Memory updated

Here's a well-organized structure for a MobX-React TypeScript project with three stores. This structure ensures scalability, maintainability, and clean separation of concerns.

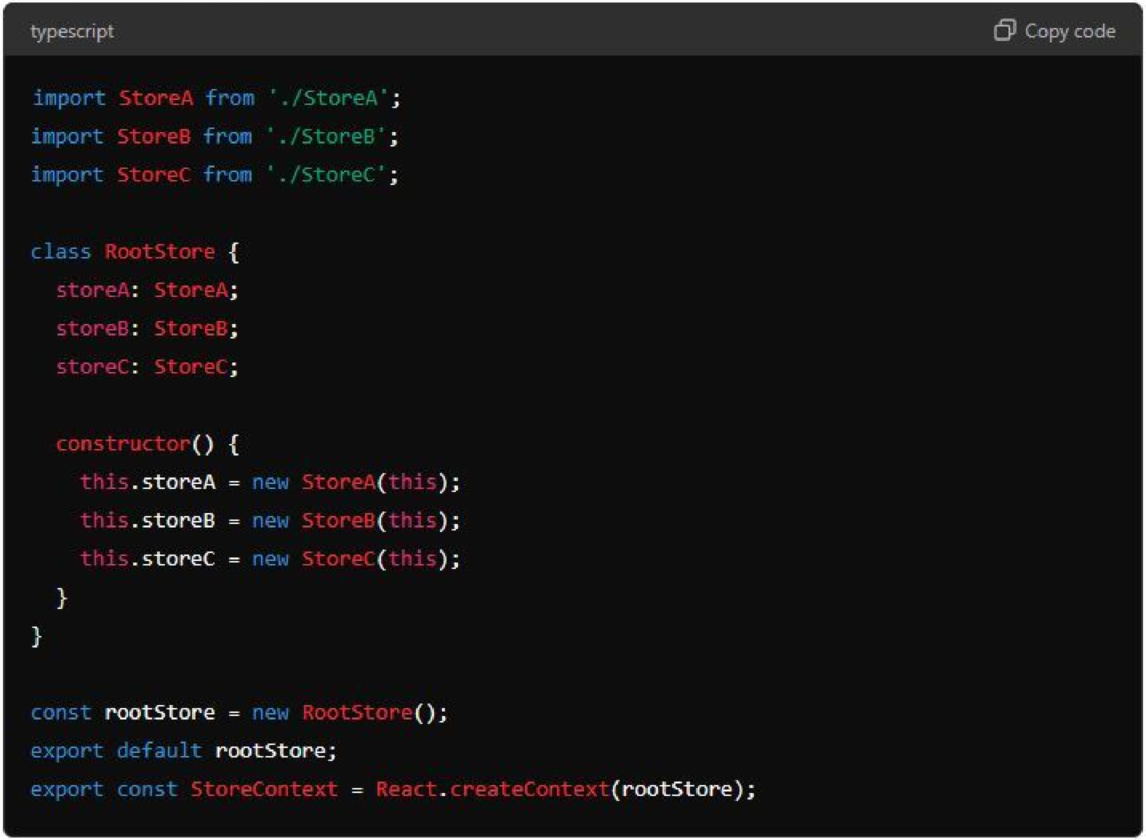
1. Project Structure

javascript Copy code

```
/src
|-- /components
|   |-- /common         // Reusable components (e.g., buttons, inputs)
|   |-- /specific       // Components specific to certain pages or features
|-- /stores
|   |-- /RootStore.ts  // Aggregates all the stores
|   |-- /StoreA.ts     // First store
|   |-- /StoreB.ts     // Second store
|   |-- /StoreC.ts     // Third store
|-- /pages
|   |-- /PageA         // Components related to Page A
|   |-- /PageB         // Components related to Page B
|   |-- /PageC         // Components related to Page C
|-- /hooks             // Custom hooks
|-- /types             // TypeScript types and interfaces
|-- /utils             // Utility functions
|-- App.tsx            // Main App component
|-- index.tsx          // Entry point
```

2. Detailed Breakdown

`/stores/RootStore.ts`

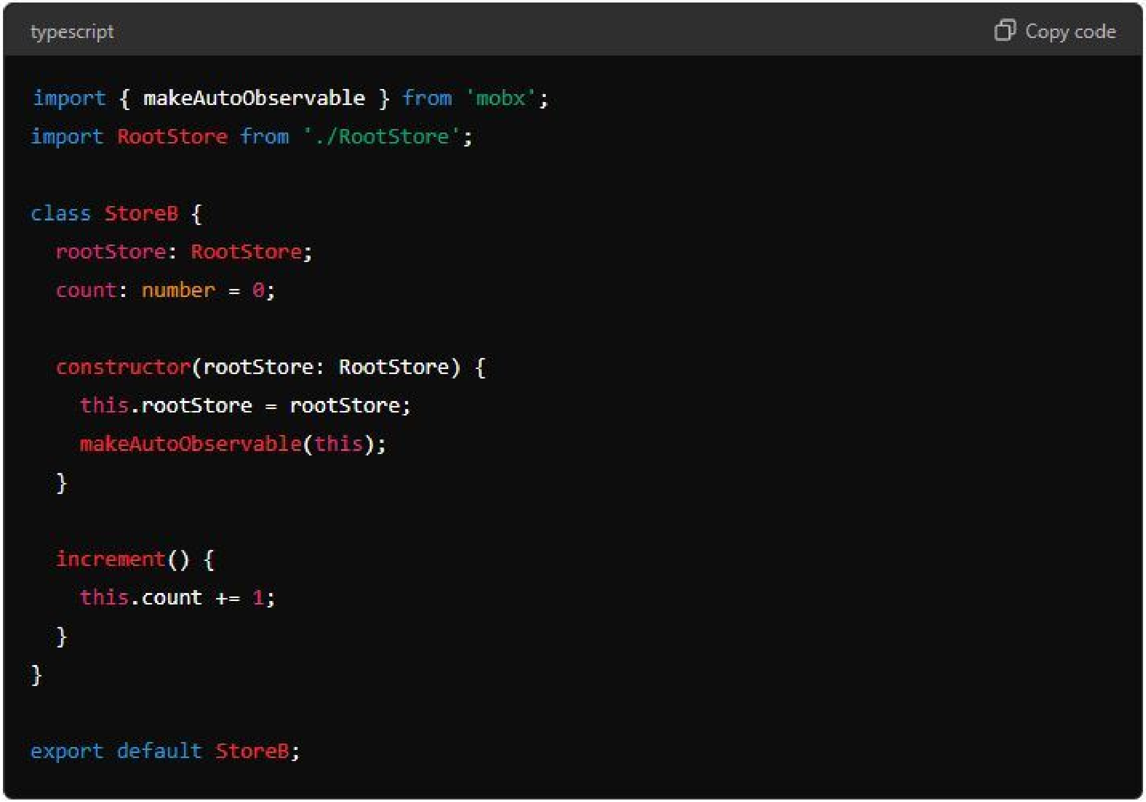

typescript Copy code

```typescript
import StoreA from './StoreA';
import StoreB from './StoreB';
import StoreC from './StoreC';

class RootStore {
  storeA: StoreA;
  storeB: StoreB;
  storeC: StoreC;

  constructor() {
    this.storeA = new StoreA(this);
    this.storeB = new StoreB(this);
    this.storeC = new StoreC(this);
  }
}

const rootStore = new RootStore();
export default rootStore;
export const StoreContext = React.createContext(rootStore);
```

`/stores/StoreA.ts`

```typescript
import { makeAutoObservable } from 'mobx';
import RootStore from './RootStore';  // For cross-store communication

class StoreA {
  rootStore: RootStore;
  data: string = '';

  constructor(rootStore: RootStore) {
    this.rootStore = rootStore;
    makeAutoObservable(this);
  }

  setData(data: string) {
    this.data = data;
  }
}

export default StoreA;
```

`/stores/StoreB.ts`



```typescript
import { makeAutoObservable } from 'mobx';
import RootStore from './RootStore';

class StoreB {
  rootStore: RootStore;
  count: number = 0;

  constructor(rootStore: RootStore) {
    this.rootStore = rootStore;
    makeAutoObservable(this);
  }

  increment() {
    this.count += 1;
  }
}

export default StoreB;
```

`/stores/StoreC.ts`

typescript Copy code

```typescript
import { makeAutoObservable } from 'mobx';
import RootStore from './RootStore';

class StoreC {
  rootStore: RootStore;
  items: string[] = [];

  constructor(rootStore: RootStore) {
    this.rootStore = rootStore;
    makeAutoObservable(this);
  }

  addItem(item: string) {
    this.items.push(item);
  }
}

export default StoreC;
```

3. Using Stores in Components

Example Usage in a Component

```typescript
import React, { useContext } from 'react';
import { observer } from 'mobx-react-lite';
import { StoreContext } from '../stores/RootStore';

const MyComponent: React.FC = observer(() => {
  const { storeA, storeB, storeC } = useContext(StoreContext);

  return (
    <div>
      <h1>{storeA.data}</h1>
      <button onClick={() => storeB.increment()}>Increment: {storeB.count}</button>
      <ul>
        {storeC.items.map((item, index) => (
          <li key={index}>{item}</li>
        ))}
      </ul>
    </div>
  );
});

export default MyComponent;
```

4. Custom Hooks

Create custom hooks if you find yourself repeatedly accessing stores in your components:

`/hooks/useStore.ts`

```typescript
import { useContext } from 'react';
import { StoreContext } from '../stores/RootStore';

export const useStore = () => useContext(StoreContext);
```

Usage in a Component

```typescript
import React from 'react';
import { observer } from 'mobx-react-lite';
import { useStore } from '../hooks/useStore';

const MyComponent: React.FC = observer(() => {
  const { storeA, storeB, storeC } = useStore();

  // Same as before...
});
```

5. Types and Interfaces

Place shared types and interfaces in the `/types` folder for easy reuse and consistency across the project.

6. Utilities

Put utility functions in the `/utils` folder, keeping the business logic clean and separated from UI concerns.