MobX React TS Structure

GIVE ME THE BEST STRUCTURE FOR MOBX REACT TS WITH 2 DIFFRENT STORES AND RENDER JUST WHAT HAVE CHANGE

To create a well-structured MobX project in React with TypeScript and ensure that only the necessary parts of your UI are re-rendered when the state changes, you can follow this structure:

Project Structure

```
src/
│
├── stores/
│   ├── StoreA.ts
│   ├── StoreB.ts
│   └── index.ts
│
├── components/
│   ├── ComponentA.tsx
│   ├── ComponentB.tsx
│   └── index.ts
│
├── App.tsx
└── index.tsx
```

1. Setting Up MobX Stores

StoreA.ts

```typescript
import { makeAutoObservable } from "mobx";

class StoreA {
  valueA: string = "Initial Value A";

  constructor() {
    makeAutoObservable(this);
  }
```

```
  setValueA(newValue: string) {
    this.valueA = newValue;
  }
}

export const storeA = new StoreA();
```

StoreB.ts

```typescript
import { makeAutoObservable } from "mobx";

class StoreB {
  valueB: number = 0;

  constructor() {
    makeAutoObservable(this);
  }

  incrementValueB() {
    this.valueB += 1;
  }
}

export const storeB = new StoreB();
```

index.ts (Store Index)

```typescript
import { storeA } from "./StoreA";
import { storeB } from "./StoreB";

export const stores = {
  storeA,
  storeB,
};
```

2. Components

ComponentA.tsx

```typescript
import React from "react";
import { observer } from "mobx-react-lite";
import { stores } from "../stores";

const ComponentA: React.FC = observer(() => {
  const { storeA } = stores;

  const handleChange = (event: React.ChangeEvent<HTMLInputElement>) => {
    storeA.setValueA(event.target.value);
  };

  return (
    <div>
      <h1>Component A</h1>
      <input type="text" value={storeA.valueA} onChange={handleChange} />
      <p>Value A: {storeA.valueA}</p>
    </div>
  );
});

export default ComponentA;
```

ComponentB.tsx

```typescript
import React from "react";
import { observer } from "mobx-react-lite";
import { stores } from "../stores";

const ComponentB: React.FC = observer(() => {
  const { storeB } = stores;

  const handleClick = () => {
    storeB.incrementValueB();
  };

  return (
    <div>
```

```
      <h1>Component B</h1>
      <button onClick={handleClick}>Increment Value B</button>
      <p>Value B: {storeB.valueB}</p>
    </div>
  );
});

export default ComponentB;
```

3. App.tsx

```typescript
import React from "react";
import ComponentA from "./components/ComponentA";
import ComponentB from "./components/ComponentB";

const App: React.FC = () => {
  return (
    <div>
      <ComponentA />
      <ComponentB />
    </div>
  );
};

export default App;
```

4. index.tsx

```typescript
import React from "react";
import ReactDOM from "react-dom";
import App from "./App";

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
```

```
  document.getElementById("root")
);
```

Key Points:

1. **MobX Store Setup**: Use `makeAutoObservable` to automatically track the observables and actions in each store.
2. **Observer Components**: Wrap components with `observer` to ensure they only re-render when the relevant observables change.
3. **Component Re-Rendering**: Each component will only re-render when the observable data they depend on changes. This keeps the UI responsive and efficient.

With this structure, you ensure that only the necessary parts of your React application re-render when the MobX state changes.