, onDelete }) {
const [isEditing, setIsEditing] = useState(false);
let taskContent;
if (isEditing) {
taskContent = (
<>
<input
value={task.text}
onChange={e => {
onChange({
...task,
text: e.target.value
});
}} />
<button onClick={() => setIsEditing(false)}>
Save
</button>
</>
);
} else {
taskContent = (
<>
{task.text}
<button onClick={() => setIsEditing(true)}>
Edit
</button>
</>
);
}
return (
<label>
<input
type="checkbox"
checked={task.done}
```

onChange={e => {
onChange({
...task,
done: e.target.checked
});
}}
/>
{taskContent}
<button onClick={() => onDelete(task.id)}>
Delete
</button>
</label>
);
}
```

```css
button { margin: 5px; }
li { list-style-type: none; }
ul, li { margin: 0; padding: 0; }
```

```json package.json
{
"dependencies": {
"immer": "1.7.3",
"react": "latest",
"react-dom": "latest",
"react-scripts": "latest",
"use-immer": "0.5.1"
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
}

}
```

</Sandpack>

<Solution />

</Recipes>

Avoiding recreating the initial state {/*avoiding-recreating-the-initial-state*/}

React saves the initial state once and ignores it on the next renders.

```js
function createInitialState(username) {
// ...
}

function TodoList({ username }) {
const [state, dispatch] = useReducer(reducer, createInitialState(username));
// ...
```

Although the result of `createInitialState(username)` is only used for the initial render, you're still calling this function on every render. This can be wasteful if it's creating large arrays or performing expensive calculations.

To solve this, you may **pass it as an _initializer_ function** to `useReducer` as the third argument instead:

```js {6}
function createInitialState(username) {
// ...
}

function TodoList({ username }) {
const [state, dispatch] = useReducer(reducer, username, createInitialState);
// ...
```

Notice that you’re passing `createInitialState`, which is the *function itself*, and not `createInitialState()`, which is the result of calling it. This way, the initial state does not get re-created after initialization.
```

In the above example, `createInitialState` takes a `username` argument. If your initializer doesn't need any information to compute the initial state, you may pass `null` as the second argument to `useReducer`.

<Recipes titleText="The difference between passing an initializer and passing the initial state directly" titleId="examples-initializer">

Passing the initializer function {/*passing-the-initializer-function*/}

This example passes the initializer function, so the `createInitialState` function only runs during initialization. It does not run when component re-renders, such as when you type into the input.

<Sandpack>

```js App.js hidden
import TodoList from './TodoList.js';

export default function App() {
return <TodoList username="Taylor" />;
}
```

```js TodoList.js active
import { useReducer } from 'react';

function createInitialState(username) {
const initialTodos = [];
for (let i = 0; i < 50; i++) {
initialTodos.push({
id: i,
text: username + "'s task #" + (i + 1)
});
}
return {
draft: '',
todos: initialTodos,
};
}

function reducer(state, action) {
switch (action.type) {
case 'changed_draft': {
return {
```

```
draft: action.nextDraft,
todos: state.todos,
};
};
case 'added_todo': {
return {
draft: '',
todos: [{
id: state.todos.length,
text: state.draft
}, ...state.todos]
}
}
}
throw Error('Unknown action: ' + action.type);
}

export default function TodoList({ username }) {
const [state, dispatch] = useReducer(
reducer,
username,
createInitialState
);
return (
<>
<input
value={state.draft}
onChange={e => {
dispatch({
type: 'changed_draft',
nextDraft: e.target.value
})
}}
/>
<button onClick={() => {
dispatch({ type: 'added_todo' });
```

}}>Add</button>

{state.todos.map(item => (

<li key={item.id}>

{item.text}

))}

</>

);

}

```

</Sandpack>

<Solution />

#### Passing the initial state directly {/*passing-the-initial-state-directly*/}

This example **does not** pass the initializer function, so the `createInitialState` function runs on every render, such as when you type into the input. There is no observable difference in behavior, but this code is less efficient.

<Sandpack>

```js App.js hidden

import TodoList from './TodoList.js';

export default function App() {

return <TodoList username="Taylor" />;

}

```

```js TodoList.js active

import { useReducer } from 'react';

function createInitialState(username) {

const initialTodos = [];

for (let i = 0; i < 50; i++) {

initialTodos.push({

id: i,

text: username + "'s task #" + (i + 1)

```
});
}
return {
draft: '',
todos: initialTodos,
};
}

function reducer(state, action) {
switch (action.type) {
case 'changed_draft': {
return {
draft: action.nextDraft,
todos: state.todos,
};
};
case 'added_todo': {
return {
draft: '',
todos: [{
id: state.todos.length,
text: state.draft
}, ...state.todos]
}
}
}
throw Error('Unknown action: ' + action.type);
}

export default function TodoList({ username }) {
const [state, dispatch] = useReducer(
reducer,
createInitialState(username)
);
return (
<>
<input
```

value={state.draft}
onChange={e => {
dispatch({
type: 'changed_draft',
nextDraft: e.target.value
})
}}
/>
<button onClick={() => {
dispatch({ type: 'added_todo' });
}}>Add</button>

{state.todos.map(item => (
<li key={item.id}>
{item.text}

))}

</>
);
}
```

</Sandpack>

<Solution />

</Recipes>

---

## Troubleshooting {/*troubleshooting*/}

### I've dispatched an action, but logging gives me the old state value {/*ive-dispatched-an-action-but-logging-gives-me-the-old-state-value*/}

Calling the `dispatch` function **does not change state in the running code**:

```js {4,5,8}
function handleClick() {
console.log(state.age); // 42

```
dispatch({ type: 'incremented_age' }); // Request a re-render with 43
console.log(state.age); // Still 42!

setTimeout(() => {
console.log(state.age); // Also 42!
}, 5000);
}
```

This is because [states behaves like a snapshot.](/learn/state-as-a-snapshot) Updating state requests another render with the new state value, but does not affect the `state` JavaScript variable in your already-running event handler.

If you need to guess the next state value, you can calculate it manually by calling the reducer yourself:

```js
const action = { type: 'incremented_age' };
dispatch(action);

const nextState = reducer(state, action);
console.log(state); // { age: 42 }
console.log(nextState); // { age: 43 }
```

I've dispatched an action, but the screen doesn't update
{/*ive-dispatched-an-action-but-the-screen-doesnt-update*/}

React will **ignore your update if the next state is equal to the previous state,** as determined by an [`Object.is`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/is) comparison. This usually happens when you change an object or an array in state directly:

```js {4-5,9-10}
function reducer(state, action) {
switch (action.type) {
case 'incremented_age': {
// ■ Wrong: mutating existing object
state.age++;
return state;
}
case 'changed_name': {
// ■ Wrong: mutating existing object
```

```
state.name = action.nextName;
return state;
}
// ...
}
}
```

You mutated an existing `state` object and returned it, so React ignored the update. To fix this, you need to ensure that you're always [updating objects in state](/learn/updating-objects-in-state) and [updating arrays in state](/learn/updating-arrays-in-state) instead of mutating them:

```js {4-8,11-15}
function reducer(state, action) {
switch (action.type) {
case 'incremented_age': {
// ■ Correct: creating a new object
return {
...state,
age: state.age + 1
};
}
case 'changed_name': {
// ■ Correct: creating a new object
return {
...state,
name: action.nextName
};
}
// ...
}
}
```

A part of my reducer state becomes undefined after dispatching {/*a-part-of-my-reducer-state-becomes-undefined-after-dispatching*/}

Make sure that every `case` branch **copies all of the existing fields** when returning the new state:

```js {5}
function reducer(state, action) {
switch (action.type) {
case 'incremented_age': {
return {
...state, // Don't forget this!
age: state.age + 1
};
}
// ...
```

Without `...state` above, the returned next state would only contain the `age` field and nothing else.

My entire reducer state becomes undefined after dispatching {/*my-entire-reducer-state-becomes-undefined-after-dispatching*/}

If your state unexpectedly becomes `undefined`, you're likely forgetting to `return` state in one of the cases, or your action type doesn't match any of the `case` statements. To find why, throw an error outside the `switch`:

```js {10}
function reducer(state, action) {
switch (action.type) {
case 'incremented_age': {
// ...
}
case 'edited_name': {
// ...
}
}
throw Error('Unknown action: ' + action.type);
}
```

You can also use a static type checker like TypeScript to catch such mistakes.

I'm getting an error: "Too many re-renders" {/*im-getting-an-error-too-many-re-renders*/}

You might get an error that says: `Too many re-renders. React limits the number of renders to prevent an infinite loop.` Typically, this means that you're unconditionally dispatching an action *during render*, so your component enters a loop: render, dispatch (which causes a render), render, dispatch (which causes a render), and so on. Very often, this is caused by a mistake in specifying an event handler:

```js {1-2}
// ■ Wrong: calls the handler during render
return <button onClick={handleClick()}>Click me</button>

// ■ Correct: passes down the event handler
return <button onClick={handleClick}>Click me</button>

// ■ Correct: passes down an inline function
return <button onClick={(e) => handleClick(e)}>Click me</button>
```

If you can't find the cause of this error, click on the arrow next to the error in the console and look through the JavaScript stack to find the specific `dispatch` function call responsible for the error.

My reducer or initializer function runs twice {/*my-reducer-or-initializer-function-runs-twice*/}

In [Strict Mode](/reference/react/StrictMode), React will call your reducer and initializer functions twice. This shouldn't break your code.

This **development-only** behavior helps you [keep components pure.](/learn/keeping-components-pure) React uses the result of one of the calls, and ignores the result of the other call. As long as your component, initializer, and reducer functions are pure, this shouldn't affect your logic. However, if they are accidentally impure, this helps you notice the mistakes.

For example, this impure reducer function mutates an array in state:

```js {4-6}
function reducer(state, action) {
switch (action.type) {
case 'added_todo': {
// ■ Mistake: mutating state
state.todos.push({ id: nextId++, text: action.text });
return state;
}
// ...
}
}
```

Because React calls your reducer function twice, you'll see the todo was added twice, so you'll know that there is a mistake. In this example, you can fix the mistake by [replacing the array instead of mutating it](/learn/updating-arrays-in-state#adding-to-an-array):

```js {4-11}
function reducer(state, action) {
switch (action.type) {
case 'added_todo': {
// ■ Correct: replacing with new state
return {
...state,
todos: [
...state.todos,
{ id: nextId++, text: action.text }
]
};
}
// ...
}
}
```

Now that this reducer function is pure, calling it an extra time doesn't make a difference in behavior. This is why React calling it twice helps you find mistakes. **Only component, initializer, and reducer functions need to be pure.** Event handlers don't need to be pure, so React will never call your event handlers twice.

Read [keeping components pure](/learn/keeping-components-pure) to learn more.

---
title: createRef

<Pitfall>

`createRef` is mostly used for [class components.](/reference/react/Component) Function components typically rely on [`useRef`](/reference/react/useRef) instead.

</Pitfall>

<Intro>

`createRef` creates a [ref](/learn/referencing-values-with-refs) object which can contain arbitrary value.

```js
class MyInput extends Component {
inputRef = createRef();
// ...
}
```

</Intro>

<InlineToc />

Reference {/*reference*/}

`createRef()` {/*createref*/}

Call `createRef` to declare a [ref](/learn/referencing-values-with-refs) inside a [class component.](/reference/react/Component)

```js
import { createRef, Component } from 'react';

class MyComponent extends Component {
intervalRef = createRef();
inputRef = createRef();
// ...
```

[See more examples below.](#usage)

Parameters {/*parameters*/}

`createRef` takes no parameters.

Returns {/*returns*/}

`createRef` returns an object with a single property:

* `current`: Initially, it's set to the `null`. You can later set it to something else. If you pass the ref object to React as a `ref` attribute to a JSX node, React will set its `current` property.

Caveats {/*caveats*/}

* `createRef` always returns a *different* object. It's equivalent to writing `{ current: null }` yourself.

* In a function component, you probably want [`useRef`](/reference/react/useRef) instead which always returns the same object.

* `const ref = useRef()` is equivalent to `const [ref, _] = useState(() => createRef(null))`.

Usage {/*usage*/}

Declaring a ref in a class component {/*declaring-a-ref-in-a-class-component*/}

To declare a ref inside a [class component,](/reference/react/Component) call `createRef` and assign its result to a class field:

```js {4}
import { Component, createRef } from 'react';

class Form extends Component {
inputRef = createRef();

// ...
}
```

If you now pass `ref={this.inputRef}` to an `<input>` in your JSX, React will populate `this.inputRef.current` with the input DOM node. For example, here is how you make a button that focuses the input:

<Sandpack>

```js
import { Component, createRef } from 'react';

export default class Form extends Component {
inputRef = createRef();

handleClick = () => {
this.inputRef.current.focus();
}

render() {
return (
<>
<input ref={this.inputRef} />
<button onClick={this.handleClick}>
Focus the input
</button>
</>
```

);

}

}

```

</Sandpack>

<Pitfall>

`createRef` is mostly used for [class components.](/reference/react/Component) Function components typically rely on [`useRef`](/reference/react/useRef) instead.

</Pitfall>

---

## Alternatives {/*alternatives*/}

### Migrating from a class with `createRef` to a function with `useRef` {/*migrating-from-a-class-with-createref-to-a-function-with-useref*/}

We recommend using function components instead of [class components](/reference/react/Component) in new code. If you have some existing class components using `createRef`, here is how you can convert them. This is the original code:

<Sandpack>

```js
import { Component, createRef } from 'react';

export default class Form extends Component {

inputRef = createRef();

handleClick = () => {

this.inputRef.current.focus();

}

render() {

return (

<>

<input ref={this.inputRef} />

<button onClick={this.handleClick}>

Focus the input

</button>

</>

);

}

}

```

</Sandpack>

When you [convert this component from a class to a function,](/reference/react/Component#alternatives) replace calls to `createRef` with calls to [`useRef`:](/reference/react/useRef)

<Sandpack>

```js
import { useRef } from 'react';

export default function Form() {

const inputRef = useRef(null);

function handleClick() {

inputRef.current.focus();

}

return (

<>

<input ref={inputRef} />

<button onClick={handleClick}>

Focus the input

</button>

</>

);

}

```

</Sandpack>

---

title: '"use server"'

<Wip>

This section is incomplete.
```

These directives are needed only if you're [using React Server Components](/learn/start-a-new-react-project#bleeding-edge-react-frameworks) or building a library compatible with them.

</Wip>

<Intro>

`'use server'` marks server-side functions that can be called from client-side code.

</Intro>

<InlineToc />

Reference {/*reference*/}

`'use server'` {/*use-server*/}

Add `'use server';` at the very top of an async function to mark that the function can be executed by the client.

```js
async function addToCart(data) {
'use server';
// ...
}

// <ProductDetailPage addToCart={addToCart} />
```

This function can be passed to the client. When called on the client, it will make a network request to the server that includes a serialized copy of any arguments passed. If the server function returns a value, that value will be serialized and returned to the client.

Alternatively, add `'use server';` at the very top of a file to mark all exports within that file as async server functions that can be used anywhere, including imported in client component files.

Caveats {/*caveats*/}

* Remember that parameters to functions marked with `'use server'` are fully client-controlled. For security, always treat them as untrusted input, making sure to validate and escape the arguments as appropriate.
* To avoid the confusion that might result from mixing client- and server-side code in the same file, `'use server'` can only be used in server-side files; the resulting functions can be passed to client components through props.
* Because the underlying network calls are always asynchronous, `'use server'` can be used only on async functions.

* Directives like `'use server'` must be at the very beginning of their function or file, above any other code including imports (comments above directives are OK). They must be written with single or double quotes, not backticks. (The `'use xyz'` directive format somewhat resembles the `useXyz()` Hook naming convention, but the similarity is coincidental.)

---

title: cloneElement

<Pitfall>

Using `cloneElement` is uncommon and can lead to fragile code. [See common alternatives.](#alternatives)

</Pitfall>

<Intro>

`cloneElement` lets you create a new React element using another element as a starting point.

```js
const clonedElement = cloneElement(element, props, ...children)
```

</Intro>

<InlineToc />

Reference {/*reference*/}

`cloneElement(element, props, ...children)` {/*cloneelement*/}

Call `cloneElement` to create a React element based on the `element`, but with different `props` and `children`:

```js
import { cloneElement } from 'react';

// ...
const clonedElement = cloneElement(
<Row title="Cabbage">
Hello
</Row>,
{ isHighlighted: true },
'Goodbye'
```

);

console.log(clonedElement); // <Row title="Cabbage">Goodbye</Row>
```

[See more examples below.](#usage)

#### Parameters {/*parameters*/}

* `element`: The `element` argument must be a valid React element. For example, it could be a JSX node like `<Something />`, the result of calling [`createElement`](/reference/react/createElement), or the result of another `cloneElement` call.

* `props`: The `props` argument must either be an object or `null`. If you pass `null`, the cloned element will retain all of the original `element.props`. Otherwise, for every prop in the `props` object, the returned element will "prefer" the value from `props` over the value from `element.props`. The rest of the props will be filled from the original `element.props`. If you pass `props.key` or `props.ref`, they will replace the original ones.

* **optional** `...children`: Zero or more child nodes. They can be any React nodes, including React elements, strings, numbers, [portals](/reference/react-dom/createPortal), empty nodes (`null`, `undefined`, `true`, and `false`), and arrays of React nodes. If you don't pass any `...children` arguments, the original `element.props.children` will be preserved.

#### Returns {/*returns*/}

`cloneElement` returns a React element object with a few properties:

* `type`: Same as `element.type`.
* `props`: The result of shallowly merging `element.props` with the overriding `props` you have passed.
* `ref`: The original `element.ref`, unless it was overridden by `props.ref`.
* `key`: The original `element.key`, unless it was overridden by `props.key`.

Usually, you'll return the element from your component or make it a child of another element. Although you may read the element's properties, it's best to treat every element as opaque after it's created, and only render it.

#### Caveats {/*caveats*/}

* Cloning an element **does not modify the original element.**

* You should only **pass children as multiple arguments to `cloneElement` if they are all statically known,** like `cloneElement(element, null, child1, child2, child3)`. If your children are dynamic, pass the entire array as the third argument: `cloneElement(element, null, listItems)`. This ensures that React will [warn you about missing `key`s](/learn/rendering-lists#keeping-list-items-in-order-with-key) for any dynamic lists. For static lists this is not necessary because they never reorder.

* `cloneElement` makes it harder to trace the data flow, so **try the [alternatives](#alternatives) instead.**

---
```

Usage {/*usage*/}

Overriding props of an element {/*overriding-props-of-an-element*/}

To override the props of some <CodeStep step={1}>React element</CodeStep>, pass it to `cloneElement` with the <CodeStep step={2}>props you want to override</CodeStep>:

```js [[1, 5, "<Row title=\\"Cabbage\\" />"], [2, 6, "{ isHighlighted: true }"], [3, 4, "clonedElement"]]
import { cloneElement } from 'react';

// ...
const clonedElement = cloneElement(
<Row title="Cabbage" />,
{ isHighlighted: true }
);
```

Here, the resulting <CodeStep step={3}>cloned element</CodeStep> will be `<Row title="Cabbage" isHighlighted={true} />`.

Let's walk through an example to see when it's useful.

Imagine a `List` component that renders its [`children`](/learn/passing-props-to-a-component#passing-jsx-as-children) as a list of selectable rows with a "Next" button that changes which row is selected. The `List` component needs to render the selected `Row` differently, so it clones every `<Row>` child that it has received, and adds an extra `isHighlighted: true` or `isHighlighted: false` prop:

```js {6-8}
export default function List({ children }) {
const [selectedIndex, setSelectedIndex] = useState(0);
return (
<div className="List">
{Children.map(children, (child, index) =>
cloneElement(child, {
isHighlighted: index === selectedIndex
})
)}
```

Let's say the original JSX received by `List` looks like this:

```js {2-4}
<List>
```

<Row title="Cabbage" />

<Row title="Garlic" />

<Row title="Apple" />

</List>

```

By cloning its children, the `List` can pass extra information to every `Row` inside. The result looks like this:

```js {4,8,12}

<List>

<Row

title="Cabbage"

isHighlighted={true}

/>

<Row

title="Garlic"

isHighlighted={false}

/>

<Row

title="Apple"

isHighlighted={false}

/>

</List>

```

Notice how pressing "Next" updates the state of the `List`, and highlights a different row:

<Sandpack>

```js

import List from './List.js';

import Row from './Row.js';

import { products } from './data.js';

export default function App() {

return (

<List>

{products.map(product =>

<Row

```
key={product.id}
title={product.title}
/>
)}
</List>
);
}
```

```js List.js active
import { Children, cloneElement, useState } from 'react';

export default function List({ children }) {
const [selectedIndex, setSelectedIndex] = useState(0);
return (
<div className="List">
{Children.map(children, (child, index) =>
cloneElement(child, {
isHighlighted: index === selectedIndex
})
)}
<hr />
<button onClick={() => {
setSelectedIndex(i =>
(i + 1) % Children.count(children)
);
}}>
Next
</button>
</div>
);
}
```

```js Row.js
export default function Row({ title, isHighlighted }) {
return (
```

```
<div className={[
'Row',
isHighlighted ? 'RowHighlighted' : ''
].join(' ')}>
{title}
</div>
);
}
```

```js data.js
export const products = [
{ title: 'Cabbage', id: 1 },
{ title: 'Garlic', id: 2 },
{ title: 'Apple', id: 3 },
];
```

```css
.List {
display: flex;
flex-direction: column;
border: 2px solid grey;
padding: 5px;
}

.Row {
border: 2px dashed black;
padding: 5px;
margin: 5px;
}

.RowHighlighted {
background: #ffa;
}

button {
height: 40px;
font-size: 20px;
```

}
```

</Sandpack>

To summarize, the `List` cloned the `<Row />` elements it received and added an extra prop to them.

<Pitfall>

Cloning children makes it hard to tell how the data flows through your app. Try one of the [alternatives.](#alternatives)

</Pitfall>

---

## Alternatives {/*alternatives*/}

### Passing data with a render prop {/*passing-data-with-a-render-prop*/}

Instead of using `cloneElement`, consider accepting a *render prop* like `renderItem`. Here, `List` receives `renderItem` as a prop. `List` calls `renderItem` for every item and passes `isHighlighted` as an argument:

```js {1,7}
export default function List({ items, renderItem }) {
const [selectedIndex, setSelectedIndex] = useState(0);
return (
<div className="List">
{items.map((item, index) => {
const isHighlighted = index === selectedIndex;
return renderItem(item, isHighlighted);
})}
```

The `renderItem` prop is called a "render prop" because it's a prop that specifies how to render something. For example, you can pass a `renderItem` implementation that renders a `<Row>` with the given `isHighlighted` value:

```js {3,7}
<List
items={products}
renderItem={(product, isHighlighted) =>
<Row
key={product.id}

title={product.title}
isHighlighted={isHighlighted}
/>
}
/>
```

The end result is the same as with `cloneElement`:

```js {4,8,12}
<List>
<Row
title="Cabbage"
isHighlighted={true}
/>
<Row
title="Garlic"
isHighlighted={false}
/>
<Row
title="Apple"
isHighlighted={false}
/>
</List>
```

However, you can clearly trace where the `isHighlighted` value is coming from.

<Sandpack>

```js
import List from './List.js';
import Row from './Row.js';
import { products } from './data.js';

export default function App() {
return (
<List
items={products}

```
renderItem={(product, isHighlighted) =>
<Row
key={product.id}
title={product.title}
isHighlighted={isHighlighted}
/>
}
/>
);
}
```

```js List.js active
import { useState } from 'react';

export default function List({ items, renderItem }) {
const [selectedIndex, setSelectedIndex] = useState(0);
return (
<div className="List">
{items.map((item, index) => {
const isHighlighted = index === selectedIndex;
return renderItem(item, isHighlighted);
})}
<hr />
<button onClick={() => {
setSelectedIndex(i =>
(i + 1) % items.length
);
}}>
Next
</button>
</div>
);
}
```

```js Row.js
```

```
export default function Row({ title, isHighlighted }) {
return (
<div className={[
'Row',
isHighlighted ? 'RowHighlighted' : ''
].join(' ')}>
{title}
</div>
);
}
```

```js data.js
export const products = [
{ title: 'Cabbage', id: 1 },
{ title: 'Garlic', id: 2 },
{ title: 'Apple', id: 3 },
];
```

```css
.List {
display: flex;
flex-direction: column;
border: 2px solid grey;
padding: 5px;
}

.Row {
border: 2px dashed black;
padding: 5px;
margin: 5px;
}

.RowHighlighted {
background: #ffa;
}

button {
```

height: 40px;

font-size: 20px;

}

```

</Sandpack>

This pattern is preferred to `cloneElement` because it is more explicit.

---

### Passing data through context {/*passing-data-through-context*/}

Another alternative to `cloneElement` is to [pass data through context.](/learn/passing-data-deeply-with-context)

For example, you can call [`createContext`](/reference/react/createContext) to define a `HighlightContext`:

```js
export const HighlightContext = createContext(false);
```

Your `List` component can wrap every item it renders into a `HighlightContext` provider:

```js {8,10}
export default function List({ items, renderItem }) {
const [selectedIndex, setSelectedIndex] = useState(0);
return (
<div className="List">
{items.map((item, index) => {
const isHighlighted = index === selectedIndex;
return (
<HighlightContext.Provider key={item.id} value={isHighlighted}>
{renderItem(item)}
</HighlightContext.Provider>
);
})}
```

With this approach, `Row` does not need to receive an `isHighlighted` prop at all. Instead, it reads the context:
```

```js Row.js {2}
export default function Row({ title }) {
const isHighlighted = useContext(HighlightContext);
// ...
```

This allows the calling component to not know or worry about passing `isHighlighted` to `<Row>`:

```js {4}
<List
items={products}
renderItem={product =>
<Row title={product.title} />
}
/>
```

Instead, `List` and `Row` coordinate the highlighting logic through context.

<Sandpack>

```js
import List from './List.js';
import Row from './Row.js';
import { products } from './data.js';

export default function App() {
return (
<List
items={products}
renderItem={(product) =>
<Row title={product.title} />
}
/>
);
}
```

```js List.js active
import { useState } from 'react';
```

```
import { HighlightContext } from './HighlightContext.js';

export default function List({ items, renderItem }) {
const [selectedIndex, setSelectedIndex] = useState(0);
return (
<div className="List">
{items.map((item, index) => {
const isHighlighted = index === selectedIndex;
return (
<HighlightContext.Provider
key={item.id}
value={isHighlighted}
>
{renderItem(item)}
</HighlightContext.Provider>
);
})}
<hr />
<button onClick={() => {
setSelectedIndex(i =>
(i + 1) % items.length
);
}}>
Next
</button>
</div>
);
}
```

```js Row.js
import { useContext } from 'react';
import { HighlightContext } from './HighlightContext.js';

export default function Row({ title }) {
const isHighlighted = useContext(HighlightContext);
return (
```

```
<div className={[
'Row',
isHighlighted ? 'RowHighlighted' : ''
].join(' ')}>
{title}
</div>
);
}
```

```js HighlightContext.js
import { createContext } from 'react';

export const HighlightContext = createContext(false);
```

```js data.js
export const products = [
{ title: 'Cabbage', id: 1 },
{ title: 'Garlic', id: 2 },
{ title: 'Apple', id: 3 },
];
```

```css
.List {
display: flex;
flex-direction: column;
border: 2px solid grey;
padding: 5px;
}

.Row {
border: 2px dashed black;
padding: 5px;
margin: 5px;
}

.RowHighlighted {
```

background: #ffa;

}

button {

height: 40px;

font-size: 20px;

}

```

</Sandpack>

[Learn more about passing data through context.](/reference/react/useContext#passing-data-deeply-into-the-tree)

---

### Extracting logic into a custom Hook {/*extracting-logic-into-a-custom-hook*/}

Another approach you can try is to extract the "non-visual" logic into your own Hook, and use the information returned by your Hook to decide what to render. For example, you could write a `useList` custom Hook like this:

```js
import { useState } from 'react';

export default function useList(items) {

const [selectedIndex, setSelectedIndex] = useState(0);

function onNext() {

setSelectedIndex(i =>

(i + 1) % items.length

);

}

const selected = items[selectedIndex];

return [selected, onNext];

}
```

Then you could use it like this:

```js {2,9,13}
export default function App() {

const [selected, onNext] = useList(products);

```
return (
<div className="List">
{products.map(product =>
<Row
key={product.id}
title={product.title}
isHighlighted={selected === product}
/>
)}
<hr />
<button onClick={onNext}>
Next
</button>
</div>
);
}
```

The data flow is explicit, but the state is inside the `useList` custom Hook that you can use from any component:

<Sandpack>

```js
import Row from './Row.js';
import useList from './useList.js';
import { products } from './data.js';

export default function App() {
const [selected, onNext] = useList(products);
return (
<div className="List">
{products.map(product =>
<Row
key={product.id}
title={product.title}
isHighlighted={selected === product}
/>
```

)}
<hr />
<button onClick={onNext}>
Next
</button>

);
}
```

```js useList.js
import { useState } from 'react';

export default function useList(items) {
const [selectedIndex, setSelectedIndex] = useState(0);

function onNext() {
setSelectedIndex(i =>
(i + 1) % items.length
);
}

const selected = items[selectedIndex];
return [selected, onNext];
}
```

```js Row.js
export default function Row({ title, isHighlighted }) {
return (
<div className={[
'Row',
isHighlighted ? 'RowHighlighted' : ''
].join(' ')}>
{title}
</div>
);
}
```
```

```js data.js
export const products = [
{ title: 'Cabbage', id: 1 },
{ title: 'Garlic', id: 2 },
{ title: 'Apple', id: 3 },
];
```

```css
.List {
display: flex;
flex-direction: column;
border: 2px solid grey;
padding: 5px;
}

.Row {
border: 2px dashed black;
padding: 5px;
margin: 5px;
}

.RowHighlighted {
background: #ffa;
}

button {
height: 40px;
font-size: 20px;
}
```

</Sandpack>

This approach is particularly useful if you want to reuse this logic between different components.

---

title: "Built-in React APIs"

<Intro>

In addition to [Hooks](/reference/react) and [Components](/reference/react/components), the `react` package exports a few other APIs that are useful for defining components. This page lists all the remaining modern React APIs.

</Intro>

---

* [`createContext`](/reference/react/createContext) lets you define and provide context to the child components. Used with [`useContext`.](/reference/react/useContext)

* [`forwardRef`](/reference/react/forwardRef) lets your component expose a DOM node as a ref to the parent. Used with [`useRef`.](/reference/react/useRef)

* [`lazy`](/reference/react/lazy) lets you defer loading a component's code until it's rendered for the first time.

* [`memo`](/reference/react/memo) lets your component skip re-renders with same props. Used with [`useMemo`](/reference/react/useMemo) and [`useCallback`.](/reference/react/useCallback)

* [`startTransition`](/reference/react/startTransition) lets you mark a state update as non-urgent. Similar to [`useTransition`.](/reference/react/useTransition)

---

title: createElement

<Intro>

`createElement` lets you create a React element. It serves as an alternative to writing [JSX.](/learn/writing-markup-with-jsx)

```js
const element = createElement(type, props, ...children)
```

</Intro>

<InlineToc />

Reference {/*reference*/}

`createElement(type, props, ...children)` {/*createelement*/}

Call `createElement` to create a React element with the given `type`, `props`, and `children`.

```js
import { createElement } from 'react';

function Greeting({ name }) {
```

return createElement(

'h1',

{ className: 'greeting' },

'Hello'

);

}

```

[See more examples below.](#usage)

#### Parameters {/*parameters*/}

* `type`: The `type` argument must be a valid React component type. For example, it could be a tag name string (such as `'div'` or `'span'`), or a React component (a function, a class, or a special component like [`Fragment`](/reference/react/Fragment)).

* `props`: The `props` argument must either be an object or `null`. If you pass `null`, it will be treated the same as an empty object. React will create an element with props matching the `props` you have passed. Note that `ref` and `key` from your `props` object are special and will *not* be available as `element.props.ref` and `element.props.key` on the returned `element`. They will be available as `element.ref` and `element.key`.

* **optional** `...children`: Zero or more child nodes. They can be any React nodes, including React elements, strings, numbers, [portals](/reference/react-dom/createPortal), empty nodes (`null`, `undefined`, `true`, and `false`), and arrays of React nodes.

#### Returns {/*returns*/}

`createElement` returns a React element object with a few properties:

* `type`: The `type` you have passed.

* `props`: The `props` you have passed except for `ref` and `key`. If the `type` is a component with legacy `type.defaultProps`, then any missing or undefined `props` will get the values from `type.defaultProps`.

* `ref`: The `ref` you have passed. If missing, `null`.

* `key`: The `key` you have passed, coerced to a string. If missing, `null`.

Usually, you'll return the element from your component or make it a child of another element. Although you may read the element's properties, it's best to treat every element as opaque after it's created, and only render it.

#### Caveats {/*caveats*/}

* You must **treat React elements and their props as [immutable](https://en.wikipedia.org/wiki/Immutable_object)** and never change their contents after creation. In development, React will [freeze](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/freeze) the returned element and its `props` property shallowly to enforce this.
```

* When you use JSX, **you must start a tag with a capital letter to render your own custom component.** In other words, `<Something />` is equivalent to `createElement(Something)`, but `<something />` (lowercase) is equivalent to `createElement('something')` (note it's a string, so it will be treated as a built-in HTML tag).

* You should only **pass children as multiple arguments to `createElement` if they are all statically known,** like `createElement('h1', {}, child1, child2, child3)`. If your children are dynamic, pass the entire array as the third argument: `createElement('ul', {}, listItems)`. This ensures that React will [warn you about missing `key`s](/learn/rendering-lists#keeping-list-items-in-order-with-key) for any dynamic lists. For static lists this is not necessary because they never reorder.

Usage {/*usage*/}

Creating an element without JSX {/*creating-an-element-without-jsx*/}

If you don't like [JSX](/learn/writing-markup-with-jsx) or can't use it in your project, you can use `createElement` as an alternative.

To create an element without JSX, call `createElement` with some <CodeStep step={1}>type</CodeStep>, <CodeStep step={2}>props</CodeStep>, and <CodeStep step={3}>children</CodeStep>:

```js [[1, 5, "'h1'"], [2, 6, "{ className: 'greeting' }"], [3, 7, "'Hello ',"], [3, 8, "createElement('i', null, name),"], [3, 9, "'. Welcome!'"]]
import { createElement } from 'react';

function Greeting({ name }) {
return createElement(
'h1',
{ className: 'greeting' },
'Hello ',
createElement('i', null, name),
'. Welcome!'
);
}
```

The <CodeStep step={3}>children</CodeStep> are optional, and you can pass as many as you need (the example above has three children). This code will display a `<h1>` header with a greeting. For comparison, here is the same example rewritten with JSX:

```js [[1, 3, "h1"], [2, 3, "className=\\"greeting\\""], [3, 4, "Hello <i>{name}</i>. Welcome!"], [1, 5, "h1"]]
function Greeting({ name }) {
return (
```

<h1 className="greeting">

Hello {name}. Welcome!

</h1>

);

}

```

To render your own React component, pass a function like `Greeting` as the <CodeStep step={1}>type</CodeStep> instead of a string like `'h1'`:

```js [[1, 2, "Greeting"], [2, 2, "{ name: 'Taylor' }"]]

export default function App() {

return createElement(Greeting, { name: 'Taylor' });

}

```

With JSX, it would look like this:

```js [[1, 2, "Greeting"], [2, 2, "name=\\"Taylor\\""]]

export default function App() {

return <Greeting name="Taylor" />;

}

```

Here is a complete example written with `createElement`:

<Sandpack>

```js

import { createElement } from 'react';

function Greeting({ name }) {

return createElement(

'h1',

{ className: 'greeting' },

'Hello ',

createElement('i', null, name),

'. Welcome!'

);

}

export default function App() {

```
return createElement(
Greeting,
{ name: 'Taylor' }
);
}
```

```css
.greeting {
color: darkgreen;
font-family: Georgia;
}
```

</Sandpack>

And here is the same example written using JSX:

<Sandpack>

```js
function Greeting({ name }) {
return (
<h1 className="greeting">
Hello <i>{name}</i>. Welcome!
</h1>
);
}
export default function App() {
return <Greeting name="Taylor" />;
}
```

```css
.greeting {
color: darkgreen;
font-family: Georgia;
}
```

</Sandpack>

Both coding styles are fine, so you can use whichever one you prefer for your project. The main benefit of using JSX compared to `createElement` is that it's easy to see which closing tag corresponds to which opening tag.

<DeepDive>

What is a React element, exactly? {/*what-is-a-react-element-exactly*/}

An element is a lightweight description of a piece of the user interface. For example, both `<Greeting name="Taylor" />` and `createElement(Greeting, { name: 'Taylor' })` produce an object like this:

```js
// Slightly simplified
{
type: Greeting,
props: {
name: 'Taylor'
},
key: null,
ref: null,
}
```

Note that creating this object does not render the `Greeting` component or create any DOM elements.

A React element is more like a description--an instruction for React to later render the `Greeting` component. By returning this object from your `App` component, you tell React what to do next.

Creating elements is extremely cheap so you don't need to try to optimize or avoid it.

</DeepDive>

---

title: React DOM APIs

<Intro>

The `react-dom` package contains methods that are only supported for the web applications (which run in the browser DOM environment). They are not supported for React Native.

</Intro>

APIs {/*apis*/}

These APIs can be imported from your components. They are rarely used:

* [`createPortal`](/reference/react-dom/createPortal) lets you render child components in a different part of the DOM tree.

* [`flushSync`](/reference/react-dom/flushSync) lets you force React to flush a state update and update the DOM synchronously.

Entry points {/*entry-points*/}

The `react-dom` package provides two additional entry points:

* [`react-dom/client`](/reference/react-dom/client) contains APIs to render React components on the client (in the browser).

* [`react-dom/server`](/reference/react-dom/server) contains APIs to render React components on the server.

Deprecated APIs {/*deprecated-apis*/}

<Deprecated>

These APIs will be removed in a future major version of React.

</Deprecated>

* [`findDOMNode`](/reference/react-dom/findDOMNode) finds the closest DOM node corresponding to a class component instance.

* [`hydrate`](/reference/react-dom/hydrate) mounts a tree into the DOM created from server HTML. Deprecated in favor of [`hydrateRoot`](/reference/react-dom/client/hydrateRoot).

* [`render`](/reference/react-dom/render) mounts a tree into the DOM. Deprecated in favor of [`createRoot`](/reference/react-dom/client/createRoot).

* [`unmountComponentAtNode`](/reference/react-dom/unmountComponentAtNode) unmounts a tree from the DOM. Deprecated in favor of [`root.unmount()`](/reference/react-dom/client/createRoot#root-unmount).

---

title: createPortal

<Intro>

`createPortal` lets you render some children into a different part of the DOM.

```js
```

<SomeComponent />

{createPortal(children, domNode, key?)}

```

</Intro>

<InlineToc />

---

## Reference {/*reference*/}

### `createPortal(children, domNode, key?)` {/*createportal*/}

To create a portal, call `createPortal`, passing some JSX, and the DOM node where it should be rendered:

```js
import { createPortal } from 'react-dom';

// ...

<div>

<p>This child is placed in the parent div.</p>

{createPortal(

<p>This child is placed in the document body.</p>,

document.body

)}

</div>
```

[See more examples below.](#usage)

A portal only changes the physical placement of the DOM node. In every other way, the JSX you render into a portal acts as a child node of the React component that renders it. For example, the child can access the context provided by the parent tree, and events bubble up from children to parents according to the React tree.

Parameters {/*parameters*/}

* `children`: Anything that can be rendered with React, such as a piece of JSX (e.g. `<div />` or `<SomeComponent />`), a [Fragment](/reference/react/Fragment) (`<>...</>`), a string or a number, or an array of these.
```

* `domNode`: Some DOM node, such as those returned by `document.getElementById()`. The node must already exist. Passing a different DOM node during an update will cause the portal content to be recreated.

* **optional** `key`: A unique string or number to be used as the portal's [key.](/learn/rendering-lists/#keeping-list-items-in-order-with-key)

Returns {/*returns*/}

`createPortal` returns a React node that can be included into JSX or returned from a React component. If React encounters it in the render output, it will place the provided `children` inside the provided `domNode`.

Caveats {/*caveats*/}

* Events from portals propagate according to the React tree rather than the DOM tree. For example, if you click inside a portal, and the portal is wrapped in `<div onClick>`, that `onClick` handler will fire. If this causes issues, either stop the event propagation from inside the portal, or move the portal itself up in the React tree.

Usage {/*usage*/}

Rendering to a different part of the DOM {/*rendering-to-a-different-part-of-the-dom*/}

Portals let your components render some of their children into a different place in the DOM. This lets a part of your component "escape" from whatever containers it may be in. For example, a component can display a modal dialog or a tooltip that appears above and outside of the rest of the page.

To create a portal, render the result of `createPortal` with <CodeStep step={1}>some JSX</CodeStep> and the <CodeStep step={2}>DOM node where it should go</CodeStep>:

```js [[1, 8, "<p>This child is placed in the document body.</p>"], [2, 9, "document.body"]]
import { createPortal } from 'react-dom';

function MyComponent() {
return (
<div style={{ border: '2px solid black' }}>
<p>This child is placed in the parent div.</p>
{createPortal(
<p>This child is placed in the document body.</p>,
document.body
)}
</div>
);
}
```

```

React will put the DOM nodes for <CodeStep step={1}>the JSX you passed</CodeStep> inside of the <CodeStep step={2}>DOM node you provided</CodeStep>.

Without a portal, the second `<p>` would be placed inside the parent `<div>`, but the portal "teleported" it into the [`document.body`:](https://developer.mozilla.org/en-US/docs/Web/API/Document/body)

<Sandpack>

```js
import { createPortal } from 'react-dom';

export default function MyComponent() {
return (
<div style={{ border: '2px solid black' }}>
<p>This child is placed in the parent div.</p>
{createPortal(
<p>This child is placed in the document body.</p>,
document.body
)}
</div>
);
}
```

</Sandpack>

Notice how the second paragraph visually appears outside the parent `<div>` with the border. If you inspect the DOM structure with developer tools, you'll see that the second `<p>` got placed directly into the `<body>`:

```html {4-6,9}
<body>
<div id="root">
...
<div style="border: 2px solid black">
<p>This child is placed inside the parent div.</p>
</div>
...
</div>
<p>This child is placed in the document body.</p>
```
```

</body>
```

A portal only changes the physical placement of the DOM node. In every other way, the JSX you render into a portal acts as a child node of the React component that renders it. For example, the child can access the context provided by the parent tree, and events still bubble up from children to parents according to the React tree.

---

### Rendering a modal dialog with a portal {/*rendering-a-modal-dialog-with-a-portal*/}

You can use a portal to create a modal dialog that floats above the rest of the page, even if the component that summons the dialog is inside a container with `overflow: hidden` or other styles that interfere with the dialog.

In this example, the two containers have styles that disrupt the modal dialog, but the one rendered into a portal is unaffected because, in the DOM, the modal is not contained within the parent JSX elements.

<Sandpack>

```js App.js active
import NoPortalExample from './NoPortalExample';
import PortalExample from './PortalExample';

export default function App() {
return (
<>
<div className="clipping-container">
<NoPortalExample />
</div>
<div className="clipping-container">
<PortalExample />
</div>
</>
);
}
```

```js NoPortalExample.js
import { useState } from 'react';
import ModalContent from './ModalContent.js';

export default function NoPortalExample() {

```
const [showModal, setShowModal] = useState(false);
return (
<>
<button onClick={() => setShowModal(true)}>
Show modal without a portal
</button>
{showModal && (
<ModalContent onClose={() => setShowModal(false)} />
)}
</>
);
}
```

```js PortalExample.js active
import { useState } from 'react';
import { createPortal } from 'react-dom';
import ModalContent from './ModalContent.js';

export default function PortalExample() {
const [showModal, setShowModal] = useState(false);
return (
<>
<button onClick={() => setShowModal(true)}>
Show modal using a portal
</button>
{showModal && createPortal(
<ModalContent onClose={() => setShowModal(false)} />,
document.body
)}
</>
);
}
```

```js ModalContent.js
export default function ModalContent({ onClose }) {
```

```
return (
<div className="modal">
<div>I'm a modal dialog</div>
<button onClick={onClose}>Close</button>
</div>
);
}
```

```css styles.css
.clipping-container {
position: relative;
border: 1px solid #aaa;
margin-bottom: 12px;
padding: 12px;
width: 250px;
height: 80px;
overflow: hidden;
}

.modal {
display: flex;
justify-content: space-evenly;
align-items: center;
box-shadow: rgba(100, 100, 111, 0.3) 0px 7px 29px 0px;
background-color: white;
border: 2px solid rgb(240, 240, 240);
border-radius: 12px;
position: absolute;
width: 250px;
top: 70px;
left: calc(50% - 125px);
bottom: 70px;
}
```

</Sandpack>

<Pitfall>

It's important to make sure that your app is accessible when using portals. For instance, you may need to manage keyboard focus so that the user can move the focus in and out of the portal in a natural way.

Follow the [WAI-ARIA Modal Authoring Practices](https://www.w3.org/WAI/ARIA/apg/#dialog_modal) when creating modals. If you use a community package, ensure that it is accessible and follows these guidelines.

</Pitfall>

Rendering React components into non-React server markup
{/*rendering-react-components-into-non-react-server-markup*/}

Portals can be useful if your React root is only part of a static or server-rendered page that isn't built with React. For example, if your page is built with a server framework like Rails, you can create areas of interactivity within static areas such as sidebars. Compared with having [multiple separate React roots,](/reference/react-dom/client/createRoot#rendering-a-page-partially-built-with-react) portals let you treat the app as a single React tree with shared state even though its parts render to different parts of the DOM.

<Sandpack>

```html index.html
<!DOCTYPE html>
<html>
<head><title>My app</title></head>
<body>
<h1>Welcome to my hybrid app</h1>
<div class="parent">
<div class="sidebar">
This is server non-React markup
<div id="sidebar-content"></div>
</div>
<div id="root"></div>
</div>
</body>
</html>
```

```js index.js
import { StrictMode } from 'react';
```

````
import { createRoot } from 'react-dom/client';
import App from './App.js';
import './styles.css';

const root = createRoot(document.getElementById('root'));
root.render(
<StrictMode>
<App />
</StrictMode>
);
```

```js App.js active
import { createPortal } from 'react-dom';

const sidebarContentEl = document.getElementById('sidebar-content');

export default function App() {
return (
<>
<MainContent />
{createPortal(
<SidebarContent />,
sidebarContentEl
)}
</>
);
}

function MainContent() {
return <p>This part is rendered by React</p>;
}

function SidebarContent() {
return <p>This part is also rendered by React!</p>;
}
```

```css
.parent {
````

```
display: flex;
flex-direction: row;
}

#root {
margin-top: 12px;
}

.sidebar {
padding: 12px;
background-color: #eee;
width: 200px;
height: 200px;
margin-right: 12px;
}

#sidebar-content {
margin-top: 18px;
display: block;
background-color: white;
}

p {
margin: 0;
}
```

</Sandpack>

Rendering React components into non-React DOM nodes
{/*rendering-react-components-into-non-react-dom-nodes*/}

You can also use a portal to manage the content of a DOM node that's managed outside of React. For example, suppose you're integrating with a non-React map widget and you want to render React content inside a popup. To do this, declare a `popupContainer` state variable to store the DOM node you're going to render into:

```js
const [popupContainer, setPopupContainer] = useState(null);
```

When you create the third-party widget, store the DOM node returned by the widget so you can render into it:

```js {5-6}
useEffect(() => {
if (mapRef.current === null) {
const map = createMapWidget(containerRef.current);
mapRef.current = map;
const popupDiv = addPopupToMapWidget(map);
setPopupContainer(popupDiv);
}
}, []);
```

This lets you use `createPortal` to render React content into `popupContainer` once it becomes available:

```js {3-6}
return (
<div style={{ width: 250, height: 250 }} ref={containerRef}>
{popupContainer !== null && createPortal(
<p>Hello from React!</p>,
popupContainer
)}
</div>
);
```

Here is a complete example you can play with:

<Sandpack>

```json package.json hidden
{
"dependencies": {
"leaflet": "1.9.1",
"react": "latest",
"react-dom": "latest",
"react-scripts": "latest",
"remarkable": "2.0.1"
```

```
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
}
}
```

```js App.js
import { useRef, useEffect, useState } from 'react';
import { createPortal } from 'react-dom';
import { createMapWidget, addPopupToMapWidget } from './map-widget.js';

export default function Map() {
const containerRef = useRef(null);
const mapRef = useRef(null);
const [popupContainer, setPopupContainer] = useState(null);

useEffect(() => {
if (mapRef.current === null) {
const map = createMapWidget(containerRef.current);
mapRef.current = map;
const popupDiv = addPopupToMapWidget(map);
setPopupContainer(popupDiv);
}
}, []);

return (
<div style={{ width: 250, height: 250 }} ref={containerRef}>
{popupContainer !== null && createPortal(
<p>Hello from React!</p>,
popupContainer
)}
</div>
);
}
```

```

```js map-widget.js
import 'leaflet/dist/leaflet.css';
import * as L from 'leaflet';

export function createMapWidget(containerDomNode) {
const map = L.map(containerDomNode);
map.setView([0, 0], 0);
L.tileLayer('https://tile.openstreetmap.org/{z}/{x}/{y}.png', {
maxZoom: 19,
attribution: '© OpenStreetMap'
}).addTo(map);
return map;
}

export function addPopupToMapWidget(map) {
const popupDiv = document.createElement('div');
L.popup()
.setLatLng([0, 0])
.setContent(popupDiv)
.openOn(map);
return popupDiv;
}
```

```css
button { margin: 5px; }
```

</Sandpack>

---

title: flushSync

<Pitfall>

Using `flushSync` is uncommon and can hurt the performance of your app.

</Pitfall>
```

<Intro>

`flushSync` lets you force React to flush any updates inside the provided callback synchronously. This ensures that the DOM is updated immediately.

```js
flushSync(callback)
```

</Intro>

<InlineToc />

Reference {/*reference*/}

`flushSync(callback)` {/*flushsync*/}

Call `flushSync` to force React to flush any pending work and update the DOM synchronously.

```js
import { flushSync } from 'react-dom';

flushSync(() => {
setSomething(123);
});
```

Most of the time, `flushSync` can be avoided. Use `flushSync` as last resort.

[See more examples below.](#usage)

Parameters {/*parameters*/}

* `callback`: A function. React will immediately call this callback and flush any updates it contains synchronously. It may also flush any pending updates, or Effects, or updates inside of Effects. If an update suspends as a result of this `flushSync` call, the fallbacks may be re-shown.

Returns {/*returns*/}

`flushSync` returns `undefined`.

Caveats {/*caveats*/}

* `flushSync` can significantly hurt performance. Use sparingly.
* `flushSync` may force pending Suspense boundaries to show their `fallback` state.

* `flushSync` may run pending effects and synchronously apply any updates they contain before returning.

* `flushSync` may flush updates outside the callback when necessary to flush the updates inside the callback. For example, if there are pending updates from a click, React may flush those before flushing the updates inside the callback.

Usage {/*usage*/}

Flushing updates for third-party integrations {/*flushing-updates-for-third-party-integrations*/}

When integrating with third-party code such as browser APIs or UI libraries, it may be necessary to force React to flush updates. Use `flushSync` to force React to flush any <CodeStep step={1}>state updates</CodeStep> inside the callback synchronously:

```js [[1, 2, "setSomething(123)"]]
flushSync(() => {
setSomething(123);
});
// By this line, the DOM is updated.
```

This ensures that, by the time the next line of code runs, React has already updated the DOM.

Using `flushSync` is uncommon, and using it often can significantly hurt the performance of your app. If your app only uses React APIs, and does not integrate with third-party libraries, `flushSync` should be unnecessary.

However, it can be helpful for integrating with third-party code like browser APIs.

Some browser APIs expect results inside of callbacks to be written to the DOM synchronously, by the end of the callback, so the browser can do something with the rendered DOM. In most cases, React handles this for you automatically. But in some cases it may be necessary to force a synchronous update.

For example, the browser `onbeforeprint` API allows you to change the page immediately before the print dialog opens. This is useful for applying custom print styles that allow the document to display better for printing. In the example below, you use `flushSync` inside of the `onbeforeprint` callback to immediately "flush" the React state to the DOM. Then, by the time the print dialog opens, `isPrinting` displays "yes":

<Sandpack>

```js App.js active
import { useState, useEffect } from 'react';
import { flushSync } from 'react-dom';

export default function PrintApp() {
```

```
const [isPrinting, setIsPrinting] = useState(false);

useEffect(() => {
function handleBeforePrint() {
flushSync(() => {
setIsPrinting(true);
})
}

function handleAfterPrint() {
setIsPrinting(false);
}

window.addEventListener('beforeprint', handleBeforePrint);
window.addEventListener('afterprint', handleAfterPrint);
return () => {
window.removeEventListener('beforeprint', handleBeforePrint);
window.removeEventListener('afterprint', handleAfterPrint);
}
}, []);

return (
<>
<h1>isPrinting: {isPrinting ? 'yes' : 'no'}</h1>
<button onClick={() => window.print()}>
Print
</button>
</>
);
}
```

</Sandpack>

Without `flushSync`, when the print dialog will display `isPrinting` as "no". This is because React batches the updates asynchronously and the print dialog is displayed before the state is updated.

<Pitfall>

`flushSync` can significantly hurt performance, and may unexpectedly force pending Suspense boundaries to show their fallback state.

Most of the time, `flushSync` can be avoided, so use `flushSync` as a last resort.

</Pitfall>

---

title: render

<Deprecated>

This API will be removed in a future major version of React.

In React 18, `render` was replaced by [`createRoot`.](/reference/react-dom/client/createRoot) Using `render` in React 18 will warn that your app will behave as if it's running React 17. Learn more [here.](/blog/2022/03/08/react-18-upgrade-guide#updates-to-client-rendering-apis)

</Deprecated>

<Intro>

`render` renders a piece of [JSX](/learn/writing-markup-with-jsx) ("React node") into a browser DOM node.

```js
render(reactNode, domNode, callback?)
```

</Intro>

<InlineToc />

Reference {/*reference*/}

`render(reactNode, domNode, callback?)` {/*render*/}

Call `render` to display a React component inside a browser DOM element.

```js
import { render } from 'react-dom';

const domNode = document.getElementById('root');

render(<App />, domNode);
```

React will display `<App />` in the `domNode`, and take over managing the DOM inside it.

An app fully built with React will usually only have one `render` call with its root component. A page that uses "sprinkles" of React for parts of the page may have as many `render` calls as needed.

[See more examples below.](#usage)

Parameters {/*parameters*/}

* `reactNode`: A *React node* that you want to display. This will usually be a piece of JSX like `<App />`, but you can also pass a React element constructed with [`createElement()`](/reference/react/createElement), a string, a number, `null`, or `undefined`.

* `domNode`: A [DOM element.](https://developer.mozilla.org/en-US/docs/Web/API/Element) React will display the `reactNode` you pass inside this DOM element. From this moment, React will manage the DOM inside the `domNode` and update it when your React tree changes.

* **optional** `callback`: A function. If passed, React will call it after your component is placed into the DOM.

Returns {/*returns*/}

`render` usually returns `null`. However, if the `reactNode` you pass is a *class component*, then it will return an instance of that component.

Caveats {/*caveats*/}

* In React 18, `render` was replaced by [`createRoot`.](/reference/react-dom/client/createRoot) Please use `createRoot` for React 18 and beyond.

* The first time you call `render`, React will clear all the existing HTML content inside the `domNode` before rendering the React component into it. If your `domNode` contains HTML generated by React on the server or during the build, use [`hydrate()`](/reference/react-dom/hydrate) instead, which attaches the event handlers to the existing HTML.

* If you call `render` on the same `domNode` more than once, React will update the DOM as necessary to reflect the latest JSX you passed. React will decide which parts of the DOM can be reused and which need to be recreated by ["matching it up"](/learn/preserving-and-resetting-state) with the previously rendered tree. Calling `render` on the same `domNode` again is similar to calling the [`set` function](/reference/react/useState#setstate) on the root component: React avoids unnecessary DOM updates.

* If your app is fully built with React, you'll likely have only one `render` call in your app. (If you use a framework, it might do this call for you.) When you want to render a piece of JSX in a different part of the DOM tree that isn't a child of your component (for example, a modal or a tooltip), use [`createPortal`](/reference/react-dom/createPortal) instead of `render`.

Usage {/*usage*/}

Call `render` to display a <CodeStep step={1}>React component</CodeStep> inside a <CodeStep step={2}>browser DOM node</CodeStep>.

```js [[1, 4, "<App />"], [2, 4, "document.getElementById('root')"]]
import { render } from 'react-dom';
```

````
import App from './App.js';

render(<App />, document.getElementById('root'));
```

### Rendering the root component {/*rendering-the-root-component*/}

In apps fully built with React, **you will usually only do this once at startup**--to render the "root" component.

<Sandpack>

```js index.js active
import './styles.css';
import { render } from 'react-dom';
import App from './App.js';

render(<App />, document.getElementById('root'));
```

```js App.js
export default function App() {
return <h1>Hello, world!</h1>;
}
```

</Sandpack>

Usually you shouldn't need to call `render` again or to call it in more places. From this point on, React will be managing the DOM of your application. To update the UI, your components will [use state.](/reference/react/useState)

---

### Rendering multiple roots {/*rendering-multiple-roots*/}

If your page [isn't fully built with React](/learn/add-react-to-an-existing-project#using-react-for-a-part-of-your-existing-page), call `render` for each top-level piece of UI managed by React.

<Sandpack>

```html public/index.html
<nav id="navigation"></nav>
<main>
<p>This paragraph is not rendered by React (open index.html to verify).</p>
````

```
<section id="comments"></section>
</main>
```

```js index.js active
import './styles.css';
import { render } from 'react-dom';
import { Comments, Navigation } from './Components.js';

render(
<Navigation />,
document.getElementById('navigation')
);

render(
<Comments />,
document.getElementById('comments')
);
```

```js Components.js
export function Navigation() {
return (
<ul>
<NavLink href="/">Home</NavLink>
<NavLink href="/about">About</NavLink>
</ul>
);
}

function NavLink({ href, children }) {
return (
<li>
<a href={href}>{children}</a>
</li>
);
}

export function Comments() {
```

return (

<>

<h2>Comments</h2>

<Comment text="Hello!" author="Sophie" />

<Comment text="How are you?" author="Sunil" />

</>

);

}

function Comment({ text, author }) {

return (

{text} — {author}

);

}

```

```css

nav ul { padding: 0; margin: 0; }

nav ul li { display: inline-block; margin-right: 20px; }

```

</Sandpack>

You can destroy the rendered trees with [`unmountComponentAtNode()`.](/reference/react-dom/unmountComponentAtNode)

Updating the rendered tree {/*updating-the-rendered-tree*/}

You can call `render` more than once on the same DOM node. As long as the component tree structure matches up with what was previously rendered, React will [preserve the state.](/learn/preserving-and-resetting-state) Notice how you can type in the input, which means that the updates from repeated `render` calls every second are not destructive:

<Sandpack>

```js index.js active

import { render } from 'react-dom';

import './styles.css';

import App from './App.js';

let i = 0;

setInterval(() => {
render(
<App counter={i} />,
document.getElementById('root')
);
i++;
}, 1000);
```

```js App.js
export default function App({counter}) {
return (
<>
<h1>Hello, world! {counter}</h1>
<input placeholder="Type something here" />
</>
);
}
```

</Sandpack>

It is uncommon to call `render` multiple times. Usually, you'll [update state](/reference/react/useState) inside your components instead.

---

title: hydrate

<Deprecated>

This API will be removed in a future major version of React.

In React 18, `hydrate` was replaced by [`hydrateRoot`.](/reference/react-dom/client/hydrateRoot) Using `hydrate` in React 18 will warn that your app will behave as if it’s running React 17. Learn more [here.](/blog/2022/03/08/react-18-upgrade-guide#updates-to-client-rendering-apis)

</Deprecated>

<Intro>

`hydrate` lets you display React components inside a browser DOM node whose HTML content was previously generated by [`react-dom/server`](/reference/react-dom/server) in React 17 and below.
```

```js
hydrate(reactNode, domNode, callback?)
```

</Intro>

<InlineToc />

Reference {/*reference*/}

`hydrate(reactNode, domNode, callback?)` {/*hydrate*/}

Call `hydrate` in React 17 and below to “attach” React to existing HTML that was already rendered by React in a server environment.

```js
import { hydrate } from 'react-dom';

hydrate(reactNode, domNode);
```

React will attach to the HTML that exists inside the `domNode`, and take over managing the DOM inside it. An app fully built with React will usually only have one `hydrate` call with its root component.

[See more examples below.](#usage)

Parameters {/*parameters*/}

* `reactNode`: The "React node" used to render the existing HTML. This will usually be a piece of JSX like `<App />` which was rendered with a `ReactDOM Server` method such as `renderToString(<App />)` in React 17.

* `domNode`: A [DOM element](https://developer.mozilla.org/en-US/docs/Web/API/Element) that was rendered as the root element on the server.

* **optional**: `callback`: A function. If passed, React will call it after your component is hydrated.

Returns {/*returns*/}

`hydrate` returns null.

Caveats {/*caveats*/}

* `hydrate` expects the rendered content to be identical with the server-rendered content. React 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.

* You'll likely have only one `hydrate` call in your app. If you use a framework, it might do this call for you.

* If your app is client-rendered with no HTML rendered already, using `hydrate()` is not supported. Use [render()](/reference/react-dom/render) (for React 17 and below) or [createRoot()](/reference/react-dom/client/createRoot) (for React 18+) instead.

Usage {/*usage*/}

Call `hydrate` to attach a <CodeStep step={1}>React component</CodeStep> into a server-rendered <CodeStep step={2}>browser DOM node</CodeStep>.

```js [[1, 3, "<App />"], [2, 3, "document.getElementById('root')"]]
import { hydrate } from 'react-dom';

hydrate(<App />, document.getElementById('root'));
```

Using `hydrate()` to render a client-only app (an app without server-rendered HTML) is not supported. Use [`render()`](/reference/react-dom/render) (in React 17 and below) or [`createRoot()`](/reference/react-dom/client/createRoot) (in React 18+) instead.

Hydrating server-rendered HTML {/*hydrating-server-rendered-html*/}

In React, "hydration" is how React "attaches" to existing HTML that was already rendered by React in a server environment. During hydration, React will attempt to attach event listeners to the existing markup and take over rendering the app on the client.

In apps fully built with React, **you will usually only hydrate one "root", once at startup for your entire app**.

<Sandpack>

```html public/index.html
<!--
HTML content inside <div id="root">...</div>
was generated from App by react-dom/server.
-->
<div id="root"><h1>Hello, world!</h1></div>
```

```js index.js active
import './styles.css';
import { hydrate } from 'react-dom';
import App from './App.js';
```

hydrate(<App />, document.getElementById('root'));
```

```js App.js
export default function App() {
return <h1>Hello, world!</h1>;
}
```

</Sandpack>

Usually you shouldn't need to call `hydrate` again or to call it in more places. From this point on, React will be managing the DOM of your application. To update the UI, your components will [use state.](/reference/react/useState)

For more information on hydration, see the docs for [`hydrateRoot`.](/reference/react-dom/client/hydrateRoot)

Suppressing unavoidable hydration mismatch errors {/*suppressing-unavoidable-hydration-mismatch-errors*/}

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 hydration mismatch warning.

To silence hydration warnings on an element, add `suppressHydrationWarning={true}`:

<Sandpack>

```html public/index.html
<!--
HTML content inside <div id="root">...</div>
was generated from App by react-dom/server.
-->
<div id="root"><h1>Current Date: 01/01/2020</h1></div>
```

```js index.js
import './styles.css';
import { hydrate } from 'react-dom';
import App from './App.js';

hydrate(<App />, document.getElementById('root'));
```
```

```js App.js active
export default function App() {
return (
<h1 suppressHydrationWarning={true}>
Current Date: {new Date().toLocaleDateString()}
</h1>
);
}
```

</Sandpack>

This 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.

Handling different client and server content {/*handling-different-client-and-server-content*/}

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](/reference/react/useState) like `isClient`, which you can set to `true` in an [effect](/reference/react/useEffect):

<Sandpack>

```html public/index.html
<!--
HTML content inside <div id="root">...</div>
was generated from App by react-dom/server.
-->
<div id="root"><h1>Is Server</h1></div>
```

```js index.js
import './styles.css';
import { hydrate } from 'react-dom';
import App from './App.js';

hydrate(<App />, document.getElementById('root'));
```

```js App.js active
```

```
import { useState, useEffect } from "react";

export default function App() {
const [isClient, setIsClient] = useState(false);

useEffect(() => {
setIsClient(true);
}, []);

return (
<h1>
{isClient ? 'Is Client' : 'Is Server'}
</h1>
);
}
```

</Sandpack>

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.

<Pitfall>

This approach makes hydration slower because your components have to render twice. Be mindful of the user experience on slow connections. The JavaScript code may load significantly later than the initial HTML render, so rendering a different UI immediately after hydration may feel jarring to the user.

</Pitfall>

---

title: unmountComponentAtNode

<Deprecated>

This API will be removed in a future major version of React.

In React 18, `unmountComponentAtNode` was replaced by [`root.unmount()`](/reference/react-dom/client/createRoot#root-unmount).

</Deprecated>

<Intro>

`unmountComponentAtNode` removes a mounted React component from the DOM.

```js
```

unmountComponentAtNode(domNode)
```

</Intro>

<InlineToc />

---

## Reference {/*reference*/}

### `unmountComponentAtNode(domNode)` {/*unmountcomponentatnode*/}

Call `unmountComponentAtNode` to remove a mounted React component from the DOM and clean up its event handlers and state.

```js
import { unmountComponentAtNode } from 'react-dom';

const domNode = document.getElementById('root');
render(<App />, domNode);

unmountComponentAtNode(domNode);
```

[See more examples below.](#usage)

Parameters {/*parameters*/}

* `domNode`: A [DOM element.](https://developer.mozilla.org/en-US/docs/Web/API/Element) React will remove a mounted React component from this element.

Returns {/*returns*/}

`unmountComponentAtNode` returns `true` if a component was unmounted and `false` otherwise.

Usage {/*usage*/}

Call `unmountComponentAtNode` to remove a <CodeStep step={1}>mounted React component</CodeStep> from a <CodeStep step={2}>browser DOM node</CodeStep> and clean up its event handlers and state.

```js [[1, 5, "<App />"], [2, 5, "rootNode"], [2, 8, "rootNode"]]
import { render, unmountComponentAtNode } from 'react-dom';
import App from './App.js';

const rootNode = document.getElementById('root');

```
render(<App />, rootNode);

// ...
unmountComponentAtNode(rootNode);
```

Removing a React app from a DOM element {/*removing-a-react-app-from-a-dom-element*/}

Occasionally, you may want to "sprinkle" React on an existing page, or a page that is not fully written in React. In those cases, you may need to "stop" the React app, by removing all of the UI, state, and listeners from the DOM node it was rendered to.

In this example, clicking "Render React App" will render a React app. Click "Unmount React App" to destroy it:

<Sandpack>

```html index.html
<!DOCTYPE html>
<html>
<head><title>My app</title></head>
<body>
<button id='render'>Render React App</button>
<button id='unmount'>Unmount React App</button>
<!-- This is the React App node -->
<div id='root'></div>
</body>
</html>
```

```js index.js active
import './styles.css';
import { render, unmountComponentAtNode } from 'react-dom';
import App from './App.js';

const domNode = document.getElementById('root');

document.getElementById('render').addEventListener('click', () => {
render(<App />, domNode);
});

document.getElementById('unmount').addEventListener('click', () => {
unmountComponentAtNode(domNode);
```

});
```

```js App.js
export default function App() {
return <h1>Hello, world!</h1>;
}
```

</Sandpack>

---
title: findDOMNode

<Deprecated>

This API will be removed in a future major version of React. [See the alternatives.](#alternatives)

</Deprecated>

<Intro>

`findDOMNode` finds the browser DOM node for a React [class component](/reference/react/Component) instance.

```js
const domNode = findDOMNode(componentInstance)
```

</Intro>

<InlineToc />

Reference {/*reference*/}

`findDOMNode(componentInstance)` {/*finddomnode*/}

Call `findDOMNode` to find the browser DOM node for a given React [class component](/reference/react/Component) instance.

```js
import { findDOMNode } from 'react-dom';

const domNode = findDOMNode(componentInstance);
```
```

[See more examples below.](#usage)

Parameters {/*parameters*/}

* `componentInstance`: An instance of the [`Component`](/reference/react/Component) subclass. For example, `this` inside a class component.

Returns {/*returns*/}

`findDOMNode` returns the first closest browser DOM node within the given `componentInstance`. When a component renders to `null`, or renders `false`, `findDOMNode` returns `null`. When a component renders to a string, `findDOMNode` returns a text DOM node containing that value.

Caveats {/*caveats*/}

* A component may return an array or a [Fragment](/reference/react/Fragment) with multiple children. In that case `findDOMNode`, will return the DOM node corresponding to the first non-empty child.

* `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` only returns the result at the time of your call. If a child component renders a different node later, there is no way for you to be notified of this change.

* `findDOMNode` accepts a class component instance, so it can't be used with function components.

Usage {/*usage*/}

Finding the root DOM node of a class component
{/*finding-the-root-dom-node-of-a-class-component*/}

Call `findDOMNode` with a [class component](/reference/react/Component) instance (usually, `this`) to find the DOM node it has rendered.

```js {3}
class AutoselectingInput extends Component {
componentDidMount() {
const input = findDOMNode(this);
input.select()
}
render() {
return <input defaultValue="Hello" />
}
}
```

```

Here, the `input` variable will be set to the `&lt;input&gt;` DOM element. This lets you do something with it. For example, when clicking "Show example" below mounts the input, [`input.select()`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLInputElement/select) selects all text in the input:

<Sandpack>

```js App.js
import { useState } from 'react';
import AutoselectingInput from './AutoselectingInput.js';

export default function App() {
const [show, setShow] = useState(false);
return (
<>
<button onClick={() => setShow(true)}>
Show example
</button>
<hr />
{show && <AutoselectingInput />}
</>
);
}
```

```js AutoselectingInput.js active
import { Component } from 'react';
import { findDOMNode } from 'react-dom';

class AutoselectingInput extends Component {
componentDidMount() {
const input = findDOMNode(this);
input.select()
}

render() {
return <input defaultValue="Hello" />
}
}
```
```

export default AutoselectingInput;
```

</Sandpack>

---

## Alternatives {/*alternatives*/}

### Reading component's own DOM node from a ref
{/*reading-components-own-dom-node-from-a-ref*/}

Code using `findDOMNode` is fragile because the connection between the JSX node and the code manipulating the corresponding DOM node is not explicit. For example, try wrapping this `<input />` into a `<div>`:

<Sandpack>

```js App.js
import { useState } from 'react';
import AutoselectingInput from './AutoselectingInput.js';

export default function App() {
const [show, setShow] = useState(false);
return (
<>
<button onClick={() => setShow(true)}>
Show example
</button>
<hr />
{show && <AutoselectingInput />}
</>
);
}
```

```js AutoselectingInput.js active
import { Component } from 'react';
import { findDOMNode } from 'react-dom';

class AutoselectingInput extends Component {
componentDidMount() {
const input = findDOMNode(this);

input.select()

}

render() {

return <input defaultValue="Hello" />

}

}

export default AutoselectingInput;

```

</Sandpack>

This will break the code because now, `findDOMNode(this)` finds the `<div>` DOM node, but the code expects an `<input>` DOM node. To avoid these kinds of problems, use [`createRef`](/reference/react/createRef) to manage a specific DOM node.

In this example, `findDOMNode` is no longer used. Instead, `inputRef = createRef(null)` is defined as an instance field on the class. To read the DOM node from it, you can use `this.inputRef.current`. To attach it to the JSX, you render `<input ref={this.inputRef} />`. This connects the code using the DOM node to its JSX:

<Sandpack>

```js App.js

import { useState } from 'react';

import AutoselectingInput from './AutoselectingInput.js';

export default function App() {

const [show, setShow] = useState(false);

return (

<>

<button onClick={() => setShow(true)}>

Show example

</button>

<hr />

{show && <AutoselectingInput />}

</>

);

}

```

```js AutoselectingInput.js active

import { createRef, Component } from 'react';

class AutoselectingInput extends Component {
inputRef = createRef(null);

componentDidMount() {
const input = this.inputRef.current;
input.select()
}

render() {
return (
<input ref={this.inputRef} defaultValue="Hello" />
);
}
}

export default AutoselectingInput;
```

</Sandpack>

In modern React without class components, the equivalent code would call [`useRef`](/reference/react/useRef) instead:

<Sandpack>

```js App.js
import { useState } from 'react';
import AutoselectingInput from './AutoselectingInput.js';

export default function App() {
const [show, setShow] = useState(false);
return (
<>
<button onClick={() => setShow(true)}>
Show example
</button>
<hr />
{show && <AutoselectingInput />}
</>
);

}
```

```js AutoselectingInput.js active
import { useRef, useEffect } from 'react';

export default function AutoselectingInput() {
const inputRef = useRef(null);

useEffect(() => {
const input = inputRef.current;
input.select();
}, []);

return <input ref={inputRef} defaultValue="Hello" />
}
```

</Sandpack>

[Read more about manipulating the DOM with refs.](/learn/manipulating-the-dom-with-refs)

Reading a child component's DOM node from a forwarded ref
{/*reading-a-child-components-dom-node-from-a-forwarded-ref*/}

In this example, `findDOMNode(this)` finds a DOM node that belongs to another component. The `AutoselectingInput` renders `MyInput`, which is your own component that renders a browser `<input>`.

<Sandpack>

```js App.js
import { useState } from 'react';
import AutoselectingInput from './AutoselectingInput.js';

export default function App() {
const [show, setShow] = useState(false);
return (
<>
<button onClick={() => setShow(true)}>
Show example
</button>
<hr />

```
{show && <AutoselectingInput />}
</>
);
}
```

```js AutoselectingInput.js active
import { Component } from 'react';
import { findDOMNode } from 'react-dom';
import MyInput from './MyInput.js';

class AutoselectingInput extends Component {
componentDidMount() {
const input = findDOMNode(this);
input.select()
}
render() {
return <MyInput />;
}
}

export default AutoselectingInput;
```

```js MyInput.js
export default function MyInput() {
return <input defaultValue="Hello" />;
}
```

</Sandpack>

Notice that calling `findDOMNode(this)` inside `AutoselectingInput` still gives you the DOM `<input>`--even though the JSX for this `<input>` is hidden inside the `MyInput` component. This seems convenient for the above example, but it leads to fragile code. Imagine that you wanted to edit `MyInput` later and add a wrapper `<div>` around it. This would break the code of `AutoselectingInput` (which expects to find an `<input>`).

To replace `findDOMNode` in this example, the two components need to coordinate:

1. `AutoSelectingInput` should declare a ref, like [in the earlier example](#reading-components-own-dom-node-from-a-ref), and pass it to `<MyInput>`.

2. `MyInput` should be declared with [`forwardRef`](/reference/react/forwardRef) to take that ref and forward it down to the `<input>` node.

This version does that, so it no longer needs `findDOMNode`:

<Sandpack>

```js App.js
import { useState } from 'react';
import AutoselectingInput from './AutoselectingInput.js';

export default function App() {
const [show, setShow] = useState(false);
return (
<>
<button onClick={() => setShow(true)}>
Show example
</button>
<hr />
{show && <AutoselectingInput />}
</>
);
}
```

```js AutoselectingInput.js active
import { createRef, Component } from 'react';
import MyInput from './MyInput.js';

class AutoselectingInput extends Component {
inputRef = createRef(null);

componentDidMount() {
const input = this.inputRef.current;
input.select()
}

render() {
return (
<MyInput ref={this.inputRef} />
);
```

```
}
}

export default AutoselectingInput;
```

```js MyInput.js
import { forwardRef } from 'react';

const MyInput = forwardRef(function MyInput(props, ref) {
return <input ref={ref} defaultValue="Hello" />;
});

export default MyInput;
```

</Sandpack>

Here is how this code would look like with function components instead of classes:

<Sandpack>

```js App.js
import { useState } from 'react';
import AutoselectingInput from './AutoselectingInput.js';

export default function App() {
const [show, setShow] = useState(false);
return (
<>
<button onClick={() => setShow(true)}>
Show example
</button>
<hr />
{show && <AutoselectingInput />}
</>
);
}
```

```js AutoselectingInput.js active
import { useRef, useEffect } from 'react';
```

```
import MyInput from './MyInput.js';

export default function AutoselectingInput() {
const inputRef = useRef(null);

useEffect(() => {
const input = inputRef.current;
input.select();
}, []);

return <MyInput ref={inputRef} defaultValue="Hello" />
}
```

```js MyInput.js
import { forwardRef } from 'react';

const MyInput = forwardRef(function MyInput(props, ref) {
return <input ref={ref} defaultValue="Hello" />;
});

export default MyInput;
```

</Sandpack>

Adding a wrapper `<div>` element {/*adding-a-wrapper-div-element*/}

Sometimes a component needs to know the position and size of its children. This makes it tempting to find the children with `findDOMNode(this)`, and then use DOM methods like [`getBoundingClientRect`](https://developer.mozilla.org/en-US/docs/Web/API/Element/getBoundingClientRect) for measurements.

There is currently no direct equivalent for this use case, which is why `findDOMNode` is deprecated but is not yet removed completely from React. In the meantime, you can try rendering a wrapper `<div>` node around the content as a workaround, and getting a ref to that node. However, extra wrappers can break styling.

```js
<div ref={someRef}>
{children}
</div>
```

This also applies to focusing and scrolling to arbitrary children.

---

title: "React DOM Components"

<Intro>

React supports all of the browser built-in
[HTML](https://developer.mozilla.org/en-US/docs/Web/HTML/Element) and
[SVG](https://developer.mozilla.org/en-US/docs/Web/SVG/Element) components.

</Intro>

Common components {/*common-components*/}

All of the built-in browser components support some props and events.

* [Common components (e.g. `<div>`)](/reference/react-dom/components/common)

This includes React-specific props like `ref` and `dangerouslySetInnerHTML`.

Form components {/*form-components*/}

These built-in browser components accept user input:

* [`<input>`](/reference/react-dom/components/input)
* [`<select>`](/reference/react-dom/components/select)
* [`<textarea>`](/reference/react-dom/components/textarea)

They are special in React because passing the `value` prop to them makes them
[controlled.](/reference/react-dom/components/input#controlling-an-input-with-a-state-variable)

All HTML components {/*all-html-components*/}

React supports all built-in browser HTML components. This includes:

* [`<aside>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/aside)
* [`<audio>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/audio)
* [`<b>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/b)
* [`<base>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/base)
* [`<bdi>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/bdi)
* [`<bdo>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/bdo)

* [`<blockquote>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/blockquote)
* [`<body>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/body)
* [`<br>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/br)
* [`<button>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/button)
* [`<canvas>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/canvas)
* [`<caption>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/caption)
* [`<cite>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/cite)
* [`<code>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/code)
* [`<col>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/col)
* [`<colgroup>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/colgroup)
* [`<data>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/data)
* [`<datalist>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/datalist)
* [`<dd>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/dd)
* [`<del>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/del)
* [`<details>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/details)
* [`<dfn>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/dfn)
* [`<dialog>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/dialog)
* [`<div>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/div)
* [`<dl>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/dl)
* [`<dt>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/dt)
* [`<em>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/em)
* [`<embed>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/embed)
* [`<fieldset>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/fieldset)
* [`<figcaption>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/figcaption)
* [`<figure>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/figure)
* [`<footer>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/footer)
* [`<form>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/form)
* [`<h1>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/h1)
* [`<head>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/head)
* [`<header>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/header)
* [`<hgroup>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/hgroup)
* [`<hr>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/hr)
* [`<html>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/html)
* [`<i>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/i)
* [`<iframe>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe)

* [`<img>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/img)
* [`<input>`](/reference/react-dom/components/input)
* [`<ins>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/ins)
* [`<kbd>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/kbd)
* [`<label>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/label)
* [`<legend>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/legend)
* [`<li>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/li)
* [`<link>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/link)
* [`<main>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/main)
* [`<map>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/map)
* [`<mark>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/mark)
* [`<menu>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/menu)
* [`<meta>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/meta)
* [`<meter>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/meter)
* [`<nav>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/nav)
* [`<noscript>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/noscript)
* [`<object>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/object)
* [`<ol>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/ol)
* [`<optgroup>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/optgroup)
* [`<option>`](/reference/react-dom/components/option)
* [`<output>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/output)
* [`<p>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/p)
* [`<picture>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/picture)
* [`<pre>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/pre)
* [`<progress>`](/reference/react-dom/components/progress)
* [`<q>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/q)
* [`<rp>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/rp)
* [`<rt>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/rt)
* [`<ruby>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/ruby)
* [`<s>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/s)
* [`<samp>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/samp)
* [`<script>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/script)
* [`<section>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/section)
* [`<select>`](/reference/react-dom/components/select)
* [`<slot>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/slot)

* [`<small>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/small)
* [`<source>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/source)
* [`<span>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/span)
* [`<strong>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/strong)
* [`<style>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/style)
* [`<sub>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/sub)
* [`<summary>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/summary)
* [`<sup>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/sup)
* [`<table>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/table)
* [`<tbody>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/tbody)
* [`<td>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/td)
* [`<template>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/template)
* [`<textarea>`](/reference/react-dom/components/textarea)
* [`<tfoot>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/tfoot)
* [`<th>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/th)
* [`<thead>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/thead)
* [`<time>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/time)
* [`<title>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/title)
* [`<tr>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/tr)
* [`<track>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/track)
* [`<u>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/u)
* [`<ul>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/ul)
* [`<var>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/var)
* [`<video>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/video)
* [`<wbr>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/wbr)

<Note>

Similar to the [DOM standard,](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model) React uses a `camelCase` convention for prop names. For example, you'll write `tabIndex` instead of `tabindex`. You can convert existing HTML to JSX with an [online converter.](https://transform.tools/html-to-jsx)

</Note>

Custom HTML elements {/*custom-html-elements*/}

If you render a tag with a dash, like `<my-element>`, React will assume you want to render a [custom HTML

element.](https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_custom_elements)
In React, rendering custom elements works differently from rendering built-in browser tags:

- All custom element props are serialized to strings and are always set using attributes.
- Custom elements accept `class` rather than `className`, and `for` rather than `htmlFor`.

If you render a built-in browser HTML element with an [`is`](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/is) attribute, it will also be treated as a custom element.

<Note>

[A future version of React will include more comprehensive support for custom elements.](https://github.com/facebook/react/issues/11347#issuecomment-1122275286)

You can try it by upgrading React packages to the most recent experimental version:

- `react@experimental`
- `react-dom@experimental`

Experimental versions of React may contain bugs. Don't use them in production.

</Note>

All SVG components {/*all-svg-components*/}

React supports all built-in browser SVG components. This includes:

* [`<a>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/a)
* [`<animate>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/animate)
* [`<animateMotion>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/animateMotion)
* [`<animateTransform>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/animateTransform)
* [`<circle>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/circle)
* [`<clipPath>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/clipPath)
* [`<defs>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/defs)
* [`<desc>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/desc)
* [`<discard>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/discard)
* [`<ellipse>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/ellipse)
* [`<feBlend>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/feBlend)
* [`<feColorMatrix>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/feColorMatrix)
* [`<feComponentTransfer>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/feComponentTransfer)
* [`<feComposite>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/feComposite)

*
[`<feConvolveMatrix>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/feConvolveMatrix)
* [`<feDiffuseLighting>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/feDiffuseLighting)
* [`<feDisplacementMap>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/feDisplacementMap)
* [`<feDistantLight>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/feDistantLight)
* [`<feDropShadow>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/feDropShadow)
* [`<feFlood>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/feFlood)
* [`<feFuncA>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/feFuncA)
* [`<feFuncB>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/feFuncB)
* [`<feFuncG>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/feFuncG)
* [`<feFuncR>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/feFuncR)
* [`<feGaussianBlur>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/feGaussianBlur)
* [`<feImage>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/feImage)
* [`<feMerge>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/feMerge)
* [`<feMergeNode>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/feMergeNode)
* [`<feMorphology>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/feMorphology)
* [`<feOffset>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/feOffset)
* [`<fePointLight>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/fePointLight)
* [`<feSpecularLighting>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/feSpecularLighting)
* [`<feSpotLight>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/feSpotLight)
* [`<feTile>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/feTile)
* [`<feTurbulence>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/feTurbulence)
* [`<filter>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/filter)
* [`<foreignObject>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/foreignObject)
* [`<g>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/g)
* `<hatch>`
* `<hatchpath>`
* [`<image>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/image)
* [`<line>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/line)
* [`<linearGradient>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/linearGradient)
* [`<marker>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/marker)
* [`<mask>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/mask)
* [`<metadata>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/metadata)
* [`<mpath>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/mpath)

* [`<path>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/path)
* [`<pattern>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/pattern)
* [`<polygon>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/polygon)
* [`<polyline>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/polyline)
* [`<radialGradient>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/radialGradient)
* [`<rect>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/rect)
* [`<script>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/script)
* [`<set>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/set)
* [`<stop>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/stop)
* [`<style>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/style)
* [`<svg>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/svg)
* [`<switch>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/switch)
* [`<symbol>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/symbol)
* [`<text>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/text)
* [`<textPath>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/textPath)
* [`<title>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/title)
* [`<tspan>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/tspan)
* [`<use>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/use)
* [`<view>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Element/view)

<Note>

Similar to the [DOM standard,](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model) React uses a `camelCase` convention for prop names. For example, you'll write `tabIndex` instead of `tabindex`. You can convert existing SVG to JSX with an [online converter.](https://transform.tools/)

Namespaced attributes also have to be written without the colon:

* `xlink:actuate` becomes `xlinkActuate`.
* `xlink:arcrole` becomes `xlinkArcrole`.
* `xlink:href` becomes `xlinkHref`.
* `xlink:role` becomes `xlinkRole`.
* `xlink:show` becomes `xlinkShow`.
* `xlink:title` becomes `xlinkTitle`.
* `xlink:type` becomes `xlinkType`.
* `xml:base` becomes `xmlBase`.
* `xml:lang` becomes `xmlLang`.
* `xml:space` becomes `xmlSpace`.

* `xmlns:xlink` becomes `xmlnsXlink`.

</Note>

---

title: "<option>"

<Intro>

The [built-in browser `<option>` component](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/option) lets you render an option inside a [`<select>`](/reference/react-dom/components/select) box.

```js
<select>
<option value="someOption">Some option</option>
<option value="otherOption">Other option</option>
</select>
```

</Intro>

<InlineToc />

Reference {/*reference*/}

`<option>` {/*option*/}

The [built-in browser `<option>` component](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/option) lets you render an option inside a [`<select>`](/reference/react-dom/components/select) box.

```js
<select>
<option value="someOption">Some option</option>
<option value="otherOption">Other option</option>
</select>
```

[See more examples below.](#usage)

Props {/*props*/}

`<option>` supports all [common element props.](/reference/react-dom/components/common#props)

Additionally, `<option>` supports these props:

* [`disabled`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/option#disabled): A boolean. If `true`, the option will not be selectable and will appear dimmed.

* [`label`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/option#label): A string. Specifies the meaning of the option. If not specified, the text inside the option is used.

* [`value`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/option#value): The value to be used [when submitting the parent `<select>` in a form](/reference/react-dom/components/select#reading-the-select-box-value-when-submitting-a-form) if this option is selected.

Caveats {/*caveats*/}

* React does not support the `selected` attribute on `<option>`. Instead, pass this option's `value` to the parent [`<select defaultValue>`](/reference/react-dom/components/select#providing-an-initially-selected-option) for an uncontrolled select box, or [`<select value>`](/reference/react-dom/components/select#controlling-a-select-box-with-a-state-variable) for a controlled select.

Usage {/*usage*/}

Displaying a select box with options {/*displaying-a-select-box-with-options*/}

Render a `<select>` with a list of `<option>` components inside to display a select box. Give each `<option>` a `value` representing the data to be submitted with the form.

[Read more about displaying a `<select>` with a list of `<option>` components.](/reference/react-dom/components/select)

<Sandpack>

```js
export default function FruitPicker() {
return (
<label>
Pick a fruit:
<select name="selectedFruit">
<option value="apple">Apple</option>
<option value="banana">Banana</option>
<option value="orange">Orange</option>
</select>
</label>
);
```

}
```

```css
select { margin: 5px; }
```

</Sandpack>

---
title: "Common components (e.g. <div>)"

<Intro>

All built-in browser components, such as [`<div>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/div), support some common props and events.

</Intro>

<InlineToc />

Reference {/*reference*/}

Common components (e.g. `<div>`) {/*common*/}

```js
<div className="wrapper">Some content</div>
```

[See more examples below.](#usage)

Props {/*common-props*/}

These special React props are supported for all built-in components:

* `children`: A React node (an element, a string, a number, [a portal,](/reference/react-dom/createPortal) an empty node like `null`, `undefined` and booleans, or an array of other React nodes). Specifies the content inside the component. When you use JSX, you will usually specify the `children` prop implicitly by nesting tags like `<div><span /></div>`.

* `dangerouslySetInnerHTML`: An object of the form `{ __html: '<p>some html</p>' }` with a raw HTML string inside. Overrides the [`innerHTML`](https://developer.mozilla.org/en-US/docs/Web/API/Element/innerHTML) property of the DOM node and displays the passed HTML inside. This should be used with extreme caution! If the HTML inside isn't trusted (for example, if it's based on user data), you risk introducing an
```

[XSS](https://en.wikipedia.org/wiki/Cross-site_scripting) vulnerability. [Read more about using `dangerouslySetInnerHTML`.](#dangerously-setting-the-inner-html)

* `ref`: A ref object from [`useRef`](/reference/react/useRef) or [`createRef`](/reference/react/createRef), or a [`ref` callback function,](#ref-callback) or a string for [legacy refs.](https://reactjs.org/docs/refs-and-the-dom.html#legacy-api-string-refs) Your ref will be filled with the DOM element for this node. [Read more about manipulating the DOM with refs.](#manipulating-a-dom-node-with-a-ref)

* `suppressContentEditableWarning`: A boolean. If `true`, suppresses the warning that React shows for elements that both have `children` and `contentEditable={true}` (which normally do not work together). Use this if you're building a text input library that manages the `contentEditable` content manually.

* `suppressHydrationWarning`: A boolean. If you use [server rendering,](/reference/react-dom/server) normally there is a warning when the server and the client render different content. In some rare cases (like timestamps), it is very hard or impossible to guarantee an exact match. 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. [Read about suppressing hydration errors.](/reference/react-dom/client/hydrateRoot#suppressing-unavoidable-hydration-mismatch-errors)

* `style`: An object with CSS styles, for example `{ fontWeight: 'bold', margin: 20 }`. Similarly to the DOM [`style`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/style) property, the CSS property names need to be written as `camelCase`, for example `fontWeight` instead of `font-weight`. You can pass strings or numbers as values. If you pass a number, like `width: 100`, React will automatically append `px` ("pixels") to the value unless it's a [unitless property.](https://github.com/facebook/react/blob/81d4ee9ca5c405dce62f64e61506b8e155f38d8d/packages/react-dom-bindings/src/shared/CSSProperty.js#L8-L57) We recommend using `style` only for dynamic styles where you don't know the style values ahead of time. In other cases, applying plain CSS classes with `className` is more efficient. [Read more about `className` and `style`.](#applying-css-styles)

These standard DOM props are also supported for all built-in components:

* [`accessKey`](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/accesskey): A string. Specifies a keyboard shortcut for the element. [Not generally recommended.](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/accesskey#accessibility_concerns)

* [`aria-*`](https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/Attributes): ARIA attributes let you specify the accessibility tree information for this element. See [ARIA attributes](https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/Attributes) for a complete reference. In React, all ARIA attribute names are exactly the same as in HTML.

* [`autoCapitalize`](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/autocapitalize): A string. Specifies whether and how the user input should be capitalized.

* [`className`](https://developer.mozilla.org/en-US/docs/Web/API/Element/className): A string. Specifies the element's CSS class name. [Read more about applying CSS styles.](#applying-css-styles)

* [`contentEditable`](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/contenteditable): A boolean. If `true`, the browser lets the user edit the rendered element directly. This is used to implement rich text input libraries like [Lexical.](https://lexical.dev/) React warns if you try to pass React children to an element with `contentEditable={true}` because React will not be able to update its content after user edits.

* [`data-*`](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/data-*): Data attributes let you attach some string data to the element, for example `data-fruit="banana"`. In React, they are not commonly used because you would usually read data from props or state instead.

* [`dir`](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/dir): Either `'ltr'` or `'rtl'`. Specifies the text direction of the element.

* [`draggable`](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/draggable): A boolean. Specifies whether the element is draggable. Part of [HTML Drag and Drop API.](https://developer.mozilla.org/en-US/docs/Web/API/HTML_Drag_and_Drop_API)

* [`enterKeyHint`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/enterKeyHint): A string. Specifies which action to present for the enter key on virtual keyboards.

* [`htmlFor`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLLabelElement/htmlFor): A string. For [`<label>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/label) and [`<output>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/output), lets you [associate the label with some control.](/reference/react-dom/components/input#providing-a-label-for-an-input) Same as [`for` HTML attribute.](https://developer.mozilla.org/en-US/docs/Web/HTML/Attributes/for) React uses the standard DOM property names (`htmlFor`) instead of HTML attribute names.

* [`hidden`](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/hidden): A boolean or a string. Specifies whether the element should be hidden.

* [`id`](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/id): A string. Specifies a unique identifier for this element, which can be used to find it later or connect it with other elements. Generate it with [`useId`](/reference/react/useId) to avoid clashes between multiple instances of the same component.

* [`is`](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/is): A string. If specified, the component will behave like a [custom element.](/reference/react-dom/components#custom-html-elements)

* [`inputMode`](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/inputmode): A string. Specifies what kind of keyboard to display (for example, text, number or telephone).

* [`itemProp`](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/itemprop): A string. Specifies which property the element represents for structured data crawlers.

* [`lang`](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/lang): A string. Specifies the language of the element.

* [`onAnimationEnd`](https://developer.mozilla.org/en-US/docs/Web/API/Element/animationend_event): An [`AnimationEvent` handler](#animationevent-handler) function. Fires when a CSS animation completes.

* `onAnimationEndCapture`: A version of `onAnimationEnd` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onAnimationIteration`](https://developer.mozilla.org/en-US/docs/Web/API/Element/animationiteration_event): An [`AnimationEvent` handler](#animationevent-handler) function. Fires when an iteration of a CSS animation ends, and another one begins.

* `onAnimationIterationCapture`: A version of `onAnimationIteration` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onAnimationStart`](https://developer.mozilla.org/en-US/docs/Web/API/Element/animationstart_event): An [`AnimationEvent` handler](#animationevent-handler) function. Fires when a CSS animation starts.

* `onAnimationStartCapture`: `onAnimationStart`, but fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onAuxClick`](https://developer.mozilla.org/en-US/docs/Web/API/Element/auxclick_event): A [`MouseEvent` handler](#mouseevent-handler) function. Fires when a non-primary pointer button was clicked.

* `onAuxClickCapture`: A version of `onAuxClick` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* `onBeforeInput`: An [`InputEvent` handler](#inputevent-handler) function. Fires before the value of an editable element is modified. React does *not* yet use the native [`beforeinput`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/beforeinput_event) event, and instead attempts to polyfill it using other events.

* `onBeforeInputCapture`: A version of `onBeforeInput` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* `onBlur`: A [`FocusEvent` handler](#focusevent-handler) function. Fires when an element lost focus. Unlike the built-in browser [`blur`](https://developer.mozilla.org/en-US/docs/Web/API/Element/blur_event) event, in React the `onBlur` event bubbles.

* `onBlurCapture`: A version of `onBlur` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onClick`](https://developer.mozilla.org/en-US/docs/Web/API/Element/click_event): A [`MouseEvent` handler](#mouseevent-handler) function. Fires when the primary button was clicked on the pointing device.

* `onClickCapture`: A version of `onClick` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onCompositionStart`](https://developer.mozilla.org/en-US/docs/Web/API/Element/compositionstart_event): A [`CompositionEvent` handler](#compositionevent-handler) function. Fires when an [input method editor](https://developer.mozilla.org/en-US/docs/Glossary/Input_method_editor) starts a new composition session.

* `onCompositionStartCapture`: A version of `onCompositionStart` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onCompositionEnd`](https://developer.mozilla.org/en-US/docs/Web/API/Element/compositionend_event): A [`CompositionEvent` handler](#compositionevent-handler) function. Fires when an [input method editor](https://developer.mozilla.org/en-US/docs/Glossary/Input_method_editor) completes or cancels a composition session.

* `onCompositionEndCapture`: A version of `onCompositionEnd` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onCompositionUpdate`](https://developer.mozilla.org/en-US/docs/Web/API/Element/compositionupdate_event): A [`CompositionEvent` handler](#compositionevent-handler) function. Fires when an [input method editor](https://developer.mozilla.org/en-US/docs/Glossary/Input_method_editor) receives a new character.

* `onCompositionUpdateCapture`: A version of `onCompositionUpdate` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onContextMenu`](https://developer.mozilla.org/en-US/docs/Web/API/Element/contextmenu_event): A [`MouseEvent` handler](#mouseevent-handler) function. Fires when the user tries to open a context menu.

* `onContextMenuCapture`: A version of `onContextMenu` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onCopy`](https://developer.mozilla.org/en-US/docs/Web/API/Element/copy_event): A [`ClipboardEvent` handler](#clipboardevent-handler) function. Fires when the user tries to copy something into the clipboard.

* `onCopyCapture`: A version of `onCopy` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onCut`](https://developer.mozilla.org/en-US/docs/Web/API/Element/cut_event): A [`ClipboardEvent` handler](#clipboardevent-handler) function. Fires when the user tries to cut something into the clipboard.

* `onCutCapture`: A version of `onCut` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* `onDoubleClick`: A [`MouseEvent` handler](#mouseevent-handler) function. Fires when the user clicks twice. Corresponds to the browser [`dblclick` event.](https://developer.mozilla.org/en-US/docs/Web/API/Element/dblclick_event)

* `onDoubleClickCapture`: A version of `onDoubleClick` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onDrag`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/drag_event): A [`DragEvent` handler](#dragevent-handler) function. Fires while the user is dragging something.

* `onDragCapture`: A version of `onDrag` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onDragEnd`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/dragend_event): A [`DragEvent` handler](#dragevent-handler) function. Fires when the user stops dragging something.

* `onDragEndCapture`: A version of `onDragEnd` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onDragEnter`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/dragenter_event): A [`DragEvent` handler](#dragevent-handler) function. Fires when the dragged content enters a valid drop target.

* `onDragEnterCapture`: A version of `onDragEnter` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onDragOver`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/dragover_event): A [`DragEvent` handler](#dragevent-handler) function. Fires on a valid drop target while the dragged content is dragged over it. You must call `e.preventDefault()` here to allow dropping.

* `onDragOverCapture`: A version of `onDragOver` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onDragStart`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/dragstart_event): A [`DragEvent` handler](#dragevent-handler) function. Fires when the user starts dragging an element.

* `onDragStartCapture`: A version of `onDragStart` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onDrop`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/drop_event): A [`DragEvent` handler](#dragevent-handler) function. Fires when something is dropped on a valid drop target.

* `onDropCapture`: A version of `onDrop` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* `onFocus`: A [`FocusEvent` handler](#focusevent-handler) function. Fires when an element receives focus. Unlike the built-in browser [`focus`](https://developer.mozilla.org/en-US/docs/Web/API/Element/focus_event) event, in React the `onFocus` event bubbles.

* `onFocusCapture`: A version of `onFocus` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onGotPointerCapture`](https://developer.mozilla.org/en-US/docs/Web/API/Element/gotpointercapture_event): A [`PointerEvent` handler](#pointerevent-handler) function. Fires when an element programmatically captures a pointer.

* `onGotPointerCaptureCapture`: A version of `onGotPointerCapture` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onKeyDown`](https://developer.mozilla.org/en-US/docs/Web/API/Element/keydown_event): A [`KeyboardEvent` handler](#pointerevent-handler) function. Fires when a key is pressed.

* `onKeyDownCapture`: A version of `onKeyDown` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onKeyPress`](https://developer.mozilla.org/en-US/docs/Web/API/Element/keypress_event): A [`KeyboardEvent` handler](#pointerevent-handler) function. Deprecated. Use `onKeyDown` or `onBeforeInput` instead.

* `onKeyPressCapture`: A version of `onKeyPress` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onKeyUp`](https://developer.mozilla.org/en-US/docs/Web/API/Element/keyup_event): A [`KeyboardEvent` handler](#pointerevent-handler) function. Fires when a key is released.

* `onKeyUpCapture`: A version of `onKeyUp` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onLostPointerCapture`](https://developer.mozilla.org/en-US/docs/Web/API/Element/lostpointercapture_event): A [`PointerEvent` handler](#pointerevent-handler) function. Fires when an element stops capturing a pointer.

* `onLostPointerCaptureCapture`: A version of `onLostPointerCapture` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onMouseDown`](https://developer.mozilla.org/en-US/docs/Web/API/Element/mousedown_event): A [`MouseEvent` handler](#mouseevent-handler) function. Fires when the pointer is pressed down.

* `onMouseDownCapture`: A version of `onMouseDown` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onMouseEnter`](https://developer.mozilla.org/en-US/docs/Web/API/Element/mouseenter_event): A [`MouseEvent` handler](#mouseevent-handler) function. Fires when the pointer moves inside an element. Does not have a capture phase. Instead, `onMouseLeave` and `onMouseEnter` propagate from the element being left to the one being entered.

* [`onMouseLeave`](https://developer.mozilla.org/en-US/docs/Web/API/Element/mouseleave_event): A [`MouseEvent` handler](#mouseevent-handler) function. Fires when the pointer moves outside an element. Does not have a capture phase. Instead, `onMouseLeave` and `onMouseEnter` propagate from the element being left to the one being entered.

* [`onMouseMove`](https://developer.mozilla.org/en-US/docs/Web/API/Element/mousemove_event): A [`MouseEvent` handler](#mouseevent-handler) function. Fires when the pointer changes coordinates.

* `onMouseMoveCapture`: A version of `onMouseMove` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onMouseOut`](https://developer.mozilla.org/en-US/docs/Web/API/Element/mouseout_event): A [`MouseEvent` handler](#mouseevent-handler) function. Fires when the pointer moves outside an element, or if it moves into a child element.

* `onMouseOutCapture`: A version of `onMouseOut` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onMouseUp`](https://developer.mozilla.org/en-US/docs/Web/API/Element/mouseup_event): A [`MouseEvent` handler](#mouseevent-handler) function. Fires when the pointer is released.

* `onMouseUpCapture`: A version of `onMouseUp` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onPointerCancel`](https://developer.mozilla.org/en-US/docs/Web/API/Element/pointercancel_event): A [`PointerEvent` handler](#pointerevent-handler) function. Fires when the browser cancels a pointer interaction.

* `onPointerCancelCapture`: A version of `onPointerCancel` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onPointerDown`](https://developer.mozilla.org/en-US/docs/Web/API/Element/pointerdown_event): A [`PointerEvent` handler](#pointerevent-handler) function. Fires when a pointer becomes active.

* `onPointerDownCapture`: A version of `onPointerDown` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onPointerEnter`](https://developer.mozilla.org/en-US/docs/Web/API/Element/pointerenter_event): A [`PointerEvent` handler](#pointerevent-handler) function. Fires when a pointer moves inside an element. Does not have a capture phase. Instead, `onPointerLeave` and `onPointerEnter` propagate from the element being left to the one being entered.

* [`onPointerLeave`](https://developer.mozilla.org/en-US/docs/Web/API/Element/pointerleave_event): A [`PointerEvent` handler](#pointerevent-handler) function. Fires when a pointer moves outside an element. Does not have a capture phase. Instead, `onPointerLeave` and `onPointerEnter` propagate from the element being left to the one being entered.

* [`onPointerMove`](https://developer.mozilla.org/en-US/docs/Web/API/Element/pointermove_event): A [`PointerEvent` handler](#pointerevent-handler) function. Fires when a pointer changes coordinates.

* `onPointerMoveCapture`: A version of `onPointerMove` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onPointerOut`](https://developer.mozilla.org/en-US/docs/Web/API/Element/pointerout_event): A [`PointerEvent` handler](#pointerevent-handler) function. Fires when a pointer moves outside an element, if the pointer interaction is cancelled, and [a few other reasons.](https://developer.mozilla.org/en-US/docs/Web/API/Element/pointerout_event)

* `onPointerOutCapture`: A version of `onPointerOut` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onPointerUp`](https://developer.mozilla.org/en-US/docs/Web/API/Element/pointerup_event): A [`PointerEvent` handler](#pointerevent-handler) function. Fires when a pointer is no longer active.

* `onPointerUpCapture`: A version of `onPointerUp` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onPaste`](https://developer.mozilla.org/en-US/docs/Web/API/Element/paste_event): A [`ClipboardEvent` handler](#clipboardevent-handler) function. Fires when the user tries to paste something from the clipboard.

* `onPasteCapture`: A version of `onPaste` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onScroll`](https://developer.mozilla.org/en-US/docs/Web/API/Element/scroll_event): An [`Event` handler](#event-handler) function. Fires when an element has been scrolled. This event does not bubble.

* `onScrollCapture`: A version of `onScroll` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onSelect`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLInputElement/select_event): An [`Event` handler](#event-handler) function. Fires after the selection inside an editable element like an input changes. React extends the `onSelect` event to work for `contentEditable={true}` elements as well. In addition, React extends it to fire for empty selection and on edits (which may affect the selection).

* `onSelectCapture`: A version of `onSelect` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onTouchCancel`](https://developer.mozilla.org/en-US/docs/Web/API/Element/touchcancel_event): A [`TouchEvent` handler](#touchevent-handler) function. Fires when the browser cancels a touch interaction.

* `onTouchCancelCapture`: A version of `onTouchCancel` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onTouchEnd`](https://developer.mozilla.org/en-US/docs/Web/API/Element/touchend_event): A [`TouchEvent` handler](#touchevent-handler) function. Fires when one or more touch points are removed.

* `onTouchEndCapture`: A version of `onTouchEnd` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onTouchMove`](https://developer.mozilla.org/en-US/docs/Web/API/Element/touchmove_event): A [`TouchEvent` handler](#touchevent-handler) function. Fires one or more touch points are moved.

* `onTouchMoveCapture`: A version of `onTouchMove` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onTouchStart`](https://developer.mozilla.org/en-US/docs/Web/API/Element/touchstart_event): A [`TouchEvent` handler](#touchevent-handler) function. Fires when one or more touch points are placed.

* `onTouchStartCapture`: A version of `onTouchStart` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onTransitionEnd`](https://developer.mozilla.org/en-US/docs/Web/API/Element/transitionend_event): A [`TransitionEvent` handler](#transitionevent-handler) function. Fires when a CSS transition completes.

* `onTransitionEndCapture`: A version of `onTransitionEnd` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onWheel`](https://developer.mozilla.org/en-US/docs/Web/API/Element/wheel_event): A [`WheelEvent` handler](#wheelevent-handler) function. Fires when the user rotates a wheel button.

* `onWheelCapture`: A version of `onWheel` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`role`](https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/Roles): A string. Specifies the element role explicitly for assistive technologies.

* [`slot`](https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/Roles): A string. Specifies the slot name when using shadow DOM. In React, an equivalent pattern is typically achieved by passing JSX as props, for example `<Layout left={<Sidebar />} right={<Content />} />`.

* [`spellCheck`](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/spellcheck): A boolean or null. If explicitly set to `true` or `false`, enables or disables spellchecking.

* [`tabIndex`](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/tabindex): A number. Overrides the default Tab button behavior. [Avoid using values other than `-1` and `0`.](https://www.tpgi.com/using-the-tabindex-attribute/)

* [`title`](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/title): A string. Specifies the tooltip text for the element.

* [`translate`](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/translate): Either `'yes'` or `'no'`. Passing `'no'` excludes the element content from being translated.

You can also pass custom attributes as props, for example `mycustomprop="someValue"`. This can be useful when integrating with third-party libraries. The custom attribute name must be lowercase and must not start with `on`. The value will be converted to a string. If you pass `null` or `undefined`, the custom attribute will be removed.

These events fire only for the [`<form>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/form) elements:

* [`onReset`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLFormElement/reset_event): An [`Event` handler](#event-handler) function. Fires when a form gets reset.

* `onResetCapture`: A version of `onReset` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onSubmit`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLFormElement/submit_event): An [`Event` handler](#event-handler) function. Fires when a form gets submitted.

* `onSubmitCapture`: A version of `onSubmit` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

These events fire only for the [`<dialog>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/dialog) elements. Unlike browser events, they bubble in React:

* [`onCancel`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLDialogElement/cancel_event): An [`Event` handler](#event-handler) function. Fires when the user tries to dismiss the dialog.

* `onCancelCapture`: A version of `onCancel` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onClose`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLDialogElement/close_event): An [`Event` handler](#event-handler) function. Fires when a dialog has been closed.

* `onCloseCapture`: A version of `onClose` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

These events fire only for the [`<details>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/details) elements. Unlike

browser events, they bubble in React:

* [`onToggle`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLDetailsElement/toggle_event): An [`Event` handler](#event-handler) function. Fires when the user toggles the details.

* `onToggleCapture`: A version of `onToggle` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

These events fire for [`<img>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/img), [`<iframe>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe), [`<object>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/object), [`<embed>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/embed), [`<link>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/link), and [SVG `<image>`](https://developer.mozilla.org/en-US/docs/Web/SVG/Tutorial/SVG_Image_Tag) elements. Unlike browser events, they bubble in React:

* `onLoad`: An [`Event` handler](#event-handler) function. Fires when the resource has loaded.

* `onLoadCapture`: A version of `onLoad` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onError`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement/error_event): An [`Event` handler](#event-handler) function. Fires when the resource could not be loaded.

* `onErrorCapture`: A version of `onError` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

These events fire for resources like [`<audio>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/audio) and [`<video>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/video). Unlike browser events, they bubble in React:

* [`onAbort`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement/abort_event): An [`Event` handler](#event-handler) function. Fires when the resource has not fully loaded, but not due to an error.

* `onAbortCapture`: A version of `onAbort` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onCanPlay`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement/canplay_event): An [`Event` handler](#event-handler) function. Fires when there's enough data to start playing, but not enough to play to the end without buffering.

* `onCanPlayCapture`: A version of `onCanPlay` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onCanPlayThrough`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement/canplaythrough_event): An [`Event` handler](#event-handler) function. Fires when there's enough data that it's likely possible to start playing without buffering until the end.

* `onCanPlayThroughCapture`: A version of `onCanPlayThrough` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onDurationChange`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement/durationchange_event): An [`Event` handler](#event-handler) function. Fires when the media duration has updated.

* `onDurationChangeCapture`: A version of `onDurationChange` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onEmptied`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement/emptied_event): An [`Event` handler](#event-handler) function. Fires when the media has become empty.

* `onEmptiedCapture`: A version of `onEmptied` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onEncrypted`](https://w3c.github.io/encrypted-media/#dom-evt-encrypted): An [`Event` handler](#event-handler) function. Fires when the browser encounters encrypted media.

* `onEncryptedCapture`: A version of `onEncrypted` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onEnded`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement/ended_event): An [`Event` handler](#event-handler) function. Fires when the playback stops because there's nothing left to play.

* `onEndedCapture`: A version of `onEnded` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onError`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement/error_event): An [`Event` handler](#event-handler) function. Fires when the resource could not be loaded.

* `onErrorCapture`: A version of `onError` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onLoadedData`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement/loadeddata_event): An [`Event` handler](#event-handler) function. Fires when the current playback frame has loaded.

* `onLoadedDataCapture`: A version of `onLoadedData` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onLoadedMetadata`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement/loadedmetadata_event): An [`Event` handler](#event-handler) function. Fires when metadata has loaded.

* `onLoadedMetadataCapture`: A version of `onLoadedMetadata` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onLoadStart`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement/loadstart_event): An [`Event` handler](#event-handler) function. Fires when the browser started loading the resource.

* `onLoadStartCapture`: A version of `onLoadStart` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onPause`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement/pause_event): An [`Event` handler](#event-handler) function. Fires when the media was paused.

* `onPauseCapture`: A version of `onPause` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onPlay`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement/play_event): An [`Event` handler](#event-handler) function. Fires when the media is no longer paused.

* `onPlayCapture`: A version of `onPlay` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onPlaying`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement/playing_event): An [`Event` handler](#event-handler) function. Fires when the media starts or restarts playing.

* `onPlayingCapture`: A version of `onPlaying` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onProgress`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement/progress_event): An [`Event` handler](#event-handler) function. Fires periodically while the resource is loading.

* `onProgressCapture`: A version of `onProgress` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onRateChange`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement/ratechange_event): An [`Event` handler](#event-handler) function. Fires when playback rate changes.

* `onRateChangeCapture`: A version of `onRateChange` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* `onResize`: An [`Event` handler](#event-handler) function. Fires when video changes size.

* `onResizeCapture`: A version of `onResize` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onSeeked`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement/seeked_event): An [`Event` handler](#event-handler) function. Fires when a seek operation completes.

* `onSeekedCapture`: A version of `onSeeked` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onSeeking`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement/seeking_event): An [`Event` handler](#event-handler) function. Fires when a seek operation starts.

* `onSeekingCapture`: A version of `onSeeking` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onStalled`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement/stalled_event): An [`Event` handler](#event-handler) function. Fires when the browser is waiting for data but it keeps not loading.

* `onStalledCapture`: A version of `onStalled` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onSuspend`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement/suspend_event): An [`Event` handler](#event-handler) function. Fires when loading the resource was suspended.

* `onSuspendCapture`: A version of `onSuspend` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onTimeUpdate`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement/timeupdate_event): An [`Event` handler](#event-handler) function. Fires when the current playback time updates.

* `onTimeUpdateCapture`: A version of `onTimeUpdate` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onVolumeChange`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement/volumechange_event): An [`Event` handler](#event-handler) function. Fires when the volume has changed.

* `onVolumeChangeCapture`: A version of `onVolumeChange` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onWaiting`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLMediaElement/waiting_event): An [`Event` handler](#event-handler) function. Fires when the playback stopped due to temporary lack of data.

* `onWaitingCapture`: A version of `onWaiting` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

Caveats {/*common-caveats*/}

- You cannot pass both `children` and `dangerouslySetInnerHTML` at the same time.
- Some events (like `onAbort` and `onLoad`) don't bubble in the browser, but bubble in React.

`ref` callback function {/*ref-callback*/}

Instead of a ref object (like the one returned by [`useRef`](/reference/react/useRef#manipulating-the-dom-with-a-ref)), you may pass a function to the `ref` attribute.

```js
<div ref={(node) => console.log(node)} />
```

[See an example of using the `ref` callback.](/learn/manipulating-the-dom-with-refs#how-to-manage-a-list-of-refs-using-a-ref-callback)

When the `<div>` DOM node is added to the screen, React will call your `ref` callback with the DOM `node` as the argument. When that `<div>` DOM node is removed, React will call your `ref` callback with `null`.

React will also call your `ref` callback whenever you pass a *different* `ref` callback. In the above example, `(node) => { ... }` is a different function on every render. When your component re-renders, the *previous* function will be called with `null` as the argument, and the *next* function will be called with the DOM node.

Parameters {/*ref-callback-parameters*/}

* `node`: A DOM node or `null`. React will pass you the DOM node when the ref gets attached, and `null` when the ref gets detached. Unless you pass the same function reference for the `ref` callback on every render, the callback will get temporarily detached and re-attached during every re-render of the component.

Returns {/*returns*/}

Do not return anything from the `ref` callback.

React event object {/*react-event-object*/}

Your event handlers will receive a *React event object.* It is also sometimes known as a "synthetic event".

```js
```

<button onClick={e => {

console.log(e); // React event object

}} />

```

It conforms to the same standard as the underlying DOM events, but fixes some browser inconsistencies.

Some React events do not map directly to the browser's native events. For example in `onMouseLeave`, `e.nativeEvent` will point to a `mouseout` event. The specific mapping is not part of the public API and may change in the future. If you need the underlying browser event for some reason, read it from `e.nativeEvent`.

#### Properties {/*react-event-object-properties*/}

React event objects implement some of the standard [`Event`](https://developer.mozilla.org/en-US/docs/Web/API/Event) properties:

* [`bubbles`](https://developer.mozilla.org/en-US/docs/Web/API/Event/bubbles): A boolean. Returns whether the event bubbles through the DOM.

* [`cancelable`](https://developer.mozilla.org/en-US/docs/Web/API/Event/cancelable): A boolean. Returns whether the event can be canceled.

* [`currentTarget`](https://developer.mozilla.org/en-US/docs/Web/API/Event/currentTarget): A DOM node. Returns the node to which the current handler is attached in the React tree.

* [`defaultPrevented`](https://developer.mozilla.org/en-US/docs/Web/API/Event/defaultPrevented): A boolean. Returns whether `preventDefault` was called.

* [`eventPhase`](https://developer.mozilla.org/en-US/docs/Web/API/Event/eventPhase): A number. Returns which phase the event is currently in.

* [`isTrusted`](https://developer.mozilla.org/en-US/docs/Web/API/Event/isTrusted): A boolean. Returns whether the event was initiated by user.

* [`target`](https://developer.mozilla.org/en-US/docs/Web/API/Event/target): A DOM node. Returns the node on which the event has occurred (which could be a distant child).

* [`timeStamp`](https://developer.mozilla.org/en-US/docs/Web/API/Event/timeStamp): A number. Returns the time when the event occurred.

Additionally, React event objects provide these properties:

* `nativeEvent`: A DOM [`Event`](https://developer.mozilla.org/en-US/docs/Web/API/Event). The original browser event object.

#### Methods {/*react-event-object-methods*/}

React event objects implement some of the standard [`Event`](https://developer.mozilla.org/en-US/docs/Web/API/Event) methods:

* [`preventDefault()`](https://developer.mozilla.org/en-US/docs/Web/API/Event/preventDefault): Prevents the default browser action for the event.
```

* [`stopPropagation()`](https://developer.mozilla.org/en-US/docs/Web/API/Event/stopPropagation): Stops the event propagation through the React tree.

Additionally, React event objects provide these methods:

* `isDefaultPrevented()`: Returns a boolean value indicating whether `preventDefault` was called.

* `isPropagationStopped()`: Returns a boolean value indicating whether `stopPropagation` was called.

* `persist()`: Not used with React DOM. With React Native, call this to read event's properties after the event.

* `isPersistent()`: Not used with React DOM. With React Native, returns whether `persist` has been called.

Caveats {/*react-event-object-caveats*/}

* The values of `currentTarget`, `eventPhase`, `target`, and `type` reflect the values your React code expects. Under the hood, React attaches event handlers at the root, but this is not reflected in React event objects. For example, `e.currentTarget` may not be the same as the underlying `e.nativeEvent.currentTarget`. For polyfilled events, `e.type` (React event type) may differ from `e.nativeEvent.type` (underlying type).

`AnimationEvent` handler function {/*animationevent-handler*/}

An event handler type for the [CSS animation](https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Animations/Using_CSS_animations) events.

```js
<div
onAnimationStart={e => console.log('onAnimationStart')}
onAnimationIteration={e => console.log('onAnimationIteration')}
onAnimationEnd={e => console.log('onAnimationEnd')}
/>
```

Parameters {/*animationevent-handler-parameters*/}

* `e`: A [React event object](#react-event-object) with these extra [`AnimationEvent`](https://developer.mozilla.org/en-US/docs/Web/API/AnimationEvent) properties:

* [`animationName`](https://developer.mozilla.org/en-US/docs/Web/API/AnimationEvent/animationName)

* [`elapsedTime`](https://developer.mozilla.org/en-US/docs/Web/API/AnimationEvent/elapsedTime)

* [`pseudoElement`](https://developer.mozilla.org/en-US/docs/Web/API/AnimationEvent)

`ClipboardEvent` handler function {/*clipboadevent-handler*/}

An event handler type for the [Clipboard API](https://developer.mozilla.org/en-US/docs/Web/API/Clipboard_API) events.

```js
<input
onCopy={e => console.log('onCopy')}
onCut={e => console.log('onCut')}
onPaste={e => console.log('onPaste')}
/>
```

Parameters {/*clipboadevent-handler-parameters*/}

* `e`: A [React event object](#react-event-object) with these extra [`ClipboardEvent`](https://developer.mozilla.org/en-US/docs/Web/API/ClipboardEvent) properties:

* [`clipboardData`](https://developer.mozilla.org/en-US/docs/Web/API/ClipboardEvent/clipboardData)

`CompositionEvent` handler function {/*compositionevent-handler*/}

An event handler type for the [input method editor (IME)](https://developer.mozilla.org/en-US/docs/Glossary/Input_method_editor) events.

```js
<input
onCompositionStart={e => console.log('onCompositionStart')}
onCompositionUpdate={e => console.log('onCompositionUpdate')}
onCompositionEnd={e => console.log('onCompositionEnd')}
/>
```

Parameters {/*compositionevent-handler-parameters*/}

* `e`: A [React event object](#react-event-object) with these extra [`CompositionEvent`](https://developer.mozilla.org/en-US/docs/Web/API/CompositionEvent) properties:

* [`data`](https://developer.mozilla.org/en-US/docs/Web/API/CompositionEvent/data)

`DragEvent` handler function {/*dragevent-handler*/}

An event handler type for the [HTML Drag and Drop API](https://developer.mozilla.org/en-US/docs/Web/API/HTML_Drag_and_Drop_API) events.

```js
<>
<div
draggable={true}
onDragStart={e => console.log('onDragStart')}
onDragEnd={e => console.log('onDragEnd')}
>
Drag source
</div>

<div
onDragEnter={e => console.log('onDragEnter')}
onDragLeave={e => console.log('onDragLeave')}
onDragOver={e => { e.preventDefault(); console.log('onDragOver'); }}
onDrop={e => console.log('onDrop')}
>
Drop target
</div>
</>
```

Parameters {/*dragevent-handler-parameters*/}

* `e`: A [React event object](#react-event-object) with these extra [`DragEvent`](https://developer.mozilla.org/en-US/docs/Web/API/DragEvent) properties:
* [`dataTransfer`](https://developer.mozilla.org/en-US/docs/Web/API/DragEvent/dataTransfer)

It also includes the inherited [`MouseEvent`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent) properties:

* [`altKey`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/altKey)
* [`button`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/button)
* [`buttons`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/buttons)
* [`ctrlKey`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/ctrlKey)
* [`clientX`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/clientX)
* [`clientY`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/clientY)
* [`getModifierState(key)`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/getModifierState)
* [`metaKey`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/metaKey)
* [`movementX`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/movementX)

* [`movementY`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/movementY)
* [`pageX`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/pageX)
* [`pageY`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/pageY)
* [`relatedTarget`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/relatedTarget)
* [`screenX`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/screenX)
* [`screenY`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/screenY)
* [`shiftKey`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/shiftKey)

It also includes the inherited [`UIEvent`](https://developer.mozilla.org/en-US/docs/Web/API/UIEvent) properties:

* [`detail`](https://developer.mozilla.org/en-US/docs/Web/API/UIEvent/detail)
* [`view`](https://developer.mozilla.org/en-US/docs/Web/API/UIEvent/view)

`FocusEvent` handler function {/*focusevent-handler*/}

An event handler type for the focus events.

```js
<input
onFocus={e => console.log('onFocus')}
onBlur={e => console.log('onBlur')}
/>
```

[See an example.](#handling-focus-events)

Parameters {/*focusevent-handler-parameters*/}

* `e`: A [React event object](#react-event-object) with these extra [`FocusEvent`](https://developer.mozilla.org/en-US/docs/Web/API/FocusEvent) properties:
* [`relatedTarget`](https://developer.mozilla.org/en-US/docs/Web/API/FocusEvent/relatedTarget)

It also includes the inherited [`UIEvent`](https://developer.mozilla.org/en-US/docs/Web/API/UIEvent) properties:

* [`detail`](https://developer.mozilla.org/en-US/docs/Web/API/UIEvent/detail)
* [`view`](https://developer.mozilla.org/en-US/docs/Web/API/UIEvent/view)

`Event` handler function {/*event-handler*/}

An event handler type for generic events.

Parameters {/*event-handler-parameters*/}

* `e`: A [React event object](#react-event-object) with no additional properties.

`InputEvent` handler function {/*inputevent-handler*/}

An event handler type for the `onBeforeInput` event.

```js
<input onBeforeInput={e => console.log('onBeforeInput')} />
```

Parameters {/*inputevent-handler-parameters*/}

* `e`: A [React event object](#react-event-object) with these extra [`InputEvent`](https://developer.mozilla.org/en-US/docs/Web/API/InputEvent) properties:
* [`data`](https://developer.mozilla.org/en-US/docs/Web/API/InputEvent/data)

`KeyboardEvent` handler function {/*keyboardevent-handler*/}

An event handler type for keyboard events.

```js
<input
onKeyDown={e => console.log('onKeyDown')}
onKeyUp={e => console.log('onKeyUp')}
/>
```

[See an example.](#handling-keyboard-events)

Parameters {/*keyboardevent-handler-parameters*/}

* `e`: A [React event object](#react-event-object) with these extra [`KeyboardEvent`](https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent) properties:
* [`altKey`](https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent/altKey)
* [`charCode`](https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent/charCode)
* [`code`](https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent/code)
* [`ctrlKey`](https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent/ctrlKey)
* [`getModifierState(key)`](https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent/getModifierState)
* [`key`](https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent/key)

* [`keyCode`](https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent/keyCode)
* [`locale`](https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent/locale)
* [`metaKey`](https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent/metaKey)
* [`location`](https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent/location)
* [`repeat`](https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent/repeat)
* [`shiftKey`](https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent/shiftKey)
* [`which`](https://developer.mozilla.org/en-US/docs/Web/API/KeyboardEvent/which)

It also includes the inherited [`UIEvent`](https://developer.mozilla.org/en-US/docs/Web/API/UIEvent) properties:

* [`detail`](https://developer.mozilla.org/en-US/docs/Web/API/UIEvent/detail)
* [`view`](https://developer.mozilla.org/en-US/docs/Web/API/UIEvent/view)

`MouseEvent` handler function {/*mouseevent-handler*/}

An event handler type for mouse events.

```js
<div
onClick={e => console.log('onClick')}
onMouseEnter={e => console.log('onMouseEnter')}
onMouseOver={e => console.log('onMouseOver')}
onMouseDown={e => console.log('onMouseDown')}
onMouseUp={e => console.log('onMouseUp')}
onMouseLeave={e => console.log('onMouseLeave')}
/>
```

[See an example.](#handling-mouse-events)

Parameters {/*mouseevent-handler-parameters*/}

* `e`: A [React event object](#react-event-object) with these extra [`MouseEvent`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent) properties:
* [`altKey`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/altKey)
* [`button`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/button)
* [`buttons`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/buttons)
* [`ctrlKey`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/ctrlKey)
* [`clientX`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/clientX)

* [`clientY`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/clientY)
* [`getModifierState(key)`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/getModifierState)
* [`metaKey`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/metaKey)
* [`movementX`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/movementX)
* [`movementY`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/movementY)
* [`pageX`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/pageX)
* [`pageY`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/pageY)
* [`relatedTarget`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/relatedTarget)
* [`screenX`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/screenX)
* [`screenY`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/screenY)
* [`shiftKey`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/shiftKey)

It also includes the inherited [`UIEvent`](https://developer.mozilla.org/en-US/docs/Web/API/UIEvent) properties:

* [`detail`](https://developer.mozilla.org/en-US/docs/Web/API/UIEvent/detail)
* [`view`](https://developer.mozilla.org/en-US/docs/Web/API/UIEvent/view)

`PointerEvent` handler function {/*pointerevent-handler*/}

An event handler type for [pointer events.](https://developer.mozilla.org/en-US/docs/Web/API/Pointer_events)

```js
<div
onPointerEnter={e => console.log('onPointerEnter')}
onPointerMove={e => console.log('onPointerMove')}
onPointerDown={e => console.log('onPointerDown')}
onPointerUp={e => console.log('onPointerUp')}
onPointerLeave={e => console.log('onPointerLeave')}
/>
```

[See an example.](#handling-pointer-events)

Parameters {/*pointerevent-handler-parameters*/}

* `e`: A [React event object](#react-event-object) with these extra [`PointerEvent`](https://developer.mozilla.org/en-US/docs/Web/API/PointerEvent) properties:
* [`height`](https://developer.mozilla.org/en-US/docs/Web/API/PointerEvent/height)

* [`isPrimary`](https://developer.mozilla.org/en-US/docs/Web/API/PointerEvent/isPrimary)
* [`pointerId`](https://developer.mozilla.org/en-US/docs/Web/API/PointerEvent/pointerId)
* [`pointerType`](https://developer.mozilla.org/en-US/docs/Web/API/PointerEvent/pointerType)
* [`pressure`](https://developer.mozilla.org/en-US/docs/Web/API/PointerEvent/pressure)
* [`tangentialPressure`](https://developer.mozilla.org/en-US/docs/Web/API/PointerEvent/tangentialPressure)
* [`tiltX`](https://developer.mozilla.org/en-US/docs/Web/API/PointerEvent/tiltX)
* [`tiltY`](https://developer.mozilla.org/en-US/docs/Web/API/PointerEvent/tiltY)
* [`twist`](https://developer.mozilla.org/en-US/docs/Web/API/PointerEvent/twist)
* [`width`](https://developer.mozilla.org/en-US/docs/Web/API/PointerEvent/width)

It also includes the inherited [`MouseEvent`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent) properties:

* [`altKey`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/altKey)
* [`button`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/button)
* [`buttons`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/buttons)
* [`ctrlKey`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/ctrlKey)
* [`clientX`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/clientX)
* [`clientY`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/clientY)
* [`getModifierState(key)`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/getModifierState)
* [`metaKey`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/metaKey)
* [`movementX`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/movementX)
* [`movementY`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/movementY)
* [`pageX`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/pageX)
* [`pageY`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/pageY)
* [`relatedTarget`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/relatedTarget)
* [`screenX`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/screenX)
* [`screenY`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/screenY)
* [`shiftKey`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/shiftKey)

It also includes the inherited [`UIEvent`](https://developer.mozilla.org/en-US/docs/Web/API/UIEvent) properties:

* [`detail`](https://developer.mozilla.org/en-US/docs/Web/API/UIEvent/detail)
* [`view`](https://developer.mozilla.org/en-US/docs/Web/API/UIEvent/view)

`TouchEvent` handler function {/*touchevent-handler*/}

An event handler type for [touch events.](https://developer.mozilla.org/en-US/docs/Web/API/Touch_events)

```js
<div
onTouchStart={e => console.log('onTouchStart')}
onTouchMove={e => console.log('onTouchMove')}
onTouchEnd={e => console.log('onTouchEnd')}
onTouchCancel={e => console.log('onTouchCancel')}
/>
```

Parameters {/*touchevent-handler-parameters*/}

* `e`: A [React event object](#react-event-object) with these extra [`TouchEvent`](https://developer.mozilla.org/en-US/docs/Web/API/TouchEvent) properties:
* [`altKey`](https://developer.mozilla.org/en-US/docs/Web/API/TouchEvent/altKey)
* [`ctrlKey`](https://developer.mozilla.org/en-US/docs/Web/API/TouchEvent/ctrlKey)
* [`changedTouches`](https://developer.mozilla.org/en-US/docs/Web/API/TouchEvent/changedTouches)
* [`getModifierState(key)`](https://developer.mozilla.org/en-US/docs/Web/API/TouchEvent/getModifierState)
* [`metaKey`](https://developer.mozilla.org/en-US/docs/Web/API/TouchEvent/metaKey)
* [`shiftKey`](https://developer.mozilla.org/en-US/docs/Web/API/TouchEvent/shiftKey)
* [`touches`](https://developer.mozilla.org/en-US/docs/Web/API/TouchEvent/touches)
* [`targetTouches`](https://developer.mozilla.org/en-US/docs/Web/API/TouchEvent/targetTouches)

It also includes the inherited [`UIEvent`](https://developer.mozilla.org/en-US/docs/Web/API/UIEvent) properties:

* [`detail`](https://developer.mozilla.org/en-US/docs/Web/API/UIEvent/detail)
* [`view`](https://developer.mozilla.org/en-US/docs/Web/API/UIEvent/view)

`TransitionEvent` handler function {/*transitionevent-handler*/}

An event handler type for the CSS transition events.

```js
<div
onTransitionEnd={e => console.log('onTransitionEnd')}
/>
```

```

#### Parameters {/*transitionevent-handler-parameters*/}

* `e`: A [React event object](#react-event-object) with these extra [`TransitionEvent`](https://developer.mozilla.org/en-US/docs/Web/API/TransitionEvent) properties:

* [`elapsedTime`](https://developer.mozilla.org/en-US/docs/Web/API/TransitionEvent/elapsedTime)

* [`propertyName`](https://developer.mozilla.org/en-US/docs/Web/API/TransitionEvent/propertyName)

* [`pseudoElement`](https://developer.mozilla.org/en-US/docs/Web/API/TransitionEvent/pseudoElement)

---

### `UIEvent` handler function {/*uievent-handler*/}

An event handler type for generic UI events.

```js
<div
onScroll={e => console.log('onScroll')}
/>
```

Parameters {/*uievent-handler-parameters*/}

* `e`: A [React event object](#react-event-object) with these extra [`UIEvent`](https://developer.mozilla.org/en-US/docs/Web/API/UIEvent) properties:

* [`detail`](https://developer.mozilla.org/en-US/docs/Web/API/UIEvent/detail)

* [`view`](https://developer.mozilla.org/en-US/docs/Web/API/UIEvent/view)

`WheelEvent` handler function {/*wheelevent-handler*/}

An event handler type for the `onWheel` event.

```js
<div
onScroll={e => console.log('onScroll')}
/>
```

Parameters {/*wheelevent-handler-parameters*/}

* `e`: A [React event object](#react-event-object) with these extra [`WheelEvent`](https://developer.mozilla.org/en-US/docs/Web/API/WheelEvent) properties:
```

* [`deltaMode`](https://developer.mozilla.org/en-US/docs/Web/API/WheelEvent/deltaMode)
* [`deltaX`](https://developer.mozilla.org/en-US/docs/Web/API/WheelEvent/deltaX)
* [`deltaY`](https://developer.mozilla.org/en-US/docs/Web/API/WheelEvent/deltaY)
* [`deltaZ`](https://developer.mozilla.org/en-US/docs/Web/API/WheelEvent/deltaZ)

It also includes the inherited [`MouseEvent`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent) properties:

* [`altKey`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/altKey)
* [`button`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/button)
* [`buttons`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/buttons)
* [`ctrlKey`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/ctrlKey)
* [`clientX`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/clientX)
* [`clientY`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/clientY)
* [`getModifierState(key)`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/getModifierState)
* [`metaKey`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/metaKey)
* [`movementX`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/movementX)
* [`movementY`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/movementY)
* [`pageX`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/pageX)
* [`pageY`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/pageY)
* [`relatedTarget`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/relatedTarget)
* [`screenX`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/screenX)
* [`screenY`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/screenY)
* [`shiftKey`](https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/shiftKey)

It also includes the inherited [`UIEvent`](https://developer.mozilla.org/en-US/docs/Web/API/UIEvent) properties:

* [`detail`](https://developer.mozilla.org/en-US/docs/Web/API/UIEvent/detail)
* [`view`](https://developer.mozilla.org/en-US/docs/Web/API/UIEvent/view)

Usage {/*usage*/}

Applying CSS styles {/*applying-css-styles*/}

In React, you specify a CSS class with [`className`.](https://developer.mozilla.org/en-US/docs/Web/API/Element/className) It works like the `class` attribute in HTML:

```js
```


```

Then you write the CSS rules for it in a separate CSS file:

```css
/* In your CSS */
.avatar {
border-radius: 50%;
}
```

React does not prescribe how you add CSS files. In the simplest case, you'll add a [`<link>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/link) tag to your HTML. If you use a build tool or a framework, consult its documentation to learn how to add a CSS file to your project.

Sometimes, the style values depend on data. Use the `style` attribute to pass some styles dynamically:

```js {3-6}
<img
className="avatar"
style={{
width: user.imageSize,
height: user.imageSize
}}
/>
```

In the above example, `style={{}}` is not a special syntax, but a regular `{}` object inside the `style={ }` [JSX curly braces.](/learn/javascript-in-jsx-with-curly-braces) We recommend only using the `style` attribute when your styles depend on JavaScript variables.

<Sandpack>

```js App.js
import Avatar from './Avatar.js';

const user = {
name: 'Hedy Lamarr',
imageUrl: 'https://i.imgur.com/yXOvdOSs.jpg',
imageSize: 90,
};
```
```

````
export default function App() {
return <Avatar user={user} />;
}
```

```js Avatar.js active
export default function Avatar({ user }) {
return (
<img
src={user.imageUrl}
alt={'Photo of ' + user.name}
className="avatar"
style={{
width: user.imageSize,
height: user.imageSize
}}
/>
);
}
```

```css styles.css
.avatar {
border-radius: 50%;
}
```

</Sandpack>

<DeepDive>

#### How to apply multiple CSS classes conditionally?
{/*how-to-apply-multiple-css-classes-conditionally*/}

To apply CSS classes conditionally, you need to produce the `className` string yourself using JavaScript.

For example, `className={'row ' + (isSelected ? 'selected': '')}` will produce either `className="row"` or `className="row selected"` depending on whether `isSelected` is `true`.

To make this more readable, you can use a tiny helper library like [`classnames`:](https://github.com/JedWatson/classnames)

```js
import cn from 'classnames';

function Row({ isSelected }) {
return (
<div className={cn('row', isSelected && 'selected')}>
...
</div>
);
}
```

It is especially convenient if you have multiple conditional classes:

```js
import cn from 'classnames';

function Row({ isSelected, size }) {
return (
<div className={cn('row', {
selected: isSelected,
large: size === 'large',
small: size === 'small',
})}>
...
</div>
);
}
```

</DeepDive>

---

### Manipulating a DOM node with a ref {/*manipulating-a-dom-node-with-a-ref*/}

Sometimes, you'll need to get the browser DOM node associated with a tag in JSX. For example, if you want to focus an `<input>` when a button is clicked, you need to call [`focus()`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/focus) on the browser `<input>` DOM node.

To obtain the browser DOM node for a tag, [declare a ref](/reference/react/useRef) and pass it as the `ref` attribute to that tag:

```js {7}
import { useRef } from 'react';

export default function Form() {
const inputRef = useRef(null);
// ...
return (
<input ref={inputRef} />
// ...
```

React will put the DOM node into `inputRef.current` after it's been rendered to the screen.

<Sandpack>

```js
import { useRef } from 'react';

export default function Form() {
const inputRef = useRef(null);

function handleClick() {
inputRef.current.focus();
}

return (
<>
<input ref={inputRef} />
<button onClick={handleClick}>
Focus the input
</button>
</>
);
}
```

</Sandpack>

Read more about [manipulating DOM with refs](/learn/manipulating-the-dom-with-refs) and [check out more examples.](/reference/react/useRef#examples-dom)

For more advanced use cases, the `ref` attribute also accepts a [callback function.](#ref-callback)

---

### Dangerously setting the inner HTML {/*dangerously-setting-the-inner-html*/}

You can pass a raw HTML string to an element like so:

```js
const markup = { __html: '<p>some raw html</p>' };
return <div dangerouslySetInnerHTML={markup} />;
```

**This is dangerous. As with the underlying DOM [`innerHTML`](https://developer.mozilla.org/en-US/docs/Web/API/Element/innerHTML) property, you must exercise extreme caution! Unless the markup is coming from a completely trusted source, it is trivial to introduce an [XSS](https://en.wikipedia.org/wiki/Cross-site_scripting) vulnerability this way.**

For example, if you use a Markdown library that converts Markdown to HTML, you trust that its parser doesn't contain bugs, and the user only sees their own input, you can display the resulting HTML like this:

<Sandpack>

```js
import { useState } from 'react';
import MarkdownPreview from './MarkdownPreview.js';

export default function MarkdownEditor() {
const [postContent, setPostContent] = useState('_Hello,_ **Markdown**!');
return (
<>
<label>
Enter some markdown:
<textarea
value={postContent}
onChange={e => setPostContent(e.target.value)}
/>
</label>
<hr />
<MarkdownPreview markdown={postContent} />
</>
);
}
```

```

```js MarkdownPreview.js active
import { Remarkable } from 'remarkable';

const md = new Remarkable();

function renderMarkdownToHTML(markdown) {
// This is ONLY safe because the output HTML
// is shown to the same user, and because you
// trust this Markdown parser to not have bugs.
const renderedHTML = md.render(markdown);
return {__html: renderedHTML};
}

export default function MarkdownPreview({ markdown }) {
const markup = renderMarkdownToHTML(markdown);
return <div dangerouslySetInnerHTML={markup} />;
}
```

```json package.json
{
"dependencies": {
"react": "latest",
"react-dom": "latest",
"react-scripts": "latest",
"remarkable": "2.0.1"
},
"scripts": {
"start": "react-scripts start",
"build": "react-scripts build",
"test": "react-scripts test --env=jsdom",
"eject": "react-scripts eject"
}
}
```

```css

textarea { display: block; margin-top: 5px; margin-bottom: 10px; }
```

</Sandpack>

To see why rendering arbitrary HTML is dangerous, replace the code above with this:

```js {1-4,7,8}
const post = {
// Imagine this content is stored in the database.
content: `<img src="" onerror='alert("you were hacked")'>`
};

export default function MarkdownPreview() {
// ■ SECURITY HOLE: passing untrusted input to dangerouslySetInnerHTML
const markup = { __html: post.content };
return <div dangerouslySetInnerHTML={markup} />;
}
```

The code embedded in the HTML will run. A hacker could use this security hole to steal user information or to perform actions on their behalf. **Only use `dangerouslySetInnerHTML` with trusted and sanitized data.**

---

### Handling mouse events {/*handling-mouse-events*/}

This example shows some common [mouse events](#mouseevent-handler) and when they fire.

<Sandpack>

```js
export default function MouseExample() {
return (
<div
onMouseEnter={e => console.log('onMouseEnter (parent)')}
onMouseLeave={e => console.log('onMouseLeave (parent)')}
>
<button
onClick={e => console.log('onClick (first button)')}
onMouseDown={e => console.log('onMouseDown (first button)')}
onMouseEnter={e => console.log('onMouseEnter (first button)')}

onMouseLeave={e => console.log('onMouseLeave (first button)')}
onMouseOver={e => console.log('onMouseOver (first button)')}
onMouseUp={e => console.log('onMouseUp (first button)')}
>
First button
</button>
<button
onClick={e => console.log('onClick (second button)')}
onMouseDown={e => console.log('onMouseDown (second button)')}
onMouseEnter={e => console.log('onMouseEnter (second button)')}
onMouseLeave={e => console.log('onMouseLeave (second button)')}
onMouseOver={e => console.log('onMouseOver (second button)')}
onMouseUp={e => console.log('onMouseUp (second button)')}
>
Second button
</button>

);
}
```

```css
label { display: block; }
input { margin-left: 10px; }
```

</Sandpack>

---

### Handling pointer events {/*handling-pointer-events*/}

This example shows some common [pointer events](#pointerevent-handler) and when they fire.

<Sandpack>

```js
export default function PointerExample() {
return (
<div

onPointerEnter={e => console.log('onPointerEnter (parent)')}
onPointerLeave={e => console.log('onPointerLeave (parent)')}
style={{ padding: 20, backgroundColor: '#ddd' }}
>
<div
onPointerDown={e => console.log('onPointerDown (first child)')}
onPointerEnter={e => console.log('onPointerEnter (first child)')}
onPointerLeave={e => console.log('onPointerLeave (first child)')}
onPointerMove={e => console.log('onPointerMove (first child)')}
onPointerUp={e => console.log('onPointerUp (first child)')}
style={{ padding: 20, backgroundColor: 'lightyellow' }}
>
First child

<div
onPointerDown={e => console.log('onPointerDown (second child)')}
onPointerEnter={e => console.log('onPointerEnter (second child)')}
onPointerLeave={e => console.log('onPointerLeave (second child)')}
onPointerMove={e => console.log('onPointerMove (second child)')}
onPointerUp={e => console.log('onPointerUp (second child)')}
style={{ padding: 20, backgroundColor: 'lightblue' }}
>
Second child

);
}
```

```css
label { display: block; }
input { margin-left: 10px; }
```

</Sandpack>

---
```

### Handling focus events {/*handling-focus-events*/}

In React, [focus events](#focusevent-handler) bubble. You can use the `currentTarget` and `relatedTarget` to differentiate if the focusing or blurring events originated from outside of the parent element. The example shows how to detect focusing a child, focusing the parent element, and how to detect focus entering or leaving the whole subtree.

<Sandpack>

```js
export default function FocusExample() {
return (
<div
tabIndex={1}
onFocus={(e) => {
if (e.currentTarget === e.target) {
console.log('focused parent');
} else {
console.log('focused child', e.target.name);
}
if (!e.currentTarget.contains(e.relatedTarget)) {
// Not triggered when swapping focus between children
console.log('focus entered parent');
}
}}
onBlur={(e) => {
if (e.currentTarget === e.target) {
console.log('unfocused parent');
} else {
console.log('unfocused child', e.target.name);
}
if (!e.currentTarget.contains(e.relatedTarget)) {
// Not triggered when swapping focus between children
console.log('focus left parent');
}
}}
>
<label>
```

First name:
<input name="firstName" />
</label>
<label>
Last name:
<input name="lastName" />
</label>

);
}
```

```css
label { display: block; }
input { margin-left: 10px; }
```

</Sandpack>

---

### Handling keyboard events {/*handling-keyboard-events*/}

This example shows some common [keyboard events](#keyboardevent-handler) and when they fire.

<Sandpack>

```js
export default function KeyboardExample() {
return (
<label>
First name:
<input
name="firstName"
onKeyDown={e => console.log('onKeyDown:', e.key, e.code)}
onKeyUp={e => console.log('onKeyUp:', e.key, e.code)}
/>
</label>
);
}

```

```css
label { display: block; }
input { margin-left: 10px; }
```

</Sandpack>

---

title: "<progress>"

---

<Intro>

The [built-in browser `<progress>` component](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/progress) lets you render a progress indicator.

```js
<progress value={0.5} />
```

</Intro>

<InlineToc />

---

## Reference {/*reference*/}

### `<progress>` {/*progress*/}

To display a progress indicator, render the [built-in browser `<progress>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/progress) component.

```js
<progress value={0.5} />
```

[See more examples below.](#usage)

#### Props {/*props*/}

`<progress>` supports all [common element props.](/reference/react-dom/components/common#props)

Additionally, `<progress>` supports these props:
```

* [`max`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/progress#attr-max): A number. Specifies the maximum `value`. Defaults to `1`.

* [`value`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/progress#attr-value): A number between `0` and `max`, or `null` for indeterminate progress. Specifies how much was done.

---

## Usage {/*usage*/}

### Controlling a progress indicator {/*controlling-a-progress-indicator*/}

To display a progress indicator, render a `<progress>` component. You can pass a number `value` between `0` and the `max` value you specify. If you don't pass a `max` value, it will assumed to be `1` by default.

If the operation is not ongoing, pass `value={null}` to put the progress indicator into an indeterminate state.

<Sandpack>

```js
export default function App() {
return (
<>
<progress value={0} />
<progress value={0.5} />
<progress value={0.7} />
<progress value={75} max={100} />
<progress value={1} />
<progress value={null} />
</>
);
}
```

```css
progress { display: block; }
```

</Sandpack>

---

title: "<select>"

---

<Intro>

The [built-in browser `<select>` component](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/select) lets you render a select box with options.

```js
<select>
<option value="someOption">Some option</option>
<option value="otherOption">Other option</option>
</select>
```

</Intro>

<InlineToc />

---

## Reference {/*reference*/}

### `<select>` {/*select*/}

To display a select box, render the [built-in browser `<select>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/select) component.

```js
<select>
<option value="someOption">Some option</option>
<option value="otherOption">Other option</option>
</select>
```

[See more examples below.](#usage)

#### Props {/*props*/}

`<select>` supports all [common element props.](/reference/react-dom/components/common#props)

You can [make a select box controlled](#controlling-a-select-box-with-a-state-variable) by passing a `value` prop:

* `value`: A string (or an array of strings for [`multiple={true}`](#enabling-multiple-selection)). Controls which option is selected. Every value string match the `value` of some `<option>` nested inside the `<select>`.

When you pass `value`, you must also pass an `onChange` handler that updates the passed value.

If your `<select>` is uncontrolled, you may pass the `defaultValue` prop instead:

* `defaultValue`: A string (or an array of strings for [`multiple={true}`](#enabling-multiple-selection)). Specifies [the initially selected option.](#providing-an-initially-selected-option)

These `<select>` props are relevant both for uncontrolled and controlled select boxes:

* [`autoComplete`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/select#attr-autocomplete): A string. Specifies one of the possible [autocomplete behaviors.](https://developer.mozilla.org/en-US/docs/Web/HTML/Attributes/autocomplete#values)

* [`autoFocus`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/select#attr-autofocus): A boolean. If `true`, React will focus the element on mount.

* `children`: `<select>` accepts [`<option>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/option), [`<optgroup>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/optgroup), and [`<datalist>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/optgroup) components as children. You can also pass your own components as long as they eventually render one of the allowed components. If you pass your own components that eventually render `<option>` tags, each `<option>` you render must have a `value`.

* [`disabled`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/select#attr-disabled): A boolean. If `true`, the select box will not be interactive and will appear dimmed.

* [`form`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/select#attr-form): A string. Specifies the `id` of the `<form>` this select box belongs to. If omitted, it's the closest parent form.

* [`multiple`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/select#attr-multiple): A boolean. If `true`, the browser allows [multiple selection.](#enabling-multiple-selection)

* [`name`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/select#attr-name): A string. Specifies the name for this select box that's [submitted with the form.](#reading-the-select-box-value-when-submitting-a-form)

* `onChange`: An [`Event` handler](/reference/react-dom/components/common#event-handler) function. Required for [controlled select boxes.](#controlling-a-select-box-with-a-state-variable) Fires immediately when the user picks a different option. Behaves like the browser [`input` event.](https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/input_event)

* `onChangeCapture`: A version of `onChange` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onInput`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/input_event): An [`Event` handler](/reference/react-dom/components/common#event-handler) function. Fires immediately when the value is changed by the user. For historical reasons, in React it is idiomatic to use `onChange` instead which works similarly.

* `onInputCapture`: A version of `onInput` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onInvalid`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLInputElement/invalid_event): An [`Event` handler](/reference/react-dom/components/common#event-handler) function. Fires if an input fails validation on form submit. Unlike the built-in `invalid` event, the React `onInvalid` event bubbles.

* `onInvalidCapture`: A version of `onInvalid` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`required`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/select#attr-required): A boolean. If `true`, the value must be provided for the form to submit.

* [`size`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/select#attr-size): A number. For `multiple={true}` selects, specifies the preferred number of initially visible items.

#### Caveats {/*caveats*/}

- Unlike in HTML, passing a `selected` attribute to `<option>` is not supported. Instead, use [`<select defaultValue>`](#providing-an-initially-selected-option) for uncontrolled select boxes and [`<select value>`](#controlling-a-select-box-with-a-state-variable) for controlled select boxes.

- If a select box receives a `value` prop, it will be [treated as controlled.](#controlling-a-select-box-with-a-state-variable)

- A select box can't be both controlled and uncontrolled at the same time.

- A select box cannot switch between being controlled or uncontrolled over its lifetime.

- Every controlled select box needs an `onChange` event handler that synchronously updates its backing value.

---

## Usage {/*usage*/}

### Displaying a select box with options {/*displaying-a-select-box-with-options*/}

Render a `<select>` with a list of `<option>` components inside to display a select box. Give each `<option>` a `value` representing the data to be submitted with the form.

<Sandpack>

```js
export default function FruitPicker() {
return (
<label>
Pick a fruit:
<select name="selectedFruit">
<option value="apple">Apple</option>
<option value="banana">Banana</option>
<option value="orange">Orange</option>
</select>
</label>
);
}
```

```css
```

select { margin: 5px; }
```

</Sandpack>

---

### Providing a label for a select box {/*providing-a-label-for-a-select-box*/}

Typically, you will place every `<select>` inside a [`<label>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/label) tag. This tells the browser that this label is associated with that select box. When the user clicks the label, the browser will automatically focus the select box. It's also essential for accessibility: a screen reader will announce the label caption when the user focuses the select box.

If you can't nest `<select>` into a `<label>`, associate them by passing the same ID to `<select id>` and [`<label htmlFor>`.](https://developer.mozilla.org/en-US/docs/Web/API/HTMLLabelElement/htmlFor) To avoid conflicts between multiple instances of one component, generate such an ID with [`useId`.](/reference/react/useId)

<Sandpack>

```js
import { useId } from 'react';

export default function Form() {
const vegetableSelectId = useId();
return (
<>
<label>
Pick a fruit:
<select name="selectedFruit">
<option value="apple">Apple</option>
<option value="banana">Banana</option>
<option value="orange">Orange</option>
</select>
</label>
<hr />
<label htmlFor={vegetableSelectId}>
Pick a vegetable:
</label>
<select id={vegetableSelectId} name="selectedVegetable">
<option value="cucumber">Cucumber</option>

<option value="corn">Corn</option>

<option value="tomato">Tomato</option>

</select>

</>

);

}

```

```css

select { margin: 5px; }

```

</Sandpack>

---

### Providing an initially selected option {/*providing-an-initially-selected-option*/}

By default, the browser will select the first `<option>` in the list. To select a different option by default, pass that `<option>`'s `value` as the `defaultValue` to the `<select>` element.

<Sandpack>

```js

export default function FruitPicker() {

return (

<label>

Pick a fruit:

<select name="selectedFruit" defaultValue="orange">

<option value="apple">Apple</option>

<option value="banana">Banana</option>

<option value="orange">Orange</option>

</select>

</label>

);

}

```

```css

select { margin: 5px; }

```
```

</Sandpack>

<Pitfall>

Unlike in HTML, passing a `selected` attribute to an individual `<option>` is not supported.

</Pitfall>

---

### Enabling multiple selection {/*enabling-multiple-selection*/}

Pass `multiple={true}` to the `<select>` to let the user select multiple options. In that case, if you also specify `defaultValue` to choose the initially selected options, it must be an array.

<Sandpack>

```js
export default function FruitPicker() {
return (
<label>
Pick some fruits:
<select
name="selectedFruit"
defaultValue={['orange', 'banana']}
multiple={true}
>
<option value="apple">Apple</option>
<option value="banana">Banana</option>
<option value="orange">Orange</option>
</select>
</label>
);
}
```

```css
select { display: block; margin-top: 10px; width: 200px; }
```

</Sandpack>

---

### Reading the select box value when submitting a form
{/*reading-the-select-box-value-when-submitting-a-form*/}

Add a [`<form>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/form) around your select box with a [`<button type="submit">`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/button) inside. It will call your `<form onSubmit>` event handler. By default, the browser will send the form data to the current URL and refresh the page. You can override that behavior by calling `e.preventDefault()`. Read the form data with [`new FormData(e.target)`](https://developer.mozilla.org/en-US/docs/Web/API/FormData).

<Sandpack>

```js
export default function EditPost() {
function handleSubmit(e) {
// Prevent the browser from reloading the page
e.preventDefault();
// Read the form data
const form = e.target;
const formData = new FormData(form);
// You can pass formData as a fetch body directly:
fetch('/some-api', { method: form.method, body: formData });
// You can generate a URL out of it, as the browser does by default:
console.log(new URLSearchParams(formData).toString());
// You can work with it as a plain object.
const formJson = Object.fromEntries(formData.entries());
console.log(formJson); // (!) This doesn't include multiple select values
// Or you can get an array of name-value pairs.
console.log([...formData.entries()]);
}
return (
<form method="post" onSubmit={handleSubmit}>
<label>
Pick your favorite fruit:
<select name="selectedFruit" defaultValue="orange">
<option value="apple">Apple</option>
<option value="banana">Banana</option>
<option value="orange">Orange</option>
</select>
```

</label>
<label>
Pick all your favorite vegetables:
<select
name="selectedVegetables"
multiple={true}
defaultValue={['corn', 'tomato']}
>
<option value="cucumber">Cucumber</option>
<option value="corn">Corn</option>
<option value="tomato">Tomato</option>
</select>
</label>
<hr />
<button type="reset">Reset</button>
<button type="submit">Submit</button>
</form>
);
}
```

```css
label, select { display: block; }
label { margin-bottom: 20px; }
```

</Sandpack>

<Note>

Give a `name` to your `<select>`, for example `<select name="selectedFruit" />`. The `name` you specified will be used as a key in the form data, for example `{ selectedFruit: "orange" }`.

If you use `<select multiple={true}>`, the [`FormData`](https://developer.mozilla.org/en-US/docs/Web/API/FormData) you'll read from the form will include each selected value as a separate name-value pair. Look closely at the console logs in the example above.

</Note>

<Pitfall>
```

By default, *any* `<button>` inside a `<form>` will submit it. This can be surprising! If you have your own custom `Button` React component, consider returning [`<button type="button">`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input/button) instead of `<button>`. Then, to be explicit, use `<button type="submit">` for buttons that *are* supposed to submit the form.

</Pitfall>

---

### Controlling a select box with a state variable {/*controlling-a-select-box-with-a-state-variable*/}

A select box like `<select />` is *uncontrolled.* Even if you [pass an initially selected value](#providing-an-initially-selected-option) like `<select defaultValue="orange" />`, your JSX only specifies the initial value, not the value right now.

**To render a _controlled_ select box, pass the `value` prop to it.** React will force the select box to always have the `value` you passed. Typically, you will control a select box by declaring a [state variable:](/reference/react/useState)

```js {2,6,7}
function FruitPicker() {
const [selectedFruit, setSelectedFruit] = useState('orange'); // Declare a state variable...
// ...
return (
<select
value={selectedFruit} // ...force the select's value to match the state variable...
onChange={e => setSelectedFruit(e.target.value)} // ... and update the state variable on any change!
>
<option value="apple">Apple</option>
<option value="banana">Banana</option>
<option value="orange">Orange</option>
</select>
);
}
```

This is useful if you want to re-render some part of the UI in response to every selection.

<Sandpack>

```js
import { useState } from 'react';

export default function FruitPicker() {
```

```
const [selectedFruit, setSelectedFruit] = useState('orange');
const [selectedVegs, setSelectedVegs] = useState(['corn', 'tomato']);
return (
<>
<label>
Pick a fruit:
<select
value={selectedFruit}
onChange={e => setSelectedFruit(e.target.value)}
>
<option value="apple">Apple</option>
<option value="banana">Banana</option>
<option value="orange">Orange</option>
</select>
</label>
<hr />
<label>
Pick all your favorite vegetables:
<select
multiple={true}
value={selectedVegs}
onChange={e => {
const options = [...e.target.selectedOptions];
const values = options.map(option => option.value);
setSelectedVegs(values);
}}
>
<option value="cucumber">Cucumber</option>
<option value="corn">Corn</option>
<option value="tomato">Tomato</option>
</select>
</label>
<hr />
<p>Your favorite fruit: {selectedFruit}</p>
<p>Your favorite vegetables: {selectedVegs.join(', ')}</p>
```

</>

);

}

```

```css

select { margin-bottom: 10px; display: block; }

```

</Sandpack>

<Pitfall>

**If you pass `value` without `onChange`, it will be impossible to select an option.** When you control a select box by passing some `value` to it, you *force* it to always have the value you passed. So if you pass a state variable as a `value` but forget to update that state variable synchronously during the `onChange` event handler, React will revert the select box after every keystroke back to the `value` that you specified.

Unlike in HTML, passing a `selected` attribute to an individual `<option>` is not supported.

</Pitfall>

---

title: "<textarea>"

---

<Intro>

The [built-in browser `<textarea>` component](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/textarea) lets you render a multiline text input.

```js

<textarea />

```

</Intro>

<InlineToc />

---

## Reference {/*reference*/}

### `<textarea>` {/*textarea*/}

To display a text area, render the [built-in browser `<textarea>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/textarea) component.
```

```js
<textarea name="postContent" />
```

[See more examples below.](#usage)

#### Props {/*props*/}

`<textarea>` supports all [common element props.](/reference/react-dom/components/common#props)

You can [make a text area controlled](#controlling-a-text-area-with-a-state-variable) by passing a `value` prop:

* `value`: A string. Controls the text inside the text area.

When you pass `value`, you must also pass an `onChange` handler that updates the passed value.

If your `<textarea>` is uncontrolled, you may pass the `defaultValue` prop instead:

* `defaultValue`: A string. Specifies [the initial value](#providing-an-initial-value-for-a-text-area) for a text area.

These `<textarea>` props are relevant both for uncontrolled and controlled text areas:

* [`autoComplete`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/textarea#attr-autocomplete): Either `'on'` or `'off'`. Specifies the autocomplete behavior.

* [`autoFocus`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/textarea#attr-autofocus): A boolean. If `true`, React will focus the element on mount.

* `children`: `<textarea>` does not accept children. To set the initial value, use `defaultValue`.

* [`cols`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/textarea#attr-cols): A number. Specifies the default width in average character widths. Defaults to `20`.

* [`disabled`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/textarea#attr-disabled): A boolean. If `true`, the input will not be interactive and will appear dimmed.

* [`form`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/textarea#attr-form): A string. Specifies the `id` of the `<form>` this input belongs to. If omitted, it's the closest parent form.

* [`maxLength`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/textarea#attr-maxlength): A number. Specifies the maximum length of text.

* [`minLength`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/textarea#attr-minlength): A number. Specifies the minimum length of text.

* [`name`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#name): A string. Specifies the name for this input that's [submitted with the form.](#reading-the-textarea-value-when-submitting-a-form)

* `onChange`: An [`Event` handler](/reference/react-dom/components/common#event-handler) function. Required for [controlled text areas.](#controlling-a-text-area-with-a-state-variable) Fires immediately when the input's value is changed by the user (for example, it fires on every keystroke). Behaves like the browser [`input`

event.](https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/input_event)

* `onChangeCapture`: A version of `onChange` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onInput`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/input_event): An [`Event` handler](/reference/react-dom/components/common#event-handler) function. function. Fires immediately when the value is changed by the user. For historical reasons, in React it is idiomatic to use `onChange` instead which works similarly.

* `onInputCapture`: A version of `onInput` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onInvalid`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLInputElement/invalid_event): An [`Event` handler](/reference/react-dom/components/common#event-handler) function. Fires if an input fails validation on form submit. Unlike the built-in `invalid` event, the React `onInvalid` event bubbles.

* `onInvalidCapture`: A version of `onInvalid` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onSelect`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLTextAreaElement/select_event): An [`Event` handler](/reference/react-dom/components/common#event-handler) function. Fires after the selection inside the `<textarea>` changes. React extends the `onSelect` event to also fire for empty selection and on edits (which may affect the selection).

* `onSelectCapture`: A version of `onSelect` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`placeholder`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/textarea#attr-placeholder): A string. Displayed in a dimmed color when the text area value is empty.

* [`readOnly`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/textarea#attr-readonly): A boolean. If `true`, the text area is not editable by the user.

* [`required`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/textarea#attr-required): A boolean. If `true`, the value must be provided for the form to submit.

* [`rows`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/textarea#attr-rows): A number. Specifies the default height in average character heights. Defaults to `2`.

* [`wrap`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/textarea#attr-wrap): Either `'hard'`, `'soft'`, or `'off'`. Specifies how the text should be wrapped when submitting a form.

#### Caveats {/*caveats*/}

- Passing children like `<textarea>something</textarea>` is not allowed. [Use `defaultValue` for initial content.](#providing-an-initial-value-for-a-text-area)

- If a text area receives a string `value` prop, it will be [treated as controlled.](#controlling-a-text-area-with-a-state-variable)

- A text area can't be both controlled and uncontrolled at the same time.

- A text area cannot switch between being controlled or uncontrolled over its lifetime.

- Every controlled text area needs an `onChange` event handler that synchronously updates its backing value.

---

## Usage {/*usage*/}

### Displaying a text area {/*displaying-a-text-area*/}

Render `<textarea>` to display a text area. You can specify its default size with the [`rows`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/textarea#rows) and [`cols`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/textarea#cols) attributes, but by default the user will be able to resize it. To disable resizing, you can specify `resize: none` in the CSS.

<Sandpack>

```js
export default function NewPost() {
return (
<label>
Write your post:
<textarea name="postContent" rows={4} cols={40} />
</label>
);
}
```

```css
input { margin-left: 5px; }
textarea { margin-top: 10px; }
label { margin: 10px; }
label, textarea { display: block; }
```

</Sandpack>

---

### Providing a label for a text area {/*providing-a-label-for-a-text-area*/}

Typically, you will place every `<textarea>` inside a [`<label>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/label) tag. This tells the browser that this label is associated with that text area. When the user clicks the label, the browser will focus the text area. It's also essential for accessibility: a screen reader will announce the label caption when the user focuses the text area.

If you can't nest `<textarea>` into a `<label>`, associate them by passing the same ID to `<textarea id>` and [`<label htmlFor>`.](https://developer.mozilla.org/en-US/docs/Web/API/HTMLLabelElement/htmlFor) To avoid conflicts between instances of one component, generate such an ID with [`useId`.](/reference/react/useId)

<Sandpack>

```js
import { useId } from 'react';

export default function Form() {
const postTextAreaId = useId();
return (
<>
<label htmlFor={postTextAreaId}>
Write your post:
</label>
<textarea
id={postTextAreaId}
name="postContent"
rows={4}
cols={40}
/>
</>
);
}
```

```css
input { margin: 5px; }
```

</Sandpack>

---

### Providing an initial value for a text area {/*providing-an-initial-value-for-a-text-area*/}

You can optionally specify the initial value for the text area. Pass it as the `defaultValue` string.

<Sandpack>

```js
export default function EditPost() {
return (
<label>
Edit your post:
<textarea
```

name="postContent"

defaultValue="I really enjoyed biking yesterday!"

rows={4}

cols={40}

/>

</label>

);

}

```

```css

input { margin-left: 5px; }

textarea { margin-top: 10px; }

label { margin: 10px; }

label, textarea { display: block; }

```

</Sandpack>

<Pitfall>

Unlike in HTML, passing initial text like `<textarea>Some content</textarea>` is not supported.

</Pitfall>

---

### Reading the text area value when submitting a form
{/*reading-the-text-area-value-when-submitting-a-form*/}

Add a [`<form>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/form) around your textarea with a [`<button type="submit">`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/button) inside. It will call your `<form onSubmit>` event handler. By default, the browser will send the form data to the current URL and refresh the page. You can override that behavior by calling `e.preventDefault()`. Read the form data with [`new FormData(e.target)`](https://developer.mozilla.org/en-US/docs/Web/API/FormData).

<Sandpack>

```js

export default function EditPost() {

function handleSubmit(e) {

// Prevent the browser from reloading the page

e.preventDefault();

```
// Read the form data
const form = e.target;
const formData = new FormData(form);

// You can pass formData as a fetch body directly:
fetch('/some-api', { method: form.method, body: formData });

// Or you can work with it as a plain object:
const formJson = Object.fromEntries(formData.entries());
console.log(formJson);
}

return (
<form method="post" onSubmit={handleSubmit}>
<label>
Post title: <input name="postTitle" defaultValue="Biking" />
</label>
<label>
Edit your post:
<textarea
name="postContent"
defaultValue="I really enjoyed biking yesterday!"
rows={4}
cols={40}
/>
</label>
<hr />
<button type="reset">Reset edits</button>
<button type="submit">Save post</button>
</form>
);
}
```

```css
label { display: block; }
input { margin: 5px; }
```

</Sandpack>

<Note>

Give a `name` to your `<textarea>`, for example `<textarea name="postContent" />`. The `name` you specified will be used as a key in the form data, for example `{ postContent: "Your post" }`.

</Note>

<Pitfall>

By default, *any* `<button>` inside a `<form>` will submit it. This can be surprising! If you have your own custom `Button` React component, consider returning [`<button type="button">`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input/button) instead of `<button>`. Then, to be explicit, use `<button type="submit">` for buttons that *are* supposed to submit the form.

</Pitfall>

---

### Controlling a text area with a state variable {/*controlling-a-text-area-with-a-state-variable*/}

A text area like `<textarea />` is *uncontrolled.* Even if you [pass an initial value](#providing-an-initial-value-for-a-text-area) like `<textarea defaultValue="Initial text" />`, your JSX only specifies the initial value, not the value right now.

**To render a _controlled_ text area, pass the `value` prop to it.** React will force the text area to always have the `value` you passed. Typically, you will control a text area by declaring a [state variable:](/reference/react/useState)

```js {2,6,7}
function NewPost() {
const [postContent, setPostContent] = useState(''); // Declare a state variable...
// ...
return (
<textarea
value={postContent} // ...force the input's value to match the state variable...
onChange={e => setPostContent(e.target.value)} // ... and update the state variable on any edits!
/>
);
}
```

This is useful if you want to re-render some part of the UI in response to every keystroke.

<Sandpack>

```js
import { useState } from 'react';
import MarkdownPreview from './MarkdownPreview.js';

export default function MarkdownEditor() {
const [postContent, setPostContent] = useState('_Hello,_ **Markdown**!');
return (
<>
<label>
Enter some markdown:
<textarea
value={postContent}
onChange={e => setPostContent(e.target.value)}
/>
</label>
<hr />
<MarkdownPreview markdown={postContent} />
</>
);
}
```

```js MarkdownPreview.js
import { Remarkable } from 'remarkable';

const md = new Remarkable();

export default function MarkdownPreview({ markdown }) {
const renderedHTML = md.render(markdown);
return <div dangerouslySetInnerHTML={{__html: renderedHTML}} />;
}
```

```json package.json
{
"dependencies": {
"react": "latest",
"react-dom": "latest",
"react-scripts": "latest",
```

"remarkable": "2.0.1"

},

"scripts": {

"start": "react-scripts start",

"build": "react-scripts build",

"test": "react-scripts test --env=jsdom",

"eject": "react-scripts eject"

}

}

```

```css

textarea { display: block; margin-top: 5px; margin-bottom: 10px; }

```

</Sandpack>

<Pitfall>

**If you pass `value` without `onChange`, it will be impossible to type into the text area.** When you control an text area by passing some `value` to it, you *force* it to always have the value you passed. So if you pass a state variable as a `value` but forget to update that state variable synchronously during the `onChange` event handler, React will revert the text area after every keystroke back to the `value` that you specified.

</Pitfall>

---

## Troubleshooting {/*troubleshooting*/}

### My text area doesn't update when I type into it {/*my-text-area-doesnt-update-when-i-type-into-it*/}

If you render a text area with `value` but no `onChange`, you will see an error in the console:

```js

// ■ Bug: controlled text area with no onChange handler

<textarea value={something} />

```

<ConsoleBlock level="error">

You provided a `value` prop to a form field without an `onChange` handler. This will render a read-only field. If the field should be mutable use `defaultValue`. Otherwise, set either `onChange` or `readOnly`.

</ConsoleBlock>
```

As the error message suggests, if you only wanted to [specify the *initial* value,](#providing-an-initial-value-for-a-text-area) pass `defaultValue` instead:

```js
// ■ Good: uncontrolled text area with an initial value
<textarea defaultValue={something} />
```

If you want [to control this text area with a state variable,](#controlling-a-text-area-with-a-state-variable) specify an `onChange` handler:

```js
// ■ Good: controlled text area with onChange
<textarea value={something} onChange={e => setSomething(e.target.value)} />
```

If the value is intentionally read-only, add a `readOnly` prop to suppress the error:

```js
// ■ Good: readonly controlled text area without on change
<textarea value={something} readOnly={true} />
```

---

### My text area caret jumps to the beginning on every keystroke {/*my-text-area-caret-jumps-to-the-beginning-on-every-keystroke*/}

If you [control a text area,](#controlling-a-text-area-with-a-state-variable) you must update its state variable to the text area's value from the DOM during `onChange`.

You can't update it to something other than `e.target.value`:

```js
function handleChange(e) {
// ■ Bug: updating an input to something other than e.target.value
setFirstName(e.target.value.toUpperCase());
}
```

You also can't update it asynchronously:

```js
function handleChange(e) {
// ■ Bug: updating an input asynchronously
```

```
setTimeout(() => {
setFirstName(e.target.value);
}, 100);
}
```

To fix your code, update it synchronously to `e.target.value`:

```js
function handleChange(e) {
// ■ Updating a controlled input to e.target.value synchronously
setFirstName(e.target.value);
}
```

If this doesn't fix the problem, it's possible that the text area gets removed and re-added from the DOM on every keystroke. This can happen if you're accidentally [resetting state](/learn/preserving-and-resetting-state) on every re-render. For example, this can happen if the text area or one of its parents always receives a different `key` attribute, or if you nest component definitions (which is not allowed in React and causes the "inner" component to remount on every render).

---

### I'm getting an error: "A component is changing an uncontrolled input to be controlled" {/*im-getting-an-error-a-component-is-changing-an-uncontrolled-input-to-be-controlled*/}

If you provide a `value` to the component, it must remain a string throughout its lifetime.

You cannot pass `value={undefined}` first and later pass `value="some string"` because React won't know whether you want the component to be uncontrolled or controlled. A controlled component should always receive a string `value`, not `null` or `undefined`.

If your `value` is coming from an API or a state variable, it might be initialized to `null` or `undefined`. In that case, either set it to an empty string (`''`) initially, or pass `value={someValue ?? ''}` to ensure `value` is a string.

---

title: "<input>"

---

<Intro>

The [built-in browser `<input>` component](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input) lets you render different kinds of form inputs.

```js
<input />
```

</Intro>

<InlineToc />

---

## Reference {/*reference*/}

### `<input>` {/*input*/}

To display an input, render the [built-in browser `<input>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input) component.

```js
<input name="myInput" />
```

[See more examples below.](#usage)

#### Props {/*props*/}

`<input>` supports all [common element props.](/reference/react-dom/components/common#props)

You can [make an input controlled](#controlling-an-input-with-a-state-variable) by passing one of these props:

* [`checked`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLInputElement#checked): A boolean. For a checkbox input or a radio button, controls whether it is selected.
* [`value`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLInputElement#value): A string. For a text input, controls its text. (For a radio button, specifies its form data.)

When you pass either of them, you must also pass an `onChange` handler that updates the passed value.

These `<input>` props are only relevant for uncontrolled inputs:

* [`defaultChecked`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLInputElement#defaultChecked): A boolean. Specifies [the initial value](#providing-an-initial-value-for-an-input) for `type="checkbox"` and `type="radio"` inputs.
* [`defaultValue`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLInputElement#defaultValue): A string. Specifies [the initial value](#providing-an-initial-value-for-an-input) for a text input.

These `<input>` props are relevant both for uncontrolled and controlled inputs:

* [`accept`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#accept): A string. Specifies which filetypes are accepted by a `type="file"` input.

* [`alt`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#alt): A string. Specifies the alternative image text for a `type="image"` input.

* [`capture`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#capture): A string. Specifies the media (microphone, video, or camera) captured by a `type="file"` input.

* [`autoComplete`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#autocomplete): A string. Specifies one of the possible [autocomplete behaviors.](https://developer.mozilla.org/en-US/docs/Web/HTML/Attributes/autocomplete#values)

* [`autoFocus`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#autofocus): A boolean. If `true`, React will focus the element on mount.

* [`dirname`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#dirname): A string. Specifies the form field name for the element's directionality.

* [`disabled`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#disabled): A boolean. If `true`, the input will not be interactive and will appear dimmed.

* `children`: `<input>` does not accept children.

* [`form`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#form): A string. Specifies the `id` of the `<form>` this input belongs to. If omitted, it's the closest parent form.

* [`formAction`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#formaction): A string. Overrides the parent `<form action>` for `type="submit"` and `type="image"`.

* [`formEnctype`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#formenctype): A string. Overrides the parent `<form enctype>` for `type="submit"` and `type="image"`.

* [`formMethod`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#formmethod): A string. Overrides the parent `<form method>` for `type="submit"` and `type="image"`.

* [`formNoValidate`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#formnovalidate): A string. Overrides the parent `<form noValidate>` for `type="submit"` and `type="image"`.

* [`formTarget`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#formtarget): A string. Overrides the parent `<form target>` for `type="submit"` and `type="image"`.

* [`height`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#height): A string. Specifies the image height for `type="image"`.

* [`list`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#list): A string. Specifies the `id` of the `<datalist>` with the autocomplete options.

* [`max`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#max): A number. Specifies the maximum value of numerical and datetime inputs.

* [`maxLength`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#maxlength): A number. Specifies the maximum length of text and other inputs.

* [`min`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#min): A number. Specifies the minimum value of numerical and datetime inputs.

* [`minLength`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#minlength): A number. Specifies the minimum length of text and other inputs.

* [`multiple`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#multiple): A boolean. Specifies whether multiple values are allowed for `<type="file"` and `type="email"`.

* [`name`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#name): A string. Specifies the name for this input that's [submitted with the form.](#reading-the-input-values-when-submitting-a-form)

* `onChange`: An [`Event` handler](/reference/react-dom/components/common#event-handler) function. Required for [controlled inputs.](#controlling-an-input-with-a-state-variable) Fires immediately when the input's value is changed by the user (for example, it fires on every keystroke). Behaves like the browser [`input` event.](https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/input_event)

* `onChangeCapture`: A version of `onChange` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onInput`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/input_event): An [`Event` handler](/reference/react-dom/components/common#event-handler) function. Fires immediately when the value is changed by the user. For historical reasons, in React it is idiomatic to use `onChange` instead which works similarly.

* `onInputCapture`: A version of `onInput` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onInvalid`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLInputElement/invalid_event): An [`Event` handler](/reference/react-dom/components/common#event-handler) function. Fires if an input fails validation on form submit. Unlike the built-in `invalid` event, the React `onInvalid` event bubbles.

* `onInvalidCapture`: A version of `onInvalid` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`onSelect`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLInputElement/select_event): An [`Event` handler](/reference/react-dom/components/common#event-handler) function. Fires after the selection inside the `<input>` changes. React extends the `onSelect` event to also fire for empty selection and on edits (which may affect the selection).

* `onSelectCapture`: A version of `onSelect` that fires in the [capture phase.](/learn/responding-to-events#capture-phase-events)

* [`pattern`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#pattern): A string. Specifies the pattern that the `value` must match.

* [`placeholder`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#placeholder): A string. Displayed in a dimmed color when the input value is empty.

* [`readOnly`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#readonly): A boolean. If `true`, the input is not editable by the user.

* [`required`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#required): A boolean. If `true`, the value must be provided for the form to submit.

* [`size`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#size): A number. Similar to setting width, but the unit depends on the control.

* [`src`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#src): A string. Specifies the image source for a `type="image"` input.

* [`step`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#step): A positive number or an `'any'` string. Specifies the distance between valid values.

* [`type`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#type): A string. One of the [input types.](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#input_types)

* [`width`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#width): A string. Specifies the image width for a `type="image"` input.

#### Caveats {/*caveats*/}

- Checkboxes need `checked` (or `defaultChecked`), not `value` (or `defaultValue`).

- If a text input receives a string `value` prop, it will be [treated as controlled.](#controlling-an-input-with-a-state-variable)

- If a checkbox or a radio button receives a boolean `checked` prop, it will be [treated as controlled.](#controlling-an-input-with-a-state-variable)

- An input can't be both controlled and uncontrolled at the same time.

- An input cannot switch between being controlled or uncontrolled over its lifetime.

- Every controlled input needs an `onChange` event handler that synchronously updates its backing value.

---

## Usage {/*usage*/}

### Displaying inputs of different types {/*displaying-inputs-of-different-types*/}

To display an input, render an `<input>` component. By default, it will be a text input. You can pass `type="checkbox"` for a checkbox, `type="radio"` for a radio button, [or one of the other input types.](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input#input_types)

<Sandpack>

```js
export default function MyForm() {
return (
<>
<label>
Text input: <input name="myInput" />
</label>
<hr />
<label>
Checkbox: <input type="checkbox" name="myCheckbox" />
</label>
<hr />
<p>
Radio buttons:
<label>
<input type="radio" name="myRadio" value="option1" />
```

Option 1

</label>

<label>

<input type="radio" name="myRadio" value="option2" />

Option 2

</label>

<label>

<input type="radio" name="myRadio" value="option3" />

Option 3

</label>

</p>

</>

);

}

```

```css

label { display: block; }

input { margin: 5px; }

```

</Sandpack>

---

### Providing a label for an input {/*providing-a-label-for-an-input*/}

Typically, you will place every `<input>` inside a [`<label>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/label) tag. This tells the browser that this label is associated with that input. When the user clicks the label, the browser will automatically focus the input. It's also essential for accessibility: a screen reader will announce the label caption when the user focuses the associated input.

If you can't nest `<input>` into a `<label>`, associate them by passing the same ID to `<input id>` and [`<label htmlFor>`.](https://developer.mozilla.org/en-US/docs/Web/API/HTMLLabelElement/htmlFor) To avoid conflicts between multiple instances of one component, generate such an ID with [`useId`.](/reference/react/useId)

<Sandpack>

```js

import { useId } from 'react';

export default function Form() {

````
const ageInputId = useId();
return (
<>
<label>
Your first name:
<input name="firstName" />
</label>
<hr />
<label htmlFor={ageInputId}>Your age:</label>
<input id={ageInputId} name="age" type="number" />
</>
);
}
```

```css
input { margin: 5px; }
```

</Sandpack>

Providing an initial value for an input {/*providing-an-initial-value-for-an-input*/}

You can optionally specify the initial value for any input. Pass it as the `defaultValue` string for text inputs. Checkboxes and radio buttons should specify the initial value with the `defaultChecked` boolean instead.

<Sandpack>

```js
export default function MyForm() {
return (
<>
<label>
Text input: <input name="myInput" defaultValue="Some initial value" />
</label>
<hr />
<label>
Checkbox: <input type="checkbox" name="myCheckbox" defaultChecked={true} />
````

</label>
<hr />

Radio buttons:
<label>
<input type="radio" name="myRadio" value="option1" />
Option 1
</label>
<label>
<input
type="radio"
name="myRadio"
value="option2"
defaultChecked={true}
/>
Option 2
</label>
<label>
<input type="radio" name="myRadio" value="option3" />
Option 3
</label>

</>
);
}
```

```css
label { display: block; }
input { margin: 5px; }
```

</Sandpack>

Reading the input values when submitting a form
{/*reading-the-input-values-when-submitting-a-form*/}
```

Add a [`<form>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/form) around your inputs with a [`<button type="submit">`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/button) inside. It will call your `<form onSubmit>` event handler. By default, the browser will send the form data to the current URL and refresh the page. You can override that behavior by calling `e.preventDefault()`. Read the form data with [`new FormData(e.target)`](https://developer.mozilla.org/en-US/docs/Web/API/FormData).

<Sandpack>

```js
export default function MyForm() {
function handleSubmit(e) {
// Prevent the browser from reloading the page
e.preventDefault();

// Read the form data
const form = e.target;
const formData = new FormData(form);

// You can pass formData as a fetch body directly:
fetch('/some-api', { method: form.method, body: formData });

// Or you can work with it as a plain object:
const formJson = Object.fromEntries(formData.entries());
console.log(formJson);
}

return (
<form method="post" onSubmit={handleSubmit}>
<label>
Text input: <input name="myInput" defaultValue="Some initial value" />
</label>
<hr />
<label>
Checkbox: <input type="checkbox" name="myCheckbox" defaultChecked={true} />
</label>
<hr />
<p>
Radio buttons:
<label><input type="radio" name="myRadio" value="option1" /> Option 1</label>
```

<label><input type="radio" name="myRadio" value="option2" defaultChecked={true} /> Option 2</label>

<label><input type="radio" name="myRadio" value="option3" /> Option 3</label>

</p>

<hr />

<button type="reset">Reset form</button>

<button type="submit">Submit form</button>

</form>

);

}

```

```css

label { display: block; }

input { margin: 5px; }

```

</Sandpack>

<Note>

Give a `name` to every `<input>`, for example `<input name="firstName" defaultValue="Taylor" />`. The `name` you specified will be used as a key in the form data, for example `{ firstName: "Taylor" }`.

</Note>

<Pitfall>

By default, *any* `<button>` inside a `<form>` will submit it. This can be surprising! If you have your own custom `Button` React component, consider returning [`<button type="button">`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input/button) instead of `<button>`. Then, to be explicit, use `<button type="submit">` for buttons that *are* supposed to submit the form.

</Pitfall>

Controlling an input with a state variable {/*controlling-an-input-with-a-state-variable*/}

An input like `<input />` is *uncontrolled.* Even if you [pass an initial value](#providing-an-initial-value-for-an-input) like `<input defaultValue="Initial text" />`, your JSX only specifies the initial value. It does not control what the value should be right now.

To render a _controlled_ input, pass the `value` prop to it (or `checked` for checkboxes and radios). React will force the input to always have the `value` you passed. Usually, you would do this by declaring a [state variable:](/reference/react/useState)
```

```js {2,6,7}
function Form() {
const [firstName, setFirstName] = useState(''); // Declare a state variable...
// ...
return (
<input
value={firstName} // ...force the input's value to match the state variable...
onChange={e => setFirstName(e.target.value)} // ... and update the state variable on any edits!
/>
);
}
```

A controlled input makes sense if you needed state anyway--for example, to re-render your UI on every edit:

```js {2,9}
function Form() {
const [firstName, setFirstName] = useState('');
return (
<>
<label>
First name:
<input value={firstName} onChange={e => setFirstName(e.target.value)} />
</label>
{firstName !== '' && <p>Your name is {firstName}.</p>}
...
```

It's also useful if you want to offer multiple ways to adjust the input state (for example, by clicking a button):

```js {3-4,10-11,14}
function Form() {
// ...
const [age, setAge] = useState('');
const ageAsNumber = Number(age);
return (
<>
```

<label>

Age:

<input

value={age}

onChange={e => setAge(e.target.value)}

type="number"

/>

<button onClick={() => setAge(ageAsNumber + 10)}>

Add 10 years

</button>

```

The `value` you pass to controlled components should not be `undefined` or `null`. If you need the initial value to be empty (such as with the `firstName` field below), initialize your state variable to an empty string (`''`).

<Sandpack>

```js

import { useState } from 'react';

export default function Form() {

const [firstName, setFirstName] = useState('');

const [age, setAge] = useState('20');

const ageAsNumber = Number(age);

return (

<>

<label>

First name:

<input

value={firstName}

onChange={e => setFirstName(e.target.value)}

/>

</label>

<label>

Age:

<input

value={age}

onChange={e => setAge(e.target.value)}

type="number"

/>

<button onClick={() => setAge(ageAsNumber + 10)}>

Add 10 years

</button>

</label>

{firstName !== '' &&

Your name is {firstName}.

}

{ageAsNumber > 0 &&

Your age is {ageAsNumber}.

}

</>

);

}

```

```css

label { display: block; }

input { margin: 5px; }

p { font-weight: bold; }

```

</Sandpack>

<Pitfall>

If you pass `value` without `onChange`, it will be impossible to type into the input. When you control an input by passing some `value` to it, you *force* it to always have the value you passed. So if you pass a state variable as a `value` but forget to update that state variable synchronously during the `onChange` event handler, React will revert the input after every keystroke back to the `value` that you specified.

</Pitfall>

Optimizing re-rendering on every keystroke {/*optimizing-re-rendering-on-every-keystroke*/}

When you use a controlled input, you set the state on every keystroke. If the component containing your state re-renders a large tree, this can get slow. There's a few ways you can optimize re-rendering performance.
```

For example, suppose you start with a form that re-renders all page content on every keystroke:

```js {5-8}
function App() {
const [firstName, setFirstName] = useState('');
return (
<>
<form>
<input value={firstName} onChange={e => setFirstName(e.target.value)} />
</form>
<PageContent />
</>
);
}
```

Since `<PageContent />` doesn't rely on the input state, you can move the input state into its own component:

```js {4,10-17}
function App() {
return (
<>
<SignupForm />
<PageContent />
</>
);
}

function SignupForm() {
const [firstName, setFirstName] = useState('');
return (
<form>
<input value={firstName} onChange={e => setFirstName(e.target.value)} />
</form>
);
}
```

This significantly improves performance because now only `SignupForm` re-renders on every keystroke.

If there is no way to avoid re-rendering (for example, if `PageContent` depends on the search input's value),
[`useDeferredValue`](/reference/react/useDeferredValue#deferring-re-rendering-for-a-part-of-the-ui) lets you keep the controlled input responsive even in the middle of a large re-render.

Troubleshooting {/*troubleshooting*/}

My text input doesn't update when I type into it {/*my-text-input-doesnt-update-when-i-type-into-it*/}

If you render an input with `value` but no `onChange`, you will see an error in the console:

```js
// ■ Bug: controlled text input with no onChange handler
<input value={something} />
```

<ConsoleBlock level="error">

You provided a `value` prop to a form field without an `onChange` handler. This will render a read-only field. If the field should be mutable use `defaultValue`. Otherwise, set either `onChange` or `readOnly`.

</ConsoleBlock>

As the error message suggests, if you only wanted to [specify the *initial* value,](#providing-an-initial-value-for-an-input) pass `defaultValue` instead:

```js
// ■ Good: uncontrolled input with an initial value
<input defaultValue={something} />
```

If you want [to control this input with a state variable,](#controlling-an-input-with-a-state-variable) specify an `onChange` handler:

```js
// ■ Good: controlled input with onChange
<input value={something} onChange={e => setSomething(e.target.value)} />
```

If the value is intentionally read-only, add a `readOnly` prop to suppress the error:

```js
// ■ Good: readonly controlled input without on change
```

<input value={something} readOnly={true} />
```

---

### My checkbox doesn't update when I click on it {/*my-checkbox-doesnt-update-when-i-click-on-it*/}

If you render a checkbox with `checked` but no `onChange`, you will see an error in the console:

```js
// ■ Bug: controlled checkbox with no onChange handler
<input type="checkbox" checked={something} />
```

<ConsoleBlock level="error">

You provided a `checked` prop to a form field without an `onChange` handler. This will render a read-only field. If the field should be mutable use `defaultChecked`. Otherwise, set either `onChange` or `readOnly`.

</ConsoleBlock>

As the error message suggests, if you only wanted to [specify the *initial* value,](#providing-an-initial-value-for-an-input) pass `defaultChecked` instead:

```js
// ■ Good: uncontrolled checkbox with an initial value
<input type="checkbox" defaultChecked={something} />
```

If you want [to control this checkbox with a state variable,](#controlling-an-input-with-a-state-variable) specify an `onChange` handler:

```js
// ■ Good: controlled checkbox with onChange
<input type="checkbox" checked={something} onChange={e => setSomething(e.target.checked)} />
```

<Pitfall>

You need to read `e.target.checked` rather than `e.target.value` for checkboxes.

</Pitfall>

If the checkbox is intentionally read-only, add a `readOnly` prop to suppress the error:

```js

// ■ Good: readonly controlled input without on change

<input type="checkbox" checked={something} readOnly={true} />

```

My input caret jumps to the beginning on every keystroke
{/*my-input-caret-jumps-to-the-beginning-on-every-keystroke*/}

If you [control an input,](#controlling-an-input-with-a-state-variable) you must update its state variable to the input's value from the DOM during `onChange`.

You can't update it to something other than `e.target.value` (or `e.target.checked` for checkboxes):

```js
function handleChange(e) {
// ■ Bug: updating an input to something other than e.target.value
setFirstName(e.target.value.toUpperCase());
}
```

You also can't update it asynchronously:

```js
function handleChange(e) {
// ■ Bug: updating an input asynchronously
setTimeout(() => {
setFirstName(e.target.value);
}, 100);
}
```

To fix your code, update it synchronously to `e.target.value`:

```js
function handleChange(e) {
// ■ Updating a controlled input to e.target.value synchronously
setFirstName(e.target.value);
}
```

If this doesn't fix the problem, it's possible that the input gets removed and re-added from the DOM on every keystroke. This can happen if you're accidentally [resetting
```

state](/learn/preserving-and-resetting-state) on every re-render, for example if the input or one of its parents always receives a different `key` attribute, or if you nest component function definitions (which is not supported and causes the "inner" component to always be considered a different tree).

I'm getting an error: "A component is changing an uncontrolled input to be controlled" {/*im-getting-an-error-a-component-is-changing-an-uncontrolled-input-to-be-controlled*/}

If you provide a `value` to the component, it must remain a string throughout its lifetime.

You cannot pass `value={undefined}` first and later pass `value="some string"` because React won't know whether you want the component to be uncontrolled or controlled. A controlled component should always receive a string `value`, not `null` or `undefined`.

If your `value` is coming from an API or a state variable, it might be initialized to `null` or `undefined`. In that case, either set it to an empty string (`''`) initially, or pass `value={someValue ?? ''}` to ensure `value` is a string.

Similarly, if you pass `checked` to a checkbox, ensure it's always a boolean.

---

title: Client React DOM APIs

<Intro>

The `react-dom/client` APIs let you render React components on the client (in the browser). These APIs are typically used at the top level of your app to initialize your React tree. A [framework](/learn/start-a-new-react-project#production-grade-react-frameworks) may call them for you. Most of your components don't need to import or use them.

</Intro>

Client APIs {/*client-apis*/}

* [`createRoot`](/reference/react-dom/client/createRoot) lets you create a root to display React components inside a browser DOM node.

* [`hydrateRoot`](/reference/react-dom/client/hydrateRoot) lets you display React components inside a browser DOM node whose HTML content was previously generated by [`react-dom/server`.](/reference/react-dom/server)

Browser support {/*browser-support*/}

React supports all popular browsers, including Internet Explorer 9 and above. Some polyfills are required for older browsers such as IE 9 and IE 10.

---

title: createRoot

<Intro>

`createRoot` lets you create a root to display React components inside a browser DOM node.

```js
const root = createRoot(domNode, options?)
```

</Intro>

<InlineToc />

Reference {/*reference*/}

`createRoot(domNode, options?)` {/*createroot*/}

Call `createRoot` to create a React root for displaying content inside a browser DOM element.

```js
import { createRoot } from 'react-dom/client';

const domNode = document.getElementById('root');
const root = createRoot(domNode);
```

React will create a root for the `domNode`, and take over managing the DOM inside it. After you've created a root, you need to call [`root.render`](#root-render) to display a React component inside of it:

```js
root.render(<App />);
```

An app fully built with React will usually only have one `createRoot` call for its root component. A page that uses "sprinkles" of React for parts of the page may have as many separate roots as needed.

[See more examples below.](#usage)

Parameters {/*parameters*/}

* `domNode`: A [DOM element.](https://developer.mozilla.org/en-US/docs/Web/API/Element) React will create a root for this DOM element and allow you to call functions on the root, such as `render` to display rendered React content.

* **optional** `options`: An object with options for this React root.

* **optional** `onRecoverableError`: Callback called when React automatically recovers from errors.

* **optional** `identifierPrefix`: A string prefix React uses for IDs generated by [`useId`.](/reference/react/useId) Useful to avoid conflicts when using multiple roots on the same page.

Returns {/*returns*/}

`createRoot` returns an object with two methods: [`render`](#root-render) and [`unmount`.](#root-unmount)

Caveats {/*caveats*/}

* If your app is server-rendered, using `createRoot()` is not supported. Use [`hydrateRoot()`](/reference/react-dom/client/hydrateRoot) instead.

* You'll likely have only one `createRoot` call in your app. If you use a framework, it might do this call for you.

* When you want to render a piece of JSX in a different part of the DOM tree that isn't a child of your component (for example, a modal or a tooltip), use [`createPortal`](/reference/react-dom/createPortal) instead of `createRoot`.

`root.render(reactNode)` {/*root-render*/}

Call `root.render` to display a piece of [JSX](/learn/writing-markup-with-jsx) ("React node") into the React root's browser DOM node.

```js
root.render(<App />);
```

React will display `<App />` in the `root`, and take over managing the DOM inside it.

[See more examples below.](#usage)

Parameters {/*root-render-parameters*/}

* `reactNode`: A *React node* that you want to display. This will usually be a piece of JSX like `<App />`, but you can also pass a React element constructed with [`createElement()`](/reference/react/createElement), a string, a number, `null`, or `undefined`.

Returns {/*root-render-returns*/}

`root.render` returns `undefined`.

Caveats {/*root-render-caveats*/}

* The first time you call `root.render`, React will clear all the existing HTML content inside the React root before rendering the React component into it.

* If your root's DOM node contains HTML generated by React on the server or during the build, use [`hydrateRoot()`](/reference/react-dom/client/hydrateRoot) instead, which attaches the event handlers to the existing HTML.

* If you call `render` on the same root more than once, React will update the DOM as necessary to reflect the latest JSX you passed. React will decide which parts of the DOM can be reused and which need to be recreated by ["matching it up"](/learn/preserving-and-resetting-state) with the previously rendered tree. Calling `render` on the same root again is similar to calling the [`set` function](/reference/react/useState#setstate) on the root component: React avoids unnecessary DOM updates.

`root.unmount()` {/*root-unmount*/}

Call `root.unmount` to destroy a rendered tree inside a React root.

```js
root.unmount();
```

An app fully built with React will usually not have any calls to `root.unmount`.

This is mostly useful if your React root's DOM node (or any of its ancestors) may get removed from the DOM by some other code. For example, imagine a jQuery tab panel that removes inactive tabs from the DOM. If a tab gets removed, everything inside it (including the React roots inside) would get removed from the DOM as well. In that case, you need to tell React to "stop" managing the removed root's content by calling `root.unmount`. Otherwise, the components inside the removed root won't know to clean up and free up global resources like subscriptions.

Calling `root.unmount` will unmount all the components in the root and "detach" React from the root DOM node, including removing any event handlers or state in the tree.

Parameters {/*root-unmount-parameters*/}

`root.unmount` does not accept any parameters.

Returns {/*root-unmount-returns*/}

`root.unmount` returns `undefined`.

Caveats {/*root-unmount-caveats*/}

* Calling `root.unmount` will unmount all the components in the tree and "detach" React from the root DOM node.

* Once you call `root.unmount` you cannot call `root.render` again on the same root. Attempting to call `root.render` on an unmounted root will throw a "Cannot update an unmounted root" error. However, you can create a new root for the same DOM node after the previous root for that node has been unmounted.

Usage {/*usage*/}

Rendering an app fully built with React {/*rendering-an-app-fully-built-with-react*/}

If your app is fully built with React, create a single root for your entire app.

```js [[1, 3, "document.getElementById('root')"], [2, 4, "<App />"]]
import { createRoot } from 'react-dom/client';

const root = createRoot(document.getElementById('root'));
root.render(<App />);
```

Usually, you only need to run this code once at startup. It will:

1. Find the <CodeStep step={1}>browser DOM node</CodeStep> defined in your HTML.
2. Display the <CodeStep step={2}>React component</CodeStep> for your app inside.

<Sandpack>

```html index.html
<!DOCTYPE html>
<html>
<head><title>My app</title></head>
<body>
<!-- This is the DOM node -->
<div id="root"></div>
</body>
</html>
```

```js index.js active
import { createRoot } from 'react-dom/client';
import App from './App.js';
import './styles.css';

const root = createRoot(document.getElementById('root'));
root.render(<App />);
```

```js App.js
```

```
import { useState } from 'react';

export default function App() {
return (
<>
<h1>Hello, world!</h1>
<Counter />
</>
);
}

function Counter() {
const [count, setCount] = useState(0);
return (
<button onClick={() => setCount(count + 1)}>
You clicked me {count} times
</button>
);
}
```

</Sandpack>

If your app is fully built with React, you shouldn't need to create any more roots, or to call [`root.render`](#root-render) again.

From this point on, React will manage the DOM of your entire app. To add more components, [nest them inside the `App` component.](/learn/importing-and-exporting-components) When you need to update the UI, each of your components can do this by [using state.](/reference/react/useState) When you need to display extra content like a modal or a tooltip outside the DOM node, [render it with a portal.](/reference/react-dom/createPortal)

<Note>

When your HTML is empty, the user sees a blank page until the app's JavaScript code loads and runs:

```html
<div id="root"></div>
```

This can feel very slow! To solve this, you can generate the initial HTML from your components [on the server or during the build.](/reference/react-dom/server) Then your visitors can read text, see images, and click links before any of the JavaScript code loads. We recommend [using a framework](/learn/start-a-new-react-project#production-grade-react-frameworks) that does this

optimization out of the box. Depending on when it runs, this is called *server-side rendering (SSR)* or *static site generation (SSG).*

</Note>

<Pitfall>

Apps using server rendering or static generation must call [`hydrateRoot`](/reference/react-dom/client/hydrateRoot) instead of `createRoot`. React will then *hydrate* (reuse) the DOM nodes from your HTML instead of destroying and re-creating them.

</Pitfall>

Rendering a page partially built with React {/*rendering-a-page-partially-built-with-react*/}

If your page [isn't fully built with React](/learn/add-react-to-an-existing-project#using-react-for-a-part-of-your-existing-page), you can call `createRoot` multiple times to create a root for each top-level piece of UI managed by React. You can display different content in each root by calling [`root.render`.](#root-render)

Here, two different React components are rendered into two DOM nodes defined in the `index.html` file:

<Sandpack>

```html public/index.html
<!DOCTYPE html>
<html>
<head><title>My app</title></head>
<body>
<nav id="navigation"></nav>
<main>
<p>This paragraph is not rendered by React (open index.html to verify).</p>
<section id="comments"></section>
</main>
</body>
</html>
```

```js index.js active
import './styles.css';
import { createRoot } from 'react-dom/client';
import { Comments, Navigation } from './Components.js';
```

```
const navDomNode = document.getElementById('navigation');
const navRoot = createRoot(navDomNode);
navRoot.render(<Navigation />);

const commentDomNode = document.getElementById('comments');
const commentRoot = createRoot(commentDomNode);
commentRoot.render(<Comments />);
```

```js Components.js
export function Navigation() {
return (
<ul>
<NavLink href="/">Home</NavLink>
<NavLink href="/about">About</NavLink>
</ul>
);
}

function NavLink({ href, children }) {
return (
<li>
<a href={href}>{children}</a>
</li>
);
}

export function Comments() {
return (
<>
<h2>Comments</h2>
<Comment text="Hello!" author="Sophie" />
<Comment text="How are you?" author="Sunil" />
</>
);
}

function Comment({ text, author }) {
return (
```

{text} — {author}

);

}

```

```css

nav ul { padding: 0; margin: 0; }

nav ul li { display: inline-block; margin-right: 20px; }

```

</Sandpack>

You could also create a new DOM node with [`document.createElement()`](https://developer.mozilla.org/en-US/docs/Web/API/Document/createElement) and add it to the document manually.

```js

const domNode = document.createElement('div');

const root = createRoot(domNode);

root.render(<Comment />);

document.body.appendChild(domNode); // You can add it anywhere in the document

```

To remove the React tree from the DOM node and clean up all the resources used by it, call [`root.unmount`.](#root-unmount)

```js

root.unmount();

```

This is mostly useful if your React components are inside an app written in a different framework.

Updating a root component {/*updating-a-root-component*/}

You can call `render` more than once on the same root. As long as the component tree structure matches up with what was previously rendered, React will [preserve the state.](/learn/preserving-and-resetting-state) Notice how you can type in the input, which means that the updates from repeated `render` calls every second in this example are not destructive:

<Sandpack>

```js index.js active

import { createRoot } from 'react-dom/client';

import './styles.css';

```
import App from './App.js';

const root = createRoot(document.getElementById('root'));

let i = 0;
setInterval(() => {
root.render(<App counter={i} />);
i++;
}, 1000);
```

```js App.js
export default function App({counter}) {
return (
<>
<h1>Hello, world! {counter}</h1>
<input placeholder="Type something here" />
</>
);
}
```

</Sandpack>

It is uncommon to call `render` multiple times. Usually, your components will [update state](/reference/react/useState) instead.

Troubleshooting {/*troubleshooting*/}

I've created a root, but nothing is displayed {/*ive-created-a-root-but-nothing-is-displayed*/}

Make sure you haven't forgotten to actually *render* your app into the root:

```js {5}
import { createRoot } from 'react-dom/client';
import App from './App.js';

const root = createRoot(document.getElementById('root'));
root.render(<App />);
```

Until you do that, nothing is displayed.

I'm getting an error: "Target container is not a DOM element"
{/*im-getting-an-error-target-container-is-not-a-dom-element*/}

This error means that whatever you're passing to `createRoot` is not a DOM node.

If you're not sure what's happening, try logging it:

```js {2}
const domNode = document.getElementById('root');
console.log(domNode); // ???
const root = createRoot(domNode);
root.render(<App />);
```

For example, if `domNode` is `null`, it means that [`getElementById`](https://developer.mozilla.org/en-US/docs/Web/API/Document/getElementById) returned `null`. This will happen if there is no node in the document with the given ID at the time of your call. There may be a few reasons for it:

1. The ID you're looking for might differ from the ID you used in the HTML file. Check for typos!
2. Your bundle's `<script>` tag cannot "see" any DOM nodes that appear *after* it in the HTML.

Another common way to get this error is to write `createRoot(<App />)` instead of `createRoot(domNode)`.

I'm getting an error: "Functions are not valid as a React child."
{/*im-getting-an-error-functions-are-not-valid-as-a-react-child*/}

This error means that whatever you're passing to `root.render` is not a React component.

This may happen if you call `root.render` with `Component` instead of `<Component />`:

```js {2,5}
// ■ Wrong: App is a function, not a Component.
root.render(App);

// ■ Correct: <App /> is a component.
root.render(<App />);
```

Or if you pass a function to `root.render`, instead of the result of calling it:

```js {2,5}
```

// ■ Wrong: createApp is a function, not a component.

root.render(createApp);

// ■ Correct: call createApp to return a component.

root.render(createApp());

```

---

### My server-rendered HTML gets re-created from scratch
{/*my-server-rendered-html-gets-re-created-from-scratch*/}

If your app is server-rendered and includes the initial HTML generated by React, you might notice that creating a root and calling `root.render` deletes all that HTML, and then re-creates all the DOM nodes from scratch. This can be slower, resets focus and scroll positions, and may lose other user input.

Server-rendered apps must use [`hydrateRoot`](/reference/react-dom/client/hydrateRoot) instead of `createRoot`:

```js {1,4-7}

import { hydrateRoot } from 'react-dom/client';

import App from './App.js';

hydrateRoot(

document.getElementById('root'),

<App />

);

```

Note that its API is different. In particular, usually there will be no further `root.render` call.

---

title: hydrateRoot

<Intro>

`hydrateRoot` lets you display React components inside a browser DOM node whose HTML content was previously generated by [`react-dom/server`.](/reference/react-dom/server)

```js

const root = hydrateRoot(domNode, reactNode, options?)

```

</Intro>

<InlineToc />
```

Reference {/*reference*/}

`hydrateRoot(domNode, reactNode, options?)` {/*hydrateroot*/}

Call `hydrateRoot` to “attach” React to existing HTML that was already rendered by React in a server environment.

```js
import { hydrateRoot } from 'react-dom/client';

const domNode = document.getElementById('root');
const root = hydrateRoot(domNode, reactNode);
```

React will attach to the HTML that exists inside the `domNode`, and take over managing the DOM inside it. An app fully built with React will usually only have one `hydrateRoot` call with its root component.

[See more examples below.](#usage)

Parameters {/*parameters*/}

* `domNode`: A [DOM element](https://developer.mozilla.org/en-US/docs/Web/API/Element) that was rendered as the root element on the server.

* `reactNode`: The "React node" used to render the existing HTML. This will usually be a piece of JSX like `<App />` which was rendered with a `ReactDOM Server` method such as `renderToPipeableStream(<App />)`.

* **optional** `options`: An object with options for this React root.

* **optional** `onRecoverableError`: Callback called when React automatically recovers from errors.

* **optional** `identifierPrefix`: A string prefix React uses for IDs generated by [`useId`.](/reference/react/useId) Useful to avoid conflicts when using multiple roots on the same page. Must be the same prefix as used on the server.

Returns {/*returns*/}

`hydrateRoot` returns an object with two methods: [`render`](#root-render) and [`unmount`.](#root-unmount)

Caveats {/*caveats*/}

* `hydrateRoot()` expects the rendered content to be identical with the server-rendered content. 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.

* You'll likely have only one `hydrateRoot` call in your app. If you use a framework, it might do this call for you.

* If your app is client-rendered with no HTML rendered already, using `hydrateRoot()` is not supported. Use [`createRoot()`](/reference/react-dom/client/createRoot) instead.

`root.render(reactNode)` {/*root-render*/}

Call `root.render` to update a React component inside a hydrated React root for a browser DOM element.

```js
root.render(<App />);
```

React will update `<App />` in the hydrated `root`.

[See more examples below.](#usage)

Parameters {/*root-render-parameters*/}

* `reactNode`: A "React node" that you want to update. This will usually be a piece of JSX like `<App />`, but you can also pass a React element constructed with [`createElement()`](/reference/react/createElement), a string, a number, `null`, or `undefined`.

Returns {/*root-render-returns*/}

`root.render` returns `undefined`.

Caveats {/*root-render-caveats*/}

* If you call `root.render` before the root has finished hydrating, React will clear the existing server-rendered HTML content and switch the entire root to client rendering.

`root.unmount()` {/*root-unmount*/}

Call `root.unmount` to destroy a rendered tree inside a React root.

```js
root.unmount();
```

An app fully built with React will usually not have any calls to `root.unmount`.

This is mostly useful if your React root's DOM node (or any of its ancestors) may get removed from the DOM by some other code. For example, imagine a jQuery tab panel that removes inactive tabs from the DOM. If a tab gets removed, everything inside it (including the React roots inside) would get removed from the DOM as well. You need to tell React to "stop" managing the removed root's content by calling `root.unmount`. Otherwise, the components inside the removed root won't clean up and free up resources like subscriptions.

Calling `root.unmount` will unmount all the components in the root and "detach" React from the root DOM node, including removing any event handlers or state in the tree.

Parameters {/*root-unmount-parameters*/}

`root.unmount` does not accept any parameters.

Returns {/*root-unmount-returns*/}

`root.unmount` returns `undefined`.

Caveats {/*root-unmount-caveats*/}

* Calling `root.unmount` will unmount all the components in the tree and "detach" React from the root DOM node.

* Once you call `root.unmount` you cannot call `root.render` again on the root. Attempting to call `root.render` on an unmounted root will throw a "Cannot update an unmounted root" error.

Usage {/*usage*/}

Hydrating server-rendered HTML {/*hydrating-server-rendered-html*/}

If your app's HTML was generated by [`react-dom/server`](/reference/react-dom/client/createRoot), you need to *hydrate* it on the client.

```js [[1, 3, "document.getElementById('root')"], [2, 3, "<App />"]]
import { hydrateRoot } from 'react-dom/client';

hydrateRoot(document.getElementById('root'), <App />);
```

This will hydrate the server HTML inside the <CodeStep step={1}>browser DOM node</CodeStep> with the <CodeStep step={2}>React component</CodeStep> for your app. Usually, you will do it once at startup. If you use a framework, it might do this behind the scenes for you.

To hydrate your app, React will "attach" your components' logic to the initial generated HTML from the server. Hydration turns the initial HTML snapshot from the server into a fully interactive app that runs in the browser.

<Sandpack>

```html public/index.html
<!--
HTML content inside <div id="root">...</div>
was generated from App by react-dom/server.
-->
<div id="root"><h1>Hello, world!</h1><button>You clicked me <!-- -->0<!-- --> times</button></div>
```

```js index.js active
import './styles.css';
import { hydrateRoot } from 'react-dom/client';
import App from './App.js';

hydrateRoot(
document.getElementById('root'),
<App />
);
```

```js App.js
import { useState } from 'react';

export default function App() {
return (
<>
<h1>Hello, world!</h1>
<Counter />
</>
);
}

function Counter() {
const [count, setCount] = useState(0);
return (
<button onClick={() => setCount(count + 1)}>
You clicked me {count} times
</button>
);
}
```

```

</Sandpack>

You shouldn't need to call `hydrateRoot` again or to call it in more places. From this point on, React will be managing the DOM of your application. To update the UI, your components will [use state](/reference/react/useState) instead.

<Pitfall>

The React tree you pass to `hydrateRoot` needs to produce **the same output** as it did on the server.

This is important for the user experience. The user will spend some time looking at the server-generated HTML before your JavaScript code loads. Server rendering creates an illusion that the app loads faster by showing the HTML snapshot of its output. Suddenly showing different content breaks that illusion. This is why the server render output must match the initial render output on the client.

The most common causes leading to hydration errors include:

* Extra whitespace (like newlines) around the React-generated HTML inside the root node.

* Using checks like `typeof window !== 'undefined'` in your rendering logic.

* Using browser-only APIs like [`window.matchMedia`](https://developer.mozilla.org/en-US/docs/Web/API/Window/matchMedia) in your rendering logic.

* Rendering different data on the server and the client.

React recovers from some hydration errors, but **you must fix them like other bugs.** In the best case, they'll lead to a slowdown; in the worst case, event handlers can get attached to the wrong elements.

</Pitfall>

---

### Hydrating an entire document {/*hydrating-an-entire-document*/}

Apps fully built with React can render the entire document as JSX, including the [`<html>`](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/html) tag:

```js {3,13}
function App() {
return (
<html>
<head>
<meta charSet="utf-8" />
<meta name="viewport" content="width=device-width, initial-scale=1" />
<link rel="stylesheet" href="/styles.css"></link>
```
```

<title>My app</title>

</head>

<body>

<Router />

</body>

</html>

);

}

```

To hydrate the entire document, pass the [`document`](https://developer.mozilla.org/en-US/docs/Web/API/Window/document) global as the first argument to `hydrateRoot`:

```js {4}

import { hydrateRoot } from 'react-dom/client';

import App from './App.js';

hydrateRoot(document, <App />);

```

Suppressing unavoidable hydration mismatch errors
{/*suppressing-unavoidable-hydration-mismatch-errors*/}

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 hydration mismatch warning.

To silence hydration warnings on an element, add `suppressHydrationWarning={true}`:

<Sandpack>

```html public/index.html

<!--

HTML content inside <div id="root">...</div>

was generated from App by react-dom/server.

-->

<div id="root"><h1>Current Date: <!-- -->01/01/2020</h1></div>

```

```js index.js

import './styles.css';

```
import { hydrateRoot } from 'react-dom/client';
import App from './App.js';

hydrateRoot(document.getElementById('root'), <App />);
```

```js App.js active
export default function App() {
return (
<h1 suppressHydrationWarning={true}>
Current Date: {new Date().toLocaleDateString()}
</h1>
);
}
```

</Sandpack>

This 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.

Handling different client and server content {/*handling-different-client-and-server-content*/}

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](/reference/react/useState) like `isClient`, which you can set to `true` in an [Effect](/reference/react/useEffect):

<Sandpack>

```html public/index.html
<!--
HTML content inside <div id="root">...</div>
was generated from App by react-dom/server.
-->
<div id="root"><h1>Is Server</h1></div>
```

```js index.js
import './styles.css';
import { hydrateRoot } from 'react-dom/client';
```

```
import App from './App.js';

hydrateRoot(document.getElementById('root'), <App />);
```

```js App.js active
import { useState, useEffect } from "react";

export default function App() {
const [isClient, setIsClient] = useState(false);

useEffect(() => {
setIsClient(true);
}, []);

return (
<h1>
{isClient ? 'Is Client' : 'Is Server'}
</h1>
);
}
```

</Sandpack>

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.

<Pitfall>

This approach makes hydration slower because your components have to render twice. Be mindful of the user experience on slow connections. The JavaScript code may load significantly later than the initial HTML render, so rendering a different UI immediately after hydration may also feel jarring to the user.

</Pitfall>

Updating a hydrated root component {/*updating-a-hydrated-root-component*/}

After the root has finished hydrating, you can call [`root.render`](#root-render) to update the root React component. **Unlike with [`createRoot`](/reference/react-dom/client/createRoot), you don't usually need to do this because the initial content was already rendered as HTML.**

If you call `root.render` at some point after hydration, and the component tree structure matches up with what was previously rendered, React will [preserve the state.](/learn/preserving-and-resetting-state)

Notice how you can type in the input, which means that the updates from repeated `render` calls every second in this example are not destructive:

<Sandpack>

```html public/index.html
<!--
All HTML content inside <div id="root">...</div> was
generated by rendering <App /> with react-dom/server.
-->
<div id="root"><h1>Hello, world! <!-- -->0</h1><input placeholder="Type something here"/></div>
```

```js index.js active
import { hydrateRoot } from 'react-dom/client';
import './styles.css';
import App from './App.js';

const root = hydrateRoot(
document.getElementById('root'),
<App counter={0} />
);

let i = 0;
setInterval(() => {
root.render(<App counter={i} />);
i++;
}, 1000);
```

```js App.js
export default function App({counter}) {
return (
<>
<h1>Hello, world! {counter}</h1>
<input placeholder="Type something here" />
</>
);
}
```

</Sandpack>

It is uncommon to call [`root.render`](#root-render) on a hydrated root. Usually, you'll [update state](/reference/react/useState) inside one of the components instead.

---

title: renderToReadableStream

<Intro>

`renderToReadableStream` renders a React tree to a [Readable Web Stream.](https://developer.mozilla.org/en-US/docs/Web/API/ReadableStream)

```js
const stream = await renderToReadableStream(reactNode, options?)
```

</Intro>

<InlineToc />

<Note>

This API depends on [Web Streams.](https://developer.mozilla.org/en-US/docs/Web/API/Streams_API) For Node.js, use [`renderToPipeableStream`](/reference/react-dom/server/renderToPipeableStream) instead.

</Note>

Reference {/*reference*/}

`renderToReadableStream(reactNode, options?)` {/*rendertoreadablestream*/}

Call `renderToReadableStream` to render your React tree as HTML into a [Readable Web Stream.](https://developer.mozilla.org/en-US/docs/Web/API/ReadableStream)

```js
import { renderToReadableStream } from 'react-dom/server';

async function handler(request) {
const stream = await renderToReadableStream(<App />, {
bootstrapScripts: ['/main.js']
});
return new Response(stream, {
headers: { 'content-type': 'text/html' },
```

});

}
```

On the client, call [`hydrateRoot`](/reference/react-dom/client/hydrateRoot) to make the server-generated HTML interactive.

[See more examples below.](#usage)

#### Parameters {/*parameters*/}

* `reactNode`: A React node you want to render to HTML. For example, a JSX element like `<App />`. It is expected to represent the entire document, so the `App` component should render the `<html>` tag.

* **optional** `options`: An object with streaming options.

* **optional** `bootstrapScriptContent`: If specified, this string will be placed in an inline `<script>` tag.

* **optional** `bootstrapScripts`: An array of string URLs for the `<script>` tags to emit on the page. Use this to include the `<script>` that calls [`hydrateRoot`.](/reference/react-dom/client/hydrateRoot) Omit it if you don't want to run React on the client at all.

* **optional** `bootstrapModules`: Like `bootstrapScripts`, but emits [`<script type="module">`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules) instead.

* **optional** `identifierPrefix`: A string prefix React uses for IDs generated by [`useId`.](/reference/react/useId) Useful to avoid conflicts when using multiple roots on the same page. Must be the same prefix as passed to [`hydrateRoot`.](/reference/react-dom/client/hydrateRoot#parameters)

* **optional** `namespaceURI`: A string with the root [namespace URI](https://developer.mozilla.org/en-US/docs/Web/API/Document/createElementNS#important_namespace_uris) for the stream. Defaults to regular HTML. Pass `'http://www.w3.org/2000/svg'` for SVG or `'http://www.w3.org/1998/Math/MathML'` for MathML.

* **optional** `nonce`: A [`nonce`](http://developer.mozilla.org/en-US/docs/Web/HTML/Element/script#nonce) string to allow scripts for [`script-src` Content-Security-Policy](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/script-src).

* **optional** `onError`: A callback that fires whenever there is a server error, whether [recoverable](#recovering-from-errors-outside-the-shell) or [not.](#recovering-from-errors-inside-the-shell) By default, this only calls `console.error`. If you override it to [log crash reports,](#logging-crashes-on-the-server) make sure that you still call `console.error`. You can also use it to [adjust the status code](#setting-the-status-code) before the shell is emitted.

* **optional** `progressiveChunkSize`: The number of bytes in a chunk. [Read more about the default heuristic.](https://github.com/facebook/react/blob/14c2be8dac2d5482fda8a0906a31d239df8551fc/packages/react-server/src/ReactFizzServer.js#L210-L225)

* **optional** `signal`: An [abort signal](https://developer.mozilla.org/en-US/docs/Web/API/AbortSignal) that lets you [abort server rendering](#aborting-server-rendering) and render the rest on the client.

#### Returns {/*returns*/}

`renderToReadableStream` returns a Promise:
```

- If rendering the [shell](#specifying-what-goes-into-the-shell) is successful, that Promise will resolve to a [Readable Web Stream.](https://developer.mozilla.org/en-US/docs/Web/API/ReadableStream)

- If rendering the shell fails, the Promise will be rejected. [Use this to output a fallback shell.](#recovering-from-errors-inside-the-shell)

The returned stream has an additional property:

* `allReady`: A Promise that resolves when all rendering is complete, including both the [shell](#specifying-what-goes-into-the-shell) and all additional [content.](#streaming-more-content-as-it-loads) You can `await stream.allReady` before returning a response [for crawlers and static generation.](#waiting-for-all-content-to-load-for-crawlers-and-static-generation) If you do that, you won't get any progressive loading. The stream will contain the final HTML.

Usage {/*usage*/}

Rendering a React tree as HTML to a Readable Web Stream {/*rendering-a-react-tree-as-html-to-a-readable-web-stream*/}

Call `renderToReadableStream` to render your React tree as HTML into a [Readable Web Stream:](https://developer.mozilla.org/en-US/docs/Web/API/ReadableStream)

```js [[1, 4, "<App />"], [2, 5, "['/main.js']"]]
import { renderToReadableStream } from 'react-dom/server';

async function handler(request) {
const stream = await renderToReadableStream(<App />, {
bootstrapScripts: ['/main.js']
});
return new Response(stream, {
headers: { 'content-type': 'text/html' },
});
}
```

Along with the <CodeStep step={1}>root component</CodeStep>, you need to provide a list of <CodeStep step={2}>bootstrap `<script>` paths</CodeStep>. Your root component should return **the entire document including the root `<html>` tag.**

For example, it might look like this:

```js [[1, 1, "App"]]
export default function App() {
return (
```

<html>

<head>

<meta charSet="utf-8" />

<meta name="viewport" content="width=device-width, initial-scale=1" />

<link rel="stylesheet" href="/styles.css"></link>

<title>My app</title>

</head>

<body>

<Router />

</body>

</html>

);

}

```

React will inject the [doctype](https://developer.mozilla.org/en-US/docs/Glossary/Doctype) and your <CodeStep step={2}>bootstrap `<script>` tags</CodeStep> into the resulting HTML stream:

```html [[2, 5, "/main.js"]]

<!DOCTYPE html>

<html>

<!-- ... HTML from your components ... -->

</html>

<script src="/main.js" async=""></script>

```

On the client, your bootstrap script should [hydrate the entire `document` with a call to `hydrateRoot`:](/reference/react-dom/client/hydrateRoot#hydrating-an-entire-document)

```js [[1, 4, "<App />"]]

import { hydrateRoot } from 'react-dom/client';

import App from './App.js';

hydrateRoot(document, <App />);

```

This will attach event listeners to the server-generated HTML and make it interactive.

<DeepDive>

Reading CSS and JS asset paths from the build output
{/*reading-css-and-js-asset-paths-from-the-build-output*/}
```

The final asset URLs (like JavaScript and CSS files) are often hashed after the build. For example, instead of `styles.css` you might end up with `styles.123456.css`. Hashing static asset filenames guarantees that every distinct build of the same asset will have a different filename. This is useful because it lets you safely enable long-term caching for static assets: a file with a certain name would never change content.

However, if you don't know the asset URLs until after the build, there's no way for you to put them in the source code. For example, hardcoding `"/styles.css"` into JSX like earlier wouldn't work. To keep them out of your source code, your root component can read the real filenames from a map passed as a prop:

```js {1,6}
export default function App({ assetMap }) {
return (
<html>
<head>
<title>My app</title>
<link rel="stylesheet" href={assetMap['styles.css']}></link>
</head>
...
</html>
);
}
```

On the server, render `<App assetMap={assetMap} />` and pass your `assetMap` with the asset URLs:

```js {1-5,8,9}
// You'd need to get this JSON from your build tooling, e.g. read it from the build output.
const assetMap = {
'styles.css': '/styles.123456.css',
'main.js': '/main.123456.js'
};

async function handler(request) {
const stream = await renderToReadableStream(<App assetMap={assetMap} />, {
bootstrapScripts: [assetMap['/main.js']]
});
return new Response(stream, {
headers: { 'content-type': 'text/html' },
});
```

}
```

Since your server is now rendering `&lt;App assetMap={assetMap} /&gt;`, you need to render it with `assetMap` on the client too to avoid hydration errors. You can serialize and pass `assetMap` to the client like this:

```js {9-10}
// You'd need to get this JSON from your build tooling.
const assetMap = {
'styles.css': '/styles.123456.css',
'main.js': '/main.123456.js'
};

async function handler(request) {
const stream = await renderToReadableStream(<App assetMap={assetMap} />, {
// Careful: It's safe to stringify() this because this data isn't user-generated.
bootstrapScriptContent: `window.assetMap = ${JSON.stringify(assetMap)};`,
bootstrapScripts: [assetMap['/main.js']],
});
return new Response(stream, {
headers: { 'content-type': 'text/html' },
});
}
```

In the example above, the `bootstrapScriptContent` option adds an extra inline `<script>` tag that sets the global `window.assetMap` variable on the client. This lets the client code read the same `assetMap`:

```js {4}
import { hydrateRoot } from 'react-dom/client';
import App from './App.js';

hydrateRoot(document, <App assetMap={window.assetMap} />);
```

Both client and server render `App` with the same `assetMap` prop, so there are no hydration errors.

</DeepDive>

```

Streaming more content as it loads {/*streaming-more-content-as-it-loads*/}

Streaming allows the user to start seeing the content even before all the data has loaded on the server. For example, consider a profile page that shows a cover, a sidebar with friends and photos, and a list of posts:

```js
function ProfilePage() {
return (
<ProfileLayout>
<ProfileCover />
<Sidebar>
<Friends />
<Photos />
</Sidebar>
<Posts />
</ProfileLayout>
);
}
```

Imagine that loading data for `<Posts />` takes some time. Ideally, you'd want to show the rest of the profile page content to the user without waiting for the posts. To do this, [wrap `Posts` in a `<Suspense>` boundary:](/reference/react/Suspense#displaying-a-fallback-while-content-is-loading)

```js {9,11}
function ProfilePage() {
return (
<ProfileLayout>
<ProfileCover />
<Sidebar>
<Friends />
<Photos />
</Sidebar>
<Suspense fallback={<PostsGlimmer />}>
<Posts />
</Suspense>
</ProfileLayout>
);
```

}
```

This tells React to start streaming the HTML before `Posts` loads its data. React will send the HTML for the loading fallback (`PostsGlimmer`) first, and then, when `Posts` finishes loading its data, React will send the remaining HTML along with an inline `<script>` tag that replaces the loading fallback with that HTML. From the user's perspective, the page will first appear with the `PostsGlimmer`, later replaced by the `Posts`.

You can further [nest `<Suspense>` boundaries](/reference/react/Suspense#revealing-nested-content-as-it-loads) to create a more granular loading sequence:

```js {5,13}
function ProfilePage() {
return (
<ProfileLayout>
<ProfileCover />
<Suspense fallback={<BigSpinner />}>
<Sidebar>
<Friends />
<Photos />
</Sidebar>
<Suspense fallback={<PostsGlimmer />}>
<Posts />
</Suspense>
</Suspense>
</ProfileLayout>
);
}
```

In this example, React can start streaming the page even earlier. Only `ProfileLayout` and `ProfileCover` must finish rendering first because they are not wrapped in any `<Suspense>` boundary. However, if `Sidebar`, `Friends`, or `Photos` need to load some data, React will send the HTML for the `BigSpinner` fallback instead. Then, as more data becomes available, more content will continue to be revealed until all of it becomes visible.

Streaming does not need to wait for React itself to load in the browser, or for your app to become interactive. The HTML content from the server will get progressively revealed before any of the `<script>` tags load.

[Read more about how streaming HTML works.](https://github.com/reactwg/react-18/discussions/37)
```

<Note>

Only Suspense-enabled data sources will activate the Suspense component. They include:

- Data fetching with Suspense-enabled frameworks like [Relay](https://relay.dev/docs/guided-tour/rendering/loading-states/) and [Next.js](https://nextjs.org/docs/getting-started/react-essentials)
- Lazy-loading component code with [`lazy`](/reference/react/lazy)

Suspense **does not** detect when data is fetched inside an Effect or event handler.

The exact way you would load data in the `Posts` component above depends on your framework. If you use a Suspense-enabled framework, you'll find the details in its data fetching documentation.

Suspense-enabled data fetching without the use of an opinionated framework is not yet supported. The requirements for implementing a Suspense-enabled data source are unstable and undocumented. An official API for integrating data sources with Suspense will be released in a future version of React.

</Note>

Specifying what goes into the shell {/*specifying-what-goes-into-the-shell*/}

The part of your app outside of any `<Suspense>` boundaries is called *the shell:*

```js {3-5,13,14}
function ProfilePage() {
return (
<ProfileLayout>
<ProfileCover />
<Suspense fallback={<BigSpinner />}>
<Sidebar>
<Friends />
<Photos />
</Sidebar>
<Suspense fallback={<PostsGlimmer />}>
<Posts />
</Suspense>
</Suspense>
</ProfileLayout>
);
}
```

It determines the earliest loading state that the user may see:

```js {3-5,13
<ProfileLayout>
<ProfileCover />
<BigSpinner />
</ProfileLayout>
```

If you wrap the whole app into a `<Suspense>` boundary at the root, the shell will only contain that spinner. However, that's not a pleasant user experience because seeing a big spinner on the screen can feel slower and more annoying than waiting a bit more and seeing the real layout. This is why usually you'll want to place the `<Suspense>` boundaries so that the shell feels *minimal but complete*--like a skeleton of the entire page layout.

The async call to `renderToReadableStream` will resolve to a `stream` as soon as the entire shell has been rendered. Usually, you'll start streaming then by creating and returning a response with that `stream`:

```js {5}
async function handler(request) {
const stream = await renderToReadableStream(<App />, {
bootstrapScripts: ['/main.js']
});
return new Response(stream, {
headers: { 'content-type': 'text/html' },
});
}
```

By the time the `stream` is returned, components in nested `<Suspense>` boundaries might still be loading data.

Logging crashes on the server {/*logging-crashes-on-the-server*/}

By default, all errors on the server are logged to console. You can override this behavior to log crash reports:

```js {4-7}
async function handler(request) {
const stream = await renderToReadableStream(<App />, {
bootstrapScripts: ['/main.js'],
```

```
onError(error) {
console.error(error);
logServerCrashReport(error);
}
});
return new Response(stream, {
headers: { 'content-type': 'text/html' },
});
}
```

If you provide a custom `onError` implementation, don't forget to also log errors to the console like above.

Recovering from errors inside the shell {/*recovering-from-errors-inside-the-shell*/}

In this example, the shell contains `ProfileLayout`, `ProfileCover`, and `PostsGlimmer`:

```
```js {3-5,7-8}
function ProfilePage() {
return (
<ProfileLayout>
<ProfileCover />
<Suspense fallback={<PostsGlimmer />}>
<Posts />
</Suspense>
</ProfileLayout>
);
}
```

If an error occurs while rendering those components, React won't have any meaningful HTML to send to the client. Wrap your `renderToReadableStream` call in a `try...catch` to send a fallback HTML that doesn't rely on server rendering as the last resort:

```
```js {2,13-18}
async function handler(request) {
try {
const stream = await renderToReadableStream(<App />, {
```

```
bootstrapScripts: ['/main.js'],
onError(error) {
console.error(error);
logServerCrashReport(error);
}
});
return new Response(stream, {
headers: { 'content-type': 'text/html' },
});
} catch (error) {
return new Response('<h1>Something went wrong</h1>', {
status: 500,
headers: { 'content-type': 'text/html' },
});
}
}
```

If there is an error while generating the shell, both `onError` and your `catch` block will fire. Use `onError` for error reporting and use the `catch` block to send the fallback HTML document. Your fallback HTML does not have to be an error page. Instead, you may include an alternative shell that renders your app on the client only.

Recovering from errors outside the shell {/*recovering-from-errors-outside-the-shell*/}

In this example, the `<Posts />` component is wrapped in `<Suspense>` so it is *not* a part of the shell:

```js {6}
function ProfilePage() {
return (
<ProfileLayout>
<ProfileCover />
<Suspense fallback={<PostsGlimmer />}>
<Posts />
</Suspense>
</ProfileLayout>
);
}
```

```

If an error happens in the `Posts` component or somewhere inside it, React will [try to recover from it:](/reference/react/Suspense#providing-a-fallback-for-server-errors-and-server-only-content)

1. It will emit the loading fallback for the closest `&lt;Suspense&gt;` boundary (`PostsGlimmer`) into the HTML.
2. It will "give up" on trying to render the `Posts` content on the server anymore.
3. When the JavaScript code loads on the client, React will *retry* rendering `Posts` on the client.

If retrying rendering `Posts` on the client *also* fails, React will throw the error on the client. As with all the errors thrown during rendering, the [closest parent error boundary](/reference/react/Component#static-getderivedstatefromerror) determines how to present the error to the user. In practice, this means that the user will see a loading indicator until it is certain that the error is not recoverable.

If retrying rendering `Posts` on the client succeeds, the loading fallback from the server will be replaced with the client rendering output. The user will not know that there was a server error. However, the server `onError` callback and the client [`onRecoverableError`](/reference/react-dom/client/hydrateRoot#hydrateroot) callbacks will fire so that you can get notified about the error.

---

### Setting the status code {/*setting-the-status-code*/}

Streaming introduces a tradeoff. You want to start streaming the page as early as possible so that the user can see the content sooner. However, once you start streaming, you can no longer set the response status code.

By [dividing your app](#specifying-what-goes-into-the-shell) into the shell (above all `&lt;Suspense&gt;` boundaries) and the rest of the content, you've already solved a part of this problem. If the shell errors, your `catch` block will run which lets you set the error status code. Otherwise, you know that the app may recover on the client, so you can send "OK".

```js {11}
async function handler(request) {
try {
const stream = await renderToReadableStream(<App />, {
bootstrapScripts: ['/main.js'],
onError(error) {
console.error(error);
logServerCrashReport(error);
}
});
return new Response(stream, {
```
```

status: 200,

headers: { 'content-type': 'text/html' },

});

} catch (error) {

return new Response('<h1>Something went wrong</h1>', {

status: 500,

headers: { 'content-type': 'text/html' },

});

}

}

```

If a component *outside* the shell (i.e. inside a `&lt;Suspense&gt;` boundary) throws an error, React will not stop rendering. This means that the `onError` callback will fire, but your code will continue running without getting into the `catch` block. This is because React will try to recover from that error on the client, [as described above.](#recovering-from-errors-outside-the-shell)

However, if you'd like, you can use the fact that something has errored to set the status code:

```js {3,7,13}

async function handler(request) {

try {

let didError = false;

const stream = await renderToReadableStream(<App />, {

bootstrapScripts: ['/main.js'],

onError(error) {

didError = true;

console.error(error);

logServerCrashReport(error);

}

});

return new Response(stream, {

status: didError ? 500 : 200,

headers: { 'content-type': 'text/html' },

});

} catch (error) {

return new Response('<h1>Something went wrong</h1>', {

status: 500,

headers: { 'content-type': 'text/html' },

});

}

}

```

This will only catch errors outside the shell that happened while generating the initial shell content, so it's not exhaustive. If knowing whether an error occurred for some content is critical, you can move it up into the shell.

Handling different errors in different ways {/*handling-different-errors-in-different-ways*/}

You can [create your own `Error` subclasses](https://javascript.info/custom-errors) and use the [`instanceof`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/instanceof) operator to check which error is thrown. For example, you can define a custom `NotFoundError` and throw it from your component. Then you can save the error in `onError` and do something different before returning the response depending on the error type:

```js {2-3,5-15,22,28,33}

async function handler(request) {

let didError = false;

let caughtError = null;

function getStatusCode() {

if (didError) {

if (caughtError instanceof NotFoundError) {

return 404;

} else {

return 500;

}

} else {

return 200;

}

}

try {

const stream = await renderToReadableStream(<App />, {

bootstrapScripts: ['/main.js'],

onError(error) {

didError = true;

```
caughtError = error;
console.error(error);
logServerCrashReport(error);
}
});
return new Response(stream, {
status: getStatusCode(),
headers: { 'content-type': 'text/html' },
});
} catch (error) {
return new Response('<h1>Something went wrong</h1>', {
status: getStatusCode(),
headers: { 'content-type': 'text/html' },
});
}
}
```

Keep in mind that once you emit the shell and start streaming, you can't change the status code.

Waiting for all content to load for crawlers and static generation
{/*waiting-for-all-content-to-load-for-crawlers-and-static-generation*/}

Streaming offers a better user experience because the user can see the content as it becomes available.

However, when a crawler visits your page, or if you're generating the pages at the build time, you might want to let all of the content load first and then produce the final HTML output instead of revealing it progressively.

You can wait for all the content to load by awaiting the `stream.allReady` Promise:

```js {12-15}
async function handler(request) {
try {
let didError = false;
const stream = await renderToReadableStream(<App />, {
bootstrapScripts: ['/main.js'],
onError(error) {
```

didError = true;
console.error(error);
logServerCrashReport(error);
}
});
let isCrawler = // ... depends on your bot detection strategy ...
if (isCrawler) {
await stream.allReady;
}
return new Response(stream, {
status: didError ? 500 : 200,
headers: { 'content-type': 'text/html' },
});
} catch (error) {
return new Response('<h1>Something went wrong</h1>', {
status: 500,
headers: { 'content-type': 'text/html' },
});
}
}
```

A regular visitor will get a stream of progressively loaded content. A crawler will receive the final HTML output after all the data loads. However, this also means that the crawler will have to wait for *all* data, some of which might be slow to load or error. Depending on your app, you could choose to send the shell to the crawlers too.

---

### Aborting server rendering {/*aborting-server-rendering*/}

You can force the server rendering to "give up" after a timeout:

```js {3,4-6,9}
async function handler(request) {
try {
const controller = new AbortController();
setTimeout(() => {
controller.abort();
}, 10000);

```
const stream = await renderToReadableStream(<App />, {
signal: controller.signal,
bootstrapScripts: ['/main.js'],
onError(error) {
didError = true;
console.error(error);
logServerCrashReport(error);
}
});
// ...
```

React will flush the remaining loading fallbacks as HTML, and will attempt to render the rest on the client.

---

title: Server React DOM APIs

<Intro>

The `react-dom/server` APIs let you render React components to HTML on the server. These APIs are only used on the server at the top level of your app to generate the initial HTML. A [framework](/learn/start-a-new-react-project#production-grade-react-frameworks) may call them for you. Most of your components don't need to import or use them.

</Intro>

Server APIs for Node.js Streams {/*server-apis-for-nodejs-streams*/}

These methods are only available in the environments with [Node.js Streams:](https://nodejs.org/api/stream.html)

* [`renderToPipeableStream`](/reference/react-dom/server/renderToPipeableStream) renders a React tree to a pipeable [Node.js Stream.](https://nodejs.org/api/stream.html)
* [`renderToStaticNodeStream`](/reference/react-dom/server/renderToStaticNodeStream) renders a non-interactive React tree to a [Node.js Readable Stream.](https://nodejs.org/api/stream.html#readable-streams)

Server APIs for Web Streams {/*server-apis-for-web-streams*/}

These methods are only available in the environments with [Web Streams](https://developer.mozilla.org/en-US/docs/Web/API/Streams_API), which includes browsers,

Deno, and some modern edge runtimes:

* [`renderToReadableStream`](/reference/react-dom/server/renderToReadableStream) renders a React tree to a [Readable Web Stream.](https://developer.mozilla.org/en-US/docs/Web/API/ReadableStream)

Server APIs for non-streaming environments {/*server-apis-for-non-streaming-environments*/}

These methods can be used in the environments that don't support streams:

* [`renderToString`](/reference/react-dom/server/renderToString) renders a React tree to a string.
* [`renderToStaticMarkup`](/reference/react-dom/server/renderToStaticMarkup) renders a non-interactive React tree to a string.

They have limited functionality compared to the streaming APIs.

Deprecated server APIs {/*deprecated-server-apis*/}

<Deprecated>

These APIs will be removed in a future major version of React.

</Deprecated>

* [`renderToNodeStream`](/reference/react-dom/server/renderToNodeStream) renders a React tree to a [Node.js Readable stream.](https://nodejs.org/api/stream.html#readable-streams) (Deprecated.)

---

title: renderToStaticNodeStream

<Intro>

`renderToStaticNodeStream` renders a non-interactive React tree to a [Node.js Readable Stream.](https://nodejs.org/api/stream.html#readable-streams)

```js
const stream = renderToStaticNodeStream(reactNode)
```

</Intro>

<InlineToc />

Reference {/*reference*/}

`renderToStaticNodeStream(reactNode)` {/*rendertostaticnodestream*/}

On the server, call `renderToStaticNodeStream` to get a [Node.js Readable Stream](https://nodejs.org/api/stream.html#readable-streams).

```js
import { renderToStaticNodeStream } from 'react-dom/server';

const stream = renderToStaticNodeStream(<Page />);

stream.pipe(response);
```

[See more examples below.](#usage)

The stream will produce non-interactive HTML output of your React components.

Parameters {/*parameters*/}

* `reactNode`: A React node you want to render to HTML. For example, a JSX element like `<Page />`.

Returns {/*returns*/}

A [Node.js Readable Stream](https://nodejs.org/api/stream.html#readable-streams) that outputs an HTML string. The resulting HTML can't be hydrated on the client.

Caveats {/*caveats*/}

* `renderToStaticNodeStream` output cannot be hydrated.

* This method will wait for all [Suspense boundaries](/reference/react/Suspense) to complete before returning any output.

* As of React 18, this method buffers all of its output, so it doesn't actually provide any streaming benefits.

* The returned stream is a byte stream encoded in utf-8. If you need a stream in another encoding, take a look at a project like [iconv-lite](https://www.npmjs.com/package/iconv-lite), which provides transform streams for transcoding text.

Usage {/*usage*/}

Rendering a React tree as static HTML to a Node.js Readable Stream {/*rendering-a-react-tree-as-static-html-to-a-nodejs-readable-stream*/}

Call `renderToStaticNodeStream` to get a [Node.js Readable Stream](https://nodejs.org/api/stream.html#readable-streams) which you can pipe to your server response:

```js {5-6}
```

```
import { renderToStaticNodeStream } from 'react-dom/server';

// The route handler syntax depends on your backend framework
app.use('/', (request, response) => {
const stream = renderToStaticNodeStream(<Page />);
stream.pipe(response);
});
```

The stream will produce the initial non-interactive HTML output of your React components.

<Pitfall>

This method renders **non-interactive HTML that cannot be hydrated.** This is useful if you want to use React as a simple static page generator, or if you're rendering completely static content like emails.

Interactive apps should use [`renderToPipeableStream`](/reference/react-dom/server/renderToPipeableStream) on the server and [`hydrateRoot`](/reference/react-dom/client/hydrateRoot) on the client.

</Pitfall>

---

title: renderToStaticMarkup

<Intro>

`renderToStaticMarkup` renders a non-interactive React tree to an HTML string.

```js
const html = renderToStaticMarkup(reactNode)
```

</Intro>

<InlineToc />

Reference {/*reference*/}

`renderToStaticMarkup(reactNode)` {/*rendertostaticmarkup*/}

On the server, call `renderToStaticMarkup` to render your app to HTML.

```js
import { renderToStaticMarkup } from 'react-dom/server';
```

```
const html = renderToStaticMarkup(<Page />);
```

It will produce non-interactive HTML output of your React components.

[See more examples below.](#usage)

Parameters {/*parameters*/}

* `reactNode`: A React node you want to render to HTML. For example, a JSX node like `<Page />`.

Returns {/*returns*/}

An HTML string.

Caveats {/*caveats*/}

* `renderToStaticMarkup` output cannot be hydrated.

* `renderToStaticMarkup` has limited Suspense support. If a component suspends, `renderToStaticMarkup` immediately sends its fallback as HTML.

* `renderToStaticMarkup` works in the browser, but using it in the client code is not recommended. If you need to render a component to HTML in the browser, [get the HTML by rendering it into a DOM node.](/reference/react-dom/server/renderToString#removing-rendertostring-from-the-client-code)

Usage {/*usage*/}

Rendering a non-interactive React tree as HTML to a string {/*rendering-a-non-interactive-react-tree-as-html-to-a-string*/}

Call `renderToStaticMarkup` to render your app to an HTML string which you can send with your server response:

```js {5-6}
import { renderToStaticMarkup } from 'react-dom/server';

// The route handler syntax depends on your backend framework
app.use('/', (request, response) => {
const html = renderToStaticMarkup(<Page />);
response.send(html);
});
```

This will produce the initial non-interactive HTML output of your React components.

<Pitfall>

This method renders **non-interactive HTML that cannot be hydrated.** This is useful if you want to use React as a simple static page generator, or if you're rendering completely static content like emails.

Interactive apps should use [`renderToString`](/reference/react-dom/server/renderToString) on the server and [`hydrateRoot`](/reference/react-dom/client/hydrateRoot) on the client.

</Pitfall>

---

title: renderToPipeableStream

<Intro>

`renderToPipeableStream` renders a React tree to a pipeable [Node.js Stream.](https://nodejs.org/api/stream.html)

```js
const { pipe, abort } = renderToPipeableStream(reactNode, options?)
```

</Intro>

<InlineToc />

<Note>

This API is specific to Node.js. Environments with [Web Streams,](https://developer.mozilla.org/en-US/docs/Web/API/Streams_API) like Deno and modern edge runtimes, should use [`renderToReadableStream`](/reference/react-dom/server/renderToReadableStream) instead.

</Note>

Reference {/*reference*/}

`renderToPipeableStream(reactNode, options?)` {/*rendertopipeablestream*/}

Call `renderToPipeableStream` to render your React tree as HTML into a [Node.js Stream.](https://nodejs.org/api/stream.html#writable-streams)

```js
import { renderToPipeableStream } from 'react-dom/server';

const { pipe } = renderToPipeableStream(<App />, {
bootstrapScripts: ['/main.js'],
onShellReady() {
```

response.setHeader('content-type', 'text/html');

pipe(response);

}

});

```

On the client, call [`hydrateRoot`](/reference/react-dom/client/hydrateRoot) to make the server-generated HTML interactive.

[See more examples below.](#usage)

#### Parameters {/*parameters*/}

* `reactNode`: A React node you want to render to HTML. For example, a JSX element like `<App />`. It is expected to represent the entire document, so the `App` component should render the `<html>` tag.

* **optional** `options`: An object with streaming options.

* **optional** `bootstrapScriptContent`: If specified, this string will be placed in an inline `<script>` tag.

* **optional** `bootstrapScripts`: An array of string URLs for the `<script>` tags to emit on the page. Use this to include the `<script>` that calls [`hydrateRoot`.](/reference/react-dom/client/hydrateRoot) Omit it if you don't want to run React on the client at all.

* **optional** `bootstrapModules`: Like `bootstrapScripts`, but emits [`<script type="module">`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules) instead.

* **optional** `identifierPrefix`: A string prefix React uses for IDs generated by [`useId`.](/reference/react/useId) Useful to avoid conflicts when using multiple roots on the same page. Must be the same prefix as passed to [`hydrateRoot`.](/reference/react-dom/client/hydrateRoot#parameters)

* **optional** `namespaceURI`: A string with the root [namespace URI](https://developer.mozilla.org/en-US/docs/Web/API/Document/createElementNS#important_namespace_uris) for the stream. Defaults to regular HTML. Pass `'http://www.w3.org/2000/svg'` for SVG or `'http://www.w3.org/1998/Math/MathML'` for MathML.

* **optional** `nonce`: A [`nonce`](http://developer.mozilla.org/en-US/docs/Web/HTML/Element/script#nonce) string to allow scripts for [`script-src` Content-Security-Policy](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/script-src).

* **optional** `onAllReady`: A callback that fires when all rendering is complete, including both the [shell](#specifying-what-goes-into-the-shell) and all additional [content.](#streaming-more-content-as-it-loads) You can use this instead of `onShellReady` [for crawlers and static generation.](#waiting-for-all-content-to-load-for-crawlers-and-static-generation) If you start streaming here, you won't get any progressive loading. The stream will contain the final HTML.

* **optional** `onError`: A callback that fires whenever there is a server error, whether [recoverable](#recovering-from-errors-outside-the-shell) or [not.](#recovering-from-errors-inside-the-shell) By default, this only calls `console.error`. If you override it to [log crash reports,](#logging-crashes-on-the-server) make sure that you still call `console.error`. You can also use it to [adjust the status code](#setting-the-status-code) before the shell is emitted.
```

* **optional** `onShellReady`: A callback that fires right after the [initial shell](#specifying-what-goes-into-the-shell) has been rendered. You can [set the status code](#setting-the-status-code) and call `pipe` here to start streaming. React will [stream the additional content](#streaming-more-content-as-it-loads) after the shell along with the inline `<script>` tags that place that replace the HTML loading fallbacks with the content.

* **optional** `onShellError`: A callback that fires if there was an error rendering the initial shell. It receives the error as an argument. No bytes were emitted from the stream yet, and neither `onShellReady` nor `onAllReady` will get called, so you can [output a fallback HTML shell.](#recovering-from-errors-inside-the-shell)

* **optional** `progressiveChunkSize`: The number of bytes in a chunk. [Read more about the default heuristic.](https://github.com/facebook/react/blob/14c2be8dac2d5482fda8a0906a31d239df8551fc/packages/react-server/src/ReactFizzServer.js#L210-L225)

Returns {/*returns*/}

`renderToPipeableStream` returns an object with two methods:

* `pipe` outputs the HTML into the provided [Writable Node.js Stream.](https://nodejs.org/api/stream.html#writable-streams) Call `pipe` in `onShellReady` if you want to enable streaming, or in `onAllReady` for crawlers and static generation.

* `abort` lets you [abort server rendering](#aborting-server-rendering) and render the rest on the client.

Usage {/*usage*/}

Rendering a React tree as HTML to a Node.js Stream {/*rendering-a-react-tree-as-html-to-a-nodejs-stream*/}

Call `renderToPipeableStream` to render your React tree as HTML into a [Node.js Stream:](https://nodejs.org/api/stream.html#writable-streams)

```js [[1, 5, "<App />"], [2, 6, "['/main.js']"]]
import { renderToPipeableStream } from 'react-dom/server';

// The route handler syntax depends on your backend framework
app.use('/', (request, response) => {
const { pipe } = renderToPipeableStream(<App />, {
bootstrapScripts: ['/main.js'],
onShellReady() {
response.setHeader('content-type', 'text/html');
pipe(response);
}
});
});
```

```

Along with the <CodeStep step={1}>root component</CodeStep>, you need to provide a list of <CodeStep step={2}>bootstrap `&lt;script&gt;` paths</CodeStep>. Your root component should return **the entire document including the root `&lt;html&gt;` tag.**

For example, it might look like this:

```js [[1, 1, "App"]]
export default function App() {
return (
<html>
<head>
<meta charSet="utf-8" />
<meta name="viewport" content="width=device-width, initial-scale=1" />
<link rel="stylesheet" href="/styles.css"></link>
<title>My app</title>
</head>
<body>
<Router />
</body>
</html>
);
}
```

React will inject the [doctype](https://developer.mozilla.org/en-US/docs/Glossary/Doctype) and your <CodeStep step={2}>bootstrap `<script>` tags</CodeStep> into the resulting HTML stream:

```html [[2, 5, "/main.js"]]
<!DOCTYPE html>
<html>
<!-- ... HTML from your components ... -->
</html>
<script src="/main.js" async=""></script>
```

On the client, your bootstrap script should [hydrate the entire `document` with a call to `hydrateRoot`:](/reference/react-dom/client/hydrateRoot#hydrating-an-entire-document)

```js [[1, 4, "<App />"]]

```
import { hydrateRoot } from 'react-dom/client';
import App from './App.js';

hydrateRoot(document, <App />);
```

This will attach event listeners to the server-generated HTML and make it interactive.

<DeepDive>

Reading CSS and JS asset paths from the build output
{/*reading-css-and-js-asset-paths-from-the-build-output*/}

The final asset URLs (like JavaScript and CSS files) are often hashed after the build. For example, instead of `styles.css` you might end up with `styles.123456.css`. Hashing static asset filenames guarantees that every distinct build of the same asset will have a different filename. This is useful because it lets you safely enable long-term caching for static assets: a file with a certain name would never change content.

However, if you don't know the asset URLs until after the build, there's no way for you to put them in the source code. For example, hardcoding `"/styles.css"` into JSX like earlier wouldn't work. To keep them out of your source code, your root component can read the real filenames from a map passed as a prop:

```js {1,6}
export default function App({ assetMap }) {
return (
<html>
<head>
...
<link rel="stylesheet" href={assetMap['styles.css']}></link>
...
</head>
...
</html>
);
}
```

On the server, render `<App assetMap={assetMap} />` and pass your `assetMap` with the asset URLs:

```js {1-5,8,9}
// You'd need to get this JSON from your build tooling, e.g. read it from the build output.
const assetMap = {
```

```
'styles.css': '/styles.123456.css',
'main.js': '/main.123456.js'
};

app.use('/', (request, response) => {
const { pipe } = renderToPipeableStream(<App assetMap={assetMap} />, {
bootstrapScripts: [assetMap['main.js']],
onShellReady() {
response.setHeader('content-type', 'text/html');
pipe(response);
}
});
});
```

Since your server is now rendering `<App assetMap={assetMap} />`, you need to render it with `assetMap` on the client too to avoid hydration errors. You can serialize and pass `assetMap` to the client like this:

```js {9-10}
// You'd need to get this JSON from your build tooling.
const assetMap = {
'styles.css': '/styles.123456.css',
'main.js': '/main.123456.js'
};

app.use('/', (request, response) => {
const { pipe } = renderToPipeableStream(<App assetMap={assetMap} />, {
// Careful: It's safe to stringify() this because this data isn't user-generated.
bootstrapScriptContent: `window.assetMap = ${JSON.stringify(assetMap)};`,
bootstrapScripts: [assetMap['main.js']],
onShellReady() {
response.setHeader('content-type', 'text/html');
pipe(response);
}
});
});
```

In the example above, the `bootstrapScriptContent` option adds an extra inline `<script>` tag that sets the global `window.assetMap` variable on the client. This lets the client code read the same `assetMap`:

```js {4}
import { hydrateRoot } from 'react-dom/client';
import App from './App.js';

hydrateRoot(document, <App assetMap={window.assetMap} />);
```

Both client and server render `App` with the same `assetMap` prop, so there are no hydration errors.

</DeepDive>

Streaming more content as it loads {/*streaming-more-content-as-it-loads*/}

Streaming allows the user to start seeing the content even before all the data has loaded on the server. For example, consider a profile page that shows a cover, a sidebar with friends and photos, and a list of posts:

```js
function ProfilePage() {
return (
<ProfileLayout>
<ProfileCover />
<Sidebar>
<Friends />
<Photos />
</Sidebar>
<Posts />
</ProfileLayout>
);
}
```

Imagine that loading data for `<Posts />` takes some time. Ideally, you'd want to show the rest of the profile page content to the user without waiting for the posts. To do this, [wrap `Posts` in a `<Suspense>` boundary:](/reference/react/Suspense#displaying-a-fallback-while-content-is-loading)

```js {9,11}
function ProfilePage() {
```

return (

<ProfileLayout>

<ProfileCover />

<Sidebar>

<Friends />

<Photos />

</Sidebar>

<Suspense fallback={<PostsGlimmer />}>

<Posts />

</Suspense>

</ProfileLayout>

);

}

```

This tells React to start streaming the HTML before `Posts` loads its data. React will send the HTML for the loading fallback (`PostsGlimmer`) first, and then, when `Posts` finishes loading its data, React will send the remaining HTML along with an inline `<script>` tag that replaces the loading fallback with that HTML. From the user's perspective, the page will first appear with the `PostsGlimmer`, later replaced by the `Posts`.

You can further [nest `<Suspense>` boundaries](/reference/react/Suspense#revealing-nested-content-as-it-loads) to create a more granular loading sequence:

```js {5,13}

function ProfilePage() {

return (

<ProfileLayout>

<ProfileCover />

<Suspense fallback={<BigSpinner />}>

<Sidebar>

<Friends />

<Photos />

</Sidebar>

<Suspense fallback={<PostsGlimmer />}>

<Posts />

</Suspense>

</Suspense>

</ProfileLayout>

);

}

```

In this example, React can start streaming the page even earlier. Only `ProfileLayout` and `ProfileCover` must finish rendering first because they are not wrapped in any `<Suspense>` boundary. However, if `Sidebar`, `Friends`, or `Photos` need to load some data, React will send the HTML for the `BigSpinner` fallback instead. Then, as more data becomes available, more content will continue to be revealed until all of it becomes visible.

Streaming does not need to wait for React itself to load in the browser, or for your app to become interactive. The HTML content from the server will get progressively revealed before any of the `<script>` tags load.

[Read more about how streaming HTML works.](https://github.com/reactwg/react-18/discussions/37)

<Note>

Only Suspense-enabled data sources will activate the Suspense component. They include:

- Data fetching with Suspense-enabled frameworks like [Relay](https://relay.dev/docs/guided-tour/rendering/loading-states/) and [Next.js](https://nextjs.org/docs/getting-started/react-essentials)
- Lazy-loading component code with [`lazy`](/reference/react/lazy)

Suspense **does not** detect when data is fetched inside an Effect or event handler.

The exact way you would load data in the `Posts` component above depends on your framework. If you use a Suspense-enabled framework, you'll find the details in its data fetching documentation.

Suspense-enabled data fetching without the use of an opinionated framework is not yet supported. The requirements for implementing a Suspense-enabled data source are unstable and undocumented. An official API for integrating data sources with Suspense will be released in a future version of React.

</Note>

Specifying what goes into the shell {/*specifying-what-goes-into-the-shell*/}

The part of your app outside of any `<Suspense>` boundaries is called *the shell:*

```js {3-5,13,14}
function ProfilePage() {
return (
<ProfileLayout>
<ProfileCover />

<Suspense fallback={<BigSpinner />}>

<Sidebar>

<Friends />

<Photos />

</Sidebar>

<Suspense fallback={<PostsGlimmer />}>

<Posts />

</Suspense>

</Suspense>

</ProfileLayout>

);

}

```

It determines the earliest loading state that the user may see:

```js {3-5,13

<ProfileLayout>

<ProfileCover />

<BigSpinner />

</ProfileLayout>

```

If you wrap the whole app into a `<Suspense>` boundary at the root, the shell will only contain that spinner. However, that's not a pleasant user experience because seeing a big spinner on the screen can feel slower and more annoying than waiting a bit more and seeing the real layout. This is why usually you'll want to place the `<Suspense>` boundaries so that the shell feels *minimal but complete*--like a skeleton of the entire page layout.

The `onShellReady` callback fires when the entire shell has been rendered. Usually, you'll start streaming then:

```js {3-6}

const { pipe } = renderToPipeableStream(<App />, {

bootstrapScripts: ['/main.js'],

onShellReady() {

response.setHeader('content-type', 'text/html');

pipe(response);

}

});

```

By the time `onShellReady` fires, components in nested `<Suspense>` boundaries might still be loading data.

Logging crashes on the server {/*logging-crashes-on-the-server*/}

By default, all errors on the server are logged to console. You can override this behavior to log crash reports:

```js {7-10}
const { pipe } = renderToPipeableStream(<App />, {
bootstrapScripts: ['/main.js'],
onShellReady() {
response.setHeader('content-type', 'text/html');
pipe(response);
},
onError(error) {
console.error(error);
logServerCrashReport(error);
}
});
```

If you provide a custom `onError` implementation, don't forget to also log errors to the console like above.

Recovering from errors inside the shell {/*recovering-from-errors-inside-the-shell*/}

In this example, the shell contains `ProfileLayout`, `ProfileCover`, and `PostsGlimmer`:

```js {3-5,7-8}
function ProfilePage() {
return (
<ProfileLayout>
<ProfileCover />
<Suspense fallback={<PostsGlimmer />}>
<Posts />
</Suspense>
```
```

</ProfileLayout>
);
}
```

If an error occurs while rendering those components, React won't have any meaningful HTML to send to the client. Override `onShellError` to send a fallback HTML that doesn't rely on server rendering as the last resort:

```js {7-11}
const { pipe } = renderToPipeableStream(<App />, {
bootstrapScripts: ['/main.js'],
onShellReady() {
response.setHeader('content-type', 'text/html');
pipe(response);
},
onShellError(error) {
response.statusCode = 500;
response.setHeader('content-type', 'text/html');
response.send('<h1>Something went wrong</h1>');
},
onError(error) {
console.error(error);
logServerCrashReport(error);
}
});
```

If there is an error while generating the shell, both `onError` and `onShellError` will fire. Use `onError` for error reporting and use `onShellError` to send the fallback HTML document. Your fallback HTML does not have to be an error page. Instead, you may include an alternative shell that renders your app on the client only.

Recovering from errors outside the shell {/*recovering-from-errors-outside-the-shell*/}

In this example, the `<Posts />` component is wrapped in `<Suspense>` so it is *not* a part of the shell:

```js {6}
function ProfilePage() {
return (

<ProfileLayout>

<ProfileCover />

<Suspense fallback={<PostsGlimmer />}>

<Posts />

</Suspense>

</ProfileLayout>

);

}

```

If an error happens in the `Posts` component or somewhere inside it, React will [try to recover from it:](/reference/react/Suspense#providing-a-fallback-for-server-errors-and-server-only-content)

1. It will emit the loading fallback for the closest `<Suspense>` boundary (`PostsGlimmer`) into the HTML.

2. It will "give up" on trying to render the `Posts` content on the server anymore.

3. When the JavaScript code loads on the client, React will *retry* rendering `Posts` on the client.

If retrying rendering `Posts` on the client *also* fails, React will throw the error on the client. As with all the errors thrown during rendering, the [closest parent error boundary](/reference/react/Component#static-getderivedstatefromerror) determines how to present the error to the user. In practice, this means that the user will see a loading indicator until it is certain that the error is not recoverable.

If retrying rendering `Posts` on the client succeeds, the loading fallback from the server will be replaced with the client rendering output. The user will not know that there was a server error. However, the server `onError` callback and the client [`onRecoverableError`](/reference/react-dom/client/hydrateRoot#hydrateroot) callbacks will fire so that you can get notified about the error.

Setting the status code {/*setting-the-status-code*/}

Streaming introduces a tradeoff. You want to start streaming the page as early as possible so that the user can see the content sooner. However, once you start streaming, you can no longer set the response status code.

By [dividing your app](#specifying-what-goes-into-the-shell) into the shell (above all `<Suspense>` boundaries) and the rest of the content, you've already solved a part of this problem. If the shell errors, you'll get the `onShellError` callback which lets you set the error status code. Otherwise, you know that the app may recover on the client, so you can send "OK".

```js {4}

const { pipe } = renderToPipeableStream(<App />, {

bootstrapScripts: ['/main.js'],

```
onShellReady() {
response.statusCode = 200;
response.setHeader('content-type', 'text/html');
pipe(response);
},
onShellError(error) {
response.statusCode = 500;
response.setHeader('content-type', 'text/html');
response.send('<h1>Something went wrong</h1>');
},
onError(error) {
console.error(error);
logServerCrashReport(error);
}
});
```

If a component *outside* the shell (i.e. inside a `<Suspense>` boundary) throws an error, React will not stop rendering. This means that the `onError` callback will fire, but you will still get `onShellReady` instead of `onShellError`. This is because React will try to recover from that error on the client, [as described above.](#recovering-from-errors-outside-the-shell)

However, if you'd like, you can use the fact that something has errored to set the status code:

```js {1,6,16}
let didError = false;

const { pipe } = renderToPipeableStream(<App />, {
bootstrapScripts: ['/main.js'],
onShellReady() {
response.statusCode = didError ? 500 : 200;
response.setHeader('content-type', 'text/html');
pipe(response);
},
onShellError(error) {
response.statusCode = 500;
response.setHeader('content-type', 'text/html');
response.send('<h1>Something went wrong</h1>');
},
```

```
onError(error) {
didError = true;
console.error(error);
logServerCrashReport(error);
}
});
```

This will only catch errors outside the shell that happened while generating the initial shell content, so it's not exhaustive. If knowing whether an error occurred for some content is critical, you can move it up into the shell.

Handling different errors in different ways {/*handling-different-errors-in-different-ways*/}

You can [create your own `Error` subclasses](https://javascript.info/custom-errors) and use the [`instanceof`](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/instanceof) operator to check which error is thrown. For example, you can define a custom `NotFoundError` and throw it from your component. Then your `onError`, `onShellReady`, and `onShellError` callbacks can do something different depending on the error type:

```js {2,4-14,19,24,30}
let didError = false;
let caughtError = null;

function getStatusCode() {
if (didError) {
if (caughtError instanceof NotFoundError) {
return 404;
} else {
return 500;
}
} else {
return 200;
}
}

const { pipe } = renderToPipeableStream(<App />, {
bootstrapScripts: ['/main.js'],
onShellReady() {
response.statusCode = getStatusCode();
```

```
response.setHeader('content-type', 'text/html');
pipe(response);
},
onShellError(error) {
response.statusCode = getStatusCode();
response.setHeader('content-type', 'text/html');
response.send('<h1>Something went wrong</h1>');
},
onError(error) {
didError = true;
caughtError = error;
console.error(error);
logServerCrashReport(error);
}
});
```

Keep in mind that once you emit the shell and start streaming, you can't change the status code.

Waiting for all content to load for crawlers and static generation
{/*waiting-for-all-content-to-load-for-crawlers-and-static-generation*/}

Streaming offers a better user experience because the user can see the content as it becomes available.

However, when a crawler visits your page, or if you're generating the pages at the build time, you might want to let all of the content load first and then produce the final HTML output instead of revealing it progressively.

You can wait for all the content to load using the `onAllReady` callback:

```js {2,7,11,18-24}
let didError = false;
let isCrawler = // ... depends on your bot detection strategy ...
const { pipe } = renderToPipeableStream(<App />, {
bootstrapScripts: ['/main.js'],
onShellReady() {
if (!isCrawler) {
response.statusCode = didError ? 500 : 200;
```

```
response.setHeader('content-type', 'text/html');
pipe(response);
}
},
onShellError(error) {
response.statusCode = 500;
response.setHeader('content-type', 'text/html');
response.send('<h1>Something went wrong</h1>');
},
onAllReady() {
if (isCrawler) {
response.statusCode = didError ? 500 : 200;
response.setHeader('content-type', 'text/html');
pipe(response);
}
},
onError(error) {
didError = true;
console.error(error);
logServerCrashReport(error);
}
});
```

A regular visitor will get a stream of progressively loaded content. A crawler will receive the final HTML output after all the data loads. However, this also means that the crawler will have to wait for *all* data, some of which might be slow to load or error. Depending on your app, you could choose to send the shell to the crawlers too.

Aborting server rendering {/*aborting-server-rendering*/}

You can force the server rendering to "give up" after a timeout:

```js {1,5-7}
const { pipe, abort } = renderToPipeableStream(<App />, {
// ...
});
```

```
setTimeout(() => {
abort();
}, 10000);
```

React will flush the remaining loading fallbacks as HTML, and will attempt to render the rest on the client.

---

title: renderToNodeStream

<Deprecated>

This API will be removed in a future major version of React. Use [`renderToPipeableStream`](/reference/react-dom/server/renderToPipeableStream) instead.

</Deprecated>

<Intro>

`renderToNodeStream` renders a React tree to a [Node.js Readable Stream.](https://nodejs.org/api/stream.html#readable-streams)

```js
const stream = renderToNodeStream(reactNode)
```

</Intro>

<InlineToc />

Reference {/*reference*/}

`renderToNodeStream(reactNode)` {/*rendertonodestream*/}

On the server, call `renderToNodeStream` to get a [Node.js Readable Stream](https://nodejs.org/api/stream.html#readable-streams) which you can pipe into the response.

```js
import { renderToNodeStream } from 'react-dom/server';

const stream = renderToNodeStream(<App />);
stream.pipe(response);
```

On the client, call [`hydrateRoot`](/reference/react-dom/client/hydrateRoot) to make the server-generated HTML interactive.

[See more examples below.](#usage)

Parameters {/*parameters*/}

* `reactNode`: A React node you want to render to HTML. For example, a JSX element like `<App />`.

Returns {/*returns*/}

A [Node.js Readable Stream](https://nodejs.org/api/stream.html#readable-streams) that outputs an HTML string.

Caveats {/*caveats*/}

* This method will wait for all [Suspense boundaries](/reference/react/Suspense) to complete before returning any output.

* As of React 18, this method buffers all of its output, so it doesn't actually provide any streaming benefits. This is why it's recommended that you migrate to [`renderToPipeableStream`](/reference/react-dom/server/renderToPipeableStream) instead.

* The returned stream is a byte stream encoded in utf-8. If you need a stream in another encoding, take a look at a project like [iconv-lite](https://www.npmjs.com/package/iconv-lite), which provides transform streams for transcoding text.

Usage {/*usage*/}

Rendering a React tree as HTML to a Node.js Readable Stream {/*rendering-a-react-tree-as-html-to-a-nodejs-readable-stream*/}

Call `renderToNodeStream` to get a [Node.js Readable Stream](https://nodejs.org/api/stream.html#readable-streams) which you can pipe to your server response:

```js {5-6}
import { renderToNodeStream } from 'react-dom/server';

// The route handler syntax depends on your backend framework
app.use('/', (request, response) => {
const stream = renderToNodeStream(&lt;App /&gt;);
stream.pipe(response);
});
```

The stream will produce the initial non-interactive HTML output of your React components. On the client, you will need to call [`hydrateRoot`](/reference/react-dom/client/hydrateRoot) to *hydrate* that

server-generated HTML and make it interactive.

---

title: renderToString

<Pitfall>

`renderToString` does not support streaming or waiting for data. [See the alternatives.](#alternatives)

</Pitfall>

<Intro>

`renderToString` renders a React tree to an HTML string.

```js
const html = renderToString(reactNode)
```

</Intro>

<InlineToc />

Reference {/*reference*/}

`renderToString(reactNode)` {/*rendertostring*/}

On the server, call `renderToString` to render your app to HTML.

```js
import { renderToString } from 'react-dom/server';

const html = renderToString(<App />);
```

On the client, call [`hydrateRoot`](/reference/react-dom/client/hydrateRoot) to make the server-generated HTML interactive.

[See more examples below.](#usage)

Parameters {/*parameters*/}

* `reactNode`: A React node you want to render to HTML. For example, a JSX node like `<App />`.

Returns {/*returns*/}

An HTML string.

Caveats {/*caveats*/}

* `renderToString` has limited Suspense support. If a component suspends, `renderToString` immediately sends its fallback as HTML.

* `renderToString` works in the browser, but using it in the client code is [not recommended.](#removing-rendertostring-from-the-client-code)

Usage {/*usage*/}

Rendering a React tree as HTML to a string {/*rendering-a-react-tree-as-html-to-a-string*/}

Call `renderToString` to render your app to an HTML string which you can send with your server response:

```js {5-6}
import { renderToString } from 'react-dom/server';

// The route handler syntax depends on your backend framework
app.use('/', (request, response) => {
const html = renderToString(<App />);
response.send(html);
});
```

This will produce the initial non-interactive HTML output of your React components. On the client, you will need to call [`hydrateRoot`](/reference/react-dom/client/hydrateRoot) to *hydrate* that server-generated HTML and make it interactive.

<Pitfall>

`renderToString` does not support streaming or waiting for data. [See the alternatives.](#alternatives)

</Pitfall>

Alternatives {/*alternatives*/}

Migrating from `renderToString` to a streaming method on the server {/*migrating-from-rendertostring-to-a-streaming-method-on-the-server*/}

`renderToString` returns a string immediately, so it does not support streaming or waiting for data.

When possible, we recommend using these fully-featured alternatives:

* If you use Node.js, use [`renderToPipeableStream`.](/reference/react-dom/server/renderToPipeableStream)

* If you use Deno or a modern edge runtime with [Web Streams](https://developer.mozilla.org/en-US/docs/Web/API/Streams_API), use [`renderToReadableStream`.](/reference/react-dom/server/renderToReadableStream)

You can continue using `renderToString` if your server environment does not support streams.

Removing `renderToString` from the client code {/*removing-rendertostring-from-the-client-code*/}

Sometimes, `renderToString` is used on the client to convert some component to HTML.

```js {1-2}
// ■ Unnecessary: using renderToString on the client
import { renderToString } from 'react-dom/server';

const html = renderToString(<MyIcon />);
console.log(html); // For example, "<svg>...</svg>"
```

Importing `react-dom/server` **on the client** unnecessarily increases your bundle size and should be avoided. If you need to render some component to HTML in the browser, use [`createRoot`](/reference/react-dom/client/createRoot) and read HTML from the DOM:

```js
import { createRoot } from 'react-dom/client';
import { flushSync } from 'react-dom';

const div = document.createElement('div');
const root = createRoot(div);
flushSync(() => {
root.render(<MyIcon />);
});
console.log(div.innerHTML); // For example, "<svg>...</svg>"
```

The [`flushSync`](/reference/react-dom/flushSync) call is necessary so that the DOM is updated before reading its [`innerHTML`](https://developer.mozilla.org/en-US/docs/Web/API/Element/innerHTML) property.

Troubleshooting {/*troubleshooting*/}

When a component suspends, the HTML always contains a fallback {/*when-a-component-suspends-the-html-always-contains-a-fallback*/}

`renderToString` does not fully support Suspense.

If some component suspends (for example, because it's defined with [`lazy`](/reference/react/lazy) or fetches data), `renderToString` will not wait for its content to resolve. Instead, `renderToString` will find the closest [`<Suspense>`](/reference/react/Suspense) boundary above it and render its `fallback` prop in the HTML. The content will not appear until the client code loads.

To solve this, use one of the [recommended streaming solutions.](#migrating-from-rendertostring-to-a-streaming-method-on-the-server) They can stream content in chunks as it resolves on the server so that the user sees the page being progressively filled in before the client code loads.

---

title: Special Props Warning

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, you can't read `props.key` from a component. 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} />` and read `props.id`). While this may seem redundant, it's important to separate app logic from hints to React.

---

title: Rules of Hooks

You are probably here because you got the following error message:

<ConsoleBlock level="error">

Hooks can only be called inside the body of a function component.

</ConsoleBlock>

There are three common reasons you might be seeing it:

1. You might be **breaking the Rules of Hooks**.
2. You might have **mismatching versions** of React and React DOM.
3. You might have **more than one copy of React** in the same app.

Let's look at each of these cases.

Breaking Rules of Hooks {/*breaking-rules-of-hooks*/}

Functions whose names start with `use` are called [*Hooks*](/reference/react) in React.

Don’t call Hooks inside loops, conditions, or nested functions. Instead, always use Hooks at the top level of your React function, before any early returns. You can only call Hooks while React is rendering a function component:

* ■ Call them at the top level in the body of a [function component](/learn/your-first-component).
* ■ Call them at the top level in the body of a [custom Hook](/learn/reusing-logic-with-custom-hooks).

```js{2-3,8-9}
function Counter() {
// ■ Good: top-level in a function component
const [count, setCount] = useState(0);
// ...
}

function useWindowWidth() {
// ■ Good: top-level in a custom Hook
const [width, setWidth] = useState(window.innerWidth);
// ...
}
```

It’s **not** supported to call Hooks (functions starting with `use`) in any other cases, for example:

* ■ Do not call Hooks inside conditions or loops.
* ■ Do not call Hooks after a conditional `return` statement.
* ■ Do not call Hooks in event handlers.
* ■ Do not call Hooks in class components.
* ■ Do not call Hooks inside functions passed to `useMemo`, `useReducer`, or `useEffect`.

If you break these rules, you might see this error.

```js{3-4,11-12,20-21}
function Bad({ cond }) {
if (cond) {
// ■ Bad: inside a condition (to fix, move it outside!)
const theme = useContext(ThemeContext);
}
// ...
}

function Bad() {
```

```
for (let i = 0; i < 10; i++) {
// ■ Bad: inside a loop (to fix, move it outside!)
const theme = useContext(ThemeContext);
}
// ...
}

function Bad({ cond }) {
if (cond) {
return;
}
// ■ Bad: after a conditional return (to fix, move it before the return!)
const theme = useContext(ThemeContext);
// ...
}

function Bad() {
function handleClick() {
// ■ Bad: inside an event handler (to fix, move it outside!)
const theme = useContext(ThemeContext);
}
// ...
}

function Bad() {
const style = useMemo(() => {
// ■ Bad: inside useMemo (to fix, move it outside!)
const theme = useContext(ThemeContext);
return createStyle(theme);
});
// ...
}

class Bad extends React.Component {
render() {
// ■ Bad: inside a class component (to fix, write a function component instead of a class!)
useEffect(() => {})
// ...
```

}

}
```

You can use the [`eslint-plugin-react-hooks`
plugin](https://www.npmjs.com/package/eslint-plugin-react-hooks) to catch these mistakes.

<Note>

[Custom Hooks](/learn/reusing-logic-with-custom-hooks) *may* call other Hooks (that's their whole purpose). This works because custom Hooks are also supposed to only be called while a function component is rendering.

</Note>

## Mismatching Versions of React and React DOM {/*mismatching-versions-of-react-and-react-dom*/}

You might be using a version of `react-dom` (< 16.8.0) or `react-native` (< 0.59) that doesn't yet support Hooks. You can run `npm ls react-dom` or `npm ls react-native` in your application folder to check which version you're using. If you find more than one of them, this might also create problems (more on that below).

## Duplicate React {/*duplicate-react*/}

In order for Hooks to work, the `react` import from your application code needs to resolve to the same module as the `react` import from inside the `react-dom` package.

If these `react` imports resolve to two different exports objects, you will see this warning. This may happen if you **accidentally end up with two copies** of the `react` package.

If you use Node for package management, you can run this check in your project folder:

<TerminalBlock>

npm ls react

</TerminalBlock>

If you see more than one React, you'll need to figure out why this happens and fix your dependency tree. For example, maybe a library you're using incorrectly specifies `react` as a dependency (rather than a peer dependency). Until that library is fixed, [Yarn
resolutions](https://yarnpkg.com/lang/en/docs/selective-version-resolutions/) is one possible workaround.

You can also try to debug this problem by adding some logs and restarting your development server:

```js
// Add this in node_modules/react-dom/index.js
window.React1 = require('react');

// Add this in your component file

```
require('react-dom');

window.React2 = require('react');

console.log(window.React1 === window.React2);
```

If it prints `false` then you might have two Reacts and need to figure out why that happened. [This issue](https://github.com/facebook/react/issues/13991) includes some common reasons encountered by the community.

This problem can also come up when you use `npm link` or an equivalent. In that case, your bundler might "see" two Reacts — one in application folder and one in your library folder. Assuming `myapp` and `mylib` are sibling folders, one possible fix is to run `npm link ../myapp/node_modules/react` from `mylib`. This should make the library use the application's React copy.

<Note>

In general, React supports using multiple independent copies on one page (for example, if an app and a third-party widget both use it). It only breaks if `require('react')` resolves differently between the component and the `react-dom` copy it was rendered with.

</Note>

Other Causes {/*other-causes*/}

If none of this worked, please comment in [this issue](https://github.com/facebook/react/issues/13991) and we'll try to help. Try to create a small reproducing example — you might discover the problem as you're doing it.

---

title: Unknown Prop Warning

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 `{...props}` or `cloneElement(element, props)`? When copying props to a child component, you should ensure that you are not accidentally forwarding props that were intended only for the parent component. See common fixes for this problem below.

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](https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Using_data_attributes).

3. React does not yet recognize the attribute you specified. This will likely be fixed in a future version of React. React will allow you to pass it without a warning if you write the attribute name lowercase.

4. You are using a React component without an upper case, for example `<myButton />`. 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. For your own React components, use PascalCase. For example, write `<MyButton />` instead of `<myButton />`.

If you get this warning because you pass props like `{...props}`, your parent component needs to "consume" any prop that is intended for the parent component and not intended for the child component. Example:

Bad: Unexpected `layout` prop is forwarded to the `div` tag.

```js
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 syntax can be used to pull variables off props, and put the remaining props into a variable.

```js
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.

```js
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()} />
}
}
```

---

title: Invalid ARIA Prop Warning

This 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](https://www.w3.org/TR/wai-aria-1.1/#states_and_properties).

1. If you feel that you are using a valid prop, check the spelling carefully. `aria-labelledby` and `aria-activedescendant` are often misspelled.

2. If you wrote `aria-role`, you may have meant `role`.

3. Otherwise, if you're on the latest version of React DOM and verified that you're using a valid property name listed in the ARIA specification, please [report a bug](https://github.com/facebook/react/issues/new/choose).

---

title: Versioning Policy

<Intro>

All stable builds of React go through a high level of testing and follow semantic versioning (semver). React also offers unstable release channels to encourage early feedback on experimental features. This page describes what you can expect from React releases.

</Intro>

Stable releases {/*stable-releases*/}

Stable React releases (also known as "Latest" release channel) follow [semantic versioning (semver)](https://semver.org/) principles.

That means that with a version number **x.y.z**:

* When releasing **critical bug fixes**, we make a **patch release** by changing the **z** number (ex: 15.6.2 to 15.6.3).

* When releasing **new features** or **non-critical fixes**, we make a **minor release** by changing the **y** number (ex: 15.6.2 to 15.7.0).

* When releasing **breaking changes**, we make a **major release** by changing the **x** number (ex: 15.6.2 to 16.0.0).

Major releases can also contain new features, and any release can include bug fixes.

Minor releases are the most common type of release.

Breaking Changes {/*breaking-changes*/}

Breaking changes are inconvenient for everyone, so we try to minimize the number of major releases – for example, React 15 was released in April 2016 and React 16 was released in September 2017, and React 17 was released in October 2020.

Instead, we release new features in minor versions. That means that minor releases are often more interesting and compelling than majors, despite their unassuming name.

Commitment to stability {/*commitment-to-stability*/}

As we change React over time, we try to minimize the effort required to take advantage of new features. When possible, we'll keep an older API working, even if that means putting it in a separate package. For example, [mixins have been discouraged for years](https://legacy.reactjs.org/blog/2016/07/13/mixins-considered-harmful.html) but they're supported to this day [via create-react-class](https://legacy.reactjs.org/docs/react-without-es6.html#mixins) and many codebases continue to use them in stable, legacy code.

Over a million developers use React, collectively maintaining millions of components. The Facebook codebase alone has over 50,000 React components. That means we need to make it as easy as possible to upgrade to new versions of React; if we make large changes without a migration path, people will be stuck on old versions. We test these upgrade paths on Facebook itself – if our team of less than 10 people can update 50,000+ components alone, we hope the upgrade will be manageable for anyone using React. In many cases, we write [automated scripts](https://github.com/reactjs/react-codemod) to upgrade component syntax, which we then include in the open-source release for everyone to use.

Gradual upgrades via warnings {/*gradual-upgrades-via-warnings*/}

Development builds of React include many helpful warnings. Whenever possible, we add warnings in preparation for future breaking changes. That way, if your app has no warnings on the latest release, it will be compatible with the next major release. This allows you to upgrade your apps one component at a time.

Development warnings won't affect the runtime behavior of your app. That way, you can feel confident that your app will behave the same way between the development and production builds -- the only differences are that the production build won't log the warnings and that it is more efficient. (If you ever notice otherwise, please file an issue.)

What counts as a breaking change? {/*what-counts-as-a-breaking-change*/}

In general, we *don't* bump the major version number for changes to:

* **Development warnings.** Since these don't affect production behavior, we may add new warnings or modify existing warnings in between major versions. In fact, this is what allows us to reliably warn about upcoming breaking changes.

* **APIs starting with `unstable_`.** These are provided as experimental features whose APIs we are not yet confident in. By releasing these with an `unstable_` prefix, we can iterate faster and get to a stable API sooner.

* **Alpha and canary versions of React.** We provide alpha versions of React as a way to test new features early, but we need the flexibility to make changes based on what we learn in the alpha period. If you use these versions, note that APIs may change before the stable release.

* **Undocumented APIs and internal data structures.** If you access internal property names like `__SECRET_INTERNALS_DO_NOT_USE_OR_YOU_WILL_BE_FIRED` or `__reactInternalInstance$uk43rzhitjg`, there is no warranty. You are on your own.

This policy is designed to be pragmatic: certainly, we don't want to cause headaches for you. If we bumped the major version for all of these changes, we would end up releasing more major versions and ultimately causing more versioning pain for the community. It would also mean that we can't make progress in improving React as fast as we'd like.

That said, if we expect that a change on this list will cause broad problems in the community, we will still do our best to provide a gradual migration path.

If a minor release includes no new features, why isn't it a patch? {/*if-a-minor-release-includes-no-new-features-why-isnt-it-a-patch*/}

It's possible that a minor release will not include new features. [This is allowed by semver](https://semver.org/#spec-item-7), which states **"[a minor version] MAY be incremented if substantial new functionality or improvements are introduced within the private code. It MAY include patch level changes."**

However, it does raise the question of why these releases aren't versioned as patches instead.

The answer is that any change to React (or other software) carries some risk of breaking in unexpected ways. Imagine a scenario where a patch release that fixes one bug accidentally introduces a different bug. This would not only be disruptive to developers, but also harm their confidence in future patch releases. It's especially regrettable if the original fix is for a bug that is rarely encountered in practice.

We have a pretty good track record for keeping React releases free of bugs, but patch releases have an even higher bar for reliability because most developers assume they can be adopted without adverse consequences.

For these reasons, we reserve patch releases only for the most critical bugs and security vulnerabilities.

If a release includes non-essential changes — such as internal refactors, changes to implementation details, performance improvements, or minor bugfixes — we will bump the minor version even when there are no new features.

All release channels {/*all-release-channels*/}

React relies on a thriving open source community to file bug reports, open pull requests, and [submit RFCs](https://github.com/reactjs/rfcs). To encourage feedback we sometimes share special builds of React that include unreleased features.

<Note>

This section will be most relevant to developers who work on frameworks, libraries, or developer tooling. Developers who use React primarily to build user-facing applications should not need to worry about our prerelease channels.

</Note>

Each of React's release channels is designed for a distinct use case:

- [**Latest**](#latest-channel) is for stable, semver React releases. It's what you get when you install React from npm. This is the channel you're already using today. **User-facing applications that consume React directly use this channel.**
- [**Canary**](#canary-channel) tracks the main branch of the React source code repository. Think of these as release candidates for the next semver release. **[Frameworks or other curated setups may choose to use this channel with a pinned version of React.](/blog/2023/05/03/react-canaries) You can also Canaries for integration testing between React and third party projects.**
- [**Experimental**](#experimental-channel) includes experimental APIs and features that aren't available in the stable releases. These also track the main branch, but with additional feature flags turned on. Use this to try out upcoming features before they are released.

All releases are published to npm, but only Latest uses semantic versioning. Prereleases (those in the Canary and Experimental channels) have versions generated from a hash of their contents and the commit date, e.g. `18.3.0-canary-388686f29-20230503` for Canary and `0.0.0-experimental-388686f29-20230503` for Experimental.

Both Latest and Canary channels are officially supported for user-facing applications, but with different expectations:

* Latest releases follow the traditional semver model.
* Canary releases [must be pinned](/blog/2023/05/03/react-canaries) and may include breaking changes. They exist for curated setups (like frameworks) that want to gradually release new React features and bugfixes on their own release schedule.

The Experimental releases are provided for testing purposes only, and we provide no guarantees that behavior won't change between releases. They do not follow the semver protocol that we use for releases from Latest.

By publishing prereleases to the same registry that we use for stable releases, we are able to take advantage of the many tools that support the npm workflow, like [unpkg](https://unpkg.com) and [CodeSandbox](https://codesandbox.io).

Latest channel {/*latest-channel*/}

Latest is the channel used for stable React releases. It corresponds to the `latest` tag on npm. It is the recommended channel for all React apps that are shipped to real users.

If you're not sure which channel you should use, it's Latest. If you're using React directly, this is what you're already using. You can expect updates to Latest to be extremely stable. Versions follow the semantic versioning scheme, as [described earlier.](#stable-releases)

Canary channel {/*canary-channel*/}

The Canary channel is a prerelease channel that tracks the main branch of the React repository. We use prereleases in the Canary channel as release candidates for the Latest channel. You can think of Canary as a superset of Latest that is updated more frequently.

The degree of change between the most recent Canary release and the most recent Latest release is approximately the same as you would find between two minor semver releases. However, **the Canary channel does not conform to semantic versioning.** You should expect occasional breaking changes between successive releases in the Canary channel.

Do not use prereleases in user-facing applications directly unless you're following the [Canary workflow](/blog/2023/05/03/react-canaries).

Releases in Canary are published with the `canary` tag on npm. Versions are generated from a hash of the build's contents and the commit date, e.g. `18.3.0-canary-388686f29-20230503`.

Using the canary channel for integration testing
{/*using-the-canary-channel-for-integration-testing*/}

The Canary channel also supports integration testing between React and other projects.

All changes to React go through extensive internal testing before they are released to the public. However, there are a myriad of environments and configurations used throughout the React ecosystem, and it's not possible for us to test against every single one.

If you're the author of a third party React framework, library, developer tool, or similar infrastructure-type project, you can help us keep React stable for your users and the entire React community by periodically running your test suite against the most recent changes. If you're interested, follow these steps:

- Set up a cron job using your preferred continuous integration platform. Cron jobs are supported by both [CircleCI](https://circleci.com/docs/2.0/triggers/#scheduled-builds) and [Travis CI](https://docs.travis-ci.com/user/cron-jobs/).

- In the cron job, update your React packages to the most recent React release in the Canary channel, using `canary` tag on npm. Using the npm cli:

```console
npm update react@canary react-dom@canary
```

Or yarn:

```console
yarn upgrade react@canary react-dom@canary
```

- Run your test suite against the updated packages.

- If everything passes, great! You can expect that your project will work with the next minor React release.

- If something breaks unexpectedly, please let us know by [filing an issue](https://github.com/facebook/react/issues).

A project that uses this workflow is Next.js. You can refer to their [CircleCI configuration](https://github.com/zeit/next.js/blob/c0a1c0f93966fe33edd93fb53e5fafb0dcd80a9e/.circleci/config.yml) as an example.

Experimental channel {/*experimental-channel*/}

Like Canary, the Experimental channel is a prerelease channel that tracks the main branch of the React repository. Unlike Canary, Experimental releases include additional features and APIs that are not ready for wider release.

Usually, an update to Canary is accompanied by a corresponding update to Experimental. They are based on the same source revision, but are built using a different set of feature flags.

Experimental releases may be significantly different than releases to Canary and Latest. **Do not use Experimental releases in user-facing applications.** You should expect frequent breaking changes between releases in the Experimental channel.

Releases in Experimental are published with the `experimental` tag on npm. Versions are generated from a hash of the build's contents and the commit date, e.g. `0.0.0-experimental-68053d940-20210623`.

What goes into an experimental release? {/*what-goes-into-an-experimental-release*/}

Experimental features are ones that are not ready to be released to the wider public, and may change drastically before they are finalized. Some experiments may never be finalized -- the reason we have experiments is to test the viability of proposed changes.

For example, if the Experimental channel had existed when we announced Hooks, we would have released Hooks to the Experimental channel weeks before they were available in Latest.

You may find it valuable to run integration tests against Experimental. This is up to you. However, be advised that Experimental is even less stable than Canary. **We do not guarantee any stability between Experimental releases.**

How can I learn more about experimental features? {/*how-can-i-learn-more-about-experimental-features*/}

Experimental features may or may not be documented. Usually, experiments aren't documented until they are close to shipping in Canary or Latest.

If a feature is not documented, they may be accompanied by an [RFC](https://github.com/reactjs/rfcs).

We will post to the [React blog](/blog) when we're ready to announce new experiments, but that doesn't mean we will publicize every experiment.

You can always refer to our public GitHub repository's [history](https://github.com/facebook/react/commits/main) for a comprehensive list of changes.

---

title: React Community

<Intro>

React has a community of millions of developers. On this page we've listed some React-related communities that you can be a part of; see the other pages in this section for additional online and in-person learning materials.

</Intro>

Code of Conduct {/*code-of-conduct*/}

Before participating in React's communities, [please read our Code of Conduct.](https://github.com/facebook/react/blob/main/CODE_OF_CONDUCT.md) We have adopted the [Contributor Covenant](https://www.contributor-covenant.org/) and we expect that all community members adhere to the guidelines within.

Stack Overflow {/*stack-overflow*/}

Stack Overflow is a popular forum to ask code-level questions or if you're stuck with a specific error. Read through the [existing questions](https://stackoverflow.com/questions/tagged/reactjs) tagged with **reactjs** or [ask your own](https://stackoverflow.com/questions/ask?tags=reactjs)!

Popular Discussion Forums {/*popular-discussion-forums*/}

There are many online forums which are a great place for discussion about best practices and application architecture as well as the future of React. If you have an answerable code-level question, Stack Overflow is usually a better fit.

Each community consists of many thousands of React users.

* [DEV's React community](https://dev.to/t/react)

* [Hashnode's React community](https://hashnode.com/n/reactjs)

* [Reactiflux online chat](https://discord.gg/reactiflux)

* [Reddit's React community](https://www.reddit.com/r/reactjs/)

News {/*news*/}

For the latest news about React, [follow **@reactjs** on Twitter](https://twitter.com/reactjs) and the [official React blog](/blog/) on this website.

---

title: Acknowledgements

<Intro>

React was originally created by [Jordan Walke.](https://github.com/jordwalke) Today, React has a [dedicated full-time team working on it](/community/team), as well as over a thousand [open source contributors.](https://github.com/facebook/react/blob/main/AUTHORS)

</Intro>

Past contributors {/*past-contributors*/}

We'd like to recognize a few people who have made significant contributions to React and its documentation in the past and have helped maintain them over the years:

* [Almero Steyn](https://github.com/AlmeroSteyn)
* [Andreas Svensson](https://github.com/syranide)
* [Alex Krolick](https://github.com/alexkrolick)
* [Alexey Pyltsyn](https://github.com/lex111)
* [Brandon Dail](https://github.com/aweary)
* [Brian Vaughn](https://github.com/bvaughn)
* [Caleb Meredith](https://github.com/calebmer)
* [Chang Yan](https://github.com/cyan33)
* [Cheng Lou](https://github.com/chenglou)
* [Christoph Nakazawa](https://github.com/cpojer)
* [Christopher Chedeau](https://github.com/vjeux)
* [Clement Hoang](https://github.com/clemmy)
* [Dominic Gannaway](https://github.com/trueadm)
* [Flarnie Marchan](https://github.com/flarnie)
* [Jason Quense](https://github.com/jquense)
* [Jesse Beach](https://github.com/jessebeach)
* [Jessica Franco](https://github.com/Jessidhia)
* [Jim Sproch](https://github.com/jimfb)
* [Josh Duck](https://github.com/joshduck)
* [Joe Critchley](https://github.com/joecritch)
* [Jeff Morrison](https://github.com/jeffmo)
* [Keyan Zhang](https://github.com/keyz)
* [Marco Salazar](https://github.com/salazarm)
* [Nat Alison](https://github.com/tesseralis)
* [Nathan Hunzaker](https://github.com/nhunzaker)
* [Nicolas Gallagher](https://github.com/necolas)
* [Paul O'Shannessy](https://github.com/zpao)

* [Pete Hunt](https://github.com/petehunt)

* [Philipp Spiess](https://github.com/philipp-spiess)

* [Rachel Nabors](https://github.com/rachelnabors)

* [Robert Zhang](https://github.com/robertzhidealx)

* [Sander Spies](https://github.com/sanderspies)

* [Sasha Aickin](https://github.com/aickin)

* [Seth Webster](https://github.com/sethwebster)

* [Sophia Shoemaker](https://github.com/mrscobbler)

* [Sunil Pai](https://github.com/threepointone)

* [Tim Yung](https://github.com/yungsters)

* [Xuan Huang](https://github.com/huxpro)

This list is not exhaustive.

We'd like to give special thanks to [Tom Occhino](https://github.com/tomocchino) and [Adam Wolff](https://github.com/wolffiex) for their guidance and support over the years. We are also thankful to all the volunteers who [translated React into other languages.](https://translations.reactjs.org/)

Additional Thanks {/*additional-thanks*/}

Additionally, we're grateful to:

* [Jeff Barczewski](https://github.com/jeffbski) for allowing us to use the `react` package name on npm

* [Christopher Aue](https://christopheraue.net/) for letting us use the reactjs.com domain name and the [@reactjs](https://twitter.com/reactjs) username on Twitter

* [ProjectMoon](https://github.com/ProjectMoon) for letting us use the [flux](https://www.npmjs.com/package/flux) package name on npm

* Shane Anderson for allowing us to use the [react](https://github.com/react) org on GitHub

---

title: React Videos

<Intro>

Videos dedicated to the discussion of React and the React ecosystem.

</Intro>

React Conf 2021 {/*react-conf-2021*/}

React 18 Keynote {/*react-18-keynote*/}

In the keynote, we shared our vision for the future of React starting with React 18.

Watch the full keynote from [Andrew Clark](https://twitter.com/acdlite), [Juan Tejada](https://twitter.com/_jstejada), [Lauren Tan](https://twitter.com/potetotes), and [Rick Hanlon](https://twitter.com/rickhanlonii) here:

<YouTubeIframe src="https://www.youtube.com/embed/FZ0cG47msEk" title="YouTube video player" />

React 18 for Application Developers {/*react-18-for-application-developers*/}

For a demo of upgrading to React 18, see [Shruti Kapoor](https://twitter.com/shrutikapoor08)'s talk here:

<YouTubeIframe src="https://www.youtube.com/embed/ytudH8je5ko" title="YouTube video player" />

Streaming Server Rendering with Suspense {/*streaming-server-rendering-with-suspense*/}

React 18 also includes improvements to server-side rendering performance using Suspense.

Streaming server rendering lets you generate HTML from React components on the server, and stream that HTML to your users. In React 18, you can use `Suspense` to break down your app into smaller independent units which can be streamed independently of each other without blocking the rest of the app. This means users will see your content sooner and be able to start interacting with it much faster.

For a deep dive, see [Shaundai Person](https://twitter.com/shaundai)'s talk here:

<YouTubeIframe src="https://www.youtube.com/embed/pj5N-Khihgc" title="YouTube video player" />

The first React working group {/*the-first-react-working-group*/}

For React 18, we created our first Working Group to collaborate with a panel of experts, developers, library maintainers, and educators. Together we worked to create our gradual adoption strategy and refine new APIs such as `useId`, `useSyncExternalStore`, and `useInsertionEffect`.

For an overview of this work, see [Aakansha' Doshi](https://twitter.com/aakansha1216)'s talk:

<YouTubeIframe src="https://www.youtube.com/embed/qn7gRClrC9U" title="YouTube video player" />

React Developer Tooling {/*react-developer-tooling*/}

To support the new features in this release, we also announced the newly formed React DevTools team and a new Timeline Profiler to help developers debug their React apps.

For more information and a demo of new DevTools features, see [Brian Vaughn](https://twitter.com/brian_d_vaughn)'s talk:

<YouTubeIframe src="https://www.youtube.com/embed/oxDfrke8rZg" title="YouTube video player" />

React without memo {/*react-without-memo*/}

Looking further into the future, [Xuan Huang (■■)](https://twitter.com/Huxpro) shared an update from our React Labs research into an auto-memoizing compiler. Check out this talk for more information and a demo of the compiler prototype:

<YouTubeIframe src="https://www.youtube.com/embed/lGEMwh32soc" title="YouTube video player" />

React docs keynote {/*react-docs-keynote*/}

[Rachel Nabors](https://twitter.com/rachelnabors) kicked off a section of talks about learning and designing with React with a keynote about our investment in React's new docs ([now shipped as react.dev](/blog/2023/03/16/introducing-react-dev)):

<YouTubeIframe src="https://www.youtube.com/embed/mneDaMYOKP8" title="YouTube video player" />

And more... {/*and-more*/}

We also heard talks on learning and designing with React:

* Debbie O'Brien: [Things I learnt from the new React docs](https://youtu.be/-7odLW_hG7s).
* Sarah Rainsberger: [Learning in the Browser](https://youtu.be/5X-WEQflCL0).
* Linton Ye: [The ROI of Designing with React](https://youtu.be/7cPWmID5XAk).
* Delba de Oliveira: [Interactive playgrounds with React](https://youtu.be/zL8cz2W0z34).

Talks from the Relay, React Native, and PyTorch teams:

* Robert Balicki: [Re-introducing Relay](https://youtu.be/lhVGdErZuN4).
* Eric Rozell and Steven Moyes: [React Native Desktop](https://youtu.be/9L4FFrvwJwY).
* Roman Rädle: [On-device Machine Learning for React Native](https://youtu.be/NLj73vrc2I8)

And talks from the community on accessibility, tooling, and Server Components:

* Daishi Kato: [React 18 for External Store Libraries](https://youtu.be/oPfSC5bQPR8).
* Diego Haz: [Building Accessible Components in React 18](https://youtu.be/dcm8fjBfro8).
* Tafu Nakazaki: [Accessible Japanese Form Components with React](https://youtu.be/S4a0QlsH0pU).
* Lyle Troxell: [UI tools for artists](https://youtu.be/b3l4WxipFsE).
* Helen Lin: [Hydrogen + React 18](https://youtu.be/HS6vIYkSNks).

Older videos {/*older-videos*/}

React Conf 2019 {/*react-conf-2019*/}

A playlist of videos from React Conf 2019.

<YouTubeIframe title="React Conf 2019" src="https://www.youtube-nocookie.com/embed/playlist?list=PLPxbbTqCLbGHPxZpw4xj_Wwg8-fdNxJRh" />

React Conf 2018 {/*react-conf-2018*/}

A playlist of videos from React Conf 2018.

<YouTubeIframe title="React Conf 2018" src="https://www.youtube-nocookie.com/embed/playlist?list=PLPxbbTqCLbGE5AihOSExAa4wUM-P42EIJ" />

React.js Conf 2017 {/*reactjs-conf-2017*/}

A playlist of videos from React.js Conf 2017.

<YouTubeIframe title="React.js Conf 2017" src="https://www.youtube-nocookie.com/embed/playlist?list=PLb0IAmt7-GS3fZ46IGFirdqKTIxlws7e0" />

React.js Conf 2016 {/*reactjs-conf-2016*/}

A playlist of videos from React.js Conf 2016.

<YouTubeIframe title="React.js Conf 2016" src="https://www.youtube-nocookie.com/embed/playlist?list=PLb0IAmt7-GS0M8Q95RIc2lOM6nc77q1IY" />

React.js Conf 2015 {/*reactjs-conf-2015*/}

A playlist of videos from React.js Conf 2015.

<YouTubeIframe title="React.js Conf 2015" src="https://www.youtube-nocookie.com/embed/playlist?list=PLb0IAmt7-GS1cbw4qonlQztYV1TAW0sCr" />

Rethinking Best Practices {/*rethinking-best-practices*/}

Pete Hunt's talk at JSConf EU 2013 covers three topics: throwing out the notion of templates and building views with JavaScript, “re-rendering” your entire application when your data changes, and a lightweight implementation of the DOM and events - (2013 - 0h30m).

<YouTubeIframe title="Pete Hunt: React: Rethinking Best Practices - JSConf EU 2013" src="https://www.youtube-nocookie.com/embed/x7cQ3mrcKaY" />

Introduction to React {/*introduction-to-react*/}

Tom Occhino and Jordan Walke introduce React at Facebook Seattle - (2013 - 1h20m).

<YouTubeIframe title="Tom Occhino and Jordan Walke introduce React at Facebook Seattle" src="https://www.youtube-nocookie.com/embed/XxVg_s8xAms" />

---

title: React Conferences

<Intro>

Do you know of a local React.js conference? Add it here! (Please keep the list chronological)

</Intro>

Upcoming Conferences {/*upcoming-conferences*/}

React Rally 2023 ■ {/*react-rally-2023*/}

August 17 & 18, 2023. Salt Lake City, UT, USA

[Website](https://www.reactrally.com/) - [Twitter](https://twitter.com/ReactRally) - [Instagram](https://www.instagram.com/reactrally/)

React India 2023 {/*react-india-2023*/}

Oct 5 - 7, 2023. In-person in Goa, India (hybrid event) + Oct 3 2023 - remote day

[Website](https://www.reactindia.io) - [Twitter](https://twitter.com/react_india) - [Facebook](https://www.facebook.com/ReactJSIndia) - [Youtube](https://www.youtube.com/channel/UCaFbHCBkPvVv1bWs_jwYt3w)

React Advanced 2023 {/*react-advanced-2023*/}

October 20 & 23, 2023. In-person in London, UK + remote first interactivity (hybrid event)

[Website](https://www.reactadvanced.com/) - [Twitter](https://twitter.com/ReactAdvanced) - [Facebook](https://www.facebook.com/ReactAdvanced) - [Videos](https://portal.gitnation.org/events/react-advanced-conference-2023)

React Summit US 2023 {/*react-summit-us-2023*/}

November 13 & 15, 2023. In-person in New York, US + remote first interactivity (hybrid event)

[Website](https://reactsummit.us) - [Twitter](https://twitter.com/reactsummit) - [Facebook](https://www.facebook.com/reactamsterdam) - [Videos](https://portal.gitnation.org/events/react-summit-us-2023)

React Day Berlin 2023 {/*react-day-berlin-2023*/}

December 8 & 12, 2023. In-person in Berlin, Germany + remote first interactivity (hybrid event)

[Website](https://reactday.berlin) - [Twitter](https://twitter.com/reactdayberlin) - [Facebook](https://www.facebook.com/reactdayberlin/) - [Videos](https://portal.gitnation.org/events/react-day-berlin-2023)

Past Conferences {/*past-conferences*/}

React Nexus 2023 {/*react-nexus-2023*/}

July 07 & 08, 2023. Bangalore, India (In-person event)

[Website](https://reactnexus.com/) - [Twitter](https://twitter.com/ReactNexus) - [Linkedin](https://www.linkedin.com/company/react-nexus) - [YouTube](https://www.youtube.com/reactify_in)

ReactNext 2023 {/*reactnext-2023*/}

June 27th, 2023. Tel Aviv, Israel

[Website](https://www.react-next.com/) - [Facebook](https://www.facebook.com/ReactNextConf) - [Youtube](https://www.youtube.com/@ReactNext)

React Norway 2023 {/*react-norway-2023*/}

June 16th, 2023. Larvik, Norway

[Website](https://reactnorway.com/) - [Twitter](https://twitter.com/ReactNorway/) - [Facebook](https://www.facebook.com/reactdaynorway/)

React Summit 2023 {/*react-summit-2023*/}

June 2 & 6, 2023. In-person in Amsterdam, Netherlands + remote first interactivity (hybrid event)

[Website](https://reactsummit.com) - [Twitter](https://twitter.com/reactsummit) - [Facebook](https://www.facebook.com/reactamsterdam) - [Videos](https://portal.gitnation.org/events/react-summit-2023)

Render(ATL) 2023 ■ {/*renderatl-2023-*/}

May 31 - June 2, 2023. Atlanta, GA, USA

[Website](https://renderatl.com) - [Discord](https://www.renderatl.com/discord) - [Twitter](https://twitter.com/renderATL) - [Instagram](https://www.instagram.com/renderatl/) - [Facebook](https://www.facebook.com/renderatl/) - [LinkedIn](https://www.linkedin.com/company/renderatl) - [Podcast](https://www.renderatl.com/culture-and-code#/)

Chain React 2023 {/*chain-react-2023*/}

May 17 - 19, 2023. Portland, OR, USA

[Website](https://chainreactconf.com/) - [Twitter](https://twitter.com/ChainReactConf) - [Facebook](https://www.facebook.com/ChainReactConf/) - [Youtube](https://www.youtube.com/channel/UCwpSzVt7QpLDbCnPXqR97-g/playlists)

App.js Conf 2023 {/*appjs-conf-2023*/}

May 10 - 12, 2023. In-person in Kraków, Poland + remote

[Website](https://appjs.co) - [Twitter](https://twitter.com/appjsconf)

RemixConf 2023 {/*remixconf-2023*/}

May, 2023. Salt Lake City, UT

[Website](https://remix.run/conf/2023) - [Twitter](https://twitter.com/remix_run)

Reactathon 2023 {/*reactathon-2023*/}

May 2 - 3, 2023. San Francisco, CA, USA

[Website](https://reactathon.com) - [Twitter](https://twitter.com/reactathon) - [YouTube](https://www.youtube.com/realworldreact)

React Miami 2023 {/*react-miami-2023*/}

April 20 - 21, 2023. Miami, FL, USA

[Website](https://www.reactmiami.com/) - [Twitter](https://twitter.com/ReactMiamiConf)

React Day Berlin 2022 {/*react-day-berlin-2022*/}

December 2, 2022. In-person in Berlin, Germany + remote (hybrid event)

[Website](https://reactday.berlin) - [Twitter](https://twitter.com/reactdayberlin) - [Facebook](https://www.facebook.com/reactdayberlin/) - [Videos](https://www.youtube.com/c/ReactConferences)

React Global Online Summit 22.2 by Geekle {/*react-global-online-summit-222-by-geekle*/}

November 8 - 9, 2022 - Online Summit

[Website](https://events.geekle.us/react3/) - [LinkedIn](https://www.linkedin.com/posts/geekle-us_event-react-reactjs-activity-6964904611207864320-gpDx?utm_source=share&utm_medium=member_desktop)

Remix Conf Europe 2022 {/*remix-conf-europe-2022*/}

November 18, 2022, 7am PST / 10am EST / 4pm CET - remote event

[Website](https://remixconf.eu/) - [Twitter](https://twitter.com/remixconfeu) - [Videos](https://portal.gitnation.org/events/remix-conf-europe-2022)

React Advanced 2022 {/*react-advanced-2022*/}

October 21 & 25, 2022. In-person in London, UK + remote (hybrid event)

[Website](https://www.reactadvanced.com/) - [Twitter](https://twitter.com/ReactAdvanced) - [Facebook](https://www.facebook.com/ReactAdvanced) - [Videos](https://portal.gitnation.org/events/react-advanced-conference-2022)

ReactJS Day 2022 {/*reactjs-day-2022*/}

October 21, 2022 in Verona, Italy

[Website](https://2022.reactjsday.it/) - [Twitter](https://twitter.com/reactjsday) - [LinkedIn](https://www.linkedin.com/company/grusp/) - [Facebook](https://www.facebook.com/reactjsday/) - [Videos](https://www.youtube.com/c/grusp)

React Brussels 2022 {/*react-brussels-2022*/}

October 14, 2022. In-person in Brussels, Belgium + remote (hybrid event)

[Website](https://www.react.brussels/) - [Twitter](https://twitter.com/BrusselsReact) - [LinkedIn](https://www.linkedin.com/events/6938421827153088512/) - [Facebook](https://www.facebook.com/events/1289080838167252/) - [Videos](https://www.youtube.com/channel/UCvES7IMpnx-t934qGxD4w4g)

React Alicante 2022 {/*react-alicante-2022*/}

September 29 - October 1, 2022. In-person in Alicante, Spain + remote (hybrid event)

[Website](https://reactalicante.es/) - [Twitter](https://twitter.com/reactalicante) - [Facebook](https://www.facebook.com/ReactAlicante) - [Videos](https://www.youtube.com/channel/UCaSdUaITU1Cz6PvC97A7e0w)

React India 2022 {/*react-india-2022*/}

September 22 - 24, 2022. In-person in Goa, India + remote (hybrid event)

[Website](https://www.reactindia.io) - [Twitter](https://twitter.com/react_india) - [Facebook](https://www.facebook.com/ReactJSIndia) - [Videos](https://www.youtube.com/channel/UCaFbHCBkPvVv1bWs_jwYt3w)

React Finland 2022 {/*react-finland-2022*/}

September 12 - 16, 2022. In-person in Helsinki, Finland

[Website](https://react-finland.fi/) - [Twitter](https://twitter.com/ReactFinland) - [Schedule](https://react-finland.fi/schedule/) - [Speakers](https://react-finland.fi/speakers/)

React Native EU 2022: Powered by callstack {/*react-native-eu-2022-powered-by-callstack*/}

September 1-2, 2022 - Remote event

[Website](https://www.react-native.eu/?utm_campaign=React_Native_EU&utm_source=referral&utm_content=reactjs_community_conferences) -

[Twitter](https://twitter.com/react_native_eu) -

[Linkedin](https://www.linkedin.com/showcase/react-native-eu) -

[Facebook](https://www.facebook.com/reactnativeeu/) -

[Instagram](https://www.instagram.com/reactnative_eu/)

ReactNext 2022 {/*reactnext-2022*/}

June 28, 2022. Tel-Aviv, Israel

[Website](https://react-next.com) - [Twitter](https://twitter.com/ReactNext) - [Videos](https://www.youtube.com/c/ReactNext)

React Norway 2022 {/*react-norway-2022*/}

June 24, 2022. In-person at Farris Bad Hotel in Larvik, Norway and online (hybrid event).

[Website](https://reactnorway.com/) - [Twitter](https://twitter.com/ReactNorway)

React Summit 2022 {/*react-summit-2022*/}

June 17 & 21, 2022. In-person in Amsterdam, Netherlands + remote first interactivity (hybrid event)

[Website](https://reactsummit.com) - [Twitter](https://twitter.com/reactsummit) - [Facebook](https://www.facebook.com/reactamsterdam) - [Videos](https://portal.gitnation.org/events/react-summit-2022)

App.js Conf 2022 {/*appjs-conf-2022*/}

June 8 - 10, 2022. In-person in Kraków, Poland + remote

[Website](https://appjs.co) - [Twitter](https://twitter.com/appjsconf)

React Day Bangalore 2022 {/*react-day-bangalore-2022*/}

June 8 - 9, 2022. Remote

[Website](https://reactday.in/) - [Twitter](https://twitter.com/ReactDayIn) - [Linkedin](https://www.linkedin.com/company/react-day/) - [YouTube](https://www.youtube.com/reactify_in)

render(ATL) 2022 ■ {/*renderatl-2022-*/}

June 1 - 4, 2022. Atlanta, GA, USA

[Website](https://renderatl.com) - [Discord](https://www.renderatl.com/discord) - [Twitter](https://twitter.com/renderATL) - [Instagram](https://www.instagram.com/renderatl/) - [Facebook](https://www.facebook.com/renderatl/) - [LinkedIn](https://www.linkedin.com/company/renderatl) - [Podcast](https://www.renderatl.com/culture-and-code#/)

RemixConf 2022 {/*remixconf-2022*/}

May 24 - 25, 2022. Salt Lake City, UT

[Website](https://remix.run/conf/2022) - [Twitter](https://twitter.com/remix_run) - [YouTube](https://www.youtube.com/playlist?list=PLXoynULbYuEC36XutMMWEuTu9uuh171wx)

Reactathon 2022 {/*reactathon-2022*/}

May 3 - 5, 2022. Berkeley, CA

[Website](https://reactathon.com) - [Twitter](https://twitter.com/reactathon) -[YouTube](https://www.youtube.com/watch?v=-YG5cljNXIA)

React Global Online Summit 2022 by Geekle {/*react-global-online-summit-2022-by-geekle*/}

April 20 - 21, 2022 - Online Summit

[Website](https://events.geekle.us/react2/) - [LinkedIn](https://www.linkedin.com/events/reactglobalonlinesummit-22688741766454161408 1/)

React Miami 2022 ■ {/*react-miami-2022-*/}

April 18 - 19, 2022. Miami, Florida

[Website](https://www.reactmiami.com/)

React Live 2022 {/*react-live-2022*/}

April 1, 2022. Amsterdam, The Netherlands

[Website](https://www.reactlive.nl/) - [Twitter](https://twitter.com/reactlivenl)

AgentConf 2022 {/*agentconf-2022*/}

January 27 - 30, 2022. In-person in Dornbirn and Lech Austria

[Website](https://agent.sh/) - [Twitter](https://twitter.com/AgentConf) - [Instagram](https://www.instagram.com/teamagent/)

React Conf 2021 {/*react-conf-2021*/}

December 8, 2021 - remote event (replay event on December 9)

[Website](https://conf.reactjs.org/)

ReactEurope 2021 {/*reacteurope-2021*/}

December 9-10, 2021 - remote event

[Videos](https://www.youtube.com/c/ReacteuropeOrgConf)

ReactNext 2021 {/*reactnext-2021*/}

December 15, 2021. Tel-Aviv, Israel

[Website](https://react-next.com) - [Twitter](https://twitter.com/ReactNext) -
[Videos](https://www.youtube.com/channel/UC3BT8hh3yTTYxbLQy_wbk2w)

React India 2021 {/*react-india-2021*/}

November 12-13, 2021 - remote event

[Website](https://www.reactindia.io) - [Twitter](https://twitter.com/react_india) -
[Facebook](https://www.facebook.com/ReactJSIndia/) -
[LinkedIn](https://www.linkedin.com/showcase/14545585) -
[YouTube](https://www.youtube.com/channel/UCaFbHCBkPvVv1bWs_jwYt3w/videos)

React Global by Geekle {/*react-global-by-geekle*/}

November 3-4, 2021 - remote event

[Website](https://geekle.us/react) -
[LinkedIn](https://www.linkedin.com/events/javascriptglobalsummit6721691514176720896/) -
[YouTube](https://www.youtube.com/watch?v=0HhWIvPhbu0)

React Advanced 2021 {/*react-advanced-2021*/}

October 22-23, 2021. In-person in London, UK + remote (hybrid event)

[Website](https://reactadvanced.com) - [Twitter](https://twitter.com/reactadvanced) -
[Facebook](https://www.facebook.com/ReactAdvanced) -
[Videos](https://youtube.com/c/ReactConferences)

React Conf Brasil 2021 {/*react-conf-brasil-2021*/}

October 16, 2021 - remote event

[Website](http://reactconf.com.br) - [Twitter](https://twitter.com/reactconfbr) -
[Slack](https://react.now.sh) - [Facebook](https://facebook.com/reactconf) -
[Instagram](https://instagram.com/reactconfbr) -
[YouTube](https://www.youtube.com/channel/UCJL5eorStQfC0x1iiWhvqPA/videos)

React Brussels 2021 {/*react-brussels-2021*/}

October 15, 2021 - remote event

[Website](https://www.react.brussels/) - [Twitter](https://twitter.com/BrusselsReact) -
[LinkedIn](https://www.linkedin.com/events/6805708233819336704/)

render(ATL) 2021 {/*renderatl-2021*/}

September 13-15, 2021. Atlanta, GA, USA

[Website](https://renderatl.com) - [Twitter](https://twitter.com/renderATL) - [Instagram](https://www.instagram.com/renderatl/) - [Facebook](https://www.facebook.com/renderatl/) - [LinkedIn](https://www.linkedin.com/company/renderatl)

React Native EU 2021 {/*react-native-eu-2021*/}

September 1-2, 2021 - remote event

[Website](https://www.react-native.eu/) - [Twitter](https://twitter.com/react_native_eu) - [Facebook](https://www.facebook.com/reactnativeeu/) - [Instagram](https://www.instagram.com/reactnative_eu/)

React Finland 2021 {/*react-finland-2021*/}

August 30 - September 3, 2021 - remote event

[Website](https://react-finland.fi/) - [Twitter](https://twitter.com/ReactFinland) - [LinkedIn](https://www.linkedin.com/company/react-finland/)

React Case Study Festival 2021 {/*react-case-study-festival-2021*/}

April 27-28, 2021 - remote event

[Website](https://link.geekle.us/react/offsite) - [LinkedIn](https://www.linkedin.com/events/reactcasestudyfestival6721300943411015680/) - [Facebook](https://www.facebook.com/events/255715435820203)

React Summit - Remote Edition 2021 {/*react-summit---remote-edition-2021*/}

April 14-16, 2021, 7am PST / 10am EST / 4pm CEST - remote event

[Website](https://remote.reactsummit.com) - [Twitter](https://twitter.com/reactsummit) - [Facebook](https://www.facebook.com/reactamsterdam) - [Videos](https://portal.gitnation.org/events/react-summit-remote-edition-2021)

React fwdays'21 {/*react-fwdays21*/}

March 27, 2021 - remote event

[Website](https://fwdays.com/en/event/react-fwdays-2021) - [Twitter](https://twitter.com/fwdays) - [Facebook](https://www.facebook.com/events/1133828147054286) - [LinkedIn](https://www.linkedin.com/events/reactfwdays-21onlineconference6758046347334582273) - [Meetup](https://www.meetup.com/ru-RU/Fwdays/events/275764431/)

React Next 2020 {/*react-next-2020*/}

December 1-2, 2020 - remote event

[Website](https://react-next.com/) - [Twitter](https://twitter.com/reactnext) - [Facebook](https://www.facebook.com/ReactNext2016/)

React Conf Brasil 2020 {/*react-conf-brasil-2020*/}

November 21, 2020 - remote event

[Website](https://reactconf.com.br/) - [Twitter](https://twitter.com/reactconfbr) - [Slack](https://react.now.sh/)

React Summit 2020 {/*react-summit-2020*/}

October 15-16, 2020, 7am PST / 10am EST / 4pm CEST - remote event

[Website](https://reactsummit.com) - [Twitter](https://twitter.com/reactsummit) - [Facebook](https://www.facebook.com/reactamsterdam) - [Videos](https://youtube.com/c/ReactConferences)

React Native EU 2020 {/*react-native-eu-2020*/}

September 3-4, 2020 - remote event

[Website](https://www.react-native.eu/) - [Twitter](https://twitter.com/react_native_eu) - [Facebook](https://www.facebook.com/reactnativeeu/) - [YouTube](https://www.youtube.com/watch?v=m0GfmlGFh3E&list=PLZ3MwD-soTTHy9_88QPLF8DEJkvoB5Tl-) - [Instagram](https://www.instagram.com/reactnative_eu/)

ReactEurope 2020 {/*reacteurope-2020*/}

May 14-15, 2020 in Paris, France

[Videos](https://www.youtube.com/c/ReacteuropeOrgConf)

Byteconf React 2020 {/*byteconf-react-2020*/}

May 1, 2020. Streamed online on YouTube.

[Website](https://www.bytesized.xyz) - [Twitter](https://twitter.com/bytesizedcode) - [YouTube](https://www.youtube.com/channel/UC046lFvJZhiwSRWsoH8SFjg)

React Summit - Remote Edition 2020 {/*react-summit---remote-edition-2020*/}

3pm CEST time, April 17, 2020 - remote event

[Website](https://remote.reactsummit.com) - [Twitter](https://twitter.com/reactsummit) - [Facebook](https://www.facebook.com/reactamsterdam) - [Videos](https://youtube.com/c/ReactConferences)

Reactathon 2020 {/*reactathon-2020*/}

March 30 - 31, 2020 in San Francisco, CA

[Website](https://www.reactathon.com) - [Twitter](https://twitter.com/reactathon) - [Facebook](https://www.facebook.com/events/575942819854160/)

ReactConf AU 2020 {/*reactconf-au-2020*/}

February 27 & 28, 2020 in Sydney, Australia

[Website](https://reactconfau.com/) - [Twitter](https://twitter.com/reactconfau) - [Facebook](https://www.facebook.com/reactconfau) -

[Instagram](https://www.instagram.com/reactconfau/)

React Barcamp Cologne 2020 {/*react-barcamp-cologne-2020*/}

February 1-2, 2020 in Cologne, Germany

[Website](https://react-barcamp.de/) - [Twitter](https://twitter.com/ReactBarcamp) - [Facebook](https://www.facebook.com/reactbarcamp)

React Day Berlin 2019 {/*react-day-berlin-2019*/}

December 6, 2019 in Berlin, Germany

[Website](https://reactday.berlin) - [Twitter](https://twitter.com/reactdayberlin) - [Facebook](https://www.facebook.com/reactdayberlin/) - [Videos](https://www.youtube.com/reactdayberlin)

React Summit 2019 {/*react-summit-2019*/}

November 30, 2019 in Lagos, Nigeria

[Website](https://reactsummit2019.splashthat.com) -[Twitter](https://twitter.com/react_summit)

React Conf Brasil 2019 {/*react-conf-brasil-2019*/}

October 19, 2019 in São Paulo, BR

[Website](https://reactconf.com.br/) - [Twitter](https://twitter.com/reactconfbr) - [Facebook](https://www.facebook.com/ReactAdvanced) - [Slack](https://react.now.sh/)

React Advanced 2019 {/*react-advanced-2019*/}

October 25, 2019 in London, UK

[Website](https://reactadvanced.com) - [Twitter](http://twitter.com/reactadvanced) - [Facebook](https://www.facebook.com/ReactAdvanced) - [Videos](https://youtube.com/c/ReactConferences)

React Conf 2019 {/*react-conf-2019*/}

October 24-25, 2019 in Henderson, Nevada USA

[Website](https://conf.reactjs.org/) - [Twitter](https://twitter.com/reactjs)

React Alicante 2019 {/*react-alicante-2019*/}

September 26-28, 2019 in Alicante, Spain

[Website](http://reactalicante.es/) - [Twitter](https://twitter.com/reactalicante) - [Facebook](https://www.facebook.com/ReactAlicante)

React India 2019 {/*react-india-2019*/}

September 26-28, 2019 in Goa, India

[Website](https://www.reactindia.io/) - [Twitter](https://twitter.com/react_india) - [Facebook](https://www.facebook.com/ReactJSIndia)

React Boston 2019 {/*react-boston-2019*/}

September 21-22, 2019 in Boston, Massachusetts USA

[Website](https://www.reactboston.com/) - [Twitter](https://twitter.com/reactboston)

React Live 2019 {/*react-live-2019*/}

September 13th, 2019. Amsterdam, The Netherlands

[Website](https://www.reactlive.nl/) - [Twitter](https://twitter.com/reactlivenl)

React New York 2019 {/*react-new-york-2019*/}

September 13th, 2019. New York, USA

[Website](https://reactnewyork.com/) - [Twitter](https://twitter.com/reactnewyork)

ComponentsConf 2019 {/*componentsconf-2019*/}

September 6, 2019 in Melbourne, Australia

[Website](https://www.componentsconf.com.au/) - [Twitter](https://twitter.com/componentsconf)

React Native EU 2019 {/*react-native-eu-2019*/}

September 5-6 in Wroc■aw, Poland

[Website](https://react-native.eu) - [Twitter](https://twitter.com/react_native_eu) - [Facebook](https://www.facebook.com/reactnativeeu)

React Conf Iran 2019 {/*react-conf-iran-2019*/}

August 29, 2019. Tehran, Iran.

[Website](https://reactconf.ir/) - [Videos](https://www.youtube.com/playlist?list=PL-VNqZFI5Nf-Nsj0rD3CWXGPkH-DI_0VY) - [Highlights](https://github.com/ReactConf/react-conf-highlights)

React Rally 2019 {/*react-rally-2019*/}

August 22-23, 2019. Salt Lake City, USA.

[Website](https://www.reactrally.com/) - [Twitter](https://twitter.com/ReactRally) - [Instagram](https://www.instagram.com/reactrally/)

Chain React 2019 {/*chain-react-2019*/}

July 11-12, 2019. Portland, OR, USA.

[Website](https://infinite.red/ChainReactConf)

React Loop 2019 {/*react-loop-2019*/}

June 21, 2019 Chicago, Illinois USA

[Website](https://reactloop.com) - [Twitter](https://twitter.com/ReactLoop)

React Norway 2019 {/*react-norway-2019*/}

June 12, 2019. Larvik, Norway

[Website](https://reactnorway.com) - [Twitter](https://twitter.com/ReactNorway)

ReactNext 2019 {/*reactnext-2019*/}

June 11, 2019. Tel Aviv, Israel

[Website](https://react-next.com) - [Twitter](https://twitter.com/ReactNext) - [Videos](https://www.youtube.com/channel/UC3BT8hh3yTTYxbLQy_wbk2w)

React Conf Armenia 2019 {/*react-conf-armenia-2019*/}

May 25, 2019 in Yerevan, Armenia

[Website](https://reactconf.am/) - [Twitter](https://twitter.com/ReactConfAM) - [Facebook](https://www.facebook.com/reactconf.am/) - [YouTube](https://www.youtube.com/c/JavaScriptConferenceArmenia) - [CFP](http://bit.ly/speakReact)

ReactEurope 2019 {/*reacteurope-2019*/}

May 23-24, 2019 in Paris, France

[Videos](https://www.youtube.com/c/ReacteuropeOrgConf)

React.NotAConf 2019 {/*reactnotaconf-2019*/}

May 11 in Sofia, Bulgaria

[Website](http://react-not-a-conf.com/) - [Twitter](https://twitter.com/reactnotaconf) - [Facebook](https://www.facebook.com/events/780891358936156)

ReactJS Girls Conference {/*reactjs-girls-conference*/}

May 3, 2019 in London, UK

[Website](https://reactjsgirls.com/) - [Twitter](https://twitter.com/reactjsgirls)

React Finland 2019 {/*react-finland-2019*/}

April 24-26 in Helsinki, Finland

[Website](https://react-finland.fi/) - [Twitter](https://twitter.com/ReactFinland)

React Amsterdam 2019 {/*react-amsterdam-2019*/}

April 12, 2019 in Amsterdam, The Netherlands

[Website](https://reactsummit.com) - [Twitter](https://twitter.com/reactsummit) - [Facebook](https://www.facebook.com/reactamsterdam) - [Videos](https://youtube.com/c/ReactConferences)

App.js Conf 2019 {/*appjs-conf-2019*/}

April 4-5, 2019 in Kraków, Poland

[Website](https://appjs.co) - [Twitter](https://twitter.com/appjsconf)

Reactathon 2019 {/*reactathon-2019*/}

March 30-31, 2019 in San Francisco, USA

[Website](https://www.reactathon.com/) - [Twitter](https://twitter.com/reactathon)

React Iran 2019 {/*react-iran-2019*/}

January 31, 2019 in Tehran, Iran

[Website](http://reactiran.com) - [Instagram](https://www.instagram.com/reactiran/)

React Day Berlin 2018 {/*react-day-berlin-2018*/}

November 30, Berlin, Germany

[Website](https://reactday.berlin) - [Twitter](https://twitter.com/reactdayberlin) - [Facebook](https://www.facebook.com/reactdayberlin/) - [Videos](https://www.youtube.com/channel/UC1EYHmQYBUJjkmL6OtK4rlw)

ReactNext 2018 {/*reactnext-2018*/}

November 4 in Tel Aviv, Israel

[Website](https://react-next.com) - [Twitter](https://twitter.com/ReactNext) - [Facebook](https://facebook.com/ReactNext2016)

React Conf 2018 {/*react-conf-2018*/}

October 25-26 in Henderson, Nevada USA

[Website](https://conf.reactjs.org/)

React Conf Brasil 2018 {/*react-conf-brasil-2018*/}

October 20 in Sao Paulo, Brazil

[Website](http://reactconfbr.com.br) - [Twitter](https://twitter.com/reactconfbr) - [Facebook](https://www.facebook.com/reactconf)

ReactJS Day 2018 {/*reactjs-day-2018*/}

October 5 in Verona, Italy

[Website](http://2018.reactjsday.it) - [Twitter](https://twitter.com/reactjsday)

React Boston 2018 {/*react-boston-2018*/}

September 29-30 in Boston, Massachusetts USA

[Website](http://www.reactboston.com/) - [Twitter](https://twitter.com/ReactBoston)

React Alicante 2018 {/*react-alicante-2018*/}

September 13-15 in Alicante, Spain

[Website](http://reactalicante.es) - [Twitter](https://twitter.com/ReactAlicante)

React Native EU 2018 {/*react-native-eu-2018*/}

September 5-6 in Wroc■aw, Poland

[Website](https://react-native.eu) - [Twitter](https://twitter.com/react_native_eu) - [Facebook](https://www.facebook.com/reactnativeeu)

Byteconf React 2018 {/*byteconf-react-2018*/}

August 31 streamed online, via Twitch

[Website](https://byteconf.com) - [Twitch](https://twitch.tv/byteconf) - [Twitter](https://twitter.com/byteconf)

ReactFoo Delhi {/*reactfoo-delhi*/}

August 18 in Delhi, India

[Website](https://reactfoo.in/2018-delhi/) - [Twitter](https://twitter.com/reactfoo) - [Past talks](https://hasgeek.tv)

React DEV Conf China {/*react-dev-conf-china*/}

August 18 in Guangzhou, China

[Website](https://react.w3ctech.com)

React Rally 2018{/*react-rally-2018*/}

August 16-17 in Salt Lake City, Utah USA

[Website](http://www.reactrally.com) - [Twitter](https://twitter.com/reactrally)

Chain React 2018 {/*chain-react-2018*/}

July 11-13 in Portland, Oregon USA

[Website](https://infinite.red/ChainReactConf) - [Twitter](https://twitter.com/chainreactconf)

ReactFoo Mumbai {/*reactfoo-mumbai*/}

May 26 in Mumbai, India

[Website](https://reactfoo.in/2018-mumbai/) - [Twitter](https://twitter.com/reactfoo) - [Past talks](https://hasgeek.tv)

ReactEurope 2018 {/*reacteurope-2018*/}

May 17-18 in Paris, France

[Videos](https://www.youtube.com/c/ReacteuropeOrgConf)

React.NotAConf 2018 {/*reactnotaconf-2018*/}

April 28 in Sofia, Bulgaria

[Website](http://react-not-a-conf.com/) - [Twitter](https://twitter.com/reactnotaconf) - [Facebook](https://www.facebook.com/groups/1614950305478021/)

React Finland 2018 {/*react-finland-2018*/}

April 24-26 in Helsinki, Finland

[Website](https://react-finland.fi/) - [Twitter](https://twitter.com/ReactFinland)

React Amsterdam 2018 {/*react-amsterdam-2018*/}

April 13 in Amsterdam, The Netherlands

[Website](https://reactsummit.com) - [Twitter](https://twitter.com/reactsummit) - [Facebook](https://www.facebook.com/reactamsterdam)

React Native Camp UA 2018 {/*react-native-camp-ua-2018*/}

March 31 in Kiev, Ukraine

[Website](http://reactnative.com.ua/) - [Twitter](https://twitter.com/reactnativecamp) - [Facebook](https://www.facebook.com/reactnativecamp/)

Reactathon 2018 {/*reactathon-2018*/}

March 20-22 in San Francisco, USA

[Website](https://www.reactathon.com/) - [Twitter](https://twitter.com/reactathon) - [Videos (fundamentals)](https://www.youtube.com/watch?v=knn364bssQU&list=PLRvKvw42Rc7OWK5s-YGGFSmByDzzgC0HP), [Videos (advanced day1)](https://www.youtube.com/watch?v=57hmk4GvJpk&list=PLRvKvw42Rc7N0QpX2Rc5CdrqGuxzwD_0H), [Videos (advanced day2)](https://www.youtube.com/watch?v=1hvQ8p8q0a0&list=PLRvKvw42Rc7Ne46QAjWNWFo1Jf0mQdnIW)

ReactFest 2018 {/*reactfest-2018*/}

March 8-9 in London, UK

[Website](https://reactfest.uk/) - [Twitter](https://twitter.com/ReactFest) - [Videos](https://www.youtube.com/watch?v=YOCrJ5vRCnw&list=PLRgweB8YtNRt-Sf-A0y446wTJNUaAAmle)

AgentConf 2018 {/*agentconf-2018*/}

January 25-28 in Dornbirn, Austria

[Website](http://agent.sh/)

ReactFoo Pune {/*reactfoo-pune*/}

January 19-20, Pune, India

[Website](https://reactfoo.in/2018-pune/) - [Twitter](https://twitter.com/ReactFoo)

React Day Berlin 2017 {/*react-day-berlin-2017*/}

December 2, Berlin, Germany

[Website](https://reactday.berlin) - [Twitter](https://twitter.com/reactdayberlin) - [Facebook](https://www.facebook.com/reactdayberlin/) - [Videos](https://www.youtube.com/watch?v=UnNLJvHKfSY&list=PL-3BrJ5CiIx5GoXci54-VsrO6GwLhSHEK)

React Seoul 2017 {/*react-seoul-2017*/}

November 4 in Seoul, South Korea

[Website](http://seoul.reactjs.kr/en)

ReactiveConf 2017 {/*reactiveconf-2017*/}

October 25–27, Bratislava, Slovakia

[Website](https://reactiveconf.com) - [Videos](https://www.youtube.com/watch?v=BOKxSFB2hOE&list=PLa2ZZ09WYepMB-I7AiDjDYR8TjO8uoNjs)

React Summit 2017 {/*react-summit-2017*/}

October 21 in Lagos, Nigeria

[Website](https://reactsummit2017.splashthat.com/) - [Twitter](https://twitter.com/DevCircleLagos/) - [Facebook](https://www.facebook.com/groups/DevCLagos/)

State.js Conference 2017 {/*statejs-conference-2017*/}

October 13 in Stockholm, Sweden

[Website](https://statejs.com/)

React Conf Brasil 2017 {/*react-conf-brasil-2017*/}

October 7 in Sao Paulo, Brazil

[Website](http://reactconfbr.com.br) - [Twitter](https://twitter.com/reactconfbr) - [Facebook](https://www.facebook.com/reactconf/)

ReactJS Day 2017 {/*reactjs-day-2017*/}

October 6 in Verona, Italy

[Website](http://2017.reactjsday.it) - [Twitter](https://twitter.com/reactjsday) - [Videos](https://www.youtube.com/watch?v=bUqqJPIgjNU&list=PLWK9j6ps_unl293VhhN4RYMCISxye3xH9)

React Alicante 2017 {/*react-alicante-2017*/}

September 28-30 in Alicante, Spain

[Website](http://reactalicante.es) - [Twitter](https://twitter.com/ReactAlicante) - [Videos](https://www.youtube.com/watch?v=UMZvRCWo6Dw&list=PLd7nkr8mN0sWvBH_s0foCE6eZTX8BmLUM)

React Boston 2017 {/*react-boston-2017*/}

September 23-24 in Boston, Massachusetts USA

[Website](http://www.reactboston.com/) - [Twitter](https://twitter.com/ReactBoston) - [Videos](https://www.youtube.com/watch?v=2iPE5l3cl_s&list=PL-fCkV3wv4ub8zJMIhmrrLcQqSR5XPlIT)

ReactFoo 2017 {/*reactfoo-2017*/}

September 14 in Bangalore, India

[Website](https://reactfoo.in/2017/) - [Videos](https://www.youtube.com/watch?v=3G6tMg29Wnw&list=PL279M8GbNsespKKm1L0NAzYLO6gU5LvfH)

ReactNext 2017 {/*reactnext-2017*/}

September 8-10 in Tel Aviv, Israel

[Website](http://react-next.com/) - [Twitter](https://twitter.com/ReactNext) - [Videos (Hall A)](https://www.youtube.com/watch?v=eKXQw5kR86c&list=PLMYVq3z1QxSqq6D7jxVdqttOX7H_Brq8Z), [Videos (Hall B)](https://www.youtube.com/watch?v=1InokWxYGnE&list=PLMYVq3z1QxSqCZmaqgTXLsrcJ8mZmBF7T)

React Native EU 2017 {/*react-native-eu-2017*/}

September 6-7 in Wroclaw, Poland

[Website](http://react-native.eu/) - [Videos](https://www.youtube.com/watch?v=453oKJAqfy0&list=PLzUKC1ci01h_hkn7_KoFA-Au0DXLAQZR7)

React Rally 2017 {/*react-rally-2017*/}

August 24-25 in Salt Lake City, Utah USA

[Website](http://www.reactrally.com) - [Twitter](https://twitter.com/reactrally) - [Videos](https://www.youtube.com/watch?v=f4KnHNCZcH4&list=PLUD4kD-wL_zZUhvAIHJjueJDPr6qHvkni)

Chain React 2017 {/*chain-react-2017*/}

July 10-11 in Portland, Oregon USA

[Website](https://infinite.red/ChainReactConf) - [Twitter](https://twitter.com/chainreactconf) - [Videos](https://www.youtube.com/watch?v=cz5BzwgATpc&list=PLFHvL21g9bk3RxJ1Ut5nR_uTZFVOxu522)

ReactEurope 2017 {/*reacteurope-2017*/}

May 18th & 19th in Paris, France

[Videos](https://www.youtube.com/c/ReacteuropeOrgConf)

React Amsterdam 2017 {/*react-amsterdam-2017*/}

April 21st in Amsterdam, The Netherlands

[Website](https://reactsummit.com) - [Twitter](https://twitter.com/reactsummit) - [Videos](https://youtube.com/c/ReactConferences)

React London 2017 {/*react-london-2017*/}

March 28th at the [QEII Centre, London](http://qeiicentre.london/)

[Website](http://react.london/) - [Videos](https://www.youtube.com/watch?v=2j9rSur_mnk&list=PLW6ORi0XZU0CFjdoYeC0f5QReBG-NeNKJ)

React Conf 2017 {/*react-conf-2017*/}

March 13-14 in Santa Clara, CA

[Website](http://conf.reactjs.org/) - [Videos](https://www.youtube.com/watch?v=7HSd1sk07uU&list=PLb0IAmt7-GS3fZ46IGFirdqKTIxlws7e0)

Agent Conference 2017 {/*agent-conference-2017*/}

January 20-21 in Dornbirn, Austria

[Website](http://agent.sh/)

React Remote Conf 2016 {/*react-remote-conf-2016*/}

October 26-28 online

[Website](https://allremoteconfs.com/react-2016) - [Schedule](https://allremoteconfs.com/react-2016#schedule)

Reactive 2016 {/*reactive-2016*/}

October 26-28 in Bratislava, Slovakia

[Website](https://reactiveconf.com/)

ReactNL 2016 {/*reactnl-2016*/}

October 13 in Amsterdam, The Netherlands

[Website](http://reactnl.org/) - [Schedule](http://reactnl.org/#program)

ReactNext 2016 {/*reactnext-2016*/}

September 15 in Tel Aviv, Israel

[Website](http://react-next.com/) - [Schedule](http://react-next.com/#schedule) - [Videos](https://www.youtube.com/channel/UC3BT8hh3yTTYxbLQy_wbk2w)

ReactRally 2016 {/*reactrally-2016*/}

August 25-26 in Salt Lake City, UT

[Website](http://www.reactrally.com/) - [Schedule](http://www.reactrally.com/#/schedule) - [Videos](https://www.youtube.com/playlist?list=PLUD4kD-wL_zYSfU3tIYsb4WqfFQzO_EjQ)

ReactEurope 2016 {/*reacteurope-2016*/}

June 2 & 3 in Paris, France

[Videos](https://www.youtube.com/c/ReacteuropeOrgConf)

React Amsterdam 2016 {/*react-amsterdam-2016*/}

April 16 in Amsterdam, The Netherlands

[Website](https://reactsummit.com) - [Twitter](https://twitter.com/reactsummit) - [Facebook](https://www.facebook.com/reactamsterdam) - [Videos](https://youtube.com/c/ReactConferences)

React.js Conf 2016 {/*reactjs-conf-2016*/}

February 22 & 23 in San Francisco, CA

[Website](http://conf2016.reactjs.org/) - [Schedule](http://conf2016.reactjs.org/schedule.html) - [Videos](https://www.youtube.com/playlist?list=PLb0IAmt7-GS0M8Q95RIc2lOM6nc77q1IY)

Reactive 2015 {/*reactive-2015*/}

November 2-4 in Bratislava, Slovakia

[Website](https://reactive2015.com/) - [Schedule](https://reactive2015.com/schedule_speakers.html#schedule)

ReactEurope 2015 {/*reacteurope-2015*/}

July 2 & 3 in Paris, France

[Videos](https://www.youtube.com/c/ReacteuropeOrgConf)

React.js Conf 2015 {/*reactjs-conf-2015*/}

January 28 & 29 in Facebook HQ, CA

[Website](http://conf2015.reactjs.org/) - [Schedule](http://conf2015.reactjs.org/schedule.html) - [Videos](https://www.youtube.com/playlist?list=PLb0IAmt7-GS1cbw4qonlQztYV1TAW0sCr)

---

title: Docs Contributors

<Intro>

React documentation is written and maintained by the [React team](/community/team) and [external contributors.](https://github.com/reactjs/reactjs.org/graphs/contributors) On this page, we'd like to thank a few people who've made significant contributions to this site.

</Intro>

Content {/*content*/}

* [Rachel Nabors](https://twitter.com/RachelNabors): editing, writing, illustrating

* [Dan Abramov](https://twitter.com/dan_abramov): writing, curriculum design

* [Sylwia Vargas](https://twitter.com/SylwiaVargas): example code

* [Rick Hanlon](https://twitter.com/rickhanlonii): writing

* [David McCabe](https://twitter.com/mcc_abe): writing

* [Sophie Alpert](https://twitter.com/sophiebits): writing

* [Pete Hunt](https://twitter.com/floydophone): writing

* [Andrew Clark](https://twitter.com/acdlite): writing

* [Matt Carroll](https://twitter.com/mattcarrollcode): editing, writing

* [Natalia Tepluhina](https://twitter.com/n_tepluhina): reviews, advice

* [Sebastian Markbåge](https://twitter.com/sebmarkbage): feedback

Design {/*design*/}

* [Dan Lebowitz](https://twitter.com/lebo): site design

* [Razvan Gradinar](https://dribbble.com/GradinarRazvan): sandbox design

* [Maggie Appleton](https://maggieappleton.com/): diagram system

* [Sophie Alpert](https://twitter.com/sophiebits): color-coded explanations

Development {/*development*/}

* [Jared Palmer](https://twitter.com/jaredpalmer): site development

* [ThisDotLabs](https://www.thisdot.co/) ([Dane Grant](https://twitter.com/danecando), [Dustin Goodman](https://twitter.com/dustinsgoodman)): site development

* [CodeSandbox](https://codesandbox.io/) ([Ives van Hoorne](https://twitter.com/CompuIves), [Alex Moldovan](https://twitter.com/alexnmoldovan), [Jasper De Moor](https://twitter.com/JasperDeMoor), [Danilo Woznica](https://twitter.com/danilowoz)): sandbox integration

* [Dan Abramov](https://twitter.com/dan_abramov): site development

* [Rick Hanlon](https://twitter.com/rickhanlonii): site development

* [Harish Kumar](https://www.strek.in/): development and maintenance

* [Luna Ruan](https://twitter.com/lunaruan): sandbox improvements

We'd also like to thank countless alpha testers and community members who gave us feedback along the way.

---

title: React Meetups

<Intro>

Do you have a local React.js meetup? Add it here! (Please keep the list alphabetical)

</Intro>

Albania {/*albania*/}

* [Tirana](https://www.meetup.com/React-User-Group-Albania/)

Argentina {/*argentina*/}

* [Buenos Aires](https://www.meetup.com/es/React-en-Buenos-Aires)

* [Rosario](https://www.meetup.com/es/reactrosario)

Australia {/*australia*/}

* [Brisbane](https://www.meetup.com/reactbris/)
* [Melbourne](https://www.meetup.com/React-Melbourne/)
* [Sydney](https://www.meetup.com/React-Sydney/)

Austria {/*austria*/}

* [Vienna](https://www.meetup.com/Vienna-ReactJS-Meetup/)

Belgium {/*belgium*/}

* [Belgium](https://www.meetup.com/ReactJS-Belgium/)

Brazil {/*brazil*/}

* [Belo Horizonte](https://www.meetup.com/reactbh/)
* [Curitiba](https://www.meetup.com/pt-br/ReactJS-CWB/)
* [Florianópolis](https://www.meetup.com/pt-br/ReactJS-Floripa/)
* [Goiânia](https://www.meetup.com/pt-br/React-Goiania/)
* [Joinville](https://www.meetup.com/pt-BR/React-Joinville/)
* [Juiz de Fora](https://www.meetup.com/pt-br/React-Juiz-de-Fora/)
* [Maringá](https://www.meetup.com/pt-BR/React-Maringa/)
* [Porto Alegre](https://www.meetup.com/pt-BR/React-Porto-Alegre/)
* [Rio de Janeiro](https://www.meetup.com/pt-BR/React-Rio-de-Janeiro/)
* [Salvador](https://www.meetup.com/pt-BR/ReactSSA)
* [São Paulo](https://www.meetup.com/pt-BR/ReactJS-SP/)
* [Vila Velha](https://www.meetup.com/pt-BR/React-ES/)

Bolivia {/*bolivia*/}

* [Bolivia](https://www.meetup.com/ReactBolivia/)

Canada {/*canada*/}

* [Halifax, NS](https://www.meetup.com/Halifax-ReactJS-Meetup/)
* [Montreal, QC - React Native](https://www.meetup.com/fr-FR/React-Native-MTL/)
* [Vancouver, BC](https://www.meetup.com/ReactJS-Vancouver-Meetup/)
* [Ottawa, ON](https://www.meetup.com/Ottawa-ReactJS-Meetup/)
* [Toronto, ON](https://www.meetup.com/Toronto-React-Native/events/)

Chile {/*chile*/}

* [Santiago](https://www.meetup.com/es-ES/react-santiago/)

China {/*china*/}

* [Beijing](https://www.meetup.com/Beijing-ReactJS-Meetup/)

Colombia {/*colombia*/}

* [Bogotá](https://www.meetup.com/meetup-group-iHIeHykY/)
* [Medellin](https://www.meetup.com/React-Medellin/)
* [Cali](https://www.meetup.com/reactcali/)

Denmark {/*denmark*/}

* [Aalborg](https://www.meetup.com/Aalborg-React-React-Native-Meetup/)
* [Aarhus](https://www.meetup.com/Aarhus-ReactJS-Meetup/)

Egypt {/*egypt*/}

* [Cairo](https://www.meetup.com/react-cairo/)

England (UK) {/*england-uk*/}

* [Manchester](https://www.meetup.com/Manchester-React-User-Group/)
* [React.JS Girls London](https://www.meetup.com/ReactJS-Girls-London/)
* [React London : Bring Your Own Project](https://www.meetup.com/React-London-Bring-Your-Own-Project/)

France {/*france*/}

* [Nantes](https://www.meetup.com/React-Nantes/)
* [Lille](https://www.meetup.com/ReactBeerLille/)
* [Paris](https://www.meetup.com/ReactJS-Paris/)

Germany {/*germany*/}

* [Cologne](https://www.meetup.com/React-Cologne/)
* [Düsseldorf](https://www.meetup.com/de-DE/ReactJS-Meetup-Dusseldorf/)
* [Hamburg](https://www.meetup.com/Hamburg-React-js-Meetup/)
* [Karlsruhe](https://www.meetup.com/react_ka/)
* [Kiel](https://www.meetup.com/Kiel-React-Native-Meetup/)
* [Munich](https://www.meetup.com/ReactJS-Meetup-Munich/)
* [React Berlin](https://www.meetup.com/React-Open-Source/)

Greece {/*greece*/}

* [Athens](https://www.meetup.com/React-To-React-Athens-MeetUp/)
* [Thessaloniki](https://www.meetup.com/Thessaloniki-ReactJS-Meetup/)

Hungary {/*hungary*/}

* [Budapest](https://www.meetup.com/React-Budapest/)

India {/*india*/}

* [Ahmedabad](https://www.meetup.com/react-ahmedabad/)
* [Bangalore](https://www.meetup.com/ReactJS-Bangalore/)
* [Bangalore](https://www.meetup.com/React-Native-Bangalore-Meetup)
* [Chandigarh](https://www.meetup.com/Chandigarh-React-Developers/)
* [Chennai](https://www.meetup.com/React-Chennai/)
* [Delhi NCR](https://www.meetup.com/React-Delhi-NCR/)
* [Jaipur](https://www.meetup.com/JaipurJS-Developer-Meetup/)
* [Pune](https://www.meetup.com/ReactJS-and-Friends/)

Indonesia {/*indonesia*/}

* [Indonesia](https://www.meetup.com/reactindonesia/)

Ireland {/*ireland*/}

* [Dublin](https://www.meetup.com/ReactJS-Dublin/)

Israel {/*israel*/}

* [Tel Aviv](https://www.meetup.com/ReactJS-Israel/)

Italy {/*italy*/}

* [Milan](https://www.meetup.com/React-JS-Milano/)

Kenya {/*kenya*/}

* [Nairobi - Reactdevske](https://kommunity.com/reactjs-developer-community-kenya-reactdevske)

Malaysia {/*malaysia*/}

* [Kuala Lumpur](https://www.kl-react.com/)
* [Penang](https://www.facebook.com/groups/reactpenang/)

Netherlands {/*netherlands*/}

* [Amsterdam](https://www.meetup.com/React-Amsterdam/)

New Zealand {/*new-zealand*/}

* [Wellington](https://www.meetup.com/React-Wellington/)

Norway {/*norway*/}

* [Norway](https://reactjs-norway.webflow.io/)
* [Oslo](https://www.meetup.com/ReactJS-Oslo-Meetup/)

Pakistan {/*pakistan*/}

* [Karachi](https://www.facebook.com/groups/902678696597634/)
* [Lahore](https://www.facebook.com/groups/ReactjsLahore/)

Panama {/*panama*/}

* [Panama](https://www.meetup.com/React-Panama/)

Peru {/*peru*/}

* [Lima](https://www.meetup.com/ReactJS-Peru/)

Philippines {/*philippines*/}

* [Manila](https://www.meetup.com/reactjs-developers-manila/)
* [Manila - ReactJS PH](https://www.meetup.com/ReactJS-Philippines/)

Poland {/*poland*/}

* [Warsaw](https://www.meetup.com/React-js-Warsaw/)
* [Wroc■aw](https://www.meetup.com/ReactJS-Wroclaw/)

Portugal {/*portugal*/}

* [Lisbon](https://www.meetup.com/JavaScript-Lisbon/)

Scotland (UK) {/*scotland-uk*/}

* [Edinburgh](https://www.meetup.com/React-Scotland/)

Spain {/*spain*/}

* [Barcelona](https://www.meetup.com/ReactJS-Barcelona/)
* [Canarias](https://www.meetup.com/React-Canarias/)

Sweden {/*sweden*/}

* [Goteborg](https://www.meetup.com/ReactJS-Goteborg/)
* [Stockholm](https://www.meetup.com/Stockholm-ReactJS-Meetup/)

Switzerland {/*switzerland*/}

* [Zurich](https://www.meetup.com/Zurich-ReactJS-Meetup/)

Turkey {/*turkey*/}

* [Istanbul](https://kommunity.com/reactjs-istanbul)

Ukraine {/*ukraine*/}

* [Kyiv](https://www.meetup.com/Kyiv-ReactJS-Meetup)

US {/*us*/}

* [Ann Arbor, MI - ReactJS](https://www.meetup.com/AnnArbor-jsx/)
* [Atlanta, GA - ReactJS](https://www.meetup.com/React-ATL/)
* [Austin, TX - ReactJS](https://www.meetup.com/ReactJS-Austin-Meetup/)
* [Boston, MA - ReactJS](https://www.meetup.com/ReactJS-Boston/)
* [Boston, MA - React Native](https://www.meetup.com/Boston-React-Native-Meetup/)
* [Charlotte, NC - ReactJS](https://www.meetup.com/ReactJS-Charlotte/)

* [Charlotte, NC - React Native](https://www.meetup.com/cltreactnative/)
* [Chicago, IL - ReactJS](https://www.meetup.com/React-Chicago/)
* [Cleveland, OH - ReactJS](https://www.meetup.com/Cleveland-React/)
* [Columbus, OH - ReactJS](https://www.meetup.com/ReactJS-Columbus-meetup/)
* [Dallas, TX - ReactJS](https://www.meetup.com/ReactDallas/)
* [Dallas, TX - [Remote] React JS](https://www.meetup.com/React-JS-Group/)
* [Detroit, MI - Detroit React User Group](https://www.meetup.com/Detroit-React-User-Group/)
* [Indianapolis, IN - React.Indy](https://www.meetup.com/React-Indy)
* [Irvine, CA - ReactJS](https://www.meetup.com/ReactJS-OC/)
* [Kansas City, MO - ReactJS](https://www.meetup.com/Kansas-City-React-Meetup/)
* [Las Vegas, NV - ReactJS](https://www.meetup.com/ReactVegas/)
* [Leesburg, VA - ReactJS](https://www.meetup.com/React-NOVA/)
* [Los Angeles, CA - ReactJS](https://www.meetup.com/socal-react/)
* [Los Angeles, CA - React Native](https://www.meetup.com/React-Native-Los-Angeles/)
* [Miami, FL - ReactJS](https://www.meetup.com/React-Miami/)
* [Nashville, TN - ReactJS](https://www.meetup.com/NashReact-Meetup/)
* [New York, NY - ReactJS](https://www.meetup.com/NYC-Javascript-React-Group/)
* [New York, NY - React Ladies](https://www.meetup.com/React-Ladies/)
* [New York, NY - React Native](https://www.meetup.com/React-Native-NYC/)
* [New York, NY - useReactNYC](https://www.meetup.com/useReactNYC/)
* [Omaha, NE - ReactJS/React Native](https://www.meetup.com/omaha-react-meetup-group/)
* [Palo Alto, CA - React Native](https://www.meetup.com/React-Native-Silicon-Valley/)
* [Philadelphia, PA - ReactJS](https://www.meetup.com/Reactadelphia/)
* [Phoenix, AZ - ReactJS](https://www.meetup.com/ReactJS-Phoenix/)
* [Pittsburgh, PA - ReactJS/React Native](https://www.meetup.com/ReactPgh/)
* [Portland, OR - ReactJS](https://www.meetup.com/Portland-ReactJS/)
* [Provo, UT - ReactJS](https://www.meetup.com/ReactJS-Utah/)
* [Sacramento, CA - ReactJS](https://www.meetup.com/Sacramento-ReactJS-Meetup/)
* [San Diego, CA - San Diego JS](https://www.meetup.com/sandiegojs/)
* [San Francisco - Real World React](https://www.meetup.com/Real-World-React)
* [San Francisco - ReactJS](https://www.meetup.com/ReactJS-San-Francisco/)
* [San Francisco, CA - React Native](https://www.meetup.com/React-Native-San-Francisco/)
* [San Ramon, CA - TriValley Coders](https://www.meetup.com/trivalleycoders/)
* [Santa Monica, CA - ReactJS](https://www.meetup.com/Los-Angeles-ReactJS-User-Group/)
* [Seattle, WA - React Native](https://www.meetup.com/Seattle-React-Native-Meetup/)

* [Seattle, WA - ReactJS](https://www.meetup.com/seattle-react-js/)

* [Tampa, FL - ReactJS](https://www.meetup.com/ReactJS-Tampa-Bay/)

* [Tucson, AZ - ReactJS](https://www.meetup.com/Tucson-ReactJS-Meetup/)

* [Washington, DC - ReactJS](https://www.meetup.com/React-DC/)

---

title: "Meet the Team"

<Intro>

React development is led by a dedicated team working full time at Meta. It also receives contributions from people all over the world.

</Intro>

React Core {/*react-core*/}

The React Core team members work full time on the core component APIs, the engine that powers React DOM and React Native, React DevTools, and the React documentation website.

Current members of the React team are listed in alphabetical order below.

<TeamMember name="Andrew Clark" permalink="andrew-clark" photo="/images/team/acdlite.jpg" github="acdlite" twitter="acdlite" title="Engineer at Vercel">

Andrew got started with web development by making sites with WordPress, and eventually tricked himself into doing JavaScript. His favorite pastime is karaoke. Andrew is either a Disney villain or a Disney princess, depending on the day.

</TeamMember>

<TeamMember name="Andrey Lunyov" permalink="andrey-lunyov" photo="/images/team/andrey-lunyov.jpg" github="alunyov" twitter="alunyov" title="Engineer at Meta">

Andrey started his career as a designer and then gradually transitioned into web development. After joining the React Data team at Meta he worked on adding an incremental JavaScript compiler to Relay, and then later on, worked on removing the same compiler from Relay. Outside of work, Andrey likes to play music and engage in various sports.

</TeamMember>

<TeamMember name="Dan Abramov" permalink="dan-abramov" photo="/images/team/gaearon.jpg" github="gaearon" twitter="dan_abramov" title="Engineer at Meta">

Dan got into programming after he accidentally discovered Visual Basic inside Microsoft PowerPoint. He has found his true calling in turning [Sebastian](#sebastian-markbåge)'s tweets into long-form blog posts. Dan occasionally wins at Fortnite by hiding in a bush until the game ends.

</TeamMember>

<TeamMember name="Dave McCabe" permalink="dave-mccabe" photo="/images/team/dave-mccabe.jpg" github="davidmccabe" twitter="mcc_abe" title="Engineer at Meta">

An engineer by trade and outdoorsman at heart, David has long been an innovator in the field of programming-while-sunbathing. Besides surprising his colleagues with unique outdoor video-call backgrounds, he enjoys playing guitar (in sunlit meadows, of course) and martial arts (still indoors, gotta work on that).

</TeamMember>

<TeamMember name="Eli White" permalink="eli-white" photo="/images/team/eli-white.jpg" github="TheSavior" twitter="Eli_White" title="Engineering Manager at Meta">

Eli got into programming after he got suspended from middle school for hacking. He has been working on React and React Native since 2017. He enjoys eating treats, especially ice cream and apple pie. You can find Eli trying quirky activities like parkour, indoor skydiving, and aerial silks.

</TeamMember>

<TeamMember name="Jason Bonta" permalink="jason-bonta" photo="/images/team/jasonbonta.jpg" title="Engineering Manager at Meta">

Jason likes having large volumes of Amazon packages delivered to the office so that he can build forts. Despite literally walling himself off from his team at times and not understanding how for-of loops work, we appreciate him for the unique qualities he brings to his work.

</TeamMember>

<TeamMember name="Joe Savona" permalink="joe-savona" photo="/images/team/joe.jpg" github="josephsavona" twitter="en_JS" title="Engineer at Meta">

Joe was planning to major in math and philosophy but got into computer science after writing physics simulations in Matlab. Prior to React, he worked on Relay, RSocket.js, and the Skip programming language. While he’s not building some sort of reactive system he enjoys running, studying Japanese, and spending time with his family.

</TeamMember>

<TeamMember name="Josh Story" permalink="josh-story" photo="/images/team/josh.jpg" github="gnoff" twitter="joshcstory" title="Engineer at Vercel">

Josh majored in Mathematics and discovered programming while in college. His first professional developer job was to program insurance rate calculations in Microsoft Excel, the paragon of Reactive Programming which must be why he now works on React. In between that time Josh has been an IC, Manager, and Executive at a few startups. outside of work he likes to push his limits with cooking.

</TeamMember>

<TeamMember name="Kathryn Middleton" permalink="kathryn-middleton" photo="/images/team/kathryn-middleton.jpg" github="kmiddleton14" twitter="kmiddleton14" title="Engineering Manager at Meta">

Kathryn initially discovered web development when she wanted to make her myspace page look cool. She ended up majoring in Computer Science, and quickly became a huge fan of React building features on the Instagram.com team. Outside of work she loves playing pingpong, teaching spin classes, and going plant shopping.

</TeamMember>

<TeamMember name="Lauren Tan" permalink="lauren-tan" photo="/images/team/lauren.jpg" github="poteto" twitter="potetotes" personal="no.lol" title="Engineer at Meta">

Lauren’s programming career peaked when she first discovered the `<marquee>` tag. She’s been chasing that high ever since. When she’s not adding bugs into React, she enjoys dropping cheeky memes in chat, and playing all too many video games with her partner, and her dog Zelda.

</TeamMember>

<TeamMember name="Luna Ruan" permalink="luna-ruan" photo="/images/team/lunaruan.jpg" github="lunaruan" twitter="lunaruan" title="Independent Engineer">

Luna learned programming because she thought it meant creating video games. Instead, she ended up working on the Pinterest web app, and now on React itself. Luna doesn't want to make video games anymore, but she plans to do creative writing if she ever gets bored.

</TeamMember>

<TeamMember name="Luna Wei" permalink="luna-wei" photo="/images/team/luna-wei.jpg" github="lunaleaps" twitter="lunaleaps" title="Engineer at Meta">

Luna first learnt the fundamentals of python at the age of 6 from her father. Since then, she has been unstoppable. Luna aspires to be a gen z, and the road to success is paved with environmental advocacy, urban gardening and lots of quality time with her Voo-Doo’d (as pictured).

</TeamMember>

<TeamMember name="Matt Carroll" permalink="matt-carroll" photo="/images/team/matt-carroll.png" github="mattcarrollcode" twitter="mattcarrollcode" title="Developer Advocate at Meta">

Matt stumbled into coding, and since then, has become enamored with creating things in communities that can’t be created alone. Prior to React, he worked on YouTube, the Google Assistant, Fuchsia, and Google Cloud AI and Evernote. When he's not trying to make better developer tools he enjoys the mountains, jazz, and spending time with his family.

</TeamMember>

<TeamMember name="Mengdi Chen" permalink="mengdi-chen" photo="/images/team/mengdi-chen.jpg" github="mondaychen" twitter="mengdi_en" title="Engineer at Meta">

While working on his Digital Arts degree Mengdi was conceited about his front-end skills because his CSS worked perfectly even on IE6. But soon React opened a new door of programming for him, and he has been dreaming of joining the React team ever since. Outside of work, he is usually busy chasing his two kids around or collecting strange recipes.

</TeamMember>

<TeamMember name="Mofei Zhang" permalink="mofei-zhang" photo="/images/team/mofei-zhang.png" github="mofeiZ" title="Engineer at Meta">

Mofei started programming when she realized it can help her cheat in video games. She focused on operating systems in undergrad / grad school, but now finds herself happily tinkering on React. Outside of work, she enjoys debugging bouldering problems and planning her next backpacking trip(s).

</TeamMember>

<TeamMember name="Rick Hanlon" permalink="rick-hanlon" photo="/images/team/rickhanlonii.jpg" github="rickhanlonii" twitter="rickhanlonii" personal="rickhanlon.codes" title="Engineer at Meta">

Ricky majored in theoretical math and somehow found himself on the React Native team for a couple years before joining the React team. When he's not programming you can find him snowboarding,

biking, climbing, golfing, or closing GitHub issues that do not match the issue template.

</TeamMember>

<TeamMember name="Samuel Susla" permalink="samuel-susla" photo="/images/team/sam.jpg" github="sammy-SC" twitter="SamuelSusla" title="Engineer at Meta">

Samuel’s interest in programming started with the movie Matrix. He still has Matrix screen saver. Before working on React, he was focused on writing iOS apps. Outside of work, Samuel enjoys playing beach volleyball, squash, badminton and spending time with his family.

</TeamMember>

<TeamMember name="Sathya Gunasekaran " permalink="sathya-gunasekaran" photo="/images/team/sathya.jpg" github="gsathya" twitter="_gsathya" title="Engineer at Meta">

Sathya hated the Dragon Book in school but somehow ended up working on compilers all his career. When he's not compiling React components, he's either drinking coffee or eating yet another Dosa.

</TeamMember>

<TeamMember name="Sean Keegan" permalink="sean-keegan" photo="/images/team/sean-keegan.jpg" github="seanryankeegan" twitter="DevRelSean" title="Developer Advocate at Meta">

After a first career as a math teacher, Sean remembered that one intro to comp sci class he had to take as a prereq and thought “that was kind of fun!”. One coding bootcamp and several tech jobs later, Sean discovered developer advocacy and hasn’t looked back. Outside of work, Sean enjoys ultimate frisbee, video games, and researching (but rarely implementing) better ways to care for his houseplants.

</TeamMember>

<TeamMember name="Sebastian Markbåge" permalink="sebastian-markbåge" photo="/images/team/sebmarkbage.jpg" github="sebmarkbage" twitter="sebmarkbage" title="Engineer at Vercel">

Sebastian majored in psychology. He's usually quiet. Even when he says something, it often doesn't make sense to the rest of us until a few months later. The correct way to pronounce his surname is "mark-boa-geh" but he settled for "mark-beige" out of pragmatism -- and that's how he approaches React.

</TeamMember>

<TeamMember name="Sebastian Silbermann" permalink="sebastian-silbermann" photo="/images/team/sebsilbermann.jpg" github="eps1lon" twitter="sebsilbermann" title="Independent Engineer">

Sebastian learned programming to make the browser games he played during class more enjoyable. Eventually this lead to contributing to as much open source code as possible. Outside of coding he's busy making sure people don't confuse him with the other Sebastians and Zilberman of the React community.

</TeamMember>

<TeamMember name="Seth Webster" permalink="seth-webster" photo="/images/team/seth.jpg" github="sethwebster" twitter="sethwebster" personal="sethwebster.com" title="Engineering Manager at Meta">

Seth started programming as a kid growing up in Tucson, AZ. After school, he was bitten by the music bug and was a touring musician for about 10 years before returning to *work*, starting with Intuit. In his spare time, he loves [taking pictures](https://www.sethwebster.com) and flying for animal rescues in the northeastern United States.

</TeamMember>

<TeamMember name="Sophie Alpert" permalink="sophie-alpert" photo="/images/team/sophiebits.jpg" github="sophiebits" twitter="sophiebits" personal="sophiebits.com" title="Independent Engineer">

Four days after React was released, Sophie rewrote the entirety of her then-current project to use it, which she now realizes was perhaps a bit reckless. After she became the project's #1 committer, she wondered why she wasn't getting paid by Facebook like everyone else was and joined the team officially to lead React through its adolescent years. Though she quit that job years ago, somehow she's still in the team's group chats and “providing value”.

</TeamMember>

<TeamMember name="Tianyu Yao" permalink="tianyu-yao" photo="/images/team/tianyu.jpg" github="tyao1" twitter="tianyu0" title="Engineer at Meta">

Tianyu’s interest in computers started as a kid because he loves video games. So he majored in computer science and still plays childish games like League of Legends. When he is not in front of a computer, he enjoys playing with his two kittens, hiking and kayaking.

</TeamMember>

<TeamMember name="Yuzhi Zheng" permalink="yuzhi-zheng" photo="/images/team/yuzhi.jpg" github="yuzhi" twitter="yuzhiz" title="Engineering Manager at Meta">

Yuzhi studied Computer Science in school. She liked the instant gratification of seeing code come to life without having to physically be in a laboratory. Now she’s a manager in the React org. Before management, she used to work on the Relay data fetching framework. In her spare time, Yuzhi enjoys optimizing her life via gardening and home improvement projects.

</TeamMember>

Past contributors {/*past-contributors*/}

You can find the past team members and other people who significantly contributed to React over the years on the [acknowledgements](/community/acknowledgements) page.