Why React Re-Renders

So, I'll be honest. I had been working professionally with React ***for years*** without really understanding how React's re-rendering process worked. 😅

I think this is true for lots of React developers. We understand enough to get by, but if you ask a group of React developers a question like *"What triggers a re-render in React?"*, you'll likely get a handful of different hand-wavy answers.

There are a lot of misconceptions out there about this topic, and it can lead to a lot of uncertainty. If we don't understand React's render cycle, how can we understand how to use `React.memo`, or when we should wrap our functions in `useCallback`??

In this tutorial, we're going to build a mental model for when and why React re-renders. We'll also learn how to tell *why* a specific component re-rendered, using the React devtools.

> **Intended audience**
>
> This tutorial is written to help beginner-intermediate React developers get more comfortable with React. If you're taking your very first steps with React, you might wish to bookmark this one and come back to it in a few weeks!

The core React loop

So, let's start with a fundamental truth: **Every re-render in React starts with a state change.** It's the only "trigger" in React for a component to re-render.*

Now, that probably doesn't sound right... after all, don't components re-render when their props change? What about context??

Here's the thing: when a component re-renders, **it also re-renders all of its descendants.**

Let's look at an example:

Code Playground

```
import React from 'react';

function App() {
  return (
    <>
      <Counter />
      <footer>
        <p>Copyright 2022 Big Count Inc.</p>
      </footer>
    </>
  );
}

function Counter() {
  const [count, setCount] = React.useState(0);

  return (
    <main>
      <BigCountNumber count={count} />
      <button onClick={() => setCount(count + 1)}>
        Increment
      </button>
    </main>
  );
}

function BigCountNumber({ count }) {
  return (
```

Result Console

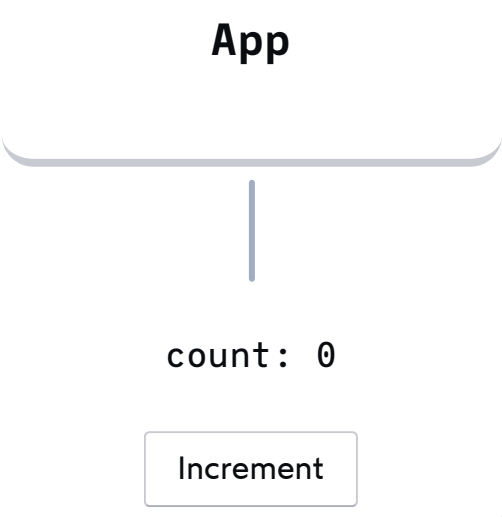
Unable to establish connection with the sandpack bundler. Make sure you are online or try again later. If the problem persists, please report it via email or submit an issue on GitHub.

In this example, we have 3 components: `App` at the top, which renders `Counter`, which renders `BigCountNumber`.

In React, every state variable is attached to a particular component instance. In this example, we have a single piece of state, `count`, which is associated with the `Counter` component.

Whenever this state changes, `Counter` re-renders. And because `BigCountNumber` is being rendered by `Counter`, it too will re-render.

Here's an *interactive graph* that shows this mechanic in action. Click the "Increment" button to trigger a state change:



BigCountNumber

Props: { count }

(The green flash signifies that a component is *re-rendering.*)

Alright, let's clear away *Big Misconception #1*: **The entire app re-renders whenever a state variable changes.**

I know some developers believe that every state change in React forces an application-wide render, but this isn't true. Re-renders only affect the component that owns the state + its descendants (if any). The `App` component, in this example, doesn't have to re-render when the `count` state variable changes.

Rather than memorize this as a rule, though, let's take a step back and see if we can figure out *why* it works this way.

React's "main job" is to keep the application UI in sync with the React state. The point of a re-render is to **figure out what needs to change**.

Let's consider the "Counter" example above. When the application first mounts, React renders all of our components and comes up with the following sketch for what the DOM should look like:

HTML

```
<main>
  <p>
    <span class="prefix">Count:</span>
    0
  </p>
```

```
  <button>
    Increment
  </button>
</main>
<footer>
  <p>Copyright 2022 Big Count Inc.</p>
</footer>
```

When the user clicks on the button, the `count` state variable flips from `0` to `1`. How does this affect the UI? Well, that's what we hope to learn from doing another render!

React re-runs the code for the `Counter` and `BigCountNumber` components, and we generate a new sketch of the DOM we want:

HTML

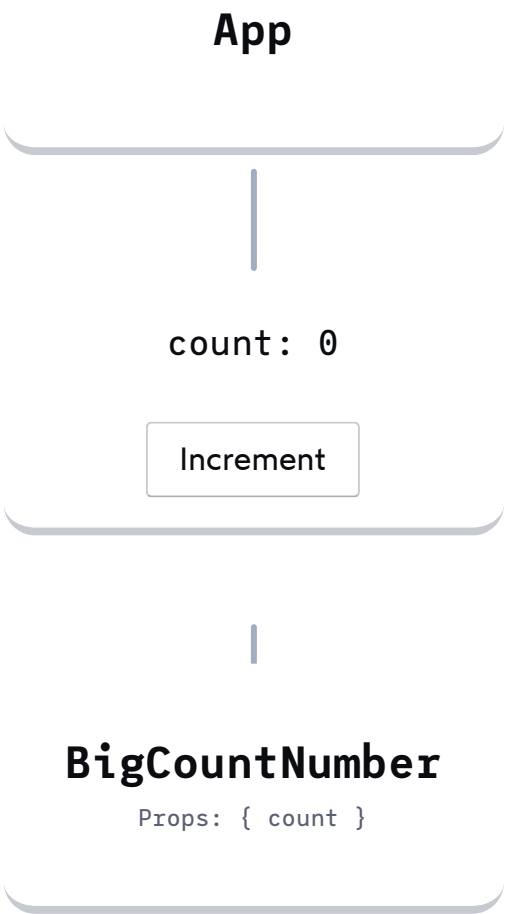
```
<main>
  <p>
    <span class="prefix">Count:</span>
    1
  </p>
  <button>
    Increment
  </button>
</main>
<footer>
  <p>Copyright 2022 Big Count Inc.</p>
</footer>
```

Each render is a snapshot, like a photo taken by a camera, that shows what the UI *should* look like, based on the current application state.

React plays a “find the differences” game to figure out what's changed between these two snapshots. In this case, it sees that our paragraph has a text node that changed from `0` to `1`, and so it edits the text node to match the snapshot. Satisfied that its work is done, React settles back and waits for the next state change.

This is the core React loop.

With this framing in mind, let's look again at our render graph:



Our `count` state is associated with the `Counter` component. Because data can't flow "up" in a React application, we know that this state change can't possibly

affect `<App />`. And so we don't need to re-render that component.

But we *do* need to re-render `Counter`'s child, `BigCountNumber`. This is the component that actually displays the `count` state. If we *don't* render it, we won't know that our paragraph's text node should change from `0` to `1`. We need to include this component in our sketch.

The point of a re-render is to figure out how a state change should affect the user interface. And so we need to re-render all potentially-affected components, to get an accurate snapshot.

It's not about the props

Alright, let's talk about ***Big Misconception #2***: **A component will re-render because its props change.**

Let's explore with an updated example.

In the code below, our "Counter" app has been given a brand new component, `Decoration`:

Code Playground

App.js Counter.js Decoration.js BigCountNumber.js

```
import React from 'react';

import Decoration from './Decoration';
import BigCountNumber from './BigCountNumber';

function Counter() {
  const [count, setCount] = React.useState(0);

  return (
    <main>
      <BigCountNumber count={count} />
      <button onClick={() => setCount(count + 1)}>
        Increment
```

```
        </button>

        {/* 👇 This fella is new 👇 */}
        <Decoration />
      </main>
    );
}

export default Counter;
```

Result Console

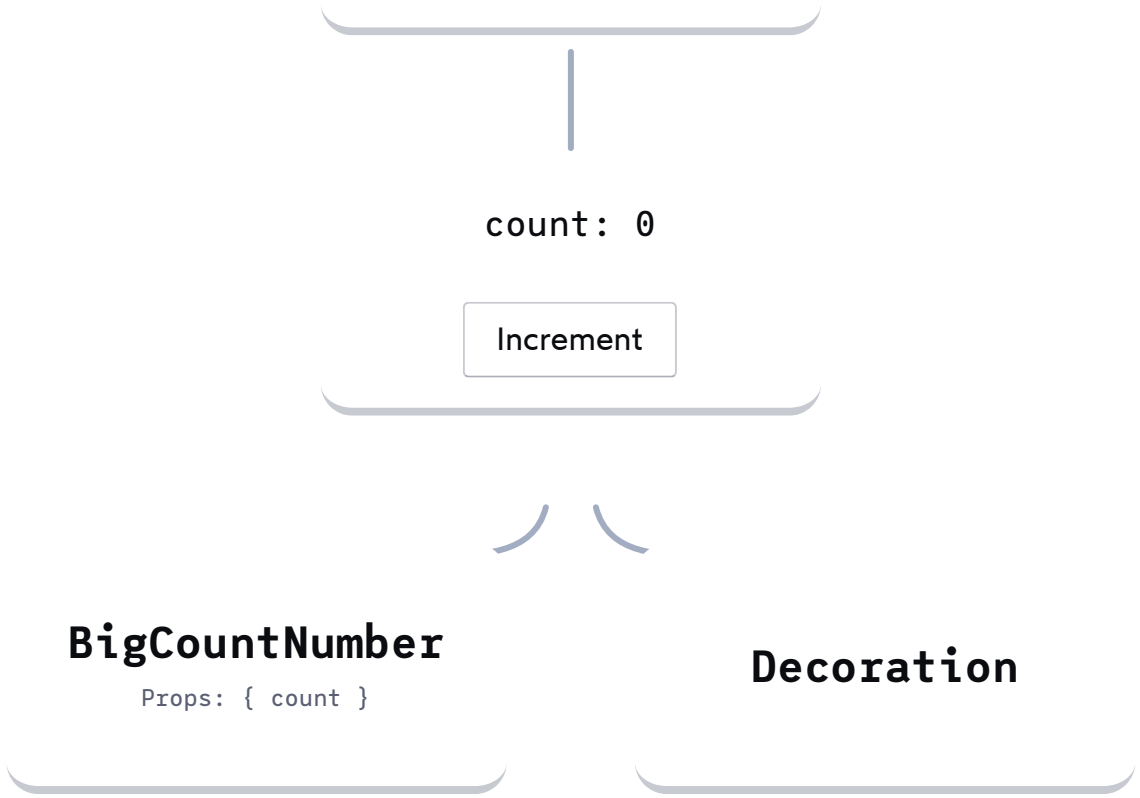
```
Unable to establish connection with the sandpack bundler. Make sure you are
online or try again later. If the problem persists, please report it via
email or submit an issue on GitHub.
```

(It was getting a bit crowded, having all of the components in a single big file, so I took the liberty of re-organizing. But the overall component structure is the same, aside from the new `Decoration` component.)

Our counter now has a cute lil' sailboat in the corner, rendered by the `Decoration` component. It doesn't depend on `count`, so it probably won't re-render when `count` changes, right?

Well, er, not quite.

App



When a component re-renders, it tries to re-render *all* descendants, regardless of whether they're being passed a particular state variable through props or not.

Now, this seems counter-intuitive... If we aren't passing `count` as a prop to `<Decoration>`, why would it need to re-render??

Here's the answer: it's hard for React to know, with 100% certainty, whether `<Decoration>` depends, directly or indirectly, on the `count` state variable.

In an ideal world, React components would always be “pure”. A pure component is one that **always** produces the same UI when given the same props.

In the real world, many of our components are impure. It's surprisingly easy to create an impure component:

JS

```
function CurrentTime() {
  const now = new Date();
```

```
  return (
    <p>It is currently {now.toString()}</p>
  );
}
```

This component will display a different value whenever it's rendered, since it relies on the current time!

A sneakier version of this problem has to do with refs. If we pass a ref as a prop, React won't be able to tell whether or not we've mutated it since the last render. And so it chooses to re-render, to be on the safe side.

React's #1 goal is to make sure that the UI that the user sees is kept "in sync" with the application state. And so, React will err on the side of *too many* renders. It doesn't want to risk showing the user a stale UI.

So, to go back to our misconception: **props have nothing to do with re-renders.** Our `<BigCountNumber>` component isn't re-rendering because the `count` prop changed.

When a component re-renders, because one of its state variables has been updated, that re-render will cascade all the way down the tree, in order for React to fill in the details of this new sketch, to capture a new snapshot.

This is the standard operating procedure, but there *is* a way to tweak it a bit.

Creating pure components

You might be familiar with `React.memo`, or the `React.PureComponent` class component. These two tools allow us to ***ignore certain re-render requests.***

Here's what it looks like:

JS

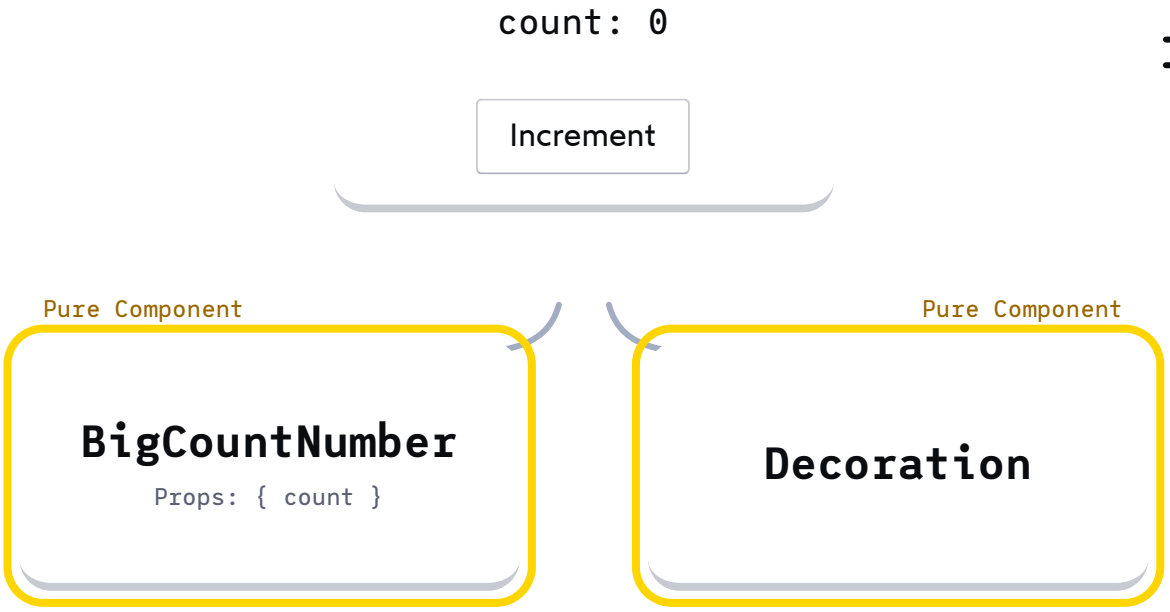
```
function Decoration() {
  return (
    <div className="decoration">
      ⛵
    </div>
  );
}

export default React.memo(Decoration);
```

By wrapping our `Decoration` component with `React.memo`, we're telling React "Hey, I *know* that this component is pure. You don't need to re-render it unless its props change."

This uses a technique known as ***memoization***.

It's missing the R, but we can sorta think of it as "memorization". The idea is that React will remember the previous snapshot. If none of the props have changed, React will re-use that stale snapshot rather than going through the trouble of generating a brand new one.

Let's suppose I wrap both `BigCountNumber` and `Decoration` with the `React.memo` helper. Here's how this would affect the re-renders:



When `count` changes, we re-render `Counter`, and React will try to render both descendant components.

Because `BigCountNumber` takes `count` as a prop, and because that prop has changed, `BigCountNumber` is re-rendered. But because `Decoration`'s props haven't changed (on account of it not having any), the *original* snapshot is used instead.

I like to pretend that `React.memo` is a bit like a lazy photographer. If you ask it to take 5 photos of the exact same thing, it'll take 1 photo and give you 5 copies of it. The photographer will only snap a new picture when your instructions change.

Here's a live-code version, if you'd like to poke at it yourself. Each memoized component has a `console.info` call added, so you can see in the console exactly when each component renders:

Code Playground

App.js Counter.js Decoration.js BigCountNumber.js

```
import React from 'react';

function Decoration() {
  console.info('Decoration render');

  return (
    <div className="decoration">
      ⛵
```

```
      </div>
    );
}

export default React.memo(Decoration);
```

Result Console

```
Unable to establish connection with the sandpack bundler. Make sure you are
online or try again later. If the problem persists, please report it via
email or submit an issue on GitHub.
```

You might be wondering: **why isn't this the default behaviour??** Isn't this what we want, most of the time? Surely we'd improve performance if we skipped rendering components that don't need to be rendered?

I think as developers, we tend to overestimate how expensive re-renders are. In the case of our `Decoration` component, re-renders are lightning quick.

If a component has a bunch of props and not a lot of descendants, it can actually be ***slower*** to check if any of the props have changed compared to re-rendering the component.*

And so, it would be counter-productive to memoize every single component we create. React is designed to capture these snapshots really quickly! But in specific circumstances, for components with a lot of descendants *or* components that do a ton of internal work, this helper can help quite a bit.

> **This could change in the future!**
>
> The React team is actively investigating whether it's possible to "auto-memoize" code during the compile step. It's still in the research phase, but early experimentation appears promising.
>
> For more information, check out this talk by Xuan Huang called **"React without memo"**.

What about context?

We haven't talked at all about context yet, but fortunately, it doesn't complicate this stuff too much.

By default, all descendants of a component will re-render if that component's state changes. And so, it doesn't really change anything if we provide that state to all descendants via context; either way, those components are gonna re-render!

Now in terms of *pure* components, context is sorta like "invisible props", or maybe "internal props".

Let's look at an example. Here we have a pure component that consumes a `UserContext` context:

JS

```
const GreetUser = React.memo(() => {
  const user = React.useContext(UserContext);

  if (!user) {
    return "Hi there!";
  }

  return `Hello ${user.name}!`;
});
```

In this example, `GreetUser` is a pure component with no props, but it has an "invisible" or "internal" dependency: the `user` being stored in React state, and passed around through context.

If that `user` state variable changes, a re-render will occur, and `GreetUser` will generate a new snapshot, rather than relying on a stale picture. React can tell that this component is consuming this particular context, and so it treats it as if it was a prop.

It's more-or-less equivalent to this:

JS

```
const GreetUser = React.memo(({ user }) => {
  if (!user) {
    return "Hi there!";
  }

  return `Hello ${user.name}!`;
});
```

Play with a live example:

Code Playground

```
import React from 'react';

const UserContext = React.createContext();

function UserProvider({ children }) {
  const [user, setUser] = React.useState(null);

  React.useEffect(() => {
    // Pretend that this is a network request,
    // fetching user data from the backend.
    window.setTimeout(() => {
      setUser({ name: 'Kiara' });
```

```
    }, 1000)
  }, [])

  return (
    <UserContext.Provider value={user}>
      {children}
    </UserContext.Provider>
  );
}

function App() {
  return (
    <UserProvider>
      <GreetUser />
    </UserProvider>
  );
```

Result Console

Unable to establish connection with the sandpack bundler. Make sure you are online or try again later. If the problem persists, please report it via email or submit an issue on GitHub.

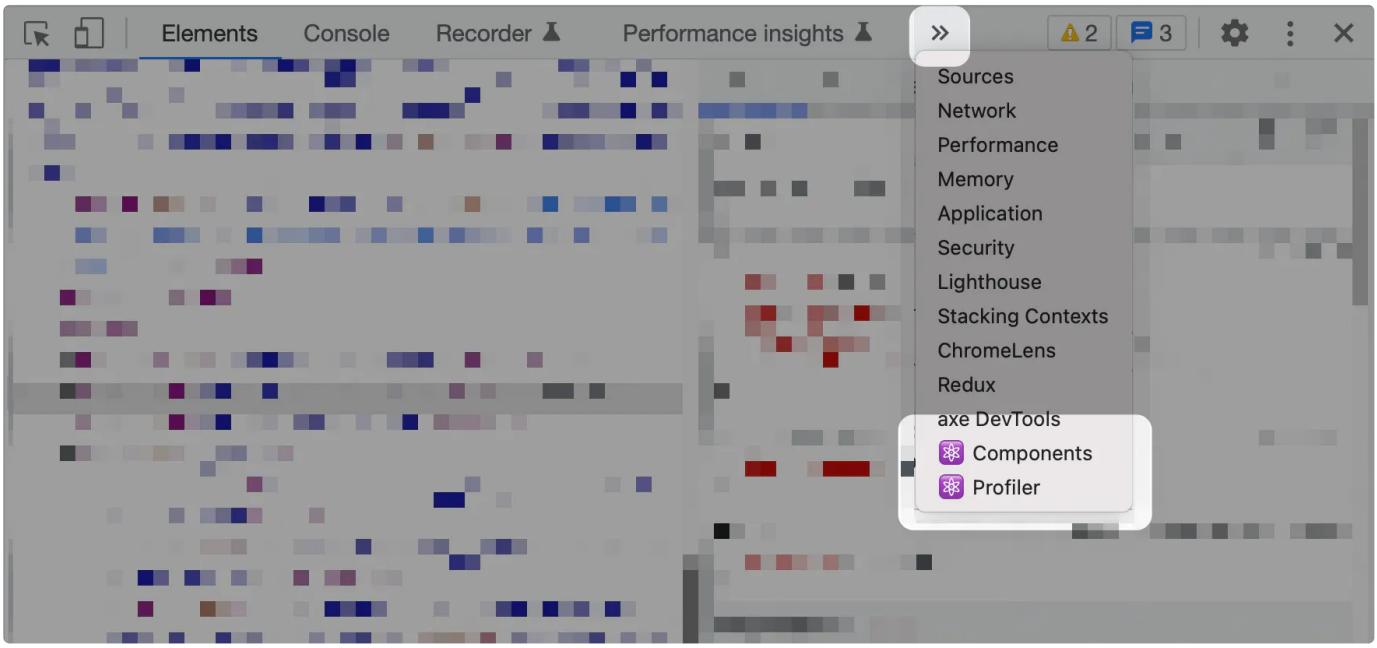
Note that this only happens if the pure component ***consumes the context*** with the `React.useContext` hook. You don't have to worry about context breaking a bunch of pure components that don't try to consume it.

Profiling with the React Devtools

If you've worked with React for a while, you've likely had the frustrating experience of trying to figure out *why* a particular component is re-rendering. In a real-world situation, it often isn't obvious at all! Fortunately, the React Devtools can help.

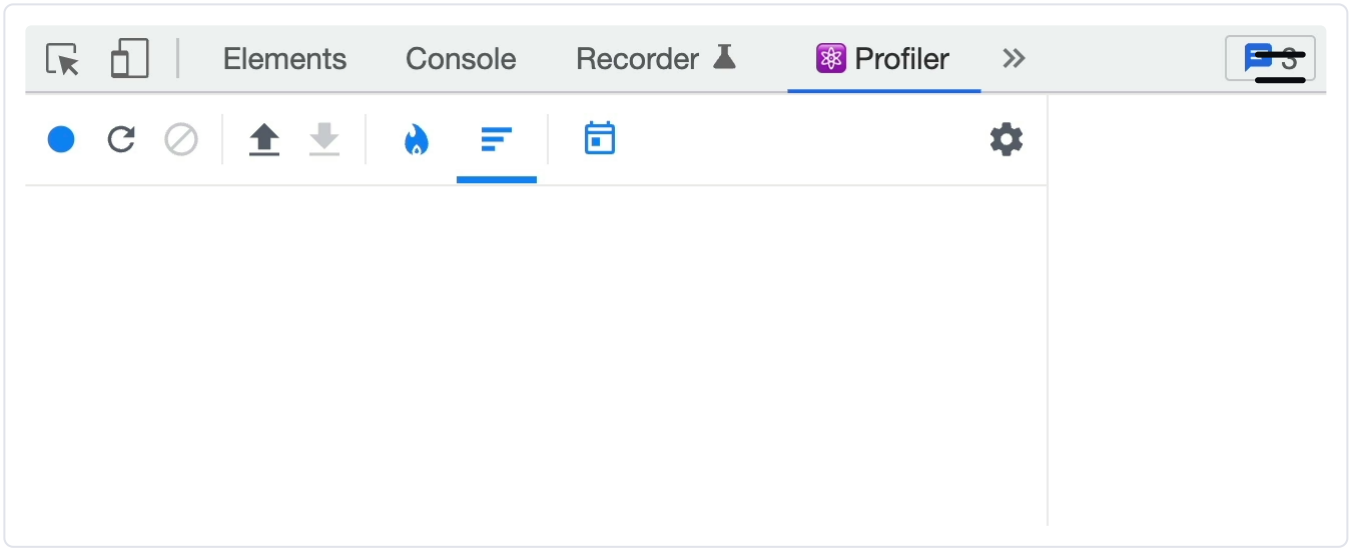
First, you'll need to download the React Devtools browser extension. It's currently available for Chrome and Firefox. For the purposes of this tutorial, I'll assume you're using Chrome, though the instructions won't vary much.

Pop open the devtools with `Ctrl + Alt + I` (or `⌘ + Option + I` on MacOS). You should see two new tabs appear:



We're interested in the "Profiler". Select that tab.

Click the little gear icon, and enable the option labeled *"Record why each component rendered while profiling"*:



The general flow looks like this:

1. Start recording by hitting the little blue “record” circle.
2. Perform some actions in your application.
3. Stop recording.
4. View the recorded snapshots to learn more about what happened.

Each render is captured as a separate snapshot, and you can browse through them using the arrows. The information about why a component rendered is available in the sidebar:

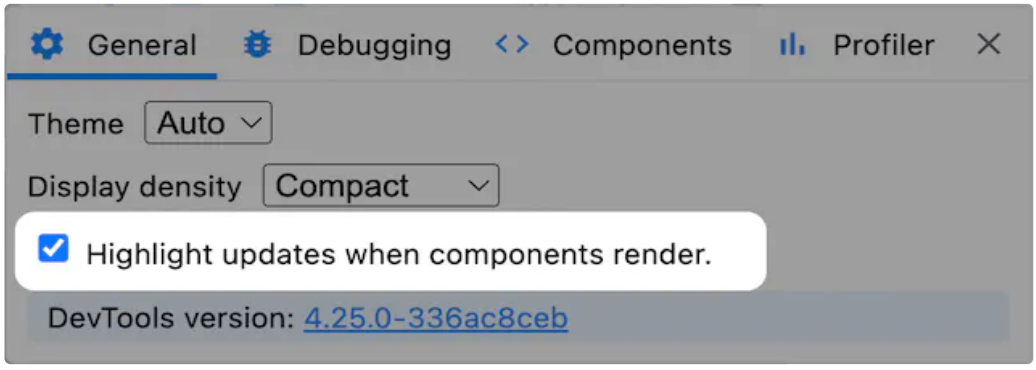
By clicking through to the component you're interested in, you can see exactly why a particular component re-rendered. In the case of a pure component, it will let us know which prop(s) are responsible for this update.

I don't personally use this tool often, but when I do, it's a lifesaver!

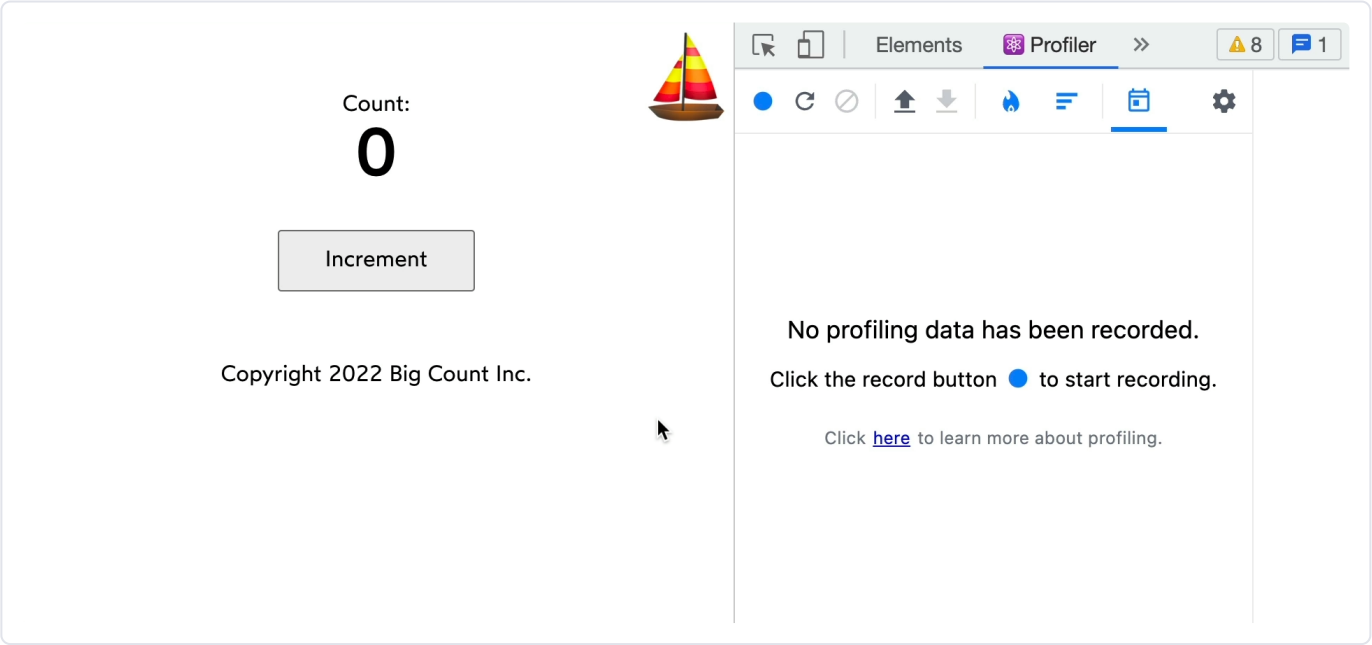
Highlighting re-renders

One more little trick: the React profiler has an option where you can highlight components that re-render.

Here's the setting in question:



With this setting enabled, you should see green rectangles flash around components that re-render:



This can help us understand exactly how far-reaching state updates are, and test whether our pure components are successfully avoiding re-rendering!

Going deeper

One of the things that you'll notice when you start using the profiler: sometimes, pure components re-render even when nothing *appears* to have changed!

One of the subtle mind-bending things about React is that components are JavaScript functions. When we render a component, we're calling the function.*

This means that anything defined inside a React component is re-created on every single render.

As a quick example, consider this:

JS

```
function App() {
  const dog = {
    name: 'Spot',
    breed: 'Jack Russell Terrier'
  };

  return (
    <DogProfile dog={dog} />
  );
}
```

Every single time we render this `App` component, we're generating a brand new object. This can wreck havoc on our pure components; this `DogProfile` child is going to re-render whether or not we wrap it with `React.memo`!

In a couple weeks, I'll be publishing a “Part II” to this blog post. We'll dig into two famously-inscrutable React hooks, `useMemo` and `useCallback`. And we'll see how to use them to solve this problem.

I also have a confession to make: these tutorials have been plucked straight from my upcoming course, **“The Joy of React”**.

I've been building with React for over 7 years now, and I've learned a lot about how to use it effectively. I absolutely love working with React; I've tried just about every front-end framework under the sun, and nothing makes me feel as productive as React.

In “The Joy of React”, we'll build a mental model for how React really works, digging into concepts like we have in this tutorial. Unlike the posts on this blog, however, my courses use a “multi-modality” approach, mixing written content like this with video content, exercises, interactive explorables, and even some minigames!

I'm looking forward to pre-launching my course in the next couple months. If you're interested, you can sign up for updates on the course homepage. 💖

Bonus: Performance tips

Performance optimization in React is a huge topic, and I could easily write several blog posts about it. Hopefully, this tutorial has helped build a solid foundation upon which you can learn about React performance!

That said, I'll share a few quick tips I've learned about React performance optimization:

- The React Profiler shows the number of milliseconds that a render took, but **this number isn't trustworthy.** We generally profile things in "development mode", and React is ***much, much faster*** in "production mode". To ***truly*** understand how performant your application is, you should measure using the "Performance" tab against the deployed production application. This will show you real-world numbers not just for re-renders, but also the layout/paint changes.
- I strongly recommend testing your applications on lower-end hardware, to see what the 90th-percentile experience is like. It'll depend on the product you're building, but for this blog, I periodically test things on a Xiaomi Redmi 8, a budget smartphone popular in India a few years ago. I shared this experience on Twitter.
- Lighthouse performance scores are ***not*** an accurate reflection of true user experience. I trust the qualitative experience of using the application much more than the stats shown by any automated tool.
- I gave a talk a few years ago at React Europe all about performance in React! It focuses more on the "post-load" experience, an area lots of developers neglect. You can watch it on YouTube.
-

Don't over-optimize! It's tempting, when learning about the React profiler, to go on an optimization spree, with the goal of reducing the # of renders as much as possible... but honestly, React is already very optimized out of the box. These tools are best used *in response to a performance problem,* if things start feeling a bit sluggish.

LAST UPDATED

August 16th, 2022

HITS