[사용자] ReactQuery와 Zustand 정리 (부제: user 정보를 꼭 상태값으로 저장해야할까?)

구현 어려움 정도	개념
link	[사용자] ReactQuery와 Zustand 정리 (부제: user 정보를 꼭 상태값으로 저장해야할까?)
구현 완료	☑
기간	@December 15, 2024
정리 완료	☑

user 정보를 zustand로 저장해서 사용하던 중에, reactQuery를 사용해 테이블에 있는 정보를 가져오는 hooks를 만들면서 굳이 zustand를 사용해야하나? 하는 의문이 생겼습니다.
이런 생각이 든 이유는 reactQuery는 “캐시” 기능이 있기 때문에 정말 굳이?라는 생각이 들었던 것 같습니다.

그래서 이에 대해 명확히 정리하고 넘어가는 게 좋을 것 같다는 생각이 들었습니다.

zustand와 reactQuery는 서로 다른 방식으로 상태를 관리합니다.

React Query

서버 상태를 관리하는 라이브러리로, API 요청을 처리하고 캐싱, 동기화, 에러 처리 등을 매우 효율적으로 지원합니다.

서버에서 가져온 데이터에 대해서는 ReactQuery가 자동으로 캐시를 관리하고 필요시 리패칭하는 등의 기능을 제공합니다.

Zustand

클라이언트 측 상태관리 라이브러리로 컴포넌트 간 공유할 필요가 있는 데이터를 전역 상태로 관리합니다. 상태를 변경하고 컴포넌트 간 상태를 쉽게 전달하는 데 유용합니다.

로그인한 사용자 정보 같은 "어디에서나 필요할 수 있는 상태"를 관리하는 데 좋습니다.

왜 Zustand를 사용해야 할까?

Zustand는 React Query와 같은 서버 상태 관리 도구와 함께 사용될 수 있지만, 주로 클라이언트 상태를 관리할 때 유용합니다.

예를들어 로그인한 사용자의 상태를 전역으로 관리하거나 UI 상태(모달 알림 여부, 테마 상태)등을 관리할 때 좋습니다.

함께 사용하기!

useUserData 같은 hooks으로 유저 데이터를 ReactQuery에서 관리하고 하면서 해당 데이터를 전역 상태로 관리할 필요가 없다면 zustand가 필요하지 않을 수 있습니다.

하지만 로그인 상태나 유저 정보를 애플리케이션 전역에서 재사용해야 한다면 Zustand를 통해 관리하는 것이 좋다고 합니다.

1. React Query로 유저 데이터를 불러오기
2. 유저 데이터를 **Zustand**로 저장하여 전역 상태로 관리

```
// useUserStore

import { User } from "@/types/db";
import { create } from "zustand";

type State = {
  user: Partial<User> | null;
}

type Action = {
  updateUser: (user: State["user"]) => void;
}
```

```
const useUserStore = create<State & Action>((set) => ({
  user: {
    id: "",
    name: "",
    email: "",
  },
  updateUser: (user) => set({ user }),
}))

export { useUserStore };
```

```
// useUserData hooks

import { useQuery } from '@tanstack/react-query';
import { useUserStore } from '@/stores/useUserStore';
import { getUserInfo } from '@/data/user';

export const useUserData = (userId: string | null) => {
  const { updateUser } = useUserStore();  // Zustand로 유저 상태 업데이트
  const { data, isLoading, isError } = useQuery({
    queryKey: ['user', userId],
    queryFn: async () => {
      if (!userId) return null;
      const data = await getUserInfo(userId);
      return data;
    },
    onSuccess: (data) => {
      if (data) {
        updateUser(data);  // React Query로 데이터를 가져온 후 Zustand에 저장
      }
    },
  });

  return { data, isLoading, isError };
};

export default useUserData;
```

React Query가 서버에서 데이터 유저를 가져오고 캐싱과 리패칭을 자동으로 관리합니다.

데이터가 성공적으로 로드되면 해당 데이터를 Zustand에 저장해 전역 상태로 사용이 가능합니다.

두 라이브러리의 장점도 함께 알아보기

1. Zustand의 장점

- 간단하고 직관적인 API
  - 설정이 간단하고 기본적인 상태 관리 기능을 손쉽게 제공
- 최소한의 리렌더링
  - Zustand는 구독 시스템을 사용해 상태가 변경된 부분만 리렌더링함
  - 특정 상태만 구독하고 다른 상태는 불필요하게 리렌더링되지 않도록 하기 때문에 성능상의 이점이 있음
- React와 독립적인 구조
  - React외의 다른 환경(예: javascript)에서도 사용할 수 있기 때문에 더 유연한 설계가 가능함
  - 상태는 store 안에서 관리되므로 React 컴포넌트 외부에서도 쉽게 접근하고 변경할 수 있음

- 미니멀한 라이브러리
  - 경량화된 라이브러리로 매우 적은 용량에 불필요한 기능이 없음(프로젝트 크기를 줄이는데 용이)
- 서버와 클라이언트 상태 함께 관리 가능
  - 로그인 상태, 테마 상태, 활성화된 페이지 등 전역적으로 필요한 상태를 쉽게 관리 가능

2. React Query의 장점

- 자동 캐싱
  - 서버에서 데이터를 가져오고 이를 자동으로 캐싱하여 불필요한 네트워크 요청을 방지
  - 이후 동일한 데이터 요청이 있을 경우, 캐시된 데이터를 바로 제공하므로 성능 향상에 유리함
- 자동 리패칭
  - 네트워크 상태가 변경되거나 데이터가 오래 됐을 때, 자동으로 리패칭해줌
    - 예) 인터넷 연결이 다시 될 때 등
- 서버 상태에 최적화된 에러 및 로딩 처리
  - isLoading, isError 같은 상태를 제공하므로 로딩 및 에러 상태를 관리하는 코드가 간결해짐
- 자동 데이터 갱신
  - API 응답 후 일정 시간마다 데이터를 갱신하는 방식으로 자동으로 최신 데이터를 유지할 수 있음
- 여러 컴포넌트에서 같은 데이터를 공유하고 있을 때, 데이터를 동기화
  - 하나의 컴포넌트가 데이터를 갱신하면 관련 모든 컴포넌트가 자동으로 리렌더링되어 같은 상태를 유지하게 해줌
- 서버 상태 관리의 간소화
  - API 요청, 캐싱, 동기화, 에러 처리 등 서버 상태 관리에 필요한 작업을 ReactQuery 하나로 처리 가능

3. 함께 사용할 때의 장점

1. 서버 데이터와 클라이언트 데이터를 동시에 관리할 수 있음
2. 자동 캐싱 및 동기화 처리가 됨
3. 코드의 유지보수가 쉬운
   a. React Query는 서버 데이터를 관리하고, Zustand는 클라이언트 데이터를 관리하므로 상태가 분리되어 관리가 더 수월함

결론!

둘을 병행해서 사용하는 것이 좋다고 합니다.