מדריך Redux Toolkit עם TypeScript

מבוא: Redux Toolkit vs. Context
למה בכלל Redux Toolkit?
Context API הוא פתרון מצוין לניהול state פשוט, אבל הוא מגיע עם כמה מגבלות:

- ביצועים: כל שינוי ב-Context גורם לרינדור מחדש של כל הקומפוננטות שמשתמשות בו
- קושי בניהול לוגיקה מורכבת: אין מבנה מוגדר לניהול לוגיקת State
- חוסר יכולת לדבג: אין כלים מובנים למעקב אחר שינויים

Redux Toolkit פותר את הבעיות האלו על ידי:

- ביצועים משופרים: רינדור מחדש רק של קומפוננטות שבאמת צריכות להתעדכן
- מבנה קוד מוגדר: Slices, Reducers, Actions - הכל מאורגן בצורה ברורה
- כלי פיתוח מעולים: Redux DevTools לדיבוג ומעקב אחר שינויים
- תמיכה מובנית ב-TypeScript

המבנה הבסיסי

Store

ה-Store הוא המקום המרכזי שמחזיק את כל ה-State של האפליקציה. זה כמו מחסן גדול שמכיל את כל המידע.

```JavaScript
       // store.ts
import { configureStore } from "@reduxjs/toolkit";
import counterReducer from './features/counter/counterSlice';

export const store = configureStore({
    reducer: {
                                        // כל reducer מייצג חלק שונה של ה-state

        counter: counterReducer
    }
});

                                                // נגדיר את הטיפוסים של ה-store
export type RootState = ReturnType<typeof store.getState>;
export type AppDispatch = typeof store.dispatch;
```

`ReturnType` הוא utility type של TypeScript שלוקח פונקציה ומחזיר את הטיפוס של מה שהפונקציה הזו מחזירה.
`typeof store.getState` מקבל את הטיפוס של הפונקציה `getState` מה-store
ביחד, `<ReturnType<typeof store.getState` אומר: "תיקח את הטיפוס של מה שהפונקציה getState מחזירה"
זה נותן לנו את הטיפוס המדויק של כל ה-state באפליקציה.
דוגמה:

```
JavaScript
// אם יש לנו פונקציה כזו:
function getMessage() {
    return { text: "Hello", id: 1 };
}

// <ReturnType<typeof getMessage יהיה שווה ל:
type MessageType = {
    text: string;
    id: number;
}
```

```
JavaScript
export type AppDispatch = typeof store.dispatch;
```

- `typeof store.dispatch` לוקח את הטיפוס של פונקציית ה-dispatch מה-store
- זה חשוב כי dispatch ב-Redux Toolkit תומך ב-Thunks וב-async actions
- זה נותן לנו את הטיפוס המדויק של ה-dispatch שכולל את כל היכולות האלו

Provider

ה-Provider הוא קומפוננטה שעוטפת את כל האפליקציה ומאפשרת גישה ל-Store מכל מקום:

```javascript
// main.tsx
import { Provider } from "react-redux";
import { store } from "./store/store";

createRoot(document.getElementById("root")!).render(
    <Provider store={store}>
        <App />
    </Provider>
);
```

האם כל קומפוננטה שעטופה ב-Provider יכולה לגשת ל-store, זה לא בעיית ביצועים?

כן, כל קומפוננטה יכולה לגשת ל-store, אבל אין בעיית ביצועים כי:

- רק קומפוננטות שמשתמשות ב-useSelector יתרנדרו מחדש, כלומר רק קומפוננטות שהסלקטור שלהן החזיר ערך שונה יתרנדרו.
- Redux מבצע השוואה מדויקת של ערכים שחזרו מ-useSelector
- ניתן לבצע ממואיזציה של סלקטורים מורכבים
- Redux משתמש במנגנון subscription יעיל

Slice

Slice הוא חלק מוגדר מה-State של האפליקציה. לדוגמה, בבלוג היינו יכולים לראות:

posts slice: ניהול הפוסטים
comments slice: ניהול התגובות
users slice: ניהול המשתמשים

הנה ה-Counter Slice שלנו עם TypeScript:

```javascript
// counterSlice.ts
import { createSlice } from "@reduxjs/toolkit";

interface CounterState {
    count: number;
}
```

```
const initialState: CounterState = {
    count: 0
};

export const counterSlice = createSlice({

                                                    // שם ייחודי לזיהוי ה-slice
    name: 'counter',
    initialState,
    reducers: {
                                        // הפונקציות שמשנות את ה-state
        increment: (state) => {
            state.count += 1;
                            // Redux Toolkit מאפשר לנו "לשנות" את ה-state ישירות
        },
        decrement: (state) => {
            state.count -= 1;
        }
    }
});

                                        // ייצוא ה-actions לשימוש בקומפוננטות
export const { increment, decrement } = counterSlice.actions;
                                            // ייצוא ה-reducer לשימוש ב-store
export default counterSlice.reducer;
```

למה אנחנו צריכים להגדיר initialState בנפרד? למה לא ישר בתוך createSlice?

- קריאות טובה יותר של הקוד
- שימוש חוזר באותו initial state במקומות אחרים
- בדיקות קלות יותר (testing)
- הגדרה קלה יותר של טיפוסי TypeScript

למה לא חייבים להגדיר את הטיפוס של state בפונקציות increment ו-decrement?

Redux Toolkit משתמש ב-TypeScript inference ומסיק את הטיפוסים אוטומטית:

- הטיפוס נלקח מה-initialState שהגדרנו
- createSlice יודע להסיק את הטיפוס של ה-state בכל reducer
- זה חלק מה"קסם" של Redux Toolkit שחוסך לנו כתיבת קוד בוילרפלייט

Counter Component

הקומפוננטה שמשתמשת ב-Redux:

```javascript
// Counter.tsx
import { useSelector, useDispatch } from "react-redux";
import { RootState } from "../store/store";
import { increment, decrement } from "../store/features/counter/counterSlice";

const Counter = () => {
                                        // useSelector מאפשר לנו לקרוא מה-state
    const count = useSelector((state: RootState) => state.counter.count);
                                        // useDispatch מאפשר לנו לשלוח actions
    const dispatch = useDispatch();

    return (
        <section>
            <p>{count}</p>
            <div>
                <button onClick={() => dispatch(increment())}>+</button>
                <button onClick={() => dispatch(decrement())}>-</button>
            </div>
        </section>
    );
};

export default Counter;
```

למה לא פשוט לייבא את הפונקציות?

כשאנחנו מייבאים את increment ו-decrement, אנחנו בעצם מייבאים Action Creators - פונקציות שיוצרות אובייקטים שמתארים מה אנחנו רוצים לעשות. אבל:

- ניהול State מרכזי: Redux מאפשר לנו לנהל את כל ה-State במקום אחד
- Predictable: כל שינוי עובר דרך ה-Store, מה שמאפשר לנו לעקוב אחרי שינויים
- Performance: Redux מבצע אופטימיזציות לרינדור מחדש

למה צריך dispatch? למה לא לקרוא ישירות ל-increment?

ה-dispatch חיוני כי:

- הוא מאפשר לעקוב אחרי כל השינויים ב-state
- מאפשר שימוש ב-middleware (למשל לוגינג או async actions)
- אם נסיר אותו, הפונקציה לא תעבוד

מה ההבדל בין RootState ל-CounterState?

`RootState` הוא הטיפוס של כל ה-state באפליקציה, כולל כל ה-slices
`CounterState` הוא הטיפוס של ה-state הספציפי של ה-counter slice
`RootState` יכיל את `CounterState` כחלק ממנו תחת המפתח `counter`

PayloadAction ופעולות נוספות

כעת נוסיף שתי פונקציות נוספות ל counterSlice. הראשונה תאפס את הקאונטר ושניה תאפשר למשתמש להגדיל את הטיימר בכל מספר שיבחר.
ראשית נייבא את PayloadAction בקובץ **counterSlice.ts:**

```
JavaScript
// counterSlice.ts
import { createSlice, PayloadAction } from "@reduxjs/toolkit";
```

`PayloadAction` הוא טיפוס ב-Redux Toolkit שמגדיר את המבנה של action שמכיל מידע נוסף:

```
JavaScript
// המבנה הבסיסי כאשר P הוא הטיפוס של ה-payload
PayloadAction<P>

// דוגמאות:
// payload מסוג מספר
incrementByAmount: (state, action: PayloadAction<number>)

// payload מסוג אובייקט
updateUser: (state, action: PayloadAction<{ id: number; name: string }>)
```

```
// payload מסוג מערך
addTodos: (state, action: PayloadAction<string[]>)
```

נעדכן את ה- slice:

```JavaScript
export const counterSlice = createSlice({
    name: 'counter',
    initialState,
    reducers: {
        increment: (state) => {
            state.count += 1;
        },
        decrement: (state) => {
            state.count -= 1;
        },
        reset: (state) => {
            state.count = 0;
        },
        incrementByAmount: (state, action: PayloadAction<number>) => {
            state.count += action.payload;
        }
    }
});
```

ונעדכן את ה export

```JavaScript
export const { increment, decrement, reset, incrementByAmount } =
counterSlice.actions;
```

איך Redux יודע מה לעשות עם ה-payload?

```javascript
// הגדרת ה-action
incrementByAmount: (state, action: PayloadAction<number>) => {
    state.count += action.payload;
}

// השימוש
dispatch(incrementByAmount(5))
```

כשאנחנו קוראים ל-`incrementByAmount(5)`, נוצר אובייקט action:

```javascript
{
    type: 'counter/incrementByAmount',
    payload: 5
}
```

Redux Toolkit מעביר את זה ל-reducer המתאים לפי ה-type
ה-payload מגיע כפרמטר ב-action

האם אפשר להעביר יותר מפרמטר אחד ל-action?

כן, דרך אובייקט:

```javascript
// הגדרה
updateCounter: (
    state,
    action: PayloadAction<{ amount: number; multiply?: boolean }>
) => {
    if (action.payload.multiply) {
        state.count *= action.payload.amount;
    } else {
        state.count += action.payload.amount;
    }
}

// שימוש
```

```
dispatch(updateCounter({ amount: 5, multiply: true }))
```

כעת נעדכן גם את הקומפוננטה:

```javascript
// Counter.tsx
import { useState } from "react";
import { useSelector, useDispatch } from "react-redux";
import { RootState } from "../store/store";
import {
    increment,
    decrement,
    reset,
    incrementByAmount
} from "../store/features/counter/counterSlice";

const Counter = () => {
    const [incrementAmount, setIncrementAmount] = useState<number>(0);
    const count = useSelector((state: RootState) => state.counter.count);
    const dispatch = useDispatch();

    const resetAll = () => {
        setIncrementAmount(0);
        dispatch(reset());
    };

    const handleIncrementAmount = (e: React.ChangeEvent<HTMLInputElement>) => {
                                    // התרת המספר מה-string של ה-input
        const amount = Number(e.target.value) || 0;
        setIncrementAmount(amount);
    };

    return (
        <section>
            <p>{count}</p>
            <div>
                <button onClick={() => dispatch(increment())}>+</button>
                <button onClick={() => dispatch(decrement())}>-</button>
            </div>
```

```
            <input
                type="number"
                value={incrementAmount}
                onChange={handleIncrementAmount}
            />

            <div>
                <button onClick={() =>
dispatch(incrementByAmount(incrementAmount))}>
                    Add Amount
                </button>
                <button onClick={resetAll}>Reset</button>
            </div>
        </section>
    );
};

export default Counter;
```

*****כעת ניתן להציג את התוצאה הסופית בדפדפן.**

האם כל שינוי ב-input גורם לרינדור מחדש של כל הקומפוננטה?

כן, אבל זה בסדר כי:

1. רק הקומפוננטה עצמה מתרנדרת
2. הערך ב-Redux לא משתנה עד ללחיצה על הכפתור
3. קומפוננטות אחרות לא מושפעות
4. React מטפל ביעילות בשינויי input

Best Practice

אציג פרוייקט לדוגמה, בלוג עם פוסטים, אשר ממחיש כיצד צריך להשתמש נכון ב redux-toolkit.

מבנה תיקיות מומלץ:

```
JavaScript
src/
  ├── store/
  │   ├── index.ts         // הגדרת ה-store
  │   ├── hooks.ts         // custom hooks
  │   └── features/
  │       ├── users/
  │       │   ├── usersSlice.ts
  │       │   ├── usersSelectors.ts
  │       │   └── usersThunks.ts
  │       └── posts/
  │           ├── postsSlice.ts
  │           ├── postsSelectors.ts
  │           └── postsThunks.ts
```

דברים להימנע מהם:

- יצירת slice לכל קומפוננטה
- ערבוב לוגיקת UI עם לוגיקת state
- שמירת מידע זמני ב-Redux

מתי להשתמש ב-Redux:

```
JavaScript
// מידע גלובלי - למשל, מידע על המשתמש
interface UserState {
  id: string;
  name: string;
  preferences: UserPreferences;
}

// מידע משותף - למשל, נתונים שמשמשים הרבה קומפוננטות
interface ProductsState {
  items: Product[];
  categories: Category[];
  filters: Filter[];
}

// מידע שצריך להישאר בין מעברי דפים
```

```
interface CartState {
  items: CartItem[];
  total: number;
}
```

❌ מתי **לא** להשתמש ב redux:

```JavaScript
// מצב טופס
const [formData, setFormData] = useState<FormData>({});

// מצב UI
const [isOpen, setIsOpen] = useState(false);

// מידע זמני
const [searchQuery, setSearchQuery] = useState('');
```

תכנון נכון של Slice:

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

interface ProductsState {
  // מידע בסיסי
  items: Record<string, Product>;
  loading: boolean;
  error: string | null;

  // מטה-דאטה
  lastFetched: number | null;

  // מצב UI קריטי
  selectedProductId: string | null;
}

const productsSlice = createSlice({
  name: 'products',
```

```
  initialState,
  reducers: {
                                                    // פעולות אטומיות וממוקדות
    setSelectedProduct: (state, action: PayloadAction<string>) => {
      state.selectedProductId = action.payload;
    }
  }
});
```

דברים להימנע מהם בתכנון Slice:

```JavaScript
                                                    // ❌ לא לערבב מידע לא קשור
interface BadSliceDesign {
  products: Product[];
                                                    // צריך להיות ב-slice נפרד
  userPreferences: UserPreferences;
                                    // רוב מצבי ה-UI צריכים להיות ב-local state
uiState: {
    isModalOpen: boolean;
    activeTab: string;
  }
}

                                                    // ❌ לא ליצור פעולות מורכבות מדי
const badReducer = {
  doManyThings: (state, action) => {
                                                    // פעולה שעושה יותר מדי דברים
    updateProducts();
    updateUI();
    fetchMoreData();
  }
}
```

שימוש נכון ב Typescript:

```javascript
// store/types.ts
export interface RootState {
  products: ProductsState;
  cart: CartState;
  user: UserState;
}

// hooks.ts
import { TypedUseSelectorHook, useDispatch, useSelector } from 'react-redux';
import { AppDispatch, RootState } from './store';

export const useAppDispatch = () => useDispatch<AppDispatch>();
export const useAppSelector: TypedUseSelectorHook<RootState> = useSelector;

// שימוש בקומפוננטה
const ProductList = () => {
  const dispatch = useAppDispatch();
  const products = useAppSelector(state => state.products.items);
}
```

מטלות

🔵 משימה 1 - Todo Toggle

רמת קושי: קלה

```javascript
// ליצור אפליקציית Todo פשוטה עם:
- רשימת משימות קבועה מראש (3-4 משימות)
- כל משימה היא אובייקט עם id, text, completed
- אפשרות לסמן משימה כבוצעה/לא בוצעה
- הצגת מספר המשימות שהושלמו מתוך הסה"כ

// מבנה ה-state:
interface TodoState {
    tasks: {
        id: number;
```

```
        text: string;
        completed: boolean;
    }[];
}
```

🔵 משימה 2 - Shopping Cart

רמת קושי: בינונית-קלה

```JavaScript
// ליצור עגלת קניות בסיסית:
- רשימה קבועה של 4-5 מוצרים עם מחיר
- כפתורי + ו- - לכל מוצר לשינוי הכמות
- הצגת סה"כ מחיר העגלה
- כפתור לריקון העגלה

// מבנה ה-state:
interface CartState {
    items: {
        id: number;
        name: string;
        price: number;
        quantity: number;
    }[];
    total: number;
}
```

🔵 משימה 3 - Note Taking App

רמת קושי: בינונית-גבוהה

```JavaScript
// ליצור אפליקציית פתקים עם ראוטינג:
```

```
- דף ראשי עם רשימת כל הפתקים
- דף יצירת פתק חדש
- דף עריכת פתק קיים
- כל פתק מכיל:
  * כותרת
  * תוכן
  * תאריך יצירה
  * קטגוריה (work/personal/shopping)

// נדרש:
- שימוש ב-React Router
- CRUD operations מלא
- אפשרות לסינון לפי קטגוריה
- הצגת מספר הפתקים בכל קטגוריה

// מבנה ה-state:
interface NotesState {
    notes: {
        id: number;
        title: string;
        content: string;
        category: 'work' | 'personal' | 'shopping';
        createdAt: string;
    }[];
    activeCategory: 'all' | 'work' | 'personal' | 'shopping';
}

// נדרש ראוטינג:
notes/ - רשימת הפתקים
notes/new/ - יצירת פתק חדש
notes/:id/ - עריכת פתק קיים
```

💡 הערות חשובות למשימות:

1. לכל המשימות יש להשתמש ב:
   - Redux Toolkit
   - TypeScript
   - Proper folder structure (features/slices)
   - פונקציונליות בסיסית של loading states
2. דגשים לבדיקה:
   - שימוש נכון ב-PayloadAction

- ארגון נכון של ה-store
- שימוש נכון ב-TypeScript
- קומפוננטות נקיות וקריאות

⏱️ הערכת זמנים:

- משימה 1: ~45 דקות
- משימה 2: ~60 דקות
- משימה 3: ~75 דקות