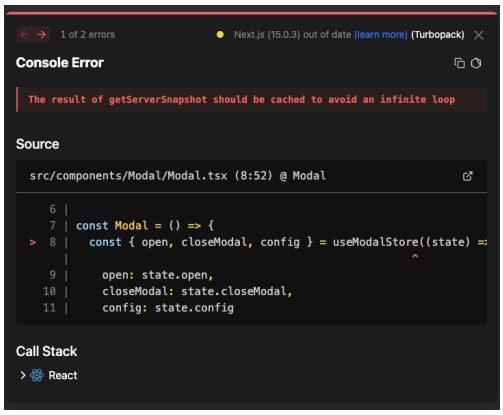
[사용자] The result of getServerSnapshot should be cached to avoid an infinite loop

구현 어려움 정도	⭐⭐
link	[사용자] **The result of getServerSnapshot should be cached to avoid an infinite loop**
구현 완료	☑
기간	@December 13, 2024
정리 완료	☑

Modal을 전역적으로 사용하려 했는데, 무한루프 에러가 발생했다..!




```
useModalStore((state) => ({
  open: state.open,
  closeModal: state.closeModal,
  config: state.config
}))
```

위의 코드가 useModalStore의 selector 내부에서 새 객체를 매번 생성하기 때문에 셀렉터가 매번 새 객체를 반환해서.. 무한루프 에러가 뜨는 것이라고 합니다.

함수 호출마다 새로운 객체를 생성하게 되는데, React가 이를 감지하고 상태가 변경되지 않았더라도 리랜더링이 발생할 수 있다고 합니다.

zustand에서는 상태를 구독할 때 불필요한 리렌더링을 피해기 위해 캐싱된 값을 사용하는 것이 중요하다고 합니다.

상태가 변하지 않을 경우 새로운 객체를 반환하지 않도록 해야합니다.

1. selector 사용으로 구독 범위 좁하기

Zustand에서는 `useStore` 호출 시 전체 상태가 아닌 필요한 상태만 선택하도록 selector를 사용할 수 있습니다. 이를 통해 특정 상태 변경에만 반응하고 나머지 상태 변경 시, 리랜더링되지 않도록 할 수 있습니다.

아래 코드는 상태를 개별적으로 구독하기 때문에 상태 변경시 불필요한 컴포넌트 리렌더링을 방지할 수 있습니다.

```
import { useModalStore } from "@/stores/useModalStore";

const Modal = () => {
  // 필요한 상태만 선택적으로 구독
  const open = useModalStore((state) => state.open);
  const config = useModalStore((state) => state.config);
```

```
  const closeModal = useModalStore((state) => state.closeModal);

  return (
    <Dialog open={open} onOpenChange={closeModal}>
      {/* ... */}
    </Dialog>
  );
};
```

2. shallow를 사용해 객체/배열 상태 비교 최적화하기

zustand는 상태 변경을 감지할 때 객체나 배열이 새로 생성되면 리렌더링을 트리거합니다.

`zustand/shallow`를 사용하면 얕은 비교로 최적화하여 불필요한 리렌더링을 방지할 수 있습니다.

상태 객체 전체가 아닌 객체의 필드 값을 기준으로 상태 변경 여부를 판단합니다.

```
import { shallow } from "zustand/shallow";
import { useModalStore } from "@/stores/useModalStore";

const Modal = () => {
  const { open, closeModal, config } = useModalStore(
    (state) => ({
      open: state.open,
      closeModal: state.closeModal,
      config: state.config,
    }),
    shallow // shallow 비교로 객체 내부 값 변경만 감지
  );

  return (
    <Dialog open={open} onOpenChange={closeModal}>
      {/* ... */}
    </Dialog>
  );
};
```

3. React.memo 사용해 리렌더링 제한하기

- React.memo는 **고차 컴포넌트**(High Order Component)
- 고차 컴포넌트는 컴포넌트를 가져와 새 컴포넌트를 반환하는 함수
- React.memo는 props가 변경되지 않으면 반환하지 않는 강화된 컴포넌트를 반환
- props 혹은 props의 객체를 비교할 때 얕은 비교를 함

useMemo와의 차이점

- useMemo는 React Hooks인 반면, React.memo는 HOC(고차 컴포넌트)
- React Hooks인 useMemo는 컴포넌트 내부에서만 사용이 가능하다는 차이점이 있음

```
import React from "react";

const Footer = React.memo(({ footer }: { footer: React.ReactNode }) => {
  return <div>{footer}</div>;
});

// 모달 컴포넌트에서 Footer를 사용
const Modal = () => {
```

```
  const { open, config, closeModal } = useModalStore(
    (state) => ({
      open: state.open,
      config: state.config,
      closeModal: state.closeModal,
    }),
    shallow
  );

  return (
    <Dialog open={open} onOpenChange={closeModal}>
      <Footer footer={config?.footer} />
    </Dialog>
  );
};
```

- footer 컴포넌트는 props가 변경되지 않는 한 리렌더링이 되지 않음
- 재사용 가능한 컴포넌트에 React.memo를 적용하면 큰 규모의 프로젝트에서 성능 향상에 도움이 됨

4. 미들웨어 사용으로 상태 로깅 및 디버깅하기

zustand의 미들웨어를 활용하여 상태 변경시 디버깅 로깅을 추가하거나 상태 변경 조건을 제어 가능

```
import { create } from "zustand";
import { devtools } from "zustand/middleware";

const useModalStore = create(
  devtools((set) => ({
    open: false,
    config: undefined,
    openModal: (config) => set({ open: true, config }),
    closeModal: () => set({ open: false, config: undefined }),
  }))
);

export default useModalStore;
```

- 개발 중 상태 변경 과정을 DevTools에서 확인 가능
- 상태 변경 과정에 로직 추가 가능

5. 상태 변경을 최소화하도록 설계하기

상태 변경은 가능한 최소한으로 해야 리렌더링을 줄일 수 있습니다.

```
// 상태를 불필요하게 업데이트
set({ open: true, config: { ...config, updatedAt: new Date() } });

// 필요한 상태만 업데이트
set((state) => ({
  open: true,
  config: { ...state.config, ...config },
}));
```