LogRocket
Frontend Analytics

Persist state with Redux Persist using Redux Toolkit in React

June 3, 2022 · 5 min read

With the Redux Persist library, developers can save the Redux store in persistent storage, for example, the local storage. Therefore, even after refreshing the browser, the site state will still be preserved. Redux Persist also includes methods that allow us to customize the state that gets persisted and rehydrated, all with an easily understandable API.

In this article, we'll learn how to use Redux Persist with Redux Toolkit in React. To follow along with this article, you should be familiar with React and Redux Toolkit. You should also have Node.js installed on your machine.

Setting up React

I've already created an app that uses Redux Toolkit for state management. We'll use it in this tutorial to learn how Redux Persist works. To start, clone the GitHub repo. You can do so with the following commands:

```
$ git clone https://github.com/Tammibriggs/auth-app.git


$ cd auth-app


$ npm install
```

Next, we can start the app with the `npm start` command. In our app, we'll see a form that has a field to enter our name and email. After entering the required inputs and submitting the form, we'll be taken to our profile page, which will look similar to the following image:

When we refresh the browser, our data will be lost. Let's learn how to use Redux Persist to save the state in persistent storage so that even after a refresh, the data will still remain intact. We'll also learn how to customize what's persisted and specify how incoming states will be merged. Let's get started!

Persisting state with Redux Persist

First, we'll add Redux Persist to our app with the following command:

```
$ npm i redux-persist
```

Next, we need to modify our store, which we'll find the in `redux` folder in the `src` directory of the cloned app. Currently, our store looks like the code below:

```
// src/redux/store.js
import { configureStore } from "@reduxjs/toolkit";
import userReducer from "./slices/userSlice";

export const store = configureStore({
  reducer: userReducer,
  devTools: process.env.NODE_ENV !== 'production',
})
```

We'll make the following modifications to our `store.js` file to use Redux Persist:

```
// src/redux/store.js
import { configureStore } from "@reduxjs/toolkit";
import userReducer from "./slices/userSlice";
import storage from 'redux-persist/lib/storage';
import { persistReducer, persistStore } from 'redux-persist';
import thunk from 'redux-thunk';

const persistConfig = {
  key: 'root',
  storage,
}

const persistedReducer = persistReducer(persistConfig, userReducer)

export const store = configureStore({
  reducer: persistedReducer,
  devTools: process.env.NODE_ENV !== 'production',
  middleware: [thunk]
})
```

In the code above, we replaced the value of the `reducer` property in the store from `userReducer` to `persistedReducer`, which is an enhanced reducer with configuration to persist the `userReducer` state to local storage. Aside from local storage, we can also use other storage engines like `sessionStorage` and Redux Persist Cookie Storage Adapter.

To use a different storage engine, we just need to modify the value of the `storage` property of `persistConfig` with the storage engine we want to use. For example, to use the `sessionStorage` engine, we'll first import it as follows:

```
import storageSession from 'reduxjs-toolkit-persist/lib/storage/session'
```

Then, modify `persistConfig` to look like the following code:

```
const persistConfig = {
  key: 'root',f
  storageSession,
}
```

In the modification to the store above, we also included the Thunk middleware, which will intercept and stop non-serializable values in action before they get to the reducer. When using Redux Persist without using the Thunk middleware, we'd get an error in the browser's console reading `a non-serializable value was detected in the state`.

Finally, we passed our store as a parameter to `persistStore`, which is the function that persists and rehydrates the state. With this function, our store will be saved to the local storage, and even after a browser refresh, our data will still remain.

In most use cases, we might want to delay the rendering of our app's UI until the persisted data is available in the Redux store. For that, Redux Persist includes the `PersistGate` component. To use `PersistGate`, go to the `index.js` file in the `src` directory and add the following import:

```
// src/index.js
import { persistor, store } from './redux/store';
import { PersistGate } from 'redux-persist/integration/react';
```

Now, modify the `render` function call to look like the code below:

```
// src/index.js
root.render(
  <React.StrictMode>
    <Provider store={store}>
      <PersistGate loading={null} persistor={persistor}>
        <App />
      </PersistGate>
    </Provider>
  </React.StrictMode>
);
```

In this section, we covered the basic setup when using Redux Persist. Now, let's explore the available options and use cases for Redux Persist.

Nested persists using Redux Persist

If we have two or more reducers in Redux Toolkit, like `userReducer` and `notesReducer`, and we want to add them to our store, we'll likely configure the store as follows:

```
const store = configureStore({
  reducer: {
    user: userReducer,
    notes: notesReducer
  },
})
```

We can also use `combineReducers` as follows, which does the same thing:

```
const rootReducer = combineReducers({
  user: userReducer,
  notes: NotesReducer
})

const store = configureStore({
  reducer: rootReducer
})
```

To use Redux Persist in this case, we'll supply `rootReducer` as a parameter of `persistReducer`, then replace `rootReducer` in our store with the persisted reducer as follows:

```
const rootReducer = combineReducers({
  user: userReducer,
  notes: NotesReducer
})

const persistedReducer = persistReducer(persistConfig, rootReducer)

const store = configureStore({
  reducer: persistedReducer
})
```

However, what if we want to set a different configuration? For example, let's say we want to change the storage engine for the `userReducer` to `sessionStorage`. To do so, we can use nested persists, a feature that allows us to nest `persistReducer`, giving us the ability to set different configurations for reducers.

Below is an example of a nested persist where I'm changing the storage of the `userReducer` to `sessionStorage`:

```
const rootPersistConfig = {
  key: 'root',
  storage,
}

const userPersistConfig = {
  key: 'user',
  storage: storageSession,
}

const rootReducer = combineReducers({
  user: persistReducer(userPersistConfig, userReducer),
  notes: notesReducer
})

const persistedReducer = persistReducer(rootPersistConfig, rootReducer)

const store = configureStore({
  reducer: persistedReducer
```

Specify how the incoming state is merged

Merging involves saving the persisted state back in the Redux store. When our app launches, our initial state is set. Shortly after, Redux Persist retrieves our persisted state from storage, then overrides any initial state. This process works automatically.

By default, the merging process auto merges one level deep. Let's say we have an incoming and initial state like the following:

```
{user: {name: 'Tammibriggs'}, isLoggedIn: true} // incoming state
{ user: {name: '', email: ''}, isLoggedIn: false, status: 'Pending'} // initial
state
```

The merged state will look like the following code:

```
{ user: {name: 'Tammibriggs'}, isLoggedIn: true, status: 'Pending'} //
reconciled/merged state
```

The initial state was merged with the incoming state and the top-level property values. In the incoming state, these are replaced and not merged, which is why the `email` property in `user` was lost. In our code, this will look similar to the following:

```
const mergedState = { ...initialState };


mergedState['user'] = persistedState['user']
mergedState['isLoggedIn'] = persistedState['isLoggedIn']
```

This type of merging in Redux Persist is called `autoMergeLevel1`, and it is the default state reconciler in Redux Persist. Other state reconcilers include `hardSet`, which completely overrides the initial state with the incoming state, and `autoMergeLevel2`, which merges two levels deep.

In our previous example, the `email` property in `user` won't be lost. The reconciled or merged state will look like the following code:

```
{ user: {name: 'Tammibriggs' email:''}, isLoggedIn: true, status: 'Pending'} //
reconciled/merged state
```

For example, to set up a state reconciler, if we want to use `autoMergeLevel2`, we just need to specify a `stateReconciler` property in `persistConfig`:

```
import autoMergeLevel2 from 'redux-persist/lib/stateReconciler/autoMergeLevel2';

const persistConfig = {
  key: 'root',
  storage,
  stateReconciler: autoMergeLevel2
}
```

Customize what’s persisted

We can customize a part of our state to persist by using the `blacklist` and `whitelist` properties of the `config` object passed to `persistReducer`. With the `blacklist` property, we can specify which part of state not to persist, while the `whitelist` property does the opposite, specifying which part of the state to persist.

For example, let’s say we have the following reducers:

```
const rootReducer = combineReducers({
  user: userReducer,
  notes: notesReducer
})
```

If we want to prevent `notes` from persisting, the `config` object should look like the following:

```
const rootPersistConfig = {
  key: 'root',
  storage,
  blacklist: ['notes']
}

// OR

const rootPersistConfig = {
  key: 'root',
  storage,
  whitelist: ['users']
}
```

The `blacklist` and `whitelist` properties take an array of strings. Each string must match a part of the state that is managed by the reducer we pass to `persistReducer` . When using `blacklist` and `whitelist` , we can only target one level deep. But, if we want to target a property in one of our states above, we can take advantage of nested persist.

For example, let’s say the `userReducer` initial state looks like the following:

```
const initialState = {
  user: {},
  isLoggedIn: false,
}
```

If we want to prevent `isLoggedIn` from persisting, our code will look like the following:

```
const rootPersistConfig = {
  key: 'root',
  storage,
}

const userPersistConfig = {
  key: 'user',
  storage,
  blacklist: ['isLoggedIn']
}

const rootReducer = combineReducers({
  user: persistReducer(userPersistConfig, userReducer),
  notes: notesReducer
})

const persistedReducer = persistReducer(rootPersistConfig, rootReducer);
```

Now, the `isLoggedIn` property won't be persisted.

Conclusion

In this tutorial, we've learned how to use Redux Persist in Redux Toolkit to save our data in persistent storage. Therefore, our data will still remain even after a browser refresh. We also explored several options for customizing Redux Persist, for example, specifying which storage engine to use, and customizing what is persisted in our state using the `blacklist` and `whitelist` properties.

Although at the time of writing, Redux Persist is under maintenance and has not been updated for some time, it is still a great tool with strong community support. I hope you enjoyed this tutorial, and be sure to leave a comment if you have any questions.

Over 200k developers use LogRocket to create better digital experiences

Learn more →

Cut through the noise of traditional React error reporting with LogRocket

LogRocket is a React analytics solution that shields you from the hundreds of false-positive errors alerts to just a few truly important items. LogRocket tells you the most impactful bugs and UX issues actually impacting users in your React applications. LogRocket automatically aggregates client side errors, React error boundaries, Redux state, slow component load times, JS exceptions, frontend performance metrics, and user interactions. Then LogRocket uses machine learning to notify you of the most impactful problems affecting the most users and provides the context you need to fix it.

Focus on the React bugs that matter — try LogRocket today.

Taminoturoko Briggs Follow

Software developer and technical writer. Core languages include JavaScript and Python.

#react #redux

4 Replies to "Persist state with Redux Persist using Redux Toolkit in..."

Peter Says:
August 28, 2022 at 7:47 am

Reply

I'm amazed that a developer suggests to use a package that hasn't been updated for 3 years. Redux, React has moved on – many people that tried this package failed for that reason. But even if that wouldn't already be the case, do you really want to recommend to developers a strategy that is doomed sooner or later?

Ayomide72 Says:
January 14, 2023 at 7:08 am

Reply

Very nice tutorial, everthing was on point

Alkshay Dixit Says:
February 27, 2023 at 1:25 am

Reply

PersistGate is not working in my project

Yoga Says:

April 1, 2023 at 10:50 am

Reply

it works, works!

Leave a Reply

Enter your comment here...