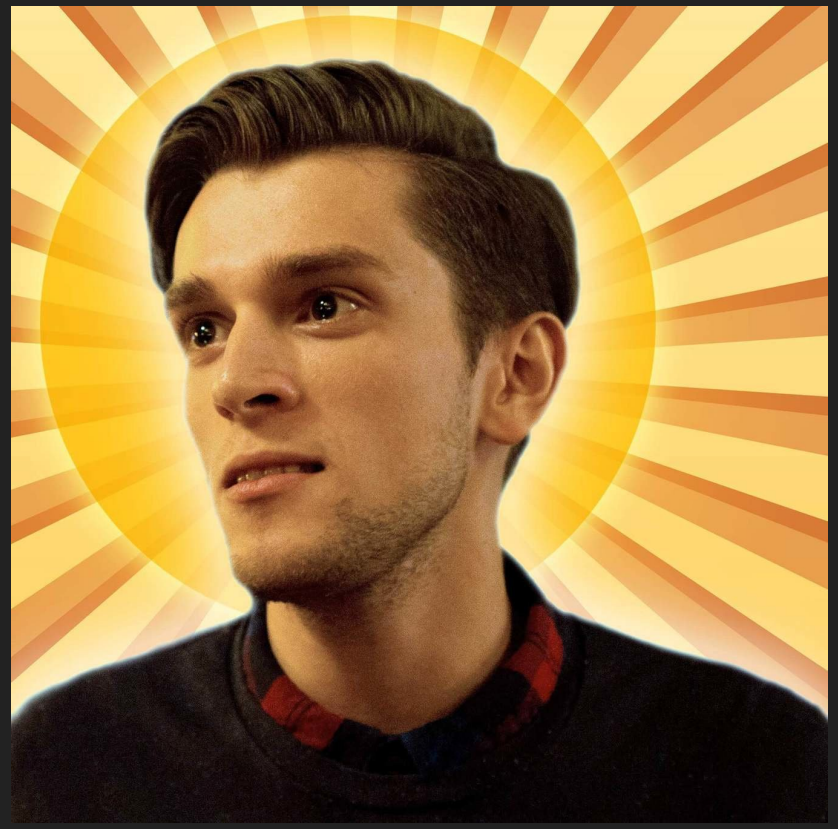
Recoil Into The Future

Understand state management shortfalls. Understand component tree. Brace for the future.

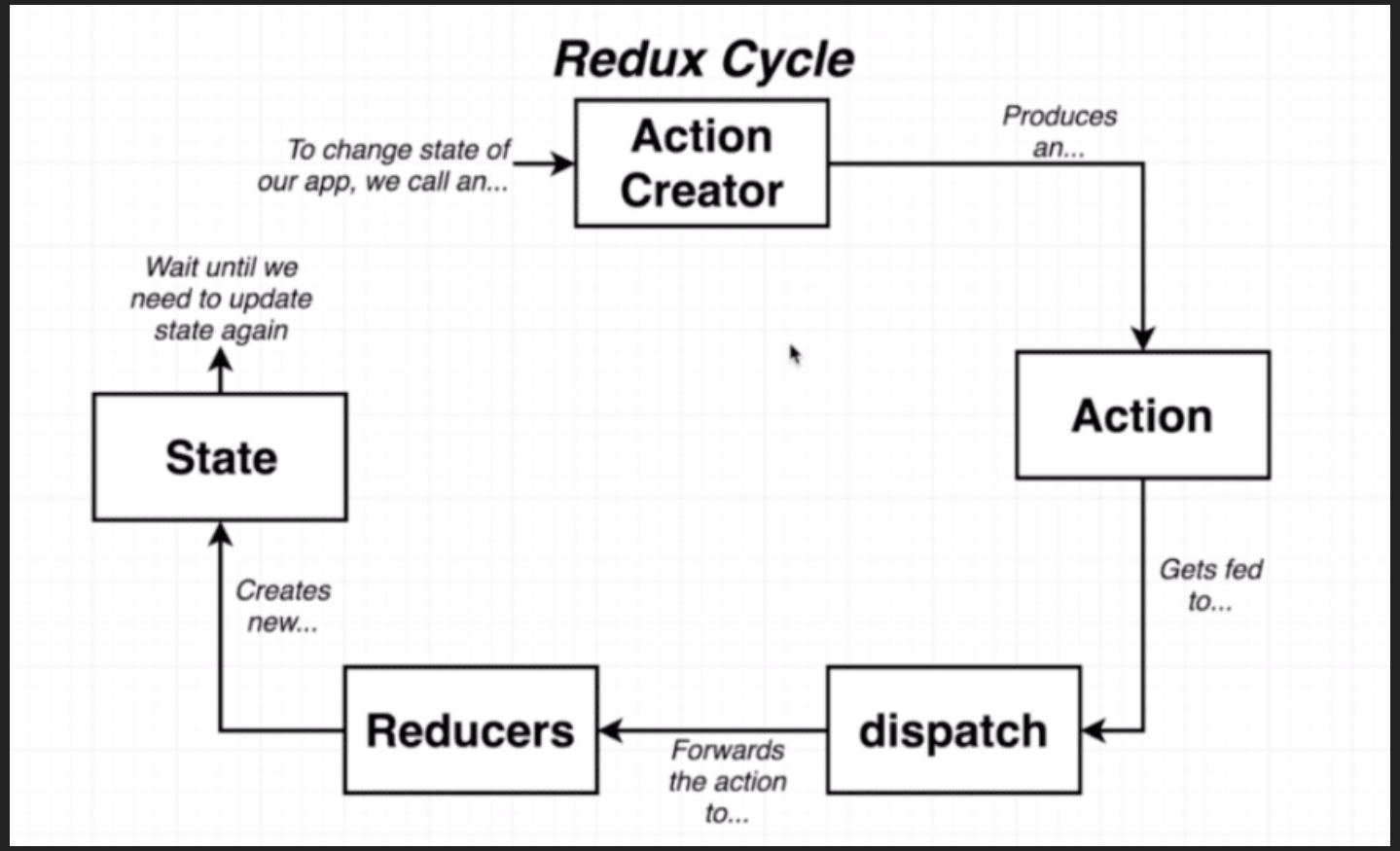
The Story of React State Management

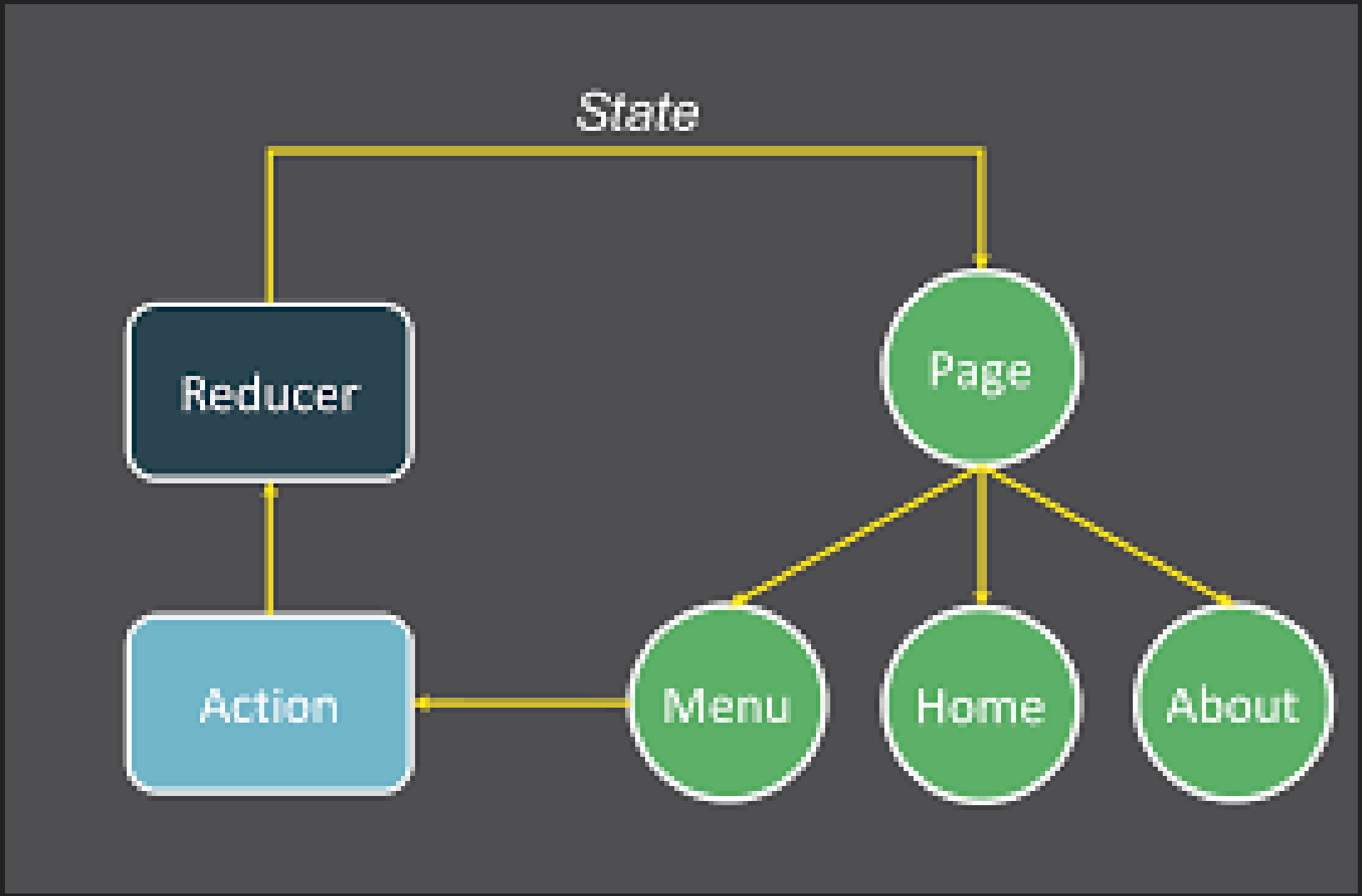
...according to me

On the first day, the gods gave us Redux...



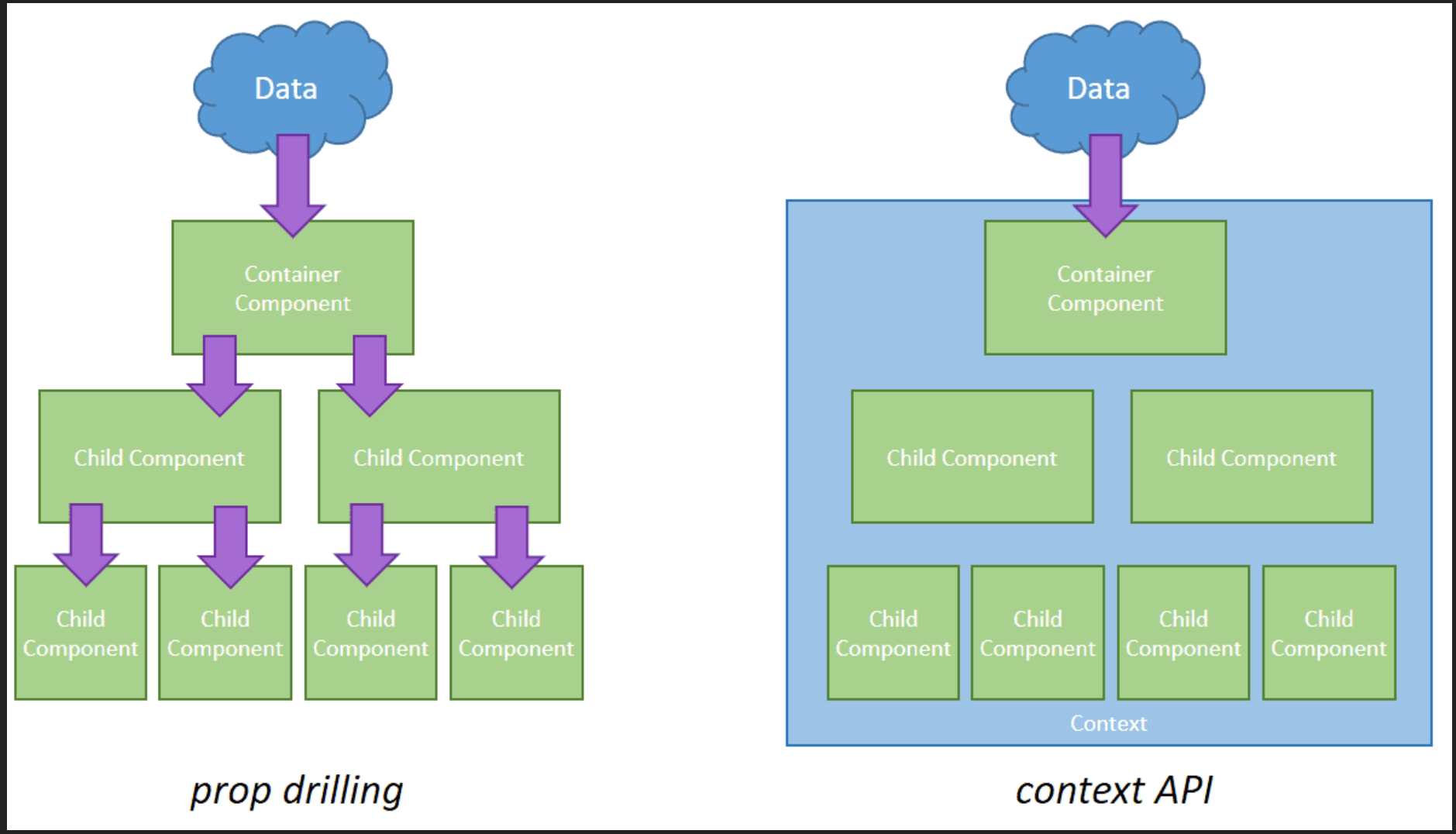
...but it's bloated af...



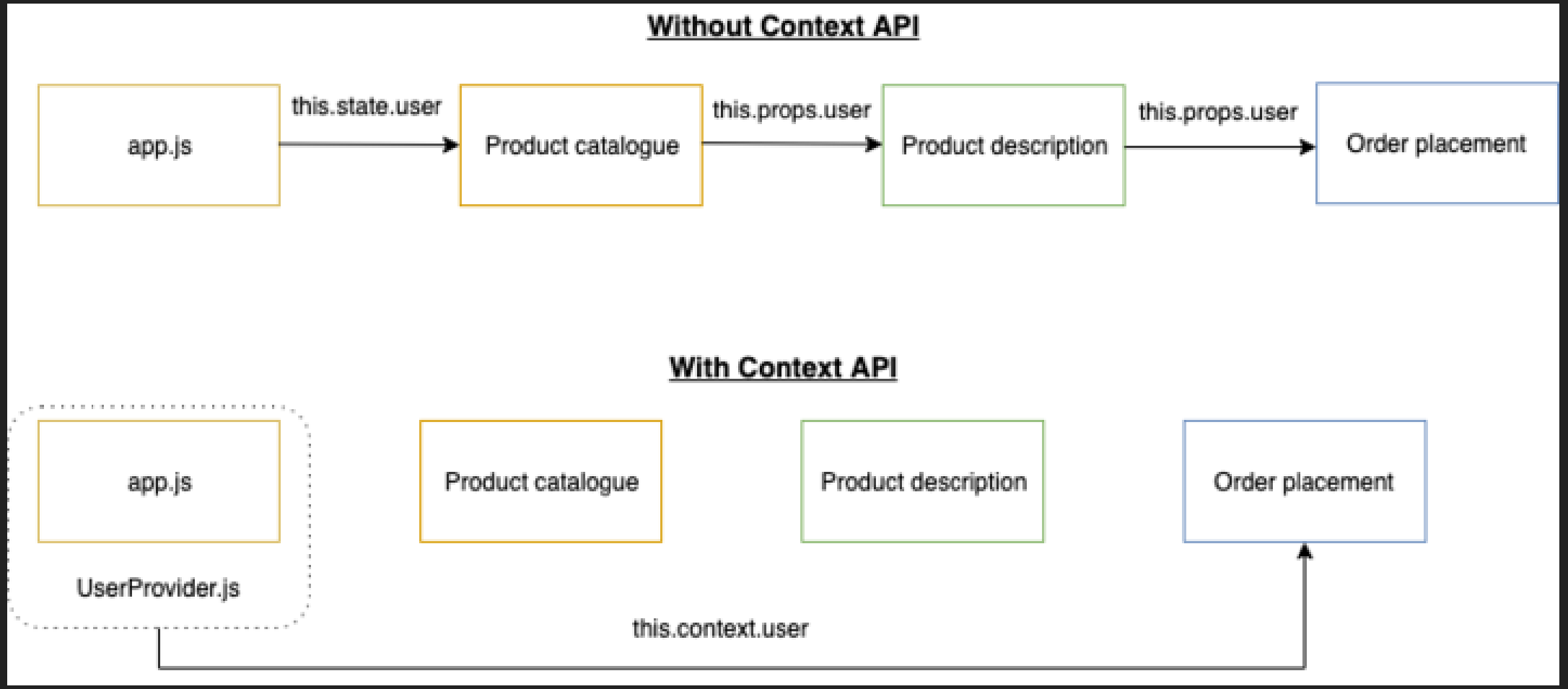


...and it's not going to help us solve prop drilling.

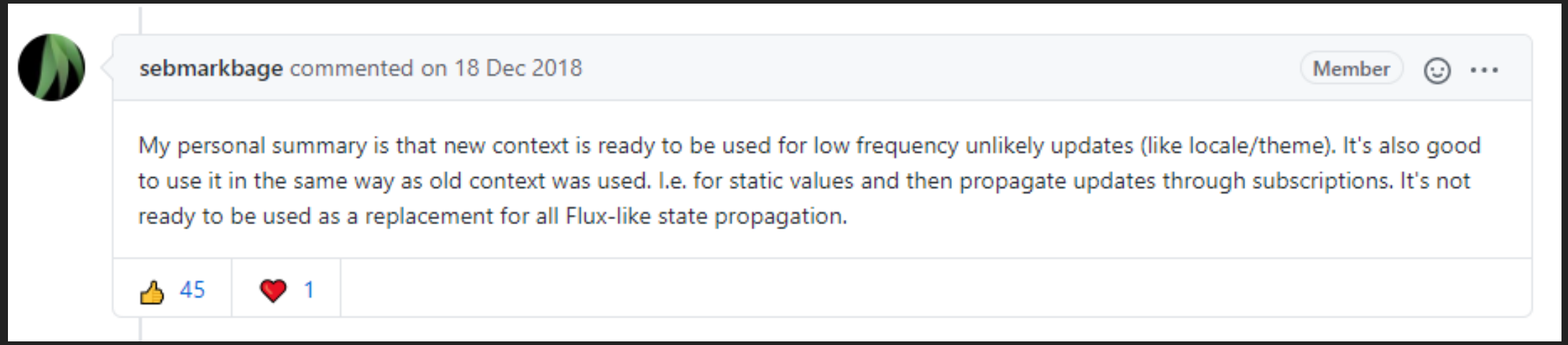
Then we were graced with React Context...



...we were given a provider/consumer relationship to stop pointless re-renders...



...but the gods were all like "yeah nah don't use for that complex stuff peeps"



sebmarkbage commented on 18 Dec 2018 Member

My personal summary is that new context is ready to be used for low frequency unlikely updates (like locale/theme). It's also good to use it in the same way as old context was used. I.e. for static values and then propagate updates through subscriptions. It's not ready to be used as a replacement for all Flux-like state propagation.

45 1

In summary:

- Redux is too bloated and boilerplatey
- Redux requires more libraries just to make the experience tolerable
- Redux won't make simple state sharing easy
- Redux won't solve unnecessary re-renders
- React Context can provide a consumer/relationship but shouldn't be used for complex structures or frequently updated state
- React Context only works if you know the amount of providers you're going to need
- React Hooks can be used but you're going to end up prop drilling the hell of out your app

What is Recoil?

Recoil is a lightweight state management library (yes, another one). Recoil has been released and maintained by Facebook and is regarded as experimental.

Recoil is the first state management library released by Facebook since Redux which is a big deal. Along with that, Recoil only makes use of React's features under-the-hood while libraries like mobx and Redux use external features.

This is critical with the introduction of concurrent mode / suspense coming out of beta soon.

What problems is Recoil solving?

- Easy to use API
- Simple hooks integration
- Control over exact state you want to use
- Flexible shared state
- Derived state/queries

Atoms are tiny units of state. They are regarded as the source of truth for our application.

```
// atoms.js

import { atom } from "recoil";

export const countState = atom({
        key: "countState",
        default: 0
)}
```

Atoms are tiny units of state. They are regarded as the source of truth for our application.

```
// atoms.js

import { atom } from "recoil";

export const countState = atom({
        key: "countState",
        default: 0
)}
```

Atoms are tiny units of state. They are regarded as the source of truth for our application.

```
// atoms.js

import { atom } from "recoil";

export const countState = atom({
        key: "countState",
        default: 0
)}
```

Atoms are tiny units of state. They are regarded as the source of truth for our application.

```
// atoms.js

import { atom } from "recoil";

export const countState = atom({
        key: "countState",
        default: 0
)}
```

Atoms are tiny units of state. They are regarded as the source of truth for our application.

```
// atoms.js

import { atom } from "recoil";

export const countState = atom({
        key: "countState",
        default: 0
)}
```

Selectors are pure functions that either accept atoms or other selectors as their input. Whenever an atom or selector that a selector subscribes to is updated, it will trigger a recalculation for the current selector.

```
// selectors.js

import { selector } from "recoil";
import { countState } from "./atoms";

export const isMultipleOfTwoState = selector({
        key: "isMultipleOfTwoState",
        get: ({ get }) => {
        const count = get(countState);
        return 0 === count % 2;
    }
)}
```

Selectors are pure functions that either accept atoms or other selectors as their input. Whenever an atom or selector that a selector subscribes to is updated, it will trigger a recalculation for the current selector.

```
// selectors.js

import { selector } from "recoil";
import { countState } from "./atoms";

export const isMultipleOfTwoState = selector({
        key: "isMultipleOfTwoState",
        get: ({ get }) => {
        const count = get(countState);
        return 0 === count % 2;
    }
)}
```

Selectors are pure functions that either accept atoms or other selectors as their input. Whenever an atom or selector that a selector subscribes to is updated, it will trigger a recalculation for the current selector.

```
// selectors.js

import { selector } from "recoil";
import { countState } from "./atoms";

export const isMultipleOfTwoState = selector({
        key: "isMultipleOfTwoState",
        get: ({ get }) => {
        const count = get(countState);
        return 0 === count % 2;
    }
)}
```

Selectors are pure functions that either accept atoms or other selectors as their input. Whenever an atom or selector that a selector subscribes to is updated, it will trigger a recalculation for the current selector.

```
// selectors.js

import { selector } from "recoil";
import { countState } from "./atoms";

export const isMultipleOfTwoState = selector({
        key: "isMultipleOfTwoState",
        get: ({ get }) => {
        const count = get(countState);
        return 0 === count % 2;
    }
)}
```

Selectors are pure functions that either accept atoms or other selectors as their input. Whenever an atom or selector that a selector subscribes to is updated, it will trigger a recalculation for the current selector.

```
// selectors.js

import { selector } from "recoil";
import { countState } from "./atoms";

export const isMultipleOfTwoState = selector({
        key: "isMultipleOfTwoState",
        get: ({ get }) => {
        const count = get(countState);
        return 0 === count % 2;
    }
)}
```

Selectors are pure functions that either accept atoms or other selectors as their input. Whenever an atom or selector that a selector subscribes to is updated, it will trigger a recalculation for the current selector.

```
// selectors.js

import { selector } from "recoil";
import { countState } from "./atoms";

export const isMultipleOfTwoState = selector({
        key: "isMultipleOfTwoState",
        get: ({ get }) => {
        const count = get(countState);
        return 0 === count % 2;
    }
)}
```

Our component can make use of hooks to access the shared state.

```
// ShoppingList.js

import { useRecoilState, useRecoilValue } from "recoil";
import { countState } from "../store/atoms";
import { isMultipleOfTwoState } from "../store/selectors";

export default function Counter() {
  const [count, setCount] = useRecoilState(countState);
  const isMultipleOfTwo = useRecoilValue(isMultipleOfTwoState);

  const increment = () => {
        setCount(count + 1);
  }

  const decrement = () => {
        setCount(count - 1);
  }

  return (
        ...
  );
}
```

Our component can make use of hooks to access the shared state.

```
// ShoppingList.js

import { useRecoilState, useRecoilValue } from "recoil";
import { countState } from "../store/atoms";
import { isMultipleOfTwoState } from "../store/selectors";

export default function Counter() {
  const [count, setCount] = useRecoilState(countState);
  const isMultipleOfTwo = useRecoilValue(isMultipleOfTwoState);

  const increment = () => {
        setCount(count + 1);
  }

  const decrement = () => {
        setCount(count - 1);
  }

  return (
        ...
  );
}
```

Our component can make use of hooks to access the shared state.

```
// ShoppingList.js

import { useRecoilState, useRecoilValue } from "recoil";
import { countState } from "../store/atoms";
import { isMultipleOfTwoState } from "../store/selectors";

export default function Counter() {
  const [count, setCount] = useRecoilState(countState);
  const isMultipleOfTwo = useRecoilValue(isMultipleOfTwoState);

  const increment = () => {
        setCount(count + 1);
  }

  const decrement = () => {
        setCount(count - 1);
  }

  return (
        ...
  );
}
```

Our component can make use of hooks to access the shared state.

```
// ShoppingList.js

import { useRecoilState, useRecoilValue } from "recoil";
import { countState } from "../store/atoms";
import { isMultipleOfTwoState } from "../store/selectors";

export default function Counter() {
  const [count, setCount] = useRecoilState(countState);
  const isMultipleOfTwo = useRecoilValue(isMultipleOfTwoState);

  const increment = () => {
        setCount(count + 1);
  }

  const decrement = () => {
        setCount(count - 1);
  }

  return (
        ...
  );
}
```

Our component can make use of hooks to access the shared state.

```
// ShoppingList.js

import { useRecoilState, useRecoilValue } from "recoil";
import { countState } from "../store/atoms";
import { isMultipleOfTwoState } from "../store/selectors";

export default function Counter() {
  const [count, setCount] = useRecoilState(countState);
  const isMultipleOfTwo = useRecoilValue(isMultipleOfTwoState);

  const increment = () => {
        setCount(count + 1);
  }

  const decrement = () => {
        setCount(count - 1);
  }

  return (
        ...
  );
}
```