Fullstack Development

Data Fetching and State Management

Case Study

> Fetching Clock from External Service

External Service

- `git clone https://github.com/fullstack-68/df-backend.git`
- `pnpm i`
- `pnpm run dev`

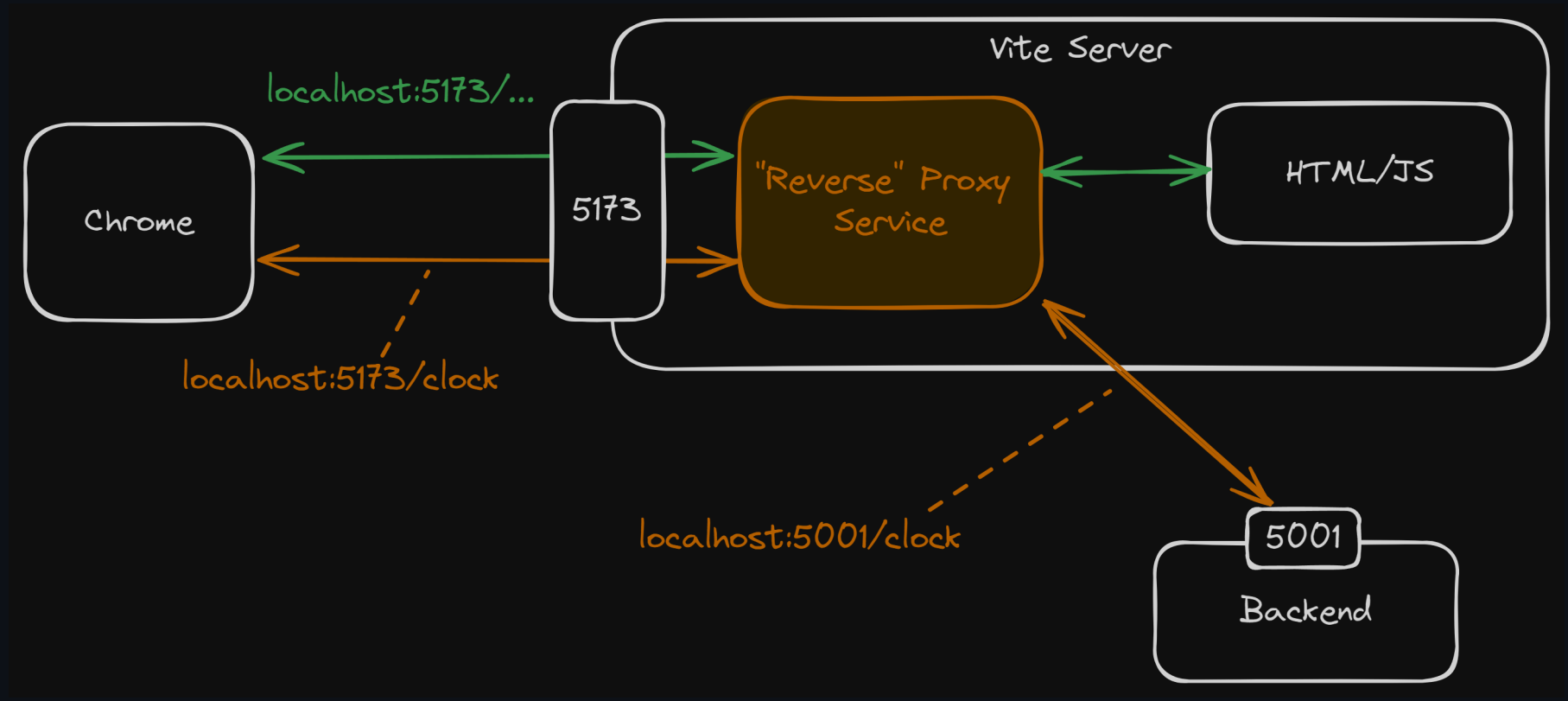
http://localhost:3001/clock

Part 1: Single Page Application (SPA)

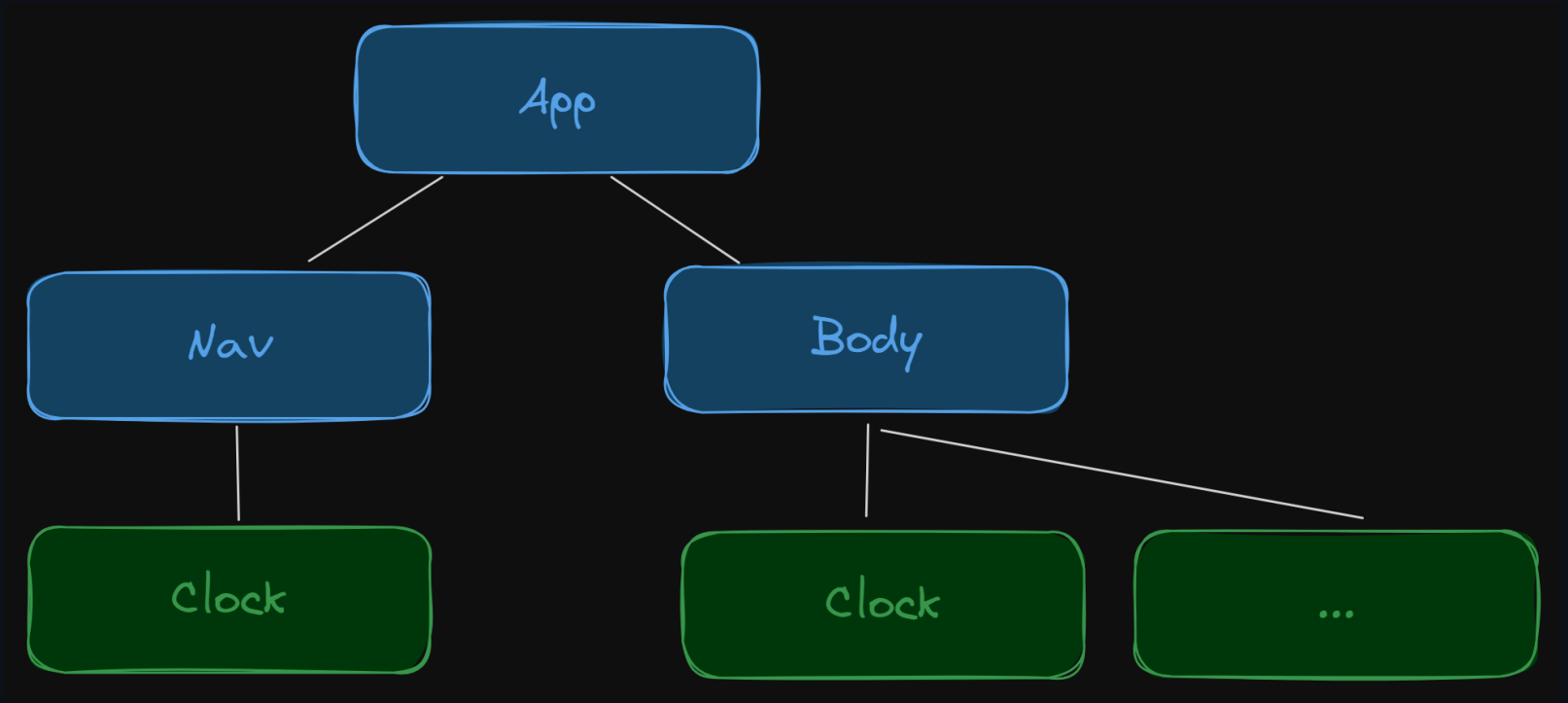
Part 1.1: `useEffect`

- `git clone https://github.com/fullstack-68/df-spa.git`
- `cd df-spa`
- `git checkout -t origin/useeffect`
- `pnpm i`
- `pnpm 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.
  - Notice data fetching from each component instance (many times).

Part 1.2: `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

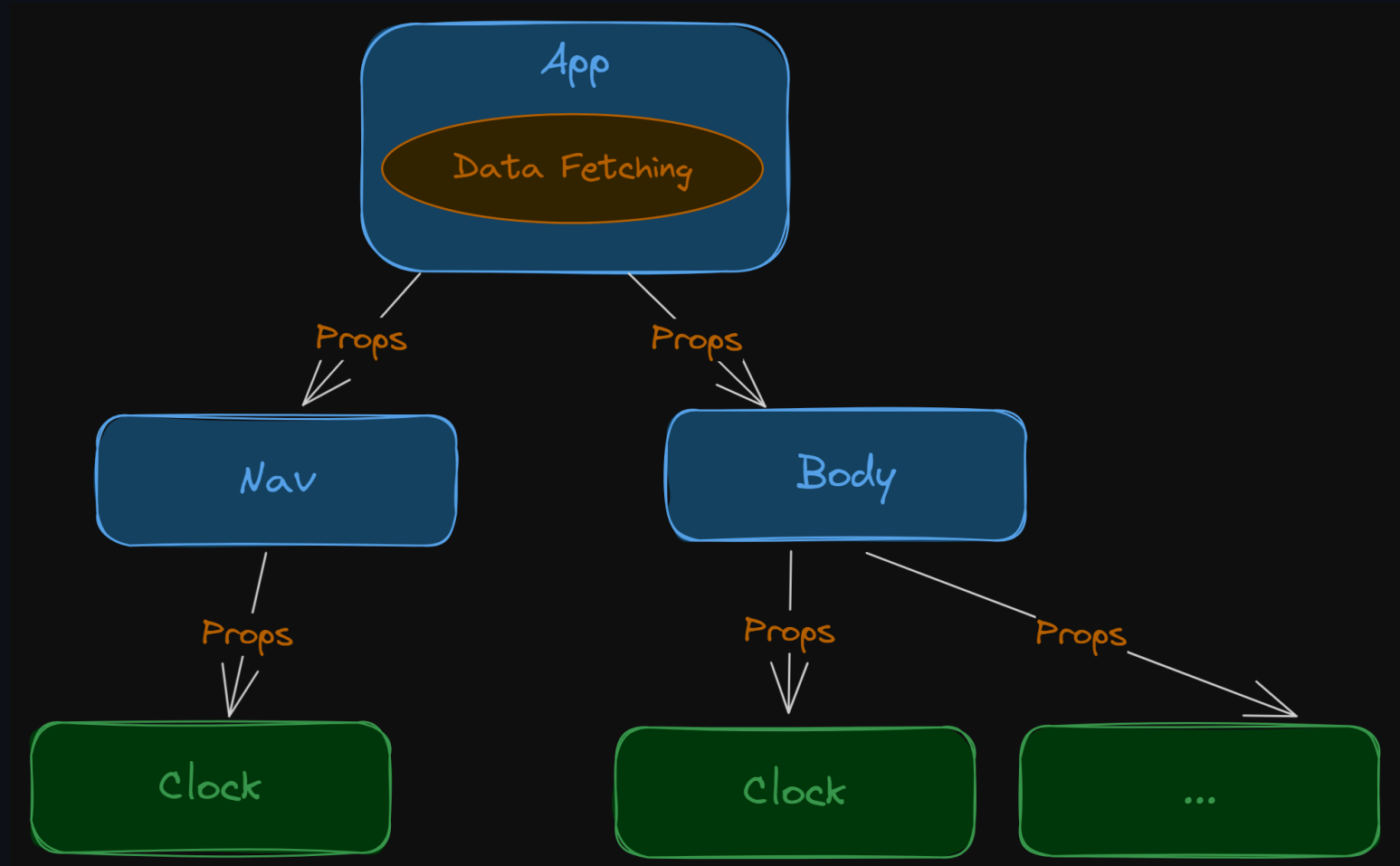
- `Good`
  - Logic encapsulation
  - Cleaner components
- `Comment`
  - States are still all local.

Part 1.3: `useEffect` + Prop drilling

Prop drilling

- `git checkout -t origin/prop-drilling`

Prop drilling



`useEffect` + Prop drilling

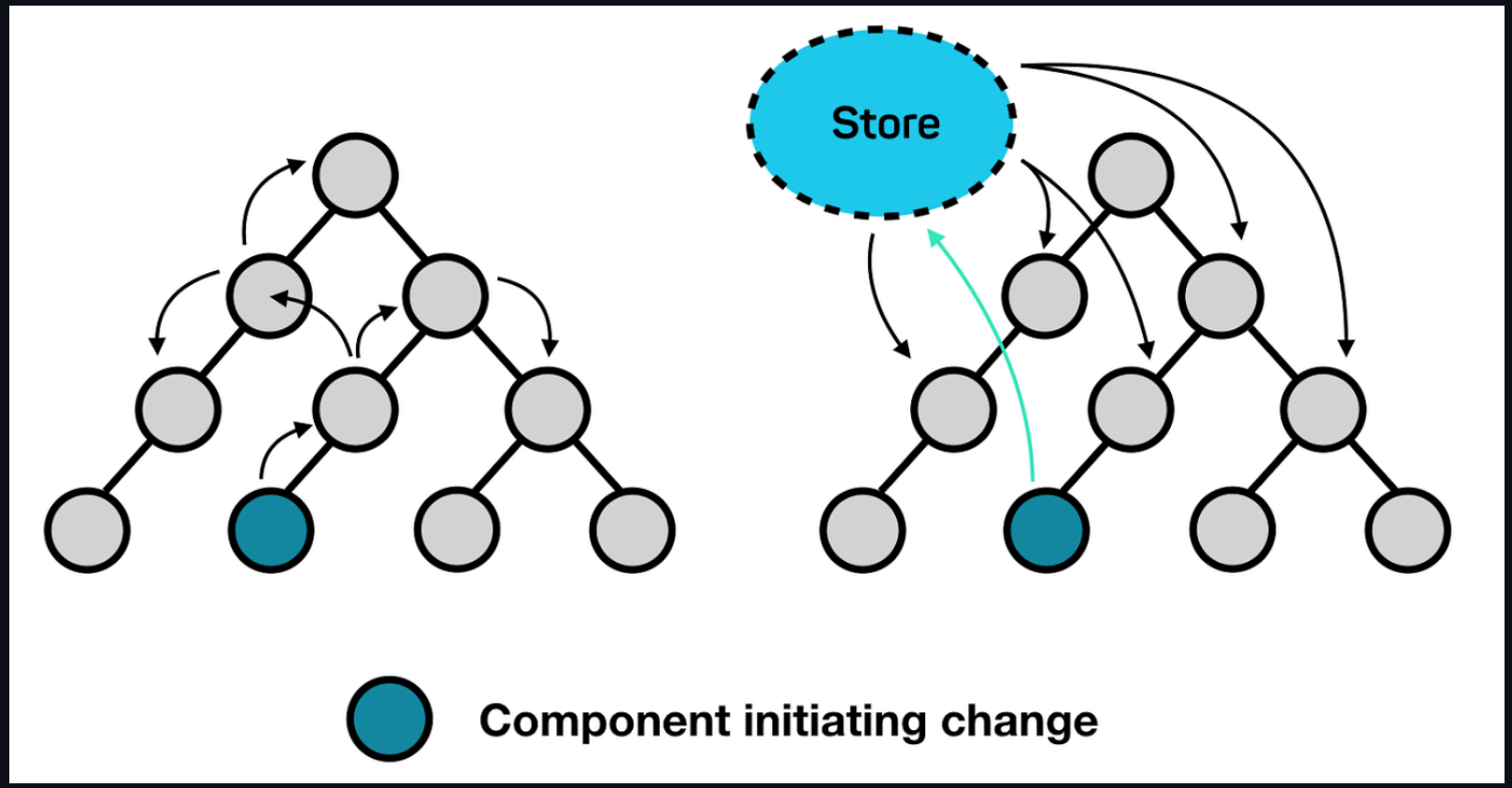
- `Good`
  - No external library required
  - Pure components
- `Bad`
  - Impractical for deep-nested components.
  - Fetching logic is too "far" from the view (JSX).
- `Comment`
  - Notice that we don't have to separately fetch data foreach component anymore (good).

Part 1.4: `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`
- `Zustand`
- `Jotai`

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

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";
import { useShallow } from "zustand/react/shallow";

const Clock: FC<Props> = () => {
  // No useState now
  const [clock, setClock] = useGlobalStore(
    useShallow((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

Jotai

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

Atom

```
import { useAtom, atom } from "jotai";

const clockAtom = atom("");

const Clock: FC<Props> = () => {
  const [clock, setClock] = useAtom(clockAtom);

  //...
};
```

Jotai vs Zustand

> Bottom Up vs Top Down
> https://github.com/pmndrs/jotai/issues/13

Part 1.5: Tanstack Query + Custom hook

Tanstack Query

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

Setup

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

Provider

./src/main.tsx

```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.

Part 2: Next.js

Setup

- `git clone https://github.com/fullstack-68/df-nextjs.git`
- `cd df-nextjs`
- `pnpm i`
- `pnpm run dev`

Strategy

- `Clock` component (`./src/components/Clock.tsx`) is a *server component*.
  - Use `fetch` API to fetch data.
- Even though `fetch` is used in multiple instances (`clock`), Next.js automatically caches data so only one API called will be sent.
- We can use `server action` to refetch data.

./src/components/Clock.tsx

```tsx
import { revalidatePath } from "next/cache";

const Clock: FC<Props> = async () => {
  const res = await fetch("http://localhost:3001/clock"); // This is cached.
  const json = await res.json();
  const clock = json.data;

  // Server action for refetching
  async function refetch() {
    "use server";
    revalidatePath("/");
  }
  return <form action={refetch}>...</form>;
};
```

Gotcha

- In `production` mode, this clock will not refresh when refreshing the browser.
  - This behavior does not occur in `dev` mode.

```
const res = await fetch("http://localhost:3001/clock");
```

- You need to make sure the cache is invalidated.

```
const res = await fetch("http://localhost:3001/clock", {
  next: { revalidate: 1 },
});
```

Server Components

- `Good`
  - Fetching server-side is better. Less problem.
- `Bad`
  - Not interactive? (I am sure there is a solution for this.)

Part 3: 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.
- Existing authentication and authorization (such as cookies, headers, session tokens, or middleware) will automatically apply to the SSE endpoint.
- 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

Part 3.1: Server-Sent Event

(Back to `df-spa` folder)

- `git checkout -t origin/sse`

Backend

```
// Server-Send Event Endpoint
app.get("/sse/clock", async (req, res, next) => {
  const headers = {
    "Content-Type": "text/event-stream",
    Connection: "keep-alive",
    "Cache-Control": "no-cache",
  };
  res.writeHead(200, headers);

  // ...
});
```

Test SSE on Console

- Make sure that you are in the tab with http://localhost:3001

```
es = new EventSource("/sse/clock");
es.onmessage = (e) => console.log(e);
es.close();
```

Frontend

./src/components/Clock.tsx

```
useEffect(() => {
  function initSSE() {
    const events = new EventSource("/api/sse/clock");
    events.onmessage = (e: any) => {
      setClock(e?.data ?? "");
    };
  }
  initialFetch && initSSE();
}, []);
```

Side Note: Todo apps (SSE)

- `git clone https://github.com/fullstack-68/df-sse-todo.git`
- `cd df-sse-todo`

How it works.

- Frontend app connects to `/subscribe` endpoints which opens SSE connection.
- Backend app keep the `res` object of subscribers in an array.
- When a new `todo` is created, backend `broadcast` the `todo` text to all subscribers.

Part 3.2: Websocket

(Back to `df-spa` folder)

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

Backend

```
// SocketIO Integration
const server = http.createServer(app);
const io = new SocketIOServer(server);

io.on("connection", (socket) => {
  console.log("a user connected");
  setInterval(function () {
    // const dtStr = dayjs().format("DD/MM/YYYY HH:mm:ss");
    const dtStr = dayjs().format("HH:mm:ss");
    io.sockets.emit("clock", { clock: dtStr });
  }, 1000);
});
```

Frontend

- Add new proxy endpoint in `vite.config.ts`
- Create socket client in `./src/socket.ts`

Frontend

./src/App.tsx

```
useEffect(() => {
  // ...
  function onClockEvent(value: { clock: string }) {
    setClock(value.clock);
  }
  socket.on("clock", onClockEvent);
  // ...
}, []);
```

Take-Home Messages

- Use `custom hook` to consolidate logic.
- Use global stores for client states.
- Use `Tanstack Query` for server states (SPA).
- Use server components for server states (NextJS).
- Consider `SSE` too (not only `Websocket`).