

how to create react application with typescript

To create a React application with TypeScript, you can use `create-react-app`, a popular tool that sets up a new React project with all the necessary configurations for TypeScript. Follow these steps to create your React application:

Step 1: Install Node.js and npm

Make sure you have Node.js and npm (Node Package Manager) installed on your system. You can download and install them from the official Node.js website: <https://nodejs.org/>

Step 2: Create a new React application with TypeScript

Open your terminal or command prompt, and run the following command to create a new React application using `create-react-app` with TypeScript template:

```
```bash
npx create-react-app my-app --template typescript
```
```

Replace `my-app` with the desired name of your application. This command will set up a new React project with TypeScript template.

Step 3: Navigate to the project directory

Move into the project directory that was just created:

```
```bash
cd my-app
```
```

Step 4: Start the development server

To see your React application in action, run the following command:

```
```bash
npm start
```
```

...

This will start the development server and open your React application in your default web browser. The application will automatically reload whenever you make changes to the code.

Step 5: Explore the project structure

Your React application with TypeScript is now set up. You can explore the project's file structure inside the ``my-app`` folder. Key files and directories include:

- ``src``: This folder contains the main source code of your application.
- ``src/index.tsx``: The entry point of your application, where the React root component is rendered.
- ``src/App.tsx``: The default component that is rendered in the application. You can modify this or create new components as needed.
- ``tsconfig.json``: The TypeScript configuration file. You can adjust TypeScript settings in this file.

Step 6: Make changes and build

Now you can start building your React application with TypeScript. Edit the existing components or create new ones in the ``src`` folder.

When you are ready to deploy your application, you can use the following command to create a production build:

```
```bash
npm run build
```
```

The production build will be created in the ``build`` folder.

That's it! You now have a React application with TypeScript up and running. Happy coding!

write a program to create a simple count increment and decrement using react js with typescript

To create a simple count increment and decrement using ReactJS with TypeScript, you'll need to set up a new React project and then create the necessary components. Follow these steps:

Step 1: Set up the React project

Make sure you have Node.js and npm (Node Package Manager) installed on your system. If not, download and install them from the official Node.js website.

Open your terminal or command prompt and run the following commands to create a new React project:

```
``bash
npx create-react-app count-increment-decrement --template typescript
cd count-increment-decrement
``
```

Step 2: Create the counter component

Inside the `src` folder, create a new file named `Counter.tsx`. This file will contain the code for our count increment and decrement component.

```
``tsx
// src/Counter.tsx

import React, { useState } from 'react';

const Counter: React.FC = () => {
  const [count, setCount] = useState<number>(0);

  const increment = () => {
    setCount((prevCount) => prevCount + 1);
  };

  const decrement = () => {
```

```

    setCount((prevCount) => prevCount - 1);
  };

  return (
    <div>
      <h2>Count: {count}</h2>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
};

export default Counter;
...

```

Step 3: Update the App component

Open the `src/App.tsx` file and import the `Counter` component. Replace the default content of the `App` component with the `Counter` component.

```

```tsx
// src/App.tsx
import React from 'react';
import './App.css';
import Counter from './Counter';

function App() {
 return (
 <div className="App">
 <header className="App-header">
 <h1>Simple Count Increment and Decrement</h1>
 <Counter />

```

```
 </header>

 </div>

);
}

export default App;
...

```

#### Step 4: Styling (Optional)

If you want, you can add some basic styling to the `App.css` file or use any CSS framework of your choice to make the UI more appealing.

#### Step 5: Run the application

Finally, save your changes and run the application using the following command:

```
```bash
npm start
...

```

Your React application with TypeScript should now be up and running, and you should see a simple counter with increment and decrement buttons in your browser. Clicking the buttons will increase or decrease the count, respectively.

write a reactjs typescript program form two way property binding where user should enter and that should display in div element

To implement a two-way property binding in ReactJS with TypeScript, you can create a simple form with an input field where the user can enter text, and the text will be displayed in a div element simultaneously. Here's how you can do it:

Step 1: Set up the React project

If you haven't set up the React project yet, follow the same steps mentioned in the previous response to create a new React project with TypeScript.

Step 2: Create the two-way binding component

Inside the `src` folder, create a new file named `TwoWayBinding.tsx`. This file will contain the code for our two-way property binding component.

```
```.tsx
// src/TwoWayBinding.tsx
import React, { useState } from 'react';

const TwoWayBinding: React.FC = () => {
 const [inputText, setInputText] = useState<string>("");

 const handleInputChange = (event: React.ChangeEvent<HTMLInputElement>) => {
 setInputText(event.target.value);
 };

 return (
 <div>
 <h2>Two-Way Property Binding Example</h2>
 <input type="text" value={inputText} onChange={handleInputChange} />
 <div>{inputText}</div>
 </div>
);
};

export default TwoWayBinding;
````
```

Step 3: Update the App component

Open the `src/App.tsx` file and import the `TwoWayBinding` component. Replace the default content of the `App` component with the `TwoWayBinding` component.

```
```tsx
// src/App.tsx
import React from 'react';
import './App.css';
import TwoWayBinding from './TwoWayBinding';

function App() {
 return (
 <div className="App">
 <header className="App-header">
 <TwoWayBinding />
 </header>
 </div>
);
}

export default App;
```
```

Step 4: Styling (Optional)

If you want, you can add some basic styling to the `App.css` file or use any CSS framework of your choice to make the UI more appealing.

Step 5: Run the application

Finally, save your changes and run the application using the following command:

```
```bash
npm start
```
```

Your React application with two-way property binding should now be up and running. When you enter text in the input field, it will simultaneously update the value in the div element below. This

demonstrates the two-way binding behavior, where changes in the input field are immediately reflected in the displayed content.

write the program "form validation for the input fields of username, email address, mobile number, password, checkbox with inline style error display" using reactjs with typescript

Sure, here's an example of a form validation program in ReactJS with TypeScript that validates the input fields for username, email address, mobile number, password, and a checkbox. We'll use inline style to display error messages for each input field.

First, make sure you have Node.js and npm (Node Package Manager) installed. Then, you can create a new React project using Create React App and add TypeScript support:

1. Create a new React project with TypeScript support:

```
```bash
npx create-react-app form-validation --template typescript
cd form-validation
```
```

2. Replace the content of `src/App.tsx` with the following code:

```
```tsx
import React, { useState } from 'react';
import './App.css';

const App: React.FC = () => {
 const [formData, setFormData] = useState({
 username: "",
 email: "",
 mobile: "",
 password: "",
 isChecked: false,
 });

```



```
});

const [errors, setErrors] = useState({
 username: "",
 email: "",
 mobile: "",
 password: "",
 isChecked: "",
});

const handleChange = (e: React.ChangeEvent<HTMLInputElement>) => {
 const { name, value, type, checked } = e.target;

 setFormData((prevData) => ({
 ...prevData,
 [name]: type === 'checkbox' ? checked : value,
 }));
};

const handleSubmit = (e: React.FormEvent<HTMLFormElement>) => {
 e.preventDefault();

 const newErrors: any = {};

 if (!formData.username) {
 newErrors.username = 'Username is required';
 }

 if (!formData.email) {
 newErrors.email = 'Email address is required';
 } else if (!isValidEmail(formData.email)) {
 newErrors.email = 'Invalid email address';
 }
};
```

```
}

if (!formData.mobile) {
 newErrors.mobile = 'Mobile number is required';
} else if (!isValidMobile(formData.mobile)) {
 newErrors.mobile = 'Invalid mobile number';
}

if (!formData.password) {
 newErrors.password = 'Password is required';
} else if (formData.password.length < 6) {
 newErrors.password = 'Password must be at least 6 characters long';
}

if (!formData.isChecked) {
 newErrors.isChecked = 'You must agree to the terms';
}

setErrors(newErrors);

// If there are no errors, you can submit the form here
// For this example, we'll just log the form data
if (Object.keys(newErrors).length === 0) {
 console.log('Form data:', formData);
}
};

const isValidEmail = (email: string) => {
 const emailPattern = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
 return emailPattern.test(email);
};
```

```
const isValidMobile = (mobile: string) => {
 const mobilePattern = /^[0-9]{10}$/;
 return mobilePattern.test(mobile);
};
```

```
return (
 <div className="App">
 <h1>Form Validation</h1>
 <form onSubmit={handleSubmit}>
 <div>
 <label htmlFor="username">Username</label>
 <input
 type="text"
 id="username"
 name="username"
 value={formData.username}
 onChange={handleChange}
 />
 {errors.username && {errors.username}}
 </div>

 <div>
 <label htmlFor="email">Email</label>
 <input
 type="email"
 id="email"
 name="email"
 value={formData.email}
 onChange={handleChange}
 />
 </div>
 </form>
 </div>
)
```

```
{errors.email && {errors.email}}
```

```
</div>
```

```
<div>
```

```
<label htmlFor="mobile">Mobile Number</label>
```

```
<input
```

```
 type="text"
```

```
 id="mobile"
```

```
 name="mobile"
```

```
 value={formData.mobile}
```

```
 onChange={handleChange}
```

```
{errors.mobile && {errors.mobile}}
```

```
</div>
```

```
<div>
```

```
<label htmlFor="password">Password</label>
```

```
<input
```

```
 type="password"
```

```
 id="password"
```

```
 name="password"
```

```
 value={formData.password}
```

```
 onChange={handleChange}
```

```
{errors.password && {errors.password}}
```

```
</div>
```

```
<div>
```

```
<input
```

```
 type="checkbox"
```

```
 id="isChecked"
```

```

 name="isChecked"
 checked={formData.isChecked}
 onChange={handleChange}
 />
 <label htmlFor="isChecked">I agree to the terms</label>
 {errors.isChecked && {errors.isChecked}}
 </div>

 <button type="submit">Submit</button>
</form>
</div>
);
};

export default App;

```

3. Create a file named `App.css` in the `src` folder with the following CSS styles:

```

``css

.App {
 display: flex;
 flex-direction: column;
 align-items: center;
 justify-content: center;
 height: 100vh;
}

form {
 display: flex;
 flex-direction: column;

```

```
align-items: center;
gap: 10px;
}

label {
 font-weight: bold;
}

.error {
 color: red;
}

button {
 margin-top: 10px;
}
...

```

#### 4. Run the application:

```
```bash
npm start
...

```

Now you should see a form with input fields for username, email address, mobile number, password, and a checkbox. If you try to submit the form without filling in the required fields or providing invalid input, the error messages will be displayed inline next to the respective input fields. Once the form is filled correctly, the form data will be logged to the console (in this example, we've simply logged the form data).

Remember that this example focuses on the frontend form validation using inline styles. In a real-world scenario, you should also perform backend validation to ensure data integrity and security.

update the above code while user entering in the input element itself it should check and validate and give either error message with styling or success with styling

Here's the updated code to resolve the error:

```
``tsx
import React, { useState } from 'react';
import './App.css';

interface FormData {
  username: string;
  email: string;
  mobile: string;
  password: string;
  isChecked: boolean;
}

interface FormErrors {
  username: string;
  email: string;
  mobile: string;
  password: string;
  isChecked: string;
}

const App: React.FC = () => {
  const [formData, setFormData] = useState<FormData>({
    username: "",
    email: "",
    mobile: "",
    password: "",
```

```
    isChecked: false,
  });

const [errors, setErrors] = useState<FormErrors>({
  username: "",
  email: "",
  mobile: "",
  password: "",
  isChecked: "",
});

const handleChange = (e: React.ChangeEvent<HTMLInputElement>) => {
  const { name, value, type, checked } = e.target;

  setFormData((prevData) => ({
    ...prevData,
    [name]: type === 'checkbox' ? checked : value,
  }));

  validateField(name as keyof FormData, type === 'checkbox' ? checked : value);
};

const validateField = (name: keyof FormData, value: string | boolean) => {
  const newErrors: FormErrors = { ...errors };

  switch (name) {
    case 'username':
      newErrors.username = value.toString().trim() === "" ? 'Username is required' : "";
      break;
    case 'email':
      newErrors.email = value.toString().trim() === "" ? 'Email address is required' : "";
```



```

        break;

    case 'mobile':

        newErrors.mobile =

            value.toString().trim() === '' ? 'Mobile number is required' : isValidMobile(value.toString()) ? '' :
'Invalid mobile number';

        break;

    case 'password':

        newErrors.password = value.toString().length < 6 ? 'Password must be at least 6 characters long' :
'';

        break;

    case 'isChecked':

        newErrors.isChecked = value ? '' : 'You must agree to the terms';

        break;

    default:

        break;

}

setErrors(newErrors);

};

const handleSubmit = (e: React.FormEvent<HTMLFormElement>) => {

    e.preventDefault();

    // Check for any remaining errors before submitting the form

    const newErrors: FormErrors = { ...errors };

    Object.keys(formData).forEach((name) => {

        validateField(name as keyof FormData, formData[name as keyof FormData]);

    });

    setErrors(newErrors);

```

```

// If there are no errors, you can submit the form here
// For this example, we'll just log the form data
if (Object.values(newErrors).every((error) => error === '')) {
  console.log('Form data:', formData);
}
};

const isValidEmail = (email: string) => {
  const emailPattern = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
  return emailPattern.test(email);
};

const isValidMobile = (mobile: string) => {
  const mobilePattern = /^[0-9]{10}$/;
  return mobilePattern.test(mobile);
};

return (
  // ... Remaining JSX ...
);
};

export default App;
...

```

By handling the different types in `handleChange` and `validateField`, TypeScript should no longer complain about the type mismatch, and the form should work correctly with real-time validation. The form will display validation messages and apply styles as the user enters data in each input field.

write the program for count incrementing and decrementing value using reactjs redux with typescript

To implement count incrementing and decrementing using React with Redux and TypeScript, follow these steps:

Step 1: Set up a new React project with TypeScript template:

```
```bash
npx create-react-app count-app --template typescript
cd count-app
...`
```

Step 2: Install the necessary dependencies:

```
```bash
npm install redux react-redux @types/react-redux
...`
```

Step 3: Create a new file `src/redux/actions.ts` for defining the actions:

```
```tsx
// src/redux/actions.ts
export enum ActionTypes {
 INCREMENT = 'INCREMENT',
 DECREMENT = 'DECREMENT',
}

export interface IncrementAction {
 type: ActionTypes.INCREMENT;
}

export interface DecrementAction {
 type: ActionTypes.DECREMENT;
}
```

```
}

export type Action = IncrementAction | DecrementAction;
...

```

Step 4: Create a new file `src/redux/reducers.ts` for defining the reducer:

```
``tsx
// src/redux/reducers.ts
import { ActionTypes, Action } from './actions';

export interface CountState {
 count: number;
}

const initialState: CountState = {
 count: 0,
};

export const countReducer = (state: CountState = initialState, action: Action): CountState => {
 switch (action.type) {
 case ActionTypes.INCREMENT:
 return {
 ...state,
 count: state.count + 1,
 };
 case ActionTypes.DECREMENT:
 return {
 ...state,
 count: state.count - 1,
 };
 }
}

```

```
 default:
 return state;
 }
};
...

```

Step 5: Create a new file `src/redux/store.ts` for setting up the Redux store:

```
```tsx
// src/redux/store.ts
import { createStore } from 'redux';
import { countReducer } from './reducers';

export const store = createStore(countReducer);
...

```

Step 6: Update the `src/App.tsx` file with the following code:

```
```tsx
// src/App.tsx
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
/* import { RootState } from './redux/store'; */
import { ActionTypes } from './redux/actions';
import './App.css';

const App: React.FC = () => {
 const count = useSelector((state: any) => state.count);
 const dispatch = useDispatch();

 const handleIncrement = () => {

```

```

 dispatch({ type: ActionTypes.INCREMENT });
 };

 const handleDecrement = () => {
 dispatch({ type: ActionTypes.DECREMENT });
 };

 return (
 <div className="App">
 <h1>Count: {count}</h1>
 <div className="buttons">
 <button onClick={handleIncrement}>Increment</button>
 <button onClick={handleDecrement}>Decrement</button>
 </div>
 </div>
);
};

export default App;

```

Step 7: Update the `src/App.css` file with some basic styling:

```

```css
/* src/App.css */

.App {
  text-align: center;
  margin-top: 50px;
}

.buttons {

```

```
margin-top: 20px;
}

button {
  margin: 5px;
  padding: 10px 20px;
  font-size: 16px;
  cursor: pointer;
}
...

```

Step 8: Modify `src/index.tsx` to wrap the `App` component with `Provider` to connect it with Redux:

```
```tsx
// src/index.tsx
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import { store } from '../redux/store';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

ReactDOM.render(
 <React.StrictMode>
 <Provider store={store}>
 <App />
 </Provider>
 </React.StrictMode>,
 document.getElementById('root')
);

```

```
reportWebVitals();
```

```
...
```

Step 9: Start the development server:

```
```bash
```

```
npm start
```

```
```
```

Now, you should see a simple app with a count value displayed on the screen and two buttons to increment and decrement the count. The app uses Redux to manage the state, and TypeScript is used for type-checking the components and actions. When you click the "Increment" or "Decrement" buttons, the count value will change accordingly.

### **write a program for fetching the data from the restapi and displaying using react js and typescript**

To fetch data from a REST API and display it using React.js and TypeScript, you'll need to set up a React project and use the `fetch` function or a library like `axios` to make the API calls. Here's a step-by-step guide to creating the program:

1. Set up the React.js project:

Ensure you have Node.js installed on your machine. Then, use `create-react-app` to set up a new React project:

```
```bash
```

```
npx create-react-app react-api-demo --template typescript
```

```
cd react-api-demo
```

```
```
```

2. Install necessary dependencies:



Install Axios to make API calls:

```
```bash
npm install axios
```
```

3. Create a component for fetching and displaying data:

Create a new file named `DataDisplay.tsx` in the `src` directory. This component will handle fetching data from the API and rendering it:

```
```tsx
import React, { useEffect, useState } from "react";
import axios from "axios";

interface DataItem {
  // Define the structure of your data item here
  id: number;
  name: string;
  // Add other properties as needed
}

const DataDisplay: React.FC = () => {
  const [data, setData] = useState<DataItem[]>([]);

  useEffect(() => {
    fetchData();
  }, []);

  const fetchData = async () => {
```

```

    try {
      const response = await axios.get("https://api.example.com/data");
      setData(response.data);
    } catch (error) {
      console.error("Error fetching data:", error);
    }
  };

  return (
    <div>
      <h1>Data from API:</h1>
      <ul>
        {data.map((item) => (
          <li key={item.id}>{item.name}</li>
          // Render other properties as needed
        ))}
      </ul>
    </div>
  );
};

export default DataDisplay;
...

```

4. Use the `DataDisplay` component in `App.tsx`:

Replace the contents of `App.tsx` with the following code:

```

````tsx
import React from "react";
import DataDisplay from "../DataDisplay";

```

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

```
export default App;
...
```

5. Run the application:

Start the development server and see the data being fetched and displayed:

```
```bash  
npm start  
...
```

The React application will be accessible at `http://localhost:3000`, and it will fetch data from the specified API and display it on the page.

Please note that you should replace `"https://api.example.com/data"` with the actual URL of the REST API you want to fetch data from, and adjust the `DataItem` interface to match the structure of the data returned by your API.

write the program for passing props from parent component to child component and child component to parent components using reactjs with typescript

Sure! In React with TypeScript, you can pass props from a parent component to a child component and from the child component back to the parent component using callback functions.

Here's an example of how you can achieve this:

Step 1: Create a new file `ParentComponent.tsx` for the parent component:

```
```.tsx
// ParentComponent.tsx
import React, { useState } from 'react';
import ChildComponent from './ChildComponent';

interface ParentProps {
 initialCount: number;
}

const ParentComponent: React.FC<ParentProps> = ({ initialCount }) => {
 const [count, setCount] = useState(initialCount);

 const handleChildClick = (increment: number) => {
 setCount(count + increment);
 };

 return (
 <div>
 <h1>Parent Component</h1>
 <p>Count: {count}</p>
 <ChildComponent count={count} onChildClick={handleChildClick} />
 </div>
);
};

export default ParentComponent;
```

...

Step 2: Create a new file `ChildComponent.tsx` for the child component:

```
```tsx
```

```
// ChildComponent.tsx
```

```
import React from 'react';
```

```
interface ChildProps {
```

```
  count: number;
```

```
  onChildClick: (increment: number) => void;
```

```
}
```

```
const ChildComponent: React.FC<ChildProps> = ({ count, onChildClick }) => {
```

```
  const handleIncrement = () => {
```

```
    onChildClick(1);
```

```
  };
```

```
  const handleDecrement = () => {
```

```
    onChildClick(-1);
```

```
  };
```

```
  return (
```

```
    <div>
```

```
      <h2>Child Component</h2>
```

```
      <p>Count from parent: {count}</p>
```

```
      <button onClick={handleIncrement}>Increment</button>
```

```
      <button onClick={handleDecrement}>Decrement</button>
```

```
    </div>
```

```
  );
```

```
};
```

```
export default ChildComponent;
```

```
...
```

Step 3: Use the `ParentComponent` in the `App.tsx` file:

```
```tsx
```

```
// App.tsx
```

```
import React from 'react';
```

```
import ParentComponent from './ParentComponent';
```

```
const App: React.FC = () => {
```

```
 return (
```

```
 <div>
```

```
 <h1>Passing Props Example</h1>
```

```
 <ParentComponent initialCount={0} />
```

```
 </div>
```

```
);
```

```
};
```

```
export default App;
```

```
...
```

Now, when you run the app using `npm start`, you'll see the "Parent Component" with a count value and two buttons: "Increment" and "Decrement". Clicking these buttons will update the count value, which is passed from the parent component to the child component through props. When the buttons in the child component are clicked, they call the callback function `onChildClick`, which updates the count value in the parent component.

This example demonstrates how to pass props from a parent component to a child component and vice versa using React with TypeScript.

## write the todo list crud operations application program with localstorage using reactjs with typescript

Sure! Here's an example of a simple Todo List application with CRUD (Create, Read, Update, Delete) operations using ReactJS with TypeScript and localStorage for data persistence.

Step 1: Set up a new React project with TypeScript template:

```
```bash
npx create-react-app todo-app --template typescript
cd todo-app
```
```

Step 2: Create a new file `src/ToDo.tsx` for the Todo component:

```
```tsx
// src/ToDo.tsx
import React, { useState, useEffect } from 'react';

interface Todo {
  id: number;
  text: string;
}

const Todo: React.FC = () => {
  const [todos, setTodos] = useState<Todo[]>([]);
  const [newTodoText, setNewTodoText] = useState('');

  useEffect(() => {
    const storedTodos = JSON.parse(localStorage.getItem('todos') || '[]');
    setTodos(storedTodos);
  }, []);
};
```

```
useEffect(() => {  
  localStorage.setItem('todos', JSON.stringify(todos));  
}, [todos]);
```

```
const handleAddTodo = () => {  
  if (newTodoText.trim() !== '') {  
    const newTodo: Todo = {  
      id: Date.now(),  
      text: newTodoText,  
    };  
    setTodos([...todos, newTodo]);  
    setNewTodoText('');  
  }  
};
```

```
const handleDeleteTodo = (id: number) => {  
  setTodos(todos.filter((todo) => todo.id !== id));  
};
```

```
return (  
  <div>  
    <h1>Todo List</h1>  
    <div>  
      <input  
        type="text"  
        value={newTodoText}  
        onChange={(e) => setNewTodoText(e.target.value)}  
        placeholder="Enter a new todo"  
      />  
      <button onClick={handleAddTodo}>Add Todo</button>
```



```

    </div>

    <ul>
      {todos.map((todo) => (
        <li key={todo.id}>
          <span>{todo.text}</span>
          <button onClick={() => handleDeleteTodo(todo.id)}>Delete</button>
        </li>
      ))}
    </ul>
  </div>

);
};

export default Todo;
...

```

Step 3: Update the `src/App.tsx` file to use the `Todo` component:

```

```tsx
// src/App.tsx
import React from 'react';
import Todo from './Todo';

const App: React.FC = () => {
 return (
 <div className="App">
 <Todo />
 </div>
);
};

```

```
export default App;
```

```
...
```

Now, when you run the app using `npm start`, you'll see a simple Todo List application. You can add new todos using the input field and the "Add Todo" button. The todos will be saved in `localStorage`, so they will persist even after you refresh the page or close the browser.

This example demonstrates a basic implementation of a Todo List application with CRUD operations using React with TypeScript and `localStorage` for data persistence. You can further enhance this application by adding more features like updating todos, marking todos as completed, etc.

### **write the program of protected routing using react js with typescript**

To implement protected routing in ReactJS with TypeScript, you can use a combination of React Router and custom authentication logic. This will ensure that certain routes are accessible only when the user is authenticated. We'll create a simple example where a user needs to log in to access a protected route.

Step 1: Set up a new React project with TypeScript template:

```
```bash
npx create-react-app protected-routing --template typescript
cd protected-routing
...

```

Step 2: Install the necessary dependencies:

```
```bash
npm install react-router-dom @types/react-router-dom
...

```

Step 3: Create a new file `src/components/Login.tsx` for the login component:

```
```tsx
```

```
// src/components/Login.tsx
```

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

```
import { useHistory } from 'react-router-dom';
```

```
const Login: React.FC = () => {
```

```
  const [username, setUsername] = useState("");
```

```
  const [password, setPassword] = useState("");
```

```
  const history = useHistory();
```

```
  const handleLogin = () => {
```

```
    // Perform authentication logic here, e.g., validate username and password
```

```
    // For simplicity, we will use a simple username/password check
```

```
    if (username === 'user' && password === 'password') {
```

```
      // Successful login, navigate to the protected route
```

```
      history.push('/dashboard');
```

```
    } else {
```

```
      alert('Invalid credentials. Please try again.');
```

```
    }
```

```
  };
```

```
  return (
```

```
    <div>
```

```
      <h2>Login</h2>
```

```
      <div>
```

```
        <label htmlFor="username">Username:</label>
```

```
        <input
```

```
          type="text"
```

```
          id="username"
```

```
          value={username}
```

```
          onChange={(e) => setUsername(e.target.value)}
```

```

    />
  </div>
  <div>
    <label htmlFor="password">Password:</label>
    <input
      type="password"
      id="password"
      value={password}
      onChange={(e) => setPassword(e.target.value)}
    />
  </div>
  <button onClick={handleLogin}>Login</button>
</div>
);
};

export default Login;
...

```

Step 4: Create a new file `src/components/Dashboard.tsx` for the protected dashboard component:

```

```tsx
// src/components/Dashboard.tsx
import React from 'react';

const Dashboard: React.FC = () => {
 return (
 <div>
 <h2>Dashboard</h2>
 <p>Welcome to the protected dashboard!</p>
 </div>
)
}

```

```
);
};

export default Dashboard;
...

```

Step 5: Update the `src/App.tsx` file to implement the protected routing:

```
```tsx  
// src/App.tsx  
  
import React from 'react';  
import { BrowserRouter as Router, Route, Switch, Redirect } from 'react-router-dom';  
import Login from './components/Login';  
import Dashboard from './components/Dashboard';  
  
const App: React.FC = () => {  
  // Simulate authentication state, replace this with your actual authentication logic  
  const isAuthenticated = true;  
  
  return (  
    <Router>  
      <div className="App">  
        <Switch>  
          <Route exact path="/">  
            {isAuthenticated ? <Redirect to="/dashboard" /> : <Login />}  
          </Route>  
          <Route path="/dashboard">  
            {isAuthenticated ? <Dashboard /> : <Redirect to="/" />}  
          </Route>  
        </Switch>  
      </div>  
    )  
  );  
};  
export default App;  
```
```

```
</Router>

);

};

export default App;
...

```

In this example, the `App` component uses React Router's `Switch` component to define two routes: the login route ("/") and the protected dashboard route ("/dashboard"). The `isAuthenticated` variable simulates the user's authentication state. When the user is authenticated, they can access the dashboard; otherwise, they are redirected to the login page. You should replace the `isAuthenticated` logic with your actual authentication logic in a real-world application.

Now, when you run the app using `npm start`, you'll see a simple protected routing example. If the `isAuthenticated` variable is set to true (simulating a logged-in user), the user will be redirected to the protected dashboard. If `isAuthenticated` is set to false (simulating a logged-out user), the user will see the login page.

This example demonstrates how to implement protected routing using React with TypeScript. Keep in mind that this is a basic example, and in a real-world application, you would have more complex authentication logic and proper security measures to protect your routes.

### **write the program for unit testing using enzyme in react js with typescript**

To perform unit testing using Enzyme in ReactJS with TypeScript, follow these steps:

Step 1: Install the necessary dependencies:

```
```bash
npm install enzyme enzyme-adapter-react-16 @types/enzyme @types/enzyme-adapter-react-16
enzyme-to-json enzyme-adapter-utils jest @types/jest --save-dev
...

```

Step 2: Configure Enzyme for your tests. Create a file `src/setupTests.ts`:

```
```tsx
// src/setupTests.ts

import { configure } from 'enzyme';
import Adapter from 'enzyme-adapter-react-16';

configure({ adapter: new Adapter() });
```
```

Step 3: Write a sample component that you want to test. For this example, we'll create a simple `Counter` component:

```
```tsx
// src/components/Counter.tsx

import React, { useState } from 'react';

const Counter: React.FC = () => {
 const [count, setCount] = useState(0);

 const handleIncrement = () => {
 setCount(count + 1);
 };

 const handleDecrement = () => {
 setCount(count - 1);
 };

 return (
 <div>
 <h2>Counter</h2>
 <p>Count: {count}</p>
 </div>
);
};
```
```

```

    <button onClick={handleIncrement}>Increment</button>

    <button onClick={handleDecrement}>Decrement</button>

  </div>

);

};

export default Counter;
...

```

Step 4: Write a test for the `Counter` component. Create a file
`src/components/__tests__/Counter.test.tsx`:

```

```tsx
// src/components/__tests__/Counter.test.tsx

import React from 'react';
import { shallow } from 'enzyme';
import Counter from '../Counter';

describe('Counter Component', () => {
 it('renders without crashing', () => {
 const wrapper = shallow(<Counter />);
 expect(wrapper.exists()).toBe(true);
 });

 it('renders the initial count', () => {
 const wrapper = shallow(<Counter />);
 expect(wrapper.find('p').text()).toContain('Count: 0');
 });

 it('increments the count on clicking increment button', () => {
 const wrapper = shallow(<Counter />);

```



```

 wrapper.find('button').at(0).simulate('click');

 expect(wrapper.find('p').text()).toContain('Count: 1');

 });

 it('decrements the count on clicking decrement button', () => {
 const wrapper = shallow(<Counter />);
 wrapper.find('button').at(1).simulate('click');
 expect(wrapper.find('p').text()).toContain('Count: -1');
 });
});
...

```

Step 5: Update `src/App.tsx` to render the `Counter` component:

```

```tsx
// src/App.tsx

import React from 'react';
import Counter from './components/Counter';

const App: React.FC = () => {
  return (
    <div className="App">
      <Counter />
    </div>
  );
};

export default App;
...

```

Step 6: Update `src/index.tsx` to include the `setupTests.ts` file:

```
```tsx
// src/index.tsx

import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

import './setupTests'; // Include the setupTests.ts file

ReactDOM.render(
 <React.StrictMode>
 <App />
 </React.StrictMode>,
 document.getElementById('root')
);

reportWebVitals();
```
```

Step 7: Run the tests:

```
```bash
npm test
```
```

You should see the test results in the terminal, indicating that all tests passed. The output will look something like this:

```
```
```

PASS src/components/\_\_tests\_\_/Counter.test.tsx

### Counter Component

- ✓ renders without crashing (11 ms)
- ✓ renders the initial count (3 ms)
- ✓ increments the count on clicking increment button (3 ms)
- ✓ decrements the count on clicking decrement button (3 ms)

...

Congratulations! You have successfully written unit tests using Enzyme in a ReactJS application with TypeScript. Enzyme makes it easy to test React components by providing a set of utilities to traverse, manipulate, and interact with the rendered output.