Interpreting the Flamegraph

Let's take a look at the flow of the Flamegraph for our bank app.

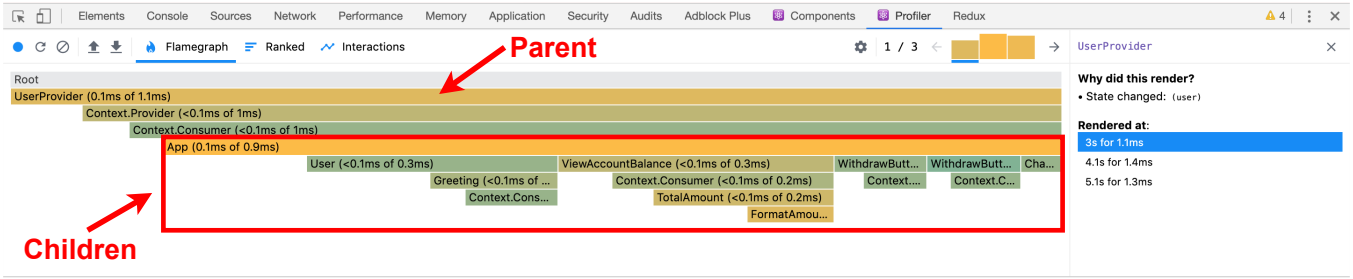
WE'LL COVER THE FOLLOWING

- Components Rendered Needlessly
- Addition of `PureComponent`

Components Rendered Needlessly

First, let's consider what's likely to be the root of the problem here. By default, whenever a `Provider` has its `value` changed, every child component is forced to re-render. That's how the `Consumer` gets the latest values from the `context` object and stays in sync.

The problem here is that every component apart from `Root` is a child of `Provider` and they all get re-rendered needlessly!

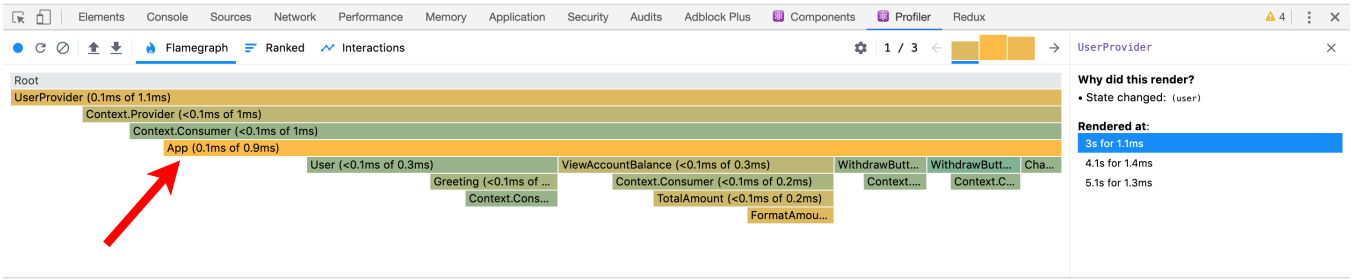


All children components re-rendered with a change in the Provider's value.

So, what can we do about this?

Some of the child components don't need to be re-rendered as they are not directly connected with the change.

Let's consider the first child component, `App`.



The `App` component doesn't receive any `prop` and it only manages the state value, `showBalance`.

```
class App extends Component {
  state = {
    showBalance: false
  }
  displayBalance = () => {
    this.setState({ showBalance: true })
  }
  render () {
    const { showBalance } = this.state
    ...
  }
}
```

Addition of `PureComponent`

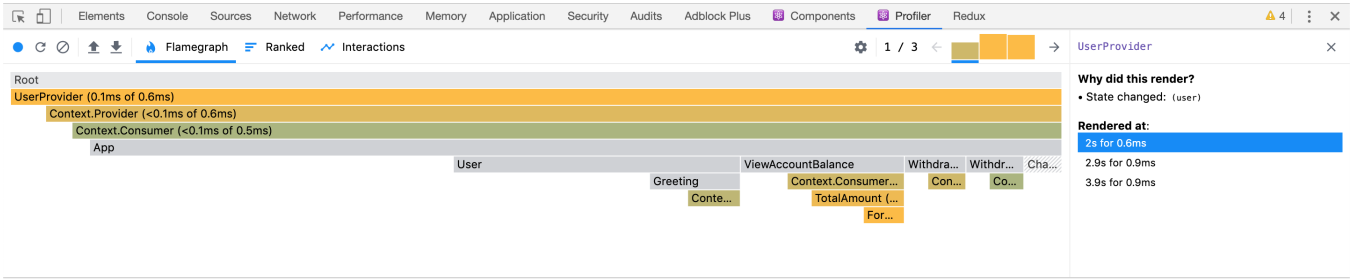
The app component isn't directly connected with the change, and it's pointless to re-render this component.

Let's fix this by making it a `PureComponent`.

```
// before
class App extends Component {
  state = {
    showBalance: false
  }
 ...
}
// after
class App extends PureComponent {
  state = {
    showBalance: false
  }
 ...
}
```

Having made `App` a `PureComponent`, did we make any decent progress?

Well, take a look at the new flamechart generated after that simple one-line change.



Can you see that?

A lot of App's children aren't re-rendered needlessly, and we have a saner flamegraph now.

Great!

In the next lesson, we'll discuss the different interactions of the profiled session.