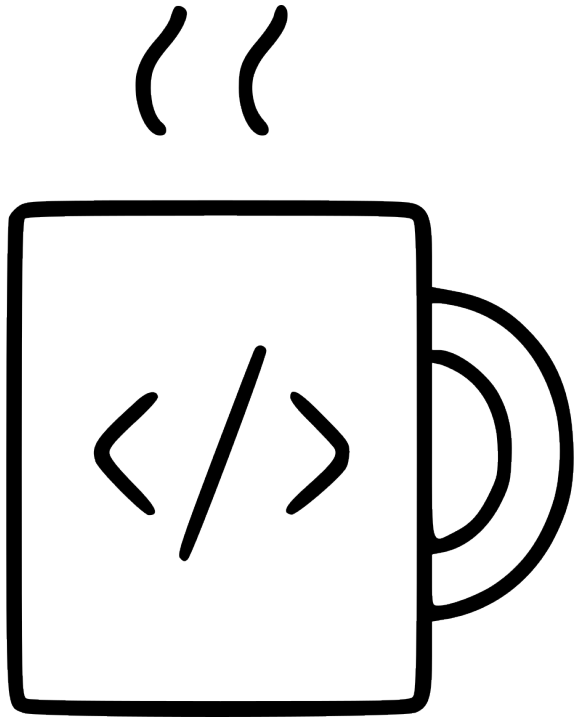
Umberto Emanuele

React - Redux

Gennaio 2023

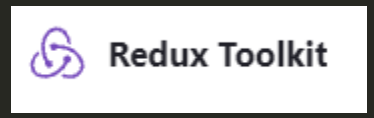


PAUSA

Ci vediamo alle ore 11.30


```
# Redux + Plain JS template
npx create-react-app my-app --template redux

# Redux + TypeScript template
npx create-react-app my-app --template redux-typescript
```



https://redux-toolkit.js.org/introduction/getting-started

Perché vogliamo che tu usi Redux #Toolkit

Come manutentori di Redux, la nostra opinione è:

> **TIP**
>
> **Vogliamo che *tutti* gli utenti Redux scrivano il loro codice Redux con Redux Toolkit, perché semplifica il tuo codice *ed* elimina molti errori e bug comuni di Redux!**

Redux Toolkit

Redux Toolkit

Utilizzo #dell'app

Il modo consigliato per avviare nuove app con React e Redux è utilizzare il modello Redux+JS ufficiale o il modello Redux+TS per Create React App , che sfrutta **Redux Toolkit** e l'integrazione di React Redux con i componenti React.

```
# Redux + Plain JS template
npx create-react-app my-app --template redux

# Redux + TypeScript template
npx create-react-app my-app --template redux-typescript
```

#Un'app esistente

Redux Toolkit è disponibile come pacchetto su NPM per l'utilizzo con un bundler di moduli o in un'applicazione Node:

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

Redux Toolkit

Crea un file chiamato **src/app/store.js**.

Importa l' **configureStoreAPI** da **Redux Toolkit**.

app/store.js

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

export const store = configureStore({
  reducer: {},
})
```

Redux Toolkit

Redux Toolkit

Una volta creato lo store, possiamo renderlo disponibile ai nostri componenti React inserendo un **React-Redux <Provider>** attorno alla nostra applicazione.

Importa il store Redux creato, metti un <Provider>intorno al tuo <App>e passa lo store come prop.

index.js

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

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

Redux Toolkit

Redux Toolkit

Aggiungi un nuovo file chiamato **slice.js**, importa l' **createSliceAPI** da Redux Toolkit.

La creazione di uno slice richiede:

- ✓ un nome per identificare lo slice,
- ✓ un valore per lo stato iniziale
- ✓ una o più funzioni reducers per definire
- ✓ come lo stato deve essere aggiornato.

Una volta creato lo slice, possiamo esportare le actions di Redux e il reducers completo.

features/counter/counterSlice.js

```
import { createSlice } from '@reduxjs/toolkit'

const initialState = {
  value: 0,
}

export const counterSlice = createSlice({
  name: 'counter',
  initialState,
  reducers: {
    increment: (state) => {
      // Redux Toolkit allows us to write "mutating" logic in reducers. It
      // doesn't actually mutate the state because it uses the Immer library,
      // which detects changes to a "draft state" and produces a brand new
      // immutable state based off those changes
      state.value += 1
    },
    decrement: (state) => {
      state.value -= 1
    },
    incrementByAmount: (state, action) => {
      state.value += action.payload
    },
  },
})

// Action creators are generated for each case reducer function
export const { increment, decrement, incrementByAmount } = counterSlice.actions

export default counterSlice.reducer
```

Redux Toolkit

Redux Toolkit

Successivamente, dobbiamo importare il reducer e aggiungerlo al nostro store, definendo un campo all'interno del parametro reducer.

Diciamo così allo store di utilizzare questo reducer per gestire tutti gli aggiornamenti dello stato.

app/store.js

```
import { configureStore } from '@reduxjs/toolkit'
import counterReducer from '../features/counter/counterSlice'

export const store = configureStore({
  reducer: {
    counter: counterReducer,
  },
})
```

Redux Toolkit

Redux Toolkit

Ora possiamo utilizzare gli **hook React-Redux** per consentire ai componenti React di interagire con lo **store Redux**.

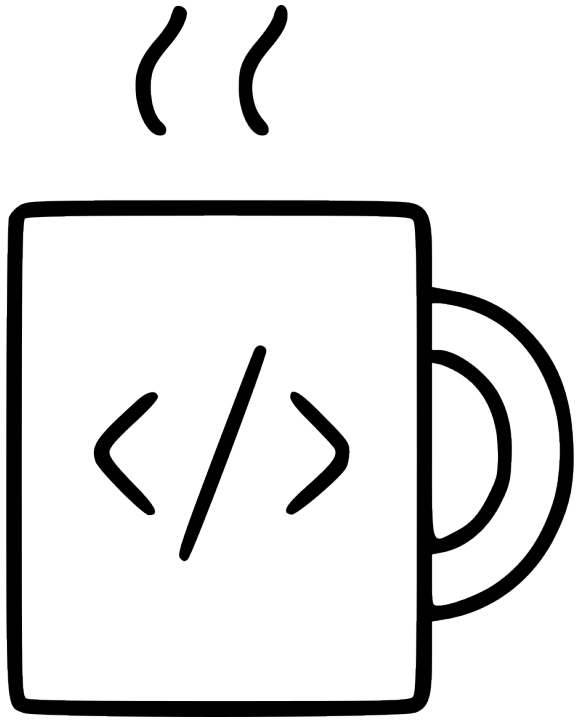
Possiamo leggere i dati dello store con **useSelector** e inviare azioni utilizzando **useDispatch**.

features/counter/Counter.js

```
import React from 'react'
import { useSelector, useDispatch } from 'react-redux'
import { decrement, increment } from './counterSlice'

export function Counter() {
  const count = useSelector((state) => state.counter.value)
  const dispatch = useDispatch()

  return (
    <div>
      <div>
        <button
          aria-label="Increment value"
          onClick={() => dispatch(increment())}
        >
          Increment
        </button>
        <span>{count}</span>
        <button
          aria-label="Decrement value"
          onClick={() => dispatch(decrement())}
        >
          Decrement
        </button>
      </div>
    </div>
  )
}
```



PAUSA

Ci vediamo alle ore 14.00

Azioni asincrone in Redux con Redux Toolkit

Quando si tratta di chiamate API, è necessario gestire tre diverse actions in Redux:

FETCH_REQUEST: quando inizia la richiesta;
FETCH_FAILURE: se la richiesta fallisce;
FETCH_SUCCESS: se la richiesta ha esito positivo;

Redux Toolkit include **redux-thunk**, quindi non è necessario installarlo.

Per gestire le azioni asincrone Redux Toolkit fornisce un metodo chiamato **createAsyncThunk** .

createAsyncThunk crea in automatico le tre actions, accetta un identificatore e una callback che esegue la logica asincrona effettiva e restituisce una promise che gestirà l'invio delle actions pertinenti in base al suo stato.

Redux Toolkit

Redux Toolkit

createAsyncThunk crea in automatico un action creator per ogni stato della promise.

Nel nostro caso, genera le tre actions con i seguenti nomi:

in attesa: **pending**;
rifiutato: **rejected**;
soddisfatto: **fulfilled**.

```
import { createAsyncThunk, createSlice } from '@reduxjs/toolkit';

const initialState = {
    loading: false,
    error: "",
    userlist: []
}

export const getUsers = createAsyncThunk("users/fetchList", () => {
    return fetch('https://jsonplaceholder.typicode.com/users')
      .then(response => {
        if (!response.ok) throw Error(response.statusText);
        return response.json();
      })
      .then(json => json);
  });
```

Redux Toolkit

Redux Toolkit

A differenza dei flussi di dati tradizionali, le actions create da createAsyncThunk verranno gestite dalla sezione **extraReducers** all'interno di una slice.

Utilizzeremo **action.error.message** per ottenere il messaggio di errore della promise rifiutata;
mentre per ottenere il payload del metodo API bisogna accedere a **action.payload** .

```
export const users_slice = createSlice(
    {
        name: 'users',
        initialState: initialState,
        reducers: {},
        extraReducers: {
            [getUsers.pending]: state => {
              state.loading = true;
            },
            [getUsers.rejected]: (state, action) => {
              state.loading = false;
              state.error = action.error.message;
            },
            [getUsers.fulfilled]: (state, action) => {
              state.loading = false;
              state.userlist = action.payload;
            }
          }
    }
);

console.log(users_slice);
const { actions, reducer } = users_slice; // destrutturo actions e reducers
//export {  } = actions; // destrutturo ed esporto actions
export default reducer; // esporto il reducers
```

In fine utilizziamo il nostro componente come siamo abituati a fare.

```
export default function Users() {

    const users = useSelector(state => state.users.userlist);
    const dispatch = useDispatch();

    useEffect(() => dispatch(getUsers()), []);


  return (
    <div>
      <h1>Azioni asincrone in Redux con Redux Toolkit </h1>
      <ul id="list-todos">
            {users.map((user,i) => (
                <li key={i}><strong>{user.name}</strong></li>
            ))}
        </ul>
    </div>
  )
}
```

Redux Toolkit

Per proteggere le rotte nei nostri progetti possiamo verificare se è presente un utente loggato nel nostro store altrimenti reindirizziamo alla pagina iniziale o nella pagina di login tramite l'hook **useNavigate().**

```
const dispatch = useDispatch()
const navigate = useNavigate()

const isUserLoggedin = !!useSelector((state) => state.user.name)
// se name è '', isUserLoggedin diventa false
// se name non è '', isUserLoggedin diventa true

console.log('isUserLoggedin', isUserLoggedin)

useEffect(() => {
  if (!isUserLoggedin) {
    navigate('/')
  }
}, [])
```

Redux Persist

Persist and rehydrate a redux store.

Quickstart

```
npm install redux-persist
```

https://github.com/rt2zz/redux-persist

Redux Persist


```
// configureStore.js

import { createStore } from 'redux'
import { persistStore, persistReducer } from 'redux-persist'
import storage from 'redux-persist/lib/storage' // defaults to localStorage for web

import rootReducer from './reducers'

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

const persistedReducer = persistReducer(persistConfig, rootReducer)

export default () => {
  let store = createStore(persistedReducer)
  let persistor = persistStore(store)
  return { store, persistor }
}
```

```
import { PersistGate } from 'redux-persist/integration/react'

// ... normal setup, create store and persistor, import components etc.

const App = () => {
  return (
    <Provider store={store}>
      <PersistGate loading={null} persistor={persistor}>
        <RootComponent />
      </PersistGate>
    </Provider>
  );
};
```

Redux Persist

Blacklist & Whitelist

By Example:

```
// BLACKLIST
const persistConfig = {
  key: 'root',
  storage: storage,
  blacklist: ['navigation'] // navigation will not be persisted
};

// WHITELIST
const persistConfig = {
  key: 'root',
  storage: storage,
  whitelist: ['navigation'] // only navigation will be persisted
};
```

Redux Persist transform-encrypt

Encrypt your Redux store.

Quickstart

```
npm install redux-persist-transform-encrypt
```

https://github.com/maxdeviant/redux-persist-transform-encrypt

Redux Persist transform-encrypt

Redux

```
import { persistReducer } from 'redux-persist';
import { encryptTransform } from 'redux-persist-transform-encrypt';

const reducer = persistReducer(
  {
    transforms: [
      encryptTransform({
        secretKey: 'my-super-secret-key',
        onError: function (error) {
          // Handle the error.
        },
      }),
    ],
  },
  baseReducer
);
```

```
const persistConfig = {
  key: 'root',
  storage: storage,
  transforms: [
    encryptTransform({
      secretKey: 'my-super-secret-key',
      onError: function (error) {
        // Handle the error.
      },
    }),
  ],
},
```

Redux Persist transform-encrypt

Redux

Aggiunta di variabili dell'ambiente di sviluppo `.env`

> Nota: questa funzione è disponibile con `react-scripts@0.5.0` e versioni successive.

Per definire variabili d'ambiente permanenti, crea un file chiamato `.env` nella root del tuo progetto:

```
REACT_APP_NOT_SECRET_CODE=abcdef
```

> Nota: è necessario creare variabili di ambiente personalizzate che inizino con `REACT_APP_`.

```
<form>
  <input type="hidden" defaultValue={process.env.REACT_APP_NOT_SECRET_CODE} />
</form>
```

Durante la compilazione, `process.env.REACT_APP_NOT_SECRET_CODE` verrà sostituito con il valore corrente

https://create-react-app.dev/docs/adding-custom-environment-variables/

challenge
NETWORK

shaping the skills of tomorrow

challengenetwork.it