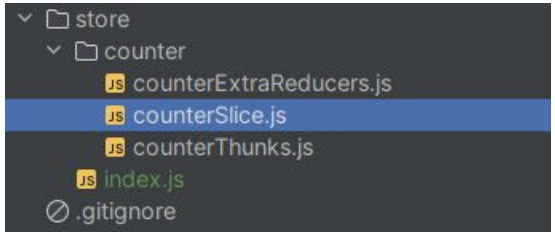
Розробка мобільних додатків

Лекція 12 - State Management у React Native(продовження)




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

Show usages  new *
export const store = configureStore({
    reducer: {
        counter: counterReducer,
    },
});
```

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

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

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

redusers

Редюсер — це функція, яка визначає, як змінюється стан додатку у відповідь на дії користувача або системи.

Редюсери в Redux повинні бути **чистими функціями**, що означає:

- **Передбачуваність**
  → для однакових вхідних даних (`state`, `action`) завжди повертається однаковий результат
- **Відсутність побічних ефектів (side effects)**
  → редюсер не повинен взаємодіяти із зовнішнім середовищем
- **Обмеження на операції**
  → заборонено виконувати:
  - HTTP-запити (API виклики)
  - роботу з випадковістю (`Math.random()`)
  - операції з датами (`Date.now()` тощо)

extraReducers

ExtraReducers у Redux Toolkit (RTK) — це спеціальне поле в `createSlice`, яке дозволяє слайсу реагувати на екшени, що були визначені **за межами** цього слайсу.

Якщо звичайні `reducers` автоматично генерують екшени для внутрішніх потреб слайсу, то `extraReducers` призначені для "прослуховування" зовнішніх подій.

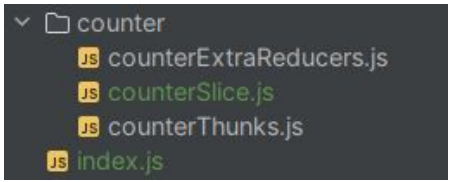
Коли варто використовувати ExtraReducers?

1. **Асинхронні запити (`createAsyncThunk`):** Найчастіший сценарій. Ви створюєте thunk окремо, а слайс має реагувати на його стани: `pending`, `fulfilled` або `rejected`.
2. **Глобальні події:** Коли один екшен має змінити стан у кількох різних слайсах (наприклад, екшен `LOGOUT`, який має очистити дані в усіх частинах стору).
3. **Екшени з інших слайсів:** Якщо слайс "А" хоче оновити свій стан, коли спрацьовує екшен зі слайсу "Б".

Thunk

Thunk — це функція, яка **відкладає виконання логіки** та дозволяє виконувати **асинхронні операції перед зміною стану**.

```
Show usages
export const fetchData = createAsyncThunk(
    'todos/fetchData',
    async () => {
        const response = await fetch('https://jsonplaceholder.typicode.com/todos/1');
        const data = await response.json();
        return data.id;
    }
);
```



createAsyncThunk

Функція приймає два аргументи:

1. **Type string:** Назва нашої дії (наприклад, `'users/fetchAll'`). Вона потрібна для дебагу.
2. **Payload Creator:** Асинхронна функція, де ми пишемо наш код запиту.

Код:

```
export const fetchUsers = createAsyncThunk(
  'users/fetchAll', // Ім'я дії
  async () => {     // Сама робота
    const response = await axios.get('/users');
    return response.data; // Те, що повернемо, стане Payload
  }
);
```

Життєвий цикл запиту

1. pending (очікування)

- запит **відправлено**
- ще немає результату

state.loading = true

2. fulfilled (успіх)

- запит **успішно завершено**
- отримані дані

state.loading = false
state.data = action.payload

3. rejected (помилка)

- запит **завершився з помилкою**

state.loading = false
state.error = action.error

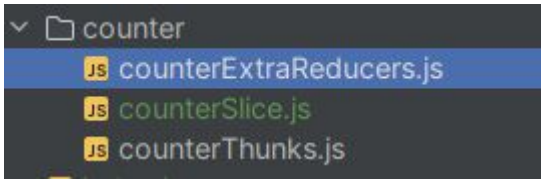
Приклад використання

Builder — це об’єкт API Redux Toolkit, який використовується для опису реакції state на різні дії в `extraReducers`.

```
extraReducers: (builder) => {
    builder
        .addCase(fetchData.pending, (state) => {
            state.loading = true;
        })
        .addCase(fetchData.fulfilled, (state, action) => {
            state.loading = false;
            state.data = action.payload;
        })
        .addCase(fetchData.rejected, (state, action) => {
            state.loading = false;
            state.error = action.error;
        });
},
```

Організація **extraReducers** у проєкті

- У великих проєктах доцільно **виносити логіку extraReducers в окремі файли**
- Це дозволяє **розділити відповідальність** та уникнути перевантаження slice




```
extraReducers: (builder) => {
    fetchDataReducer(builder);
    fetchCountReducer(builder);
    resetCountReducer(builder);
    resetAllReducer(builder);
},
```

```
import { fetchData, resetCountOnServer } from './counterThunks';

Show usages
export const fetchCountReducer = (builder) => {
    builder
        .addCase(fetchData.pending, (state) => {
            state.loading = true;
            state.error = null;
        })
        .addCase(fetchData.fulfilled, (state, action) => {
            state.loading = false;
            state.value = action.payload;
        })
        .addCase(fetchData.rejected, (state, action) => {
            state.loading = false;
            state.error = action.error.message;
        });
};
```

thunkAPI

`thunkAPI` — це об'єкт, який передається другим аргументом у асинхронну функцію. Він надає доступ до інструментів керування стором та запитом безпосередньо "всередині" Thunk.

```
export const fetchDataWithThunkAPI = createAsyncThunk(
    'counter/fetchDataWithThunkAPI',
    async (userId, thunkAPI) => {
        // getState — читаємо поточний стан стору
        const state = thunkAPI.getState();
        console.log('current value:', state.counter.value);

        // dispatch — викликаємо інший екшн з середини thunk
        // thunkAPI.dispatch(someAction());

        try {
            const response = await fetch(`https://jsonplaceholder.typicode.com/todos/${userId}`);
            return await response.json();
        } catch (error) {
            // rejectWithValue — передаємо свою помилку в rejected
            return thunkAPI.rejectWithValue('Помилка завантаження');
        }
    }
);
```

Методи та властивості **thunkAPI**

dispatch	Відправка (dispatch) інших actions	Виклик додаткових дій всередині thunk
getState	Отримання поточного глобального state	Доступ до даних перед виконанням запиту
rejectWithValue	Повернення кастомної помилки в `rejected`	Обробка помилок з власним повідомленням
fulfillWithValue	Повернення кастомного payload при успішному виконанні	Коли потрібно змінити стандартний результат

Коли потрібен **getState()** всередині **Thunk?**

Проблема Іноді для запиту нам потрібні дані, яких немає в аргументах (наприклад, токен авторизації).

Рішення Замість того, щоб передавати ці дані з компонента, ми беремо їх прямо зі стору через `getState()`.

```
export const fetchPrivateData = createAsyncThunk(
  'data/fetchPrivate',
  async (_, { getState }) => {
    const token = getState().auth.token; // Дістаємо токен зі стору
    const response = await axios.get('/private', {
      headers: { Authorization: `Bearer ${token}` }
    });
    return response.data;
  }
);
```

Обробка помилок: rejectWithValue

Проблема Якщо сервер повернув `400 Bad Request` з детальним описом (наприклад, "Пароль надто короткий"), за замовчуванням RTK покладе в `action.error` лише загальний текст "Request failed".

Як це працює Метод `rejectWithValue` дозволяє нам "спіймати" відповідь сервера і передати її в редьюсер як `payload` для стану `rejected`.

Код:

```
async (credentials, { rejectWithValue }) => {
  try {
    const res = await axios.post('/login', credentials);
    return res.data;
  } catch (err) {
    // Дані з сервера потраплять у action.payload у блоці rejected
    return rejectWithValue(err.response.data);
  }
}
```

Оптимізація: Параметр **condition**

Це спеціальна функція в налаштуваннях Thunk, яка виконується **ДО** запуску асинхронного коду. Якщо вона поверне `false`, запит взагалі не почнеться.

Навіщо це потрібно? Щоб не робити зайвий запит. Наприклад, якщо ми вже завантажуємо список користувачів, і користувач знову тисне "Завантажити", ми можемо це заблокувати.

Код:

```
export const fetchUsers = createAsyncThunk(
  'users/fetch',
  async () => { ... },
  {
    condition: (_, { getState }) => {
      const { users } = getState();
      if (users.isLoading) return false; // Запит не буде відправлений
    }
  }
);
```

Redux Persist — Збереження стану

Проблема: стейт зникає при закритті додатку

- **State Redux зберігається в оперативній пам’яті (RAM)**
  → існує лише під час роботи додатку
- **Після закриття додатку пам’ять очищається**
  → всі дані стану втрачаються
- **Redux за замовчуванням не має механізму збереження**
  → state не записується у постійне сховище

Наслідки

- повторна авторизація користувача
- втрата введених або завантажених даних
- погіршення користувацького досвіду (UX)

Redux Persist

Redux Persist — це бібліотека-посередник, яка забезпечує **автоматичну синхронізацію стану Redux із довготривалим (персистентним) сховищем пристрою**.

Механізм роботи

- **Persist (збереження)**
  → при кожній зміні store стан серіалізується у рядковий формат
  → записується в енергонезалежне сховище (наприклад, AsyncStorage)
- **Rehydrate (відновлення)**
  → під час запуску додатку стан зчитується зі сховища
  → десеріалізується та відновлюється у Redux Store до рендерингу інтерфейсу

Основні переваги

- **Збереження стану між сесіями**
- **Підтримка офлайн-режиму роботи**
- **Миттєвий доступ до останнього актуального стану**
- **Покращення користувацького досвіду (UX)**

Інсталяція

Необхідні модулі

- **redux-persist**
  → ядро бібліотеки, яке забезпечує механізми збереження та відновлення стану
- **@react-native-async-storage/async-storage**
  → офіційний драйвер локального сховища для React Native

Інсталяція

npm install redux-persist
npx expo install @react-native-async-storage/async-storage

persistReducer та **persistStore**

persistReducer — це функція, яка **обгортає reducer і додає до нього логіку збереження**.

Що робить

- перехоплює зміни state
- визначає, **які дані потрібно зберігати**
- інтегрує reducer з Redux Persist

```
import counterReducer from './counter/counterSlice';

const counterPersistConfig = {
    key: 'counter',
    storage: AsyncStorage,
    whitelist: ['value'],
};

const persistedCounterReducer = persistReducer(counterPersistConfig, counterReducer);

Show usages  new *
export const store = configureStore({
    reducer: {
        counter: persistedCounterReducer,
    },
```

persistReducer та persistStore

persistStore — це функція, яка **запускає механізм збереження та відновлення state**.

Що робить

- ініціалізує процес persist
- відновлює state при запуску (rehydrate)
- повертає об’єкт **persistor**

Конфігурація: **persistConfig**

Об'єкт конфігурації `persistConfig`, який визначає правила для механізму збереження.

Основні параметри

- **`key`:** Унікальний рядок (ідентифікатор), який використовується як префікс для збережених даних у AsyncStorage. Зазвичай `'root'`.
- **`storage`:** Драйвер сховища. У нашому випадку — `AsyncStorage` з бібліотеки `@react-native-async-storage/async-storage`.
- **`whitelist`:** Масив назв слайсів, які **потрібно** зберігати. Усе інше буде ігноруватися.
- **`blacklist`:** Масив назв слайсів, які **не потрібно** зберігати (наприклад, помилки або статуси завантаження).

Якщо ви не вкажете ні `whitelist`, ні `blacklist`, бібліотека намагатиметься зберегти **весь** глобальний стан, що може призвести до проблем із продуктивністю.

```
const counterPersistConfig = {
    key: 'counter',
    storage: AsyncStorage,
    whitelist: ['value'],
};
```

whitelist vs blacklist

Вибір стратегії фільтрації даних при збереженні стану.

Порівняння

- **Blacklist (Підхід "Виключення"):** Ви зберігаєте все, окрім того, що явно заборонили.
  - *Ризик:* Якщо додати новий слайс із конфіденційними даними (наприклад, `TemporaryCreditCardData`) і не додати його в чорний список, він потрапить у незахищене сховище.
- **Whitelist (Підхід "Дозвіл"):** Зберігає лише те, що явно дозволили.
  - *Перевага:* Це безпечніше. Ви точно знаєте, які дані потрапляють на диск пристрою.

Рекомендується використовувати **whitelist**, щоб контролювати обсяг даних у `AsyncStorage`.

Процес **Rehydration** (відновлення даних)

Rehydration — це критичний етап життєвого циклу застосунку, під час якого **збережений стан з персистентного сховища відновлюється у пам’ять (Redux Store)**.

Механізм роботи (послідовність)

1. **Ініціалізація (Boot)**
   → додаток запускається
   → Redux Store створюється з початковим станом (*initialState*)
2. **Перевірка сховища (Persist Check)**
   → middleware перевіряє наявність збережених даних у сховищі (AsyncStorage)
3. **Зчитування даних (Extraction)**
   → дані зчитуються та десеріалізуються з JSON-формату
4. **Dispatch системної дії (Rehydrate Action)**
   → у Store відправляється службова дія **persist/REHYDRATE**
   → містить відновлений стан
5. **Оновлення стану (Merge)**
   → редюсери інтегрують отримані дані
   → початковий стан замінюється або доповнюється відновленим

Налаштування **store.js** з **persistReducer**

Практична реалізація конфігурації стору в Redux Toolkit із застосуванням persistence.

```
import { configureStore } from '@reduxjs/toolkit';
import { persistStore, persistReducer } from 'redux-persist';
import AsyncStorage from '@react-native-async-storage/async-storage';
import { FLUSH, REHYDRATE, PAUSE, PERSIST, PURGE, REGISTER } from 'redux-persist';
import counterReducer from './counter/counterSlice';

const counterPersistConfig = {
    key: 'counter',
    storage: AsyncStorage,
    whitelist: ['value'],
};

const persistedCounterReducer = persistReducer(counterPersistConfig, counterReducer);

Show usages  new *
export const store = configureStore({
    reducer: {
        counter: persistedCounterReducer,
    },
    middleware: (getDefaultMiddleware) =>
        getDefaultMiddleware({
            serializableCheck: {
                ignoredActions: [FLUSH, REHYDRATE, PAUSE, PERSIST, PURGE, REGISTER],
            },
        }),
});

Show usages  new *
export const persistor = persistStore(store);
```

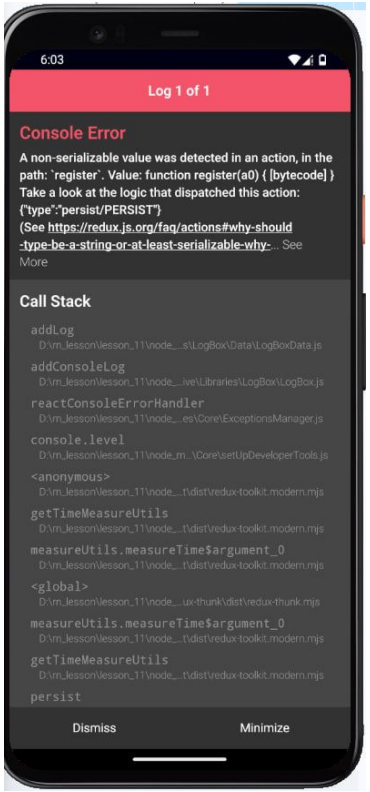
Виправлення помилки "non-serializable value"

Помилка, яка виникає через те, що `Redux Toolkit` за замовчуванням забороняє передавати не-серіалізовані дані (функції, проміси, спеціальні об'єкти бібліотек) через екшени.

Як це працює (Механізм конфлікту)
`redux-persist` використовує внутрішні типи екшенів (наприклад, FLUSH, PAUSE, PERSIST), які містять функції або спеціальні об'єкти. Валідатор RTK бачить їх і видає помилку в консоль.

При налаштуванні стору потрібно ігнорувати ці системні екшени в `serializableCheck`:

```
Show usages  new *
export const store = configureStore({
    reducer: {
        counter: persistedCounterReducer,
    },
    middleware: (getDefaultMiddleware) =>
        getDefaultMiddleware({
            serializableCheck: {
                ignoredActions: [
                    FLUSH,
                    REHYDRATE,
                    PAUSE,
                    PERSIST,
                    PURGE,
                    REGISTER
                ],
            },
        }),
});
```



PersistGate: затримка рендеру

Компонент-обгортка, який синхронізує життєвий цикл React-компонентів із процесом відновлення стану.

Оскільки зчитування з `AsyncStorage` є асинхронним, додаток може почати рендерити екрани до того, як Redux отримає збережені дані. `PersistGate` призупиняє рендеринг дочірніх компонентів, поки стан не буде повністю відновлений.

Це запобігає ситуаціям, коли користувач на мить бачить екран логіну (бо стейт ще порожній), а потім його "перекидає" на головну.

```
import { Stack } from 'expo-router';
import { Provider } from 'react-redux';
import { PersistGate } from 'redux-persist/integration/react';
import { store, persistor } from '../store/index';

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

Створення кастомних лоадерів для **PersistGate**

Візуальний супровід процесу ініціалізації додатка.

Як це працює Проп `loading` у `PersistGate` приймає будь-який React-компонент. Це ідеальне місце для відображення:

- Логотипа компанії.
- Анімованого індикатора завантаження.

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

Багаторівневий **Persist:** вкладені конфігурації

Що це: Архітектурний підхід, де різні слайси мають різні правила збереження або використовують різні сховища.

Механізм дії: Ви можете створити окремі `persistConfig` для кожного редьюсера замість одного глобального.

- Наприклад: Кошик зберігається в `AsyncStorage`, а налаштування додатка - в швидшому `MMKV`.

Основні елементи

- **Nested Persist:** Використання `persistReducer` всередині `combineReducers`.
- **Granular Control:** Можливість задати різний термін життя для різних частин стору.

```
// 🔹 persist для auth (критичні дані)
const authPersistConfig = {
  key: 'auth',
  storage: AsyncStorage, // більш стабільне сховище
}

// 🔹 persist для settings (швидкий доступ)
const settingsPersistConfig = {
  key: 'settings',
  storage: MMKV, // швидше за AsyncStorage
}

// 🔹 reducers
import authReducer from './authSlice'
import settingsReducer from './settingsSlice'
import tempReducer from './tempSlice'

// 🔹 combineReducers з вкладеним persist
const rootReducer = combineReducers({
  auth: persistReducer(authPersistConfig, authReducer),
  settings: persistReducer(settingsPersistConfig, settingsReducer),
  temp: tempReducer, // ❌ не зберігається
})

// 🔹 глобальний persist (опціонально)
const rootPersistConfig = {
  key: 'root',
  storage: AsyncStorage,
  blacklist: ['temp'], // не зберігати тимчасові дані
}

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

Міграції

Міграції стану — це механізм, який забезпечує **перетворення збережених (застарілих) даних у новий формат**, що відповідає актуальній версії додатку.

Механізм роботи

1. **Version**
   → у `persistConfig` задається версія state
2. **Migration Manifest**
   → описується функція трансформації старого state
3. **Автоматичне оновлення**
   → при запуску Redux Persist:
   - порівнює версії
   - виконує міграцію
   - передає оновлений state у Store

```
import { createMigrate } from 'redux-persist'

// 🔹 опис міграцій
const migrations = {
  1: (state) => {
    // v0 → v1
    return {
      ...state,
      fullName: state.userName, // перейменування поля
    }
  },

  2: (state) => {
    // v1 → v2
    return {
      ...state,
      isLoggedIn: true, // нове поле
    }
  },
}

// 🔹 конфігурація persist
const persistConfig = {
  key: 'root',
  version: 2,
  storage: AsyncStorage,
  migrate: createMigrate(migrations),
}
```

Трансформації

Transforms — це механізм, який дозволяє **змінювати (перетворювати) state під час збереження та відновлення**.

Навіщо це потрібно

- зберігати **лише частину даних**
- **шифрувати** або маскувати чутливу інформацію
- змінювати формат даних (наприклад, Date → string)
- оптимізувати обсяг збереження

Механізм роботи

- **inbound** (запис у сховище)
  → state трансформується перед збереженням
- **outbound** (читання зі сховища)
  → дані відновлюються у потрібний формат

```
// inbound — перед збереженням в AsyncStorage
// outbound — після відновлення з AsyncStorage
const doubleValueTransform = createTransform(
    (inboundState) => ({ ...inboundState, value: inboundState.value * 2 }),
    (outboundState) => ({ ...outboundState, value: outboundState.value / 2 }),
    { whitelist: ['counter'] }
);

const counterPersistConfig = {
    key: 'counter',
    storage: AsyncStorage,
    whitelist: ['value'],
    transforms: [doubleValueTransform],
};
```

Очищення збереженого стану

Purge — це операція, яка забезпечує **повне видалення збереженого state із персистентного сховища**.

Призначення

- повний **reset даних додатку**
- очищення стану при **виході користувача (logout)**
- усунення некоректного або застарілого state
- відновлення початкового стану після помилок

Реалізація

```
import { persistor } from './store'

persistor.purge()
```

видаляє всі дані, збережені через Redux Persist

Продуктивність: вплив на час запуску (TTR)

Вплив великого обсягу збережених даних на **Time To Render** (час до першого відображення інтерфейсу).

Оскільки `PersistGate` блокує рендер до завершення читання з диска, великий стор (наприклад, 10 МБ JSON) може додати 1–2 секунди затримки при старті.

Оптимізація

- **Granular Persistence:** Зберігайте лише те, що дійсно необхідно (ID замість повних об'єктів).
- **Throttle:** Використовуйте параметр `throttle` у конфігурації, щоб обмежити частоту записів на диск (наприклад, не частіше ніж раз на 500 мс).

```
const persistConfig = {
  key: 'root',
  storage: AsyncStorage,

  // • запис не частіше ніж раз на 500 мс
  throttle: 500,
}
```

Middleware

Middleware

Що це таке Middleware (проміжне ПЗ) — це шар коду, який розташовується між моментом відправки екшену (`dispatch`) і моментом, коли цей екшен потрапляє в редьюсер.

Як це працює (Аналогія) конвеєр на заводі:

1. **Action** — це деталь, яку кладуть на стрічку.
2. **Middleware** — це працівники вздовж стрічки. Кожен може подивитися на деталь, записати дані про неї в журнал, змінити її або навіть прибрати з конвеєра (зупинити екшен).
3. **Reducer** — це фінальна станція, де деталь встановлюється в готовий виріб (State).

Життєвий цикл:

Послідовність виконання

1. **Dispatch**
   → виклик `store.dispatch(action)` у компоненті
2. **Вхід у Middleware**
   → action передається у перший middleware у ланцюжку
3. **Передача далі (next)**
   → кожен middleware викликає `next(action)`
   → якщо `next` не викликати — action не дійде до reducer
4. **Reducer phase**
   → після проходження всіх middleware action обробляється редюсерами
   → формується новий state
5. **Зворотний шлях**
   → керування повертається назад через middleware
   → можливе виконання додаткової логіки після оновлення state

Створення власного **Middleware (Logger**
п

```
const myLogger = (store) => (next) => (action) => {
    console.log('Тип екшену:', action.type);
    console.log('Попередній стан:', store.getState());

    const result = next(action);

    console.log('Новий стан:', store.getState());

    return result;
};

const counterPersistConfig = {key: 'counter'...};

const persistedCounterReducer = persistReducer(counterPersistConfig, counterReducer);

Show usages  new *
export const store = configureStore({
    reducer: {
        counter: persistedCounterReducer,
    },
    middleware: (getDefaultMiddleware) =>
        getDefaultMiddleware({
            serializableCheck: {
                ignoredActions: [
                    FLUSH, REHYDRATE, PAUSE, PERSIST, PURGE, REGISTER
                ],
            },
        }).concat(myLogger),
});
```

```
LOG  Попередній стан: {"counter": {"_persist": {"rehydrated": true, "version": -1},
 "data": null, "error": null, "loading": false, "value": 16}}
LOG  Новий стан: {"counter": {"_persist": {"rehydrated": true, "version": -1}, "dat
a": null, "error": null, "loading": false, "value": 17}}
LOG  Тип екшену: counter/increment
LOG  Попередній стан: {"counter": {"_persist": {"rehydrated": true, "version": -1},
 "data": null, "error": null, "loading": false, "value": 17}}
LOG  Новий стан: {"counter": {"_persist": {"rehydrated": true, "version": -1}, "dat
a": null, "error": null, "loading": false, "value": 18}}
LOG  Тип екшену: counter/decrement
LOG  Попередній стан: {"counter": {"_persist": {"rehydrated": true, "version": -1},
 "data": null, "error": null, "loading": false, "value": 18}}
LOG  Новий стан: {"counter": {"_persist": {"rehydrated": true, "version": -1}, "dat
a": null, "error": null, "loading": false, "value": 17}}
```

Коли писати свій **Middleware?**

Визначення бізнес-завдань, які найкраще вирішуються на рівні проміжного шару.

Приклади на практиці

- **Аналітика:** Надсилати подію в Google Analytics на кожен специфічний екшен (наприклад, `cart/addItem`).
- **Логування помилок:** Перехоплювати всі екшени, що закінчуються на `/rejected`, і надсилати їх у систему моніторингу (Sentry).
- **Синхронізація:** При кожній зміні стану зберігати певні дані в зовнішній базі даних (як це робить Redux Persist).
- **Toast-повідомлення:** Показувати спливаюче вікно при успішному виконанні будь-якого асинхронного запиту.

Middleware для обробки помилок

Автоматизована система сповіщення користувача про помилки мережі або валідації без написання коду в кожному окремому Thunk або компоненті.

Як це працює (Механізм) Middleware перехоплює всі екшени, що мають статус `rejected`. Якщо екшен містить помилку, Middleware викликає нативний метод показу Toast або Alert.

```
const errorToastMiddleware = (store) => (next) => (action) => {
  if (action.type.endsWith('/rejected')) {
    // Викликаємо Toast з повідомленням про помилку
    Toast.show({
      type: 'error',
      text1: 'Помилка!',
      text2: action.payload || 'Щось пішло не так'
    });
  }
  return next(action);
};
```

Перевірка авторизації через **Middleware**

Централізований механізм контролю доступу, який перевіряє валідність сесії при кожній спробі виконання чутливих операцій.

Як це працює (Принцип дії)

1. Middleware перехоплює екшени, які вимагають авторизації.
2. Перевіряє стан `auth.token` у глобальному сторі за допомогою `store.getState()`.
3. Якщо токен відсутній або прострочений, Middleware зупиняє екшен (не викликає `next(action)`) і перенаправляє користувача на екран логіну.

Переваги Не потрібно писати перевірку `if (!isLoggedIn)` у кожному Thunk або компоненті — безпека працює на рівні "транспортного шару" даних.

Робота з таймерами та зовнішніми подіями

Використання Middleware як довготривалого середовища для керування асинхронними процесами, що не вписуються в логіку редьюсерів.

Механізм дії Middleware може ініціювати `setTimeout` або `setInterval`, які будуть відправляти екшени через певний час.

- **Приклад:** Після екшену `login` Middleware запускає таймер на 30 хвилин. По закінченню таймера автоматично діспатчиться `logout`.

Нюанси використання Це ідеальне місце для підписки на події акселерометра, гіроскопа або стану мережі (NetInfo), оскільки Middleware має постійний доступ до `dispatch`.

getDefaultMiddleware().concat()

Метод безпечного розширення стандартного набору Middleware в Redux Toolkit без ризику зламати базову логіку (Thunk, DevTools).

Як це працює Ми отримуємо масив стандартних інструментів і додаємо до нього кастомні Middleware.

```
export const store = configureStore({
  reducer: rootReducer,
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware().concat(errorToastMiddleware, loggerMiddleware),
});
```

Порядок у `.concat()` має значення: Middleware виконуються зліва направо. Зазвичай кастомні логери ставлять останніми, щоб вони бачили вже оброблені іншими інструментами дані.

Відмінність **Middleware** від **extraReducers**

Критерій	extraReducers	Middleware
Мета	Зміна даних у State	Виконання побічних ефектів (Side Effects)
Доступ	Тільки до локального State	До всього Store (dispatch + getState)
Здатність	Не може зупинити екшен	Може зупинити або змінити екшен

Вплив **Middleware** на швидкість роботи додатку

Middleware виконуються **послідовно (синхронно)** в ланцюжку обробки дій, що може впливати на **затримку оновлення інтерфейсу**.

Потенційні проблеми

- **Блокування основного потоку**
  → “важкі” обчислення у middleware можуть викликати затримки або “фризи” UI
- **Зростання затримки обробки**
  → кожен middleware додає час до обробки action
- **Нескінченні цикли (Recursion)**
  → повторний dispatch того самого action без умов призводить до **переповнення стеку**

Взаємодія між **Slice (extraReducers).**

Концепція **"Isolation vs Communication"** між слайсами

Архітектурна дилема проектування State Management: баланс між ізоляцією логіки окремого модуля та необхідністю реагувати на глобальні події в системі.

Як це працює

- **Isolation (Ізоляція):** Кожен слайс відповідає виключно за свою ділянку даних (наприклад, тільки за список товарів). Це робить код передбачуваним і легким для тестування.
- **Communication (Комунікація):** Коли стається подія, що впливає на весь додаток (наприклад, вихід із профілю), багато слайсів повинні синхронно змінити свій стан (очистити кошик, скинути налаштування, видалити токени).

Проблеми та обмеження Пряма зміна стану одного слайса всередині редьюсера іншого слайса технічно неможлива та архітектурно шкідлива, оскільки порушує принцип єдиної відповідальності.

Приклад архітектури: **AuthSlice** та **CartSlice**

Практична модель взаємодії двох незалежних модулів (Авторизація та Кошик) у типовому мобільному додатку.

Механізм взаємодії

1. **AuthSlice:** Керує станом входу користувача. Має екшен `logout`.
2. **CartSlice:** Керує масивом товарів. Повинен знати, коли користувач виходить, щоб очистити персональні дані.
3. **Потік:** `dispatch(auth/logout)` → `AuthSlice` занулює токен → `CartSlice` (через `extraReducers`) очищує масив товарів.

Де застосовується на практиці Будь-які системи, де розмонтування одного модуля вимагає "прибирання" в інших (наприклад, зміна теми додатку, скидання фільтрів при зміні категорії).

Проблема: Чому **CartSlice** не може напряму змінити **AuthSlice**

У Redux Toolkit реалізовано принцип **Unidirectional Data Flow** (односпрямований потік даних) та жорстка прив'язка редьюсерів до їхніх гілок стану.

Чому це неможливо (Обмеження)

- **Scope (Область видимості):** Редьюсер слайса `Cart` отримує як аргумент `state`, який є лише частиною глобального дерева (`rootState.cart`). Він фізично не має доступу до `rootState.auth`.
- **Immutability:** Redux базується на імутабельних оновленнях. Спроба "дотягнутися" до сусідньої гілки призвела б до неконтрольованих побічних ефектів і неможливості відстежити зміни.

Архітектурне рішення

Замість прямої взаємодії між slice використовується механізм **extraReducers**

- slice не змінює інший slice напряму
- він **реагує на зовнішні дії (actions)**
- таким чином реалізується взаємодія через події

Використання об'єкта **builder**

`builder` — це спеціальний об'єкт-конструктор, який використовується в `extraReducers` для декларативного опису реакцій на екшени.

```
extraReducers: (builder) => {
  builder
    .addCase(otherSliceAction, (state, action) => {
      // логіка оновлення нашого стейту
    })
```

Основні компоненти

- **`addCase`:** Для обробки конкретного екшену.
- **`addMatcher`:** Для обробки групи екшенів за певною ознакою (наприклад, усі, що закінчуються на `/pending`).
- **`addDefaultCase`:** Для логіки за замовчуванням, якщо жоден інший кейс не спрацював.

builder.addCase — підписка на екшени іншого слайсу

Метод об'єкта `builder`, який створює пряму відповідність між зовнішнім типом екшену та функцією-редьюсером у поточному слайсі.

Як це працює (Механізм) Коли в системі відбувається `dispatch(action)`, Redux проходить через усі слайси. Якщо тип цього екшену збігається з тим, що вказано в `addCase`, запускається відповідна логіка оновлення стану.

Основні компоненти

- **Action Creator:** Посилання на екшен іншого слайсу (наприклад, `authSlice.actions.logout`).
- **Reducer Function:** Логіка, яка описує, як саме *наш* слайс має змінити свої дані у відповідь.

Код: Реалізація очищення кошика при **auth/logout**

Практичний приклад того, як слайс `Cart` реагує на подію виходу користувача, ініційовану слайсом `Auth`.

```
import { createSlice } from '@reduxjs/toolkit';
import { logout } from '../auth/authSlice'; // імпорт екшену з іншого файлу

const cartSlice = createSlice({
  name: 'cart',
  initialState: { items: [], total: 0 },
  reducers: { /* внутрішні редьюсери */ },
  extraReducers: (builder) => {
    builder.addCase(logout, (state) => {
      // Коли стається logout, ми скидаємо стан кошика
      state.items = [];
      state.total = 0;
    });
  },
});
```

Аналіз механізму: слайс `cart` не створює екшен `logout`, він лише "підслуховує" його. Це дозволяє зберігати логіку авторизації в одному місці, а наслідки — по всьому додатку.

Обробка кількох екшенів (addMatcher)

Високорівневий метод `builder`, який дозволяє обробляти групу екшенів, що відповідають певному критерію (предикату), однією функцією.

Як це працює Predicate function: Перевірка типу екшену (наприклад, чи закінчується він на `/pending`).

Приклад використання

```
builder.addMatcher(
  (action) => action.type.endsWith('/pending'),
  (state) => {
    state.globalLoading = true; // вмикаємо лоадер для будь-якого запиту в слайс
  }
);
```

Потік даних: Один екшен — кілька реакцій

Архітектурна модель "Broadcasting" (мовлення), де одна подія в системі викликає ланцюгову реакцію оновлень у незалежних модулях.

Механізм (Процес)

1. **Подія:** Користувач натискає "Вийти".
2. **Dispatch:** Відправляється єдиний екшен `auth/logout`.
3. **Паралельна реакція:**
    - `AuthSlice` видаляє токен з пам'яті.
    - `CartSlice` очищує список покупок.
    - `ProfileSlice` занулює дані користувача.
    - `SettingsSlice` повертає налаштування за замовчуванням.

Результат: Система залишається синхронізованою

Переваги: Чистота коду та детермінізм

Обґрунтування використання `extraReducers` як стандарту розробки надійних систем.

Основні переваги

- **Decoupling (Розв'язність):** Компоненти не знають про внутрішні зв'язки між слайсами. Вони просто відправляють одну логічну дію.
- **Single Source of Truth:** Логіка того, що має статися при виході, зосереджена в кожному слайсі окремо, а не розкидана по UI-компонентах.
- **Predictability (Передбачуваність):** Весь процес оновлення стану відбувається в межах одного циклу редукції, що спрощує відлагодження.
- **Developer Experience:** Менше дублювання коду та чітка структура "запитання-відповідь" між модулями.

Коли НЕ варто використовувати **extraReducers**

Визначення меж застосування інструменту для запобігання надмірному ускладненню.

Основні антипатерни (Обмеження)

- **Тісна пов'язаність (Tight Coupling):** Якщо ви копіюєте логіку оновлення даних з одного слайса в інший замість того, щоб тримати дані в одному місці.
- **Складні ланцюжки:** Коли один екшен викликає зміни в 10 слайсах, і це стає важко відстежити .
- **Дублювання логіки:** Якщо два слайси зберігають одні й ті самі дані, реагуючи на один екшен — це порушує принцип "Єдиного джерела істини".

createAction

createAction

Функція RTK для створення незалежних екшенів, які не належать до конкретного слайса, але на які можуть підписатися всі.

Як це працює: Іноді подія є занадто абстрактною (наприклад, `APP_RESET` або `SYNC_DATA`). Замість того, щоб "позичати" екшен у `AuthSlice`, ми створюємо його окремо.

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

// Глобальний екшен скидання всієї системи
export const globalReset = createAction('app/reset');

// У будь-якому слайсі:
extraReducers: (builder) => {
  builder.addCase(globalReset, () => initialState);
}
```

Де застосовується на практиці Системні події: зміна мови, перехід в офлайн-режим, повне скидання кешу додатку.

Оптимізація та селектори

Проблема: **useSelector** викликає зайві рендери

Хук **useSelector** підписує компонент на частину Redux state та **ініціює його оновлення при зміні вибраного значення**.

Механізм роботи

- після кожного `dispatch`:
  - Redux оновлює state
  - `useSelector` повторно викликає функцію-селектор
- результат селектора порівнюється з попереднім значенням (за посиланням — reference equality)

```
const users = useSelector((state) =>
  state.users.list.filter((u) => u.active)
)
```

Ключовий нюанс

- якщо селектор повертає **новий об'єкт або масив**, навіть з тим самим вмістом:
  → React вважає значення зміненим
  → виконується рендер компонента

Як працює порівняння за посиланням

Базовий принцип порівняння об'єктів у JavaScript ($A === B$), на якому базується логіка оновлення React.

Як це працює (Механізм)

- **Примітиви:** Порівнюються за значенням.
- **Об'єкти/Масиви:** Порівнюються за адресою в пам'яті.

Нюанси використання Якщо селектор всередині `useSelector` повертає `state.items.filter(...)`, він створює нове посилання при кожному виклику.

```
const a = [1, 2];
const b = [1, 2];
console.log(a === b); // false, бо це різні ділянки пам'яті
```

Reselect

Reselect — це бібліотека (вбудована в Redux Toolkit), яка дозволяє створювати селектори з підтримкою мемоїзації.

Як це працює: Селектор запам'ятовує свої вхідні аргументи та результат. При наступному виклику він перевіряє: "Чи змінилися вхідні дані?".

1. Якщо **ні** — він повертає старий результат (те саме посилання).
2. Якщо **так** — він робить розрахунок заново і оновлює кеш.

Кешування результатів обчислень

Що це таке Техніка оптимізації, яка полягає у збереженні результату виконання функції для запобігання повторним складним розрахункам.

Механізм дії

1. **Request:** Компонент запитує дані (наприклад, суму кошика).
2. **Check:** Селектор дивиться на вхідний масив товарів.
3. **Compare:** Порівнює поточний масив з тим, що був минулого разу (за посиланням).
4. **Logic:** Якщо масив той самий, важкий цикл `reduce` не запускається.

Де застосовується на практиці Сортування великих списків, фільтрація за кількома критеріями,математичні розрахунки в фінансових додатках.

Приклад мемоїзованого селектора

Функція складається з двох частин:

Основні компоненти

1. **Input Selectors (Вхідні селектори):** Прості функції, які просто витягують дані зі стейту (наприклад, `state => state.items`).
2. **Result Function (Функція результату):** Отримує результати вхідних селекторів як аргументи і виконує над ними логіку.

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

const selectAllUsers = (state) => state.users.list;

export const selectActiveUsers = createSelector(
  [selectAllUsers], // вхідні дані
  (users) => {
    console.log('Виконую важку фільтрацію...');
    return users.filter(user => user.active);
  }
);
```

Лог з'явиться лише тоді, коли список `users.list` зміниться.

Розрахунок вартості кошика через **createSelector**

Реалізація обчислювального поля, яке автоматично оновлюється лише при зміні складу кошика.

Як це працює (Механізм) Замість того, щоб рахувати суму безпосередньо в компоненті при кожному рендері, ми виносимо логіку в мемоїзований селектор. Поки масив `items` ідентичний за посиланням, результат буде братися з кешу.

```
const selectCartItems = state => state.cart.items;
const selectTaxRate = state => state.settings.taxRate;

export const selectTotalWithTax = createSelector(
  [selectCartItems, selectTaxRate],
  (items, taxRate) => {
    const subtotal = items.reduce((acc, item) => acc + item.price, 0);
    return subtotal + (subtotal * taxRate);
  }
);
```

Комбінування кількох селекторів

Архітектурний підхід, при якому складні селектори будуються на основі вже існуючих простіших мемоїзованих селекторів.

Як це працює (Принцип дії) `createSelector` може приймати інший `createSelector` як вхідний параметр. Це створює ієрархію обчислень, де кожен рівень кешується окремо.

Основні елементи

- **Base Selectors:** Витягують сирі дані (state).
- **Derived Selectors:** Обробляють дані (фільтрація, сортування).
- **Final Selectors:** Комбінують оброблені дані для фінального відображення.

Комбінування селекторів

- **Селектор 1**
  → отримує всі товари зі state
- **Селектор 2**
  → фільтрує товари за категорією
  → використовує результат Селектора 1
- **Селектор 3**
  → обчислює сумарну вартість
  → використовує результат Селектора 2

Результат

- при зміні категорії:
  → перераховуються лише фільтрація та агрегування
- при зміні даних в інших slice (наприклад, user):
  → селектори, що не залежать від цих даних, **не виконуються повторно**

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

// 🔹 Селектор 1 — всі товари
const selectProducts = (state) => state.products.list

// 🔹 Селектор 2 — категорія
const selectCategory = (state) => state.filters.category

// 🔹 Селектор 3 — фільтрація
export const selectFilteredProducts = createSelector(
  [selectProducts, selectCategory],
  (products, category) => {
    console.log('Фільтрація виконується')
    return products.filter((p) => p.category === category)
  }
)

// 🔹 Селектор 4 — сума
export const selectTotalPrice = createSelector(
  [selectFilteredProducts],
  (filtered) => {
    console.log('Обчислення суми')
    return filtered.reduce((sum, p) => sum + p.price, 0)
  }
)
```

Фільтрація списків: чому це не можна робити в рендері

Пояснення критичної помилки продуктивності, коли логіка обробки масивів міститься всередині тіла React-компонента.

Проблеми / Обмеження Коли ви пишете `const filtered = items.filter(...)` прямо в компоненті:

1. Метод `.filter()` створює **новий масив** при кожному рендері.
2. Це змушує всі дочірні компоненти, що отримують цей масив (наприклад, через FlatList), перемальовуватися.
3. Якщо компонент має Input (пошук), кожна введена літера викликає повну фільтрацію великого списку, що призводить до "лагів" клавіатури.

Рішення Використання `createSelector`, де вхідними даними є `items` та `searchQuery`.

```
function ProductsScreen({ items, searchQuery }) {
  const filteredItems = items.filter((item) =>
    item.title.toLowerCase().includes(searchQuery.toLowerCase())
  )

  return (
    <FlatList
      data={filteredItems}
      renderItem={({ item }) => <ProductCard product={item} />}
      keyExtractor={(item) => item.id}
    />
  )
}
```

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

const selectItems = (state) => state.products.items
const selectSearchQuery = (state) => state.products.searchQuery

export const selectFilteredItems = createSelector(
  [selectItems, selectSearchQuery],
  (items, searchQuery) =>
    items.filter((item) =>
      item.title.toLowerCase().includes(searchQuery.toLowerCase())
    )
)
```

Селектори з аргументами **(Factory functions)**

Створення селекторів, які можуть приймати зовнішні параметри (наприклад, `userId` або `categoryId`) з пропсів компонента.

Як це працює Оскільки стандартний `createSelector` має кеш розміром 1, використання одного селектора для різних ID у різних компонентах призведе до постійного "перетирання" кешу. **Рішення:** Створення функції-фабрики, яка повертає новий екземпляр селектора для кожного компонента.

```
const makeSelectUserById = () => createSelector(
  [selectUsers, (state, userId) => userId],
  (users, userId) => users.find(user => user.id === userId)
);
```

Де застосовується на практиці Екрани деталей об'єктів, списки з індивідуальними розрахунками для кожного рядка.

Найкращі практики організації файлів селекторів

Стандарт структурування коду для IT-проєктів на базі Redux Toolkit.

Рекомендації

- **Розташування:** Тримайте прості селектори у файлі відповідного слайса (`features/cart/cartSlice.js`), а складні комбіновані — у окремих файлах (`features/cart/selectors.js`).
- **Іменування:** Використовуйте префікс `select...` (наприклад, `selectIsAuthenticated`, `selectActiveOrders`).

Debugging: Як перевірити, чи спрацювала мемоїзація

Методи контролю ефективності селекторів для виявлення "холостих" розрахунків.

1. **Console Logging:** Тимчасове додавання `console.log` у функцію результату (Result Function). Якщо лог з'являється при кожному діспатчі, хоча дані не змінилися — мемоїзація зламана.
2. **DevTools:** Використання вкладки "Trace" у Redux DevTools для відстеження того, який саме екшен спричинив зміну посилання.
3. **Reselect Tools:** Використання спеціалізованих бібліотек, які показують кількість "попадань" (hits) та "промахів" (misses) у кеш селектора.

Створення селекторів всередині компонента

Критична помилка архітектури, яка повністю нівелює переваги мемоїзації.

Як це працює (Проблема) Якщо ви викликаєте `createSelector` всередині тіла функціонального компонента, то при кожному рендері створюється **новий екземпляр** селектора з порожнім кешем.

```
// ТАК РОБИТИ НЕ МОЖНА:
const MyComponent = () => {
  const selectData = createSelector(...) // Селектор перестворюється щоразу
  const data = useSelector(selectData);
}
```

Рішення Селектори мають бути визначені **поза межами** компонентів (на рівні модуля), щоб їхній кеш зберігався протягом усього життєвого циклу додатка.