Redux Toolkit Setup with Next.js

WHAT YOU'LL LEARN

- How to set up and use Redux Toolkit with the Next.js framework

PREREQUISITES

- Familiarity with ES6 syntax and features
- Knowledge of React terminology: JSX, State, Function Components, Props, and Hooks
- Understanding of Redux terms and concepts
- Working through the Quick Start tutorial and TypeScript Quick Start tutorial is recommended, and ideally the full Redux Essentials tutorial as well

Introduction

Next.js is a popular server side rendering framework for React that presents some unique challenges for using Redux properly. These challenges include:

- **Per-request safe Redux store creation**: A Next.js server can handle multiple requests simultaneously. This means that the Redux store should be created per request and that the store should not be shared across requests.
- **SSR-friendly store hydration**: Next.js applications are rendered twice, first on the server and again on the client. Failure to render the same page contents on both the client and the server will result in a "hydration error". So the Redux store will have to be initialized on the server and then re-initialized on the client with the same data in order to avoid hydration issues.
- **SPA routing support**: Next.js supports a hybrid model for client side routing. A customer's first page load will get an SSR result from the server. Subsequent page navigation will be handled by the client. This means that with a singleton store defined in the layout, route-specific data will need to be selectively reset on route navigation, while non-route-specific data will need to be retained in the store.

- **Server caching friendly**: Recent versions of Next.js (specifically applications using the App Router architecture) support aggressive server caching. The ideal store architecture should be compatible with this caching.

There are two architectures for a Next.js application: the Pages Router and the App Router.

The Pages Router is the original architecture for Next.js. If you're using the Pages Router, Redux setup is primarily handled by using the `next-redux-wrapper` library, which integrates a Redux store with the Pages router data fetching methods like `getServerSideProps`.

This guide will focus on the App Router architecture, as it is the new default architecture option for Next.js.

How to Read This Guide

This page assumes that you already have an existing Next.js application based on the App Router architecture.

If you want to follow along, you can create a new empty Next project with `npx create-next-app my-app` - the default prompts will set up a new project with the App Router enabled. Then, add `@reduxjs/toolkit` and `react-redux` as dependencies.

You can also create a new Next+Redux project with `npx create-next-app --example with-redux my-app`, which includes the initial setup pieces described in this page.

The App Router Architecture and Redux

The primary new feature of the Next.js App Router is the addition of support for React Server Components (RSCs). RSCs are a special type of React component that only renders on the server, as opposed to "client" components that render on **both** the client and the server. RSCs can be defined as `async` functions and return promises during rendering as they make async requests for data to render.

RSCs ability to block for data requests means that with the App Router you no longer have `getServerSideProps` to fetch data for rendering. Any component in the tree can make asychronous requests for data. While this is very convenient it also means thats if you define global variables (like the Redux store) they will be shared across requests. This is a problem because the Redux store could be contaminated with data from other requests.

Based on the architecture of the App Router we have these general recommendations for appropriate use of Redux:

- **No global stores** - Because the Redux store is shared across requests, it should not be defined as a global variable. Instead, the store should be created per request.
- **RSCs should not read or write the Redux store** - RSCs cannot use hooks or context. They aren't meant to be stateful. Having an RSC read or write values from a global store violates the architecture of the Next.js App Router.
- **The store should only contain mutable data** - We recommend that you use your Redux sparingly for data intended to be global and mutable.

These recommendations are specific to applications written with the Next.js App Router. Single Page Applications (SPAs) don't execute on the server and therefore can define stores as global variables. SPAs don't need to worry about RSCs since they don't exist in SPAs. And singleton stores can store whatever data you want.

Folder Structure

Next apps can be created to have the `/app` folder either at the root, or nested under `/src/app`. Your Redux logic should go in a separate folder, alongside the `/app` folder. It's common to put the Redux logic in a folder named `/lib`, but not required.

The file and folder structure inside of that `/lib` folder is up to you, but we generally recommend a "feature folder"-based structure for the Redux logic.

A typical example might look like:

```
/app
  layout.tsx
  page.tsx
  StoreProvider.tsx
/lib
  store.ts
  /features
    /todos
      todosSlice.ts
```

We'll use that approach for this guide.

Initial Setup

Similar to the the RTK TypeScript Tutorial, we need to create a file for the Redux store, as well as the inferred `RootState` and `AppDispatch` types.

However, Next's multi-page architecture requires some differences from that single-page app setup.

Creating a Redux Store per Request

The first change is to move from defining store as a global to defining a `makeStore` function that returns a new store for each request:

TypeScript JavaScript

lib/store.ts

```ts
import { configureStore } from '@reduxjs/toolkit'

export const makeStore = () => {
  return configureStore({
    reducer: {},
  })
}

// Infer the type of makeStore
export type AppStore = ReturnType<typeof makeStore>
// Infer the `RootState` and `AppDispatch` types from the store itself
export type RootState = ReturnType<AppStore['getState']>
export type AppDispatch = AppStore['dispatch']
```

Now we have a function, `makeStore`, that we can use to create a store instance per-request while retaining the strong type safety (if you choose to use TypeScript) that Redux Toolkit provides.

We don't have a `store` variable exported, but we can infer the `RootState` and `AppDispatch` types from the return type of `makeStore`.

You'll also want to create and export pre-typed versions of the React-Redux hooks as well, to simplify usage later:

TypeScript JavaScript

lib/hooks.ts

```ts
import { useDispatch, useSelector, useStore } from 'react-redux'
import type { RootState, AppDispatch, AppStore } from './store'

// Use throughout your app instead of plain `useDispatch` and `useSelector`
export const useAppDispatch = useDispatch.withTypes<AppDispatch>()
export const useAppSelector = useSelector.withTypes<RootState>()
export const useAppStore = useStore.withTypes<AppStore>()
```

Providing the Store

To use this new `makeStore` function we need to create a new "client" component that will create the store and share it using the React-Redux `Provider` component.

TypeScript JavaScript

app/StoreProvider.tsx

```tsx
'use client'
import { useRef } from 'react'
import { Provider } from 'react-redux'
import { makeStore, AppStore } from '../lib/store'

export default function StoreProvider({
  children,
}: {
  children: React.ReactNode
}) {
  const storeRef = useRef<AppStore>()
  if (!storeRef.current) {
    // Create the store instance the first time this renders
    storeRef.current = makeStore()
  }

  return <Provider store={storeRef.current}>{children}</Provider>
}
```

In this example code we are ensuring that this client component is re-render safe by checking the value of the reference to ensure that the store is only created once. This component will only be rendered once per request on the server, but might be re-rendered multiple times on the client if there are stateful client components located above this component in the tree, or if this component also contains other mutable state that causes a re-render.

> **WHY CLIENT COMPONENTS?**
>
> Any component that interacts with the Redux store (creating it, providing it, reading from it, or writing to it) needs to be a client component. This is because **accessing the store requires React context, and context is only available in client components.**

The next step is to **include the `StoreProvider` anywhere in the tree above where the store is used**. You can locate the store in the layout component if all the routes using that layout need the store. Or if the store is only used in a specific route you can create and provide the store in that route handler. In all client components further down the tree, you can use the store exactly as you would normally using the hooks provided by `react-redux`.

Loading Initial Data

If you need to initialize the store with data from the parent component, then define that data as a prop on the client `StoreProvider` component and use a Redux action on the slice to set the data in the store as shown below.

TypeScript JavaScript

src/app/StoreProvider.tsx

```tsx
'use client'
import { useRef } from 'react'
import { Provider } from 'react-redux'
import { makeStore, AppStore } from '../lib/store'
import { initializeCount } from '../lib/features/counter/counterSlice'

export default function StoreProvider({
  count,
  children,
}: {
  count: number
  children: React.ReactNode
}) {
  const storeRef = useRef<AppStore | null>(null)
  if (!storeRef.current) {
    storeRef.current = makeStore()
    storeRef.current.dispatch(initializeCount(count))
  }
```

```
  return <Provider store={storeRef.current}>{children}</Provider>
}
```

Additional Configuration

Per-route state

If you use Next.js's support for client side SPA-style navigation by using `next/navigation`, then when customers navigate from page to page only the route component will be re-rendered. This means that if you have a Redux store created and provided in the layout component it will be preserved across route changes. This is not a problem if you are only using the store for global, mutable data. However, if you are using the store for per-route data then you will need to reset the route-specific data in the store when the route changes.

Shown below is a `ProductName` example component that uses the Redux store to manage the mutable name of a product. The `ProductName` component part of a product detail route. In order to ensure that we have the correct name in the store we need to set the value in the store any time the `ProductName` component is initially rendered, which happens on any route change to the product detail route.

TypeScript JavaScript

app/ProductName.tsx

```tsx
'use client'
import { useRef } from 'react'
import { useAppSelector, useAppDispatch, useAppStore } from '../lib/hooks'
import {
  initializeProduct,
  setProductName,
  Product,
} from '../lib/features/product/productSlice'

export default function ProductName({ product }: { product: Product }) {
  // Initialize the store with the product information
  const store = useAppStore()
  const initialized = useRef(false)
  if (!initialized.current) {
    store.dispatch(initializeProduct(product))
    initialized.current = true
  }
```

```
  const name = useAppSelector((state) => state.product.name)
  const dispatch = useAppDispatch()

  return (
    <input
      value={name}
      onChange={(e) => dispatch(setProductName(e.target.value))}
    />
  )
}
```

Here we are using the same intialization pattern as before, of dispatching actions to the store, to set the route-specific data. The `initialized` ref is used to ensure that the store is only initialized once per route change.

It is worth noting that initializing the store with a `useEffect` would not work because `useEffect` only runs on the client. This would result in hydration errors or flicker because the result from a server side render would not match the result from the client side render.

Caching

The App Router has four separate caches including `fetch` request and route caches. The most likely cache to cause issues is the route cache. If you have an application that accepts login you may have routes (e.g. the home route, `/`) that render different data based on the user you will need to disable the route cache by using the `dynamic` export from the route handler:

TypeScript JavaScript

```
export const dynamic = 'force-dynamic'
```

After a mutation you should also invalidate the cache by calling `revalidatePath` or `revalidateTag` as appropriate.

RTK Query

We recommend using RTK Query for data fetching **on the client only**. Data fetching on the server should use `fetch` requests from `async` RSCs.

You can learn more about Redux Toolkit Query in the Redux Toolkit Query tutorial.

> **NOTE**
>
> In the future, RTK Query may be able to receive data fetched on the server via React Server Components, but that is a future capability that will require changes to both React and RTK Query.

Checking Your Work

There are three key areas that you should check to ensure that you have set up Redux Toolkit correctly:

- **Server Side Rendering** - Check the HTML output of the server to ensure that the data in the Redux store is present in the server side rendered output.
- **Route Change** - Navigate between pages on the same route as well as between different routes to ensure that route-specific data is initialized properly.
- **Mutations** - Check that the store is compatible with the Next.js App Router caches by performing a mutation and then navigating away from the route and back to the original route to ensure that the data is updated.

Overall Recommendations

The App Router presents a dramatically different archtecture for React applications from either the Pages Router or a SPA application. We recommend rethinking your approach to state management in the light of this new architecture. In SPA applications it's not unusual to have a large store that contains all the data, both mutable and immutable, required to drive the application. For App Router applications we recommend that you should:

- **only use Redux for globally shared, mutable data**
- use a combination of Next.js state (search params, route parameters, form state, etc.), React context and React hooks for all other state management.

What You've Learned

That was a brief overview of how to set up and use Redux Toolkit with the App Router:

> **SUMMARY**

- **Create a Redux store per request by using `configureStore` wrapped in a `makeStore` function**
- **Provide the Redux store to the React application components using a "client" component**
- **Only interact with the Redux store in client components** because only client components have access to React context
- **Use the store as you normally would using the hooks provided in React-Redux**
- **You need to account for the case where you have per-route state in a global store located in the layout**

What's Next?

We recommend going through **the "Redux Essentials" and "Redux Fundamentals" tutorials in the Redux core docs**, which will give you a complete understanding of how Redux works, what Redux Toolkit does, and how to use it correctly.

*Last updated on **Mar 27, 2024***