MobX React Example

example mobx with react just with mobx libarary

Certainly! Below is a simple example of using MobX with React without any additional libraries:

jsx

```jsx
import React from 'react';
import { observable, action } from 'mobx';
import { observer } from 'mobx-react';

// MobX Store
class CounterStore {
  @observable count = 0;

  @action increment = () => {
    this.count++;
  };

  @action decrement = () => {
    this.count--;
  };
}

const counterStore = new CounterStore();

// React Component
const Counter = observer(() => {
  const { count, increment, decrement } = counterStore;

  return (
    <div>
      <h1>Counter: {count}</h1>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
});
```

```
export default Counter;
```

In this example:

- We create a MobX store called `CounterStore` which has an observable property `count` and two actions `increment` and `decrement`.
- We create an instance of `CounterStore` called `counterStore`.
- We create a React component called `Counter` which uses MobX's `observer` function to observe changes to the store and automatically re-render the component when any observable data changes.
- Inside the `Counter` component, we access the `count` property from the store and the `increment` and `decrement` actions, and bind them to button clicks.

This is a basic example to demonstrate MobX's usage with React. In a real-world application, you might have more complex stores and components, but the principles remain the same. MobX will automatically track dependencies between observable data and reactively update components when needed.

i must library mobx-react for mobx with react

Yes, typically you would use the `mobx-react` library when working with MobX in React. The `mobx-react` library provides some utilities and components that make it easier to integrate MobX with React.

Here's how you would modify the previous example to use `mobx-react`:

```jsx
import React from 'react';
import { observable, action } from 'mobx';
import { observer, Provider } from 'mobx-react';

// MobX Store
class CounterStore {
  @observable count = 0;

  @action increment = () => {
    this.count++;
  };
  @action decrement = () => {
    this.count--;
  };
}

const counterStore = new CounterStore();

// React Component
const Counter = observer(() => {
  const { count, increment, decrement } = counterStore;

  return (
    <div>
      <h1>Counter: {count}</h1>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
});

// Wrap the root component with the MobX Provider
const App = () => (
  <Provider counterStore={counterStore}>
    <Counter />
  </Provider>
);

export default App;
```

In this modified example:

- We import `Provider` from `mobx-react` and wrap the root component (`Counter` in this case) with it. This `Provider` component makes the MobX store available to all components in the component tree.
- Inside the `Counter` component, we don't directly import `counterStore` anymore. Instead, `counterStore` is injected into the component via props by the `Provider`.
- This pattern allows you to access the store in any component in the tree without needing to pass it down through multiple levels manually.

Using `mobx-react` simplifies the integration of MobX with React by providing a context-based approach for injecting stores into components and by offering useful utilities such as `observer`.

mobx with react without 'mobx-react-lite' and 'mobx-react' library example

11/11

Certainly! If you prefer not to use the `mobx-react` or `mobx-react-lite` libraries, you can still integrate MobX with React by using the `useContext` hook to access MobX stores. Here's an example without using `mobx-react` or `mobx-react-lite`:

jsx

```jsx
import React, { useContext } from 'react';
import { observable, action } from 'mobx';

// MobX Store
class CounterStore {
  @observable count = 0;

  @action increment = () => {
    this.count++;
  };

  @action decrement = () => {
    this.count--;
  };
}

const counterStore = new CounterStore();

// React Component
const Counter = () => {
  const { count, increment, decrement } = useContext(CounterStoreContext);

  return (
    <div>
      <h1>Counter: {count}</h1>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
};

// Context
const CounterStoreContext = React.createContext();

// Provider Component
const CounterProvider = ({ children }) => (
  <CounterStoreContext.Provider value={counterStore}>
    {children}
  </CounterStoreContext.Provider>
);

// App Component
const App = () => (
  <CounterProvider>
    <Counter />
  </CounterProvider>
);

export default App;
```

In this example:

- We create a context using `React.createContext()` to hold the MobX store instance.
- The `CounterProvider` component uses this context provider to make the store available to its child components.
- Inside the `Counter` component, we use the `useContext` hook to access the MobX store.
- We no longer need the `observer` function because we're not using `mobx-react`. However, if you want your components to reactively update when the store changes, you can manually trigger re-renders using React's state or force updates.

This approach eliminates the need for `mobx-react` or `mobx-react-lite` and relies solely on React's built-in context API. It's a bit more manual compared to using `mobx-react`, but it provides more flexibility and avoids additional dependencies.

example mobx with react ts in 2024

Certainly! Below is an example of using MobX with React in TypeScript:

```tsx
```

```
import React, { createContext, useContext } from 'react';
import { observable, action } from 'mobx';
import { observer } from 'mobx-react-lite';

// MobX Store
class CounterStore {
  @observable count = 0;

  @action increment = () => {
    this.count++;
  };

  @action decrement = () => {
    this.count--;
  };
}

const counterStore = new CounterStore();

// React Context
const CounterStoreContext = createContext<CounterStore>(counterStore);

// React Hook to access the store
const useCounterStore = () => useContext(CounterStoreContext);

// React Component
const CounterComponent: React.FC = observer(() => {
  const { count, increment, decrement } = useCounterStore();

  return (
    <div>
      <h1>Counter: {count}</h1>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
});

// App Component
const App: React.FC = () => {
  return (
    <CounterStoreContext.Provider value={counterStore}>
      <CounterComponent />
    </CounterStoreContext.Provider>
  );
};

export default App;
```

In this example:

- We create a MobX store class `CounterStore` with observable state (`count`) and actions (`increment` and `decrement`).
- We create an instance of `CounterStore` called `counterStore`.
- We define a React context `CounterStoreContext` with a type of `CounterStore` and provide it a default value of `counterStore`.
- We create a custom React hook `useCounterStore` to access the MobX store within functional components.
- The `CounterComponent` component uses the `observer` function from `mobx-react-lite` to automatically re-render when observable values change.
- In the `App` component, we wrap the `CounterComponent` with the `CounterStoreContext.Provider` to make the MobX store available throughout the component tree.

This example demonstrates a simple integration of MobX with React using TypeScript in 2024.

1/2

example mobx with react ts last version

Certainly! Below is an example of using MobX with React in TypeScript:

```tsx
import React from 'react';
import { observable, action } from 'mobx';
import { observer } from 'mobx-react-lite';

// MobX Store
```

```
class CounterStore {
  @observable count = 0;

  @action increment = () => {
    this.count++;
  };

  @action decrement = () => {
    this.count--;
  };
}

const counterStore = new CounterStore();

// React Component
const CounterComponent: React.FC = observer(() => {
  const { count, increment, decrement } = counterStore;

  return (
    <div>
      <h1>Counter: {count}</h1>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
});

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

export default App;
```

In this example:

- We define a MobX store class `CounterStore` with an observable state (`count`) and actions (`increment` and `decrement`).

- We create an instance of `CounterStore` called `counterStore`.
- The `CounterComponent` component is defined as a functional component and is wrapped with the `observer` function from `mobx-react-lite`, which automatically re-renders the component when observable values change.
- The `App` component renders the `CounterComponent`.

This example demonstrates a simple integration of MobX with React using TypeScript in the latest version.

make 2 store to rooststore mobx react ts example with context

4/4

Certainly! Below is an example of using MobX with React in TypeScript with multiple stores and context:

```tsx
import React, { createContext, useContext } from 'react';
import { observable, action } from 'mobx';
import { observer } from 'mobx-react-lite';

// First MobX Store
class CounterStore {
  @observable count = 0;

  @action increment = () => {
    this.count++;
  };

  @action decrement = () => {
    this.count--;
  };
}

const counterStore = new CounterStore();

// Second MobX Store
class TodoStore {
  @observable todos: string[] = [];
  @action addTodo = (todo: string) => {
    this.todos.push(todo);
  };

  @action removeTodo = (index: number) => {
    this.todos.splice(index, 1);
  };
}

const todoStore = new TodoStore();

// First React Context
const CounterStoreContext = createContext<CounterStore>(counterStore);

// Second React Context
const TodoStoreContext = createContext<TodoStore>(todoStore);

// Custom Hook to access Counter Store
const useCounterStore = () => useContext(CounterStoreContext);

// Custom Hook to access Todo Store
const useTodoStore = () => useContext(TodoStoreContext);

// Counter Component
const CounterComponent: React.FC = observer(() => {
  const { count, increment, decrement } = useCounterStore();

  return (
    <div>
      <h1>Counter: {count}</h1>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
});

// Todo Component
const TodoComponent: React.FC = observer(() => {
  const todoStore = useTodoStore();

  const handleAddTodo = () => {
    const todo = prompt('Enter todo:');
    if (todo) {
      todoStore.addTodo(todo);
    }
  };

  const handleRemoveTodo = (index: number) => {
    todoStore.removeTodo(index);
  };

  return (
    <div>
      <h1>Todos</h1>
      <button onClick={handleAddTodo}>Add Todo</button>
      <ul>
        {todoStore.todos.map((todo, index) => (
          <li key={index}>
            {todo}{' '}
            <button onClick={() => handleRemoveTodo(index)}>Remove</button>
          </li>
        ))}
      </ul>
    </div>
  );
});

// App Component
const App: React.FC = () => {
  return (
    <CounterStoreContext.Provider value={counterStore}>
      <TodoStoreContext.Provider value={todoStore}>
        <div>
          <CounterComponent />
          <TodoComponent />
        </div>
      </TodoStoreContext.Provider>
    </CounterStoreContext.Provider>
  );
};

export default App;
```

In this example:

- We have two MobX stores: `CounterStore` and `TodoStore`.
- Both stores are wrapped in separate React contexts: `CounterStoreContext` and `TodoStoreContext`.
- We create custom hooks `useCounterStore` and `useTodoStore` to access the respective stores within components.
- The `CounterComponent` uses the `useCounterStore` hook to access and display the counter value.
- The `TodoComponent` uses the `useTodoStore` hook to access and manipulate the list of todos.
- The `App` component renders both `CounterComponent` and `TodoComponent` wrapped with their respective context providers.

This example demonstrates how to manage multiple MobX stores with React context in TypeScript. Each component can access the relevant store using the provided custom hooks.

best parctice example react ts mobx multi store

2/2

Certainly! Here's an example showcasing best practices for managing multiple MobX stores in a React TypeScript application:

tsx

```tsx
import React, { createContext, useContext } from 'react';
import { makeAutoObservable, runInAction } from 'mobx';
import { observer } from 'mobx-react-lite';

// First MobX Store
class CounterStore {
  count = 0;

  constructor() {
    makeAutoObservable(this);
  }

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

  decrement = () => {
    this.count--;
  };
}

// Second MobX Store
class TodoStore {
  todos: string[] = [];

  constructor() {
    makeAutoObservable(this);
  }

  addTodo = (todo: string) => {
    runInAction(() => {
      this.todos.push(todo);
    });
  };

  removeTodo = (index: number) => {
    runInAction(() => {
      this.todos.splice(index, 1);
    });
  };
}

// Create instances of stores
const counterStore = new CounterStore();
const todoStore = new TodoStore();

// React Context for CounterStore
const CounterStoreContext = createContext<CounterStore | null>(null);

// React Context for TodoStore
const TodoStoreContext = createContext<TodoStore | null>(null);

// Custom hook to access CounterStore
const useCounterStore = () => {
  const store = useContext(CounterStoreContext);
  if (!store) {
    throw new Error('useCounterStore must be used within a CounterStoreProvider');
  }
  return store;
};

// Custom hook to access TodoStore
const useTodoStore = () => {
  const store = useContext(TodoStoreContext);
  if (!store) {
    throw new Error('useTodoStore must be used within a TodoStoreProvider');
  }
  return store;
};

// Counter Component
const CounterComponent: React.FC = observer(() => {
  const counterStore = useCounterStore();

  return (
    <div>
      <h1>Counter: {counterStore.count}</h1>
      <button onClick={counterStore.increment}>Increment</button>
      <button onClick={counterStore.decrement}>Decrement</button>
    </div>
  );
});

// Todo Component
const TodoComponent: React.FC = observer(() => {
  const todoStore = useTodoStore();

  const handleAddTodo = () => {
    const todo = prompt('Enter todo:');
    if (todo) {
      todoStore.addTodo(todo);
    }
  };

  const handleRemoveTodo = (index: number) => {
    todoStore.removeTodo(index);
  };

  return (
    <div>
```

```
      <h1>Todos</h1>
      <button onClick={handleAddTodo}>Add Todo</button>
      <ul>
        {todoStore.todos.map((todo, index) => (
          <li key={index}>
            {todo} <button onClick={() => handleRemoveTodo(index)}>Remove</button>
          </li>
        ))}
      </ul>
    </div>
  );
});

// App Component
const App: React.FC = () => {
  return (
    <CounterStoreContext.Provider value={counterStore}>
      <TodoStoreContext.Provider value={todoStore}>
        <div>
          <CounterComponent />
          <TodoComponent />
        </div>
      </TodoStoreContext.Provider>
    </CounterStoreContext.Provider>
  );
};

export default App;
```

In this example:

- Each store is defined as a class with observable properties and actions. We use `makeAutoObservable` to simplify the observable setup.
- We create instances of the stores (`counterStore` and `todoStore`).
- We define React contexts for each store (`CounterStoreContext` and `TodoStoreContext`).
- Custom hooks (`useCounterStore` and `useTodoStore`) are created to access the respective stores within components. These hooks provide type safety and throw errors if used outside of the correct context.

- Components (`CounterComponent` and `TodoComponent`) use the custom hooks to access and interact with the stores.
- The `App` component wraps the entire application with the context providers, making the stores available to all components.

This example demonstrates a clean and organized approach to managing multiple MobX stores in a React TypeScript application while adhering to best practices.

what is prefer to make 1 rootstore derived of all store or many separate store react ts mobx

The decision to use a single root store or multiple separate stores in a React TypeScript MobX application depends on various factors, including the complexity of your application, the organization of your data, and your personal preference. Both approaches have their advantages and disadvantages:

Single Root Store:

- **Centralized State Management:** With a single root store, you have a centralized place to manage all your application state. This can make it easier to maintain and reason about your application's data flow.
- **Simplified Dependency Injection:** You only need to inject one store into your components, which can simplify component setup and reduce boilerplate.
- **Cross-State Interactions:** If your application requires interactions between different parts of the state, having a single root store can make it easier to handle these interactions.

Multiple Separate Stores:

- **Modularity:** Multiple stores allow for better modularity and encapsulation. Each store can focus on a specific domain or feature of your application, making it easier to manage and reason about.
- **Reduced Coupling:** With separate stores, you can reduce coupling between different parts of your application. Changes to one store are less likely to impact other parts of the application, leading to better maintainability.
- **Scalability:** As your application grows, having separate stores can make it easier to scale. You can add or modify stores without affecting the rest of your application.

Recommendation:

- **Start Simple:** If you're unsure, start with a single root store. It's easier to refactor later if needed.
- **Consider Complexity:** If your application is relatively small and straightforward, a single root store may suffice. However, as your application grows, consider breaking down your state into multiple stores for better organization and maintainability.
- **Domain Driven Design:** If your application has clear domains or features that can be encapsulated separately, consider using multiple stores.
- **Experiment:** Ultimately, the best approach depends on your specific use case and preferences. Experiment with both approaches to see which one works best for your project.

In summary, there's no one-size-fits-all answer. Both single root store and multiple separate stores are valid approaches, and the choice depends on the specific requirements and structure of your application.

what is more efficient to be render mobx or contexapi in react

When comparing MobX and React Context API in terms of efficiency, it's important to understand that they serve different purposes and have different performance characteristics. Let's break down their differences and considerations for efficiency:

MobX:

- **Efficiency:** MobX is generally efficient because it utilizes reactive programming principles. It automatically tracks dependencies between observables and updates components only when necessary.
- **Granularity:** MobX allows for fine-grained reactivity, meaning only components that are directly affected by changes in observables will re-render. This can lead to optimal performance in large applications with complex state structures.
- **Simplicity:** MobX simplifies state management by automatically managing reactivity and providing a clear and intuitive API for defining observables, actions, and reactions.

React Context API:

- **Efficiency:** The React Context API provides a way to pass data through the component tree without having to pass props down manually at every level. It's efficient for sharing data that needs to be accessed by many components without the need for prop drilling.
- **Granularity:** Context updates trigger re-renders in all components that consume the context, regardless of whether they directly use the updated data. This can potentially lead to unnecessary re-renders in large component trees.
- **Complexity:** While the Context API simplifies prop management and sharing state across components, it may introduce additional complexity when used in large applications, especially if the context provider value changes frequently.

Considerations:

1. **Use Case:** Consider the specific use case and requirements of your application. If you need fine-grained reactivity and complex state management, MobX may be more suitable. If you need to share data across many components without prop drilling, the Context API may be preferable.
2. **Component Structure:** Evaluate the structure of your component tree. If you have a deeply nested component tree with components consuming the same data at various levels, the Context API may reduce prop drilling and improve code readability.
3. **Performance Profiling:** Profile the performance of your application using tools like React DevTools and Chrome DevTools to identify any performance bottlenecks. This will help you determine whether MobX or the Context API is more efficient in your specific use case.
4. **Developer Familiarity:** Consider the familiarity of your development team with each approach. Choose the one that aligns with your team's expertise and preferences to maintain code consistency and productivity.

In summary, both MobX and the React Context API have their strengths and weaknesses in terms of efficiency. The choice between them depends on the specific requirements of your application, the structure of your component tree, and the familiarity of your development team with each approach.

how to use mobx in react

2/2

To use MobX in a React application, you typically follow these steps:

1. **Install MobX and MobX React**: First, you need to install the necessary packages. You can do this using npm or yarn.

   ```bash
   npm install mobx mobx-react
   ```

   or

   ```bash
   yarn add mobx mobx-react
   ```

2. **Create MobX Store(s)**: Define MobX stores to manage your application state. Stores should contain observable properties and actions to update those properties. You can use MobX decorators (`@observable` , `@action` , etc.) or `makeObservable` to define observables and actions.
3. **Inject Stores into Components**: Inject MobX stores into your React components. You can do this using the `useContext` hook and React context or by using the `inject` higher-order component provided by `mobx-react` .
4. **Make Components Observable**: Make your React components observable using the `observer` function provided by `mobx-react` . This allows your components to reactively update when the observed data changes.
5. **Access Observables and Actions**: Access observable properties and actions from MobX stores in your components. Use them to read and update the application state as needed.

Here's a basic example to illustrate these steps:

```jsx
// CounterStore.js
import { makeObservable, observable, action } from 'mobx';

class CounterStore {
  count = 0;

  constructor() {
    makeObservable(this, {
```

```
      count: observable,
      increment: action,
      decrement: action,
    });
  }

  increment() {
    this.count++;
  }

  decrement() {
    this.count--;
  }
}

export default new CounterStore();

// Counter.js
import React from 'react';
import { observer } from 'mobx-react';
import counterStore from './CounterStore';

const Counter = observer(() => {
  const { count, increment, decrement } = counterStore;

  return (
    <div>
      <h1>Counter: {count}</h1>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
});

export default Counter;

// App.js
import React from 'react';
import Counter from './Counter';

const App = () => {
  return (
    <div>
      <Counter />
    </div>
  );
};

export default App;
```

In this example:

- We define a `CounterStore` class as a MobX store to manage the count state.
- We create a `Counter` component that observes changes to the count state using the `observer` function.
- The `Counter` component reads the count value from the store and provides buttons to increment and decrement the count.
- The `App` component renders the `Counter` component.

This is a basic example, but it demonstrates the fundamental concepts of using MobX with React. You can extend this pattern to manage more complex application state using multiple stores and actions.

methods in mobx in react

In MobX, you typically define methods within MobX stores to interact with and modify the observable state. There are several decorators provided by MobX to define these methods. Here are the commonly used ones:

1. **@observable**: Marks a property as observable. MobX will automatically track changes to this property and trigger re-renders in components that observe it.
2. **@action**: Marks a method as an action. Actions are the only way to modify observable state in MobX. MobX ensures that changes to observables only occur within actions, which helps maintain consistency and predictability.
3. **@computed**: Marks a getter method as a computed value. Computed values are derived from observable state and are automatically updated whenever their dependencies change. They are memoized, meaning MobX only recomputes them when necessary.
4. **@autorun**: Runs a function automatically whenever its dependencies change. This is useful for side effects that don't modify observable state directly, such as logging or

network requests.

Here's an example demonstrating these decorators in a MobX store:

```javascript
import { makeObservable, observable, action, computed } from 'mobx';

class CounterStore {
  count = 0;

  constructor() {
    makeObservable(this, {
      count: observable,
      increment: action,
      decrement: action,
      doubleCount: computed,
    });
  }

  increment() {
    this.count++;
  }

  decrement() {
    this.count--;
  }

  get doubleCount() {
    return this.count * 2;
  }
}

export default new CounterStore();
```

In this example:

- We define a `CounterStore` class as a MobX store.
- The `count` property is marked as observable using the `@observable` decorator.
- The `increment` and `decrement` methods are marked as actions using the `@action` decorator.

- The `doubleCount` getter method is marked as computed using the `@computed` decorator. It returns the value of `count` multiplied by 2.

By using these decorators, MobX ensures that the observable state is updated and accessed in a predictable and efficient manner, making it easier to manage state in React applications.

example loggin with mobx react ts

Logging in a React TypeScript application with MobX typically involves managing user authentication state and providing methods to authenticate users, log them in, log them out, and track their session. Here's a basic example of how you might implement this:

tsx

```tsx
// authStore.ts
import { makeAutoObservable } from 'mobx';

class AuthStore {
  isAuthenticated = false;

  constructor() {
    makeAutoObservable(this);
  }

  login = () => {
    // Simulate a login request (e.g., calling an API)
    setTimeout(() => {
      // Assuming login is successful
      this.isAuthenticated = true;
    }, 1000);
  };

  logout = () => {
    // Simulate a logout request (e.g., calling an API)
    setTimeout(() => {
      // Assuming logout is successful
      this.isAuthenticated = false;
    }, 1000);
  };
}

export default new AuthStore();
```

```
// Login.tsx
import React from 'react';
import { observer } from 'mobx-react-lite';
import authStore from './authStore';

const Login: React.FC = observer(() => {
  const handleLogin = () => {
    authStore.login();
  };

  return (
    <div>
      <h1>Login</h1>
      <button onClick={handleLogin}>Login</button>
    </div>
  );
});

export default Login;

// Logout.tsx
import React from 'react';
import { observer } from 'mobx-react-lite';
import authStore from './authStore';

const Logout: React.FC = observer(() => {
  const handleLogout = () => {
    authStore.logout();
  };

  return (
    <div>
      <h1>Logout</h1>
      <button onClick={handleLogout}>Logout</button>
    </div>
  );
});

export default Logout;

// App.tsx
```

```
import React from 'react';
import { observer } from 'mobx-react-lite';
import authStore from './authStore';
import Login from './Login';
import Logout from './Logout';

const App: React.FC = observer(() => {
  return (
    <div>
      {authStore.isAuthenticated ? (
        <div>
          <h1>Welcome, User!</h1>
          <Logout />
        </div>
      ) : (
        <div>
          <h1>Please login</h1>
          <Login />
        </div>
      )}
    </div>
  );
});

export default App;
```

In this example:

- We define an `AuthStore` class as a MobX store to manage authentication state.
- The store has an `isAuthenticated` observable property and `login` and `logout` actions to manage authentication status.
- The `Login` component allows users to log in by calling the `login` action from the `AuthStore`.
- The `Logout` component allows authenticated users to log out by calling the `logout` action from the `AuthStore`.
- The `App` component renders either the `Login` or `Logout` component based on the value of `isAuthenticated` from the `AuthStore`.

This is a basic example of logging in with MobX in a React TypeScript application. Depending on your specific requirements, you may need to expand on this implementation to include features such as user authentication tokens, user data storage, or authentication error handling.

example login with @autorun mobx   react ts

2/2

Using `@autorun` in MobX is typically not the preferred way to handle login functionality, as `@autorun` is mainly used for side effects that do not directly modify observable state. However, for the sake of demonstration, here's an example of how you could use `@autorun` for login functionality in a React TypeScript application with MobX:

tsx

```
// authStore.ts
import { makeAutoObservable } from 'mobx';

class AuthStore {
  isAuthenticated = false;

  constructor() {
    makeAutoObservable(this);
    this.autoLogin();
  }

  autoLogin = () => {
    // Simulate auto-login logic (e.g., checking local storage, cookies, etc.)
    const token = localStorage.getItem('token');
    if (token) {
      this.isAuthenticated = true;
    }
  };

  login = () => {
    // Simulate a login request (e.g., calling an API)
    setTimeout(() => {
      // Assuming login is successful
      this.isAuthenticated = true;
      // Save authentication token to local storage
      localStorage.setItem('token', 'dummyToken');
    }, 1000);
  };

  logout = () => {
    // Simulate a logout request (e.g., calling an API)
    setTimeout(() => {
      // Assuming logout is successful
      this.isAuthenticated = false;
      // Remove authentication token from local storage
      localStorage.removeItem('token');
    }, 1000);
  };
}

export default new AuthStore();

// Login.tsx
import React from 'react';
import { observer } from 'mobx-react-lite';
import authStore from './authStore';

const Login: React.FC = observer(() => {
  const handleLogin = () => {
    authStore.login();
  };

  return (
    <div>
      <h1>Login</h1>
      <button onClick={handleLogin}>Login</button>
    </div>
  );
});

export default Login;

// Logout.tsx
import React from 'react';
import { observer } from 'mobx-react-lite';
import authStore from './authStore';
```

```
const Logout: React.FC = observer(() => {
  const handleLogout = () => {
    authStore.logout();
  };

  return (
    <div>
      <h1>Logout</h1>
      <button onClick={handleLogout}>Logout</button>
    </div>
  );
});

export default Logout;

// App.tsx
import React from 'react';
import { observer } from 'mobx-react-lite';
import authStore from './authStore';
import Login from './Login';
import Logout from './Logout';

const App: React.FC = observer(() => {
  return (
    <div>
      {authStore.isAuthenticated ? (
        <div>
          <h1>Welcome, User!</h1>
          <Logout />
        </div>
      ) : (
        <div>
          <h1>Please login</h1>
          <Login />
        </div>
      )}
    </div>
  );
});

export default App;
```

In this example:

- We define an `AuthStore` class as a MobX store to manage authentication state.
- The `autoLogin` method checks for a stored authentication token (e.g., in local storage) and automatically logs in the user if a token is found.
- We use `@autorun` in the constructor of `AuthStore` to automatically call the `autoLogin` method when the store is initialized.
- The rest of the components (`Login`, `Logout`, and `App`) are similar to the previous example.

While using `@autorun` for this purpose is feasible, it's generally recommended to use `@action` for methods that directly modify observable state, like login and logout actions. This provides better clarity and maintainability in your codebase.

full example of @autorun in mobx react ts

2/2

Sure, here's a full example demonstrating the usage of `@autorun` in a MobX React TypeScript application:

tsx

```tsx
// authStore.ts
import { makeAutoObservable } from 'mobx';

class AuthStore {
  isAuthenticated = false;

  constructor() {
    makeAutoObservable(this);
    this.autoLogin();
  }

  autoLogin = () => {
    // Simulate auto-login logic (e.g., checking local storage, cookies, etc.)
    const token = localStorage.getItem('token');
    if (token) {
      this.isAuthenticated = true;
    }
  };
  login = () => {
    // Simulate a login request (e.g., calling an API)
    setTimeout(() => {
      // Assuming login is successful
      this.isAuthenticated = true;
      // Save authentication token to local storage
      localStorage.setItem('token', 'dummyToken');
    }, 1000);
  };

  logout = () => {
    // Simulate a logout request (e.g., calling an API)
    setTimeout(() => {
      // Assuming logout is successful
      this.isAuthenticated = false;
      // Remove authentication token from local storage
      localStorage.removeItem('token');
    }, 1000);
  };
}

export default new AuthStore();

// Login.tsx
import React from 'react';
import { observer } from 'mobx-react-lite';
import authStore from './authStore';

const Login: React.FC = observer(() => {
  const handleLogin = () => {
    authStore.login();
  };

  return (
    <div>
      <h1>Login</h1>
      <button onClick={handleLogin}>Login</button>
    </div>
  );
});
```

```
export default Login;

// Logout.tsx
import React from 'react';
import { observer } from 'mobx-react-lite';
import authStore from './authStore';

const Logout: React.FC = observer(() => {
  const handleLogout = () => {
    authStore.logout();
  };

  return (
    <div>
      <h1>Logout</h1>
      <button onClick={handleLogout}>Logout</button>
    </div>
  );
});

export default Logout;

// App.tsx
import React from 'react';
import { observer } from 'mobx-react-lite';
import authStore from './authStore';
import Login from './Login';
import Logout from './Logout';

const App: React.FC = observer(() => {
  return (
    <div>
      {authStore.isAuthenticated ? (
        <div>
          <h1>Welcome, User!</h1>
          <Logout />
        </div>
      ) : (
        <div>
          <h1>Please login</h1>
          <Login />
        </div>
      )}
    </div>
  );
});

export default App;
```

In this example:

- We have an `AuthStore` class to manage authentication state.
- `autoLogin` is a method that automatically logs in the user based on stored authentication data, such as a token in local storage.
- We use `@autorun` in the constructor of `AuthStore` to automatically call `autoLogin` when the store is initialized.
- The rest of the components (`Login`, `Logout`, and `App`) handle the UI and user interactions. They interact with `AuthStore` to manage the authentication state.

This example demonstrates how to use `@autorun` to automatically perform side effects (in this case, auto-login) when the store is initialized. However, keep in mind that `@autorun` is typically used for non-state-modifying side effects and is less common for managing state changes. In this specific scenario, using `@autorun` is appropriate because it's triggered once and doesn't need to be re-triggered when observable state changes.

the correct way to do login in react ts and mobx full example

2/2

Certainly! Here's a full example demonstrating the correct way to implement login functionality in a React TypeScript application using MobX:

```tsx
// authStore.ts
import { makeAutoObservable } from 'mobx';

class AuthStore {
  isAuthenticated = false;

  constructor() {
    makeAutoObservable(this);
  }

  login = async (username: string, password: string) => {
    try {
      // Simulate login request (e.g., calling an API)
      const response = await fetch('https://api.example.com/login', {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
        },
        body: JSON.stringify({ username, password }),
      });

      // Assuming login is successful if response status is 200
      if (response.status === 200) {
        this.isAuthenticated = true;
        // You can handle storing authentication token or user data here
        localStorage.setItem('token', 'dummyToken');
        return true;
      } else {
        return false;
      }
    } catch (error) {
      console.error('Error during login:', error);
      return false;
    }
  };

  logout = () => {
    // Clear authentication token or user data from storage
    localStorage.removeItem('token');
    this.isAuthenticated = false;
  };
}

export default new AuthStore();

// Login.tsx
import React, { useState } from 'react';
import { observer } from 'mobx-react-lite';
import authStore from './authStore';
```

```
const Login: React.FC = observer(() => {
  const [username, setUsername] = useState('');
  const [password, setPassword] = useState('');
  const [error, setError] = useState('');

  const handleLogin = async () => {
    setError('');
    const success = await authStore.login(username, password);
    if (!success) {
      setError('Invalid username or password');
    }
  };

  return (
    <div>
      <h1>Login</h1>
      <input
        type="text"
        placeholder="Username"
        value={username}
        onChange={(e) => setUsername(e.target.value)}
      />
      <input
        type="password"
        placeholder="Password"
        value={password}
        onChange={(e) => setPassword(e.target.value)}
      />
      <button onClick={handleLogin}>Login</button>
      {error && <p>{error}</p>}
    </div>
  );
});

export default Login;

// Logout.tsx
import React from 'react';
import { observer } from 'mobx-react-lite';
import authStore from './authStore';
```

```
const Logout: React.FC = observer(() => {
  const handleLogout = () => {
    authStore.logout();
  };

  return (
    <div>
      <h1>Logout</h1>
      <button onClick={handleLogout}>Logout</button>
    </div>
  );
});

export default Logout;

// App.tsx
import React from 'react';
import { observer } from 'mobx-react-lite';
import authStore from './authStore';
import Login from './Login';
import Logout from './Logout';

const App: React.FC = observer(() => {
  return (
    <div>
      {authStore.isAuthenticated ? (
        <div>
          <h1>Welcome, User!</h1>
          <Logout />
        </div>
      ) : (
        <Login />
      )}
    </div>
  );
});

export default App;
```

In this example:

- `AuthStore` manages the authentication state and provides methods for login and logout.
- `Login` component allows users to enter their credentials and attempt to log in. Upon successful login, it sets the `isAuthenticated` flag in `AuthStore`.
- `Logout` component allows authenticated users to log out by calling the `logout` method from `AuthStore`.
- `App` component renders either the `Login` or `Logout` component based on the value of `isAuthenticated` from `AuthStore`.

This example demonstrates a complete login functionality in a React TypeScript application using MobX. It handles user input, login requests, authentication state management, and conditional rendering of components based on authentication status. Additionally, it includes error handling for invalid login attempts.

the correct way to do auto login and login in react ts and mobx full example with cokies

2/2

Sure, here's a full example demonstrating the correct way to implement auto-login and login functionality in a React TypeScript application using MobX and cookies:

```tsx
// authStore.ts
import { makeAutoObservable } from 'mobx';

class AuthStore {
  isAuthenticated = false;

  constructor() {
    makeAutoObservable(this);
    this.autoLogin();
  }

  autoLogin = () => {
    // Check if authentication token exists in cookies
    const token = this.getCookie('token');
    if (token) {
      this.isAuthenticated = true;
    }
  };

  login = async (username: string, password: string) => {
    try {
      // Simulate login request (e.g., calling an API)
      const response = await fetch('https://api.example.com/login', {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
        },
        body: JSON.stringify({ username, password }),
      });

      // Assuming login is successful if response status is 200
      if (response.status === 200) {
        this.isAuthenticated = true;
        // Set authentication token in cookies
        this.setCookie('token', 'dummyToken', 7); // Expires in 7 days
        return true;
      } else {
        return false;
      }
    } catch (error) {
      console.error('Error during login:', error);
      return false;
    }
  };

  logout = () => {
    // Clear authentication token from cookies
    this.deleteCookie('token');
    this.isAuthenticated = false;
  };

  // Helper functions for working with cookies
  private getCookie = (name: string) => {
    const value = `; ${document.cookie}`;
    const parts = value.split(`; ${name}=`);
    if (parts.length === 2) return parts.pop()?.split(';').shift();
  };

  private setCookie = (name: string, value: string, days: number) => {
    const date = new Date();
    date.setTime(date.getTime() + days * 24 * 60 * 60 * 1000);
    const expires = `expires=${date.toUTCString()}`;
    document.cookie = `${name}=${value};${expires};path=/`;
  };

  private deleteCookie = (name: string) => {
    document.cookie = `${name}=;expires=Thu, 01 Jan 1970 00:00:00 UTC;path=/;`;
  };
}

export default new AuthStore();

// Login.tsx
import React, { useState } from 'react';
import { observer } from 'mobx-react-lite';
import authStore from './authStore';

const Login: React.FC = observer(() => {
  const [username, setUsername] = useState('');
  const [password, setPassword] = useState('');
  const [error, setError] = useState('');

  const handleLogin = async () => {
    setError('');
    const success = await authStore.login(username, password);
    if (!success) {
      setError('Invalid username or password');
    }
  };

  return (
    <div>
      <h1>Login</h1>
      <input
        type="text"
        placeholder="Username"
        value={username}
        onChange={(e) => setUsername(e.target.value)}
      />
      <input
        type="password"
```

```
          placeholder="Password"
          value={password}
          onChange={(e) => setPassword(e.target.value)}
        />
        <button onClick={handleLogin}>Login</button>
        {error && <p>{error}</p>}
      </div>
    );
  });

  export default Login;

  // Logout.tsx
  import React from 'react';
  import { observer } from 'mobx-react-lite';
  import authStore from './authStore';

  const Logout: React.FC = observer(() => {
    const handleLogout = () => {
      authStore.logout();
    };

    return (
      <div>
        <h1>Logout</h1>
        <button onClick={handleLogout}>Logout</button>
      </div>
    );
  });

  export default Logout;

  // App.tsx
  import React from 'react';
  import { observer } from 'mobx-react-lite';
  import authStore from './authStore';
  import Login from './Login';
  import Logout from './Logout';

  const App: React.FC = observer(() => {
    return (
      <div>
        {authStore.isAuthenticated ? (
          <div>
            <h1>Welcome, User!</h1>
            <Logout />
          </div>
        ) : (
          <Login />
        )}
      </div>
    );
  });

  export default App;
```

In this example:

- `AuthStore` manages the authentication state and provides methods for login and logout. It also includes methods for working with cookies.
- `autoLogin` method automatically logs in the user if a valid authentication token is found in cookies.
- `login` method sends a login request to the server, sets the authentication token in cookies upon successful login, and updates the `isAuthenticated` flag.
- `logout` method clears the authentication token from cookies and resets the `isAuthenticated` flag.
- `Login` component allows users to enter their credentials and attempt to log in. Upon successful login, it sets the authentication token in cookies and updates the authentication state.
- `Logout` component allows authenticated users to log out by calling the `logout` method from `AuthStore`.
- `App` component renders either the `Login` or `Logout` component based on the value of `isAuthenticated` from `AuthStore`.

This example demonstrates a complete login functionality in a React TypeScript application using MobX and cookies. It handles user input, login requests, authentication state management, and conditional rendering of components based on authentication status. Additionally, it includes error handling for invalid login attempts.

correct way autologin in nodejs and react ts and mobx full example with cokies

2/2

To implement auto-login functionality in a React TypeScript application with MobX and cookies, you typically need a backend server to handle authentication and store authentication tokens in cookies. Below is an example demonstrating how to implement auto-login using Node.js for the backend and React TypeScript with MobX for the frontend:

Backend (Node.js with Express):

```javascript
const express = require('express');
const cookieParser = require('cookie-parser');
const jwt = require('jsonwebtoken');

const app = express();
app.use(express.json());
app.use(cookieParser());

const secretKey = 'your_secret_key';

// Mock user data
const users = [
  { id: 1, username: 'user1', password: 'password1' },
  { id: 2, username: 'user2', password: 'password2' },
];

// Login endpoint
app.post('/login', (req, res) => {
  const { username, password } = req.body;
  const user = users.find(u => u.username === username && u.password === password);
  if (!user) {
    return res.status(401).json({ message: 'Invalid username or password' });
  }

  const token = jwt.sign({ userId: user.id }, secretKey, { expiresIn: '7d' });
  res.cookie('token', token, { httpOnly: true });
  res.json({ message: 'Login successful', token });
});
```

```
// Logout endpoint
app.post('/logout', (req, res) => {
  res.clearCookie('token');
  res.json({ message: 'Logout successful' });
});

// Validate token middleware
const authenticateToken = (req, res, next) => {
  const token = req.cookies.token;
  if (!token) {
    return res.status(401).json({ message: 'Unauthorized' });
  }

  jwt.verify(token, secretKey, (err, decoded) => {
    if (err) {
      return res.status(403).json({ message: 'Forbidden' });
    }
    req.userId = decoded.userId;
    next();
  });
};

// Protected route
app.get('/user', authenticateToken, (req, res) => {
  const user = users.find(u => u.id === req.userId);
  if (!user) {
    return res.status(404).json({ message: 'User not found' });
  }
  res.json({ id: user.id, username: user.username });
});

app.listen(3001, () => {
  console.log('Server is running on port 3001');
});
```

Frontend (React TypeScript with MobX):

tsx

```tsx
// authStore.ts
import { makeAutoObservable } from 'mobx';
import axios from 'axios';
```

```
class AuthStore {
  isAuthenticated = false;

  constructor() {
    makeAutoObservable(this);
    this.autoLogin();
  }

  autoLogin = async () => {
    try {
      const response = await axios.get('/user');
      if (response.status === 200) {
        this.isAuthenticated = true;
      }
    } catch (error) {
      console.error('Error during auto-login:', error);
    }
  };

  login = async (username: string, password: string) => {
    try {
      const response = await axios.post('/login', { username, password });
      if (response.status === 200) {
        this.isAuthenticated = true;
      }
    } catch (error) {
      console.error('Error during login:', error);
      throw error;
    }
  };

  logout = async () => {
    try {
      await axios.post('/logout');
      this.isAuthenticated = false;
    } catch (error) {
      console.error('Error during logout:', error);
      throw error;
    }
  };
}
```

```
export default new AuthStore();

// Login.tsx
import React, { useState } from 'react';
import { observer } from 'mobx-react-lite';
import authStore from './authStore';

const Login: React.FC = observer(() => {
  const [username, setUsername] = useState('');
  const [password, setPassword] = useState('');
  const [error, setError] = useState('');

  const handleLogin = async () => {
    setError('');
    try {
      await authStore.login(username, password);
    } catch (error) {
      setError('Invalid username or password');
    }
  };

  return (
    <div>
      <h1>Login</h1>
      <input
        type="text"
        placeholder="Username"
        value={username}
        onChange={(e) => setUsername(e.target.value)}
      />
      <input
        type="password"
        placeholder="Password"
        value={password}
        onChange={(e) => setPassword(e.target.value)}
      />
      <button onClick={handleLogin}>Login</button>
      {error && <p>{error}</p>}
    </div>
  );
});
```

```
export default Login;

// Logout.tsx
import React from 'react';
import { observer } from 'mobx-react-lite';
import authStore from './authStore';

const Logout: React.FC = observer(() => {
  const handleLogout = async () => {
    try {
      await authStore.logout();
    } catch (error) {
      console.error('Error during logout:', error);
    }
  };

  return (
    <div>
      <h1>Logout</h1>
      <button onClick={handleLogout}>Logout</button>
    </div>
  );
});

export default Logout;

// App.tsx
import React from 'react';
import { observer } from 'mobx-react-lite';
import authStore from './authStore';
import Login from './Login';
import Logout from './Logout';

const App: React.FC = observer(() => {
  return (
    <div>
      {authStore.isAuthenticated ? (
        <div>
          <h1>Welcome, User!</h1>
          <Logout />
        </div>
      ) : (
        <Login />
      )}
    </div>
  );
});

export default App;
```

In this example:

- The backend server (Node.js with Express) handles user authentication using JWT tokens stored in cookies.
- The frontend application (React TypeScript with MobX) includes login and logout functionality. The `AuthStore` class manages the authentication state and makes requests to the backend server for authentication.
- When the frontend application loads, it automatically tries to login the user based on the presence of a valid token in cookies.
- The `Login` component allows users to enter their credentials and attempt to log in. Upon successful login, it sets the authentication token in cookies and updates the authentication state.
- The `Logout` component allows authenticated users to log out by clearing the authentication token from cookies and updating the authentication state.
- The `App` component renders either the `Login` or `Logout` component based on the authentication status stored

how to modify the user input value with react ts

3/3

To modify user input values in a React TypeScript application, you typically use state variables and event handlers to manage the input values and update them as the user interacts with the input fields. Here's how you can achieve this:

```tsx
import React, { useState, ChangeEvent } from 'react';

const MyComponent: React.FC = () => {
  // Define state variables to store input values
  const [textInput, setTextInput] = useState('');
  const [checkboxChecked, setCheckboxChecked] = useState(false);

  // Event handler for text input changes
  const handleTextInputChange = (event: ChangeEvent<HTMLInputElement>) => {
    setTextInput(event.target.value);
  };

  // Event handler for checkbox changes
  const handleCheckboxChange = (event: ChangeEvent<HTMLInputElement>) => {
    setCheckboxChecked(event.target.checked);
  };

  return (
    <div>
      {/* Text input */}
      <input
        type="text"
        value={textInput} // Bind input value to state variable
        onChange={handleTextInputChange} // Call event handler on input change
      />

      {/* Checkbox input */}
      <input
        type="checkbox"
        checked={checkboxChecked} // Bind checkbox value to state variable
        onChange={handleCheckboxChange} // Call event handler on checkbox change
      />
    </div>
  );
};

export default MyComponent;
```

In this example:

- We use the `useState` hook to define state variables (`textInput` and `checkboxChecked`) to store the input values.
- We define event handlers (`handleTextInputChange` and `handleCheckboxChange`) to update the state variables when the input values change.

- We bind the value of each input field to the corresponding state variable (`value={textInput}` and `checked={checkboxChecked}`).
- We provide event handlers (`onChange`) to listen for changes to the input fields, triggering the corresponding state updates.
- As the user types into the text input or checks/unchecks the checkbox, the state variables are updated accordingly, and the input values are reflected in the UI.

This pattern allows you to easily manage and update user input values in React TypeScript applications.