

Reaktywne programowanie aplikacji webowych w oparciu o React i Redux

Mateusz Jabłoński



Kim jestem?

- ◆ programista od 2011 roku
- ◆ głównie: Javascript / Java / dawniej: PHP
- ◆ szkoleniowiec / trener / mentor od 2016 roku
- ◆ prywatnie tata i mąż
- ◆ ostatnimi czasy również nizołek w sesjach D&D

Ustalenia

- ▶ Cel i agenda
- ▶ Wzajemne oczekiwania
- ▶ Pytania i dyskuje
- ▶ Elastyczność
- ▶ Otwartość i uczciwość

Agenda, czyli co nas czeka?

1. Wprowadzenie architektury opartej o Redux
2. Instalacja i konfiguracja Redux w aplikacji React
3. Tworzenie reduktorów i akcji
4. Wykorzystanie selektorów i mapowanie stanu do komponentów
5. Dispatch i modyfikacja stanu za pomocą akcji
6. Middleware i rozszerzenia Reduxa
7. Organizacja dużych aplikacji z Reduxem
8. Debugowanie i testowanie aplikacji Redux
9. Optymalizacja wydajności aplikacji z Reduxem
10. Alternatywy dla Redux oraz przyszłość zarządzania stanem

Repozytorium

github.com/matwjablonski/zus-redux-2411

wprowadzenie architektury opartej o Redux

Globalne zarządzanie stanem

- ◆ pojedynczy obiekt globalny dostępny do odczytu i zmiany z dowolnej części naszej aplikacji
- ◆ wszystkie zmiany i dane mogą być śledzone z jednego miejsca
- ◆ centralizowany miejsce przechowywania i zarządzania danymi
- ◆ przydatne, gdy dane są potrzebne wielu komponentom w aplikacji, ponieważ zapewnia, że mają one dostęp do najnowszej i najbardziej aktualnej wersji danych
- ◆ komponenty mogą nasłuchiwać zmian w stanie i aktualizować swoje dane automatycznie, kiedy nastąpi zmiana w magazynie
- ◆ dzięki centralizacji unika się chaosu związanego z ręcznym przesyłaniem danych między komponentami, co ułatwia skalowanie aplikacji

Kiedy warto używać globalnego zarządzania stanem?

- ◀ dane są współdzielone pomiędzy komponentami - kiedy komponenty z różnych części aplikacji potrzebują tych samych danych, typu: koszyk w ecommerce, dane profilu, token, motyw, język, preferencje użytkownika
- ◀ złożone zależności pomiędzy komponentami - kiedy dane przekazywane są przez wiele poziomów aplikacji, pomimo że komponenty pośrednie tych danych nie potrzebują (w React zjawisko to nazywa się "prop drilling")
- ◀ stan jest często modyfikowany - kiedy stan aplikacji jest często zmieniany w różnych miejscach w aplikacji, wówczas stan globalny pozwala utrzymać spójność, przykład: system powiadomień
- ◀ zapytania asynchroniczne i ich stan - kiedy aplikacja wysyła asynchroniczne zapytania, a ich wynik wpływa na stan aplikacji w różnych jej miejscach, np: lista produktów / promocji
- ◀ zarządzanie stanem interfejsu użytkownika - do obsługi globalnych elementów widoku, takich jak modale, toasty, spinnery
- ◀ wymóg spójności danych - kiedy aplikacja wymaga spójności danych pomiędzy różnymi komponentami - jedna zmiana wpływa na kilka innych miejsc

Kiedy NIE używać globalnego zarządzania stanem?

- ◀ małe aplikacje - najczęściej w małych aplikacjach wystarczy lokalne zarządzanie stanem
- ◀ proste aplikacje bez złożonej logiki - dodanie narzędzia do zarządzania stanem globalnym, będzie dodaniem niepotrzebnej złożoności
- ◀ dane są izolowane - np. gdy dane są specyficzne per komponent

MVC

- ▶ wzorzec architektoniczny dedykowany aplikacjom GUI
- ▶ został zaprojektowany ** w 1979 roku ** przez Trygve Reenskaug (Xerox)
- ▶ głównym celem jest odseparowanie modelu (reprezentacja danych) od widoku (interfejs użytkownika)
- ▶ za logikę w aplikacji odpowiada Kontroler (Controller)
- ▶ wykorzystywany w takich frameworkach jak: Django, Ruby on Rails, AngularJS, Zend Framework, Ember i wiele innych

"Models represent knowledge. A model could be a single object (rather uninteresting), or it could be some structure of objects."

-- *Trygve Reenskaug, 2007*

Model w MVC

- ◆ odpowiada za logikę biznesową
- ◆ w modelu zorganizowana jest cała wiedza na temat danych w aplikacji
- ◆ zapewnia ustandaryzowany dostęp do danych
- ◆ dzięki modelowi reszta aplikacji staje się niezależna od pochodzenia danych

"A view is a (visual) representation of its model. It would ordinarily highlight certain attributes of the model and suppress others. It is thus acting as a presentation filter. A view is attached to its model (or model part) and gets the data necessary for the presentation from the model by asking questions. It may also update the model by sending appropriate messages."

-- Trygve Reenskaug, 2007

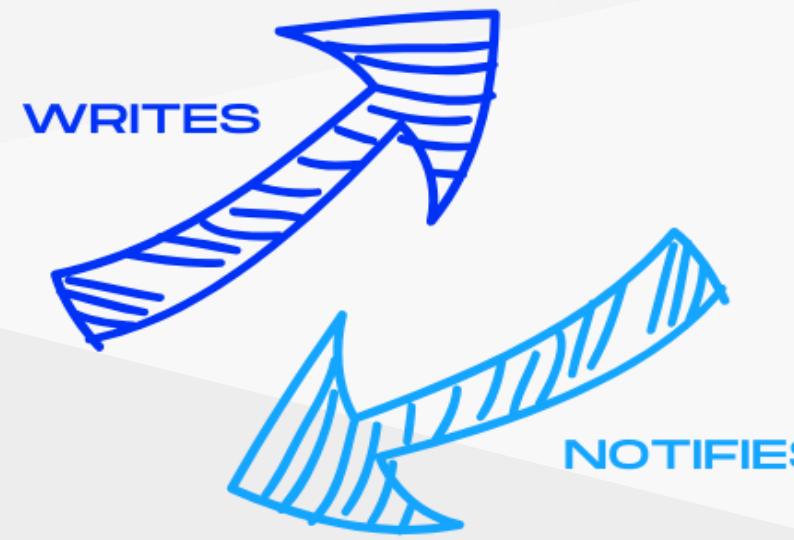
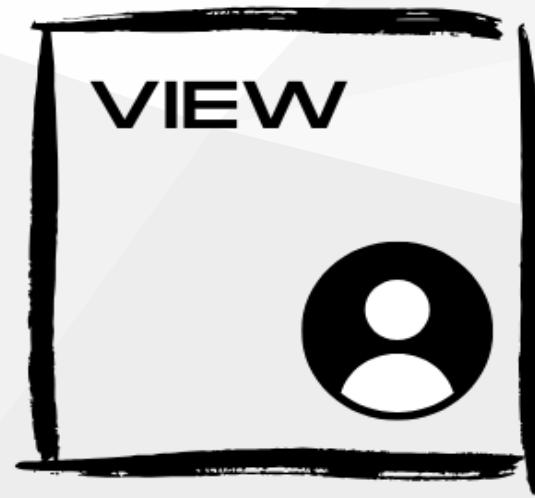
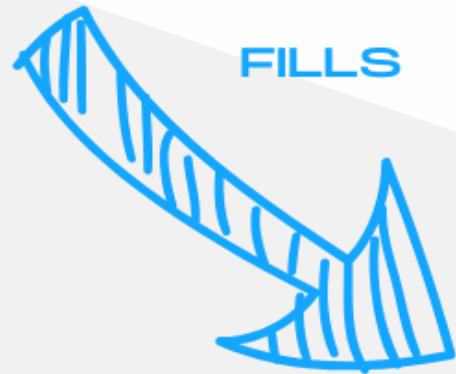
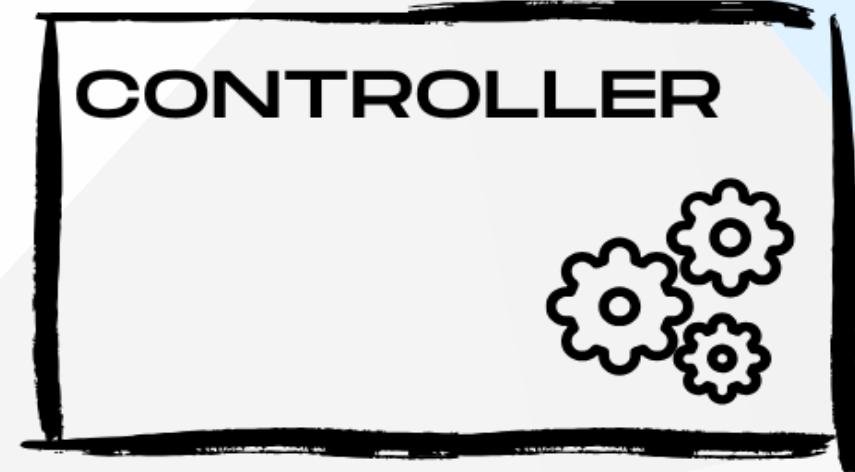
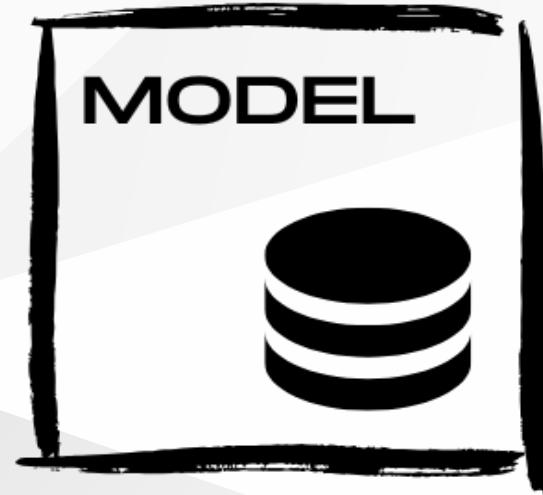
Kontroler (Controller) w MVC

Odpowiada za:

- ▶ logikę aplikacji
- ▶ przetwarzanie / przekształcanie danych z modelu
- ▶ przekazanie danych do widoku
- ▶ odebranie danych od użytkownika i ich odpowiednie zapisanie
- ▶ sterowanie całą aplikacją

widok (view) w MVC

- ◀ reprezentuje interfejs użytkownika (to, co widzi użytkownik)
- ◀ widok oddzielony od logiki pozwala na zmianę wyglądu bez konieczności modyfikacji logiki
- ◀ widok nie musi zakładać pełnego GUI (może to być jego wersja ograniczona, np. do linii poleceń)



Problemy w aplikacjach frontedowych

- ◀ w aplikacjach, gdzie warstwa Widoku jest wydzielona do klienta a warstwa Logiki (Controller i Model) pozostają na serwerze mamy tylko dwa punkty styku pomiędzy nimi: request i response
- ◀ w przypadku aplikacji CSR (Client Side Rendering) Kontroler jest bardzo mocno uzależniony od Widoku
- ◀ Widok i Kontroler tworzą wiele dwukierunkowych powiązań
- ◀ przeładowane Modele - przechowują zarówno dane stanu interfejsu użytkownika (produkty, artykuły itp), jak i stanu aplikacji (wybrane zakładka, kolor theme'u)
- ◀ złamanie zasady Single Responsibility: Kontroler - zajmuje się obsługą zdarzeń oraz logiką biznesową, natomiast Model - nie ma oddzielnych mechanizmów zarządzania stanem interfejsu użytkownika i stanem aplikacji

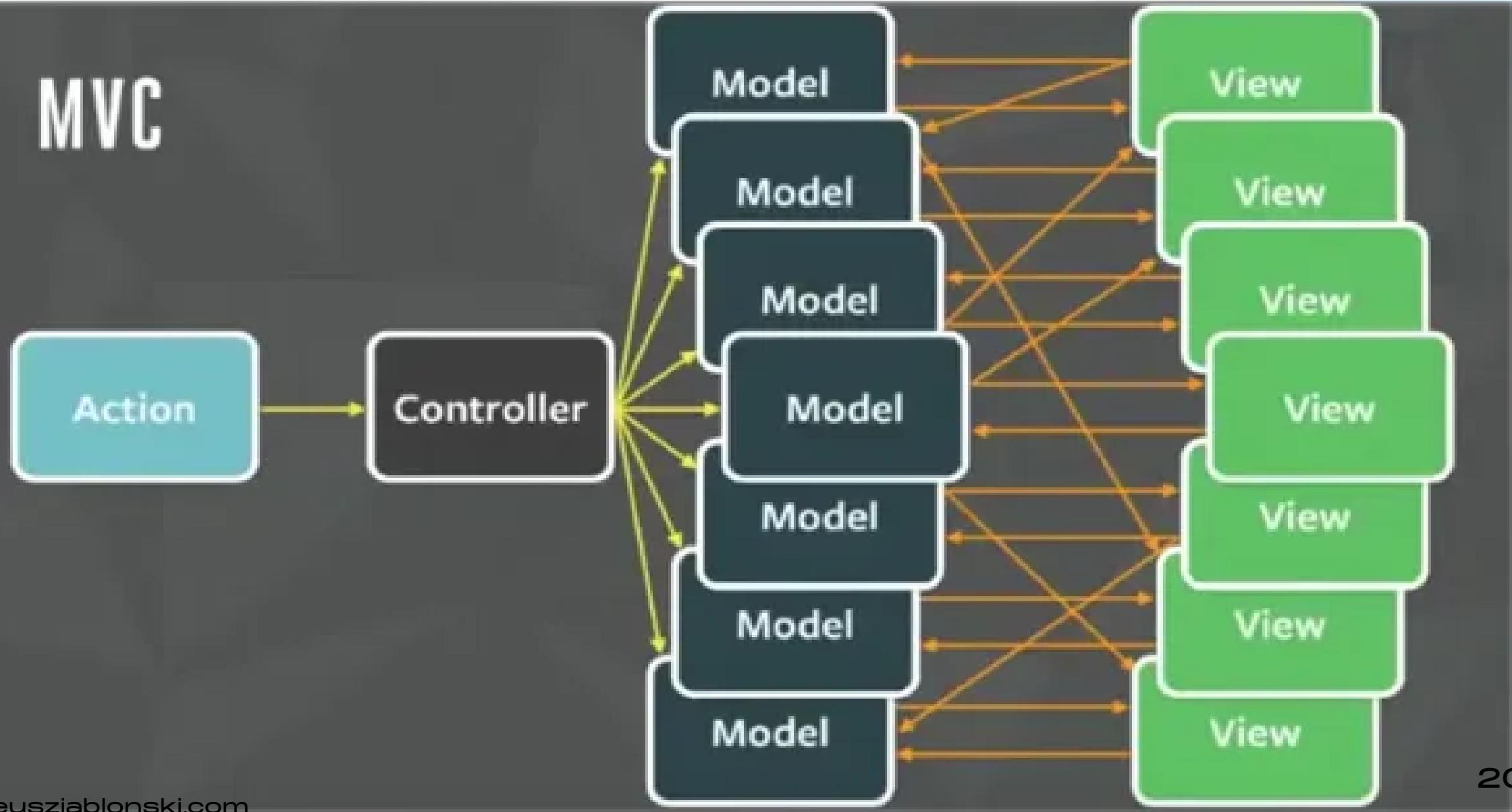
Historia Facebooka

- do roku 2014 Facebook budował swój UI z wykorzystaniem architektury MVC
- podczas konferencji Hacker Way zespół Facebook'a ogłosił, że MVC nie działa w ich przypadku oraz że postanowili przenieść się na architekturę Flux
- zespół Facebooka stwierdził, że MVC bazuje na "two-way data binding" co powoduje, że MVC nie skaluje się

Two-way data binding

- ▶ jest wykorzystywany w wielu innych rozwiązańach FE, jak: Angular czy Vue
- ▶ zakłada dwukierunkowy przepływ danych od modelu do widoku i na odwrót

MVC



Dlaczego dwukierunkowy przepływ danych się nie skaluje?

- ◀ obejmuje relacje many-to-many, które są trudne do utrzymania i zarządzania
- ◀ wraz z rozrostem aplikacji ilość powiązań niekontrolowane wzrasta, co prowadzi do problemów z refaktoryzacją i optymalizacją komunikacji

The “zombie” unseen messages count

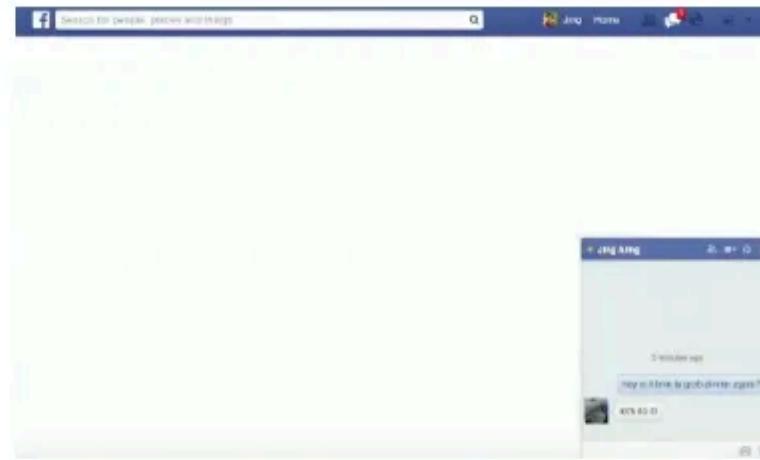
- ◀ problem z MVC w Facebook zaczął się w momencie ich największego rozwoju, kiedy dodawali do swojej aplikacji funkcjonalności związane z chatem
- ◀ problem dotyczył licznika nieprzeczytanych wiadomości, które były widoczne na różnych widokach
- ◀ użytkownicy narzekali, że pomimo iż nie mieli nieodczytanych wiadomości licznik wskazywał fałszywą wartość
- ◀ problem wracał za każdym razem po wprowadzeniu nowych funkcjonalności, ponieważ było coraz więcej edge case'ów
- ◀ problemy z czatem były tylko wierzchołkiem góry lodowej problemów z MVC



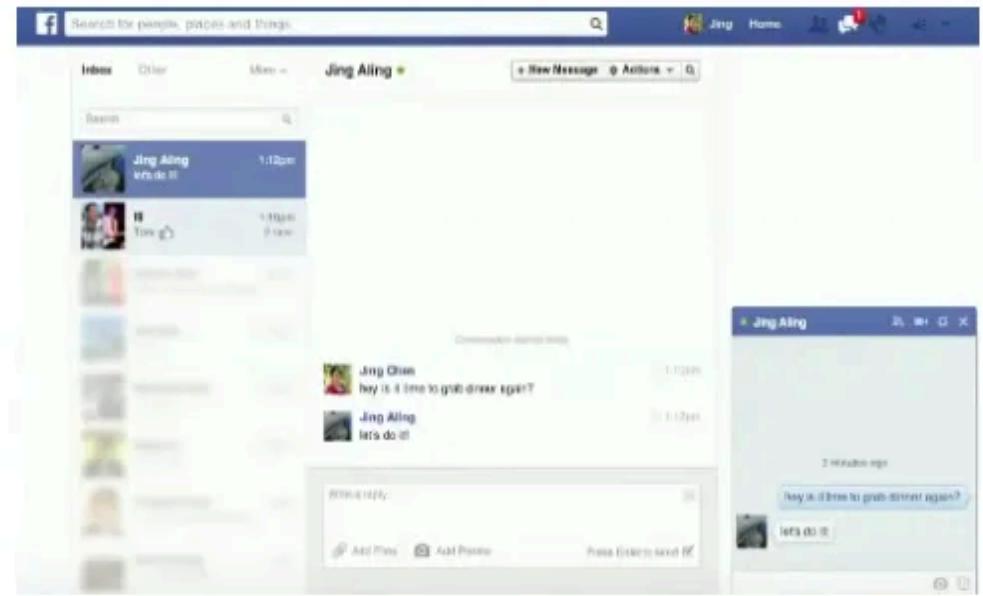
Chat tab



Chat messages tool



The larger messages view



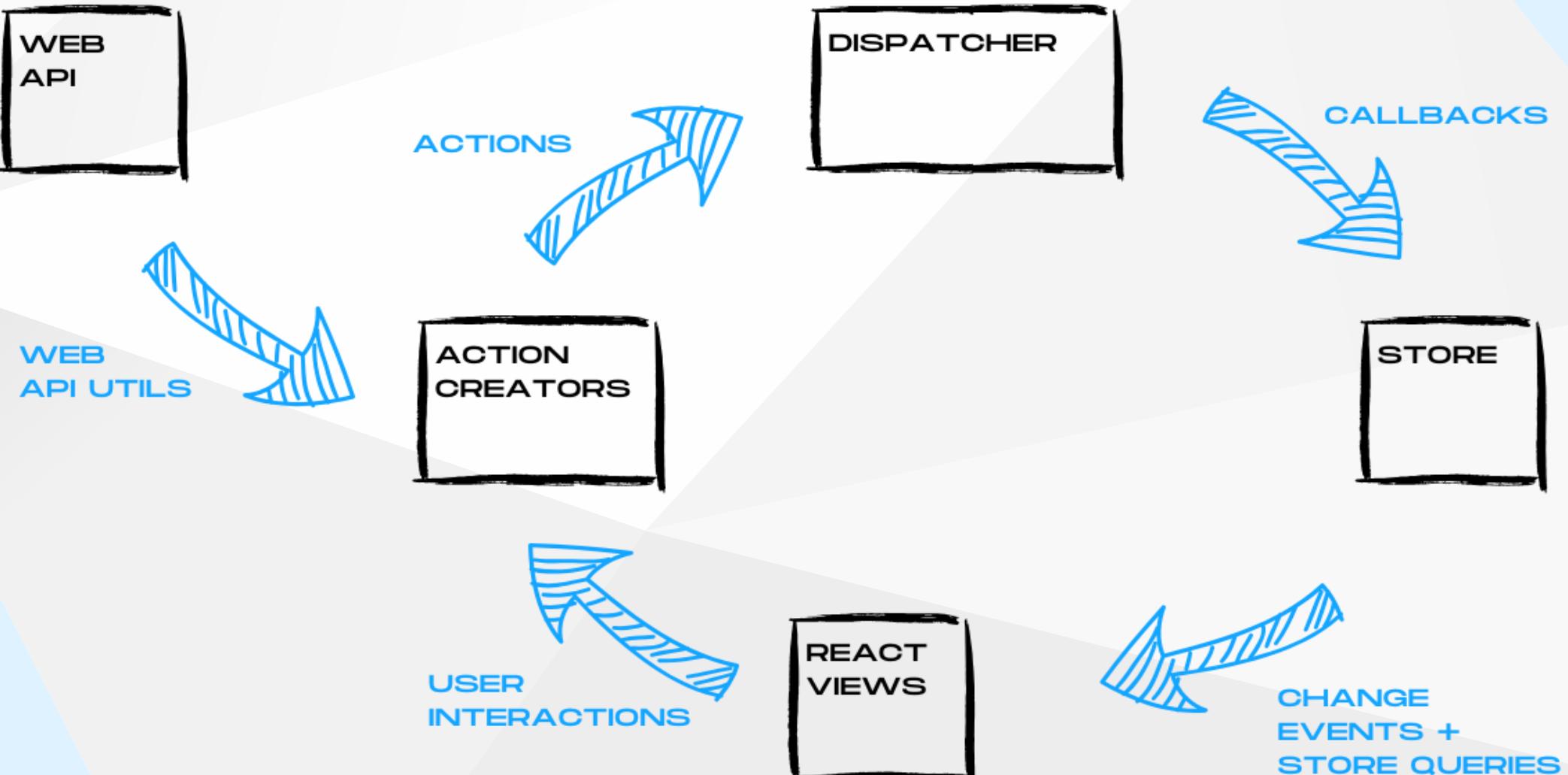
One-way data binding

- ◀ jednokierunkowe wiązanie danych
- ◀ dane przepływają tylko w jednym kierunku
- ◀ najczęściej jest to związane z wyświetleniem danych z modelu na widoku
- ◀ widok nie może wprowadzić żadnych zmian w modelu

Flux

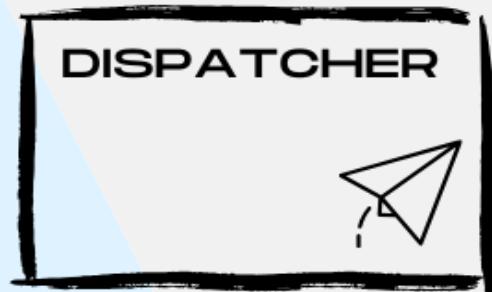
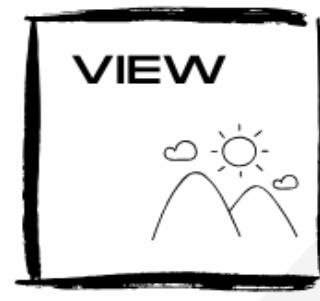
- ▶ FLUX opiera się o wzorzec projektowy CQRS (Command Query Responsibility Segregation)
- ▶ architektura FLUX zakłada jednokierunkowy przepływ danych
- ▶ akcje są jedynym sposobem na zmianę stanu aplikacji
- ▶ dispatchery przekazują akcje do specyficznego Magazynu (Store'a)
- ▶ magazyn (Store) jest jedynym źródłem prawdy, może zmienić swój stan w reakcji na akcje
- ▶ stan aplikacji jest pobierany z magazynu i przekazywany do widoku
- ▶ aplikacja może mieć wiele magazynów
- ▶ widok obserwuje magazyny i renderują zmiany w aplikacji





A co gdy mamy wiele magazynów?

- ◆ dispatcher wysyła akcje do wszystkich magazynów
- ◆ dispatcher sam w sobie nie posiada logiki związanej z aktualizacją magazynu (store'a)
- ◆ to magazyn jest odpowiedzialny za specyficzną domenę i tylko on posiada logikę biznesową
- ◆ zazwyczaj magazyn emituje zdarzenie do specjalnego widoku (zwanego też controller-view), który jest odpowiedzialny za propagację zdarzenia do pozostałych widoków
- ◆ co istotne w architekturze komponentowej controller-view powoduje przerenderowanie całego drzewa w dół od niego



Przykładowy kod:

```
const ADD_FLIGHT = {  
  type: 'ADD_FLIGHT',  
  payload: {  
    id: 1235,  
    name: 'WAW/SFO',  
  },  
};  
  
const REMOVE_FLIGHT = {  
  type: 'REMOVE_FLIGHT',  
  payload: {  
    id: 1235,  
  },  
};  
  
const UPDATE_FLIGHT = {  
  type: 'UPDATE_FLIGHT',  
  payload: {  
    id: 1235,  
    name: 'WAW/SFO',  
    status: 'canceled',  
  },  
};
```

Przykładowy kod:

```
class FlightStore extends EventEmitter {  
  constructor() {  
    this.store = {  
      flights: [],  
    }  
  }  
  
  handleAction(action) {  
    switch(action.type) {  
      case 'ADD_FLIGHT':  
        this.store.flights.push(action.payload);  
        break;  
    }  
  
    this.notifyViews();  
  }  
  
  notifyView() {  
    // logika powiadamiania widoku  
  }  
}
```

zalety architektury Flux

- ▶ wykrywanie zmian jest proste i wydajne, w związku z niemutowanymi strukturami danych. Musimy po prostu porównać obiekty stanu - nowy ze starym.
- ▶ kiedy obiekt jest taki sam, nic się zmieniło, nie ma potrzeby żeby aktualizować Widok
- ▶ daje kontrolę nad danymi i pozwala na modularne podejście w zarządzaniu nimi

wady architektury Flux

- ▶ wysoce skomplikowana architektura - trudność w opanowaniu
- ▶ wymaga bardzo dużych nakładów czasu i pracy we wdrożeniu i utrzymaniu



Czy Flux to nowe MVC?

- ◆ po wprowadzeniu Flux'a społeczność krytykowała go, że tak naprawdę to po prostu nowe MVC (przynajmniej ta sama implementacja, która była w Facebooku), do którego dodany specjalny kontroler odpowiedzialny za wykonywanie zapytań jedno po drugim
- ◆ posiadanie rozproszonej logiki na wiele magazynów może być tożsame z posiadaniem wielu kontrolerów
- ◆ koncepcje jednokierunkowego i dwukierunkowego przepływu danych są jednak zbyt różne
- ◆ Facebook nie zaimplementował MVC poprawnie stąd też problemy, które napotkał

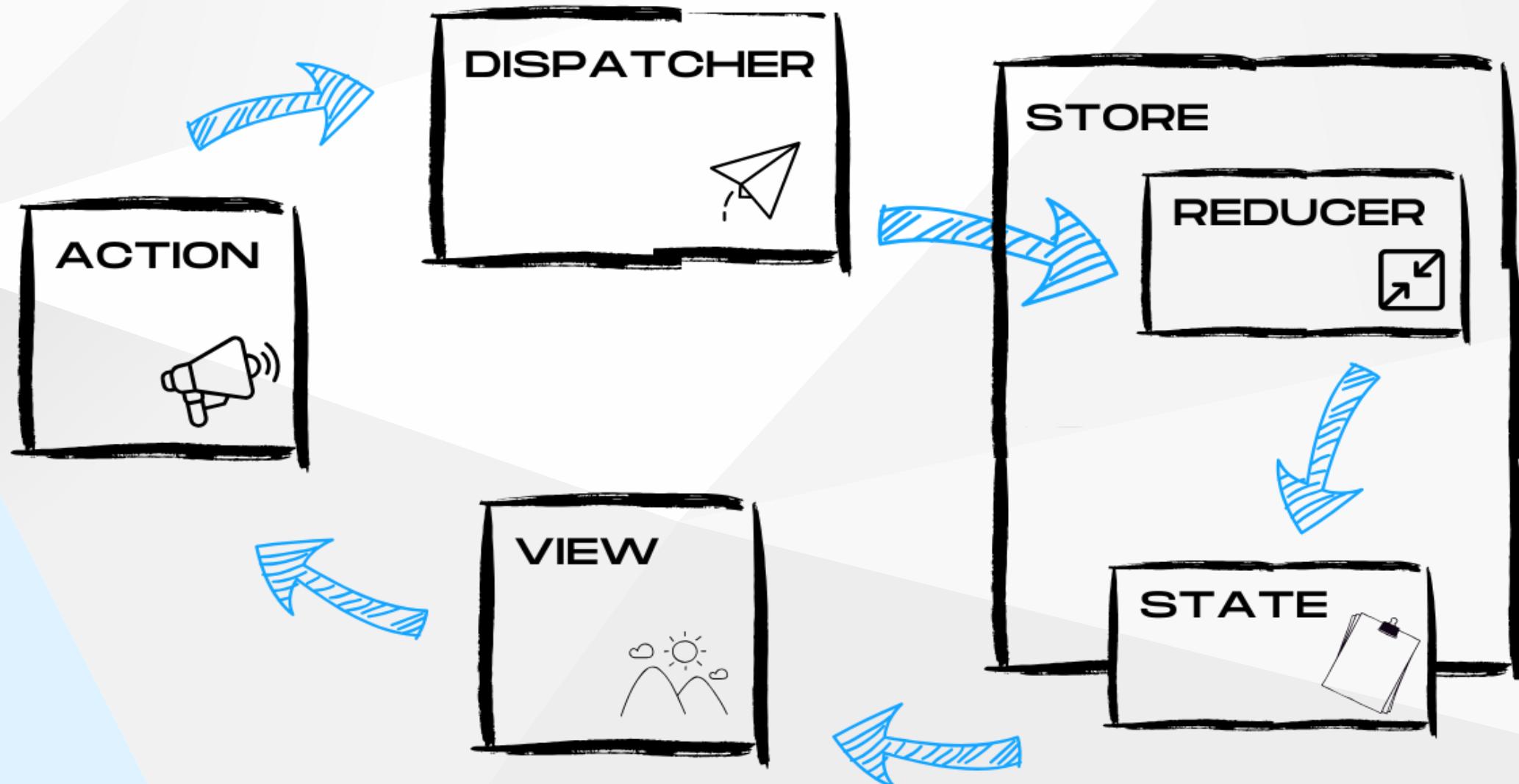
"Redux is not that different from Flux. Overall it has same architecture, but Redux is able to cut some complexity corners by using functional composition where Flux uses callback registration."

-- *Dan Abramov, 2015*

Redux – główne założenia

Redux jest implementacją architektury Flux, inspirowaną programowaniem funkcyjnym

- ▶ jeden magazyn (jedno źródło stanu)
- ▶ niemutowalny stan
- ▶ magazyn nie modyfikuje danych (logika aktualizacji stanu jest wydzielona do reduktorów)



Akcja

- ▶ obiekt, który posiada właściwość `type`
- ▶ `type` jest typu string i odpowiada za nazwę wykonywanej operacji
- ▶ akcje zwykle zawierają także informacje, które będą przekazane do reducera i który wpłyną na store
- ▶ drugie pole najczęściej nazywane jest `by` lub `payload`
- ▶ często można spotkać się z sytuacją, gdy type nie jest bezpośrednim ciągiem znaków a stałą - taki zabieg ma na celu uniknięcie literówek przy używaniu tej samej akcji w różnych miejscach naszej aplikacji

Przykładowy kod:

```
const INCREMENT_COUNTER = 'INCREMENT COUNTER';

const incrementCounterAction = {
  type: INCREMENT_COUNTER,
  payload: {
    by: 10,
  }
}
```

Kreatory akcji

- ▶ kreatory akcji to funkcje, które zwracają akcje
- ▶ jako parametr funkcja przyjmuje tylko dane, które muszą zostać przekazane do reducerów
- ▶ poprzez "zaszycie" akcji w kreatorach zostaje uproszczony zapis a jednocześnie można lepiej parametryzować akcje

Przykładowy kod:

```
const INCREMENT_COUNTER = 'INCREMENT COUNTER';

type Action<T, P = {}> = {
  type: T,
  payload: P
}

type IncrementCounterActionPayload = { by: number }

const incrementCounter = (
  byValue: number
): Action<typeof INCREMENT_COUNTER, IncrementCounterActionPayload> => ({
  type: INCREMENT_COUNTER,
  payload: {
    by: byValue
  },
});
```

Jakie typy danych mogę umieścić w akcji?

- ▶ Redux nie nakłada na nas żadnych dodatkowych ograniczeń technologicznych dotyczących kształtu i typu akcji
- ▶ należy jednak pamiętać, że wszystkie elementy akcji powinny być serializowalne za pomocą metody `JSON.stringify`

zadanie nr 1

- ◀ przygotuj akcje, która będzie opisywała sytuację dodawania, usuwania i modyfikacji nowego zadania na liście zadań
- ◀ dla każdej akcji przygotuj kreator akcji
- ◀ dodaj typy, opisujące zarówno akcje, jak i kreatory



Dispatcher

- ▶ funkcje, które pozwalają na wyemitowanie akcji do magazynu
- ▶ efektem wywołania funkcji `dispatch` jest wywołanie reducerów oraz zaktualizowanie magazynu
- ▶ funkcja `dispatch` oczekuje, że zostanie wywołana z obiektem akcji
- ▶ jest to jedyny sposób na przekazanie danych do Reduxa w celu aktualizacji stanu

Przykładowy kod:

```
type Action<T = string, P = {}> = { type: T, payload: P }

type Store = {
  dispatch<T, P>(action: Action<T, P>): void;
};

declare const store: Store;

const INCREMENT_COUNTER_ACTION = 'INCREMENT COUNTER';

const incrementCounter = (
  value: number
): Action<
  typeof INCREMENT_COUNTER_ACTION,
  { value: number }
> => ({ type: INCREMENT_COUNTER_ACTION, payload: { value } })

store.dispatch(incrementCounter(10));
```

Reduktor (Reducer)

- ▶ reducer to czysta funkcja
- ▶ reducer wywołujemy z aktualnym stanem i akcją
- ▶ na podstawie parametrów wejściowych reducer wprowadza odpowiednie modyfikacje i zwraca nowy stan
- ▶ jeśli reducer nie wie, jak obsłużyć daną akcję powinien zwrócić niezmodyfikowany stan
- ▶ reducer nie powinien mutować otrzymanego stanu
- ▶ reducer nie może zwrócić `undefined` (ale może `null` lub `false`)
- ▶ reducer jest jedyną opcją zmiany stanu (żadna inna część aplikacji nie powinna mieć takich możliwości)
- ▶ reducery mogą pracować z zagnieżdżonym stanem

Czyste funkcje

- ▶ nie korzystają ze zmiennych globalnych
- ▶ nie modyfikują danych zewnętrznych
- ▶ nie wykonują efektów ubocznych (side effect'ów)
- ▶ jej wynik zależy tylko od jej argumentów
- ▶ dla wskazanych parametrów wejściowych zawsze zwrócą ten sam wynik
- ▶ zawsze przyjmują jakiś argument i zawsze coś zwracają

Przykładowy kod:

```
function reducer(state, action) {  
  switch(action.type) {  
    case INCREMENT_COUNTER:  
      return state + action.payload.value;  
    default:  
      return state;  
  }  
}
```

Stan początkowy

Dobrą praktyką jest definiowanie stanu początkowego i przekazanie go do argumentu jako wartość domyślna.

Przykładowy kod:

```
const initialState = 0;

const reducer = (state = initialState, action) => {
  switch (action.type) {
    case INCREMENT_COUNTER:
      return state + action.payload.value;
    default:
      return state;
  }
}
```

zadanie nr 2

- ◀ przygotuj reduktor, który będzie obsługiwał akcje dodawania, usuwania i modyfikacji nowego zadania na liście zadań



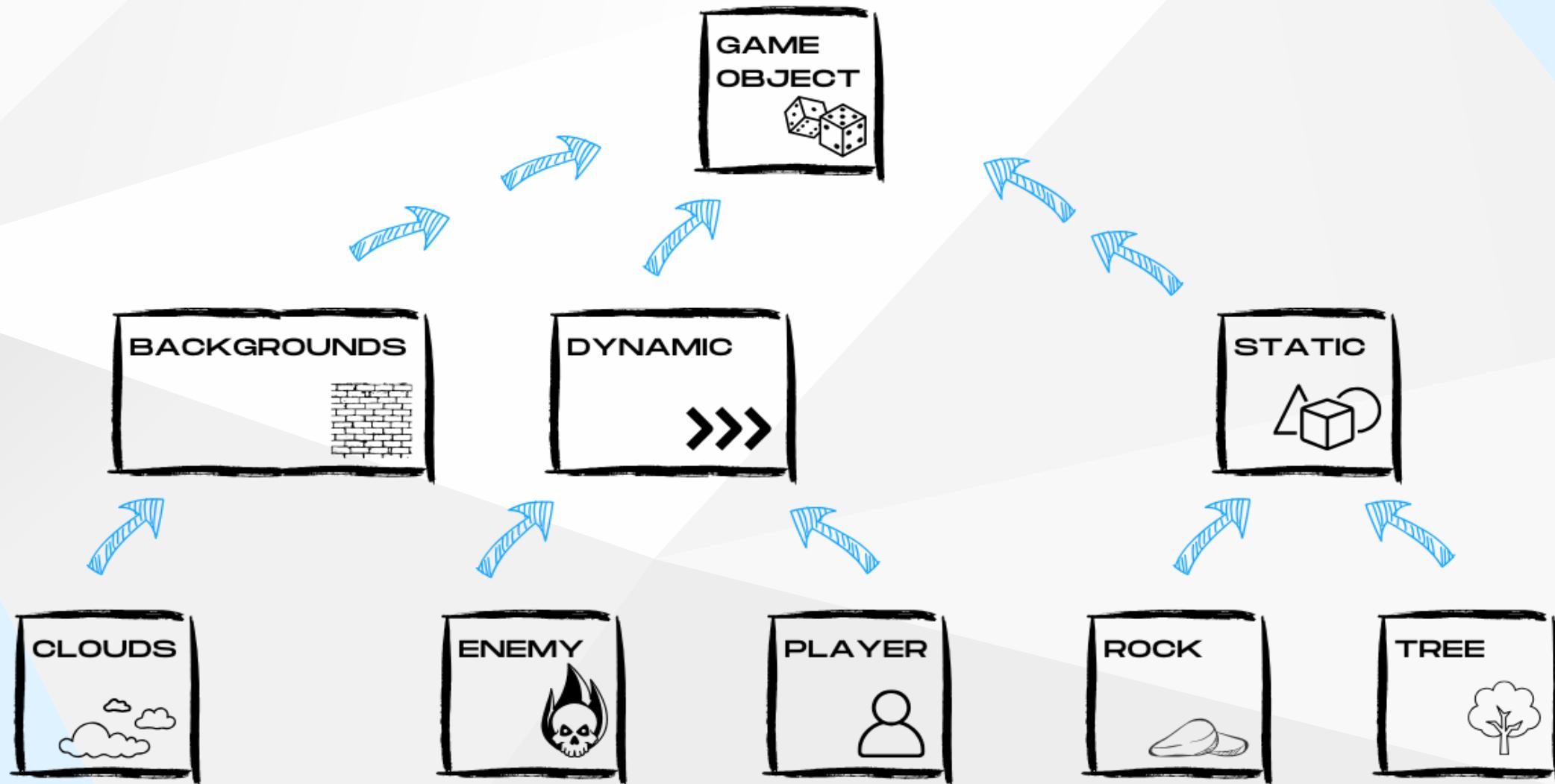
Wprowadzenie do React

Czym jest React?

- ▶ biblioteka, której głównym zadaniem jest ograniczenie liczby mutacji na strukturze DOM
- ▶ został zaprojektowany i wprowadzony na rynek przez Facebook
- ▶ React opiera się dość mocno na koncepcjach ze świata programowania funkcyjnego
- ▶ React służy do budowania dynamicznych i złożonych interfejsów użytkownika w sposób deklaratywny i modułowy
- ▶ struktura widoku i logika są trzymane w jednym miejscu (tylko w wykorzystaniem języka Javascript)
- ▶ React buduje widoki w architekturze komponentowej

Architektura komponentowa

- ◀ komponenty mogą być wykorzystane w wielu aplikacjach
- ◀ skraca czas budowy nowych aplikacji i obniża koszty projektu - poprzez wykorzystanie istniejących komponentów
- ◀ wykorzystując istniejące komponenty po pierwsze, nie musimy ich pisać od nowa, po drugie testować i dokumentować
- ◀ poszczególne komponenty mogą być tworzone równolegle
- ◀ poprzez dobre wyspecyfikowanie interfejsów poszczególnych komponentów można łatwo zlecać implementacje komponentów na zewnątrz
- ◀ wspiera tzw. modyfikowalność aplikacji - konkretna funkcjonalność skupia się w dedykowanych komponentach programowych, więc w przypadku konieczności wprowadzenia zmiany, z reguły proces ten dotyczy modyfikacji jednego lub co najwyżej kilku komponentów a nie całej aplikacji
- ◀ spójność (ang. coherence) komponentów wspiera modyfikowalność - im większa spójność tym łatwiej modyfikować aplikację



VDOM

- ◆ VirtualDOM to uproszczona reprezentacja struktury DOM (Document Object Model)
- ◆ React buduje drzewo Virtual DOM w sposób deklaratywny, a następnie przygotowuje na jego podstawie drzewo DOM w przeglądarce
- ◆ w przypadku wystąpienia zmian w strukturze, React na nowo przygotuje drzewo Virtual DOM dla zmienionych komponentów
- ◆ biblioteka ReactDOM porównuje aktualne i nowe drzewo Virtual DOM, a następnie aktualizuje tylko te miejsca, które wymagają zmiany
- ◆ VDOM należy traktować jako obiekt JS

- ◆ składnia przypominająca HTML pozwalająca na używanie rozwiniętej składni HTML w kodzie JS
- ◆ składnia wzbogaca "HTML" o możliwość użycia zmiennych i wyrażeń

Przykładowy kod:

```
const Section = <section>
  <h1>title</h1>
  <h2 className="subtitle" id="hook_to_title">daily</h2>
  <ul>
    <li>ts</li>
    <li>react</li>
  </ul>
</section>;
```

Przykładowy kod:

```
const section = {
  title: 'title',
  subtitle: 'daily',
  items: [
    {
      id: 1,
      name: 'ts',
    },
    {
      id: 2,
      name: 'react',
    },
  ],
};

const Section = <section>
  <h1>{section.title}</h1>
  <h2 className="subtitle" id="hook_to_title">{section.subtitle}</h2>
  <ul>
    {section.items.map(item => <li key={item.id}>{item.name}</li>)}
  </ul>
</section>;
```

Komponent w React

- ▶ komponent to podstawowy blok aplikacji Reactowej
- ▶ komponent to element w strukturze DOM, który jest zarządzany przez React'a
- ▶ Funkcja / Klasa komponentu zawiera kod Javascriptowy, który kontroluje wygląd i zachowanie elementu
- ▶ instancja komponentu to element, który został wyrenderowany przez React'a za pośrednictwem swojej klasy / funkcji
- ▶ komponenty mogą być zagnieżdżane w dowolne struktury, podobnie jak HTML czy XML
- ▶ możemy przekazać do komponentu dowolne dane jako atrybuty, w taki sam sposób jak w HTMLu
- ▶ w przeciwieństwie do HTMLa poprzez atrybuty możemy przekazywać również obiekty (a nie tylko stringi)
- ▶ przekazane w ten sposób parametry nazywają się właściwościami komponentu (ang. Component properties, w skrócie **props**)

Przykładowy kod:

```
// komponent funkcyjny
const Greetings = ({ name }) => {
  return <h1>Hello, {name}</h1>
}
```

Przykładowy kod:

```
// komponent klasowy
interface GreetingsProps {
  name: string;
}

class Greetings extends React.Component<GreetingsProps> {
  public render(): ReactNode {
    return <h1>Hello, {this.props.name}</h1>
  }
}
```

Create React App

Wrapper dla aplikacji Reactowej. CRA ukrywa szczegóły implementacyjne, w szczególności konfigurację bundler'a.

Przykładowy kod:

```
npx create-react-app my-app --template typescript
```

zadanie nr 3

- ◀ uruchom nowy projekt, wykorzystując create react app

```
npx create-react-app my-app --template typescript
```

- ◀ aplikacja będzie wyświetlała i zarządzała zadaniami do wykonania
- ◀ założmy, że każde **Todo** będzie spełniało następujący interfejs:

Przykładowy kod:

```
type Todo = {  
    title: string;  
    estimation: number;  
    priority: boolean;  
    description?: string;  
}
```

- ◀ przygotuj komponent **TodoList**, który będzie zawierał listę zadań
- ◀ przygotuj komponent **Todo**, który będzie reprezentacją poszczególnego zadania



react-redux

Biblioteka zawierająca specyficzną dla Reacta implementację Redux'a.

Przykładowy kod:

```
npm install react-redux
```

Store

- ◆ magazyn jest scentralizowany (cały stan aplikacji przechowywany jest w jednym obiekcie Javascriptowym)
- ◆ jednokierunkowy przepływ danych (zmiana tylko poprzez reducery)
- ◆ niezmienność stanu - stan jest niemutowalny
- ◆ każda zmiana tworzy nowy obiekt stanu zamiast modyfikowania istniejącego
- ◆ komponenty aplikacji mogą się subskrybować na zmiany stanu
- ◆ magazyn przechowuje dane w strukturze drzewistej

Przykładowy kod:

```
const store = {  
  user: {  
    id: 1,  
    name: "Jan Kowalski",  
    loggedIn: true  
  },  
  posts: [  
    { id: 1, title: "Post 1", content: "Lorem ipsum" },  
    { id: 2, title: "Post 2", content: "Dolor sit amet" }  
  ],  
  comments: [  
    { id: 1, postId: 1, text: "Super artykuł!" },  
    { id: 2, postId: 2, text: "Dzięki za wpis!" }  
  ]  
}
```

createStore()

- ▶ funkcja, która tworzy magazyn redux'owy
- ▶ jako argument `createStore` przyjmuje `reducer`

Przykładowy kod:

```
import { createStore } from 'redux';
import reducer from './reducer';

export const store = createStore(reducer);
```

...a jaki to ma typ?

- ▶ aby określić typ dla store potrzebujemy informacji o tym w jakim kształcie mamy stan
- ▶ typ dla stanu jest bezpośrednio połączony z naszymi reducerami
- ▶ utworzony `store` zwraca udostępnia nam również metodę `getState`

Przykładowy kod:

```
export type AppState = ReturnType<typeof store.getState>;
```

Preloaded state

- ▶ stan inicjujący, który przekazujemy jako drugi argument do funkcji `createStore`
- ▶ przydatny, gdy chcemy przekazać dane z serwera do naszego stanu lub przywrócić zserializowaną sesję
- ▶ kształt takiego obiektu, powinien być zgodny z reducerami i kluczami, do których są przypisane

Provider

- ◀ komponent, udostępniony z biblioteki `react-redux`, którego głównym zadaniem jest udostępnienie magazynu dla całej aplikacji (lub części, którą "owrapowuje")
- ◀ jako props `Provider` przyjmuje instancje magazynu

Przykładowy kod:

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import store from './store'
import App from './App'

const root = ReactDOM.createRoot(
  document.getElementById('root')
);

root.render(
  <Provider store={store}>
    <App />
  </Provider>,
)
```

zadanie nr 4

- ◀ doinstaluj do projektu [redux](#) oraz [react-redux](#)
- ◀ skonfiguruj magazyn
- ◀ dodaj [`<Provider>`](#) do swojej aplikacji

Przykładowy kod:

```
npm install redux react-redux
```



useDispatch

- ▶ hook dostarczony przez pakiet `react-redux`
- ▶ zwraca funkcję `dispatch`, która jest niezbędna do wysłania danych do reduktorów a następnie do magazynu

Selektory

- ◀ selektory to funkcje, które wybierają z obiektu stanu dane
- ◀ najczęściej nazywane są w taki sposób, aby łatwo określić, które dane będą zwracały z magazynu
- ◀ innymi słowy jest to dowolne funkcje, które przyjmują magazyn Redux (lub jego część) jako argument i zwracają dane oparte na tym stanie
- ◀ selektory nie muszą być pisane przy użyciu specjalnej biblioteki i nie ma znaczenia, czy napiszemy je jako funkcje strzałkowe, czy z wykorzystaniem słowa kluczowego `function`
- ◀ funkcja selektora powinna być czystą funkcją

useSelector

- ▶ hook, który przyjmuje jako argument selektor
- ▶ łączy przekazany selektor z magazynem i zwraca oczekiwane dane
- ▶ useSelector doda dodatkowo subskrypcję na magazyn i wywoła selektor za każdym razem, gdy akcja zostanie wysłana
- ▶ kiedy akcja zostanie zdispatch'owana hook `useSelector` dokona porównania referencji bieżącej i poprzedniej wartości zwróconej z selektora
- ▶ jeśli wartości będą różne, komponent zostanie zmuszony do przerenderowania

shallow vs strict

W Redux występują dwa sposoby porównywania obiektów. Porównywanie odbywa się podczas wybierania danych z magazynu w komponencie. Re-render wydarzy się, gdy obiekty porównywane się różnią.

Strict - występuje przy komponentach funkcyjnych (`useSelector`)

Shallow - występuje przy użyciu funkcji `connect()` - najczęściej komponenty klasowe

	Strict	Shallow
Co porównujemy?	Referencję obiektu	Wartości pierwszego poziomu (bez analizowania struktury zagnieżdzonej)
Jak głęboko porównujemy	Brak	Tylko pierwsza wartość
Zagnieżdżone obiekty	Ignorowane	Nie bierze pod uwagę zawartości
Przykład porównania	<code>====</code>	Funkcja, która iteruje się po właściwościach i porównuje je

Przykładowy kod:

```
function shallowEqual(obj1, obj2) {  
    if (Object.keys(obj1).length !== Object.keys(obj2).length) {  
        return false;  
    }  
    for (const key in obj1) {  
        if (obj1[key] !== obj2[key]) {  
            return false;  
        }  
    }  
    return true;  
}  
  
const obj1 = { a: 1, b: 2 };  
const obj2 = { a: 1, b: 2 };  
const obj3 = { a: 1, b: 3 };  
  
shallowEqual(obj1, obj2); // true  
shallowEqual(obj1, obj3); // false
```

useSelector z płytkim porównaniem?

Możemy zmienić porównywanie obiektów `strict` na `shallow` w `useSelector` poprzez przekazanie funkcji porównującej w sposób `shallow` jako drugi argument do wywołania `useSelector`. Redux dostarcza funkcję `shallowEqual`, której możemy użyć.

Przykładowy kod:

```
import { shallowEqual, useSelector } from 'react-redux'

// bezpośrednio jako drugi argument
const selectedData = useSelector(selectorReturningObject, shallowEqual);

// lub jako `equalityFn` do argumentu z opcjami
const selectedData = useSelector(selectorReturningObject, {
  equalityFn: shallowEqual,
});
```

zadanie nr 5

- ◀ przygotuj akcje i kreatory akcji do zmiany, dodania i usunięcia zadania
(możesz wykorzystać akcje z pierwszego zadania)
- ◀ przygotuj reducer do powyższych akcji oraz połącz go z magazynem
- ◀ do reducera dodaj `initialState` w postaci kilku zadań
- ◀ na liście zadań dodaj akcję usunięcia zadania
- ◀ użyj selektora do pobrania danych z listy, tak aby po usunięciu zadania lista się odświeżała

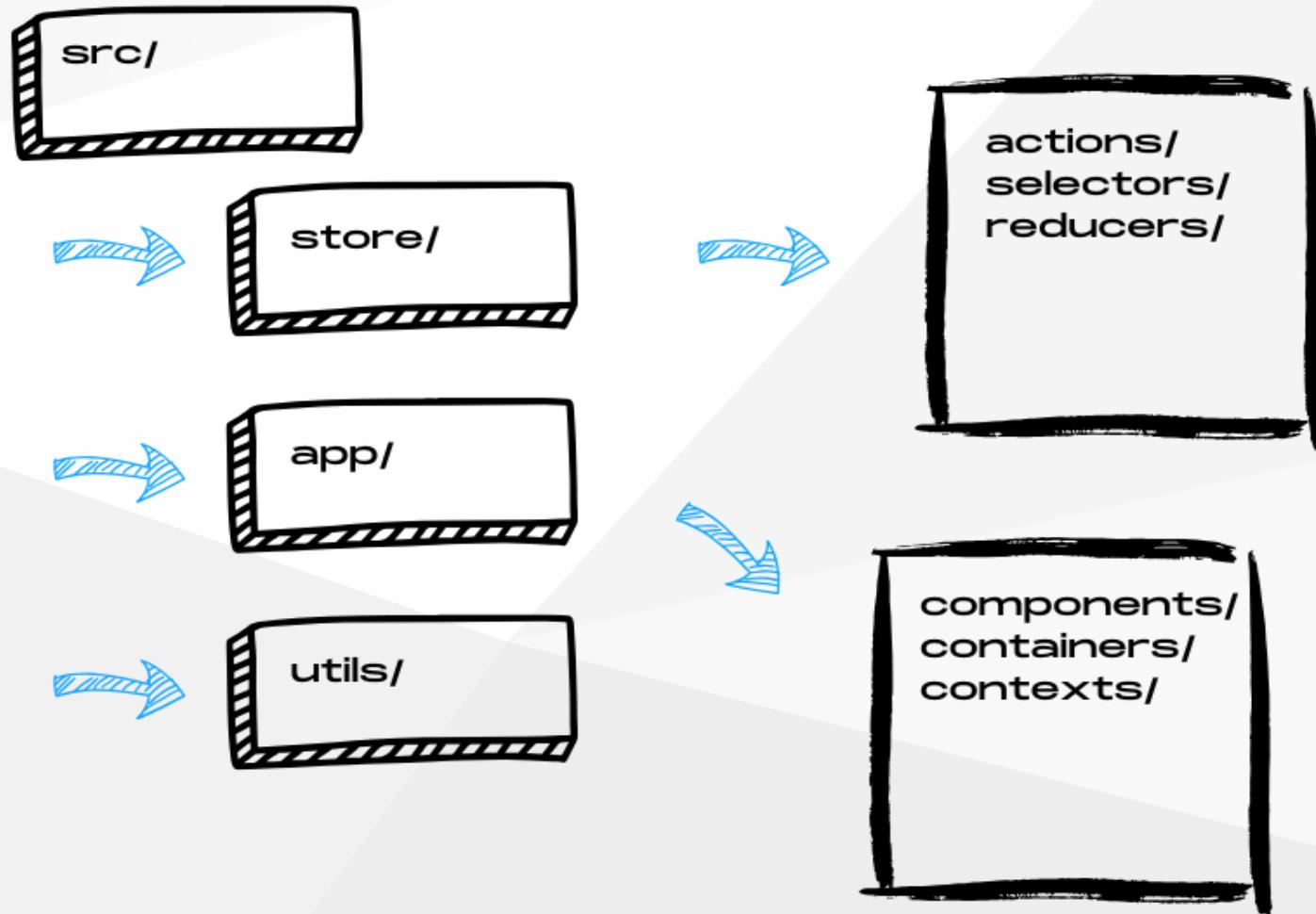


Architektura aplikacji Redux

- ◀ podejście do architektury aplikacji jest kilka - najpopularniejsze to: architektura oparta o containers i components, architektura oparta o slices, duck duck pattern
- ◀ w obu podejściach ustawienia Reduxa staramy się trzymać odseparowane od reszty aplikacji
- ◀ przy porządkowaniu Reduxa warto pamiętać o odpowiednim zbudowaniu struktury katalogów, ponieważ z czasem będzie miało to duży wpływ na koszt utrzymania aplikacji

Components i containers

- ▶ w tym podejściu wydzielamy dwa rodzaje komponentów - komponenty logiki oraz komponenty widoku
- ▶ komponenty widoku nie powinny mieć nic wspólnego z Reduxem, dane otrzymują bezpośrednio od rodziców
- ▶ komponenty logiki, czyli containers - mają możliwość pobierania danych z Reduxa i przekazywania ich dalej do komponentów widoku poprzez `props`
- ▶ najczęściej wraz z komponentami logiki pojawia się plik `connect.js`, który zawiera metody łączące reduxa i reacta



Duck duck pattern

Duck Duck Pattern to wzorzec organizacji kodu Redux, który koncentruje się na przechowywaniu całego kodu Redux dla określonej funkcji lub domeny w jednym pliku, zwanym "kaczką".

Plik `duck` zawiera akcje, kreatory akcji oraz reducery dla określonej domeny / funkcjonalności.

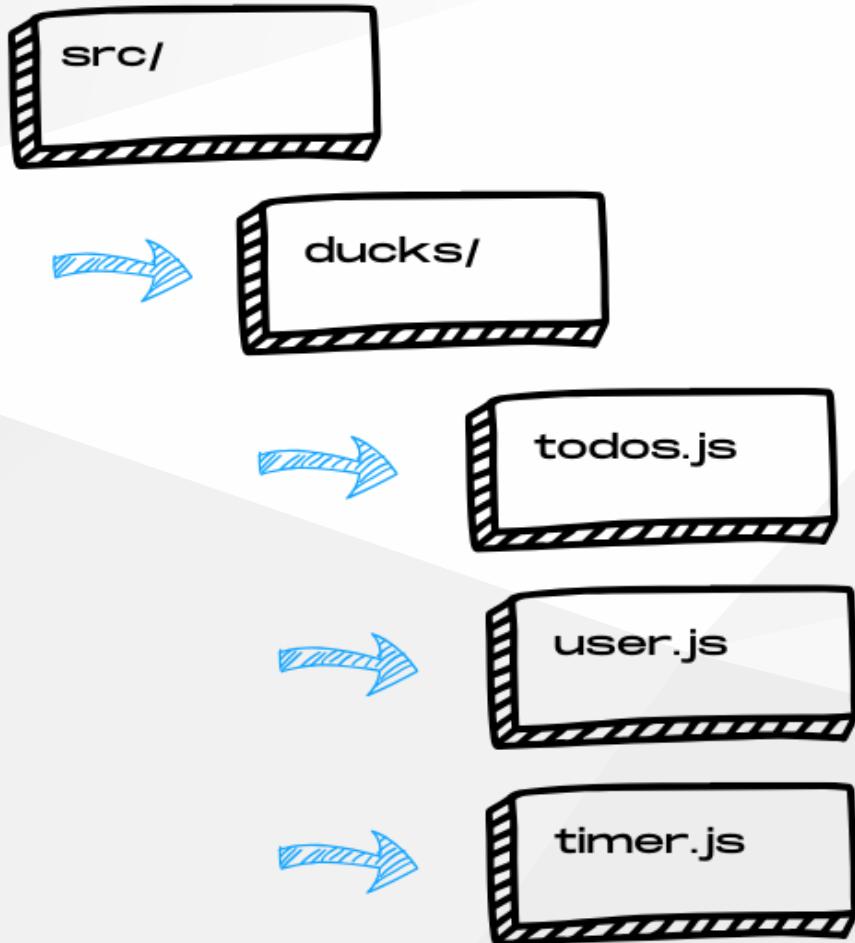
Jeśli coś kwacze jak kaczka i wygląda jak kaczka, to znaczy, że jest kaczką.

innymi słowy:

Jeśli plik zawiera cały kod Redux dla określonej funkcjonalności, to jest to "kaczką".

zalety wzorca kaczuszki

- ◀ upraszcza strukturę kodu - cały kod Reduxa w jednym miejscu
- ◀ wspomaga reużywalność - trzymanie wszystkiego w jednym pliku ułatwia korzystanie w wielu różnych miejscach
- ◀ łatwiejsze testowanie

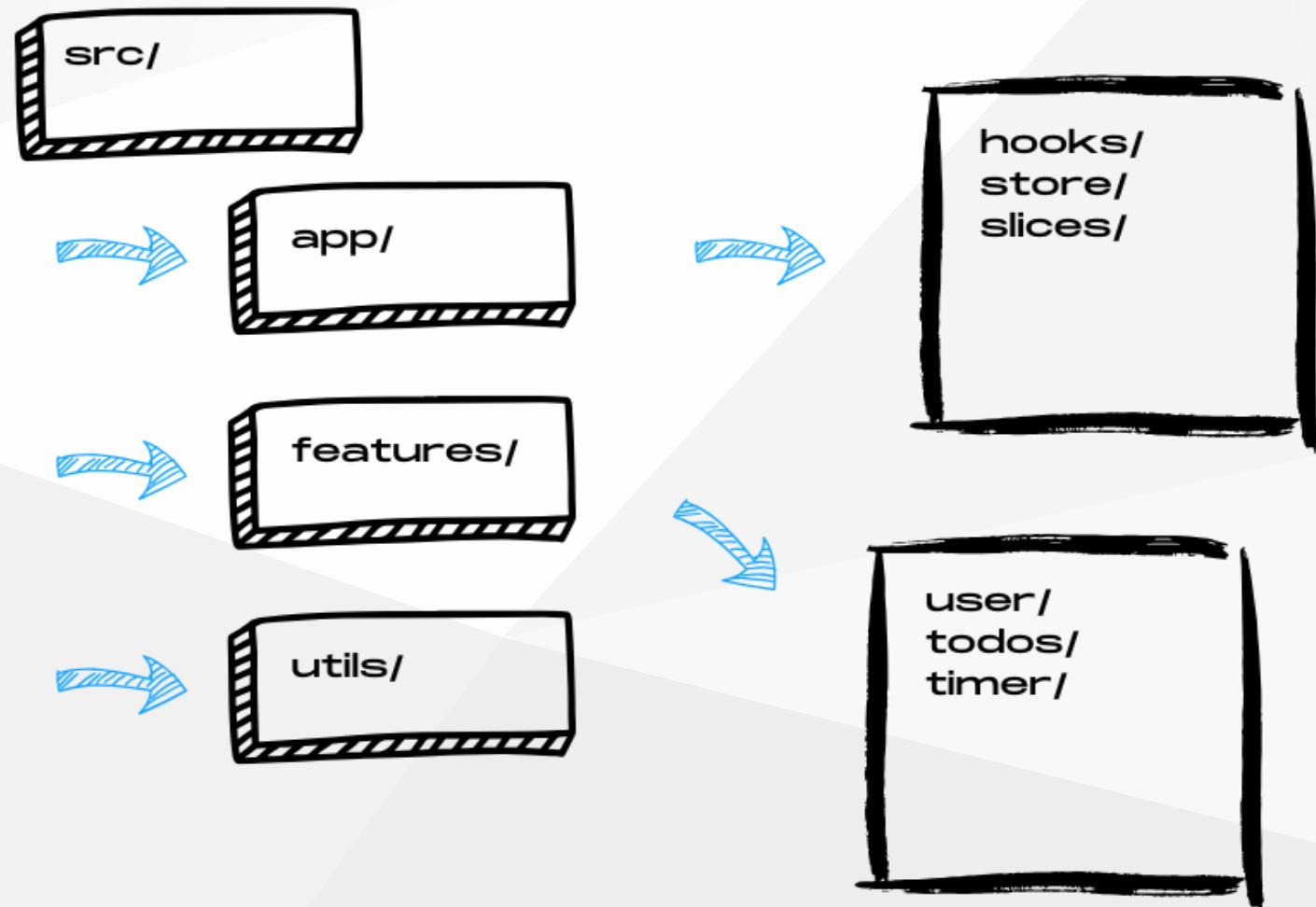


Struktura projektu feature-based

- ▶ zakłada, że aplikacja dzieli się na domeny / funkcjonalności, z których każda stanowi "samodzielna aplikację"
- ▶ każdy feature zawiera własne reducery / akcje / komponenty / widoki
- ▶ funkcjonalności nie przenikają się pomiędzy sobą

Slices

- ◀ slice to nowe podejście do budowania aplikacji Reduxowej, opiera się o podział aplikacji na domeny biznesowe
- ◀ slice to dodatkowa abstrakcja, która wydziela dla jakiegoś obszaru (domeny / części funkcjonalnej) naszej aplikacji logikę, reduktory i wszystkie związane z tym elementy
- ◀ nazwa pochodzi od dzielenia głównego reducer'a na mniejsze części - tzw. plasterki (ang. slices) stanu
- ◀ najłatwiej zaimplementować wykorzystując [redux/toolkit](#)



Przykładowy kod:

```
export const counterSlice = createSlice({  
  name: 'counter',  
  initialState,  
  reducers: {  
    increment: (state) => {  
      // co tutaj jest nie tak?  
      state.value += 1;  
    },  
    decrement: () => {  
      // co tutaj jest nie tak?  
      state.value -= 1;  
    },  
  },  
)
```

Mutowalny stan w createSlice

- ▶ w Redux Toolkit możemy wprowadzić logikę "mutującą" w reducerze, ponieważ pod spodem użyta jest biblioteka Immer, która wykrywa zmiany w "tymczasowym magazynie" i na jego podstawie tworzy całkiem nowy niemutowany stan zawierający już zmiany

wiele reducerów

- dołączenia reducerów ze sobą możemy użyć funkcji `combineReducers`
- funkcja pomocnicza, która przekształca pojedyncze reducery w jednego reducera, składającego się z mniejszych - przekazanych do niego
- wynikowy połączony reduktor wywołuje każdy składowy reduktor za każdym razem, gdy akcja jest wysyłana
- takie podejście umożliwia podzielenie logiki reduktora na oddzielne funkcje, z których każda niezależnie zarządza własnym wycinkiem stanu

Przykładowy kod:

```
// rootReducer.js
import { combineReducers } from "redux";
import { counterReducer } from "./reducer";
import { todoReducer } from "./todoReducer";

export const rootReducer = combineReducers({
  counter: counterReducer,
  todos: todoReducer,
});
```

Przykładowy kod:

```
// store.js
import { createStore } from "redux";
import { rootReducer } from "./rootReducer";

export const store = createStore(rootReducer);
```

zadanie nr 6

- ◀ uporządkuj swoją aplikację zgodnie z podejściem | [containers i components](#)
- ◀ dodaj reducery, akcje i selektory dla obsługi użytkownika
- ◀ w komponencie App dodaj przycisk, który po kliknięciu zaloguje użytkownika - akcja logowania powinna zapisać użytkownika w magazynie
- ◀ dodaj opcję wylogowywania
- ◀ utwórz prosty komponent, który wyświetli dane zalogowanego użytkownika
- ◀ pamiętaj o połączeniu reducerów za pomocą [combineReducers](#)



Connect

- ▶ funkcja, której główne zadanie to połączenie magazynu Redux z komponentami React
- ▶ connect przyjmuje 4 argumenty: `mapStateToProps` , `mapDispatchToProps` , `mergeProps` , `options`
- ▶ funkcja connect to klasyczny przykład funkcji Higher Order Component
- ▶ connect nie modyfikuje przekazanego jej komponentu, natomiast zwraca nowy, który owrapowuje ten do niej przekazany
- ▶ UWAGA! Redux wspiera connect do wersji Redux 8.x, ale zalecają przenieść się całkiem na api oparte o hook'i

Higher Order Component

- ▶ HOC to funkcja, która przyjmuje komponent jako argument
- ▶ HOC służy do dodawania dodatkowych funkcjonalności do komponentów w sposób reużywalny, bez modyfikowania ich wewnętrznej implementacji
- ▶ HOC nie modyfikuje przekazanego komponentu, a zwraca nowy, ulepszony komponent
- ▶ Higher Order Component to wariacja na temat Higher Order Function (fundament programowania funkcyjnego)

mapStateToProps

- ▶ funkcja, która jeśli zostanie przekazana do funkcji connect, doda do komponentu subskrypcję na zmiany w magazynie
- ▶ oznacza to, że za każdym razem, gdy magazyn zostanie zaktualizowany funkcja `mapStateToProps` zostanie wywołana
- ▶ wynikiem funkcji `mapStateToProps` powinien być zawsze obiekt, który zostanie zmergowany do propsów komponentu, który został przekazany do funkcji connect
- ▶ jeśli nie chcemy subskrybować się na zmiany w magazynie możemy przekazać `null` lub `undefined` do connect

Przykładowy kod:

```
const mapStateToProps = (state, ownProps) => ({  
  todos: state.todos,  
  todo: state.todos[ownProps.id],  
})
```

mapDispatchToProps

- ▶ drugi argument dla funkcji connect
- ▶ może być funkcją, obiektem lub null'em
- ▶ jeśli będzie **funkcją**, wówczas otrzyma dispatch jako argument, powinna wówczas zwrócić obiekt, który zostanie zmergowany do propsów komponentu wywołanego z connect
- ▶ jeśli będzie **obiektem** z kreatorami akcji, wówczas redux wywoła na każdym kreatorze akcji funkcję `bindActionCreators`, która połączy kreator akcji z funkcją dispatch

Przykładowy kod:

```
const createMyAction = () => ({ type: 'MY_ACTION' });

const mapDispatchToProps = (dispatch, ownProps) => {
  const boundActions = bindActionCreators({ createMyAction }, dispatch)
  return {
    dispatchPlainObject: () => dispatch({ type: 'MY_ACTION' }),
    dispatchActionCreatedByActionCreator: () => dispatch(createMyAction()),
    ...boundActions,
    // you may return dispatch here
    dispatch,
  }
}
```

Przykładowy kod:

```
import { addTodo, deleteTodo, toggleTodo } from './actionCreators'

const mapDispatchToProps = {
  addTodo,
  deleteTodo,
  toggleTodo,
}

export default connect(null, mapDispatchToProps)(TodoApp)
```

mergeProps

- ▶ funkcja, która łączy ze sobą propsy stanu, propsy typu dispatch oraz propsy komponentu przekazanego
- ▶ zwraca obiekt `props`, który jest przekazywany do komponentu przekazanego do connect
- ▶ jako trzeci argument funkcji connect jest opcjonalny

options

Opcja	Typ	Opis
context	Object	Obiekt contextu, powinien być przekazany również do <code><Provider /></code>
areStatesEqual	Function	Porównuje przychodzący stan magazynu z jego poprzednią wartością
areOwnPropsEqual	Function	Porównuje przychodzące propsy z ich poprzednią wartością
areStatePropsEqual	Function	Porównuje wynik <code>mapStateToProps</code> z poprzednią wartością
areMergedPropsEqual	Function	Porównuje wynik <code>mergeProps</code> z jego poprzednią wartością
forwardRef	boolean	Jeśli zostanie ustawiony na true, dodanie <code>ref</code> do komponentu wrappera, zwróci instancję komponentu owrappowanego.

Przykładowy kod:

```
import { login, logout } from './actionCreators';
import { Login } from './Login';

const mapState = (state) => state.user;
const mapDispatch = { login, logout };

export default connect(mapState, mapDispatch)(Login);
```

zadanie nr 7

- ◀ dodaj plik connect do komponentu logowania i wylogowywania użytkownika
- ◀ za pomocą funkcji connect połącz state i dispatch ze swoim komponentem



Enhancer

- ▶ jest to funkcja, która rozszerza lub modyfikuje możliwości magazynu, np. poprzez dodanie middleware'a, obsługę dodatkowych narzędzi czy implementację mechanizmów logowania
- ▶ działa na poziomie tworzenia store'a
- ▶ wpływa na sposób w jaki redux przetwarza akcje lub aktualizuje stan

Przykładowy kod:

```
import { createStore } from 'redux'
import rootReducer from './reducer'

const sayHiOnDispatch = (createStore) => (reducer, preloadedState, enhancer) => {
  // Tworzymy bazowy store
  const store = createStore(reducer, preloadedState, enhancer);

  // Zachowujemy oryginalną funkcję dispatch
  const originalDispatch = store.dispatch;

  // Nadpisujemy dispatch, aby dodawał logowanie
  store.dispatch = (action) => {
    console.log("hi"); // Logowanie za każdym razem, gdy wywoływany jest dispatch
    return originalDispatch(action); // Wywołanie oryginalnego dispatch
  };
}

const store = createStore(rootReducer, undefined, sayHiOnDispatch);

export default store;
```

wiele enhancerów

- aby użyć wielu enhancerów, możemy wykorzystać funkcję `compose` redux'a, której głównym zadaniem jest zmergowanie ze sobą wiele enhancer'ów

Przykładowy kod:

```
import { createStore, compose } from 'redux'  
import rootReducer from './reducer'  
import {  
  sayHiOnDispatch,  
  includeMeaningOfLife  
} from './exampleAddons/enhancers'  
  
const composedEnhancer = compose(sayHiOnDispatch, includeMeaningOfLife);  
  
const store = createStore(rootReducer, undefined, composedEnhancer);  
  
export default store;
```

zadanie nr 8

- ◀ napisz enhancer, który będzie logował w console.log komunikat: "Action <TYP AKCJI> dispatched, my Lord!"



Middleware

- ◆ enhancery są świetnym rozwiązaniem, ponieważ potrafią nadpisywać i/lub zastępować metody magazynu: `dispatch` , `getState` i `subscribe`
- ◆ w większości przypadków potrzebujemy jednak dostosować zachowanie `dispatch`
- ◆ do modyfikacji dispatcherów służy mechanizm middleware
- ◆ w frameworkach takich jak Koa czy Expres middleware to kod, który można umieścić między funkcją odbierającą żądanie a funkcją generującą odpowiedź
- ◆ middleware w Express lub Koa mogą dodawać nagłówki CORS, logowanie błędów i ostrzeżeń, kompresję
- ◆ middleware można komponować w łańcuchy
- ◆ w jednym projekcie można korzystać z wielu niezależnych middleware'ów innych dostawców

applyMiddleware

- jest to specjalny enhancer dostarczany przez Redux

Przykładowy kod:

```
import { createStore, applyMiddleware } from 'redux'  
import rootReducer from './reducer'  
import { print1, print2, print3 } from './exampleAddons/middleware'  
  
const middlewareEnhancer = applyMiddleware(print1, print2, print3)  
  
// możemy pominąć drugi argument preloadedState jeśli nie jest potrzebny i od razu przekazać enhancery  
const store = createStore(rootReducer, middlewareEnhancer)  
  
export default store
```

Przykładowy kod:

```
const exampleMiddleware = (storeAPI) => {
  return function wrapDispatch(next) {
    return function handleAction(action) {

      // w tym miejscu możemy umieścić naszą logikę:
      // - przekazać akcję dalej za pomocą funkcji next(action)
      // - zrestartować uruchomiony proces / pipeline za pomocą storeAPI.dispatch(action)
      // - pobrać bieżący stan magazynu
      return next(action);
    }
  }
}
```

...rozbijmy to na części pierwsze

- ◀ `exampleMiddleware` - zewnętrzna funkcja, która jest naszym middleware'm, przyjmuje parametr `storeApi`
- ◀ `storeAPI` jest obiektem, który ma dostęp do funkcji `dispatch` oraz `getState`, są to dokładnie te same funkcje, które są dostępne w `store`
- ◀ jeśli użyjemy `dispatch` to akcja zostanie przesłana na początek middleware'a
- ◀ `wrapDispatch` - funkcja, która otrzymuje jako parametr funkcję `next`, wywołanie `next` przechodzi o następnego middleware'a w sekwencji, jeśli aktualny middleware był ostatni - `next` jest w zasadzie wywołaniem `dispatch`
- ◀ `handleAction` - wewnętrzna funkcja, która otrzymuje `action` jako argument

zewnętrzny middleware

Przykładowy kod:

```
import { applyMiddleware, createStore } from 'redux';
import logger from 'redux-logger';

const store = createStore(
  reducer,
  applyMiddleware(logger)
)
// lub
import { createLogger } from 'redux-logger'

const logger = createLogger({
  // ...options
});

const store = createStore(
  reducer,
  applyMiddleware(logger)
);
```

zadanie nr 9

- zainstaluj `react-logger` w swojej aplikacji za pomocą `npm i -D react-logger`
- przygotuj jego implementację



Typy

- biblioteka `redux` udostępnia również typy: `Middleware` oraz `Reducer`, które możemy wykorzystać do otypowania funkcji middleware i reducerów

Przykładowy kod:

```
import { Reducer, ActionFromReducer } from 'redux';
import { LOGIN_USER, LoginUserAction, LOGOUT_USER, LogoutUserAction } from './actions';

type Action = LoginUserAction | LogoutUserAction;

type UserState = { id: number, name: string } | null

export const userReducer: Reducer<UserState, Action> = (state = null, action: Action) => {};
```

zadanie nr 10

- ◀ napisz middleware'a, który będzie dodawał do stanu związanego z użytkownikiem licznik ile zadań usunął, ile utworzył, a ile zrealizował
- ◀ założymy, że obiekt user w magazynie, będzie miał kształt:

Przykładowy kod:

```
type UserState = {  
  user: {  
    name: string;  
    email: string;  
  },  
  tasks: {  
    done: number;  
    deleted: number;  
    added: number;  
  }  
}
```

- ◀ dodaj `<button>` do zadania, który będzie odpowiadał za oznaczenie zadania jako wykonane



Operacje asynchroniczne w cyklu reduxowym

- cykl redux'owy jest synchroniczny, aby wykonać zapytanie asynchroniczne musimy wprowadzić dodatkową logikę w postaci middleware'a
- do dyspozji na rynku mamy podejścia `redux-saga` oraz `redux-thunk`, ale nic nie stoi na przeszkodzie, żeby napisać samodzielnie taki middleware
- customowy middleware do obsługi zapytań asynchronicznych daje nam większą kontrolę
- przechowywanie zapytań do zewnętrznego API pozwala nam zapisywać w magazynie również informacje o aktualnym stanie zapytania: `STARTED` , `SUCCESS` , `ERROR`

Dlaczego nie zarządzać asynchronicznością w komponentach?

- ◀ jeśli każdy komponent zarządza swoimi zapytaniami, różne części aplikacji mogą być niespójne - w Reduxie dane są wspólne i zsynchronizowane
- ◀ zarządzanie stanem asynchronicznego zapytania w komponentach wymaga więcej kodu i może prowadzić do duplikacji logiki (np. pokazywanie spinnera lub obsługa błędów w wielu miejscach)
- ◀ zarządzanie asynchronicznością w Redux pozwala oddzielić logikę biznesową od logiki prezentacyjnej, komponenty stają się prostsze, ponieważ nie mają wpływu i wiedzy o pochodzeniu danych

Dlaczego nie zarządzać asynchronicznością w Redux?

- ◀ Redux jest zaprojektowany do zarządzania stanem aplikacji, a nie do obsługi wszystkich procesów biznesowych
- ◀ włączenie asynchroniczności może go przeładować i uczynić akcje i reduktory zbyt skomplikowanymi
- ◀ nie każde zapytanie asynchroniczne musi być widoczne w całej aplikacji (część zapytań powinno być dostępnych tylko dla jednego lub kilku komponentów)
- ◀ zbyt duża ilość zapytań w Redux może prowadzić do niepotrzebnych / nadmiarowych re-renderów
- ◀ wprowadzenie dodatkowych middleware'ów zwiększa złożoność kodu
- ◀ istnieją dedykowane rozwiązania do obsługi asynchroniczności: `React Query`, `Apollo Client` czy `SWR` - obsługują cache, ponawianie zapytań i synchronizację danych
- ◀ łamanie zasady **Separation of Concerns** (**Separacja odpowiedzialności**)

zadanie nr 11

- ◀ do każdego todo dodaj `form` oraz pole `input [name="title"]` oraz przycisk `button [type="submit"]` "Dodaj zadanie"
- ◀ po wypełnieniu formularza wyślij akcję dodającą nowe zadanie do store'a
- ◀ przygotuj nowego middleware'a, który do wszystkich nowo dodanych zadań doda estymację z losową wartością od 1 do 100



REDUX_DEVTOOLS_EXTENSION

- ▶ wtyczka do przeglądarki służąca do debugowania reduxa
- ▶ pokazuje wszystkie wykonane akcje, aktualny stan drzewa w zależności od wykonanej akcji
- ▶ ułatwia debuggowanie przepływu danych w Redux

zadanie nr 12

- ◀ dodaj do swojego projektu __REDUX_DEVTOOLS_EXTENSION__
- ◀ prześledź flow całej aplikacji w tym narzędziu



121

Przykładowy kod:

```
import { compose, applyMiddleware } from 'redux';
import { thunk } from 'redux-thunk';

declare global {
  interface Window {
    __REDUX_DEVTOOLS_EXTENSION_COMPOSE__: typeof compose;
  }
}

const composeEnhancers =
  (window as Window).__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ || compose;

export const store = createStore(
  rootReducer,
  composeEnhancers(applyMiddleware(thunk)),
);
```

redux-thunk

- ◆ słowo "thunk" oznacza "fragment kodu, który wykonuje pewną opóźnioną pracę"
- ◆ zamiast wykonywać pewną logikę teraz, możemy napisać funkcję, która może zostać użyta później
- ◆ `redux-thunk` jest najbardziej prymitywnym (najprostszym) z dostępnych middleware'ów
- ◆ `redux-thunk` to middleware dla redux'a
- ◆ funkcja thunk przyjmuje jako argumenty dwie metody `dispatch` oraz `getState`
- ◆ thunk zakłada, że możemy wykonywać pracę w trybie sync lub async - a wykonanie jednej operacji `dispatch` lub `getState` może nastąpić w dowolnym momencie

Jak działają thunki?

Przykładowy kod:

```
function yell (text) {
  console.log(text + '!')
}

yell('cześć') // 'cześć!'

// Wersja lazy (lub "thunked")
function thunkedYell (text) {
  return function thunk () {
    console.log(text + '!')
  }
}

const thunk = thunkedYell('cześć') // nic się nie dzieje
// ...już za moment
thunk() // 'cześć!'
```

14 linii kodu

Przykładowy kod:

```
function createThunkMiddleware(extraArgument) {
  return ({ dispatch, getState }) => next => action => {
    if (typeof action === 'function') {
      return action(dispatch, getState, extraArgument);
    }

    return next(action);
  };
}

const thunk = createThunkMiddleware();
thunk.withExtraArgument = createThunkMiddleware;

export default thunk;
```

Przykładowy kod:

```
// kreator akcji:  
const asyncLogin = () =>  
  axios.get('/api/auth/me')  
    .then(res => res.data)  
    .then(user => {  
      // jak dostać się do tego obiektu?  
    })  
  
// gdzieś w komponencie:  
store.dispatch(asyncLogin()) // błąd! `asyncLogin()` jest obietnicą a nie akcją
```

Przykładowy kod:

```
// kreator akcji:  
import store from '../store'  
  
const simpleLogin = user => ({ type: LOGIN, user })  
  
const asyncLogin = () =>  
  axios.get('/api/auth/me')  
    .then(res => res.data)  
    .then(user => {  
      store.dispatch(simpleLogin(user))  
    })  
  
// gdzieś w komponencie:  
asyncLogin()
```

Przykładowy kod:

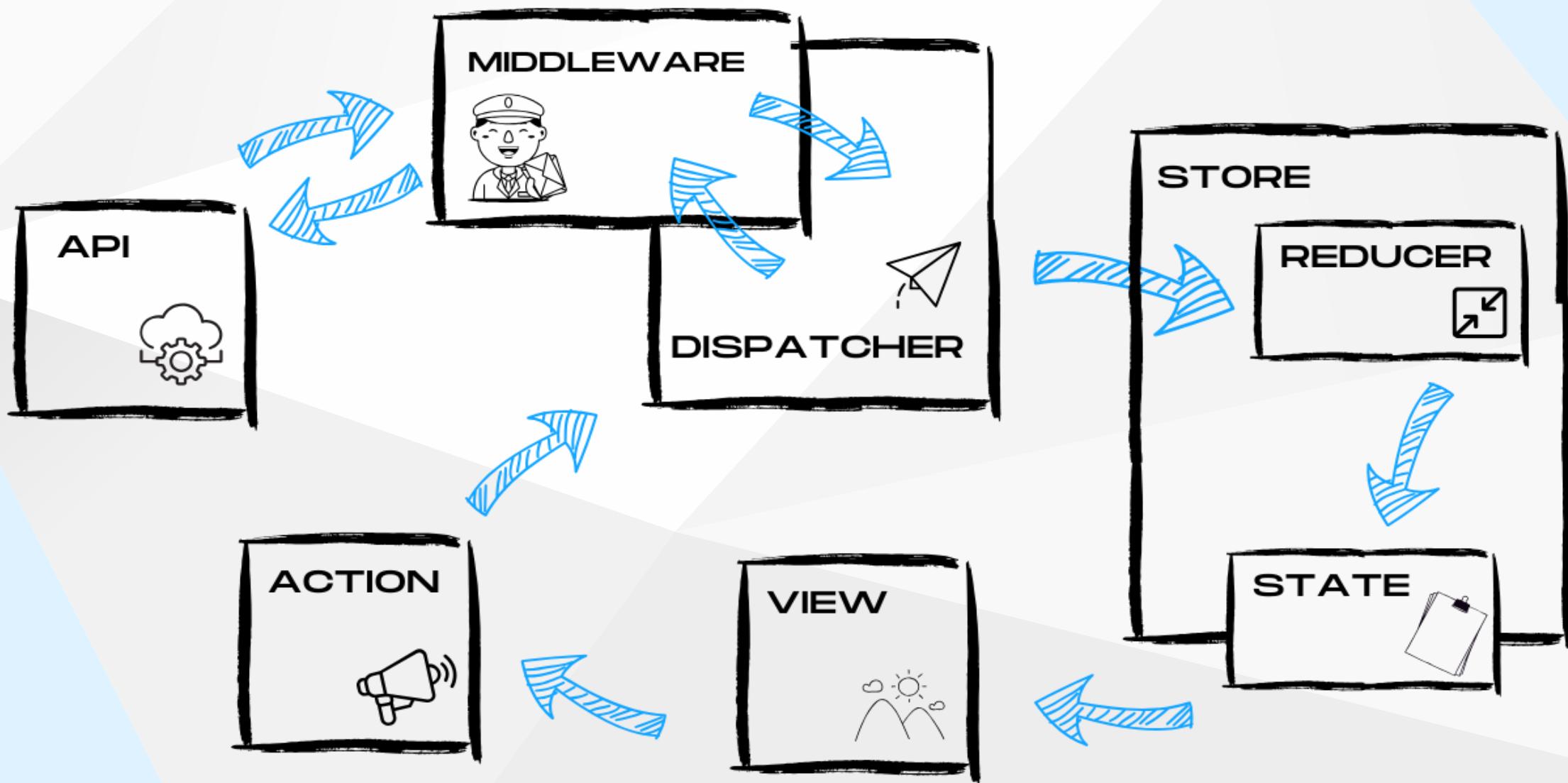
```
// kreator akcji:  
import store from '../store';  
  
const simpleLogin = user => ({ type: LOGIN, user })  
  
const thunkedLogin = () =>      // kreator akcji po wywołanie...  
  () =>                      // ...zwraca thunk'a, który po wywołaniu...  
    axios.get('/api/auth/me')   // ... wykonuje zapytanie asynchroniczne  
    .then(res => res.data)  
    .then(user => {  
      store.dispatch(simpleLogin(user))  
    })  
  
// gdzieś w komponencie:  
store.dispatch(thunkedLogin()) // dispatch'ujemy thunk'a do magazynu  
  
// Thunk `() => axios.get...` nie został jeszcze wywołany.
```

Co zatem sprawia, że powyższy kod działa?

- ◀ thunk to nie jest magiczne rozwiązanie
- ◀ aby działał musimy dodać dodatkową logikę w postaci middleware'a, który będzie sprawdzał czy otrzymany obiekt jest akcją czy funkcją
- ◀ jeśli funkcją wówczas ją wywoła, jeśli akcją przekaże ją dalej do reducera

Przykładowy kod:

```
actionOrThunk =>
  typeof actionOrThunk === 'function'
    ? actionOrThunk(dispatch, getState)
    : passAlong(actionOrThunk);
```



Dependency injection w Redux Thunk

Przykładowy kod:

```
import axios from 'axios'

const store = createStore(
  reducer,
  applyMiddleware(thunk.withExtraArgument(axios))
)

const thunkedLogin = () =>
  (dispatch, getState, axios) => // thunk ma teraz dostęp do axios'a
    axios.get('/api/auth/me')
      .then(res => res.data)
      .then(user => {
        dispatch(simpleLogin(user))
      });

```

Przykładowy kod:

```
export const FETCH_USERS_REQUEST = 'FETCH_USERS_REQUEST';
export const FETCH_USERS_SUCCESS = 'FETCH_USERS_SUCCESS';
export const FETCH_USERS_FAILURE = 'FETCH_USERS_FAILURE';

interface FetchUsersRequestAction {
  type: typeof FETCH_USERS_REQUEST;
}

interface FetchUsersSuccessAction {
  type: typeof FETCH_USERS_SUCCESS;
  payload: User[];
}

interface FetchUsersFailureAction {
  type: typeof FETCH_USERS_FAILURE;
  payload: string;
}

export type UserActionTypes =
  | FetchUsersRequestAction
  | FetchUsersSuccessAction
  | FetchUsersFailureAction;
```

Przykładowy kod:

```
const initialState: UserState = {
  users: [],
  loading: false,
  error: null,
};

export const userReducer = (
  state = initialState,
  action: UserActionTypes
): UserState => {
  switch (action.type) {
    case FETCH_USERS_REQUEST:
      return { ...state, loading: true, error: null };
    case FETCH_USERS_SUCCESS:
      return { ...state, loading: false, users: action.payload };
    case FETCH_USERS_FAILURE:
      return { ...state, loading: false, error: action.payload };
    default:
      return state;
  }
};
```

Przykładowy kod:

```
import { Dispatch } from 'redux';
import { UserActionTypes } from '../actionTypes';
import {
  FETCH_USERS_REQUEST,
  FETCH_USERS_SUCCESS,
  FETCH_USERS_FAILURE,
} from '../actionTypes';
import { User } from '../types.d';

export const fetchUsers = () => {
  return async (dispatch: Dispatch<UserActionTypes>) => {
    dispatch({ type: FETCH_USERS_REQUEST });
    try {
      const response = await fetch('https://jsonplaceholder.typicode.com/users');
      const data: User[] = await response.json();
      dispatch({ type: FETCH_USERS_SUCCESS, payload: data });
    } catch (error) {
      dispatch({ type: FETCH_USERS_FAILURE, payload: error.message });
    }
  };
};
```

zadanie nr 13

- ◀ zainstaluj w swoim projekcie [redux-thunk](#) ([npm](#) i [redux-thunk](#))
- ◀ dodaj do reduxa akcje, kreatory akcji i reducer'a związane z pobieraniem danych z API
- ◀ zadanie polega na pobraniu do reduxa komentarzy z endpointu <https://jsonplaceholder.typicode.com/comments> i wyświetlenie ich poniżej listy zadań

Przykładowy kod:

```
type Comment = {  
  postId: number;  
  id: number;  
  name: string;  
  email: string;  
  body: string;  
}
```



redux-saga

◀ **redux-saga** to middleware do wykonywania operacji asynchronicznych, opiera się o generatory (generatory) i efektów

Efekt	Opis
call	wywoływanie funkcji asynchronicznych
put	wysyłanie akcji do reduktora
take	oczekiwanie na określoną akcję
select	odczytywanie danych ze stanu
fork	uruchamianie niezależnych zadań równoległych

Przykładowy kod:

```
import { call, put, takeEvery } from 'redux-saga/effects';
import { fetchData } from './api'; // Funkcja wywołująca API

// Saga do pobierania danych
function* fetchDataSaga(action) {
  try {
    const data = yield call(fetchData, action.payload);
    yield put({ type: 'FETCH_SUCCESS', data });
  } catch (error) {
    yield put({ type: 'FETCH_FAILURE', error });
  }
}

// Główna saga
export function* watchFetchData() {
  yield takeEvery('FETCH_REQUEST', fetchDataSaga);
```

Configuring a Redux store is too complicated.

I have to add a lot of packages to get Redux to do anything useful

Redux requires too much boilerplate code

-- *Społeczność*

Redux Toolkit

- ▶ pakiet przygotowany przez zespół redux'a, którego głównym zadaniem jest uproszczenie konfiguracji i podziału aplikacji
- ▶ częścią Redux Toolkit jest także RTK Query - biblioteka do pobierania i buforowania danych
- ▶ dodaje sporo abstrakcji i kilka dodatkowych funkcji pomocniczych
- ▶ `npm i @reduxjs/toolkit`

API Redux Toolkit

Metoda	Opis
configureStore()	opakowuje <code>createStore</code> , aby zapewnić uproszczone opcje konfiguracji i dobre ustawienia domyślne. Może automatycznie łączyć reduktory, dodaje middleware'y Redux, domyślnie obejmuje redux-thunk i umożliwia korzystanie z rozszerzenia Redux DevTools.
createReducer()	pozwala na dostarczenie obiektu akcji do reducerów, zamiast pisania instrukcji switch. Ponadto automatycznie używa biblioteki Immer, aby umożliwić pisanie prostszych niezmiennych aktualizacji przy użyciu logiki mutującej
createAction()	generuje funkcję kreatora akcji dla podanego ciągu typu akcji
createSlice()	przyjmuje obiekt reducera, nazwę slice'a i wartość stanu początkowego, a następnie automatycznie generuje reducer dla slice'a z odpowiadającymi mu kreatorami akcji i typami akcji

Metoda	Opis
combineSlices()	łączy wiele slice'ów w jeden reducer i umożliwia wykorzystanie lazy loading slice'ów po inicjalizacji
createAsyncThunk	przyjmuje typ akcji i funkcję, która zwraca Promise'a, na tej podstawie generuje thunk, który wysyła akcje pending/fulfilled/rejected bazujące na Promise
createEntityAdapter	przygotowuje zestaw reduktorów i selektorów wielokrotnego użytku do zarządzania znormalizowanymi danymi w magazynie
createSelector	funkcja pomocnicza z biblioteki Reselect, re-exportowana

configureStore

Przykładowy kod:

```
import { configureStore } from '@reduxjs/toolkit';
import userReducer from './features/users/userSlice';

export const store = configureStore({
  reducer: rootReducer,
  middleware: (getDefaultMiddleware) => getDefaultMiddleware().concat(middlewares),
  enhancers: (getDefaultEnhancer) => getDefaultEnhancer().concat([sayHiOnDispatchEnhancer]),
  devTools: true,
});
```

Przykładowy kod:

```
const initialState: UserState = {};  
  
const userSlice = createSlice({  
  name: 'user',  
  initialState,  
  reducers: {  
    logoutUser: (state) => {  
      state = {};  
      return state;  
    },  
  },  
});
```

Przykładowy kod:

```
export const {
  updateAddedTasks,
  updateDeletedTasks,
  loginUser,
  logoutUser,
} = userSlice.actions;

export default userSlice.reducer;
```

Przykładowy kod:

```
const commentsSlice = createSlice({
  name: 'comments',
  initialState,
  reducers: {
    },
    extraReducers: (builder) => {
      builder.addCase(fetchComments.pending, (state) => {
        state.loading = true;
        state.error = null;
      });
      builder.addCase(fetchComments.fulfilled, (state, action) => {
        state.loading = false;
        state.comments = action.payload;
      });
      builder.addCase(fetchComments.rejected, (state, action) => {
        state.loading = false;
        state.error = action.error as string;
      });
    }
});
```

zadanie nr 14

- ◀ doinstaluj `@reduxjs/toolkit`
- ◀ zastąp `createStore` funkcją `configureStore`
- ◀ przepisz akcje i reducery na podejście slice
- ◀ przepisz pobieranie komentarzy jako `extraReducers`



Przykładowy kod:

```
const initialState: UserState = {};  
  
const userSlice = createSlice({  
  name: 'user',  
  initialState,  
  reducers: {  
    },  
    selectors: {  
      getUser: (state: UserState) => state,  
    }  
});
```

zadanie nr 15

- ◀ dopisz do slice'ów selektory i wyeksportuj je
- ◀ zastąp selektory w komponentach selektorami ze slice'ów



zadanie nr 16

- ▶ przepisz comments na podejście ze slice'ami
- ▶ wykorzystaj `createAsyncThunk` i `extraReducers` do wykonania tego zadania



Memoizacja

- ◀ technika optymalizacji w programowaniu polegająca na przechowywaniu wyników funkcji dla określonych zestawów argumentów, aby uniknąć wielokrotnego przeliczania tych samych wyników w przyszłości
- ◀ memoizacja poprawia wydajność, szczególnie w przypadku funkcji, które są kosztowne obliczeniowo lub są często wywoływane z tymi samymi argumentami
- ◀ przy pierwszym wywołaniu funkcji z danym zestawem argumentów, wynik obliczeń jest zapisywany w specjalnej strukturze danych, zazwyczaj w obiekcie
- ◀ przy kolejnych wywołaniach z tymi samymi argumentami funkcja nie wykonuje obliczeń, lecz zwraca zapisany wcześniej wynik

Gdzie przyda się memoizacja?

Poniższy kod nie jest zbyt efektywny. Funkcja `fibonacci` wykonuje się rekurencyjnie wiele razy z tymi samymi argumentami.

Czas: 797ms

Przykładowy kod:

```
function fibonacci(n) {  
    if (n <= 1) return n;  
    return fibonacci(n - 1) + fibonacci(n - 2);  
}  
  
console.log(fibonacci(40));
```

Po dodaniu memoizacji

Czas: 0.09ms

Przykładowy kod:

```
function fibonacciMemo() {  
  const memo = {};  
  
  function fibonacci(n) {  
    if (n in memo) return memo[n]; // Sprawdź, czy wynik jest zapisany  
    if (n <= 1) return n;  
    memo[n] = fibonacci(n - 1) + fibonacci(n - 2); // Zapisz wynik  
    return memo[n];  
  }  
  
  return fibonacci;  
}  
  
const fibonacci = fibonacciMemo();  
console.log(fibonacci(40));
```

Dane z selektorów

- ▶ obiekt zwracany przez selektor w Reduxie zmienia referencję za każdym razem, gdy jest tworzony na nowo
- ▶ jeśli selektor zwraca nowy obiekt jego referencja będzie za każdym razem inna, nawet jeśli dane w stanie Redux się nie zmieniły

Przykładowy kod:

```
const selector = (state) => {
  return { value: state.someValue }; // Zwracamy nowy obiekt
};

const obj1 = selector({ someValue: 42 });
const obj2 = selector({ someValue: 42 });

console.log(obj1 === obj2); // false – nowa referencja
```

Memoizacja selektorów

- ◆ `createSelector` przechowuje wynik poprzedniego obliczenia
- ◆ jeśli dane wejściowe się nie zmieniły, zwraca zapamiętany wynik, zamiast tworzyć nowy obiekt

Przykładowy kod:

```
import { createSelector } from 'reselect';

// Prosty selector
const getSomeValue = (state) => state.someValue;

// Memoizowany selector
const getTransformedValue = createSelector(
  [getSomeValue],
  (someValue) => {
    return { value: someValue }; // Tworzymy obiekt, ale memoizujemy wynik
  }
);

const state = { someValue: 42 };
const obj1 = getTransformedValue(state);
const obj2 = getTransformedValue(state);

console.log(obj1 === obj2); // true – ta sama referencja, bo dane się nie zmieniły
```

Jak brak memoizacji selektorów wpływa na React?

- ◀ jeśli komponent React otrzymuje nową referencję obiektu jako props, React uznaje, że props się zmienił, i renderuje komponent ponownie
- ◀ użycie memoizowanych selektorów redukuje liczbę niepotrzebnych renderów w React

Memoizacja w React

- ▶ `React.memo` - memoizacja komponentów
- ▶ `useMemo` - memoizacja wartości w komponencie React
- ▶ `useCallback` - memoizacja funkcji w komponencie React

React.memo

- ◀ **React.memo** dba o to, aby komponent renderował się tylko, gdy zmienią się jego propsy
- ◀ **React.memo** porównuje propsy za pomocą porównania płytkego (shallow comparison)
- ◀ dla propsów referencyjnych (np. tablice, obiekty, funkcje) sprawdzana jest tylko ich referencja, a nie rzeczywista zawartość

Przykładowy kod:

```
const Component = ({ name }) => {
  return <h2>{name}</h2>
}

export const MemoizedComponent = React.memo(Component);
```

Reselect w Redux Toolkit

- ▶ funkcja `reselect` została dołączona do Redux Toolkit
- ▶ można ją zimportować z Redux Toolkit jako `createSelector`

Przykładowy kod:

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

// klasyczny selector
const getUser = (state) => state.user;

// Memoizowany selektor
const getUserInfo = createSelector(
  [getUser],
  (user) => {
    if (!user) return null;
    return {
      fullName: `${user.firstName} ${user.lastName}`,
      email: user.email,
      isAdult: user.age >= 18,
    };
  }
);
```

useMemo

Przykładowy kod:

```
const UserProfile = () => {
  const dispatch = useDispatch();
  // Używamy useSelector do pobrania stanu z Redux
  const user = useSelector((state) => state.user);

  // Zastosowanie useMemo do memoizacji przekształconych danych
  const userInfo = useMemo(() => {
    if (!user) return null;

    // Przykładowa transformacja danych
    return {
      fullName: `${user.firstName} ${user.lastName}`,
      email: user.email,
      isAdult: user.age >= 18,
    };
  }, [user]); // Tylko przekształcone dane, kiedy user się zmienia

  return (
    <div>tutaj będą dane użytkownika</div>
  );
};
```

zadanie nr 17

- ◀ przygotuj zmemoizowany selector do pobierania danych o użytkowniku
- ◀ na poziomie komponentu przekształć obiekt `user.tasks`, tak aby policzył ile zadań zostało wykonanych / usuniętych / dodanych w sumie
- ◀ użyj `useMemo` do memoizacji nowego obiektu



why-did-you-render

- ◀ narzędzie do monitorowania i diagnostyki w React, które pomaga zrozumieć, dlaczego komponenty Reacta zostały ponownie wyrenderowane
- ◀ przydatne do optymalizacji aplikacji, ponieważ pozwala zidentyfikować niepotrzebne renderowanie komponentów
- ◀ `why-did-you-render` śledzi zmiany w propsach i stanie komponentów React, aby pokazać, czy i dlaczego komponenty zostały ponownie wyrenderowane
- ◀ `npm install @welldone-software/why-did-you-render`

zalety korzystania z why-did-you-render

- ◀ identyfikacja niepotrzebnych renderów
- ◀ optymalizacja wydajności
- ◀ lepsze zrozumienie działania aplikacji

Lazy loading reducerów

- ▶ Lazy loading reducerów w Reduxie to technika, która pozwala na załadowanie tylko tych reduktorów, które są potrzebne w danym momencie, zamiast ładować wszystkie reduktory na początku aplikacji
- ▶ szczególnie przydatne w dużych aplikacjach, które mogą mieć rozbudowaną logikę i dużo reducerów
- ▶ lazy loading pozwala na optymalizację rozmiaru początkowego bundle'a aplikacji i przyspieszenie czasu ładowania

Przykładowy kod:

```
import { createStore, combineReducers } from 'redux';

// Początkowe reduktory, które są załadowane od razu
const rootReducer = combineReducers({
  auth: authReducer,
});

const store = createStore(rootReducer);

// Funkcja do dynamicznego dodawania reducerów
store.injectReducer = (key, reducer) => {
  store.replaceReducer(
    combineReducers({
      ...store.reducerManager.getReducers(),
      [key]: reducer
    })
  );
};

export default store;
```

Przykładowy kod:

```
import store from './store';

// Tworzymy managera reducerów (możesz to zrobić w osobnym pliku)
store.reducerManager = {
  reducers: {},
  getReducers() {
    return this.reducers;
  },
  add(key, reducer) {
    this.reducers[key] = reducer;
  }
};
```

Przykładowy kod:

```
import React, { useEffect } from 'react';
import { useDispatch } from 'react-redux';
import { fetchUserData } from './actions';
import { userReducer } from './reducers/userReducer';

export const UserProfilePage = () => {
  const dispatch = useDispatch();

  useEffect(() => {
    // dynamiczne doładowanie reducera
    store.injectReducer('user', userReducer);

    // Pobieramy dane użytkownika po załadowaniu reducera
    dispatch(fetchUserData());
  }, [dispatch]);

  return (
    <div>
      <h1>User Profile</h1>
    </div>
  );
};
```

You might not need Redux

-- Dan Abramov, twórca Reduxa, 2016

Context API

Przykładowy kod:

```
const ThemeContext = React.createContext('light');

class App extends React.Component {
  render() {
    // Use a Provider to pass the current theme to the tree below.
    // Any component can read it, no matter how deep it is.
    // In this example, we're passing "dark" as the current value.
    return (
      <ThemeContext.Provider value="dark">
        <Toolbar />
      </ThemeContext.Provider>
    );
  }
}
```

Przykładowy kod:

```
function Content() {
  return (
    <ThemeContext.Consumer>
      {theme => (
        <UserContext.Consumer>
          {user => (
            <ProfilePage user={user} theme={theme} />
          )}
        </UserContext.Consumer>
      )}
    </ThemeContext.Consumer>
  );
}

function CountDisplay() {
  const {count} = React.useContext(CountContext)
  return <div>{count}</div>
}
```

Problemy z Context API

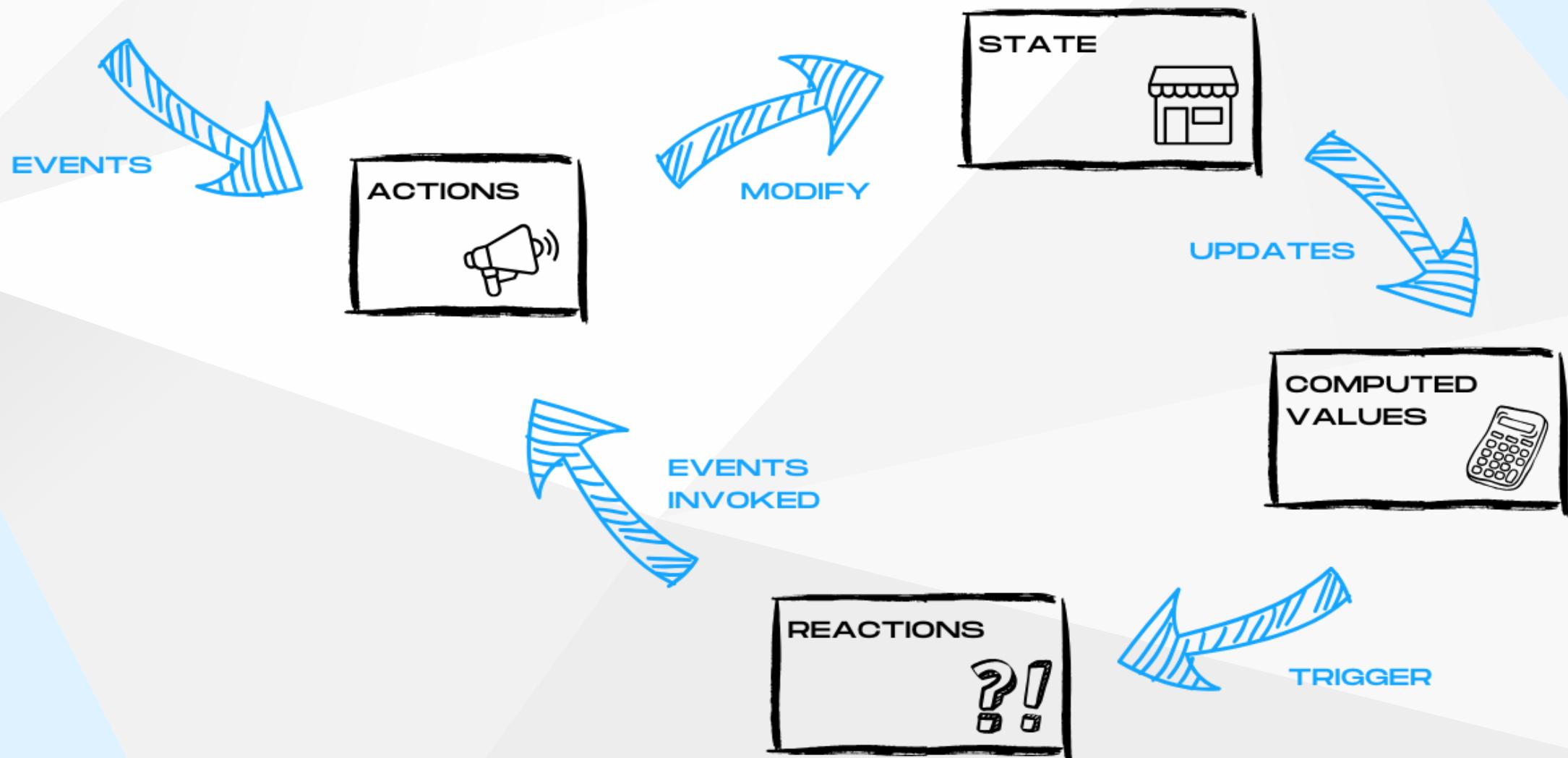
- ◀ kiedy stan w kontekście się zmienia, wszystkie komponenty, które korzystają z tego kontekstu, zostaną ponownie wyrenderowane, nawet jeśli ich dane wcale się nie zmieniły
- ◀ zarządzanie złożonym stanem w Context API może być trudne i nieefektywne - każda zmiana jakiejkolwiek części stanu może prowadzić do ponownego renderowania całego drzewa komponentów
- ◀ trudności z typowaniem w TypeScript
- ◀ w dużych aplikacjach, kiedy działa wiele kontekstów nieodpowiednie zarządzanie nimi może prowadzić do chaosu - używanie zbyt wielu kontekstów może utrudnić zarządzanie danymi i ich przepływem
- ◀ przekazanie funkcji do kontekstu (np. funkcji do aktualizacji stanu), może prowadzić do problemów z referencjami, każda zmiana funkcji spowoduje ponowne renderowanie komponentów

Redux vs Context

Context	Redux
jest częścią biblioteki React	zewnętrzna biblioteka
wymagana minimalna konfiguracja	wymagana dodatkowa konfiguracja aby zintegrować Redux z React
zaprojektowany dla danych statycznych (takich, które nie są często odświeżane lub aktualizowane) np.: theme, dane użytkownika, proste wartości słownikowe	świętne nadaje się do pracy zarówno z danymi statycznymi, jak i dynamicznymi
logika widoku i logika zarządzania stanem są w jednym miejscu	lepsza organizacja kodu (separacja logiki UI i logiki zarządzania stanem)
debuggowanie może być trudne	posiada bardzo dobre narzędzie do debugowania
niezbyt nadaje się do przechowywania dużych ilości danych, ale może przechowywać funkcje	nadaje się do przechowywania dużych ilości danych

MobX

- ◆ MobX to biblioteka open source'owa do zarządzania stanem
- ◆ nie jest powiązana z jakimkolwiek frameworkiem
- ◆ stan aplikacji odnosi się do całego modelu aplikacji i może zawierać różne typy danych, w tym tablice, liczby i obiekty
- ◆ akcje są metodami, które manipulują i aktualizują stan
- ◆ akcje można powiązać z modułem obsługi zdarzeń JavaScript, aby mieć pewność, że zdarzenie w interfejsie użytkownika je wyzwoli
- ◆ magazyn MobX jest reaktywny, a zatem reaguje na zmiany



Przykładowy kod:

```
class PetOwnerStore {  
    pets = [];  
    owners = [];  
}
```

Przykładowy kod:

```
import { action, observable, reaction } from 'mobx';

// Klasa do zarządzania danymi / store
class Data {
  @observable
  value = 0;

  @action
  incrementValue = () => this.value += 1;

  constructor() {
    // Tworzymy `reaction` która nasłuchuje na zmianę wartości właściwości i loguje wiadomość do consoli, gdy nastąpi zmiana
    reaction(() => this.value, () => console.log('Value Changed'));
  }
}

// Tworzymy instację naszego magazynu
const myData = new Data();

// Kiedy funkcja zostanie wywołana – `Value Changed` zostanie wyświetlone w konsoli
myData.incrementValue();
```

Przykładowy kod:

```
@inject("color")
@observer
class Button extends React.Component {
  render() {
    return <button style={{ background: this.props.color }}>{this.props.children}</button>
  }
}

class Message extends React.Component {
  render() {
    return (
      <div>
        {this.props.text} <Button>Delete</Button>
      </div>
    )
  }
}

class MessageList extends React.Component {
  render() {
    const children = this.props.messages.map(message => <Message text={message.text} />)
    return (
      <Provider color="red">
        <div>{children}</div>
      </Provider>
    )
  }
}
```

Przykładowy kod:

```
const TodoView = observer(({ todo }) => <div>{todo.title}</div>)
```

zustand

- Zustand jest mocno inspirowany architekturami Flux oraz Redux
- do magazynu można wrzucić wszystko: wartości prymitywne, obiekty, funkcje
- magazyn jest hookiem, z którego możemy wyciągnąć zarówno dane, jak i akcje
- funkcja `set` łączy stan
- Zustand dostarcza również podejście `redux-like`, które pozwala tworzyć reducery i dispatchery w sposób znany z klasycznej wersji Redux'a

Przykładowy kod:

```
import { create } from 'zustand';

export const useStore = create(set => ({
  population: 0,
  increasePopulation: () => set(state => ({ population: state.population + 1000 })),
  destroyHumans: () => set({ population: 0 })
  changePopulation: newPopulation => set({ population: newPopulation })
}))
```

Przykładowy kod:

```
function PopulationCounter() {  
  const humans = useStore((state) => state.population)  
  return <h1>{humans} ludzi na ziemi</h1>  
}  
  
function PopulationControls() {  
  const increasePopulation = useStore((state) => state.increasePopulation)  
  return <button onClick={increasePopulation}>+1000</button>  
}
```

Testowanie reducerów

- reducer w Reduxie to funkcja, która przyjmuje dwa argumenty: bieżący stan i akcję, a następnie zwraca nowy stan
- testowanie polega na upewnieniu się, że reducer prawidłowo przetwarza różne akcje i zwraca poprawny stan

Przykładowy kod:

```
const initialState = {  
  count: 0,  
};  
  
const counterReducer = (state = initialState, action) => {  
  switch (action.type) {  
    case 'INCREMENT':  
      return { count: state.count + 1 };  
    case 'DECREMENT':  
      return { count: state.count - 1 };  
    case 'RESET':  
      return { count: 0 };  
    default:  
      return state;  
  }  
};  
  
export default counterReducer;
```

Przykładowy kod:

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

describe('counterReducer', () => {
  it('should return the initial state', () => {
    expect(counterReducer(undefined, {})).toEqual({ count: 0 });
  });

  it('should handle INCREMENT', () => {
    const action = { type: 'INCREMENT' };
    const initialState = { count: 0 };
    const nextState = counterReducer(initialState, action);
    expect(nextState).toEqual({ count: 1 });
  });

  it('should handle DECREMENT', () => {
    const action = { type: 'DECREMENT' };
    const initialState = { count: 1 };
    const nextState = counterReducer(initialState, action);
    expect(nextState).toEqual({ count: 0 });
  });
});
```

Testowanie akcji

- ◀ testowanie akcji w Reduxie jest równie ważne jak testowanie reducerów, ponieważ akcje są odpowiedzialne za inicjowanie zmian w stanie aplikacji
- ◀ akcje mogą być synchroniczne (zwykłe obiekty) lub asynchroniczne (np. przy użyciu middleware, takiego jak redux-thunk lub redux-saga)
- ◀ testowanie akcji polega na sprawdzeniu, czy akcja jest wysyłana w odpowiednim formacie oraz, w przypadku akcji asynchronicznych, czy wykonuje się prawidłowo

Przykładowy kod:

```
export const increment = () => ({
  type: 'INCREMENT'
});

export const setUser = (user) => ({
  type: 'SET_USER',
  payload: user
});
```

Przykładowy kod:

```
import { increment, setUser } from './actions';

describe('Redux Actions', () => {
  it('should create an action to increment', () => {
    const expectedAction = { type: 'INCREMENT' };
    expect(increment()).toEqual(expectedAction);
  });

  it('should create an action to set user', () => {
    const user = { name: 'Mateusz', age: 30 };
    const expectedAction = {
      type: 'SET_USER',
      payload: user
    };
    expect(setUser(user)).toEqual(expectedAction);
  });
});

export const fetchData = () => async (dispatch) => {
  const data = await fetch('/api/data').then((res) => res.json());
  dispatch({ type: 'SET_DATA', payload: data });
};

const dataReducer = (state = { data: [] }, action) => {
  switch (action.type) {
    case 'SET_DATA':
      return { ...state, data: action.payload };
    default:
      return state;
  }
}
```

zadanie nr 18

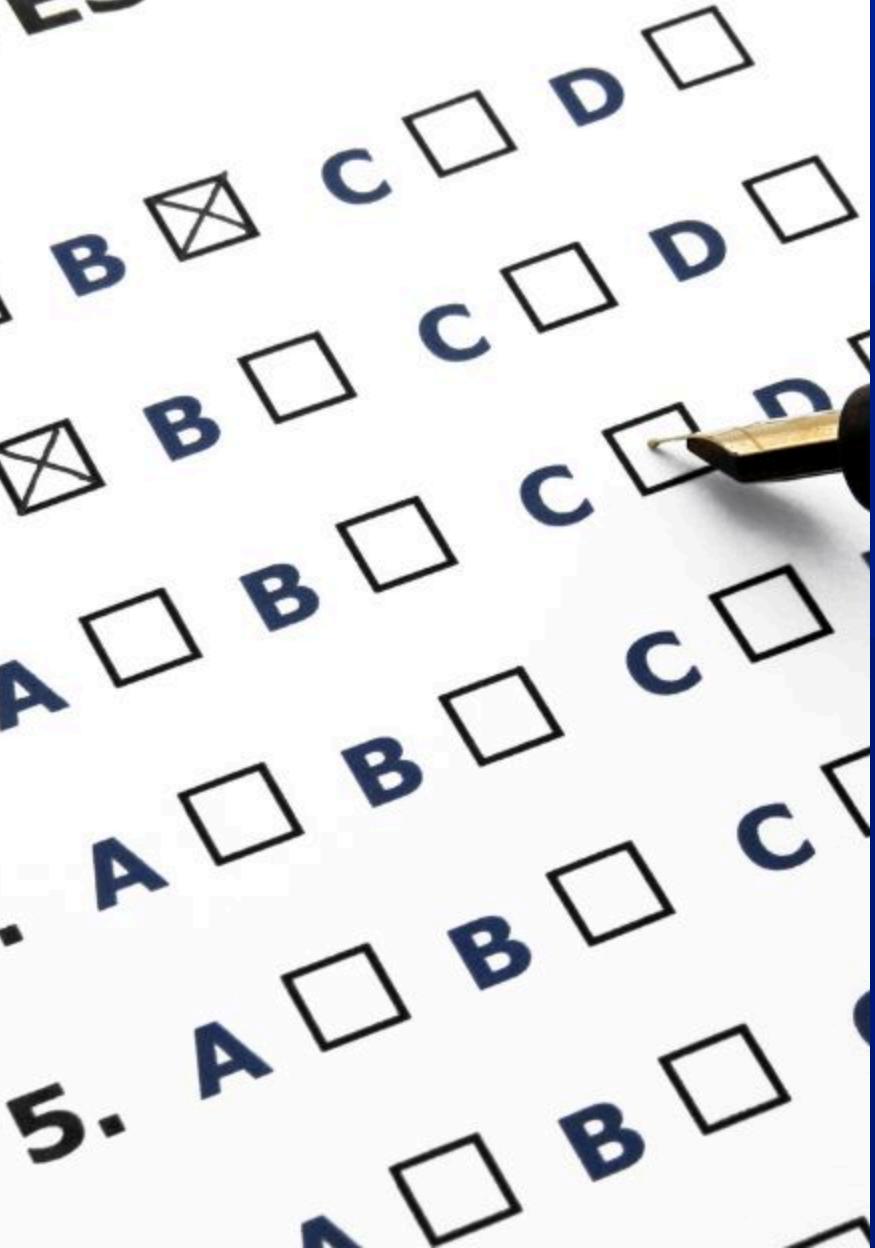
- ◀ napisz testy jednostkowe dla swoich akcji i reducerów ze slice'a tasks / todos



Dodatkowe pytania?



EST



KAHOOT

Ankieta

czyli jak mi poszło?

sages.link/196965



Dziękuję za uwagę

Mateusz Jabłoński

mail@mateuszjablonski.com

mateuszjablonski.com

mateuszjablonski.com