Fullstack Development

Data Fetching in React (SPA)

Method 1: `useEffect`

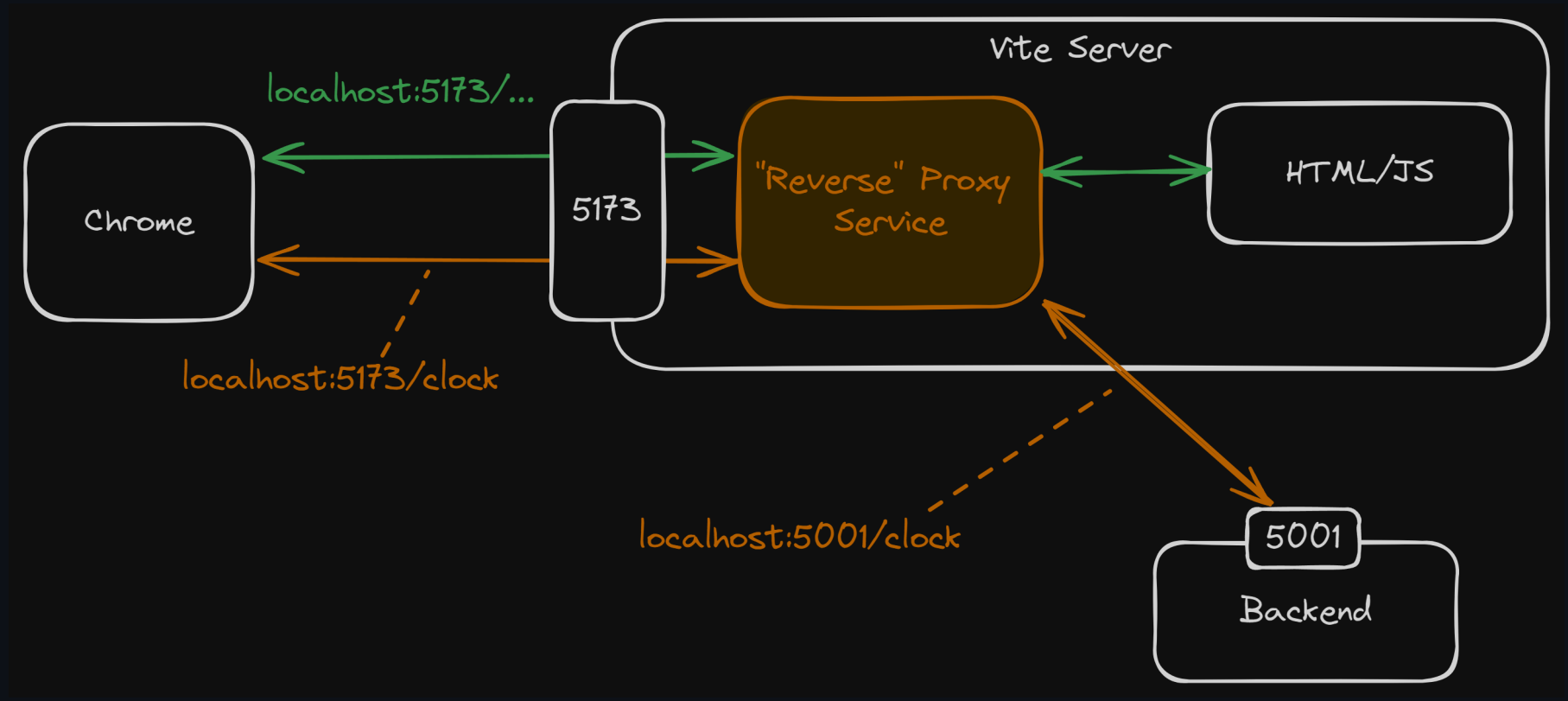
Setup

- `git clone -b useeffect https://github.com/fullstack-67/df-http.git df-http`
- `git checkout -t origin/useeffect`

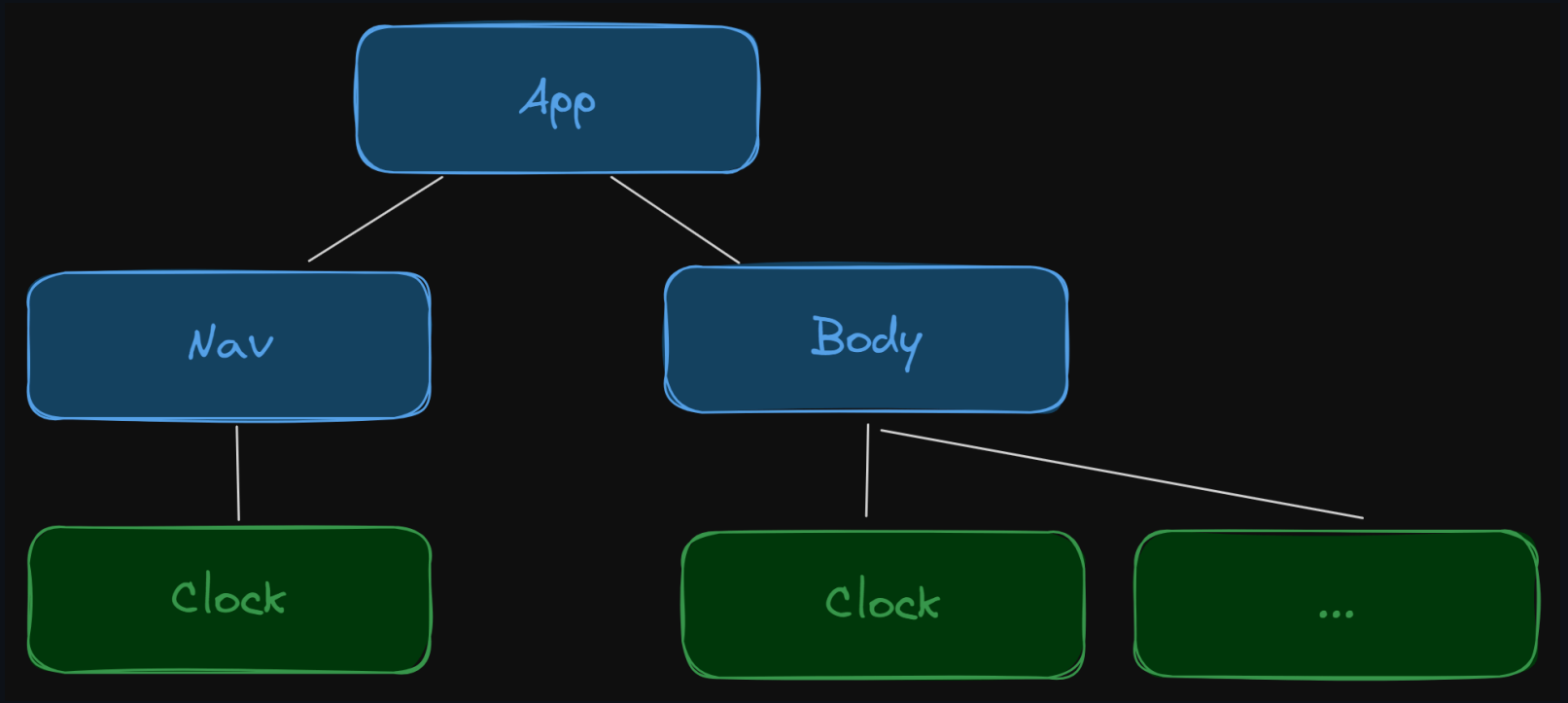
Backend / Frontend

- `cd backend` / `cd frontend`
- `pnpm i`
- `npm run dev`

Application architecture



Frontend component tree



./src/components/Clock.tsx

```tsx
const Clock: FC<Props> = () => {
  const [clock, setClock] = useState("");
  const refetch = () => {
    // Fetching logic
  };
  useEffect(() => {
    refetch();
  }, []);
  // return JSX
};
```

`useEffect`

- `Good`
  - No external library required
- `Bad`
  - Confusing to write
- `Comment`
  - States are all local.

Method 2: `useEffect` + Custom hook

Setup

- `git checkout -t origin/custom-hook`

./src/hooks/useClock.ts

```
function useClock() {
  const [clock, setClock] = useState("");
  const refetch = () => {
    // Fetching logic
  };
  useEffect(() => {
    refetch();
  }, []);

  return { clock, refetch };
}
```

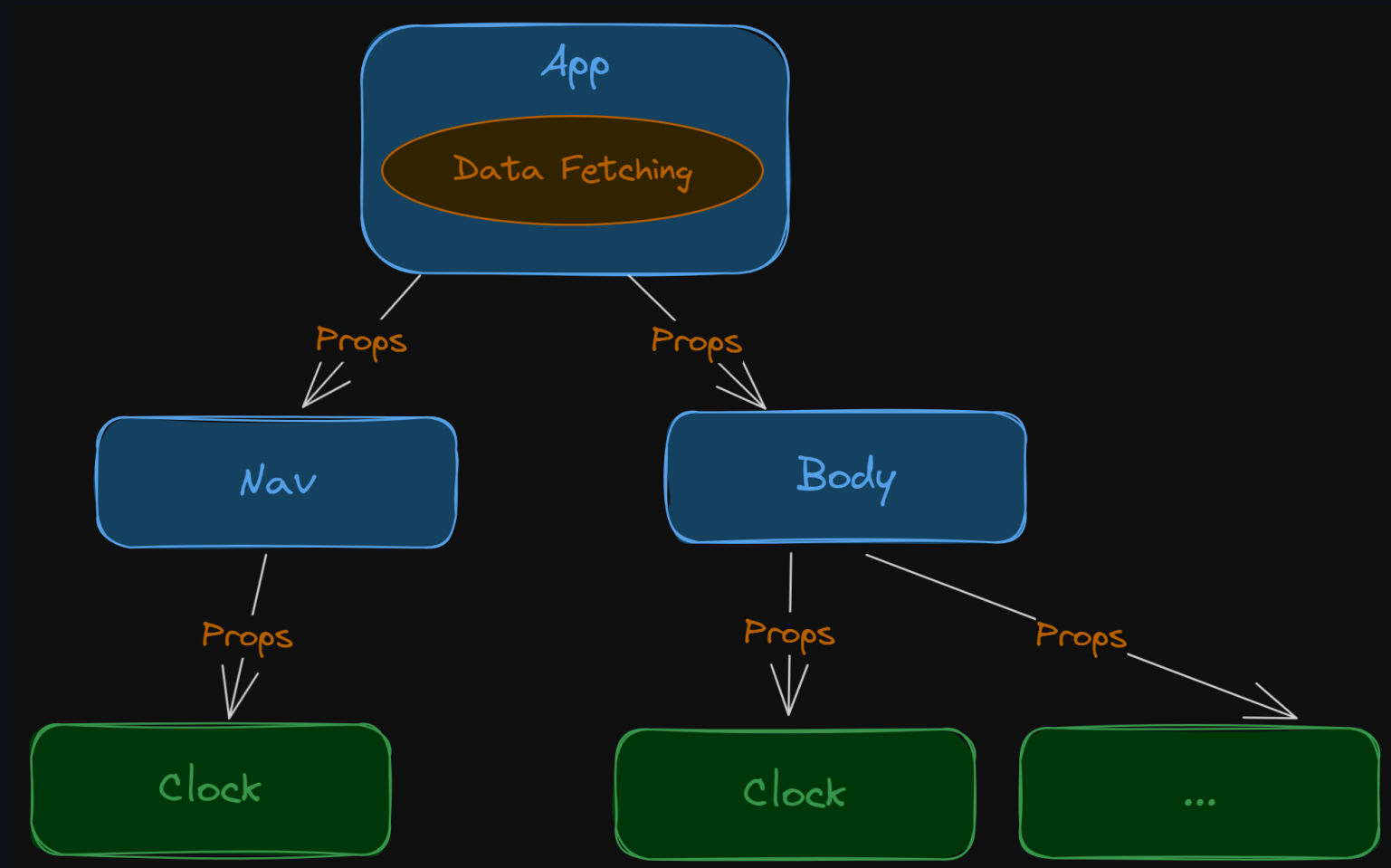
`useEffect` + Custom hook

- `Good`
  - Logic encapsulation
  - Cleaner components
- `Comment`
  - States are still all local.

Method 3: `useEffect` + Prop drilling

Prop drilling

Change



`useEffect` + Prop drilling

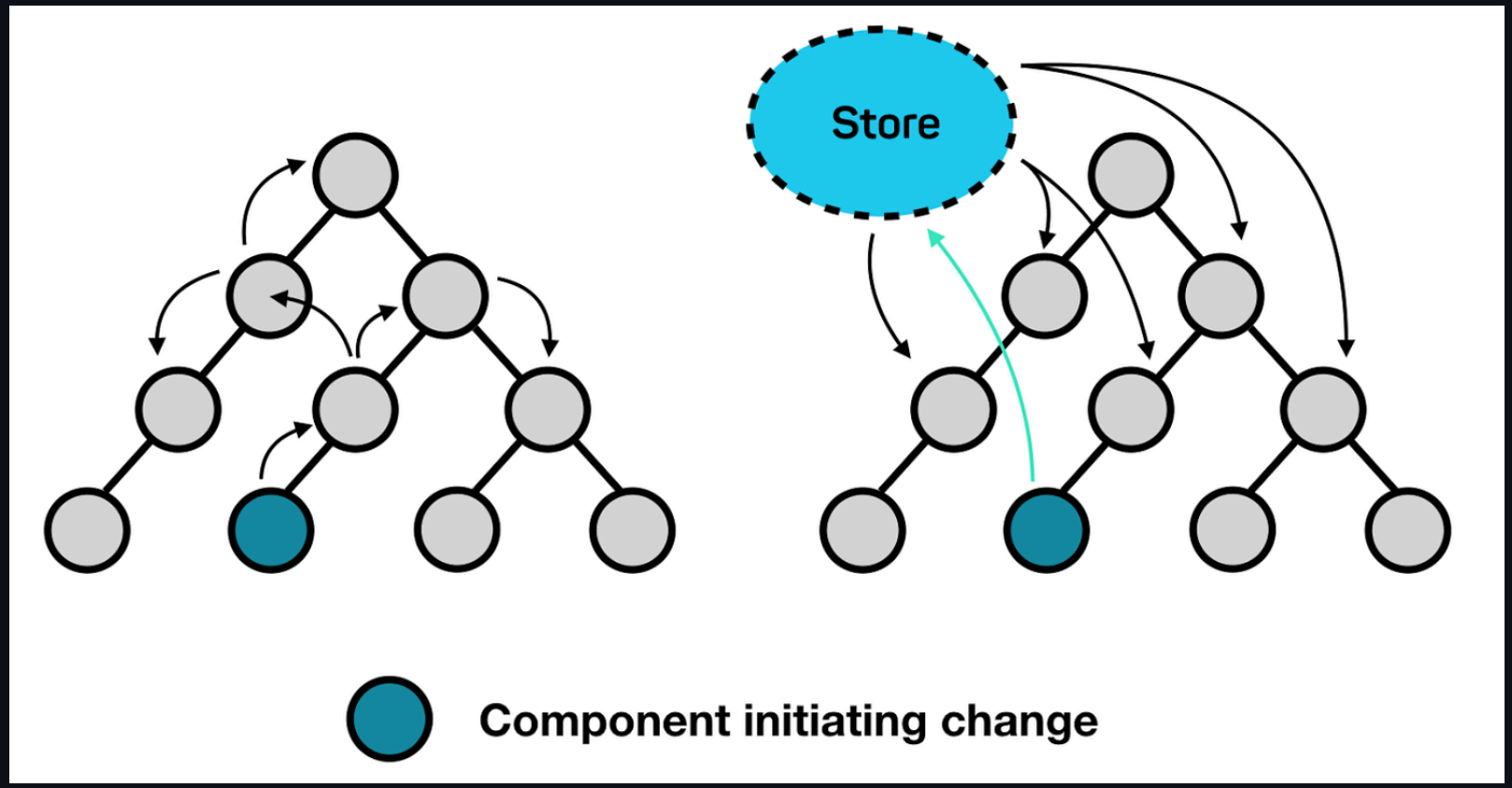
- `Good`
  - No external library required
  - Pure components
- `Bad`
  - Impractical for deep-nested components.
  - Fetching logic is too "far" from the view (JSX).

Method 4: `useEffect` + Global store

Global store pattern

What does using a global store solve?

- Multiple copies of states
- Prop drilling
- Unncessary re-render


Store
Component initiating change

Global store libraries / API

- `React Context`
- `Redux`
- `Jotai`
- `Zustand`

React Context

- Native API
- Fine, but...

```jsx
const App = () => {
  // ... some code
  return (
    <>
      <ReduxProvider value={store}>
        <ThemeProvider value={theme}>
          <OtherProvider value={otherValue}>
            <OtherOtherProvider value={otherOtherValue}>
              {/** ... other providers*/}
              <HellProvider value={hell}>
                <HelloWorld />
              </HellProvider>
              {/** ... other providers*/}
            </OtherOtherProvider>
          </OtherProvider>
        </ThemeProvider>
      </ReduxProvider>
    </>
  );
};
```

Redux

- Powerful
- Has Redux Dev Tool
- Can be used standalone
- Too much boiler plate for small projects

Redux Toolkit

The official, opinionated, batteries-included toolset for efficient Redux development

Get Started

Simple

Includes utilities to simplify common use cases like **store setup, creating reducers, immutable update logic**, and more.

Opinionated

Provides **good defaults for store setup out of the box**, and includes **the most commonly used Redux addons built-in**.

Powerful

Takes inspiration from libraries like Immer and Autodux to let you **write "mutative" immutable update logic**, and even **create entire "slices" of state automatically**.

Effective

Lets you focus on the core logic your app needs, so you can **do more work with less code**.

You Might Not Need Redux

Dan Abramov · Follow

3 min read · Sep 20, 2016

42K 99

People often choose Redux before they need it. “What if our app doesn’t scale without it?” Later, developers frown at the indirection Redux introduced to their code. “Why do I have to touch three files to get a simple feature working?” Why indeed!

Zustand

- Minimalist
- Use Redux-style (flux principle)
- No provider

Jotai

- Another cool library but I never used it.

Setup

- `git checkout -t origin/zustand`
- `pnpm i`

Store

`./src/stores/useGlobalStore.ts`

```ts
import { create } from "zustand";

interface Store {
  clock: string;
  setClock: (c: string) => void;
}

const useGlobalStore = create<Store>((set) => ({
  clock: "",
  setClock: (c) => set(() => ({ clock: c })),
}));
```

./src/components/Clock.tsx

```tsx
import useGlobalStore from "../stores/useGlobalStore";

const Clock: FC<Props> = () => {
  // No useState now
  const [clock, setClock] = useGlobalStore((state) => [
    state.clock,
    state.setClock,
  ]);
  const refetch = () => {
    // Fetching logic
  };
  useEffect(() => {
    if (initialFetch) refetch();
  }, []);

  // return JSX
};
```

`useEffect` + Global store

- `Good`
  - Shared state.
  - Less network requests
- `Bad`
  - Not pure components

Method 5: React Query + Custom hook

Reach Query

- Data-fetching + state management library
- Highly recommended!

Setup

- `git checkout -t origin/react-query`
- `pnpm i`

Provider

./src/main.tsx

```
import { QueryClient, QueryClientProvider } from "@tanstack/react-query";
import { ReactQueryDevtools } from "@tanstack/react-query-devtools";

// Create a client
const queryClient = new QueryClient();

createRoot(document.getElementById("root")!).render(
  <StrictMode>
    <QueryClientProvider client={queryClient}>
      <App />
      <ReactQueryDevtools initialIsOpen={false} />
  </StrictMode>
);
```

./src/hooks/useClock.ts

```ts
import { useQuery } from "@tanstack/react-query";

function getClock() {
  // Return promise
}

function useClock() {
  const query = useQuery({
    // Options
  });

  return { clock: query.data ?? "", refetch: query.refetch };
}

export default useClock;
```

Note

- Try inspect `query` object.
- Try navigate away and refocus the tab.
- Try option `refetchInterval`
- Try using the dev tool.

React Query + Custom hook

- `Good`
  - Do I have to repeat myself?
- `Bad`
  - A little bit of setup / learning curve
- `Note`
  - Use it please.

Real-time

Options

- Websocket
- Server-sent events

Websocket

- Protocol that establishes a full-duplex communication channel over a single TCP connection
  - Send data to the browser + receive data from the browser (`bi-directional`)
- Can transmit both binary data and UTF-8.
- Usage
  - Chat application

Server-send events

- SSE establishes a long-open HTTP channel from server to client.
  - Data only flows from a server to clients (`uni-directional`)
- Usage
  - Online stock quotes
  - Timeline or feed view

Advantages of SSE over Websockets:

- Transported over simple HTTP instead of a custom protocol
  - Simpler protocol
- Can be poly-filled with javascript to "backport" SSE to browsers that do not support it yet.
- Built in support for re-connection and event-id
- No trouble with corporate firewalls doing packet inspection

Advantages of Websockets over SSE:

- Real time, two directional communication.
- Native support in more browsers
- Only WS can transmit both binary data and UTF-8
    - SSE is limited to UTF-8.

SSE Gotchas

- Limited number of open connections
  - Maximum of 6 tabs per browser + domain
  - Browser restriction, not server

Method 6: Websocket

Setup

- `git clone -b main https://github.com/fullstack-67/df-websocket.git df-websocket`

Backend / Frontend

- `cd backend` / `cd frontend`
- `pnpm i`
- `npm run dev`

Method 7: Server-sent events

Setup

- `git clone -b main https://github.com/fullstack-67/df-sse.git df-sse`

Backend / Frontend

- `cd backend` / `cd frontend`
- `pnpm i`
- `npm run dev`