React i Redux

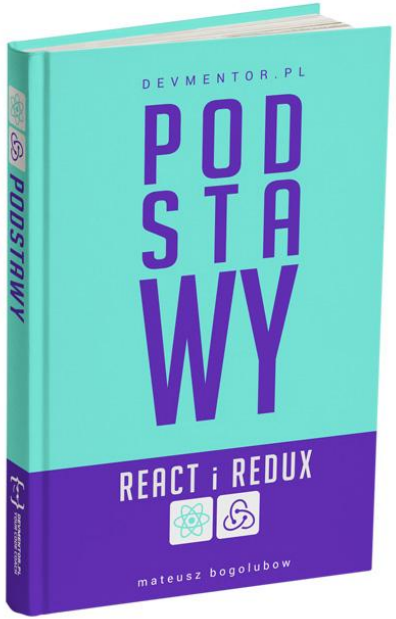
Podstawy

Redux to biblioteka, której zadaniem jest ułatwienie zarządzania przepływem danych w aplikacji.

Jest to najczęściej używane rozwiązanie w rozbudowanych aplikacjach korzystających z Reacta oraz architektury Flux.

Dokładne poznanie Reduxa znacznie ułatwi Ci budowanie aplikacji opartych o stan i zarządzanie tym stanem.

Ponieważ jest to bardzo popularne rozwiązanie, jego znajomość może zapewnić Ci angaż do nowego projektu.



Jak korzystać z materiałów?

Przedstawiony materiał to jedna, przemyślana, **spójna całość**. Dlatego nie zalecam skakania między stronami, nawet jeśli uważasz, że dany temat jest dla Ciebie jasny.

Zawsze znajdzie się jedna, drobna informacja, która może zmienić Twoją perspektywę, lub której zabraknie w najważniejszym momencie, tj. na rozmowie rekrutacyjnej.

Czytając daną stronę, **zawsze przepisuj przedstawiony kod** (nie kopiuj!) i testuj go w różnych konfiguracjach.

Nie bój się czegoś zmienić, dodać lub odjąć. Zamiast zastanawiać się „co by było gdyby”, po prostu to sprawdź!

Programowanie ma tą piękną cechę, że zawsze możemy cofnąć (`ctrl+z`) nasze zmiany bez większego wysiłku!

W materiałach będą pojawiać się odnośniki do **tematów pokrewnych lub rozszerzających** daną informację.

Zapoznaj się z nimi od razu. Następnie wróć do nich jeszcze raz po zakończeniu całego rozdziału.

Jeśli masz ograniczoną ilość czasu, to nie staraj się pogłębiać wiedzy z dodatkowych źródeł. Wystarczy, że się z nimi zapoznasz i możesz iść dalej.

Pamiętaj, że **temat główny** omawiany w materiale **jest niezbędny**. Tematy rozszerzające zagadnienie to dodatek, który warto znać, jeśli ma się na to czas.

Gdy czegoś nie rozumiesz, staraj się znaleźć dodatkowe informacje na ten temat. Czasami też warto wrócić do problematycznych zagadnień po przerwie.

Wyszukiwanie informacji jest częścią pracy programisty, dlatego już teraz zacznij rozwijać tę umiejętność. Samodzielne rozwiązywanie problemów jest **bardzo doceniane** w świecie IT.

Zadania do wykonania

Pod poniższym adresem znajdziesz zadania,
które obejmują materiał prezentowany w niniejszym ebooku.

https://github.com/devmentor-pl/practice-react-redux-basics

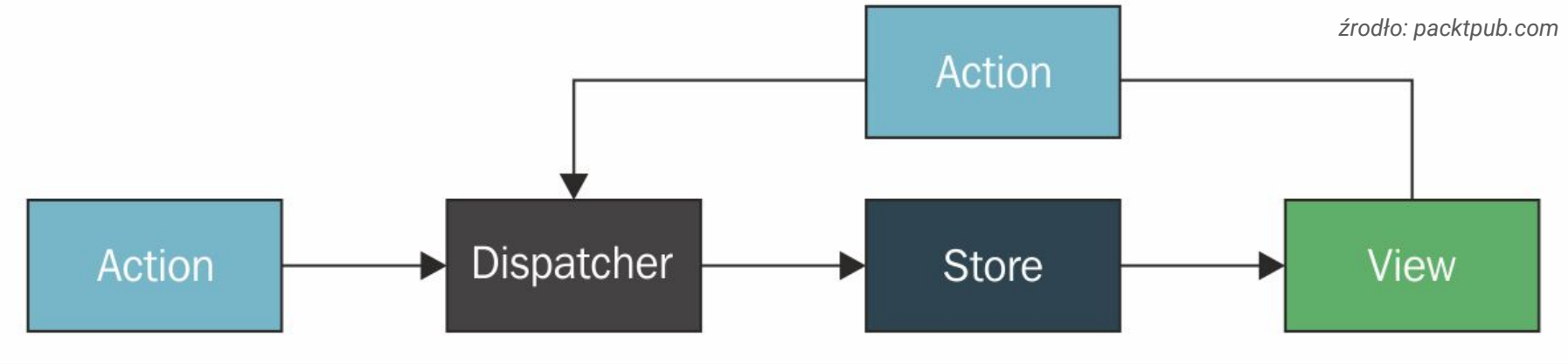
01. Flux

Flux to architektura, która ułatwia zarządzanie stanem aplikacji – niekoniecznie tylko tym znanym z Reacta.

Redux jest implementacją Fluxa, czyli Flux to koncepcja, a Redux konkretne rozwiązanie.

Flux opiera swoją architekturę o 4 elementy:

- Dyspozytor (Dispatcher),
- Akcja (Action),
- Magazyn (Store),
- Widok (View).



źrodło: packtpub.com

Flux pozwolił zmienić koncepcję budowania aplikacji opartą o wzorzec MVC na jednokierunkowy przepływ danych, który miał ułatwić zarządzanie przepływem informacji.

Całe rozwiązanie zostało zaproponowane i wdrożone przez programistów Facebooka, w którym MVC się nie sprawdzało.

Zanim przejdziemy do konkretów, zachęcam Cię do zapoznania się z krótkim filmem [~ 4 min] na YouTube, wprowadzającym w świat Fluxa.

Jeśli chcesz odbyć podróż do 2014 r. i zobaczyć, jak deweloperzy Facebooka prezentują swoje rozwiązanie, to również zapraszam do oglądania [~ 45 min].

Proponuję do powyższego filmu wrócić, gdy przerobisz cały ten materiał. Będzie Ci dużo łatwiej odnieść się do prezentowanych koncepcji.

#01 Części składowe

01. Flux

- **Dyspozytor (Dispacher)**
  Koordynuje emisję Akcji do magazynu oraz zarządza procedurami ich obsługi.

- **Akcja (Action)**
  Zdarzenie w aplikacji, które ma wpływ na jej stan. Może to być np. kliknięcie w przycisk, który zmienia wybraną opcję. Każda akcja ma swój typ, który ją jednoznacznie identyfikuje i tzw. ładunek, czyli dane związane z tą Akcją (opcjonalne).

- **Magazyn (Store)**
  Miejsce, w którym są przechowywane stany. Tylko tutaj mogą być one modyfikowane. To magazyn emituje informacje o zmianie.

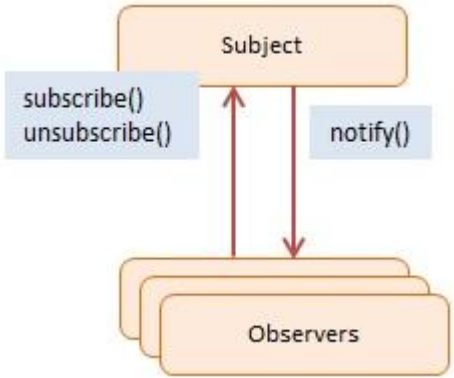
- **Widok (View)**
  Widok jest tworzony na podstawie stanu aplikacji, czyli tego, co zawiera magazyn. Ponieważ Widok jest obserwatorem zmian magazynu, to wie, kiedy ma zostać zaktualizowany.

#02 Wzorzec obserwator

01. Flux

Wspomnieliśmy wcześniej, że widok jest obserwatorem, a magazyn emituje informacje o zmianie stanu.

Tego typu zachowanie implementuje wzorzec obserwator (*Observer Pattern*), o którym teraz sobie powiemy.



źródło: dofactory.com

Najprostsza implementacja tego wzorca polega na określeniu elementu (Subject), który posiada obserwatorów (Observers).

Za pomocą metody .subscribe() mogą być dodawani kolejni obserwatorzy. Natomiast .unsubscribe() pozwala ich usuwać.

Za każdym razem, kiedy Subject chce poinformować obserwatorów o zmianie, wywołuje metodę .notify().

Utworzyliśmy klasę Subject, która posiada konstruktor. To w nim definiujemy zmienną przechowującą obserwatorów.

Na początku to pusta tablica. Za każdym wywołaniem metody .subscribe() dodajemy do wspomnianej tablicy funkcję przekazaną w parametrze.

W przypadku wywołania metody .notify() wywołujemy wszystkie przekazane do tablicy funkcje.

```
class Subject {
    constructor() {
        this.observersList = [];
    }
    subscribe(callback) {
        this.observersList.push(
            callback
        ); // dodaj kolejną funkcję do wywołania
        // przy emitowaniu informacji
    }
    notify(data) {
        this.observersList.forEach(
            callback => callback( data )
        ); // powiadom wszystkich obserwatorów
        // tj. wywołaj wszystkie callbacki
        // oraz przekaż odpowiednie dane w [data]
    }
}
```

Teraz wykorzystujemy naszą implementację wzorca obserwator.

Dodajemy dwóch obserwatorów, którzy mają być powiadomieni (przez wywołanie przekazanych funkcji) o zmianach w Subject.

Przedstawiona implementacja została możliwie najbardziej uproszczona. Jeśli chcesz przyjrzeć się bardziej rozbudowanej implementacji, to zapraszam na dofactory.com oraz addyosmani.com.

```
const subject = new Subject();

const observer1 = data => console.log(`o1 -> ${data}`);
subject.subscribe(observer1);
// dodaję pierwszego obserwatora
// przekazana funkcja powinna posiadać parametr
// to dzięki niemu Subject wysyła dane

const observer2 = d => console.log(`o2 -> ${d}`);
subject.subscribe(observer2);
// przekazywana funkcja może być również metodą obiektu
// przedstawione rozwiązanie jest możliwie najprostsze

setTimeout(() => {
    subject.notify('new data');
    // rozgłaszanie nowych danych po 2 s
}, 2000);
```

02. Redux

Zanim zabierzemy się za Reduxa, powinniśmy przygotować sobie konfigurację dla Reacta.

Najwygodniej będzie skorzystać z gotowego rozwiązania, tzw. boilerplate'a.

My wykorzystamy create-react-app, który posiada już gotową konfigurację i podstawową strukturę naszej aplikacji.

Uruchamiamy instalację za pomocą:

```
npx create-react-app@5 rr-app
```

Po zakończeniu instalacji przechodzimy do utworzonego katalogu:

```
cd rr-app
```

Następnie uruchamiamy naszą konfigurację:

```
npm start
```

Kolejnym krokiem będzie zainstalowanie niezbędnych paczek, które pozwolą nam wykorzystać w naszym projekcie Reduxa.

Otwieramy dodatkowy terminal (lub wiersz poleceń) i instalujemy od razu obie niezbędne biblioteki:

```
npm i redux@4 react-redux@8
```

Redux jest biblioteką, którą możemy używać, nie tylko korzystając z Reacta.

Dlatego instalujemy ogólną implementację (redux) i potem pobieramy paczkę (react-redux), która pozwoli nam połączyć Reduxa z Reactem.

Przed lekturą następnych stron zachęcam Cię do obejrzenia kolejnych filmów z YouTube.

Tym razem zacznij od rozwinięcia ostatniego tematu Fluxa, czyli Dlaczego akurat Redux? [~ 6 min].

Następnie zobacz [~ 11 min], na przykładzie gry tetris, czym może być stan i jak wygląda narzędzie Redux DevTools, o którym sobie później powiemy.

Po zapoznaniu się z tymi materiałami jesteś już gotowy na przejście do konkretów!

#01 Store

02. Redux

Wiesz już z koncepcji Fluxa, że Store odpowiada za przechowywanie stanu aplikacji.

Utworzymy sobie taki Magazyn w pliku `./src/index.js` przy pomocy metody `createStore()`. Metodę tę importujemy z paczki `redux`. Jako argument musimy do niej przekazać funkcję.

To wszystko piszemy poniżej dotychczasowych importów.

```
// imports
import {createStore} from 'redux';

const store = createStore(() => {
    return 'state';
});

console.log(store.getState());
// wyświetla stan magazynu,
// w konsoli zobaczymy 'state',
// czyli to, co zwraca funkcja
// przekazana przez parametr

// root
```

Dane, które są przechowywane w naszym magazynie, moglibyśmy przekazać przez props do komponentu `<App>`, a stamtąd – do komponentów potomnych – również poprzez props.

Nie będzie to jednak ani najlepsze, ani najwygodniejsze rozwiązanie. W tym materiale dowiesz się, jak zrobić to lepiej.

Uwaga: *ponieważ create-react-app domyślnie korzysta ze `<React.StrictMode />`, w wersji 18 Reacta w konsoli w trybie developerskim możesz zobaczyć „podwójny render” – przy stosowaniu hooka useEffect() lub componentDidMount().*

```
// imports
import {createStore} from 'redux';

const store = createStore(() => {
    return 'state';
});

console.log(store.getState());

const root =
ReactDOM.createRoot(document.getElementById('root'));
root.render(
    <React.StrictMode>
        <App store={store} />
    </React.StrictMode>
);
```

Jeśli Twoje IDE sugeruje (przez ~~przekreślenie~~), że to rozwiązanie jest przestarzałe (ang. *deprecated*) to na razie zignoruj tę informację.

Pod koniec rozdziału dojdziemy do rozwiązania tego problemu.

Na razie weź pod uwagę, że to, o czym teraz mówimy, jest ciągle wykorzystywane i dlatego również powinieneś to znać.

Rozwiązanie z Toolkit to coś nowego, o czym będziemy mówić później.

```
// imports
import {createStore} from 'redux';

const store = createStore(() => {
    return 'state';
});

console.log(store.getState());

const root =
ReactDOM.createRoot(document.getElementById('root'));
root.render(
    <React.StrictMode>
        <App store={store} />
    </React.StrictMode>
);
```

#02 Actions

02. Redux

Akcje to czynności, które decydują o zmianie stanu naszego magazynu. Każda akcja musi posiadać informację o swoim typie, aby można było ją zidentyfikować.

Opcjonalnie może nieść ze sobą dodatkowe informacje (tzw. ładunek, ang. *payload*), które zostaną wykorzystane do zmiany stanu naszego magazynu.

Akcja może być obiektem lub funkcją zwracającą obiekt czy inną funkcję. W naszym przypadku mamy funkcję zwracającą obiekt.

```
// imports

import {createStore} from 'redux';

const increaseCounterAction = () => {
    return {
        type: 'increaseCounter',
        payload: {
            step: 2,
        }
    }
}

// store
```

W przykładzie obok mamy dwie akcje.

Pierwsza nie przyjmuje żadnych danych z zewnątrz. Ma ona zwiększyć wartość licznika o 2 (step: 2).

Druga pozwala ustawić wartość dla licznika. Wartość tę otrzymuje poprzez parametr value.

Niestety te akcje nic na razie nie robią. Potrzebują pośrednika, który będzie śledził Akcje i wykonywał niezbędne działania w Magazynie.

```
// imports
import {createStore} from 'redux';

const increaseCounterAction = () => {
    return {
        type: 'increaseCounter',
        payload: { step: 2 },
    }
}

const setCounterAction = value => {
    return {
        type: 'setCounter',
        payload: { value: value },
    }
}
```

#03 Reducers

02. Redux

Pośrednik, o którym wspominałem w przypadku Reduxa, nosi nazwę Reducer (Reduktor). To on określa, co ma być wykonane po uruchomieniu danej akcji.

Choć jeszcze o tym nie wiesz, to już z niego skorzystałeś! To funkcja, którą przekazałeś podczas tworzenia *Store'a*.

Trochę ją teraz zmodyfikujemy. Wykorzystamy fakt, że przyjmuje ona dwa parametry – obecny stan magazynu, tj. `state`, oraz informacje o uruchomionej akcji, tj. `action`.

```
// actions
const initState = { counter: 0 };
const reducer = (state = initState, action) => {
    switch(action.type) {
        case 'increaseCounter':
            const {step} = action.payload;
            return {
                counter: state.counter + step,
            }
        // ...
        default:
            return state;
    }
}

const store = createStore(reducer);
```

Teraz nasz Reducer, w zależności od akcji, wykonuje różne czynności, ale zawsze zwraca aktualny stan Magazynu!

W przypadku, gdy akcja jest typu `increaseCounter`, to wartość obecnego licznika zwiększana jest o wartość przesłaną wraz z akcją w `.payload`.

Analogicznie będzie z drugą naszą akcją...

```
// actions
const initState = { counter: 0 };
const reducer = (state = initState, action) => {
    switch(action.type) {
        case 'increaseCounter':
            const {step} = action.payload;
            return {
                counter: state.counter + step,
            }
        // ...
        default:
            return state;
    }
}

const store = createStore(reducer);
```

W tym przypadku przypisujemy nową wartość do właściwości .counter.

Należy również zwrócić uwagę na fakt, że początkowo .counter jest równy zero. Jest to spowodowane przypisaniem wartości domyślnej z initState do pierwszego parametru Reducera o nazwie state.

Czas sprawdzić, czy nasze akcje działają.

```
// actions

const initState = { counter: 0 };
const reducer = (state = initState, action) => {
    switch(action.type) {
        // ...
        case 'setCounter':
            return {
                counter: action.payload.value,
            }
        default:
            return state;
    }
}

const store = createStore(reducer);
```

Wystarczy na naszym Magazynie wywołać metodę `.dispatch()`, do której przekażemy akcję do wykonania.

W przykładzie obok wywołujemy najpierw ustawienie licznika na wartość 5, następnie zwiększamy wartość o 2.

Po tych operacjach stan naszego magazynu dla właściwości `.counter` wynosi 7.

```
// imports
const increaseCounterAction = () => {
    return {
        type: 'increaseCounter',
        payload: { step: 2 },
    }
}
const setCounterAction = value => {
    return {
        type: 'setCounter',
        payload: { value: value },
    }
}
// reducer
const store = createStore(reducer);
store.dispatch(setCounterAction(5));
store.dispatch(increaseCounterAction());
```

Wróćmy jeszcze do naszego stanu.
W poprzednim przykładzie mamy tylko jedną właściwość, więc nie musimy się przejmować kopiowaniem pozostałych wartości.

Jeśli jednak mielibyśmy więcej elementów w `state`, to powinniśmy użyć Object.assign() z ES2015, aby utworzyć kopię pozostałych wartości.

Pamiętaj, że `Object.assign()` tworzy kopię płytką, tj. kopiuje tylko pierwsze zagnieżdżenie.

```
// actions
const initState = { counter: 0, str: 'txt' };
const reducer = (state = initState, action) => {
    switch(action.type) {
        case 'increaseCounter':
            const {step} = action.payload;
            return Object.assign({}, state, {
                counter: state.counter + step,
            });
        // ...
        default:
            return state;
    }
}

const store = createStore(reducer);
```

Możemy wykorzystać również operator rozproszenia (...) dla obiektów, który pochodzi z ES2018 i działa podobnie jak Object.assign().

Pamiętaj o użyciu transpilatora Babel, aby nie było problemów z obsługą tego rozwiązania przez starsze przeglądarki.

Używając *create-react-app* konfiguracja jest domyślnie przygotowana.

Dla utrwalenia polecam obejrzeć na YouTube nagranie dotyczące tworzenia Magazynu [~ 5 min] oraz wykonywania Akcji [~ 7 min].

```
// actions
const initState = { counter: 0, str: 'txt' };
const reducer = (state = initState, action) => {
    switch(action.type) {
        case 'increaseCounter':
            const {step} = action.payload;
            return { ...state,
                counter: state.counter + step,
            };
        // ...
        default:
            return state;
    }
}

const store = createStore(reducer);
```

#04 Rozdzielenie kodu na pliki

02. Redux

Mamy już przygotowaną część naszego kodu odpowiedzialną za implementację Reduxa. Teraz będziemy chcieli całość połączyć z Reactem.

Zanim to zrobimy, przeniesiemy część kodu do oddzielnych plików.

Zaczniemy od Reducera, który umieścimy w pliku `./src/reducers/index.js`.

```
const initState = { counter: 0 };
const reducer = (state = initState, action) => {
    switch(action.type) {
        case 'increaseCounter':
            const {step} = action.payload;
            return {
                counter: state.counter + step,
            }
        case 'setCounter':
            return {
                counter: action.payload.value,
            }
        default:
            return state;
    }
}
export default reducer;
```

Importujemy nasz Reducer do pliku `./src/index.js` i przekazujemy do `.createStore()`.

Zwróć uwagę na zmianę nazwy w imporcie. O łączeniu magazynów powiemy sobie w następnym materiale.

```
import reducers from './reducers';
// zamierzona zmiana nazwy -
// w późniejszych krokach połączymy
// wiele reducerów

// ...

const store = createStore(reducers);
```

Teraz przenosimy Actions do pliku `./src/actions/counter.js`

Pamiętaj o eksportowaniu tych funkcji!

```
export const increaseCounterAction = () => {
    return {
        type: 'increaseCounter',
        payload: { step: 2 },
    }
}

export const setCounterAction = value => {
    return {
        type: 'setCounter',
        payload: { value: value },
    }
}
```

Dzięki temu zabiegowi nasz plik `./src/index.js` bardzo się odchudził.

Znów odsyłam Cię do filmu [~ 12 min] z kursu Artura Chmaro, gdzie autor proponuje trochę inną strukturę plików.

```
// imports

import {createStore} from 'redux';
import reducers from './reducers';

const store = createStore(reducers);

const root =
ReactDOM.createRoot(document.getElementById('root'));
root.render(
    <React.StrictMode>
        <App store={store} />
    </React.StrictMode>
);
```

#05 connect()

02. Redux

Nadszedł czas na zmodyfikowanie naszego głównego komponentu.

Na początek będziemy chcieli wyświetlić zawartość licznika.

Oczywiście powinna być ona równa 0, ponieważ taki jest stan początkowy dla naszej właściwości `.counter` w Magazynie.

```
import React from 'react';
import './App.css';

function App(props) {
  const { counter } = props.store.getState();
  return (
    <section>{ counter }</section>
  );
}

export default App;
```

Teraz zobaczymy, jak Redux działa z komponentami klasowymi.

Zamieniłem komponent App na klasę oraz dodałem <button>, na którym mamy event o nazwie *click*.

Chcemy, aby po kliknięciu w przycisk zwiększyła się wartość stanu dla licznika.

Tu z pomocą przyjdzie nam wcześniej zdefiniowana Akcja. Jednak to nie wystarczy. Musimy jeszcze połączyć Reduxa z Reactem, na co pozwoli nam connect()!

```
import React from 'react';
import './App.css';
class App extends React.Component {
    handleIncrease = event => {
        console.log('increase');
    }
    render() {
        const {counter} = this.props.store.getState();
        return (
            <section>{ counter }
                <button
                    onClick={ this.handleIncrease }
                >increase</button>
            </section>
        );
    }
}
export default App;
```

Importujemy metodę connect() oraz odpowiednią Akcję.

Następnie tworzymy obiekt mapActionToProps, który łączy nam props onIncrease z nazwą zmiennej, pod którą kryje się nasza Akcja.

Dzięki connect() mamy dostęp do Akcji przez props.

Musimy jeszcze zdefiniować, jakie dane z magazynu mają być dostępne przez props w naszym komponencie.

```
// ...
import {connect} from 'react-redux';
import {increaseCounterAction} from './actions/counter';

class App extends React.Component {
    handleIncrease = event => {
        console.log('increase');
        console.log(this.props.onIncrease());
    }
    render() {
        // ...
    }
}
const mapActionToProps = {
        onIncrease: increaseCounterAction,
}
export default connect(null, mapActionToProps)(App);
```

Tworzymy funkcję mapStateToProps, która zwraca obiekt.

Ten obiekt to dane, jakie będą przekazane do naszego komponentu <App> przez props.

Metodę mapStateToProps przekazujemy jako pierwszy parametr funkcji connect(), który wcześniej był ustawiony na null (co powodowało brak aktualizacji naszego komponentu po zmianie stanu z Magazynu).

```
// ...
const mapStateToProps = (state, props) => {
    return {
        counter: state.counter,
    } // wszystkie zdefiniowane tutaj właściwości obiektu
    // będą propsami dla komponentu <App />
}
const mapActionToProps = {
    onIncrease: increaseCounterAction,
} // wszystkie zdefiniowane tutaj właściwości obiektu
// będą propsami dla komponentu <App />

export default connect(
    mapStateToProps, mapActionToProps
)(App);
```

Wywołanie `connect(mapStateToProps, mapActionToProps)(App)` zwraca nowy komponent `<App>` z odpowiednimi propsami.

Tego typu komponent jest nazywany kontenerem (ang. *container*), o czym powiemy sobie w następnym materiale.

```
// ...
const mapStateToProps = (state, props) => {
    return {
        counter: state.counter,
    } // wszystkie zdefiniowane tutaj właściwości obiektu
    // będą propsami dla komponentu <App />
}
const mapActionToProps = {
    onIncrease: increaseCounterAction,
} // wszystkie zdefiniowane tutaj właściwości obiektu
// będą propsami dla komponentu <App />

export default connect(
    mapStateToProps, mapActionToProps
)(App);
```

#06 Provider

02. Redux

Tak jak obiecałem, wracamy jeszcze na chwilę do naszego pliku ./src/index.js.

Importujemy w nim komponent <Provider>.

Dzięki niemu nie będziemy musieli przekazywać naszego Store'a przez props do każdego zagnieżdżenia.

Zrobi to za nas automatycznie <Provider> wszędzie tam, gdzie wykorzystamy metodę connect().

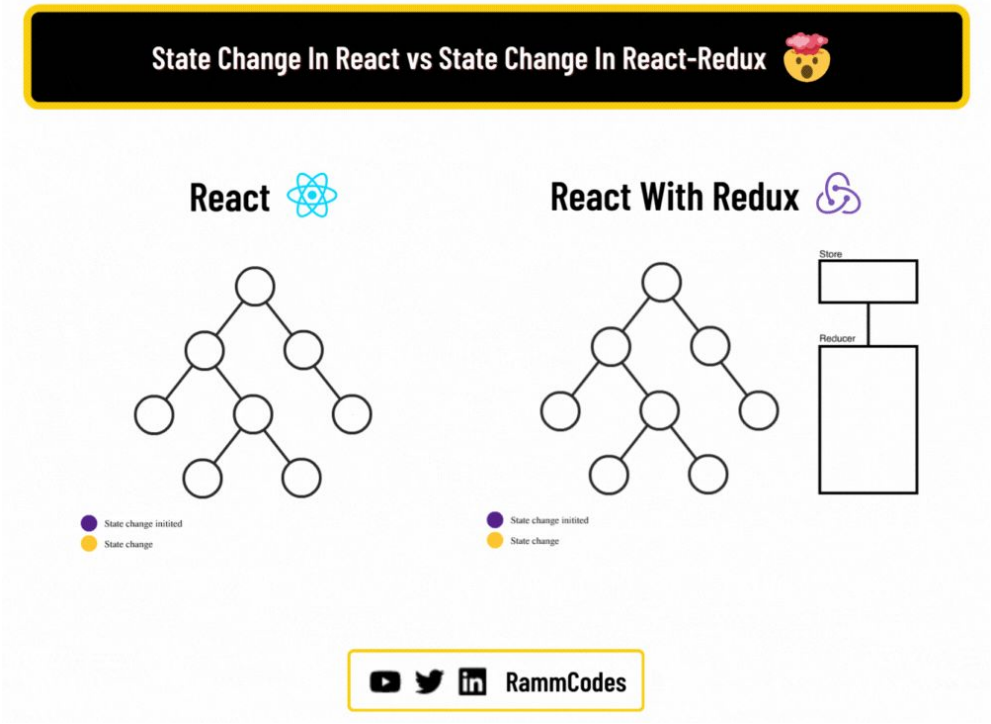
```
import React from 'react';
import ReactDOM from 'react-dom/client';

import './index.css';
import App from './App';
import * as serviceWorker from './serviceWorker';

import {createStore} from 'redux';
import {Provider} from 'react-redux';
import reducers from './reducers';

const store = createStore(reducers);

const root =
ReactDOM.createRoot(document.getElementById('root'));
root.render(
    <React.StrictMode>
        <Provider store={ store }>
            <App />
        </Provider>
    </React.StrictMode>
);
```

Skoro nasz Provider już działa, a w pliku `./src/App.js` mamy zrobione mapowanie ze state do props dzięki `connect()`, to wykorzystujemy właściwość `.counter`.

Nie potrzebujemy już jawnie odpytywać metody `.getState()` ani wyświetlać informacji o zwracanym obiekcie przez `.onIncrease()`.

Cały kod znajdziesz w repozytorium o nazwie react-redux-helloworld, a jeśli potrzebujesz uzupełnienia w formie wideo, to oto ono [~7 min].

```
// ...
class App extends React.Component {
    handleIncrease = event => {
        console.log('increase');
        this.props.onIncrease();
    }
    render() {
        const { counter } = this.props;
        return (
            <section>
                { counter }
                <button onClick={ this.handleIncrease }>
                    increase
                </button>
            </section>
        );
    }
}
// ...
```



Zobacz, jak sprawnie działa zmiana stanu dzięki zastosowaniu Reduxa.
Odtwórz GIF: devmentor.pl/react-redux-rammcodes-gif

#07 Redux DevTools

02. Redux

Bardzo przydatnym narzędziem jest Redux DevTools.

Rozszerzenie to pozwala nam zajrzeć w głąb naszego magazynu i sprawdzić jego obecny stan. Ułatwia to debugowanie.

Możemy również prześledzić zmiany (Akcje), jakie miały miejsce w czasie działania naszej aplikacji.

```
// ...

const store = createStore(reducers);

const root =
ReactDOM.createRoot(document.getElementById('root'));
root.render(
    <React.StrictMode>
        <Provider store={ store }>
            <App />
        </Provider>
    </React.StrictMode>
);
```

Aby skorzystać z dobrodziejstw tego narzędzia, najpierw musimy zainstalować je w przeglądarce.

Następnie dokonujemy zmian w naszym kodzie – o czym możesz przeczytać w dokumentacji.

Do funkcji createStore() dodajemy drugi parametr, który pozwala nawiązać połączenie między Store'em a naszą wtyczką w przeglądarce.

Więcej na temat korzystania z Redux DevTools znajdziesz na YouTube [~6 min].

```
// ...

const store = createStore(
    reducers,
    window.__REDUX_DEVTOOLS_EXTENSION__
    && window.__REDUX_DEVTOOLS_EXTENSION__()
);

const root =
ReactDOM.createRoot(document.getElementById('root'));
root.render(
    <React.StrictMode>
        <Provider store={ store }>
            <App />
        </Provider>
    </React.StrictMode>
);
```

#08 Hooks

02. Redux

W przypadku używania hooków nasz kod źródłowy będzie się prezentował trochę inaczej.

Stworzymy teraz komponent `<Form>`, który w `<input>` będzie przechowywał informacje o aktualnej wartości dla `.counter` z naszego Store'a.

W przypadku zmiany tej wartości przez użytkownika (event o nazwie *submit*) będziemy modyfikować zawartość magazynu.

```
// ...
import Form from './Form'

class App extends React.Component {
    handleIncrease = event => {
        console.log('increase');
        this.props.onIncrease();
    }
    render() {
        const { counter } = this.props;
        return (
            <section>{ counter }
                <button onClick={ this.handleIncrease }>
                    increase
                </button>
                <Form />
            </section>
        );
    }
}
// ...
```

W pliku ./src/Form.js tworzymy podstawową strukturę dla tego komponentu.

Element <input>, który będzie przechowywał informacje o stanie magazynu, będzie kontrolowany, tj. będzie zależał od state.

```
// ./src/Form.js
import React from 'react';

const Form = () => {
    return (
        <form>
            <div>
                <input />
                <input type="submit" value="zapisz" />
            </div>
        </form>
    );
}

export default Form;
```

Tworzymy stan o nazwie `inputCounter`, który będzie przechowywał informacje o wartości `.counter` w naszym Magazynie.

```
// ./src/Form.js
// ...
const Form = () => {
    const [inputCounter, setInputCounter] = useState(0);

    return (
        <form>
            <div>
                <input
                    onChange={ ({target}) =>
                        setInputCounter(target.value)
                    }
                    value={ inputCounter }
                />
                <input type="submit" value="zapisz" />
            </div>
        </form>
    );
}
```

Teraz musimy pobrać wartość z magazynu. Aby to zrobić, musimy wykorzystać hook o nazwie `useSelector()`, który importujemy z `react-redux`.

Przyjmuje on funkcję, która przyjmuje przez parametr zawartość Store'a i zwraca potrzebne nam dane.

Do `useState()` przekazujemy wartość początkową dla `inputCounter`.

```
// ./src/Form.js
import React, { useState } from 'react';
import { useSelector } from 'react-redux';

const Form = () => {
    const counter = useSelector(state => state.counter);
    const [inputCounter, setInputCounter]
                                = useState(counter);

    return (
        <form>
            {/* ... */}
        </form>
    );
}

export default Form;
```

Zwróć uwagę, że zawartość <input> nie zmienia się po aktualizacji naszego stanu w Magazynie.

Dzieje się tak, ponieważ o aktualizacji zawartości inputa decyduje inputCounter.

Teraz będę chciał zaktualizować nasz Magazyn po wysłaniu danych przez użytkownika.

```
// ./src/Form.js
import React, { useState } from 'react';
import { useSelector } from 'react-redux';

const Form = () => {
    const counter = useSelector(state => state.counter);
    const [inputCounter, setInputCounter]
                                = useState(counter);

    return (
        <form>
            {/* ... */}
        </form>
    );
}

export default Form;
```

Aby zaktualizować Store, muszę obsłużyć zdarzenie o nazwie submit, co zrobiłem obok.

Pamiętaj o zatrzymaniu domyślnej akcji formularza!

```
// ./src/Form.js
import React, { useState } from 'react';
import { useSelector } from 'react-redux';

const Form = () => {
    // ...
    const handleSubmit = event => {
        event.preventDefault();
        console.log('uruchom akcję [setCounter]');
    }

    return (
        <form onSubmit={ handleSubmit }>
            {/* ... */}
        </form>
    );
}

export default Form;
```

Teraz uruchomimy akcję setCounter. Użyjemy do tego celu kolejnego hooka o nazwie useDispatch.

Dzięki niemu tworzymy funkcję, do której przekazujemy dane uruchamianej akcji.

W naszym przypadku to akcja setCounter z odpowiednim ładunkiem.

Wszystko już powinno działać, ale...

```
// ./src/Form.js
import React, { useState } from 'react';
import { useSelector, useDispatch } from 'react-redux';

const Form = () => {
    // ...
    const dispatch = useDispatch();
    const handleSubmit = event => {
        event.preventDefault();
        console.log('uruchom akcję [setCounter]');
        dispatch({
            type: 'setCounter',
            payload: {
                value: Number(inputCounter),
            }
        });
    }
   // ...
}
```

Czy przypadkiem tej akcji nie zdefiniowaliśmy już w pliku ./src/actions/counter.js?

Pewnie, że tak! Wykorzystajmy ją!

Teraz samo sprawdzenie danych lub ich konwersję (Number) można by było przenieść do Akcji.

Całość kodu znajdziesz na GH. Zachęcam Cię także do obejrzenia odcinka na YouTubie dotyczącego hooków [~10 min] oraz ponownego zajrzenia do prezentacji twórców Facebooka [~ 45 min].

```
// ./src/Form.js
import React, { useState } from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { setCounterAction } from './actions/counter';

const Form = () => {
    // ...
    const dispatch = useDispatch();
    const handleSubmit = event => {
        event.preventDefault();
        console.log('uruchom akcję [setCounter]');
        dispatch(
            setCounterAction(Number(inputCounter))
        );
    }
    // ...
}
```

#09 Toolkit

02. Redux

Redux Toolkit powstał w celu ułatwienia budowania całej struktury ekosystemu wokół Reduxa.

Jak już pewnie zauważyłeś, tworzenie akcji wraz z Reducerem jest dość czasochłonne i wydaje się być dużą ilością pracy, którą trzeba na początku włożyć, aby móc korzystać z dobrodziejstw tego rozwiązania.

Dlatego powstało narzędzie, które ma to ułatwić: Redux Toolkit.

Pierwsze wzmianki o nim pojawiły się już w 2015 roku, ale dopiero w 2022 faktycznie przyjęło się ono w świecie Reduxa.

W tym czasie twórcy postanowili wymusić korzystanie z Toolkita poprzez oznaczenie budowania magazynu przy wykorzystaniu *createStore()* jako *deprecated* – co już sam zauważyłeś podczas czytania tego e-booka.

Pamiętaj, że w większości „starszych” aplikacji ekosystem Reduxa tworzony jest za pomocą *createStore()*. Dopiero w nowych aplikacjach wdrażany jest Toolkit.

Zaczynamy od instalacji paczki z npm:

`npm i @reduxjs/toolkit@1`

Następnie modyfikujemy kod, na którym skończyliśmy poprzedni rozdział.

Zaczynamy od pliku `src/index.js` i dodajemy do niego import funkcji `createSlice()`, która pozwoli nam zautomatyzować tworzenie całego ekosystemu Redux.

```
// ...
import { createStore } from 'redux';
import { Provider } from 'react-redux';
import reducers from './reducers';

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

const store = createStore(
    reducers,
    window.__REDUX_DEVTOOLS_EXTENSION__
    && window.__REDUX_DEVTOOLS_EXTENSION__()
);

// ...
```

Wspomniana funkcja przyjmuje jako parametr obiekt z odpowiednimi ustawieniami.

Właściwość `name` określa nazwę elementu, dla którego tworzymy rozwiązanie. Jest to dowolna nazwa, która ma to rozwiązanie identyfikować.

Kolejny element to wartości początkowe dla naszego magazynu, tj. `initalState`.

Ostatni element to `reducers` – zawiera on działania, które mają być wykonane przy określonej akcji.

```
// ...
import { createStore } from 'redux';
import { Provider } from 'react-redux';
import reducers from './reducers';

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

const counterSlice = createSlice({
    name: 'counter',
    initialState: { counter: 0 },
    reducers: {}
});

// ...
```

Przypomnę, że do tej pory nasze akcje wyglądały tak, jak na przykładzie obok.

Pierwsza dotyczyła zwiększania wartości licznika o dwa (step: 2).

Druga pozwalała ustawiać dowolną jego wartość określoną w value.

```
export const increaseCounterAction = () => {
    return {
        type: 'increaseCounter',
        payload: {
            step: 2,
        },
    }
}

export const setCounterAction = value => {
    return {
        type: 'setCounter',
        payload: {
            value: value,
        },
    }
}
```

Natomiast dotychczasowy reducer wyglądał tak, jak na przedstawionym przykładzie.

Akcje wywoływały konkretną partię kodu w reduktorze, gdzie były wykorzystywane dane przekazywane przez payload.

```
const initState = {
    counter: 0,
};

const reducer = (state = initState, action) => {
    switch(action.type) {
        case 'increaseCounter':
            const { step } = action.payload;
            return {
                counter: state.counter + step,
            };
        case 'setCounter':
            return {
                counter: action.payload.value,
            };
        default:
            return state;
    }
}

export default reducer;
```

Dzięki Toolkitowi nie będziemy tworzyć akcji! On zrobi to za nas. My musimy jedynie określić, co ma się dziać w reduktorze.

Na przykładzie obok definiuję odpowiednie funkcje. To na ich podstawie będą wygenerowane akcje, o których powiemy za chwilę.

Zwróć uwagę, że tutaj nie dodaję przyrostka *Action*, ponieważ Toolkit sam wygeneruje odpowiednie akcje na podstawie nazwy funkcji w reducers.

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

const counterSlice = createSlice({
    name: 'counter',
    initialState: { counter: 0 },
    reducers: {
      increase(state) {
        state.counter += 2;
      },
      set(state, action) {
        state.counter = action.payload;
      }
    }
})

// ...
```

W tym miejscu nie muszę również przejmować się kopiowaniem właściwości i wartości poprzedniego stanu przed każdym zwróceniem nowego stanu – Toolkit zrobi to za mnie.

Jak widać na przykładzie, pierwszy parametr funkcji to aktualny stan magazynu, a drugi to dane dotyczące akcji.

To właśnie w drugim parametrze znajdziesz właściwość payload, która przechowuje dane przekazane podczas uruchamiania akcji (o tym za chwilę).

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

const counterSlice = createSlice({
    name: 'counter',
    initialState: { counter: 0 },
    reducers: {
      increase(state) {
        state.counter += 2;
      },
      set(state, action) {
        state.counter = action.payload;
      }
    }
})

// ...
```

Teraz możemy utworzyć magazyn (store).

Wystarczy że zaimportujemy funkcję `configureStore()` i przekażemy do niej `reducer`, który został zwrócony przez `createSlice()` na podstawie naszej konfiguracji.

Zwróć uwagę, że funkcja `configureStore()` przyjmuje obiekt zawierający właściwość `reducer`, i to do tej właściwości przypisujemy Reduktor.

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

const counterSlice = createSlice({
  // ...
  reducers: {
    increase(state) {
      state.counter += 2;
    },
    set(state, action) {
      state.counter = action.payload;
    }
  }
})

const store = configureStore({
  reducer: counterSlice.reducer,
});
```

Ponieważ będziemy potrzebować dostępu do wygenerowanych akcji w wielu miejscach, wygodniej będzie przerzucić kod dotyczący magazynu do osobnego pliku.

W naszym przypadku będzie to plik index.js w lokalizacji src/store.

Z tego pliku eksportujemy zmienną store, która zawiera dane naszego magazynu.

```
// src/store/index.js
import { configureStore, createSlice }
  from '@reduxjs/toolkit';

const counterSlice = createSlice({
  name: 'counter',
  initialState: { counter: 0 },
  reducers: {
    increase(state) {
      state.counter += 2;
    },
    set(state, action) {
      state.counter = action.payload;
    }
  }
})

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

Teraz w pliku src/index.js importujemy nasz store.

Dzięki temu rozwiązaniu cały ekosystem Reduxa mamy w jednym pliku. Tutaj jedynie przekazujemy dostęp do magazyny przez Provider.

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

import { Provider } from 'react-redux';
import { store } from './store';

const root = ReactDOM.createRoot(
      document.getElementById('root'));
root.render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>
);
// ...
reportWebVitals();
```

Teraz zajmiemy się naszymi akcjami.

Dostęp do akcji wyeksportujemy z pliku `src/store/index.js`.

Wystarczy, że przypiszemy do zmiennej `actions` właściwość `.actions` z obiektu `counterSlice` i ją wyeksportujemy.

```
// src/store/index.js
import { configureStore, createSlice }
  from '@reduxjs/toolkit';

// ...

export const store = configureStore({
  reducer: counterSlice.reducer,
});

export const actions = counterSlice.actions;
```

Zanim przejdziemy do użycia obecnych akcji, przypomnę, jak do tej pory z nich korzystaliśmy.

W katalogu `actions/counter` mieliśmy zdefiniowane akcje. Importowaliśmy te, które wykorzystywaliśmy w danym komponencie, i podpinaliśmy je do `props` przez `connect()`.

```
// src/App.js
import React from 'react';
import './App.css';

import { connect } from 'react-redux';
import { increaseCounterAction } from './actions/counter';

// ...
const mapStateToProps = (state, props) => {
    return {
        counter: state.counter,
    }
}

const mapActionToProps = {
    onIncrease: increaseCounterAction,
};

export default connect(
    mapStateToProps, mapActionToProps
)(App);
```

Teraz robimy to tak samo, tylko że akcje importujemy z naszego pliku, który zawiera deklarację magazynu.

Zwróć uwagę, że dana akcja (właściwość w importowanym obiekcie `actions`) będzie nazywać się identycznie jak funkcja zdefiniowana we właściwości `.reducers` w `counterSlice` w magazynie.

```
import React from 'react';
import './App.css';

import { connect } from 'react-redux';
import { actions } from './store/index';

// ...
const mapStateToProps = (state, props) => {
    return {
        counter: state.counter,
    }
}

const mapActionToProps = {
    onIncrease: actions.increase, // przypisujemy akcję!
};

export default connect(
    mapStateToProps, mapActionToProps
)(App);
```

Oczywiście możemy użyć tego rozwiązania również podczas korzystania z hooków.

Taki przypadek mamy w pliku src/Form.js.

Jedyna zmiana, jaką muszę wprowadzić, to zaimportowanie akcji z magazynu i jej uruchomienie.

Zakomentowałem kod, który uruchamiał poprzednią akcję, aby łatwiej było porównać to z poprzednim rozwiązaniem.

```
// ./src/Form.js
import React, { useState } from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { actions } from './store';

const Form = () => {
    const counter = useSelector(state => state.counter);
    const [inputCounter, setInputCounter] = useState(counter);
    const dispatch = useDispatch();
    const handleSubmit = event => {
        event.preventDefault();
        console.log('uruchom akcję [setCounter]');
        // dispatch(
        //     setCounterAction(Number(inputCounter))
        // );
        dispatch(
            actions.set(Number(inputCounter))
        );
    }
    // ...
}
```

Warto na tym etapie zwrócić uwagę, że to, co przekazujemy do metody actions.set(), jest odbierane we właściwości .payload drugiego parametru w .reducers.

Na następnym slajdzie prezentuję raz jeszcze kod dotyczący magazynu.

```
// ./src/Form.js
import React, { useState } from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { actions } from './store';
const Form = () => {
    const counter = useSelector(state => state.counter);
    const [inputCounter, setInputCounter] =
                                        useState(counter);
    const dispatch = useDispatch();
    const handleSubmit = event => {
        event.preventDefault();
        console.log('uruchom akcję [setCounter]');
        // dispatch(
        //     setCounterAction(Number(inputCounter))
        // );
        dispatch(
            actions.set(Number(inputCounter))
        );
    }
    // ...
}
```

W funkcji `set()` mam dwa parametry.

Drugi zawiera właściwość `.payload`, która przechowuje dane przekazywane w akcji.

Jeśli chcemy zdefiniować więcej wartości do przekazania, możemy to zrobić poprzez zdefiniowanie obiektu.

Taki obiekt będzie dostępny we wspomnianej właściwości.

```
// src/store/index.js
import { configureStore, createSlice }
  from '@reduxjs/toolkit';

const counterSlice = createSlice({
  name: 'counter',
  initialState: { counter: 0 },
  reducers: {
    increase(state) {
      state.counter += 2;
    },
    set(state, action) {
      state.counter = action.payload;
    }
  }
})

// ...
```

Dla przykładu zmodyfikuję wnętrze funkcji set().

Zakładam, że .payload będzie obiektem, który zawiera właściwość .value – jej wartość przypiszę do state.

Działanie się nie zmieni, ale sposób przekazywania danych już tak.

Dlatego na następnym slajdzie modyfikuję akcję.

```
// src/store/index.js
import { configureStore, createSlice }
  from '@reduxjs/toolkit';

const counterSlice = createSlice({
  name: 'counter',
  initialState: { counter: 0 },
  reducers: {
    increase(state) {
      state.counter += 2;
    },
    set(state, action) {
      const { value } = action.payload
      state.counter = value;
    }
  }
})

// ...
```

Wystarczy, że teraz w pliku src/form.js do metody actions.set() przekażę obiekt o odpowiedniej strukturze. Wszystko zadziała.

Jeśli będę potrzebować więcej danych do akcji, to wystarczy, że przekażę je w formie rozbudowanego obiektu, który odbiorę w magazynie.

```
// ./src/Form.js
import React, { useState } from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { actions } from './store';
const Form = () => {
    const counter = useSelector(state => state.counter);
    const [inputCounter, setInputCounter] =
                                        useState(counter);
    const dispatch = useDispatch();
    const handleSubmit = event => {
        event.preventDefault();
        console.log('uruchom akcję [setCounter]');

        dispatch(
            actions.set(
                {value: Number(inputCounter)}
            )
        );
    }
    // ...
}
```

To wszystko, co musimy wiedzieć na temat Redux Toolkit.

Teraz można oczyścić nasz projekt z pozostałości standardowego użycia Reduxa.

Możemy usunąć katalog `actions`, a cały `counterSlice` przerzucić do `reducers` – w nim również będzie eksport akcji.

Pamiętajmy, że musimy wówczas zmienić ścieżki przy importowych akcjach.

Cały projekt po refaktorze znajdziesz w repozytorium [react-redux-toolkit-helloworld].

Zadania (*practice*) do tego działu zrealizuj bez korzystania z Redux Toolkit, lecz w projekcie (*task*) możesz już się posiłkować nim posiłkować.

W ten sposób poznasz oba rozwiązania, z którymi możesz spotkać się w pracy.

Uwaga: *Toolkit od razu podłącza Redux DevTools więc nic nie musimy robić, aby to działało :)*

Zadania do wykonania

Pod poniższym adresem znajdziesz zadania,
które obejmują materiał prezentowany w niniejszym ebooku.

https://github.com/devmentor-pl/practice-react-redux-basics