8

State Management

In this chapter, we'll learn about **shared state**, which is state that is used by several different components. We will explore three approaches to managing shared state, discussing the pros and cons of each approach.

To do this, we will build a simple app containing a header that displays the user's name, with the main content also referencing the user's name. The user's name will be stored in state that needs to be accessed by several components.

We will start with the simplest state solution. This is to use one of React's state hooks to store the state and pass it to other components using props. This approach is often referred to as **prop drilling**.

The second approach we will learn about is a feature in React called **context**. We will learn how to create a context containing a state and let other components access it.

The last approach we will cover is a popular library called **Redux**. We will take the time to understand what Redux is and its concepts before refactoring the app to use it.

So, we'll cover the following topics:

- Creating the project
- Using prop drilling
- Using React context
- Using Redux

Technical requirements

We will use the following technologies in this chapter:

- **Node.js** and **npm**: You can install them from `https://nodejs.org/en/download/`
- **Visual Studio Code**: You can install it from `https://code.visualstudio.com/`

All the code snippets in this chapter can be found online at `https://github.com/PacktPublishing/Learn-React-with-TypeScript-2nd-Edition/tree/main/Chapter8`.

Creating the project

We will develop our form using Visual Studio Code and a new Create React App-based project setup. We've previously covered this several times, so we will not cover the steps in this chapter – instead, see *Chapter 3*, *Setting Up React and TypeScript*.

We will style the form with Tailwind CSS. We also previously covered how to install and configure Tailwind in Create React App in *Chapter 5*, *Approaches to Styling Frontends*. So, after you have created the React and TypeScript project, install and configure Tailwind.

We will also use the `@tailwindcss/forms` plugin to style the form. So, install this plugin as well – see *Chapter 7*, *Working with Forms*, for information on how to do this.

The app we will build will contain a header and some content beneath it. Here is the component structure we will create:

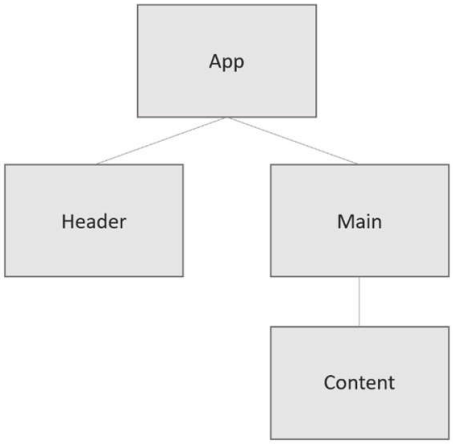

Figure 8.1 – App component structure

The header will have a **Sign in** button to authenticate and authorize a user to get their name and permissions. Once authenticated, the user's name will be displayed in the app header, and the user will be welcomed in the content. If the user has admin permissions, important content will be shown.

So, carry out the following steps to create the initial versions of the files that we need in the app without any statement management (some of the code snippets are lengthy – don't forget you can copy them from `https://github.com/PacktPublishing/Learn-React-with-TypeScript-2nd-Edition/tree/main/Chapter8/prop-drilling`):

1. We will start by creating a file containing a function to authenticate a user. Create a folder called `api` in the `src` folder. Then, create a file called `authenticate.ts` in the `api` folder with the following content:

```
export type User = {
  id: string;
```

```
  name: string;
};
export function authenticate(): Promise<User | undefined>
{
  return new Promise((resolve) =>
    setTimeout(() => resolve({ id: "1", name: "Bob" }),
      1000)
  );
}
```

The function simulates successful authentication for a user called Bob.

2. Next, we will create a file containing a function to authorize a user. So, create a file called `authorize.ts` in the `api` folder with the following content:

```
export function authorize(id: string): Promise<string[]>
{
  return new Promise((resolve) =>
    setTimeout(() => resolve(["admin"]), 1000)
  );
}
```

The function simulates a user being authorized to have admin permissions.

3. Next, we will create a component for the app header. Create a file called `Header.tsx` in the `src` folder with the following content:

```
import { User } from './api/authenticate';

type Props = {
  user: undefined | User;
  onSignInClick: () => void;
  loading: boolean;
};
```

The component has a prop for the user, which will be `undefined` if the user isn't authenticated yet. The component also has a prop called `onSignInClick` for a **Sign in** button click handler. The last component prop is called `loading` and it determines whether the app is in a loading state when the user is authenticated or authorized.

4. Add the following component implementation into `Header.tsx`:

```
export function Header({
  user,
  onSignInClick,
  loading,
}: Props) {
  return (
    <header className="flex justify-between items-center
      border-b-2 border-gray-100 py-6">
      {user ? (
        <span className="ml-auto font-bold">
          {user.name} has signed in
        </span>
      ) : (
        <button
          onClick={onSignInClick}
          className="whitespace-nowrap inline-flex items-
            center justify-center ml-auto px-4 py-2 w-36
            border border-transparent rounded-md
            shadow-sm text-base font-medium text-white
            bg-indigo-600 hover:bg-indigo-700"
          disabled={loading}
        >
          {loading ? '...' : 'Sign in'}
        </button>
      )}
    </header>
  );
}
```

The component informs the user that they have signed in if they have been authenticated. If the user is unauthenticated, the component displays a **Sign in** button.

5. Next, we will implement a component for the main app content. Create a file called `Main.tsx` in the `src` folder with the following content:

```
import { User } from './api/authenticate';
import { Content } from './Content';

type Props = {
```

```
  user: undefined | User;
  permissions: undefined | string[];
};
```

The component has a prop for the user and their permissions. We have imported a component called `Content`, which we will create in *step 7*.

6. Now, add the following component implementation in `Main.tsx`:

```
export function Main({ user, permissions }: Props) {
  return (
    <main className="py-8">
      <h1 className="text-3xl text-center font-bold
        underline">Welcome</h1>
      <p className="mt-8 text-xl text-center">
        {user ? `Hello ${user.name}!` : "Please sign in"}
      </p>
      <Content permissions={permissions} />
    </main>
  );
}
```

The component instructs the user to sign in if they are unauthenticated or shows a **Hello** message if they are authenticated. The content also references a `Content` component passing it the user's permissions.

7. The last file to create in the `src` folder is called `Content.tsx`. Add the following content to the file:

```
type Props = {
  permissions: undefined | string[];
};
export function Content({ permissions }: Props) {
  if (permissions === undefined) {
    return null;
  }

  return permissions.includes('admin') ? (
    <p className="mt-4 text-l text-center">
      Some important stuff that only an admin can do
    </p>
```

```
    ) : (
      <p className="mt-4 text-l text-center">
        Insufficient permissions
      </p>
    );
  }
```

If the user is unauthorized, the component displays nothing. If the user has admin permissions, it displays some important stuff. Otherwise, it informs the user that they lack permissions.

That completes the project setup. The app will compile and run but won't show any of the components we created yet because we haven't referenced them in the `App` component. We will do this next when we share the user and permission information across several components.

Using prop drilling

In this first state management approach, we will store the `user`, `permissions`, and `loading` state in the `App` component. The `App` component will then pass this state to the `Header` and `Main` components using props.

So, this approach uses React features that we are already aware of. The approach is referred to as **prop drilling** because the state is passed down the component tree using props.

Carry out the following steps to rework the `App` component to store the `user`, `permissions`, and `loading` state, and pass this state down to the `Header` and `Main` components:

1. Open `App.tsx` and start by removing all the existing code and adding the following import statements:

```
import { useReducer } from 'react';
import { Header } from './Header';
import { Main } from './Main';
import { authenticate, User } from './api/authenticate';
import { authorize } from './api/authorize';
```

We have imported `useReducer` from React to store the state. We have also imported the `Header` and `Main` components so that we can render them with the state values. Lastly, we've imported the `authenticate` and `authorize` functions because we will create the **Sign in** handler in this component.

2. After the import statements, add a type for the state and create a variable for the initial state values:

```
type State = {
  user: undefined | User,
```

```
  permissions: undefined | string[],
  loading: boolean,
};
const initialState: State = {
  user: undefined,
  permissions: undefined,
  loading: false,
};
```

3. Next, create a type for the different actions that can update the state:

```
type Action =
  | {
      type: "authenticate",
    }
  | {
      type: "authenticated",
      user: User | undefined,
    }
  | {
      type: "authorize",
    }
  | {
      type: "authorized",
      permissions: string[],
    };
```

The `"authenticate"` action will start the authentication process, and `"authenticated"` happens when it has been completed. Likewise, the `"authorize"` action will start the authorization process, and `"authorized"` happens when it has been completed.

4. Next, add a `reducer` function that updates the state:

```
function reducer(state: State, action: Action): State {
  switch (action.type) {
    case "authenticate":
      return { ...state, loading: true };
    case "authenticated":
      return { ...state, loading: false, user: action.
        user };
```

```
    case "authorize":
      return { ...state, loading: true };
    case "authorized":
      return {
        ...state,
        loading: false,
        permissions: action.permissions,
      };
    default:
      return state;
  }
}
```

The function takes in the existing state and the action as parameters. The function uses a switch statement on the action type to create a new version of the state in each branch.

5. Now, let's define the `App` component as follows:

```
function App() {
  const [{ user, permissions, loading }, dispatch] =
    useReducer(reducer, initialState);

  return (
    <div className="max-w-7xl mx-auto px-4">
      <Header
        user={user}
        onSignInClick={handleSignInClick}
        loading={loading}
      />
      <Main user={user} permissions={permissions} />
    </div>
  );
}

export default App;
```

The component uses `useReducer` with the `reducer` function and the `initialState` variable we defined earlier. We have destructured the `user`, `permissions`, and `loading` state values from `useReducer`. In the JSX, we have rendered both the `Header` and `Main` components passing the appropriate state values as props.

6. The `Header` element in the JSX references a handler called `handleSignInClick`, which needs implementation. Create this above the return statement as follows:

```
async function handleSignInClick() {
  dispatch({ type: "authenticate" });
  const authenticatedUser = await authenticate();
  dispatch({
    type: "authenticated",
    user: authenticatedUser,
  });
  if (authenticatedUser !== undefined) {
    dispatch({ type: "authorize" });
    const authorizedPermissions = await authorize(
      authenticatedUser.id
    );
    dispatch({
      type: "authorized",
      permissions: authorizedPermissions,
    });
  }
}
```

The sign-in handler authenticates and authorizes the user and dispatches the necessary actions along the way.

7. Run the app in development mode by running `npm start` in the terminal. The app appears as shown in the screenshot:

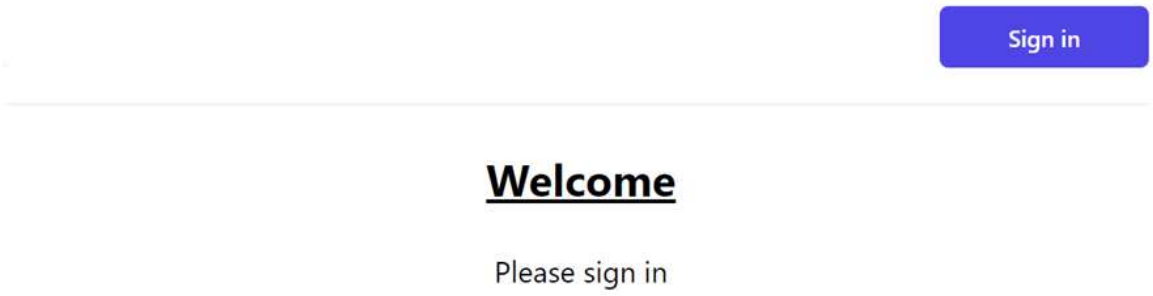

Figure 8.2 – App before signing in

8. Click the **Sign in** button. The authentication and authorization processes then happen, and after a couple of seconds, the following screen appears:

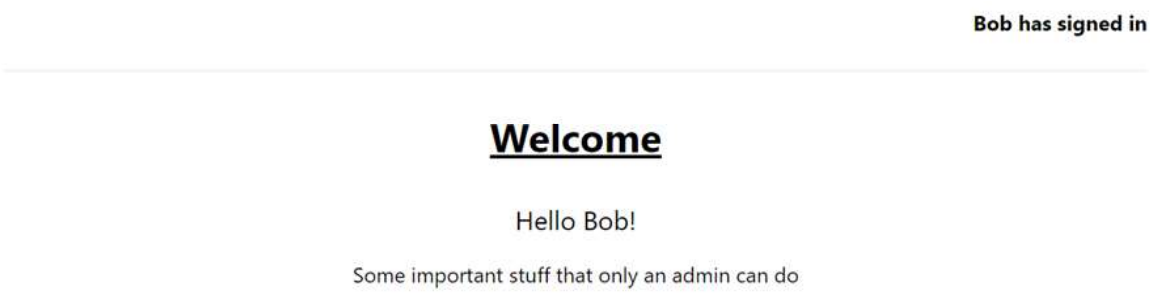

Figure 8.3 – App after signing in

That completes the prop drilling approach.

A nice thing about this approach is that it is simple and uses React features we are already familiar with. A downside of this approach is that it forces all components between the component providing state and components accessing the state to have a prop for that state. So, some components that do not need access to the state are forced to access it. An example is the `Main` component – the `permissions` state is forced to pass through it to the `Content` component.

The key point in this section is that it is fine to share state across a few adjacent components using props but isn't ideal for sharing across lots of components far apart in the component tree.

Next, keep the app running, and we will look at a more appropriate solution for sharing state across many components.

Using React context

In this section, we will learn a feature in React called **context**. We will then refactor the app from the last section to use React context.

Understanding React context

React context is an object that can be accessed by components. This object can contain state values, so it provides a mechanism for sharing state across components.

A context is created using a `createContext` function as follows:

```
const SomeContext = createContext<ContextType>(defaultValue);
```

A default value for the context must be passed into `createContext`. It also has a generic type parameter for the type that represents the object created by `createContext`.

The context also contains a `Provider` component that needs to be placed above components requiring access to the context object in the component tree. A provider wrapper component can be created that stores the shared state and passes it to the context `Provider` component as follows:

```
export function SomeProvider({ children }: Props) {
  const [someState, setSomeState] = useState(initialState);
  return (
    <SomeContext.Provider value={{ someState }}>
      {children}
    </SomeContext.Provider>
  );
}
```

`useState` has been used for the state in the preceding example, but `useReducer` could also be used.

The provider wrapper component can then be placed appropriately in the component tree, above components requiring the shared state:

```
function App() {
  return (
    <SomeProvider>
      <Header />
      <Main />
    </SomeProvider>
  );
}
```

React also contains a `useContext` hook that can be used so that the context values can be consumed as a hook, as follows:

```
const { someState } = useContext(SomeContext);
```

The context must be passed into `useContext` and properties from the context object can be destructured from its result.

So, components that want access to the shared state can access it using `useContext` as follows:

```
export function SomeComponent() {
  const { someState } = useContext(SomeContext);
  return <div>I have access to {someState}</div>;
}
```

For more information on React context, see the following link: `https://reactjs.org/docs/context.html`.

Now that we understand React context, we will use it in the app we created in the previous section.

Using React context

We will refactor the app from the last section to use React context. We will start by creating a file containing the context and the provider wrapper. Then, we will use `useReducer` in the provider wrapper to store the state. We will also create a wrapper for `useContext` to make consuming it easy.

So, to do this, carry out the following steps:

1. Start by creating a file called `AppContext.tsx` in the `src` folder. This will contain the context, the provider wrapper, and the `useContext` wrapper.
2. Add the following import statements to `AppContext.tsx`:

```
import {
  createContext,
  useContext,
  useReducer,
  ReactNode,
} from 'react';
import { User } from './api/authenticate';
```

We have imported all the functions we need from React along with the `ReactNode` type that we will need for the provider wrapper `children` prop. We have also imported the `User` type, which we will need for the user state type.

3. We need to add a type for the state and a variable for the initial state values. We already have these in `App.tsx`, so the following lines can be moved from `App.tsx` to `AppContext.tsx`:

```
type State = {
  user: undefined | User,
  permissions: undefined | string[],
  loading: boolean,
};
const initialState = {
  user: undefined,
  permissions: undefined,
  loading: false,
};
```

4. Similarly, the `Action` type and the `reducer` function can be moved from `App.tsx` to `AppContext.tsx`. Here are the lines to move:

```
type Action =
  | {
      type: "authenticate",
    }
  | {
      type: "authenticated",
      user: User | undefined,
    }
  | {
      type: "authorize",
    }
  | {
      type: "authorized",
      permissions: string[],
    };

function reducer(state: State, action: Action): State {
  switch (action.type) {
    case "authenticate":
      return { ...state, loading: true };
    case "authenticated":
      return { ...state, loading: false, user: action.
        user };
    case "authorize":
      return { ...state, loading: true };
    case "authorized":
      return { ...state, loading: false, permissions:
        action.permissions };
    default:
      return state;
  }
}
```

Note that the `App.tsx` file will raise a compile error after moving this function. We will resolve this in the next set of instructions.

5. Next, we will create a type for the context in `AppContext.tsx`:

```
type AppContextType = State & {
  dispatch: React.Dispatch<Action>,
};
```

The context will consist of the state values and a `dispatch` function to dispatch actions.

6. Now we can create the context as follows:

```
const AppContext = createContext<AppContextType>({
  ...initialState,
  dispatch: () => {},
});
```

We have called the context `AppContext`. We use the `initialState` variable and a dummy `dispatch` function as the default context value.

7. Next, we can implement the provider wrapper as follows:

```
type Props = {
  children: ReactNode;
};
export function AppProvider({ children }: Props) {
  const [{ user, permissions, loading }, dispatch] =
    useReducer(reducer, initialState);
  return (
    <AppContext.Provider
      value={{
        user,
        permissions,
        loading,
        dispatch,
      }}
    >
      {children}
    </AppContext.Provider>
  );
}
```

We have called the component `AppProvider`, and it returns the context's `Provider` component with the state values and the `dispatch` function from `useReducer`.

8. The last thing to do in `AppContext.tsx` is to create a wrapper for `useContext` as follows:

```
export const useAppContext = () =>
useContext(AppContext);
```

That completes the work we need to do in `AppContext.tsx`.

So, `AppContext.tsx` exports an `AppProvider` component that can be placed above `Header` and `Main` in the component tree so that they can access the user and permissions information. `AppContext.tsx` also exports `useAppContext` so that the `Header`, `Main`, and `Content` components can use it to get access to the user and permissions information.

Now, carry out the following steps to make the necessary changes to the `App`, `Header`, `Main`, and `Content` components to access the user and permissions information from `AppContext`:

1. We will start with `Header.tsx`. Begin by importing the `authenticate`, `authorize`, and `useAppContext` functions. Also, remove the `User` type and the props for the `Header` component:

```
import { authenticate } from './api/authenticate';
import { authorize } from './api/authorize';
import { useAppContext } from './AppContext';

export function Header() {
  return ...
}
```

2. `Header` will now handle the sign-in process instead of `App`. So, move the `handleSignInClick` handler from `App` in `App.tsx` to `Header.tsx` and place it above the return statement as follows:

```
export function Header() {
  async function handleSignInClick() {
    dispatch({ type: 'authenticate' });
    const authenticatedUser = await authenticate();
    dispatch({
      type: 'authenticated',
      user: authenticatedUser,
    });
    if (authenticatedUser !== undefined) {
      dispatch({ type: 'authorize' });
      const authorizedPermissions = await authorize(
```

```
          authenticatedUser.id
        );
        dispatch({
          type: 'authorized',
          permissions: authorizedPermissions,
        });
      }
    }
    return ...
}
```

3. Update the sign-in click handler to reference the function we just added:

```
<button
  onClick={handleSignInClick}
  className=...
  disabled={loading}
>
  {loading ? '...' : 'Sign in'}
</button>
```

4. The last thing to do in `Header.tsx` is to get `user`, `loading`, and `dispatch` from the context. Add the following call to `useAppContext` at the top of the component:

```
export function Header() {
  const { user, loading, dispatch } = useAppContext();
  ...
}
```

5. Let's move on to `Main.tsx`. Remove the import statement for the `User` type and add an import statement for `useAppContext`:

```
import { Content } from './Content';
import { useAppContext } from './AppContext';
```

6. Remove the props for the `Main` component and get `user` from `useAppContext`:

```
export function Main() {
  const { user } = useAppContext();
  return ...
}
```

7. In the JSX in `Main`, remove the `permissions` attribute on the `Content` element:

```
<Content />
```

8. Now, open `Content.tsx` and add an import statement for `useAppContext`:

```
import { useAppContext } from './AppContext';
```

9. Remove the props for the `Content` component and get `permissions` from `useAppContext`:

```
export function Content() {
  const { permissions } = useAppContext();
  if (permissions === undefined) {
    return null;
  }
  return ...
}
```

10. Lastly, we will modify `App.tsx`. Remove the import statements except for `Header` and `Main`, and add an import statement for `AppProvider`:

```
import { Header } from './Header';
import { Main } from './Main';
import { AppProvider } from './AppContext';
```

11. Still in `App.tsx`, remove the call to `useReducer` and remove all the attributes passed to `Header` and `Main`:

```
function App() {
  return (
    <div className="max-w-7xl mx-auto px-4">
      <Header />
      <Main />
    </div>
  );
}
```

12. Wrap `AppProvider` around `Header` and `Main` so that they can access the context:

```
function App() {
  return (
    <div className="max-w-7xl mx-auto px-4">
```

```
        <AppProvider>
          <Header />
          <Main />
        </AppProvider>
      </div>
    );
  }
```

The compile errors will now be resolved and the running app will look and behave like before.

13. Stop the app running by pressing *Ctrl* + *C*.

That completes the refactoring of the app to use React context instead of prop drilling.

In comparison to prop drilling, React context requires more code to be written. However, it allows components to access shared state using a hook rather than passing it through components using props. It's an elegant, shared-state solution, particularly when many components share state.

Next, we will learn about a popular third-party library that can be used to share state.

Using Redux

In this section, we will learn about Redux before using it to refactor the app we have been working on to use it.

Understanding Redux

Redux is a mature state management library that was first released in 2015. It was released before React context and became a popular approach for shared state management.

Creating a store

In Redux, the state lives in a centralized immutable object referred to as a **store**. There is only a single store for the whole app. Like `useReducer`, the state in a store is updated by dispatching an **action**, which is an object containing the type of change and any data required to make the change. An action is handled by a `reducer` function, which creates a new version of the state.

In the past, a lot of code was needed to set up a Redux store and consume it in a React component. Today, a companion library called Redux Toolkit reduces the code required to use Redux. A Redux store can be created using the Redux Toolkit's `configureStore` function as follows:

```
export const store = configureStore({
  reducer: {
    someFeature: someFeatureReducer,
```

```
    anotherFeature: anotherFeatureReducer
  },
});
```

The `configureStore` function takes in the store's reducers. Each feature in the app can have its own area of state and reducer to change the state. The different areas of state are often referred to as **slices**. In the preceding example, there are two slices called `someFeature` and `anotherFeature`.

The Redux Toolkit has a function to create slices, called `createSlice`:

```
export const someSlice = createSlice({
  name: "someFeature",
  initialState,
  reducers: {
    someAction: (state) => {
      state.someValue = "something";
    },
    anotherAction: (state) => {
      state.someOtherValue = "something else";
    },
  },
});
```

The `createSlice` function takes in an object parameter containing the slice name, the initial state, and functions to handle the different actions and update the state.

The slice created from `createSlice` contains a `reducer` function that wraps the action handlers. This `reducer` function can be referenced in the `reducer` property of `configureStore` when the store is created:

```
export const store = configureStore({
  reducer: {
    someFeature: someSlice.reducer,
    ...
  },
});
```

In the preceding code snippet, the reducer from `someSlice` has been added to the store.

Providing the store to React components

The Redux store is defined in the component tree using its `Provider` component. The value of the Redux store (from `configureStore`) needs to be specified on the `Provider` component. The `Provider` component must be placed above the components requiring access to the store:

```
<Provider store={store}>
  <SomeComponent />
  <AnotherComponent />
</Provider>
```

In the preceding example, `SomeComponent` and `AnotherComponent` have access to the store.

Accessing the store from a component

Components can access state from the Redux store using a `useSelector` hook from React Redux. A function that selects the relevant state in the store is passed into `useSelector`:

```
const someValue = useSelector(
  (state: RootState) => state.someFeature.someValue
);
```

In the preceding example, `someValue` is selected from the `someFeature` slice in the store.

Dispatching actions to the store from a component

React Redux also has a `useDispatch` hook that returns a `dispatch` function that can be used to dispatch actions. The action is a function from the slice created using `createSlice`:

```
const dispatch = useDispatch();
return (
  <button onClick={() => dispatch(someSlice.actions.
someAction())}>
    Some button
  </button>
);
```

In the preceding example, `someAction` in `someSlice` is dispatched when the button is clicked.

For more information on Redux, see the following link: `https://redux.js.org/`. And for more information on the Redux Toolkit, see the following link: `https://redux-toolkit.js.org/`.

Now that we understand Redux, we will use it in the app we created in the previous section.

Installing Redux

First, we must install Redux and the Redux Toolkit into our project. Run the following command in the terminal:

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

This will install all the Redux bits we need, including its TypeScript types.

Using Redux

Now, we can refactor the app to use Redux instead of React context. First, we will create a Redux slice for the user information before creating a Redux store with this slice. We will then move on to add the store to the React component tree and consume it in the `Header`, `Main`, and `Content` components.

Creating a Redux Slice

We will start by creating a Redux slice for the state for a user. Carry out the following steps:

1. Create a folder called `store` in the `src` folder and then a file called `userSlice.ts` within it.
2. Add the following import statements to `userSlice.ts`:

   ```
   import { createSlice } from '@reduxjs/toolkit';
   import type { PayloadAction } from '@reduxjs/toolkit';
   import { User } from '../api/authenticate';
   ```

   We will eventually use `createSlice` to create the Redux slice. `PayloadAction` is a type that we can use for action objects. We will need the `User` type when defining the type for the state.

3. Copy the following `State` type and initial state value from `AppContext.tsx` into `userSlice.ts`:

   ```
   type State = {
     user: undefined | User;
     permissions: undefined | string[];
     loading: boolean;
   };
   const initialState: State = {
     user: undefined,
     permissions: undefined,
     loading: false,
   };
   ```

4. Next, start to create the slice in `userSlice.ts`, as follows:

```
export const userSlice = createSlice({
  name: 'user',
  initialState,
  reducers: {

  }
});
```

We have named the slice `user` and passed in the initial state value. We export the slice so that we can use it later to create the Redux store.

5. Now, define the following action handlers inside the `reducers` object:

```
reducers: {
  authenticateAction: (state) => {
    state.loading = true;
  },
  authenticatedAction: (
    state,
    action: PayloadAction<User | undefined>
  ) => {
    state.user = action.payload;
    state.loading = false;
  },
  authorizeAction: (state) => {
    state.loading = true;
  },
  authorizedAction: (
    state,
    action: PayloadAction<string[]>
  ) => {
    state.permissions = action.payload;
    state.loading = false;
  }
}
```

Each action handler updates the required state. `PayloadAction` is used for the type of the action parameters. `PayloadAction` is a generic type with a parameter for the type of the action payload.

6. Lastly, export the action handlers and the `reducer` function from the slice:

```
export const {
  authenticateAction,
  authenticatedAction,
  authorizeAction,
  authorizedAction,
} = userSlice.actions;

export default userSlice.reducer;
```

A default export has been used for the `reducer` function so the consumer can name it as required.

That completes the implementation of the Redux slice.

Creating the Redux store

Next, let's create the Redux store. Carry out the following steps:

1. Create a file called `store.ts` in the `store` folder containing the following import statements:

```
import { configureStore } from '@reduxjs/toolkit';
import userReducer from './userSlice';
```

2. Next, use the `configureStore` function to create the store referencing the reducer from the slice we created earlier:

```
export const store = configureStore({
  reducer: { user: userReducer }
});
```

We export the `store` variable so that we can later use it on React Redux's `Provider` component.

3. The last thing to do in `store.ts` is export the type for Redux's full state object, which we will eventually require in the `useSelector` hook in components consuming the Redux store:

```
export type RootState = ReturnType<typeof store.
getState>;
```

`ReturnType` is a standard TypeScript utility type that returns the return type of the function type passed into it. The `getState` function in the Redux store returns the full state object. So, we use `ReturnType` to infer the type of the full state object rather than explicitly defining it.

That completes the implementation of the Redux store.

Adding the Redux store to the component tree

Next, we will add the store at an appropriate place in the component tree using the `Provider` component from React Redux. Follow these steps:

1. Open `App.tsx` and remove the `AppContext` import statement. Remove the `AppContext.tsx` file as well because this is no longer required.
2. Add an import statement for the `Provider` component from React Redux and the Redux store we created:

```
import { Provider } from 'react-redux';
import { store } from './store/store';
```

3. Replace `AppProvider` with `Provider` in the JSX, as follows:

```
<div className="max-w-7xl mx-auto px-4">
  <Provider store={store}>
    <Header />
    <Main />
  </Provider>
</div>
```

   We pass the imported Redux store into `Provider`.

The Redux store is now accessible to the `Header`, `Main`, and `Content` components.

Consuming the Redux store in the components

We will now integrate the Redux store into the `Header`, `Main`, and `Content` components. This will replace the previous React context consumption code. Follow these steps:

1. Start by opening `Header.tsx` and remove the `AppContext` import statement.
2. Add the following import statements to `Header.tsx`:

```
import { useSelector, useDispatch } from 'react-redux';
import type { RootState } from './store/store';
import {
  authenticateAction,
```

```
  authenticatedAction,
  authorizeAction,
  authorizedAction,
} from './store/userSlice';
```

We will be referencing state from Redux as well as dispatching actions, so we have imported both `useSelector` and `useDispatch`. The `RootState` type is required in the function we will eventually pass to `useSelector`. We have also imported all the actions from the slice we created because we will need them in the revised sign-in handler.

3. Inside the `Header` component, replace the `useAppContext` call with `useSelector` calls to get the required state:

```
export function Header() {
  const user = useSelector(
    (state: RootState) => state.user.user
  );
  const loading = useSelector(
    (state: RootState) => state.user.loading
  );
  async function handleSignInClick() {
    ...
  }
  return ...
}
```

4. Also, call `useDispatch` to get a `dispatch` function:

```
export function Header() {
  const user = useSelector(
    (state: RootState) => state.user.user
  );
  const loading = useSelector(
    (state: RootState) => state.user.loading
  );
  const dispatch = useDispatch();
  async function handleSignInClick() {
    ...
  }
  return ...
}
```

5. The last thing to do in `Header.tsx` is to modify `handleSignInClick` to reference the action functions from the Redux slice:

```
async function handleSignInClick() {
  dispatch(authenticateAction());
  const authenticatedUser = await authenticate();
  dispatch(authenticatedAction(authenticatedUser));
  if (authenticatedUser !== undefined) {
    dispatch(authorizeAction());
    const authorizedPermissions = await authorize(
      authenticatedUser.id
    );
    dispatch(authorizedAction(authorizedPermissions));
  }
}
```

6. Now, open `Main.tsx` and replace the `AppContext` import statement with import statements for `useSelector` and the `RootState` type:

```
import { useSelector } from 'react-redux';
import { RootState } from './store/store';
```

7. Replace the call to `useAppContext` with a call to `useSelector` to get the `user` state value:

```
export function Main() {
  const user = useSelector(
    (state: RootState) => state.user.user
  );
  return ...
}
```

8. Next, open `Content.tsx` and replace the `AppContext` import statement with import statements for `useSelector` and the `RootState` type:

```
import { useSelector } from 'react-redux';
import { RootState } from './store/store';
```

9. Replace the call to `useAppContext` with a call to `useSelector` to get the `permissions` state value:

```
export function Content() {
  const permissions = useSelector(
    (state: RootState) => state.user.permissions
  );
  if (permissions === undefined) {
    return null;
  }
  return ...
}
```

10. Run the app by running `npm start` in the terminal. The app will look and behave just as it did before.

That completes the refactoring of the app to use Redux rather than React context.

Here's a recap of the key points for using Redux:

- State is stored in a central store
- State is updated by dispatching actions that are handled by reducers
- A `Provider` component needs to be placed appropriately in the component tree to give components access to the Redux store
- Components can select state using a `useSelector` hook and dispatch actions using a `useDispatch` hook

As you have experienced, even using the Redux Toolkit requires many steps when using Redux to manage state. It is overkill for simple state management requirements but shines when there is a lot of shared application-level state.

Summary

In this chapter, we built a small one-page app that contained components that needed to share state. We started by using our existing knowledge and used props to pass the state between the components. We learned that a problem with this approach was that components not needing access to the state are forced to access it if its child components do need access to it.

We moved on to learn about React context and refactored the app to use it. We learned that React context can store state using `useState` or `useReducer`. The state can then be provided to components in the tree using the context's `Provider` component. Components then access the context state via the

`useContext` hook. We found that this was a much nicer solution than passing the state via props, particularly when many components need access to the state.

Next, we learned about Redux, which is similar to React context. A difference is that there can only be a single Redux store containing the state, but there can be many React contexts. We learned that a `Provider` component needs to be added to the component tree to give components access to the Redux store. Components select state using the `useSelector` hook and dispatch actions using the `useDispatch` hook. Reducers handle actions and then update the state accordingly.

In the next chapter, we will learn how to work with REST APIs in React.

Questions

Answer the following questions to check what you have learned in this chapter:

1. We have a context defined as follows to hold the theme state for an app:

```
type Theme = {
  name: string;
  color: 'dark' | 'light';
};
type ThemeContextType = Theme & {
  changeTheme: (
    name: string,
    color: 'dark' | 'light'
  ) => void;
};
const ThemeContext = createContext<ThemeContextType>();
```

   The code doesn't compile though; what is the problem?

2. The context from question 1 has a provider wrapper called `ThemeProvider`, which is added to the component tree as follows:

```
<ThemeProvider>
  <Header />
  <Main />
</ThemeProvider>
<Footer />
```

   The theme state is `undefined` when destructured from `useContext` in the `Footer` component. What is the problem?

3. Is it possible to have two React contexts in an app?
4. Is it possible to have two Redux stores in an app?
5. The following code dispatches an action to change the theme:

```
function handleChangeTheme({ name, color }: Theme) {
  useDispatch(changeThemeAction(name, color));
}
```

   There is a problem with this code. What is this problem?

6. In a React component, is it possible to use state only required by this component using `useState` as well as state from a Redux store?
7. In this chapter, when we implemented the Redux slice, the action handlers appeared to directly update the state, as in the following example:

```
authorizedAction: (
  state,
  action: PayloadAction<string[]>
) => {
  state.permissions = action.payload;
  state.loading = false;
}
```

   Why are we allowed to mutate the state? I thought that state in React had to be immutable?

Answers

1. `createContext` must be passed a default value when using it with TypeScript. Here's the corrected code:

```
const ThemeContext = createContext<ThemeContextType>({
  name: 'standard',
  color: 'light',
  changeTheme: (name: string, color: 'dark' | 'light') =>
{},
});
```

2. `Footer` must be placed inside `ThemeProvider` as follows:

```
<ThemeProvider>
  <Header />
  <Main />
```

```
    <Footer />
  </ThemeProvider>
```

3. Yes, there is no limit on the number of React contexts in an app.
4. No, only a single Redux store can be added to an app.
5. `useDispatch` can't be used directly to dispatch an action – it returns a function that can be used to dispatch an action:

```
const dispatch = useDispatch();
function handleChangeTheme({ name, color }: Theme) {
  dispatch(changeThemeAction(name, color));
}
```

6. Yes, local state defined using `useState` or `useReducer` can be used alongside shared state from a Redux store.
7. The Redux Toolkit uses a library called **immer** to allow developers to directly update the `state` object without mutating it. For more information on *immer*, see the following link: `https://github.com/immerjs/immer`.

9

Interacting with RESTful APIs

In this chapter, we will build a page that lists blog posts fetched from a REST API, as well as also a form to submit blog posts to the REST API. Through this, we will learn about various approaches to interacting with a REST API from a React component.

The first approach will be using React's `useEffect` hook with the browser's `fetch` function. As part of this process, we learn how to use a type assertion function to strongly type the data from a REST API. We will then use the data loading capability of **React Router** and experience its benefits. After that, we will move on to use a popular library called **React Query** and experience its benefits, before using React Query and React Router together to get the best of both these libraries.

So, in this chapter, we'll cover the following topics:

- Getting set up
- Using the effect hook with fetch
- Posting data with fetch
- Using React Router
- Using React Query
- Using React Router with React Query

Technical requirements

We will use the following technologies in this chapter:

- **Node.js** and **npm**: You can install them from `https://nodejs.org/en/download/`
- **Visual Studio Code**: You can install it from `https://code.visualstudio.com/`

All the code snippets in this chapter can be found online at `https://github.com/PacktPublishing/Learn-React-with-TypeScript-2nd-Edition/tree/main/Chapter9`.

Creating the project

In this section, we will start by creating the project for the app we will build. We will then create a REST API for the app to consume.

Setting up the project

We will develop the app using Visual Studio Code and require a new Create React App-based project setup. We've previously covered this several times, so we will not cover the steps in this chapter – instead, see *Chapter 3*, *Setting Up React and TypeScript*.

We will style the app with Tailwind CSS. We have previously covered how to install and configure Tailwind in Create React App in *Chapter 5*, *Approaches to Styling Frontends*. So, after you have created the React and TypeScript project, install and configure Tailwind.

We will use React Router to load data, so see *Chapter 6*, *Routing with React Router,* for information on how to do this.

We will use **React Hook Form** to implement the form that creates blog posts, and the `@tailwindcss/forms` plugin to style the form. See *Chapter 7*, *Working with Forms,* for a reminder of how to implement these.

Understanding the component structure

The app will be a single page containing a form for adding new posts above a list of all the existing posts. The app will be structured into the following components:

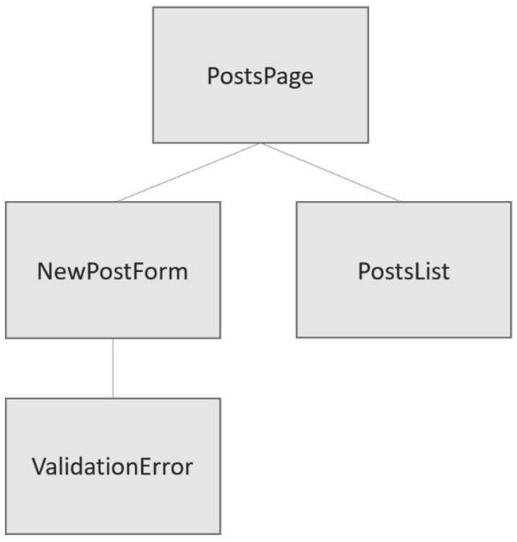

Figure 9.1 – App component structure

Here's a description of these components:

- `PostsPage` will render the whole page by referencing the `NewPostForm` and `PostsLists` components. It will also interact with the REST API.
- `NewPostForm` will render a form that allows a user to enter a new blog post. This will use the `ValidationError` component to render validation error messages. The `ValidationError` component will be the same as the one created in *Chapter 7*.
- `PostsList` will render the list of blog posts.

Right, now we know the component structure, let's create the REST API.

Creating a REST API

We will create a REST API using a tool called **JSON Server**, which allows a REST API to be quickly created. Install JSON Server by running the following command:

```
npm i -D json-server
```

We then define the data behind the API in a JSON file. Create a file called `db.json` in the root of the project containing the following:

```
{
  "posts": [
    {
      "title": "Getting started with fetch",
      "description": "How to interact with backend APIs using
        fetch",
      "id": 1
    },
    {
      "title": "Getting started with useEffect",
      "description": "How to use React's useEffect hook for
        interacting with backend APIs",
      "id": 2
    }
  ]
}
```

The preceding JSON means that the data behind the API will initially contain two blog posts (this code snippet can be copied from `https://github.com/PacktPublishing/Learn-React-with-TypeScript-2nd-Edition/blob/main/Chapter9/useEffect-fetch/db.json`).

Now we need to define an npm script to start the JSON server and handle requests. Open `package.json` and add a script called `server` as follows:

```
{
  ...,
  "scripts": {
    ...,
    "server": "json-server --watch db.json --port 3001 --delay
      2000"
  },
  ...
}
```

The script starts the JSON server and watches the JSON file we just created. We have specified that the API runs on port `3001` so that it doesn't clash with the app running on port `3000`. We have also slowed down the API responses by adding a 2-second delay, which will help us see when data is being fetched from the React app.

In a terminal, start the API by running the script we just created, as follows:

```
npm run server
```

After a few seconds, the API starts. To check that the API is working correctly, open a browser and enter the following address: `http://localhost:3001/posts`. The blog post data should appear in the browser as follows:

Figure 9.2 – Blog posts REST API

For more information on JSON Server, see the following link: `https://github.com/typicode/json-server`.

Now that the project is set up with a REST API, keeping the API running, next, we will learn how to interact with the REST API using `useEffect`.

Using the effect hook with fetch

In this section, we will create a page that lists the blog posts returned from the REST API we just created. We will use the browser's `fetch` function and React's `useEffect` hook to interact with the REST API.

Getting blog posts using fetch

We will start by creating a function that gets blog posts from the REST API using the browser's `fetch` function; we will store the API URL in an environment variable. To do this, carry out the following steps:

1. The same URL will be used to get, as well as save, new blog posts to the REST API. We will store this URL in an environment variable. So, create a file called `.env` in the root of the project containing this variable, as follows:

   ```
   REACT_APP_API_URL = http://localhost:3001/posts/
   ```

   This environment variable is injected into the code at build time and can be accessed by code using `process.env.REACT_APP_API_URL`. Environment variables in Create React App projects must be prefixed with `React_APP_`. For more information on environment variables, see the following link: `https://create-react-app.dev/docs/adding-custom-environment-variables/`.

2. Now, create a folder called `posts` in the `src` folder for all the files for the blog post feature.

3. Create a file called `getPosts.ts` in the `posts` folder. In this file, add the following function that gets the blog posts:

   ```
   export async function getPosts() {
     const response = await fetch(
       process.env.REACT_APP_API_URL!
     );
     const body = await response.json()
     return body;
   }
   ```

The `fetch` function has an argument for the URL to the REST API. We have used the `REACT_APP_API_URL` environment variable to specify this URL. Environment variable values can be `undefined`, but we know this isn't the case, so we have added a **not null assertion** (`!`) after it.

> **Note**
> A not null assertion operator is a special operator in TypeScript. It is used to inform the TypeScript compiler that the expression before it can't be `null` or `undefined`.

`fetch` returns a `Response` object and we call its `json` method to get the response body in JSON format. The `json` method is asynchronous, so we need to `await` it.

For more information on `fetch`, see the following link: `https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API`.

That completes an initial version of `getPosts`. However, the return value type from `getPosts` is currently `any`, which means no type checking will occur on it. We will improve this next.

Strongly typing response data

In *Chapter 2*, *Introducing TypeScript*, we learned how to make unknown data strongly typed using the `unknown` type and type predicates. We will use the `unknown` type with a slightly different TypeScript feature called a **type assertion function** to type the response data in the `getPosts` function. Carry out the following steps:

1. Add a type assertion to the JSON response so that the `body` variable has a type of `unknown`:

```
export async function getPosts() {
  const response = await fetch(postsUrl);
  const body = (await response.json()) as unknown;
  return body;
}
```

2. Next, add the following type assertion function beneath `getPosts`:

```
export function assertIsPosts(
  postsData: unknown
    ): asserts postsData is PostData[] {
}
```

Notice the return type annotation: `asserts postsData is PostData[]`. This is called an **assertion signature** and specifies that the `postsData` parameter is of the `PostData[]` type if no error occurs in the function execution.

Don't worry about the compile error where `PostData` is referenced – we will create the `PostData` type in *step 8*.

3. Let's carry on with the implementation of `assertIsPosts`. It will be a series of checks on the `postsData` parameter and it will throw an exception if a check fails. Start the implementation by checking that `postsData` is an array:

```
export function assertIsPosts(
  postsData: unknown
): asserts postsData is PostData[] {
  if (!Array.isArray(postsData)) {
    throw new Error("posts isn't an array");
  }
  if (postsData.length === 0) {
    return;
  }
}
```

4. Now, let's do a check on the array items to see whether they have an `id` property:

```
export function assertIsPosts(
  postsData: unknown
): asserts postsData is PostData[] {
  ...
  postsData.forEach((post) => {
    if (!('id' in post)) {
      throw new Error("post doesn't contain id");
    }
    if (typeof post.id !== 'number') {
      throw new Error('id is not a number');
    }
  });
}
```

We loop around all the posts using the array's `forEach` method. Inside the loop, we check that the `id` property exists using the `in` operator. We also check that the `id` value is of the `number` type using the `typeof` operator.

5. We can do similar checks for the `title` and `description` properties:

```
export function assertIsPosts(
  postsData: unknown
): asserts postsData is PostData[] {
  …
  postsData.forEach((post) => {
    ...
    if '!('ti'le' in post)) {
      throw new Err"r("post do'sn't contain ti"le");
    }
    if (typeof post.title !'= 'str'ng') {
      throw new Err'r('title is not a str'ng');
    }
    if '!('descript'on' in post)) {
      throw new Err"r("post do'sn't contain
        descript"on");
    }
    if (typeof post.description !'= 'str'ng') {
      throw new Err'r('description is not a str'ng');
    }
  });
}
```

That completes the implementation of the type assertion function.

6. Moving back to `getPosts`, add a call to the `assert` function:

```
export async function getPosts() {
  const response = await fetch(postsUrl);
  const body = (await response.json()) as unknown;
  assertIsPosts(body);
  return body;
}
```

The `body` variable will now be of the `PostData[]` type after a successful call to `assertIsPosts`. You can hover over the `body` variable in the return statement to confirm this.

7. The final steps are to add the `PostData` type. Add an import statement at the top of `getPosts.ts` as follows to import `PostData`:

```
import { PostData } from './types';
```

   The file will still have compile errors because the `types` file doesn't exist yet – we'll do this in the next step.

8. Add a file called `types.ts` in the `posts` folder with the following definition for the `PostData` type:

```
export type PostData = {
  id: number;
  title: string;
  description: string;
};
```

   This type represents a blog post from the REST API.

Now, we have a strongly typed function that gets blog posts from the REST API. Next, we will create a React component that lists the blog posts.

Creating a blog posts list component

We will create a React component that takes in the blog post data and renders it in a list. Carry out the following steps:

1. Create a file called `PostsList.tsx` in the `posts` folder with the following import statement:

```
import { PostData } from './types';
```

2. Next, start to implement the component as follows:

```
type Props = {
  posts: PostData[];
};
export function PostsList({ posts }: Props) {

}
```

   The component has a prop called `posts` that will contain the blog posts.

3. Now, render the blog posts in an unordered list, as follows:

```
export function PostsList({ posts }: Props) {
  return (
```

```
      <ul className="list-none">
        {posts.map((post) => (
          <li key={post.id} className="border-b py-4">
            <h3 className="text-slate-900 font-bold">
              {post.title}
            </h3>
            <p className=" text-slate-900 ">{post.
description}</p>
          </li>
        ))}
      </ul>
    );
}
```

The Tailwind CSS classes add gray lines between the blog posts with bold titles.

That completes the `PostsList` component. Next, we will create a page component that references the `PostsList` component.

Creating a blog posts page component

We will create a blog posts page component that gets blog post data using the `getPosts` function and renders it using the `PostsList` component we just created. Carry out the following steps:

1. Create a file called `PostsPage.tsx` in the `posts` folder components with the following import statements:

```
import { useEffect, useState } from 'react';
import { getPosts } from './getPosts';
import { PostData } from './types';
import { PostsList } from './PostsList';
```

We have imported the `getPosts` function, the `PostList` component, and the `PostData` type we created in the last section. We have also imported the `useState` and `useEffect` hooks from React. We will use React state to store the blog posts and use `useEffect` to call `getPosts` when the page component is mounted.

2. Start the implementation of the page component by defining the state for the blog posts and whether they are being fetched:

```
export function PostsPage() {
  const [isLoading, setIsLoading] = useState(true);
  const [posts, setPosts] = useState<PostData[]>([]);
}
```

3. Next, call the `getPosts` function using the `useEffect` hook as follows:

```
export function PostsPage() {
  …
  useEffect(() => {
    let cancel = false;
    getPosts().then((data) => {
      if (!cancel) {
        setPosts(data);
        setIsLoading(false);
      }
    });
    return () => {
      cancel = true;
    };
  }, []);
}
```

We use the older promise syntax when calling `getPosts` because the newer `async/await` syntax can't be directly used within `useEffect`.

If the `PostsPage` component is unmounted while the call to `getPosts` is still in progress, the setting of the `data` and `isLoading` state variables will result in an error. For this reason, we have used a `cancel` flag to ensure that the component is still mounted when the `data` and `isLoading` state variables are set.

We have also specified an empty array as the effect dependencies so that the effect only runs when the component is mounted.

4. Add a loading indicator while the data is being fetched after the call to `useEffect`:

```
export function PostsPage() {
  ...
  useEffect(...);
  if (isLoading) {
    return (
      <div className="w-96 mx-auto mt-6">
        Loading ...
      </div>
    );
  }
}
```

The Tailwind CSS classes position the loading indicator horizontally in the center of the page with a bit of margin above it.

5. Finally, render a page title and the posts list after the conditional loading indicator:

```
export function PostsPage() {
  ...
  if (isLoading) {
    return (
      <div className="w-96 mx-auto mt-6">
        Loading ...
      </div>
    );
  }
  return (
    <div className="w-96 mx-auto mt-6">
      <h2 className="text-xl text-slate-900 font-
bold">Posts</h2>
      <PostsList posts={posts} />
    </div>
  );
}
```

The Tailwind CSS classes position the list in the center of the page with a bit of margin above. A large **Posts** title is also rendered above the list in a dark gray color.

6. Now, open `App.tsx` and replace its contents with the following so that it renders the page component we just created:

```
import { PostsPage } from './posts/PostsPage';

function App() {
  return <PostsPage />;
}
export default App;
```

7. Run the app by running `npm start` in a new terminal separate from the one running the REST API. The loading indicator will appear briefly as the data is being fetched:

Figure 9.3 – Loading indicator

The blog post list will then appear as follows:

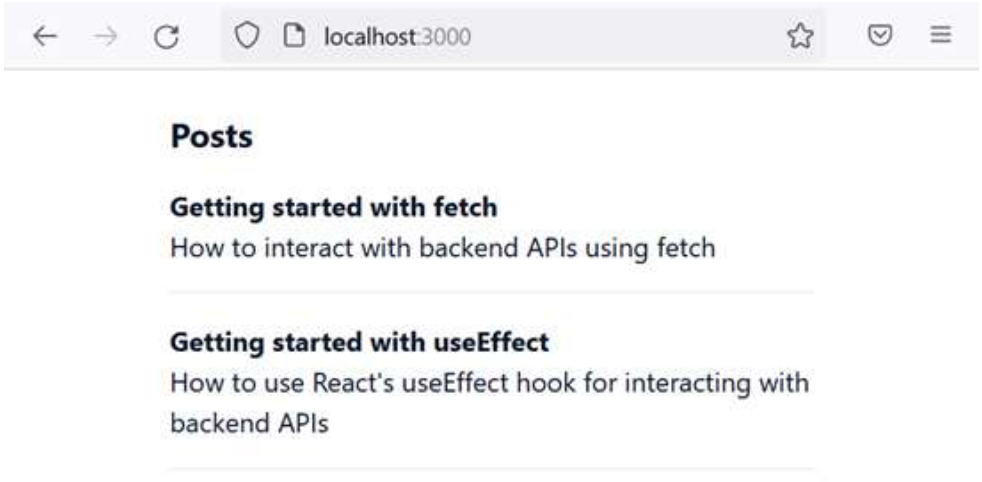

Figure 9.4 – Blog posts list

That completes this version of the `PostsPage` component.

Here are the key points we learned in this section on interacting with HTTP `GET` requests in a REST API using `fetch` and `useEffect`:

- `fetch` will make the actual HTTP request that has the REST API's URL as a parameter
- A type assertion function can be used to strongly type response data
- `useEffect` can trigger the `fetch` call when the component that holds the data in state is mounted
- A flag can be used inside `useEffect` to check that the component hasn't been unmounted during the HTTP request before the data state is set

Still keeping the app and REST API running, in the next section, we will learn how to post data to a REST API using `fetch`.

Posting data with fetch

In this section, we will create a form that submits a new blog post to our REST API. We will create a function that uses `fetch` to post to the REST API. That function will be called in the form's submit handler.

Creating new blog posts using fetch

We will start by creating the function that sends a new blog post to the REST API. This will use the browser's `fetch` function, but this time, using an HTTP `POST` request. Carry out the following steps:

1. We will start by opening `types.ts` in the `posts` folder and adding the following two types:

```
export type NewPostData = {
  title: string;
  description: string;
};

export type SavedPostData = {
  id: number;
};
```

   The first type represents a new blog post, and the second type represents the data from the API when the blog post is successfully saved.

2. Create a new file called `savePost.ts` in the `posts` folder and add the following import statement:

```
import { NewPostData, SavedPostData } from './types';
```

   We have also imported the types we just created.

3. Start to implement the `savePost` function as follows:

```
export async function savePost(
  newPostData: NewPostData
) {
  const response = await fetch(
    process.env.REACT_APP_API_URL!,
    {
      method: 'POST',
      body: JSON.stringify(newPostData),
      headers: {
```

```
          'Content-Type': 'application/json',
        },
      }
    );
  }
```

The `savePost` function has a parameter, `newPostData`, containing the title and description for the new blog post, and sends it to the REST API using `fetch`. A second argument has been specified in the `fetch` call to specify that an HTTP `POST` request should be used and that the new blog post data should be included in the request body. The request body has also been declared as being in JSON format.

4. Next, strongly type the response body as follows:

```
export async function savePost(newPostData: NewPostData)
{
  const response = await fetch( ... );
  const body = (await response.json()) as unknown;
  assertIsSavedPost(body);
}
```

We set the response body as having the `unknown` type and then use a type assertion function to give it a specific type. This will raise a compile error until we implement `assertIsSavedPost` in *step 6*.

5. Finish the implementation of `savePost` by merging the blog post ID from the response with the blog post title and description supplied to the function:

```
export async function savePost(newPostData: NewPostData)
{
  ...
  return { ...newPostData, ...body };
}
```

So, the object returned from the function will be a new blog post with the ID from the REST API.

6. The last step is to implement the type assertion function:

```
function assertIsSavedPost(
  post: any
): asserts post is SavedPostData {
  if (!('id' in post)) {
    throw new Error("post doesn't contain id");
```

```
    }
    if (typeof post.id !== 'number') {
      throw new Error('id is not a number');
    }
  }
```

The function checks whether the response data contains a numeric `id` property, and if it does, it asserts that the data is of the `SavedPostData` type.

That completes the implementation of the `savePost` function. Next, we will add a form component that allows the user to enter new blog posts.

Creating a blog post form component

We will create a component that contains a form that captures a new blog post. When the form is submitted, it will call the `savePost` function we just created.

We will use React Hook Form to implement the form along with a `ValidationError` component. We covered React Hook Form and the `ValidationError` component in detail in *Chapter 7*, so the implementation steps won't be covered in much detail.

Carry out the following steps:

1. We will start by creating a `ValidationError` component that will render form validation errors. Create a file called `ValidationError.tsx` in the `posts` folder with the following content:

```
import { FieldError } from 'react-hook-form';

type Props = {
  fieldError: FieldError | undefined,
};
export function ValidationError({ fieldError }: Props) {
  if (!fieldError) {
    return null;
  }
  return (
    <div role="alert" className="text-red-500 text-xs
      mt-1">
      {fieldError.message}
    </div>
  );
}
```

2. Create a new file in the `posts` folder called `NewPostForm.tsx`. This will contain a form to capture the title and description for a new blog post. Add the following import statements to the file:

```
import { FieldError, useForm } from 'react-hook-form';
import { ValidationError } from './ValidationError';
import { NewPostData } from './types';
```

3. Start to implement the form component as follows:

```
type Props = {
  onSave: (newPost: NewPostData) => void;
};
export function NewPostForm({ onSave }: Props) {
}
```

The component has a prop for saving a new blog post so that the interaction with the REST API can be handled outside of this form component.

4. Now, destructure the `register` and `handleSubmit` functions and useful state variables from React Hook Form's `useForm` hook:

```
type Props = {
  onSave: (newPost: NewPostData) => void;
};
export function NewPostForm({ onSave }: Props) {
  const {
    register,
    handleSubmit,
    formState: { errors, isSubmitting, isSubmitSuccessful
},
  } = useForm<NewPostData>();
}
```

We pass the type for the new post data into the `useForm` hook so it knows the shape of the data to capture.

5. Create a variable for the field container style and a function for the editor style:

```
export function NewPostForm({ onSave }: Props) {
  ...
  const fieldStyle = 'flex flex-col mb-2';
  function getEditorStyle(
```

```
    fieldError: FieldError | undefined
  ) {
    return fieldError ? 'border-red-500' : '';
  }
}
```

6. Render `title` and `description` fields in a `form` element as follows:

```
export function NewPostForm({ onSave }: Props) {
  ...
  return (
    <form
      noValidate
      className="border-b py-4"
      onSubmit={handleSubmit(onSave)}
    >
      <div className={fieldStyle}>
        <label htmlFor="title">Title</label>
        <input
          type="text"
          id="title"
          {...register('title', {
            required: 'You must enter a title',
          })}
          className={getEditorStyle(errors.title)}
        />
        <ValidationError fieldError={errors.title} />
      </div>
      <div className={fieldStyle}>
        <label htmlFor="description">Description</label>
        <textarea
          id="description"
          {...register('description', {
            required: 'You must enter the description',
          })}
          className={getEditorStyle(errors.description)}
        />
```

```
          <ValidationError fieldError={errors.description}
            />
        </div>
      </form>
    );
  }
```

7. Lastly, render a `Save` button and the success message:

```
  <form
    noValidate
    className="border-b py-4"
    onSubmit={handleSubmit(onSave)}
  >
    <div className={fieldStyle}> ... </div>
    <div className={fieldStyle}> ... </div>
    <div className={fieldStyle}>
      <button
        type="submit"
        disabled={isSubmitting}
        className="mt-2 h-10 px-6 font-semibold bg-black
          text-white"
      >
        Save
     </button>
      {isSubmitSuccessful && (
        <div
          role="alert"
          className="text-green-500 text-xs mt-1"
        >
          The post was successfully saved
       </div>
      )}
    </div>
  </form>
```

That completes the implementation of the `NewPostForm` component.

8. Now open `PostPage.tsx` and import the `NewPostForm` component and the `savePost` function we created earlier. Also, import the `NewPostData` type:

```
import { useEffect, useState } from 'react';
import { getPosts } from './getPosts';
import { PostData, NewPostData } from './types';
import { PostsList } from './PostsList';
import { savePost } from './savePost';
import { NewPostForm } from './NewPostForm';
```

9. In the `PostPage` JSX, add the `NewPostForm` form above the `PostsList`:

```
<div className="w-96 mx-auto mt-6">
  <h2 className="text-xl text-slate-900 font-
    bold">Posts</h2>
  <NewPostForm onSave={handleSave} />
  <PostsList posts={posts} />
</div>;
```

10. Add the save handler function just below the effect that gets blog posts:

```
useEffect(() => {
  ...
}, []);
async function handleSave(newPostData: NewPostData) {
  const newPost = await savePost(newPostData);
  setPosts([newPost, ...posts]);
}
```

11. The hander calls `savePost` with the data from the form. After the post has been saved, it is added to the start of the `posts` array.

12. In the running app, the new blog post form will appear above the blog post list, as follows:

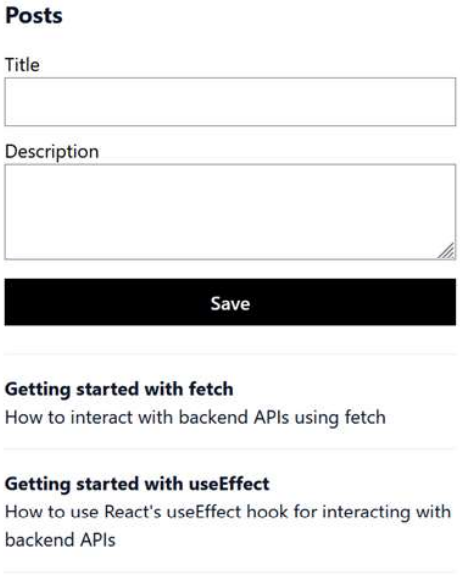

Figure 9.5 – New blog post form above the posts list

13. Fill in the form with a new blog post and press the **Save** button. The new post should appear at the top of the list after a couple of seconds.

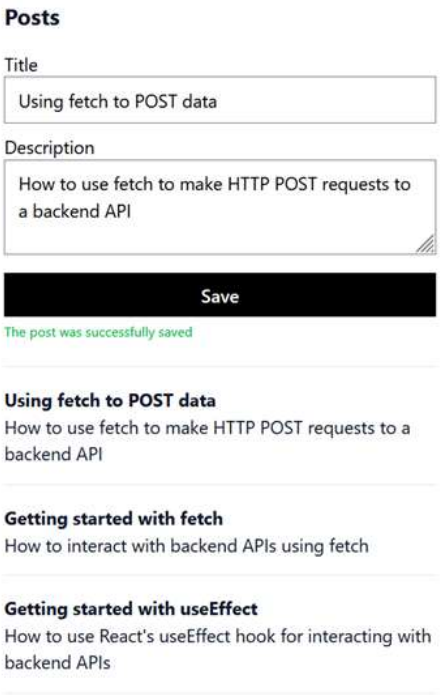

Figure 9.6 – New blog post added to posts list

That completes the implementation of the form and its integration into the blog posts page.

Here are a couple of key points we learned in this section on posting data with `fetch`:

- The second parameter in the `fetch` function allows the HTTP method to be specified. In this section, we used this parameter to make an HTTP `POST` request.
- The second parameter in `fetch` also allows the request body to be supplied.

Once again, keeping the app and REST API running, in the next section, we will use React Router's data-fetching capabilities to simplify our data-fetching code.

Using React Router

In this section, we will learn about how React Router can integrate with the data-fetching process. We will use this knowledge to simplify the code that fetches blog posts in our app.

Understanding React Router data loading

React Router data loading is similar to React Router forms, which we learned about in *Chapter 7*. Instead of defining an action that handles form submission, we define a **loader** that handles data loading. The following code snippet defines a loader on a `some-page` route:

```
const router = createBrowserRouter([
  ...,
  {
    path: '/some-page',
    element: <SomePage />,
    loader: async () => {
      const response = fetch('https://somewhere');
      return await response.json();
    }
  },
  ...
]);
```

React Router calls the loader to get the data before it renders the component defined on the route. The data is then available in the component via a `useLoaderData` hook:

```
export function SomePage() {
  const data = useLoaderData();
  ...
}
```