Redux Toolkit for React

Step-by-Step Guide

1. Install Redux Toolkit

To get started, install the necessary packages for Redux Toolkit and React-Redux:

terminal

```
npm install @reduxjs/toolkit react-redux
```

- **@reduxjs/toolkit**: Simplifies Redux setup by providing pre-configured tools for common tasks like state management and middleware.
- **react-redux**: Provides React bindings for Redux, enabling components to connect to the store and interact with state easily.

This step ensures that your project has all the required dependencies to start using Redux efficiently.

2. Create a Redux Store

The Redux store is the central place where your application state resides. Use **configureStore** to set it up:

store.js

```
// src/app/store.js
import { configureStore } from '@reduxjs/toolkit';

// Create an empty store (reducers will be added later)
export const store = configureStore({
  reducer: {}, // No reducers yet
});
```

- **configureStore**: Automatically sets up the Redux store with good defaults, such as middleware and development tools.

- **reducer**: An object that maps slice names (like "counter") to their corresponding reducers. This structure helps organize state and logic for different features. to connect to the store and interact with state easily.

By organizing reducers into slices, you maintain a modular and scalable structure.

3. Create a Slice

A slice combines state, reducers, and actions for a specific feature. Here's how to create one:

counterSlice.js

```
// src/features/counter/counterSlice.js
import { createSlice } from '@reduxjs/toolkit';

const counterSlice = createSlice({
  name: 'counter', // The unique name of this slice
  initialState: { value: 0 }, // The starting state for the slice
  reducers: {
    increment: (state) => {
      state.value += 1; // Increment the counter value by 1
    },
    incrementByAmount: (state, action) => {
      state.value += action.payload;
      // Increment by a specific number passed in the action
    },
  },
});

export const { increment, incrementByAmount } = counterSlice.actions;
export default counterSlice.reducer;
```

- **createSlice:** Automatically generates action creators and reducers, making state management easier and reducing boilerplate code.

- **initialState:** Defines the starting value of the counter. In this case, it starts at 0.
- **increment**: Adds 1 to the current value when this action is dispatched.
- **incrementByAmount**: Adds a specific number (provided as action.payload) to the current value.

For example:

- If the state starts at **0** and **increment** is called, the state becomes **1**.
- If **incrementByAmount(5)** is called, the state increases by **5**.

This setup is simple and effective for managing a counter.

4. Import the Slice into the Store

Now, update the store to include the counter slice reducer.

store.js

```
// src/app/store.js
import { configureStore } from '@reduxjs/toolkit';

import counterReducer from '../features/counter/counterSlice';

// Add the counter reducer to the store
export const store = configureStore({
  reducer: {
    counter: counterReducer, // Register the counter slice reducer
  },
});
```

Now the store is configured with the counter slice.

5. Provide the Store to React

Connect your Redux store to your React app by wrapping the root component with the Provider component:

index.js

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

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

Provider: A higher-order component that makes the Redux store available to all components in the app.

This step ensures that every component in your app can access the Redux store.

6. Use Redux State in Components

Access State with useSelector

To retrieve data from the Redux store, use the **useSelector** hook:

CounterValue.jsx

```jsx
import React from 'react';
import { useSelector } from 'react-redux';

const CounterValue = () => {
// Access counter state
  const count = useSelector((state) => state.counter.value);
  return <div>Count: {count}</div>;
};
```

useSelector: A hook that allows you to extract specific pieces of state from the Redux store. It helps your component stay updated when the state changes.

6. Use Redux Actions in Components

Dispatch Actions with **useDispatch**

To update the Redux store, use the **useDispatch** hook to dispatch actions:

CounterControls.jsx

```jsx
import React from 'react';
import { useDispatch } from 'react-redux';
import { increment, incrementByAmount } from './features/counter/counterSlice';

const CounterControls = () => {
  const dispatch = useDispatch(); // Access the dispatch function

  return (
    <div>
      <button onClick={() => dispatch(increment())}>Increment</button>
      <button onClick={() => dispatch(incrementByAmount(5))}>Increment by 5</button>
    </div>
  );
};
```

useDispatch: A hook to send actions to the Redux store, triggering state updates.

7. Folder Structure Example

Organizing your files properly is crucial for scalability and maintainability. Here's an example:

```
src/
├── app/
│   └── store.js
├── features/
│   └── counter/
│       ├── counterSlice.js
├── components/
│   ├── CounterControls.js
│   └── CounterValue.js
├── App.js
├── index.js
```

app/: Contains the Redux store configuration.
features/: Groups slices by feature, keeping your project modular and organized.

This structure helps maintain clarity, even as your application grows.

Hopefully You Found It Usefull!

“Be sure to save this post so you can come back to it later”

like Comment Share