Zustand 기반 상태관리 구조 및 선택 이유 리포트

1. 왜 Zustand를 선택했는가?

1.1. 경험 기반 선택

- 다양한 상태관리 라이브러리(Redux, Recoil, Zustand 등) 중에서, zustand를 자주 사용하여 선택하게 됨

1.2. 추가적인 선택 이유

- **가볍고 빠른 성능**: Redux 등과 달리 보일러플레이트가 적고, 미들웨어 없이도 직관적으로 상태를 관리할 수 있음.
- **직관적 API**: create/set/get 등 함수형 API로 코드가 간결해짐.
- **React와의 높은 호환성**: Hook 기반으로 React 컴포넌트와 자연스럽게 결합됨.
- **Selector/구독 최적화**: 필요한 상태만 구독할 수 있어 불필요한 리렌더링을 최소화함.
- **타입스크립트 친화적**: 타입 안전성이 높아 대규모 프로젝트에 적합.
- **미니멀한 번들 사이즈**: 번들 크기가 작아 프론트엔드 성능에 유리함.

2. 왜 분할(store별 파일) 구조로 작성했는가?

2.1. 도메인별 책임 분리

- 자산군 관리, 모멘텀, 재진입, 전략 등 각 도메인별로 store를 분리하여, 각 상태와 액션의 책임을 명확히 함.
- 유지보수 및 확장 시, 특정 도메인만 수정/추가가 가능해짐.

2.2. 코드 가독성 및 관리 용이

- 하나의 거대한 store가 아닌, 기능별로 store를 분리함으로써 코드 가독성이 높아지고, 협업 시 충돌이 줄어듦.
- 각 store의 인터페이스와 액션이 명확하게 정의되어, 타입 안정성과 예측 가능성이 높아짐.

2.3. 리렌더링 최적화

- 각 store는 필요한 컴포넌트에서만 구독되므로, 불필요한 리렌더링을 방지할 수 있음.
- selector, shallow 비교 등 zustand의 최적화 기능을 활용하기 용이함.

3. 각 store의 역할 및 구조

3.1. assetsStore.ts

- 자산군 리스트 및 자산별 정보 관리
- add/update/delete 등 CRUD 액션 제공
- 자산군별 체크박스 상태 등 도메인 특화 로직 포함

3.2. momentumStore.ts

- 모멘텀 전략 관련 상태 및 설정값 관리

- 모멘텀 on/off, 세부 설정값 업데이트 등 액션 제공

3.3. reEntryStore.ts

- 재진입 전략 관련 상태 및 설정값 관리
- 재진입 on/off, 세부 설정값 업데이트 등 액션 제공

3.4. strategyStore.ts

- 전체 전략의 이름, 알고리즘, 시드, 리밸런싱 기간 등 관리
- 전략별 세부 설정값 및 기간 관리

4. 결론

Zustand는 빠른 개발, 간결한 코드, 구독 최적화, 타입 안정성 등에서 장점이 있음 대규모 프로젝트에서 복잡한 상태 흐름, 미들웨어, DevTools, 팀 협업이 중요한 경우에는 Redux가 더 적합할 수 있습니다. 하지만 본 과제에선 빠른 개발을 위해 Zustand를 선택 하게 되었습니다.