Розробка мобільних додатків

Лекція 11 - State Management у React Native

Управління станом за спрощеною моделлю

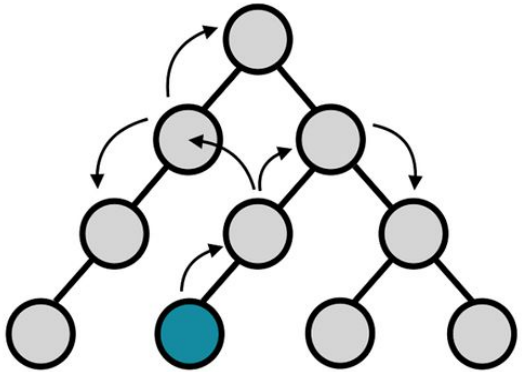
- Стан зберігається безпосередньо в компонентах.
- Для керування станом використовуються хуки **useState** та **useEffect**.
- Дані передаються між компонентами через властивості (**props drilling**).
- У разі необхідності спільного доступу до стану кількома компонентами — стан піднімається до найближчого спільного батьківського компонента (*lifting state up*).

Проблеми, які виникають у великих додатках

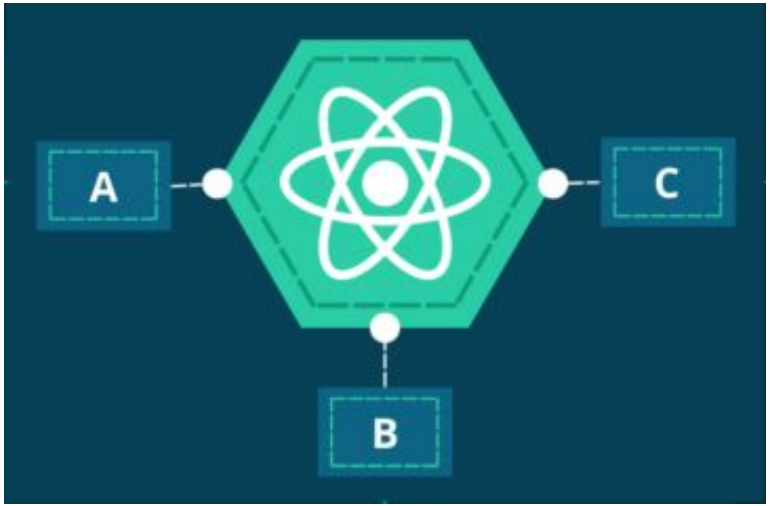
Плутанина з пропсами (props drilling hell):
Інформацію доводиться "перекидати" через кілька рівнів компонентів, навіть якщо вона потрібна лише внизу. Це зменшує гнучкість і змушує багато компонентів знати про щось, що їх взагалі не стосується.

Непередбачувані побічні ефекти:
Компоненти починають змінювати state напряму або через неконтрольовані шляхи. Це ускладнює відстеження: “Хто взагалі змінив цю змінну і чому?”.

Важко дебажити:
У великих додатках важливо знати: що викликало зміну стану, коли, і яким чином.



Як вирішити ці проблеми



Context API

Context API — це вбудований механізм React для передачі даних через дерево компонентів без необхідності явної передачі props на кожному рівні.

Призначення:

- Централізований доступ до спільних даних
- Усунення проблеми **props drilling**
- Спрощення архітектури компонентів

Принцип роботи:

- Контекст огортає частину дерева компонентів через Provider
- Всі вкладені компоненти можуть отримати доступ до значення без передачі props
- При зміні значення Provider — всі підписані компоненти перерендерюються

Приклад використання **Context API**


```jsx
import { createContext, useContext, useState } from 'react';

const ThemeContext = createContext();

Show usages  new *
export const ThemeProvider = ({ children }) => {
    const [theme, setTheme] = useState('light');

    return (
        <ThemeContext.Provider value={{ theme, setTheme }}>
            {children}
        </ThemeContext.Provider>
    );
};

Show usages  new *
export const useTheme = () => useContext(ThemeContext);
```

```tsx
import { Stack } from 'expo-router';
import { ThemeProvider } from '../context/ThemeContext';

no usages  new *
export default function Layout() {
  return (
      <ThemeProvider>
        <Stack />
      </ThemeProvider>
  );
}
```

```tsx
import { View, Text, Button } from 'react-native';
import { useTheme } from '../context/ThemeContext';

no usages
export default function Home() {
    const { theme, setTheme } = useTheme();

    Show usages
    const toggleTheme = () => {
        setTheme(prev => (prev === 'light' ? 'dark' : 'light'));
    };

    return (
        <View>
            <Text>{theme}</Text>
            <Button title="Toggle theme" onPress={toggleTheme} />
        </View>
    );
}
```

Недоліки **Context API**

Надлишкові ререндери

При зміні значення всі компоненти, що використовують контекст, перерендерюються

Обмежена масштабованість

При збільшенні обсягу стану структура ускладнюється та гірше підтримується

Складність контролю оновлень

Важко оновлювати лише окремі частини стану без впливу на інші компоненти

Обмежена придатність для складної логіки

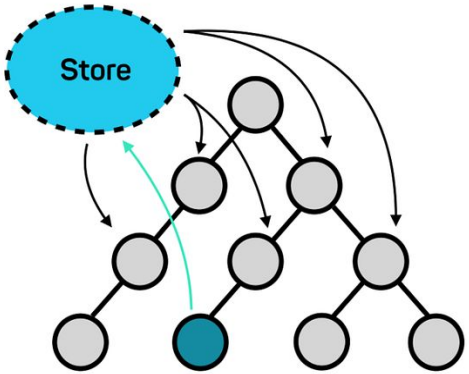
Ефективний для простих сценаріїв (тема, користувач), але не підходить для складного state management

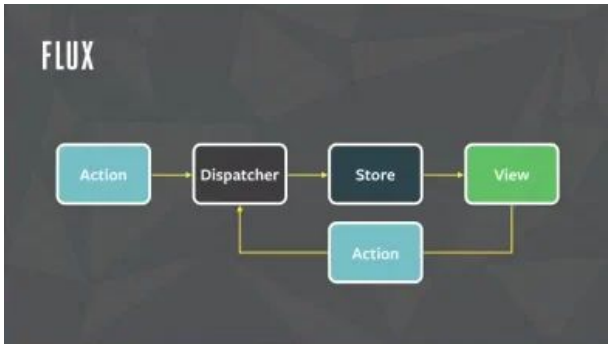
Flux-архітектура

Flux — це архітектурний патерн, запропонований Facebook для створення масштабованих і передбачуваних клієнтських додатків. Він був розроблений для вирішення проблем, подібних до "props drilling", шляхом організації потоку даних в одному напрямку. Основна ідея Flux полягає в тому, щоб зробити рух даних у додатку чітким і контрольованим.

Основні компоненти Flux

1. **Store (Сховище)**: Містить стан додатка та логіку його оновлення. Замість того, щоб передавати стан через props, усі компоненти можуть отримувати дані напряму зі Store.
2. **Actions (Дії)**: Події, які описують, що сталося в додатку (наприклад, "користувач натиснув кнопку"). Вони відправляються через Dispatcher.
3. **Dispatcher (Диспетчер)**: Центральний хаб, який розподіляє дії між усіма Store. Він забезпечує, що оновлення стану відбувається послідовно.
4. **Views (Компоненти)**: React-компоненти, які відображають стан із Store і викликають дії (Actions) у відповідь на взаємодію користувача.





Переваги Flux

- **Однонаправлений потік даних**: Дані рухаються в одному напрямку (Actions → Dispatcher → Store → Views), що робить додаток передбачуваним.
- **Централізований стан**: Усі дані зберігаються в Store, що усуває потребу в "props drilling".
- **Легше масштабування**: Додавання нових функцій не ускладнює структуру, адже нові компоненти просто підключаються до Store.

Проблеми Flux-архітектури

Складність реалізації: Flux вимагає створення кількох компонентів — Store, Dispatcher, Actions, а також їхньої взаємодії. Для невеликих додатків це може бути надмірно складно. Наприклад, потрібно вручну налаштовувати Dispatcher для обробки всіх дій, що додає багато шаблонного коду.

Множинні Store: У Flux часто використовується кілька Store для різних частин стану. Це може призводити до складнощів у синхронізації між ними, особливо якщо одна дія має впливати на кілька Store, потрібно вручну обробляти залежності між ними.

Відсутність чіткої структури для асинхронних операцій: Flux не пропонує вбудованого рішення для асинхронних дій, таких як запити до API. Це змушує розробників створювати власні патерни.

Низька передбачуваність у великих додатках: Через множинні Store і складність Dispatcher відстеження змін стану у великих проєктах стає проблематичним. Наприклад, якщо кілька Store реагують на одну дію, важко передбачити порядок оновлень.

Ці проблеми стали поштовхом до створення бібліотеки, яка спрощує принципи Flux, зберігаючи його переваги

Redux

Redux був створений у 2015 році як еволюція Flux, щоб вирішити його недоліки, спростити управління станом і зробити його більш передбачуваним. Redux взяв основну ідею Flux — однонаправлений потік даних — і переосмислив її з кількома ключовими покращеннями.

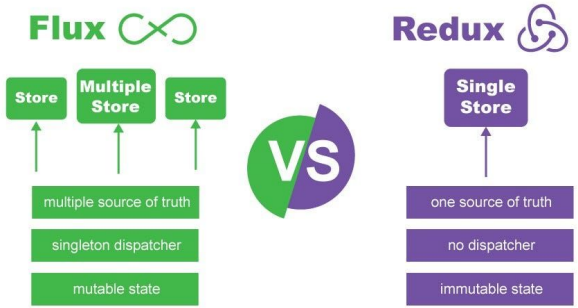
Єдине джерело правди (Single Source of Truth): На відміну від Flux, де може бути кілька Store, Redux використовує **один глобальний Store** для всього стану додатка.

Спрощена структура: Redux усуває Dispatcher, замінюючи його на **ред'юсери** — чисті функції, які описують, як стан змінюється у відповідь на дію. Це зменшує кількість шаблонного коду.

Покращена підтримка асинхронних операцій: Redux вводить поняття **middleware** , яке дозволяє легко обробляти асинхронні дії.

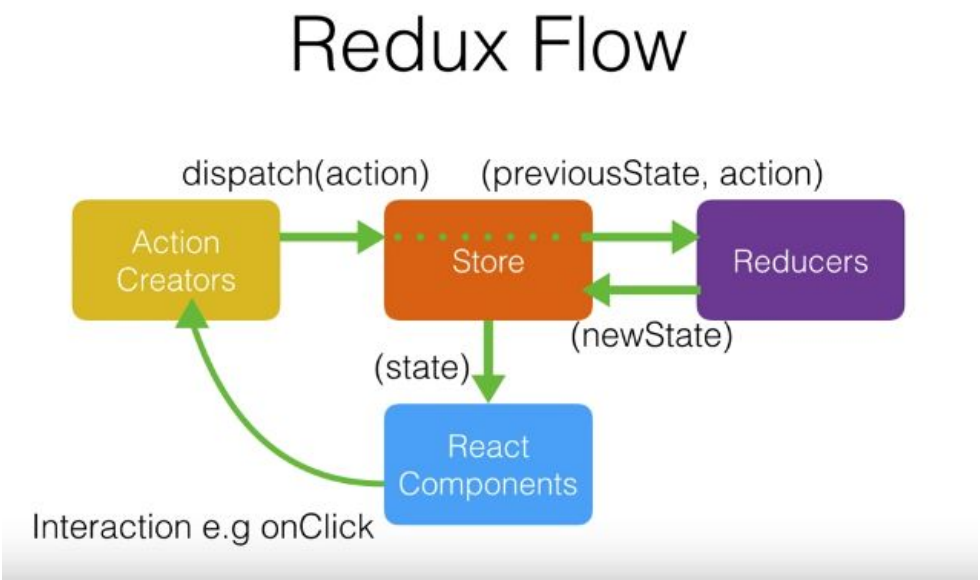
Передбачуваність через імутабельність: У Redux стан є незмінним (immutable). Ред'юсери завжди повертають новий стан, а не змінюють існуючий. Це полегшує відстеження змін і робить додаток більш передбачуваним, особливо у великих проєктах.

Інструменти розробника: Redux має інструменти, такі як Redux DevTools, які дозволяють відстежувати дії та зміни стану в реальному часі. Це вирішує проблему Flux із відстеженням змін у великих додатках.



Основна ідея **Redux**

Основна ідея Redux полягає в єдиному контейнері стану (store), до якого всі компоненти мають доступ, а зміни відбуваються лише через дії (actions), оброблені редюсерами (reducers). Це зробило стан централізованим, прогнозованим і легким для тестування.



Проблеми Redux

Шаблонний код:

Створення дій, типів дій та редюсерів потребує написання великої кількості шаблонного коду.

Робота з асинхронними діями:

- Redux за замовчуванням не підтримує асинхронні операції.
- Потрібно використовувати додаткові бібліотеки, такі як Redux Thunk або Redux Saga, що додає складності.

Складність масштабування:

- У великих проєктах управління великою кількістю редюсерів та дій стає проблематичним.
- Поєднання редюсерів та організація коду потребують чіткої структури.

Низька продуктивність за замовчуванням:

- Без оптимізації компоненти можуть перемальовуватися навіть при незначних змінах стану.

Redux Toolkit: сучасний підхід до **Redux**

Redux Toolkit(2019) — це офіційне розширення для Redux, створене для спрощення його використання.

Основні причини створення:

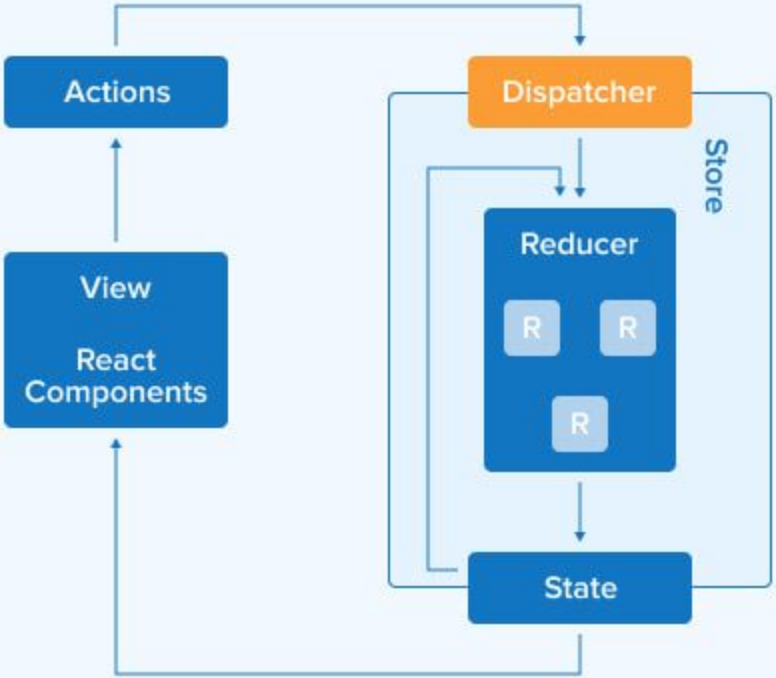
- Зменшення кількості коду
- Спрощення структури проєкту
- Запобігання типових помилок при роботі зі станом
- Прискорення процесу розробки

Ідея:

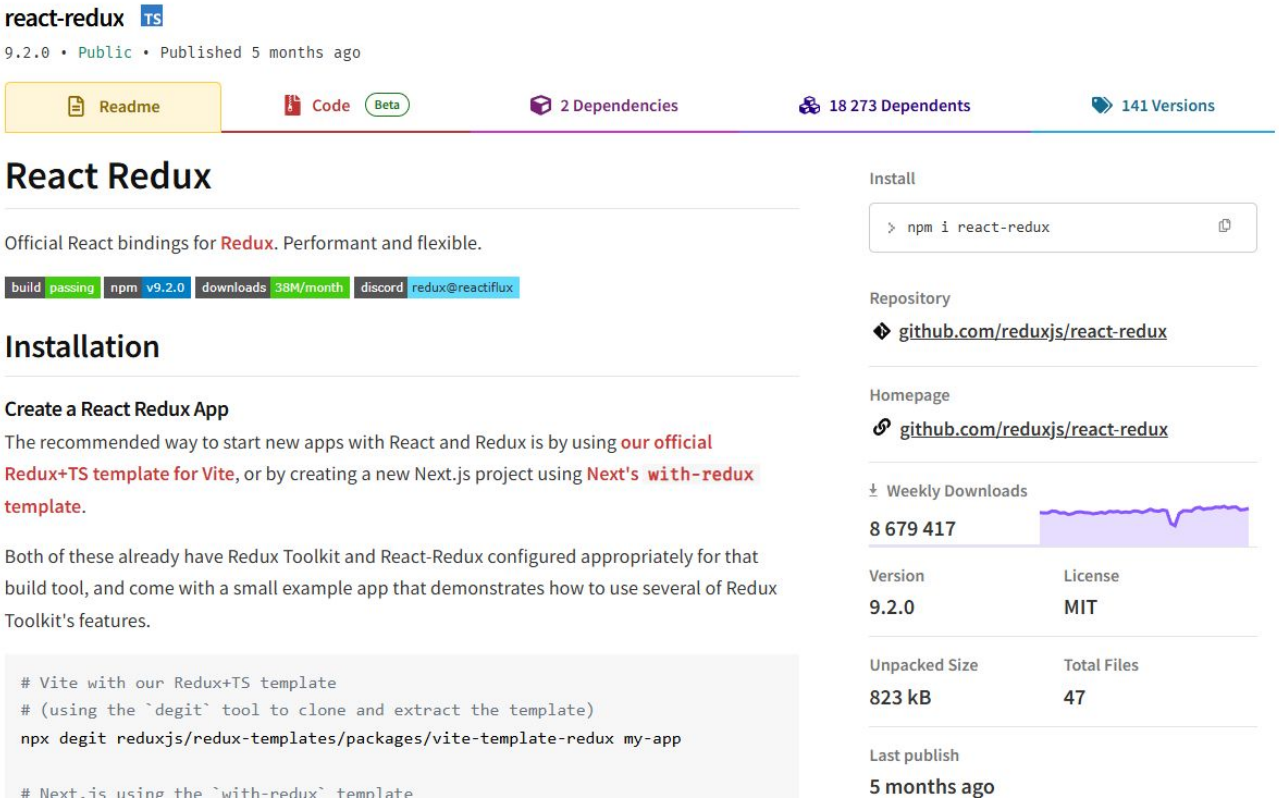
- Зробити Redux простішим та зручнішим у використанні
- Надати готові інструменти з кращими практиками

Стандарт індустрії: 90% нових проєктів на Redux використовують саме Toolkit.

How Does the Redux Toolkit Work?



Встановлення бібліотек



react-redux TS

9.2.0 • Public • Published 5 months ago

Readme | Code Beta | 2 Dependencies | 18 273 Dependents | 141 Versions

React Redux

Official React bindings for Redux. Performant and flexible.

build passing npm v9.2.0 downloads 38M/month discord redux@reactiflux

Installation

Create a React Redux App

The recommended way to start new apps with React and Redux is by using our official Redux+TS template for Vite, or by creating a new Next.js project using Next's `with-redux` template.

Both of these already have Redux Toolkit and React-Redux configured appropriately for that build tool, and come with a small example app that demonstrates how to use several of Redux Toolkit's features.

```
# Vite with our Redux+TS template
# (using the `degit` tool to clone and extract the template)
npx degit reduxjs/redux-templates/packages/vite-template-redux my-app

# Next.js using the `with-redux` template
```

Install

```
> npm i react-redux
```

Repository

github.com/reduxjs/react-redux

Homepage

github.com/reduxjs/react-redux

Weekly Downloads

8 679 417

Version 9.2.0

License MIT

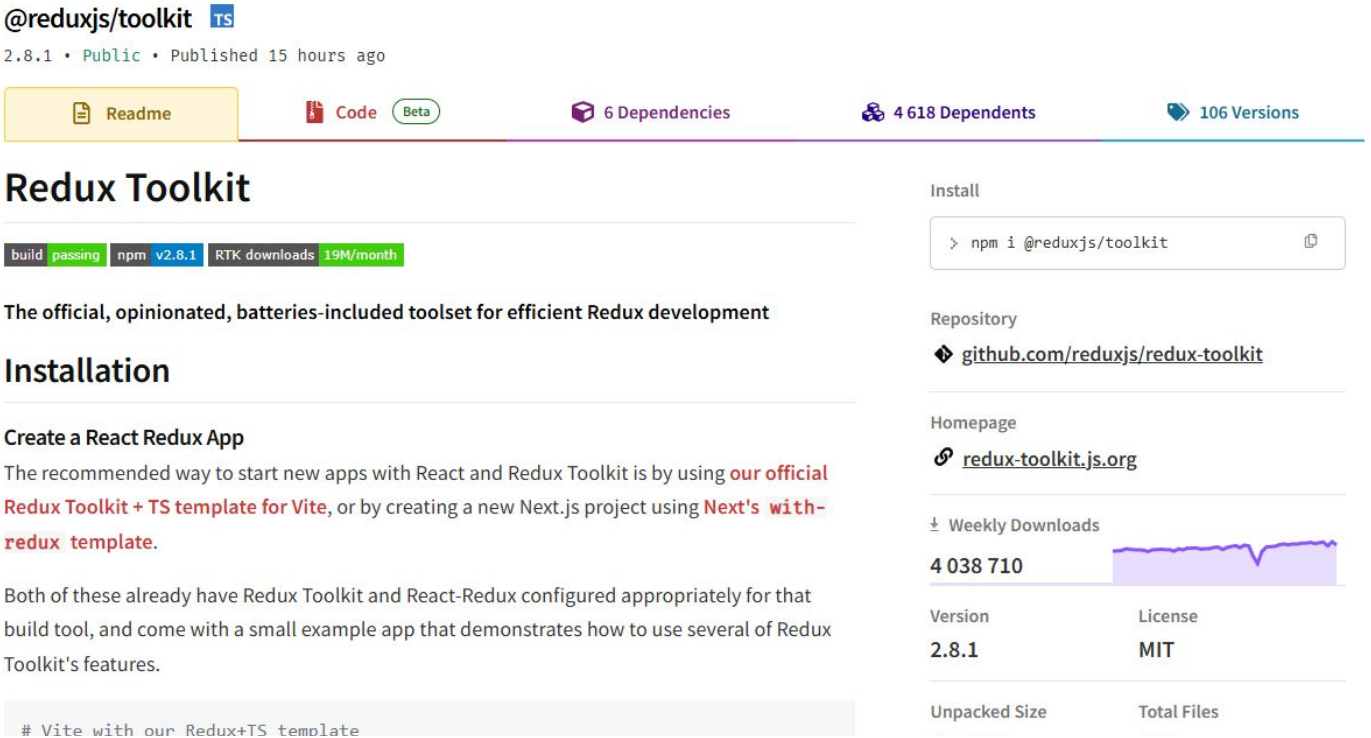
Unpacked Size 823 kB

Total Files 47

Last publish

5 months ago

Встановлення бібліотек



Встановлення бібліотек

- **`@reduxjs/toolkit`** — це офіційна рекомендована бібліотека для спрощеної роботи з Redux.
- **`react-redux`** — бібліотека, яка забезпечує інтеграцію Redux із React-компонентами. Вона дозволяє підключати компоненти до глобального стану Redux.

Встановлення бібліотек

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

Огляд основних **API Redux Toolkit**

Redux Store з configureStore

Store — це центральне місце, де зберігається стан додатку в Redux. Він дозволяє:

- **Зберігання стану**.
- **Доступ до стану**.
- **Оновлення стану**.
- **Підписка на зміни**.

configureStore

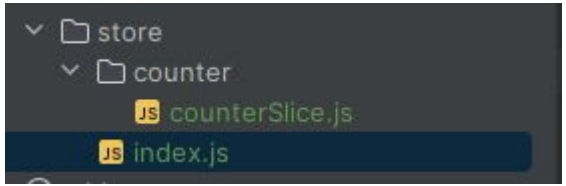
Redux Toolkit надає функцію `configureStore`, яка:

- спрощує створення store,
- вмикає підтримку Redux DevTools.
- захищає від помилок: автоматично перевіряє, чи ви випадково не змінили стан напряму.

configureStore | Redux Toolkit

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

export const index = configureStore( options: {
    reducer: {
        counter: counterReducer,
    },
});
```



Що тут відбувається:

- Ми викликаємо `configureStore` і передаємо в нього об’єкт налаштувань.
- Параметр `reducer` — це об’єкт, де кожен ключ відповідає імені гілки стану.

`counter: counterReducer` означає: у глобальному `state` з’явиться гілка `counter`, за яку відповідатиме функція `counterReducer`.

Slice

Slice — це частина (гілка) глобального стану Redux, що має:

- власне ім’я (наприклад, `'counter'`),
- початковий стан (initial state),
- набір ред’юсерів (функцій, які змінюють цей стан),
- автоматично згенеровані дії (actions) на основі ред’юсерів.

Раніше в Redux треба було писати окремо Actions, окремо Reducers та окремо константи. Slice об'єднує все це в одному місці.

createSlice

`createSlice` приймає один об’єкт конфігурації з такими полями:

`name` (рядок)

- Ім’я слайсу — використовується як **префікс** у згенерованих типах дій.
- Наприклад: якщо `name: 'counter'`, то дія `increment` матиме тип `'counter/increment'`.
- Це гарантує **унікальність дій** у межах додатку.

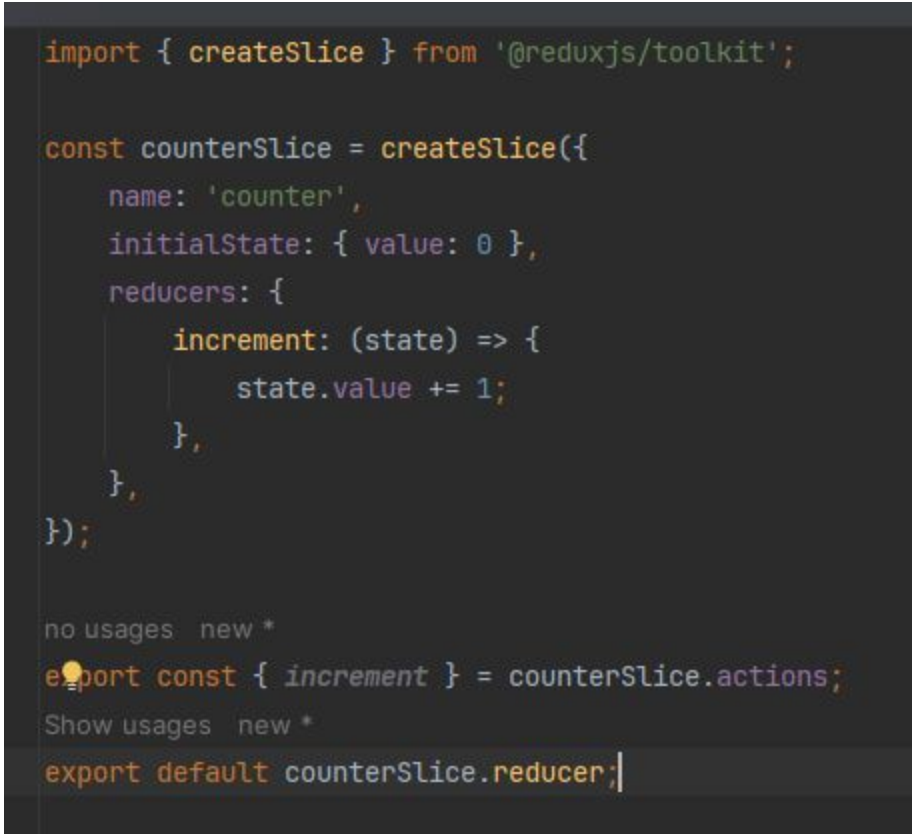
`initialState`

- Початковий стан цієї гілки.
- Може бути **будь-яким типом**: об’єкт, масив, число, булеве значення.
- Також може бути функцією, що повертає початкове значення — корисно для **відкладеної ініціалізації** (наприклад, читання з `AsyncStorage`).

`reducers` (об’єкт)

- Набір **"case-редюсерів"** — функцій, що змінюють `state` у відповідь на певну дію.
- Ключі — це **імена дій** (`increment`, `logout`, тощо).

Важливо: редюсери мають бути **чистими функціями** — не можна виконувати асинхронні операції (API, `setTimeout`) в `reducers`.


import { createSlice } from '@reduxjs/toolkit';
const counterSlice = createSlice({
name: 'counter',
initialState: { value: 0 },
reducers: {
increment: (state) => {
state.value += 1;
},
},
});
no usages new *
export const { increment } = counterSlice.actions;
Show usages new *
export default counterSlice.reducer;

`extraReducers`

`extraReducers` дозволяє обробляти **зовнішні дії** (тобто не створені у цьому slice), наприклад:

- асинхронні дії,
- або дії з інших slice'ів.

createSlice автоматично створює **actions:**

`createSlice` автоматично створює actions:

```
export const { increment } = counterSlice.actions;
```

Використання:

```
dispatch(increment()); // ➜ { type: 'counter/increment' }
```

Примітка: Action — це повідомлення Redux: "Сталася подія. Змініть стан відповідно."
Ключова ідея: назва type формується як sliceName/reducerName — тобто 'counter' + 'increment' = 'counter/increment'.

`selectors`

- Дозволяє визначати селектори прямо у slice.
- Працює в парі з `createSlice.selectors`, які можна одразу експортувати.

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

const counterSlice = createSlice({
    name: 'counter',
    initialState: { value: 0 },
    reducers: {
        increment: (state) => {
            state.value += 1;
        },
    },
    selectors: {
        selectCount: (state) => state.value,
    },
});

no usages  new *
export const { increment } = counterSlice.actions;
no usages  new *
export const { selectCount } = counterSlice.selectors;
Show usages  new *
export default counterSlice.reducer;
```

Замість того, щоб звертатися до `state.counter.value`, ми використовуємо:

```
const count = useSelector(selectCount);
```

Це означає:

- Якщо структура `state` зміниться — треба змінити лише селектор, а не всі компоненти.
- **Менше залежності від структури Redux-стану**.

Роблять компоненти чистішими

```
// ❌ Погано:

const token = useSelector((state) => state.auth.user?.token);

// ✅ Добре:

const token = useSelector(selectToken);
```

Компоненти просто викликають селектор. Це покращує **читабельність** і **тестованість**.

Приклад слайсу:

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

const initialState = {
    value: 0,
};

const counterSlice = createSlice( options: {
    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;
```

Як підключити **Redux** до **React/React Native-**додатку

Щоб компоненти могли отримувати доступ до Redux-стану (`store`), потрібно **обгорнути весь додаток у спеціальний компонент `Provider`** з бібліотеки `react-redux`.

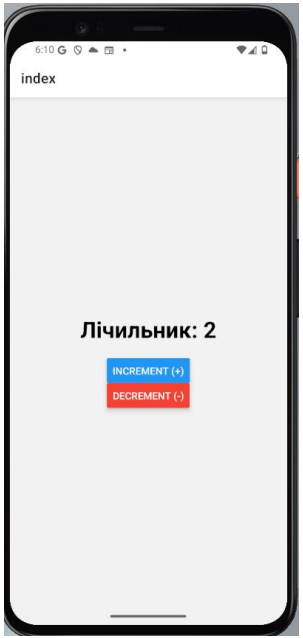
У файлі `App.js` або `_layout.jsx`, залежно від структури проєкту.

```jsx
import { Stack } from 'expo-router';
import { Provider } from 'react-redux';
import { store } from '../store';

no usages  new *
export default function Layout() {
  return (
    <Provider store={store}>
        <Stack />
    </Provider>
  );
}
```

Створюємо компонент **CounterScreen**

Компонент `<Provider>` робить Redux store доступним для всіх дочірніх компонентів за допомогою хука `useSelector()` та `useDispatch()`.




```jsx
import { useSelector, useDispatch } from 'react-redux';
import { increment, decrement } from '../store/counter/counterSlice';

no usages
export default function CounterScreen() {
    const count = useSelector((state) => state.counter.value);
    const dispatch = useDispatch();

    return (
        <View style={styles.container}>
            <Text style={styles.counter}>Лічильник: {count}</Text>
            <Button
                title="Increment (+)"
                onPress={() => dispatch(increment())}
            />
            <Button
                title="Decrement (-)"
                onPress={() => dispatch(decrement())}
                color="#f44336"
            />
        </View>
    );
}
```

useSelector

`useSelector` — це React-хук з бібліотеки `react-redux`, який дозволяє компоненту **отримати доступ до потрібних значень із Redux store**.

Він витягує дані **з глобального стану** (`state`), які потрібні **тільки цьому компоненту**, і автоматично оновлює компонент при зміні цих даних.

Пояснення:

```
useSelector((state) => state.counter.value)
```

- `state` — це **весь глобальний стан**, який зберігається в Redux Store.
- `state.counter` — ми звертаємося до "гілки" стану `counter`, яка була названа так у `configureStore`.
- `state.counter.value` — ми звертаємося до поля `value`, наприклад, `0`, `1`, `2`, і т.д.

Як це працює?

Redux автоматично “слухає” зміни у вибраній частині стану, і якщо вона зміниться — **React-компонент автоматично перерендериться**.

useDispatch

`useDispatch` — це хук з `react-redux`, який повертає функцію **`dispatch()`**. Вона дозволяє **відправити (запустити) дію (action)**, яка змінює стан у Redux.

Синтаксис:

```
const dispatch = useDispatch();
```

- `dispatch` — це функція, щоб запустити одну з дій (наприклад, `increment()`).
- Без `useDispatch` ти не можна ініціювати зміну стану з компонента.

Що таке **dispatch(increment())?**

`dispatch(increment())`

- `increment()` — це **action creator**, автоматично згенерований `createSlice`.
- Він повертає об'єкт: `{ type: 'counter/increment' }`
- `dispatch(...)` передає цю дію до **Redux Store**
- Store дивиться, **який reducer пов'язаний із типом дії `counter/increment`**
- Він знаходить `increment: (state) => { state.value += 1 }`
- Стан оновлюється → React побачив зміну через `useSelector` → компонент перерендерився

Повна схема роботи:

```
User натискає кнопку
        ↓
Компонент викликає dispatch(increment())
        ↓
Redux знаходить відповідний reducer у slice
        ↓
Reducer змінює стан (state)
        ↓
useSelector бачить зміну потрібного значення
        ↓
React перерендерює компонент
```

createAsyncThunk

`createAsyncThunk` — це функція з Redux Toolkit, яка **автоматизує створення асинхронної логіки у Redux**.

Вона:

- Створює **асинхронну дію**, яку можна `dispatch`-ити з компоненту.
- Автоматично генерує **три стани**:
  - `pending` – запит стартував
  - `fulfilled` – запит успішно завершився
  - `rejected` – запит завершився з помилкою
- **Підтримує будь-які асинхронні операції:** `fetch`, `axios`, `async/await`, `Promise`.

Коли використовувати **createAsyncThunk?**

Якщо ви виконуєте **будь-який API-запит** (GET, POST, PUT, DELETE) — `createAsyncThunk` є **оптимальним і рекомендованим способом** обробки асинхронної логіки у Redux Toolkit.

Типові сценарії використання:

- Отримання списків з сервера (наприклад: продукти, пости, юзери)
- Відправка форм або даних (реєстрація, зворотній зв'язок)
- Авторизація / логін користувача
- Завантаження профілю, деталей замовлення, картки товару

Синтаксис **createAsyncThunk**

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

// 1. Створення асинхронної дії loginUser
export const loginUser = createAsyncThunk(
    'auth/loginUser', // Унікальний тип дії (typePrefix)

    async (credentials, thunkAPI) => { // Асинхронна функція (payloadCreator)
        try {
            // Виконуємо запит на логін
            const response = await axios.post( url: '/api/login', credentials);

            // Повертаємо результат у fulfilled → action.payload
            return response.data;
        } catch (error) {
            // У разі помилки передаємо її у rejected → action.payload
            return thunkAPI.rejectWithValue( value: 'Помилка авторизації');
        }
    }
);
```

typePrefix

Це основа для генерації типів дій:

- `'auth/loginUser/pending'`
- `'auth/loginUser/fulfilled'`
- `'auth/loginUser/rejected'`

Має бути унікальним у межах slice або додатку.

payloadCreator

```
payloadCreator: async (arg, thunkAPI) => { ... }
```

`payloadCreator` — це **асинхронна функція**, яка виконує основну логіку `createAsyncThunk`

(наприклад: API-запити, обчислення, побічні ефекти).

📦 **Утиліти всередині `thunkAPI`:**

Метод	Призначення
`dispatch`	Дозволяє відправляти інші дії зсередини thunk
`getState`	Доступ до поточного стану Redux
`rejectWithValue`	Повертає кастомну помилку, яка потрапить в `action.payload`

return

Якщо `return` відпрацьовує успішно:

- Дія переходить у стан **`fulfilled`**
- Значення, яке повернув `return`, → потрапляє в `action.payload`

```
return response.data; // → action.payload = response.data
```

Якщо трапляється `throw` або `rejectWithValue(...)`:

- Дія переходить у стан **`rejected`**
- Помилка або передане повідомлення → потрапляє в `action.payload` (для `rejectWithValue`) або в `action.error` (для `throw`)

Обробка **createAsyncThunk** через **extraReducers**

Для обробки `createAsyncThunk` ми використовуємо **поле `extraReducers`**.

Воно дозволяє описати, **як змінюється стан** залежно від статусу запиту (`pending`, `fulfilled`, `rejected`).

У великих проєктах кожен `createAsyncThunk` має власну логіку: авторизація, реєстрація, 2FA, оновлення токену тощо.

```
    extraReducers:(builder) => {
        authenticationRequestReducer(builder)
    }
);
```

Як виглядає одна така функція?

```
import { loginUser } from './counterAsyncThunks';

// Функція, яка обробляє всі статуси для authenticationRequest
export const authenticationRequestReducer = (builder) => {
    builder
        // Обробка pending — початок запиту
        .addCase(loginUser.pending, (state) => {
            state.loading = true;
        })
        // Обробка fulfilled — успішна відповідь
        .addCase(loginUser.fulfilled, (state, action) => {
            state.user = action.payload.user;
            state.token = action.payload.token;
            state.loading = false;
        })
        // Обробка rejected — помилка запиту
        .addCase(loginUser.rejected, (state, action) => {
            state.loading = false;
            state.error = action.payload;
        });
};
```

Виклик **createAsyncThunk** у компоненті

Після того як ми створили `createAsyncThunk` (наприклад `loginUser`) та обробили його в `extraReducers`, ми можемо викликати його з будь-якого компонента.

```
export default function LoginScreen() {
    const dispatch = useDispatch();

    const loading = useSelector( selector: (state) => state.auth.loading);
    const error = useSelector( selector: (state) => state.auth.error);

    const handleLogin = () => {
        dispatch(loginUser({ email: 'user@example.com', password: 'secret' }));
    };

    return (
        <>
            <Button onPress={handleLogin} title="Увійти" disabled={loading} />
            {error && <Text>{error}</Text>}
        </>
    );
}
```

Обробка помилок з **rejectWithValue**

`createAsyncThunk` дозволяє обробляти помилки, що виникають під час асинхронних запитів, за допомогою функції **`rejectWithValue()`** — вона дозволяє передати **користувацьке повідомлення про помилку** у `action.payload` при `rejected`.

Якщо її викликати, `action.type` буде `rejected`, а `action.payload` = тим що було передано.

Це дозволяє показувати кастомізовані повідомлення користувачу, а не просто `error.message`.

```
// 1. Створення асинхронної дії loginUser
export const loginUser = createAsyncThunk(
    'auth/loginUser', // Унікальний тип дії (typePrefix)

    async (credentials, thunkAPI) => { // Асинхронна функція (payloadCreator)
        try {
            // Виконуємо запит на логін
            const response = await axios.post( url: '/api/login', credentials);

            // Повертаємо результат у fulfilled → action.payload
            return response.data;
        } catch (error) {
            // У разі помилки передаємо її у rejected → action.payload
            return thunkAPI.rejectWithValue( value: 'Помилка авторизації');
        }
    }
);
```

Як обробити помилку у **slice (extraReducers)**

```
.addCase(loginUser.rejected, (state, action) => {
    state.loading = false;
    state.error = action.payload || 'Невідома помилка';
});
```

- `action.payload` → кастомна помилка
- `action.error.message` → fallback, якщо `rejectWithValue` не використовувався