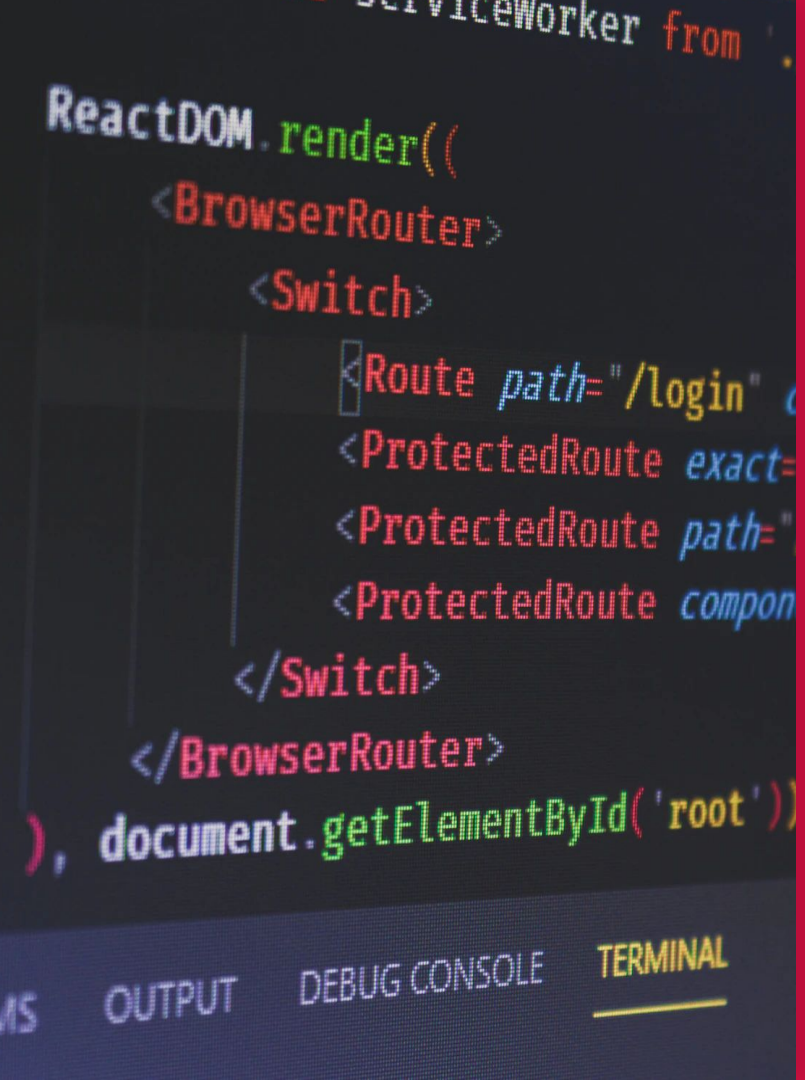


//DEV:ENV
O PROGRAMOWANIU BEZ KACA

REACT w dużej skali. Jak tworzyć aplikacje większe niż TODO list?

//DEV:ENV

O PROGRAMOWANIU BEZ KACA

Adrian Piętka

EMPHIE SOLUTIONS

Mateusz Konieczny

BRAINHUB

Mateusz Drażyk

BRAINHUB

Agenda

- Aplikacja tutorialowa vs aplikacja rzeczywista
- Współdzielenie logiki
- Zarządzanie stanem
- Dekompozycja komponentów
- Co dalej?

Tutorial

- mała skala
- mała złożoność
- mało problemów ze stanem

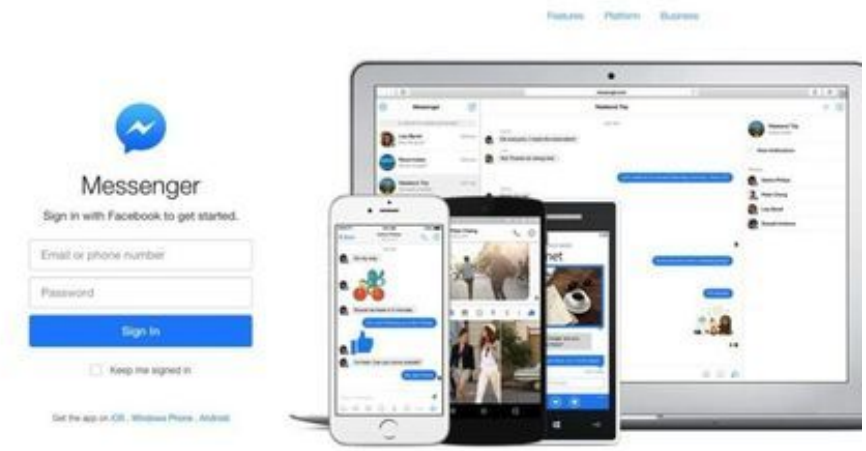
VS

Projekt

- duża skala
- duża złożoność
- dużo problemów ze stanem

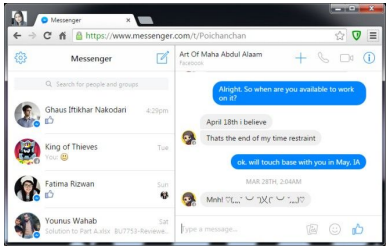
Rzeczywistość

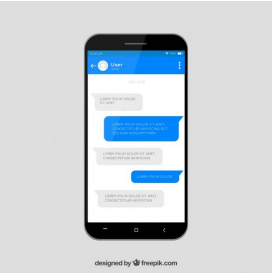
Wiele prezentacji, ta sama logika



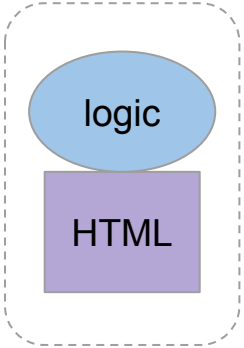
- problem
  - trzeba napisać aplikacje webowe i mobilne, które będą wyświetlały konwersacje
  - każda wersja pobiera listę konwersacji i manipuluje nią w taki sam sposób

Wiele prezentacji i duplikacje

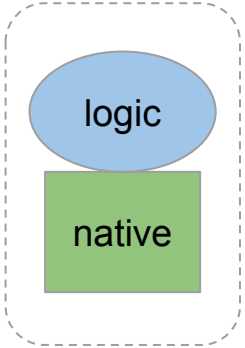




web component

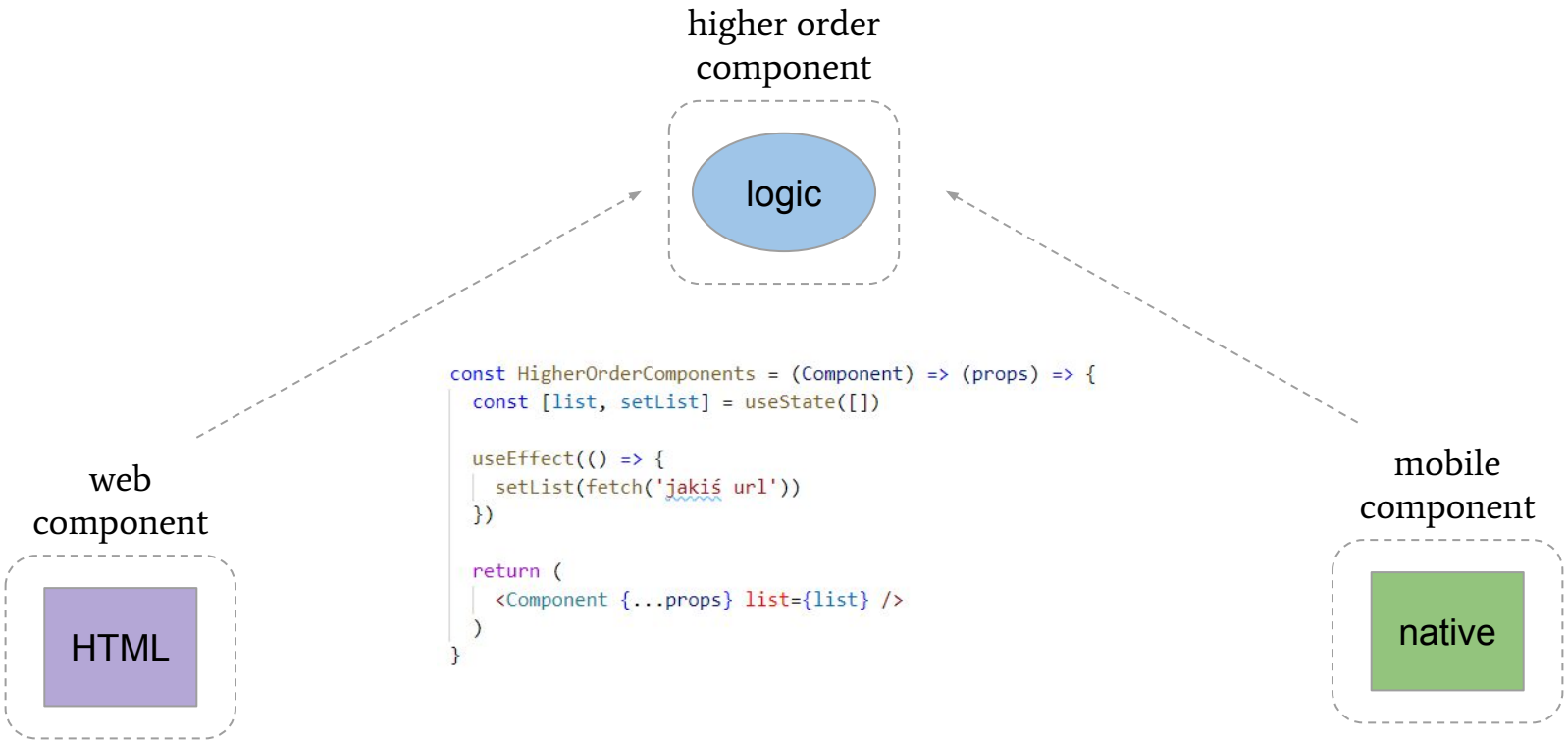


mobile component



- rozwiązanie “nieoptymalne”
  - osobne komponenty dla wersji webowej i mobilnej, ze skopiowaną logiką
- problemy
  - duplikacja
  - ~~one source of truth~~
  - wymieszane odpowiedzialności

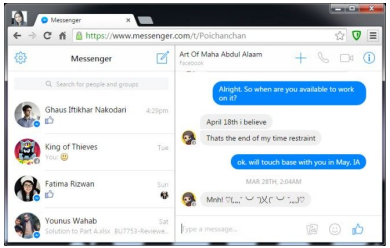
Higher order component

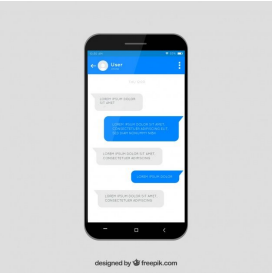


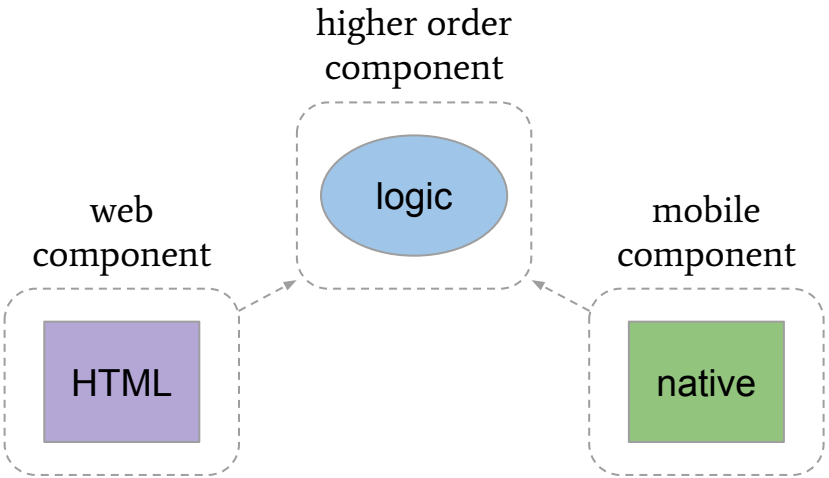

```
const HigherOrderComponents = (Component) => (props) => {
  const [list, setList] = useState([])

  useEffect(() => {
    setList(fetch('jakiś url'))
  })

  return (
    <Component {...props} list={list} />
  )
}
```

```
const WebComponentWithLogic = HigherOrderComponent(WebComponent)  const MobileComponentWithLogic = HigherOrderComponent(MobileComponent)
```

Higher order component

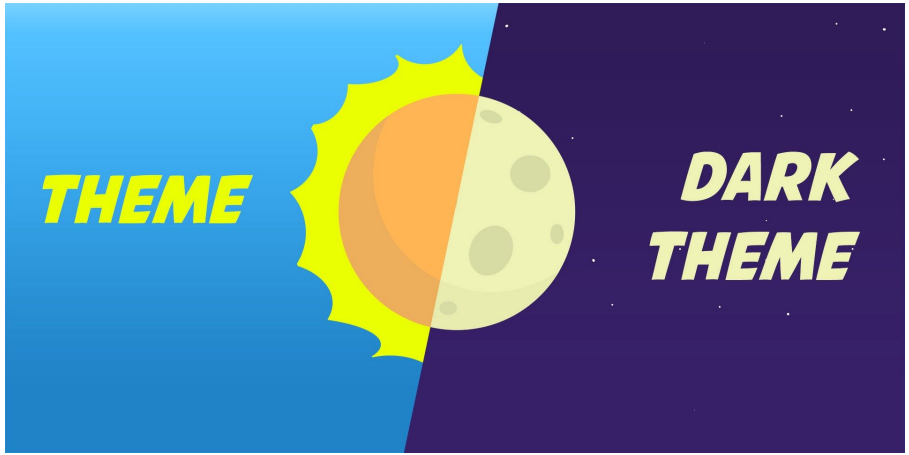






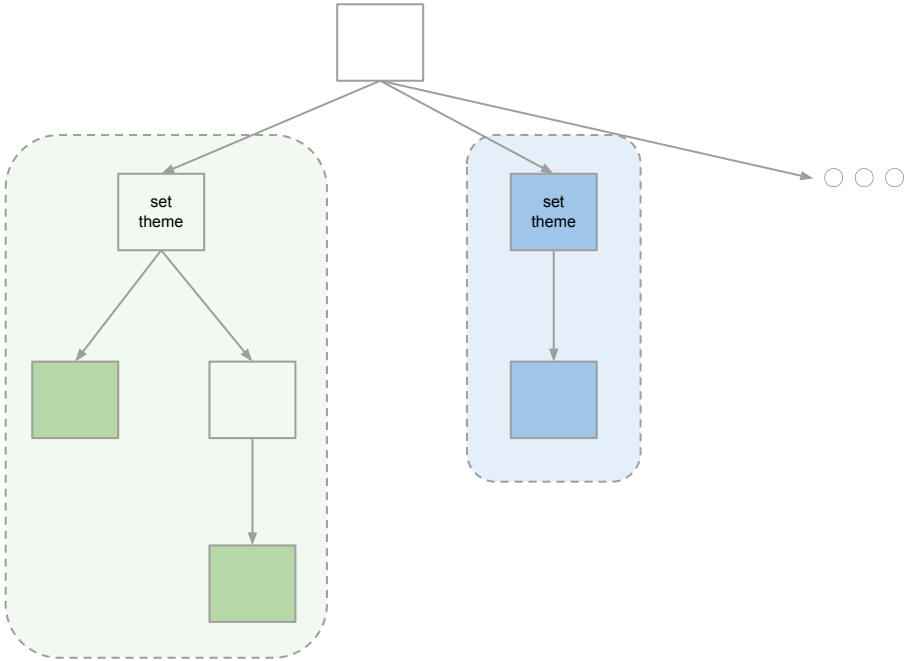
- rozwiązanie “optymalne”
  - osobne komponenty dla prezentacji oraz jeden reużywalny dla logiki
- korzyści
  - brak duplikacji
  - one source of truth
  - SRP
  - OCP

Mutli-themes problem



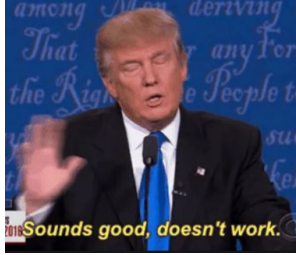
- problemy
  - możliwość ustawiania motywu w aplikacji
  - nie ma jednego “globalnego” motywu, niektóre komponenty (oraz ich dzieci) mogą posiadać własne
  - istnieje możliwość dodawania nowych komponentów, z własnymi stanem motywów

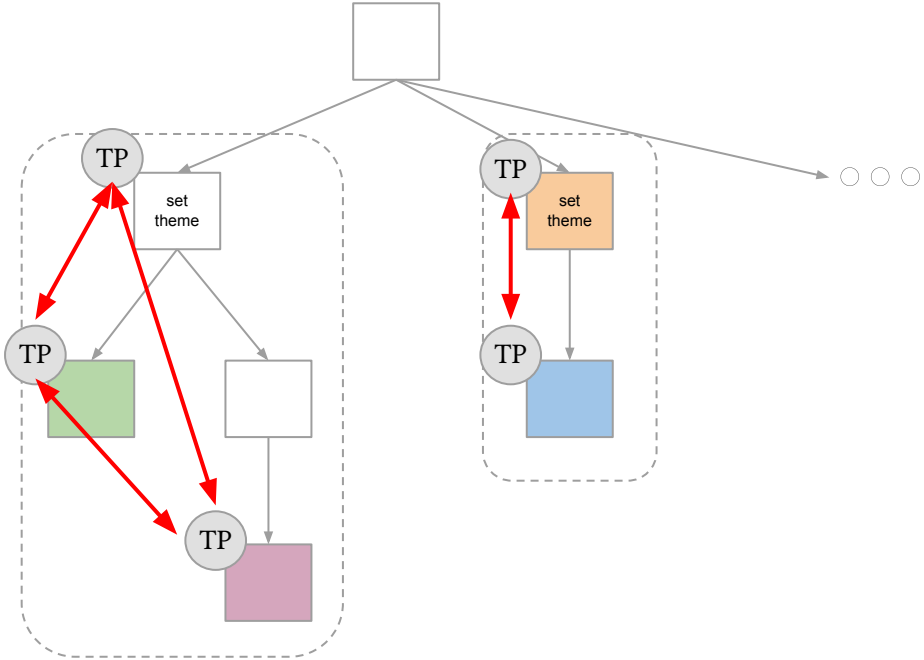
Mutli-themes problem - wizualizacja



- problemy
  - możliwość ustawiania motywu w aplikacji
  - nie ma jednego “globalnego” motywu, niektóre komponenty (oraz ich dzieci) mogą posiadać własne
  - istnieje możliwość dodawania nowych komponentów, z własnymi stanem motywów

1lvl HOC - ThemeProvider

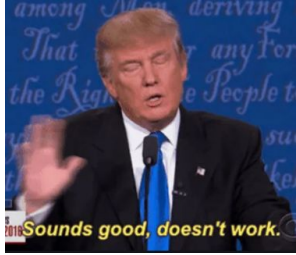


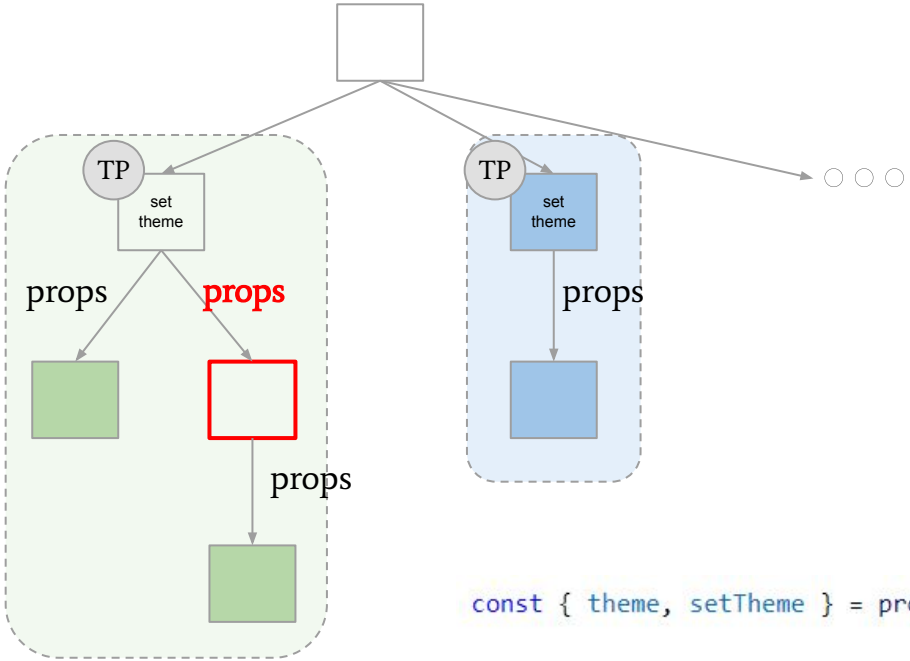


- problemy
  - HOCe nie mają wspólnego stanu, każdy jest niezależny
  - brak “przekazywania” stanu

```
const ThemeProvider = (Component) => (props) => {
  const [theme, setTheme] = useState('blue')

  return (
    <Component
      {...props}
      theme={theme}
      setTheme={setTheme}
    />
  )
}
```

HOC + props

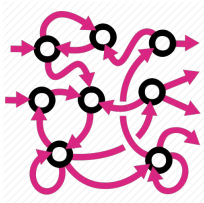





```
const { theme, setTheme } = props
```

- co prawda działa, ale...
- problemy
  - pojawiają się elementy pośredniczące, których dane propsy nie interesują
    - niepotrzebne re-rendery
  - im bardziej domena rośnie, tym więcej propsów trzeba będzie przepychać

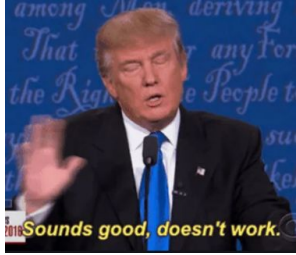
Spaghetti props

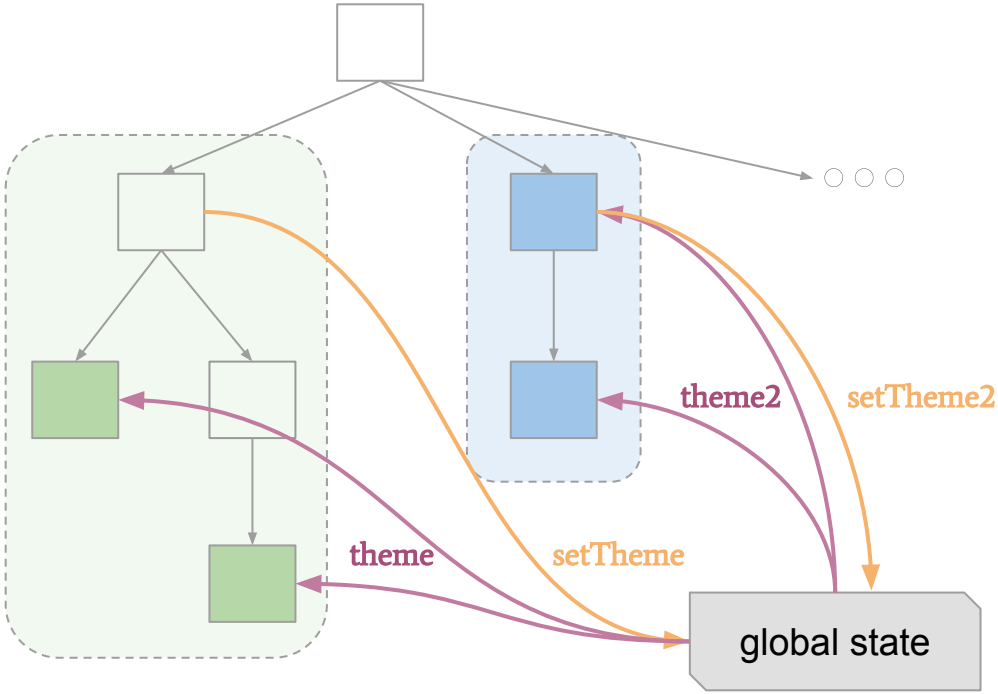


```
const Component = ({
  currentUser,
  setNextPage,
  someData,
  sayHello,
  // i kolejne mnóstwo propsów
  ...props,
}) => {
  // mnóstwo jakiegoś kodu

  return (
    <AnotherComponent
      {...props}
      currentUser={currentUser}
      loadPreviousData={loadPreviousData}
      sayHelloToDraża={() => sayHello('Draża')}
      RenderItems={<ItemsComponent someData={someData} />}
      // i kolejne mnóstwo propsów
    />
  )
}
```

- problemy
  - brak SRP
  - łatwo stworzyć kruchy kod
  - można zapomnieć przekazać jakieś propsy
  - nieczytelność
  - gigantyczne pliki
  - re-rendery
  - trudne do wykrycia błędy
  - brak reużywalności

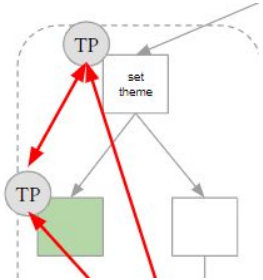
global state + actions/selectors





- rozwiązanie
  - rozdzielone akcje, selectory i stan od komponentów (flux)
- problemy
  - elastyczna architektura przechowywania *themów*
  - “duplikacje”
    - potrzebne “klucze”
  - wzrasta poziom skomplikowania

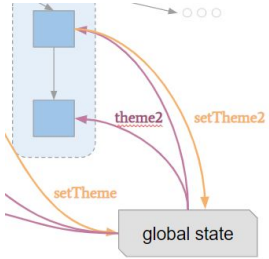
Brakujące narzędzie



- Komponenty, HOCe
  - stan lokalny (chyba, że propsy)
  - reużywalny

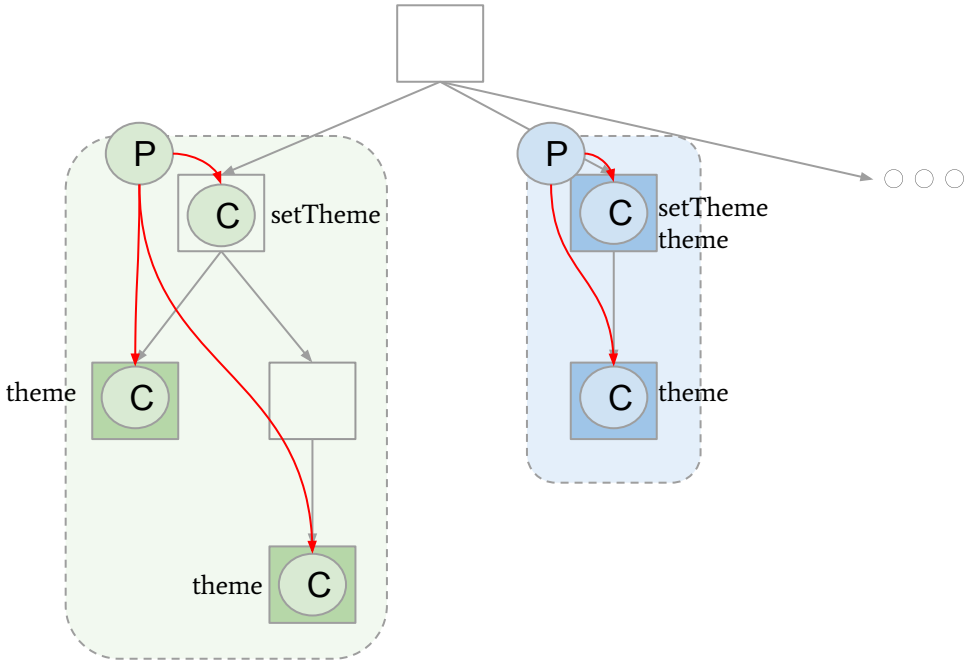
?

- ?
  - stan lokalny względem aplikacji, ale lokalny względem drzewa
  - reużywalny



- Global state, flux
  - stan globalny
  - trudno reużywalny (odpowiednia architektura)
  - wzrasta skomplikowanie

Contexty



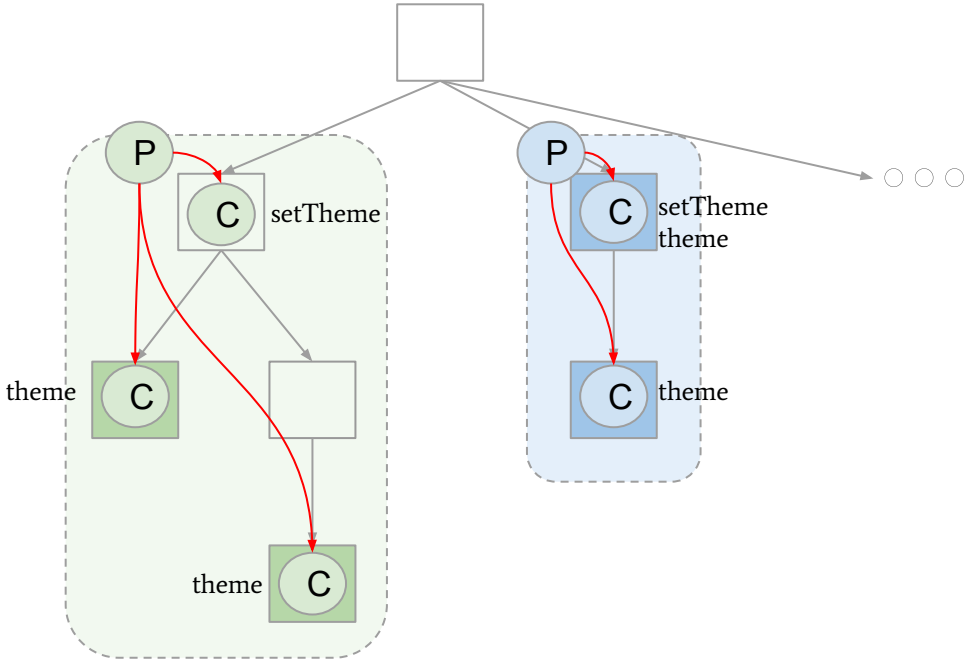
- provider

```
const ThemeContext = createContext()

const ThemeContextProvider = ({ children }) => {
  const [theme, setTheme] = useState("blue")

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

const ComponentWithContext = () => (
  <ThemeContextProvider>
    <Component />
  </ThemeContextProvider>
)
```

- consumer

```
const Component = () => {
  const { setTheme } = useContext(ThemeContext)

  return (
    <button onClick={() => setTheme("red")} />
  )
}
```

Contexty

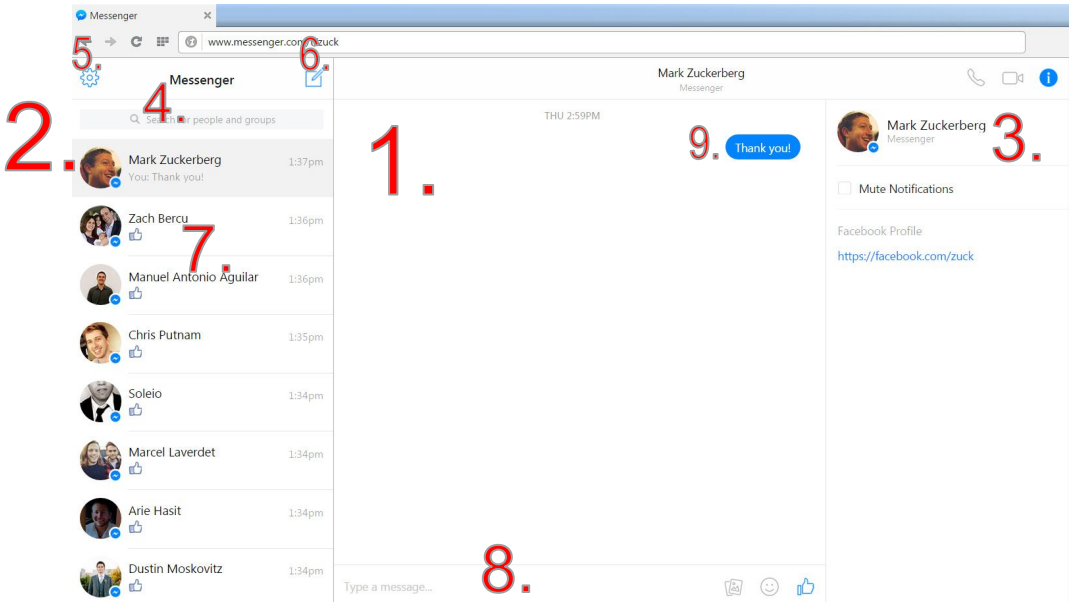


- rozwiązanie
  - dostęp tylko tam gdzie trzeba
  - brak niepotrzebnych re-renderów
  - SRP
  - hermetyzacja
  - reużywalność
- problemy
  - performance, dla szybkich zmian

Monolit

Messenger.js

1. Panel konwersacji
2. Panel nawigacji
3. Panel z informacjami
4. Wyszukiwarka znajomych
5. Opcje Messengera
6. Napisz wiadomość do innej osoby
7. Lista znajomych
8. Wysyłanie wiadomości
9. Lista wiadomości



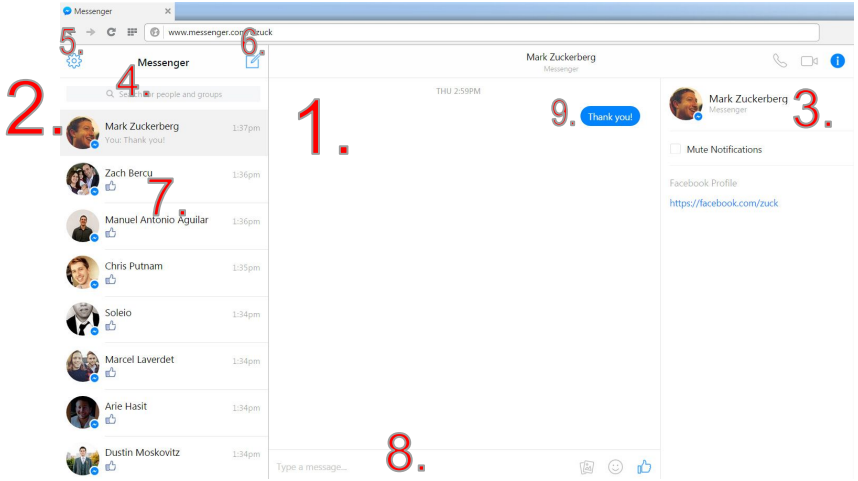
Problemy

1. Kod jest wrażliwy na zmiany. Dodanie nowej funkcjonalności może powodować nieoczekiwane zmiany w innych miejscach. Złamana zostaje zasada **Open-close principle.**

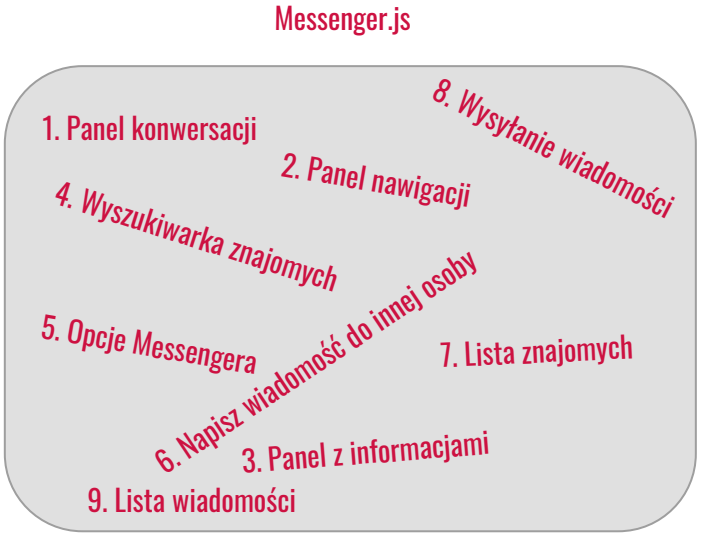
2. Kod ma zbyt dużo odpowiedzialności. To powoduje, że komponent puchnie, jest coraz trudniejszy i **droższy** w utrzymaniu. Łamie to zasadę **Single-responsibility principle (SRP).**

3. Kod jest mało czytelny. Jeden duży komponent z wieloma odpowiedzialnościami utrudnia rozeznanie się w kodzie, więc wydłuża czas potrzebny na wprowadzenie zmian.

4. Monolityczne komponenty powodują, że są nie reużywalne. Przez to będziemy czasem “wymyślać koło na nowo”, co zwiększy dług techniczny i spowoduje, że utrzymanie i rozbudowa aplikacji będzie coraz **droższa.**



Monolit vs Dekompozycja



Dlaczego dekompozycja?

Technicznie

1. Komponenty będą mniej podatne na błędy, ponieważ są mniejsze i mają węższy zakres odpowiedzialności. To powoduje, że debugowanie takiego kodu jest mniej czasochłonne, więc i tańsze.
   Dzięki temu nasz kod będzie zgody z zasadą **SRP.**

2. Małe komponenty powodują, że w razie konieczności ich wymiany, nie będziemy musieli refaktoryzować połowy projektu

3. Małe komponenty z małym zakresem odpowiedzialności charakteryzują się tym, że łatwiej jest je przetestować automatycznie. Duża ilość testów jest również gwarantem, że nasza aplikacja działa poprawnie.

Biznesowo

1. Większa ilość małych komponentów, pozwala na stworzenie biblioteki, której możemy użyć w innych miejscach w aplikacji, czy nawet wykorzystać ją do budowy nowych projektów. W ten sposób oszczędzamy czas i pieniądze.

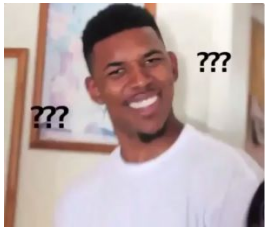
2. Małe komponenty możemy składać w coraz to większe funkcjonalności, czy strony. To pozwoli nam na stosowanie domenowego nazewnictwa, co spowoduje, że kod będzie odzwierciedleniem wymagań biznesowych.

React nie jest trudny...

- odpowiedzialności w Reakcie
  - prezentacja
  - zarządzenie logika
  - współdzielenie stanu
- narzędzia
  - dekompozycja
  - HOC
  - Context
  - Global state - (e.g. redux/flux)

..., ale dobór narzędzi do problemów już tak.

VS



DOŁĄCZ

DEVENV.PL/SLACK