Redux Thunk example

This example shows how to integrate Redux and Redux Thunk in Next.js.

Usually splitting your app state into `pages` feels natural but sometimes you'll want to have global state for your app. This is an example on how you can use redux that also works with Next.js's universal rendering approach.

Deploy your own

Deploy the example using [Vercel] or preview live with [StackBlitz]

▲ Deploy

How to use

Execute [`create-next-app`] with [npm] or [Yarn] to bootstrap the example:

```
npx create-next-app --example with-redux-thunk with-redux-thunk-app
# or
yarn create next-app --example with-redux-thunk with-redux-thunk-app
# or
pnpm create next-app -- --example with-redux-thunk with-redux-thunk-app
```

Deploy it to the cloud with [Vercel] ([Documentation]).

Notes

The Redux `Provider` is implemented in `pages/_app.js`. The `MyApp` component is wrapped in a `withReduxStore` function, the redux `store` will be initialized in the function and then passed down to `MyApp` as `this.props.initialReduxState`, which will then be utilized by the `Provider` component.

Every initial server-side request will utilize a new `store`. However, every `Router` or `Link` action will persist the same `store` as a user navigates through the `pages`. To demonstrate this example, we can navigate back and forth to `/show-redux-state` using the provided `Link`s. However, if we navigate directly to `/show-redux-state` (or refresh the page), this will cause a server-side render, which will then utilize a new store.

In the `clock` component, we are going to display a digital clock that updates every second. The first render is happening on the server and then the browser will take over. To illustrate this, the server rendered clock will initially have a black background; then, once the component has been mounted in the browser, it changes from black to a grey background.

In the `counter` component, we are going to display a user-interactive counter that can be increased or decreased when the provided buttons are pressed.

This example includes two different ways to access the `store` or to `dispatch` actions:

1.) `pages/index.js` will utilize `connect` from `react-redux` to `dispatch` the `startClock` redux action once the component has been mounted in the browser.

2.) `components/counter.js` and `components/examples.js` have access to the redux store using `useSelector` and can dispatch actions using `useDispatch` from `react-redux@^7.1.0`

You can either use the `connect` function to access redux state and/or dispatch actions or use the hook variations: `useSelector` and `useDispatch`. It's up to you.

This example also includes hot-reloading when one of the `reducers` has changed. However, there is one caveat with this implementation: If you're using the `Redux DevTools` browser extension, then all previously recorded actions will be recreated when a reducer has changed (in other words, if you increment the counter by 1 using the `+1` button, and then change the increment action to add 10 in the reducer, Redux DevTools will playback all actions and adjust the counter state by 10 to reflect the reducer change). Therefore, to avoid this issue, the store has been set up to reset back initial state upon a reducer change. If you wish to persist redux state regardless (or you don't have the extension installed), then in `store.js` change (line 19) `store.replaceReducer(createNextReducer(initialState))` to `store.replaceReducer(createNextReducer)`.