وب سایت :

https://redux-toolkit.js.org/

Redux Toolkit (RTK)

Redux Toolkit یا RTK یک لایبرری رسمی از تیم Redux است که هدفش **سادگی و استانداردسازی کار با Redux** است. به‌طور خلاصه، RTK مشکلات و boilerplate های سنتی Redux را حل کرده و توسعه‌دهنده را قادر می‌سازد سریع‌تر، امن‌تر و مقیاس‌پذیرتر state مدیریت کند.

چرا Redux Toolkit استفاده می‌کنیم ؟

دلایل اصلی استفاده از:RTK

1. **کاهش boilerplate**
   - در Redux سنتی شما باید actions, action types, reducers وstoreرا جداگانه تعریف کنید.

- RTK با createSliceو configureStore این روند را ساده می‌کند.

2. **State امن و بدون mutation**

- RTK از Immer داخلی استفاده می‌کند تا بتوانید state را به‌صورت mutable بنویسید، ولی واقعاً immutable شود.

3. **سهولت در مدیریت Async actions**

- با createAsyncThunk می‌توان API call ها را ساده و قابل مدیریت کرد، همراه با lifecycle states: pending, fulfilled, rejected.

4. **بهترین روش‌های استاندارد**

- RTK راهنما و استانداردهای مشخصی دارد که از بروز خطاهای رایج جلوگیری می‌کند.

تفاوت RTK با Context و دیگر state manager ها

ویژگی	Redux Toolkit	React Context	Zustand / Jotai / Recoil
مقیاس‌پذیری	بسیار خوب برای پروژه‌های بزرگ	مناسب برای stateهای کوچک و محلی	خوب برای متوسط
DevTools	Redux DevTools رسمی	ندارد یا محدود	برخی دارند
Middleware & Async	کاملاً پشتیبانی شده	باید دستی مدیریت شود	محدود
Immutable state	بله، با Immer	خیر، باید مراقب باشی	بسته به پیاده‌سازی
Boilerplate	کم‌تر نسبت به Redux سنتی	بسیار کم	کم

نکته Context :

خوب است برای state محلی و کوچک مثلاً theme یا auth state ساده ، اما برای state بزرگ، RTK مناسب‌تر و امن‌تر است.

برای چه پروژه‌هایی مناسب است؟

- پروژه‌های **متوسط تا بزرگ** با state global پیچیده
- پروژه‌هایی که نیاز به **DevTools و debugging** دارند
- پروژه‌هایی که API-heavy هستند و نیاز به **Async logic مدیریت شده** دارند
- تیم‌های بزرگ که نیاز به **استاندارد سازی کد** دارند

❌پروژه‌های خیلی کوچک یا state ساده ممکن است RTK اضافه بار باشد.

توضیح store :

Store یعنی **مغز مرکزی مدیریت داده‌ها در اپلیکیشن**.

همه‌یstate های مهم مثل todos, user, cart و ... داخل Store نگهداری می‌شن.

در واقع، store یه **آبجکت جاوااسکریپتی** هست که:

- کل state فعلی برنامه رو نگه می‌داره
- به reducer ها گوش می‌ده
- و وقتی state عوض می‌شه، componentهایی که به اون وصل شدن از طریق useSelector رو **به‌روزرسانی می‌کنه**.

اطلاعات کجا ذخیره میشه ؟

Redux Store فقط داخل **حافظه‌ی (RAM) مرورگر در زمان اجرا** نگهداری میشه.
یعنی:

- داخل هیچ فایل یا فولدری ذخیره نمی‌شه
- داخل دیتابیس یا localStorage هم به صورت پیش‌فرض نمی‌ره
- فقط وقتی صفحه بازه، توی حافظه (memory) نگهداری میشه

پس وقتی صفحه رفرش میشه چی میشه ؟

وقتی صفحه **رفرش بشه یا بسته بشه**:

- Store از بین می‌ره
- چون فقط در حافظه‌ی موقتی مرورگر نگهداری میشه

ولی اگر بخوای state رو **ذخیره‌ی دائم** کنی، باید از کتابخونه‌هایی مثل:

- redux-persist استفاده کنی

  که Store رو داخل **localStorage** ذخیره می‌کنه

  و وقتی کاربر برمی‌گرده، دوباره اون state رو بارگذاری می‌کنه.

خلاصه store

Store مثل مغز برنامه‌ست که همه‌ی state ها داخلش نگهداری می‌شن.

خودش فقط یه آبجکت در حافظه است (نه فایل واقعی).

با رفرش صفحه از بین می‌ره.

برای ذخیره‌ی دائمی باید از کتابخونه‌هایی مثل redux-persist استفاده کنیم.

کل برنامه از طریق Provider به Store وصل میشه.

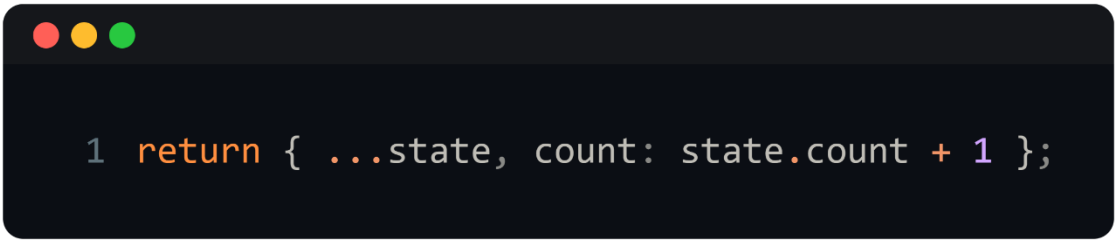
توضیح immer / immutable

در Redux معمولی، ما نمی‌تونستیم state رو مستقیماً تغییر بدیم،

چون Redux باید بتونه تشخیص بده چه زمانی state عوض شده.

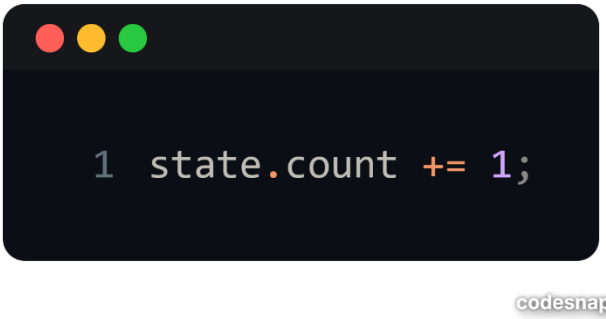
برای همین باید همیشه یه **کپی جدید از state** می‌ساختیم

با استفاده از {state...} یا map, filter و..



در **Redux Toolkit**، یه کتابخونه داخلی به نام **Immer** وجود داره که این کار رو **به صورت خودکار پشت صحنه انجام می‌ده**.

بنابراین ما می‌تونیم خیلی راحت‌تر بنویسیم:



در ظاهر به نظر می‌رسه داریم مستقیماً state رو تغییر می‌دیم، ولی در واقعیت، **Immer یه کپی جدید از state می‌سازه** و تغییرات رو روی اون اعمال می‌کنه.

در Redux معمولی باید خودمون کپی جدید از state می‌ساختیم،
ولی در Redux Toolkit به کمک کتابخونه‌ی **Immer**،
این کار به‌صورت خودکار انجام می‌شه.
پس ما می‌تونیم state رو مثل حالت عادی تغییر بدیم،
ولی در پشت صحنه هنوز immutable باقی می‌مونه».

نصب و راه اندازی :

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

ساخت Store در پروژه :

ساخت slice :

```
// counterSlice.js
import { createSlice } from '@reduxjs/toolkit';

const initialState = { value: 0 };

const counterSlice = createSlice({
  name: 'counter',
  initialState,
  reducers: {
    increment: (state) => { state.value += 1 },
    decrement: (state) => { state.value -= 1 },
    incrementByAmount: (state, action) => { state.value += action.payload }
  }
});

export const { increment, decrement, incrementByAmount } = counterSlice.actions;
export default counterSlice.reducer;

```

نحوه استفاده :

```jsx
// Counter.js
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { increment, decrement, incrementByAmount } from './counterSlice';

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

  return (
    <div>
      <h1>{count}</h1>
      <button onClick={() => dispatch(increment())}>+</button>
      <button onClick={() => dispatch(decrement())}>-</button>
      <button onClick={() => dispatch(incrementByAmount(5))}>+5</button>
    </div>
  );
}

```

ایجاد کردن provider :

```tsx
"use client";

import { Provider } from "react-redux";

import { ReactNode } from "react";
import { store } from "../store/store";

interface ReduxToolkitProviderProps {
  children: ReactNode;
}

export const ReduxToolkitProvider = ({
  children,
}: ReduxToolkitProviderProps) => {
  return <Provider store={store}>{children}</Provider>;
};

```

استفاده از provider در layout.tsx :

```tsx
import type { Metadata } from "next";
import { Geist, Geist_Mono } from "next/font/google";
import "./globals.css";
import { ReduxToolkitProvider } from "./providers/reduxToolkitProvider";

const geistSans = Geist({
  variable: "--font-geist-sans",
  subsets: ["latin"],
});

const geistMono = Geist_Mono({
  variable: "--font-geist-mono",
  subsets: ["latin"],
});

export const metadata: Metadata = {
  title: "Create Next App",
  description: "Generated by create next app",
};

export default function RootLayout({
  children,
}: Readonly<{
  children: React.ReactNode;
}>) {
  return (
    <html lang="en">
      <body
        className={`${geistSans.variable} ${geistMono.variable} antialiased`}
      >
        <ReduxToolkitProvider>{children}</ReduxToolkitProvider>
      </body>
    </html>
  );
}

```

مدیریت Async Actions با createAsyncThunk (API و هر انچه که نیاز به Async داره)

```js
// userSlice.js
import { createSlice, createAsyncThunk } from '@reduxjs/toolkit';
import axios from 'axios';

export const fetchUser = createAsyncThunk('user/fetch', async (userId) => {
  const response = await axios.get(`/api/user/${userId}`);
  return response.data;
});

const userSlice = createSlice({
  name: 'user',
  initialState: { data: null, loading: false, error: null },
  reducers: {},
  extraReducers: (builder) => {
    builder
      .addCase(fetchUser.pending, (state) => { state.loading = true })
      .addCase(fetchUser.fulfilled, (state, action) => {
        state.loading = false;
        state.data = action.payload;
      })
      .addCase(fetchUser.rejected, (state, action) => {
        state.loading = false;
        state.error = action.error.message;
      });
  }
});

export default userSlice.reducer;

```

extraReducers

درRedux Toolkit ، createSlice دو نوع reducer دارد:

1. **reducers معمولی** (reducers)
   - برای action های داخلی slice استفاده می‌شود.
   - خود RTK برای شما action creator می‌سازد.
   - مثال increment, decrement :
2. **extraReducers**
   - برای مدیریت **action هایی که در خارج از slice ساخته می شوند**.
   - معمولاً برای **AsyncThunk ها** یاaction های دیگر slice استفاده می‌شود.
   - در واقع اینجا می‌گویید: "وقتی این action اتفاق افتاد، state را چطور تغییر دهم."

builder چیست؟

- builder **یک شی object-oriented** است که Redux Toolkit برای extraReducers فراهم می‌کند.
- مزیت آن نسبت به :**object notation**

در صورت نبودنش به این شکل باید استفاده میشد که استفاده ازش کار ما رو راحت می کنه .

Builder **type-safe** است و TypeScript را بهتر پشتیبانی می‌کند.

مخصوصاً برای **asyncThunk ها** مناسب است، چون TS می‌تواند نوع action.payload را تشخیص دهد.

builder.addCase

- متد addCase روی builder برای **اضافه کردن handler به یک action خاص** استفاده می‌شود.
- سینتکس:

اگر action یک createAsyncThunk باشد، می‌توانید سه حالت آن را اضافه کنید:

1. **pending** → درخواست شروع شده
2. **fulfilled** → درخواست موفق
3. **rejected** → درخواست با خطا

نکات حرفه‌ای best practice

1. **استفاده از builderتوصیه می‌شود**
   - type-safe است و برای AsyncThunk ها کامل است.
2. **برای AsyncThunk ها همیشه ۳ حالت را مدیریت کنید**
   - pending لودینگ
   - fulfilled ذخیره داده

- ذخیره خطا rejected

3. **state mutation با Immer**
   - داخل addCase می‌توانید مستقیماً state.data = action.payload بنویسید و RTK خودش immutable handling را انجام می‌دهد.
4. **می‌توان چند addCase پشت سر هم نوشت**
   - به جای یک object با کلید action type ، builder chain خیلی خواناتر و مقیاس‌پذیرتر است.

مشکلات رایج و راه حل ها

مشکل	توضیح	راه حل
Cannot read property of undefined	معمولا `useselector` اشتباه است یا store درست وصل نشده	چک کنید که Provider درست است و keyهای reducer مطابقت دارند
Action is undefined	نام action اشتباه import شده	از `export const { ... } = slice.actions` درست استفاده کنید
AsyncThunk rejected	API خطا داده یا network error	بررسی کنید payload و error در rejected
Mutating state outside reducers	اغلب با state غیر Immer انجام شده	حتما تغییر state فقط در reducers slice انجام شود یا از Immer/`createReducer` استفاده کنید

خلاصه Flow RTK

Slice → Actions + Reducers

Store → ConfigureStore با slices

Component → useSelector + useDispatch

Async → createAsyncThunk + extraReducers

ساختار فولدر :

با RTK Query


src/
├ app/
│ └ store.js # configureStore
├ features/
│ ├ counter/
│ │ ├ counterSlice.js
│ │ ├ Counter.jsx
│ │ └ counterApi.js # اگه RTK Query مربوط به counter باشه
│ ├ user/
│ │ ├ userSlice.js
│ │ ├ User.jsx
│ │ └ userApi.js # RTK Query برای user
│ └ ...
├ components/
│ └ ...
├ hooks/
│ └ useAuth.js # هوک سفارشی
└ App.jsx
codesnap.dev

بدون RTK Query