ZROZUMIEĆ REACT

JSX - JavaScript Syntax Extension

Pełni rolę preprocesora. Babel zamienia go na czysty JavaScript.

JSX nie jest wymagany w pracy z React, ale jest oficjalnie rekomendowany (podobnie jak ES6). Został zaprojektowany po to, by używać z React.

Dzięki niemu jest SZYBCIEJ, ŁATWIEJ, PRZEJRZYŚCIEJ

JSX a HTML

Wygląda jak HTML, ale to jest JavaScript.

```
const header = <h1 className="title">Witaj na stronie!</h1/>
```

Może być używany wszędzie tam, gdzie inne typy JavaScript (bo jest tak naprawdę funkcją). JSX może być więc przypisywany do zmiennych, tablic, do właściwości obiektów, może być zwracany z funkcji itd.

JSX tworzy element React.

JSX a HTML

Składnia daje możliwość podawania atrybutów oraz zagnieżdżania elementów. Czasami nazwa atrybutu jest inna niż w HTML lub wymaga napisania za pomocą składni camelCase (np. className).

```
return (
   <div className="blog">
      <h1 id="main">Pierwszy artykuł</h1>
      <p>Lorem ipsum ...</p>
      <input type="text" />
      <img src="img/image.png" alt="obrazek"/>
   </div>
)
```

JSX - uwaga, używaj nawiasów

By uniknąć autmatycznego dodawania średnika (w procesie wykonywania kodu), kod JSX często piszemy w nawiasach.

```
//Przykład 1 - błąd, po return zostanie automatycznie wstawiony średnik
return ;
   <div className="blog">
      <h1 id="main">Pierwszy artykuł</h1>
   </div>
//Przykład 2 - dobrze, średnik nie zostanie dodany
return (
   <div className="blog">
      <h1 id="main">Pierwszy artykuł</h1>
   </div>
);
//Przykład 3 - średnio, zadziała, ale nie jest przejrzyście, wszystko w jednej linii
(tej samej co return czy przypisanie do zmiennej)
return <div className="blog"><h1 id="main">Pierwszy artykuł</h1></div>
```

JSX - jeden element główny

W takiej sytuacji jak poniżej będziemy mieli błąd. React pozwala bowiem tylko na jeden element główny.

```
return (
    <h1 id="main">Pierwszy artykuł</h1>
    <p>Lorem ipsum ...</p>
)
```

```
SyntaxError: Adjacent JSX elements must be wrapped in
an enclosing tag. Did you want a JSX fragment <>...</>?
```

JSX - jeden element główny

Rozwiązanie to dodanie jakiegoś elementu (np. div) lub skorzystanie z React.Fragment

```
return (
   <div>
      <h1 id="main">Pierwszy artykuł</h1>
      <p>Lorem ipsum ...</p>
   </div>
)

return (
   <React.Fragment>
      <h1 id="main">Pierwszy artykuł</h1>
      <p>Lorem ipsum ...</p>
   </React.Fragment>
)
```

JSX - inny zapis React.Fragment

React.Fragment można też zapisać za pomocą pustego znacznika.
React.Fragment nie tworzy żadnego elementu w DOM.

```
return (
   <React.Fragment>
      <h1 id="main">Pierwszy artykuł</h1>
      <p>Lorem ipsum ...</p>
   </React.Fragment>
)

return (
   <>
      <h1 id="main">Pierwszy artykuł</h1>
      <p>Lorem ipsum ...</p>
   </>
)
```

JSX - React.createElement()

JSX po kompilacji to tak naprawdę metoda z obiektu React.
Tekst to wywołanie metody React.createElement

React.createElement(type, props, children)

```
React.createElement(
    "p",
    null,
    "Tekst"
);
```

JSX - React.createElement()

React.createElement(type, props, children)

```
React.createElement(
    "p",
    {id: 'p10', className='title'},
);
```

JSX - React.createElement()

Hej Tomasz!

React.createElement(type, props, children)

```
React.createElement(
   "p",
   { id: "p10", "class": "title" },
   React.createElement(
      "span",
      null,
      "Hej "
   ),
   "Tomasz!"
);
```

JSX - React.createElement()

Gdy element bedzie tworzony za pomocą naszego komponentu, wtedy też korzystamy z metody createElement, ale pierwszy argument nie jest stringiem i pisany jest z dużej litery.

Np. <ListItem/>

React.createElement(type, props, children)

```
React.createElement(
    ListItem,
    null
);
```

JSX - React.createElement()

Gdy element bedzie tworzony za pomocą naszego komponentu, wtedy też korzystamy z metody createElement, ale pierwszy argument nie jest stringiem i pisany jest z dużej litery.

Np. <ListItem title="hello" options={options} />

React.createElement(type, props, children)

```
React.createElement(
   ListItem,
   { title: "hello", options: options }
);
```

JSX - Babel

Jeśli chcemy używać JSX (a chcemy), to potrzebujemy oprócz bibliteki React (bo JSX to tak naprawdę metoda React.createElement) także kompilatora Babel.

Babel dokona prekompilacji JSX (i ES6) na kod zrozumiały dla przeglądarki.

Ps. Nazwa "Babel" pochodzi od dziwnej żółtej ryby z powieści "Autostopem przez Galaktykę".

JSX - zagnieżdżenia - jak to wygląda w czystym JS

```
React.createElement(
     "article",
     { "class": "main" },
     React.createElement(
          "h1",
          null,
          "Tytu\u0142"
      ),
     React.createElement(
          "p",
           null,
           "Tekst",
           React.createElement(
                "strong",
                null,
               "Wazny"
            )
     )
);
```

```
<article className="main">
    <h1>Tytuł</h1>
    <p>Tekst<strong>Wazny</strong></p>
</article>
```

JSX - składania - wyrażenie wewnątrz

Za pomocą nawiasów klamrowych do JSX można przekazać wyrażenie czystego JavaScript.

```
const Result = () => {
    const number = 10
    return (
        <p>Wynik mnożenia przez dwa {number + 2}</p>
    )
}
```

JSX - składania - wyrażenie wewnątrz

Za pomocą nawiasów klamrowych do JSX można przekazać wyrażenie czystego JavaScript.

```
const Result = () => {
    const fn = () => {
        return /* ... */
    }
    return (
        <p>Wynikiem będzie {fn()}</p>
    )
}
```

Komentarze wewnątrz JSX

```
taki komentarz jak w JS nie jest dozwolony

<footer>
        /* <p>Stopka</p> */
        // <p>Copy</p>
</footer>


taki komentarz w JSX jest dozwolony (trzeba JSX wziąć w nawias klamrowy wcześniej)

<footer>
        {/*<p>Wynik mnożenia przez dwa {number * 2}</p>*/}
</footer>
```

Pamiętaj o zamykaniu znaczników w JSX

```
<div></div> //dobrze

<div/> //dobrze

<div> // źle

<img src=""/> //dobrze

<img src=""> //źle ( w html byłoby dobrze)

<SidebarMenu/>  /dobrze

<SidebarMenu>...</SidebarMenu> //dobrze
```

Element React czyli element Virtual DOM

Pisząc JSX (używając metody createElement), tworzymy element React będący elementem tzw. wirtualnego DOM.

```
const element = (
    <div className="title">
        Cześć!
    </div>
);
```

Powstaje obiekt (element React), który w wersji uproszczonej wygląda tak:

```
element
{
    type: 'div',
    props: {
        className: 'title',
        children: 'Cześć!'
    }
}
```

Element React - budowa

Element React to najmniejszy blok aplikacji budowanej w React.

Elementy opisują strukturę aplikacji.

Elementy są obiektami.

Komponenty w oparciu o elementy React tworzą strukturę interfejsu aplikacji.

Element React - niemodyfikowalne

Elementy React są niezmienialne!

Nie zmieniamy ich po stworzeniu. Proces aktualizacji (re-renderowanie) polega na stworzeniu nowego elementu i zastąpienie starego. Dzieje się to przy renderowaniu aplikacji i jest to jedno z zadań React (ustalenie, co się zmieniło).

Element reprezentuje fragment interfejsu w danym momencie.

Virtual DOM

Na podstawie elementów React tworzona jest struktura dokumentu w React (tzw. virtual DOM). Element React jest wirtualną wersją węzła DOM.

Za pierwszym razem (stworzenie/renderowanie aplikacji), ta struktura jest renderowana w prawdziwym DOM. Jednak każda kolejna zmiana powoduje, że React porównuje strukturę elementów React (virtual DOM) przed i po zmianie, a do prawdziwego DOM redneruje tylko zmianę, więc tylko te elementy DOM się aktualizują, które rzeczywiście sie zmieniły (na podstawie porównania starego i nowego virtual DOM).

Biblioteka React DOM

Wybrany w metodzie element DOM staje się kontenerem dla aplikacji React.

--------- index.html

```
<body>
        <div id="app"></app>
</body>
```

-------- index.js

```
ReactDOM.render(<Application />, document.getElementById('app'))
```

Kilka aplikacji na stronie

Można używać React do przygotowania widżetów na tradycyjnej stronie. Najczęściej jednak posługujemy się ReactDOM.render() tylko raz, ponieważ tworzymy SPA (cała strona, plik HTML jest wtedy aplikacją React)

Przykład kilku aplikacji React na jednej stronie (jako osobne widżety)

--------- index.html

```
<body>
        <nav id="nav"></nav>
        <!-- inne elementy -->
        <section class="comments"></section>
</body>
```

-------- index.js

```
ReactDOM.render(<Menu />, document.getElementById('nav'))
ReactDOM.render(<Comments />, document.querySelector('section.comments'))
```

Virtual DOM - po co? Podsumowanie

Szybszy, wydajniejszy sposób na interaktywny, dynamiczny interfejs.

Renderowane (przesyłane do DOM) jest tylko to, co jest zmieniane w aktualnym widoku interfejsu.

Virtual DOM jest oparty na elementach React. To bardzo ważny mechanizm React, o którym powinniśmy myśleć jako o abstrakcyjnej warstwie naszego interfejsu. React sprawia, że między nasz kod a DOM umieszczana jest warstwa pośrednia zbudowana z elementów React, które odpowiadają temu rzeczywistemu DOM.

Programista projektuje zmianę stanu, strukturę komponentu i strukturę aplikacji. Za całą relację, działanie i wymianę informacji między Virtual DOM i DOM odpowiada już biblioteka React.

Komponenty

Podstawowym elementem interfejsu użytkownika jest komponent, który składa się ze struktury i logiki (co i jak jest wyświetlane).

Wyróżniamy dwa typy komponentów w React: komponenty stanowe (klasowe) i komponenty bezstanowe (funkcyjne).

Komponenty - cechy komponentów

1. hermetyzacja (wyodrębnienie i ukrycie komponentu, niezależność)
2. możliwość umieszczania komponentu w innym komponencie (relacja, zagnieżdżenie)
3. mogą być używane wielokrotnie, w różnych miejscach (są do pewnego stopnia uniwersalne)
4. zwracają zawartość (renderują coś) i umożliwiają re-renderowanie przy zmianach (zoptymalizowane przez mechanizm Virtual DOM)
5. zbudowane z logiki i struktury. Komponent opisuje, co i jak ma być wyświetlone (jak ma się zmieniać przy zmianie stanu, czyli jak np. ma się zmieniać przy interakcji czy pobraniu nowych danych).

Cechy dobrego komponentu:

- ma cel, najlepiej mały i precyzyjnie określony
- nie jest zbyt rozbudowany (jeśli komponent jest zbytnio rozbudowany, powinien być z niego wydzielony osobny komponent)

Komponenty - nazwa wielką literą

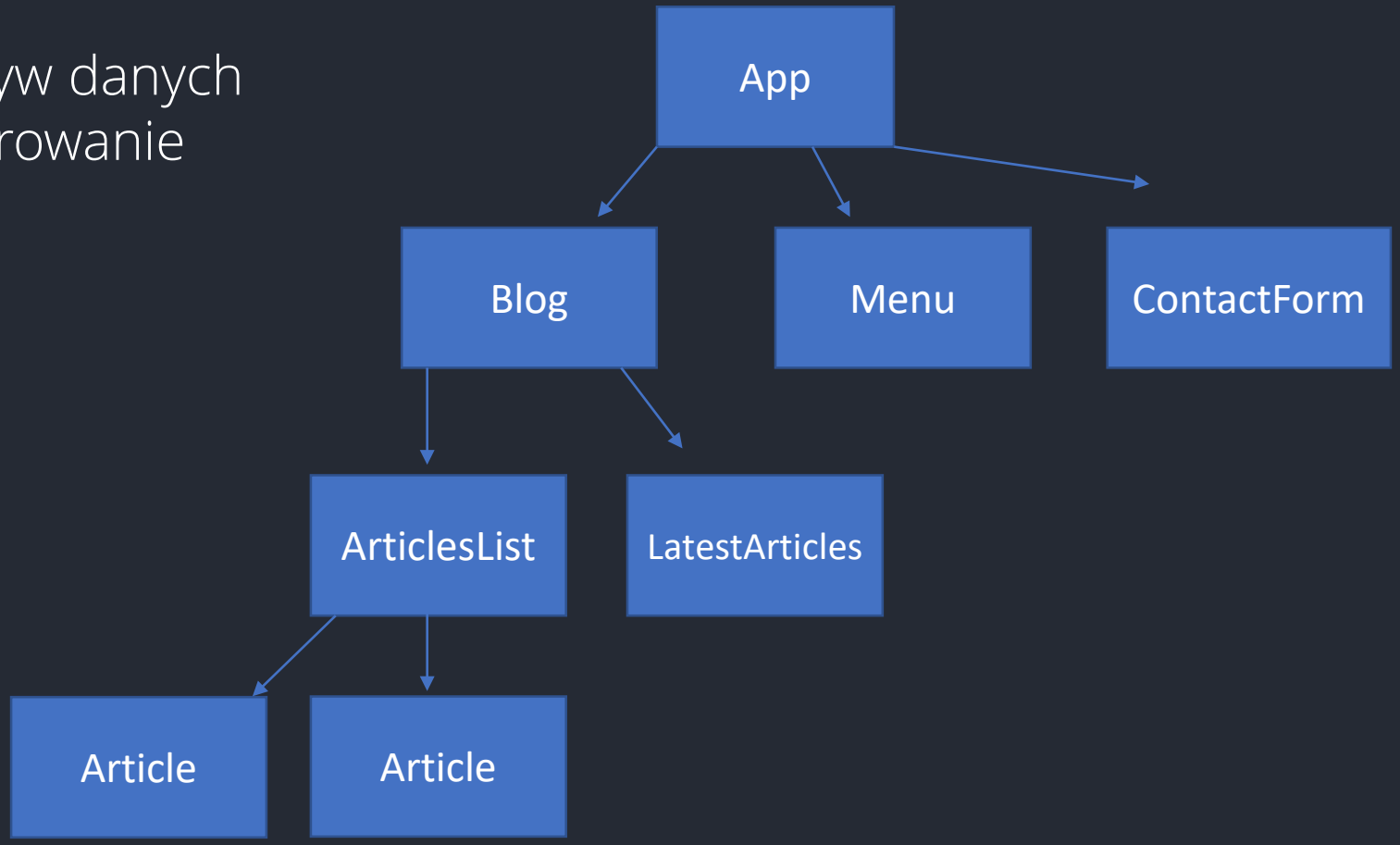
Nazwa komponentu powinna być pisana wielką literą przede wszystkim dlatego, by później prawidłowo taki komponent wywołać.

```
loginButton <loginButton/> // błąd, wywołuje jako tag "loginButton" (jak div itp.)
React.createElement(
   "listItem",
   { title: "hello", options: options }
);

LoginButton <LoginButton/> //dobrze
React.createElement(
   ListItem,
   { title: "hello", options: options }
);
```

Komponent powinien mieć nazwę zgodą z tym, do czego jest przeznaczony.

Aplikacja React to drzewo komponentów

Przepływ danych
Renderowanie



Komponent bezstanowy (funkcyjny) - tworzenie

Najprostszy sposób na stworzenie komponentu to stworzenie funkcji (można użyć funkcji strzałkowej). Taka funkcja powinna zwracać element React (JSX).

```
const ShoppingList = () => {
   return (
      <div>
         <h1>Na wieczór</h1>
         <ul>
            <li>śledzik</li>
            <li>picie</li>
            <li>paluszki</li>
         </ul>
      </div>
   )
}
```

Komponent bezstanowy - wersja tylko struktura

Jeśli komponent funkcyjny tylko zwraca elementy React, to można użyć skróconego zapisu funkcji strzałkowej (od razu return).

```
const ShoppingList = () => (
      <div>
         <h1>Na wieczór</h1>
         <ul>
            <li>śledzik</li>
            <li>picie</li>
            <li>paluszki</li>
         </ul>
      </div>
   )
```

Komponent bezstanowy - cechy

Tworzy komponent bez obiektu state (tzw. stateless component)

Zwraca element(y) React.

Duża litera w nazwie.

Może przyjmować w chwili tworzenia instancji atrybuty, które umieszczane są w obiekcie props instancji (egzemplarza) tego komponentu.

Instancja komponentu

Tworząc komponent, jeszcze go nie używamy (mamy wzór komponentu). By powstała instancja komponentu (może powstać jedna albo wiele) musimy się do niego odwołać. Niezależnie, jak stworzyliśmy komponent (używając klasy czy funkcji), będzie to wyglądało w ten sposób:

```
<NazwaKomponentu/>
```

lub

```
<NazwaKomponentu>
   Tu możemu przekazać props.chidren
</NazwaKomponentu>
```

Takie wywołanie może nastąpić w metodzie ReactDOM.render() lub, najczęściej, w innym komponencie.

Komponent a instancja komponentu

```
// komponent
const ShoppingList = () => (
      <div>
      </div>
   )

// instancja komponentu
<ShoppingList title="lista zakupów"/>
```

Potraktuj to jak wywołanie funkcji (którym w istocie jest React.createElement), tworzącej instancję/egzemplarz, gdzie do jej środka przekazywane są argumenty (którymi są atrybuty), które są umieszczane wewnątrz instancji komponetu (o props jeszcze powiemy).

Komponent funkcyjny - bezstanowy

Struktura i logika

```
const Example = props => {
      //dodatkowa logika, jeśli potrzebna
      return (
         {/*JSX*/}
      )
}
```

Komponent stanowy (klasowy)

Komponent możemy też zdefiniować za pomocą klasy. Ma on wtedy zaimplementowane dwa dodatkowe rozwiązania: obiekt state (stan obiektu) oraz cykl życia (metod w komponencie). Komponet klasowy powinien być używany tylko, jeśli potrzebujemy któregoś z tych rozwiązań (domyślnie powinien być używany komponent funkcyjny). Komponent klasowy wymaga użycia metody render, która zwraca elementy React (JSX)*. Inne metody, np. constructor, nie są wymagane.

* może też zwrócić inne akceptowane wartości, np. null. Wtedy taki komponent jest pusty i nie wpływa na DOM.

Komponent stanowy (klasowy)

Komponent klasowy wymaga dziedziczenia z klasy Component znajdującej się w obiekcie React.

```
class ShoppingList extends React.Component {
   render() {
      return (
         <div>
            <h1>Na wieczór</h1>
            <ul>
               <li>śledzik</li>
               <li>picie</li>
               <li>paluszki</li>
            </ul>
         </div>
      )
   }
}
```

komponent klasowy

Tworzy komponent z obiektem state (tzw. stateful component).

Musi być w nim metoda render a w niej return, w którym zwracany jest (są) element(y) React.

Duża litera w nazwie.

Może przyjmować w chwili tworzenia instancji atrybuty, które umieszczane są w obiekcie props tworzonego egzemplarza komponentu (dokładnie tak samo jak przy komponencie bezstanowym).

Zmiana state (zmiana właściwości stanu) powoduje ponowne wywołanie metody render (i innych, rzadziej używanych metod cyklu życia metod w komponencie).

Dostęp do obiektów props i state jest możliwy w komponencie klasowym za pomocą this.state i this.props

komponent klasowy - props = {} i state = null

Obiekt props istnieje, obiekt state musi być stworzony (domyślnie właściwość state ma przypisane null).

```
class App extends React.Component {

        state = { word: "wpisz..." } //state (stan) stworzeny bezpośrednio w klasie

        handleChange = () => {
           //obsługa zdarzenia onChange
        }

        render() {
           return (
              <div>
                  <input type="text" value={this.state.word} />
                  <h1 className="title">{this.props.title}</h1>
              </div>
        )}}
```

komponent klasowy - props = {} i state = null

Wartości początkowe state możemy zadeklarować bezpośrednio w klasie (jak w poprzednim przykładzie), lub w konstruktorze.

```
class App extends React.Component {

        constructor(props) {
           super(props);
           this.state = {word: "wpisz..." } //state (stan) stworzeny w kostruktorze
        }

        handleChange = () => {}

        render() {
           return (
              ...
        )}}
```

Komponenty - kiedy tworzyć?

Kiedy wydzielić nowy komponent z innego komponentu?

To kwestia oceny i programiści mogą podjąć różne decyzje projektowe.

Można jednak wskazać kilka zasad, którymi warto się kierować.

Zasada wstępna: Twórz komponenty zgodnie z zasadami i przyzwyczajeniami zespołu (Twoimi).

Komponenty - kiedy tworzyć (wydzielać) komponent

1. Gdy dana część interfejsu może być stosowana wielokrotnie, w różnych miejscach (nie musi być identyczna, ale może być różnicowana, np. przez props)
2. Gdy dana część interfejsu jest już skomplikowana (nie jest juz mała i prosta)
3. Gdy dana część interfejsu będzie rozwijana, zmieniana w przyszłości.
4. Gdy pojawia się w komponencie kilka funkcji obsługujących zdarzenie (handle...)
5. Gdy komponent ma więcej niż jeden cel.

Pamiętaj też, że model danych, jaki dostarcza baza/API może wymuszać pewną architekturę komponentów (dane wpływają na wygląd interfejsu).

Obiekty przechowujące dane w komponencie

Props (właściwości) - obiekt przechowujący dane przekazane do komponentu. Są to właściwości tylko do odczytu.

State (stan) - obiekt należący do komponentu stanowego (klasowego). Ich zmiana pociąga za sobą aktualizację komponentu.

State (Stan) - deklaracja w metodzie constructor

Obiekt state dostępny jest w każdym komponencie klasowym, przy czym wartość początkowa przypisana do właściwości to null. Używanie komponentu stanowego ma sens, o ile potrzebny nam w nim state.

```
class App extends React.Component {
   constructor(props) {
       super(props)
       this.state = {
          word: ""
       }
   }
   render() {
      return (
       ...
      )
   }
}
```

State - bezpośrednio w klasie

Nowszy sposób na tworzenie obiektu state. Efekt jest taki sam jak this.state w konstruktorze, ale napisaliśmy znacznie mniej kodu. Przecinek oczywiście nie jest wymagany (czy potrzebny) w ostatniej właściwości w obiekcie.

```
class App extends React.Component {
   state = {
      word: "",
   }
   render() {
      return (
        ...
      )
   }
}
```

State - zmiana stanu

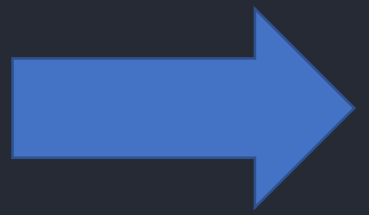
Zmiany w stanie powodują (domyślnie) ponowne renderowanie (aktualizację elementów React w komponencie i ponowne renderowanie komponentów dzieci). Zmiany (tylko zmiany) są przekazywane do DOM. Dzięki state komponenty stają się interaktywne.

```
class App extends React.Component {
   state = {
      word: "",
   }

   handleInputChange = (e) => {
      this.setState({
         word: e.target.value
      })
   }

   render() {
      return ( ... )
   }
}
```



```
render() {
   return (
      <div>
         <TextInput value={this.state.text} />
         <Result />
      </div>
   )
}
```

Zmiana state



render()

State - prywatny, dynamiczny element komponentu

State może być definiowany w komponencie i tylko sam komponent może go zmieniać (choć możemy metody do zmiany state wywołać w komponentach potomnych). Obiekt state nie jest dostępny (wprost) w innym komponencie, choć jego wartości mogą być do innego komponentu (podrzędnego) przekazane dzięki props. State często określa się jako lokalny czy prywatny.

```
//w komponencie klasowym
state = {
   user: "Adam"
}

render() {
   return (
      <Counter name={this.state.user}/>
   )
}
```

State - metoda setState

Zmiany w stanie mogą być dokonywane za pomocą metody setState.

Do metody setState przekazujemy obiekt (wersja 1).

```
//w klasie
state = {
   text: ""
}

handleClick(letter) {
   //1 Wersja - obiekt
   this.setState({
      text: this.state.text + letter
   })
}
```

State - metoda setState

Zmiany w stanie mogą być dokonywane za pomocą metody setState.

Do metody setState przekazujemy funkcję, która zwraca obiekt (wersja 2). W parametrze (w przykładzie "prevState") zostaje umieszczona aktualna wersja obiektu state.

```
//w klasie
state = {
   text: ""
}

handleClick(letter) {
   //2 Wersja - funkcja zwracajaca obiekt
   this.setState( prevState => ({
      text: prevState.text + letter
   })
 )
}
```

State - metoda setState - co się zmienia

Do setState przekazujemy tylko te właściwości obiektu state, które się zmieniają.

```
//w klasie
state = {
   text: "",
   number: 2 //nie zmienia się, więc nie musimy przekazywać w obiekcie setState
}

handleClick(letter) {
   //1 Wersja - obiekt
   this.setState({
      text: this.state.text + letter
   })

   //2 Wersja - funkcja zwracajaca obiekt
   this.setState(prevState => ({
      text: prevState.text + letter
   }))
}
```

Nie modyfikujemy obiektu state bezpośrednio.

```
this.state.name = "Damian";  // źle
```

W ten sposób nie spowodujemy aktualizacji komponentu (nie wywoła się metoda render).

Do modyfikowania stanem zawsze używamy metody setState.

Komponent sam zarządza stanem

Modyfikowaniem stanu zajmuje się sam komponent. Funkcja modyfikująca (np. handle...) powinna być w tym komponencie. Funkcja modyfikująca może być jednak przekazana do innego komponentu i wywołana w nim.

Wyobraźmy sobie aplikację kalkulator.

Funkcję modyfikującą tworzymy w komponencie <Kalkulator/>, w której jest też state odpowiadający za aktualny stan aplikacji (wiedza, co wcisnął użytkownik i jaki jest wynik). Jednak wywołanie funkcji następuje z poziomu nie komponentu <Kalkulator/> a z poziomu komponentów przycisków np. <Klawisz/>

```
<Klawisz title="add" click={this.handleClick} />
```

Renderowanie struktury

Komponent i znajdujące się w nim komponenty potomne są renderowane, kiedy zmienia się obiekt state komponentu (kiedy zmienia się stan).

Renderowanie jest procesem złożonym. Zanim zobaczymy zmiany na stronie, React porównuje, co się zmieniło w state, co zmieniło się w elementach React. Jeśli rezultat renderowania byłby dokładnie taki sam jak przed renderowaniem, to nic się nie zmieni z DOM. Tylko różnice są przekazywane do przeglądarki.

Wpływ stanu na aplikację

Dzięki state komponenty stają się interaktywne. Zmiana stanu to ponowne renderowanie struktury komponentu i jego komponentów potomnych.

Dzieki stanowi aplikacja staje się interaktywna.

Obiekt props

Skrót od properties.

Jest to przede wszystkim sposób na komunikację między komponentami (przekazywanie komponentowi danych). Przepływ danych (komponentu) od rodzica do dziecka.

Props składa się z atrybutów

Atrybuty*, które przekazujemy, tworząc instancję komponentu, stają się właściwościami obiektu props.

```
<ListItem title="hello" options={options}/>
```

*W istocie przekazujemy jako argument obiekt w React.createElement(type, props, children).

```
React.createElement(
    ListItem,
    { title: "hello", options: options }
)
```

Props - przekazanie jako atrybut do DOM

Warto jednak zauważyć, że atrybuty użyte w elementach React (nie w naszych komponentach) zostają także przekazane (z wyjątkami np. key, onClick) do elementu DOM.

```
// w komponencie
return (
     <div id="s20" className="blur" aaa={[1, 2]}
     data-user="no-name" key="1" onClick={userInfo}>Hello</div>
)

// w pliku html
<div id="s20" class="blur" aaa="1,2" data-user="noname">Hello</div>
```

Props - formatowanie - dobra praktyka

~~<ListItem title="hello" options={options}/>~~

```
<ListItem
  title="hello"
  options={options}
/>
```

Props - obiekt props

```
<ListItem
  title="hello"
  options={options}
/>
```

```
props: {
  title: 'hello'
  options: [...] //przy założeniu, że przekazaliśmy tablicę
}
```

Props - tylko atrybut? Wtedy przypisane true

```
<ListItem
  title="hello"
  options={options}
  visible
/>
```

```
props: {
  title: 'hello'
  options: [...] //przy założeniu, że przekazaliśmy tablicę
  visible: true //jeśli do nazwy właściwości nie przekażemy wartości, to przypisane zostanie true (boolean)
}
```

Props - obiekt w każdym komponencie

Obiekt props znajduje się w każdym komponencie.
Komponent stanowy - odwołujemy się przez this.props

```
class List extends React.Component {
   render() {
      console.log(this.props);
      return (
         ...
      )
   }
}
// {} - jeśli nie przekażemy atrybutów, to props jest pusty
```

Props - obiekt w każdym komponencie

Obiekt props znajduje się w każdym komponencie.

Komponent bezstanowy - odwołujemy się po nazwie argumentu (konwencja - props)

```
const List = (props) => {
   console.log(props);
   return (
      <div>{props.name}</div>
   )
}
// {} - jeśli nie przekażemy atrybutów, to props jest pusty
```

Props - użycie w metodach i JSX

Możemy z niego (obiektu props) skorzystać w metodach komponentu oraz przy renderowaniu. Odwołujemy się przez this.props w komponencie klasowym i poprzez nazwę przekazanego argumentu (konwencja dla nazwy tego argumentu to props) w komponencie funkcyjnym.

Props wypełniany danymi od rodzica

Zawartość props pochodzi często od komponentu nadrzędnego (sama instancja jest tworzona w komponencie nadrzędnym)

```
const AuthorsList = (props) => {
          return (
            <div id={props.id}>{props.names}</div>
        )
}

//w komponencie rodzica

<AuthorsList id={key} names={/*jakaś tablica*/} />


//<div id="2">Kowalski, Nowak, Kwiatkowski</div>
```

Props wypełniany atrybutami

```
class List extends React.Component {
   render() {
      return (
         <div id={this.props.id}>{this.props.name}</div>
      )
   }
}

const key = 2;
ReactDOM.render(<List id={key} name="shopping" />, document.getElementById('root'))

//<div id="2">shopping</div>
```

Props - przekazanie wartości state do dziecka

```
//inny komponent nadrzędny (rodzic) np. komponent App
return (
   <Heading id={this.state.key} name={user}>
)
-------------------------------------------------------------------------
class Heading extends React.Component {
   render() {
      return (
         <div id={this.props.id}>{this.props.name}</div>
      )
   }
}
```

Właściwości w props są tylko do odczytu

Props jest mechanizmem, który służy do propagowaniu danych między rodzicem a dzieckiem (między dwoma komponentami).

Props jest tylko do odczytu (nie można go zmieniać!).

Props powinny być zmieniane tylko w komponencie, w którym zostały stworzone za pomocą atrybutu (nowa wartość zostanie zaktualizowana w wyniku ponownego renderowania rodzica).

Tylko do odczytu

Komponent nadrzędny - to tu atrybuty mogą się zmieniać. Wtedy ponownie re-renderowany komponent przekazuje nowe wartości do obiektu props dziecka (w tym wypadku komponentu Heading).

```
//komponent nadrzędny
return (
  <Heading id={this.props.key} name={user}>
)

//komponent dziecka
const Heading = (props) => {
   return (
        <p id={props.key>props.name</p>
   )
}
```

Tylko do odczytu

<Heading id={this.state.key}>

```
const Heading = (props) => {
    props.id += 1  //źle!
    return (
        <h1>{props.id}</h1>
    )
}
```

Tylko do odczytu

<Heading id={this.state.key}>

```
const Heading = (props) => {
    const id = props.name + 1  //dobrze
    return (
        <h1>{id}</h1>
    )
}
```

Obiekt props - co zawiera

W skład obiektu props wchodzą atrybuty przekazane przy tworzeniu instancji w komponencie nadrzędnym (z wyjątkiem niektórych atrybutów, np. key, jeśli go użyjemy).

<Button title="Send!" click={this.handleClick}>{number}</Button>

//props

```
{
        children: 3  //załóżmy, że w number było 3
        click: (f) handleClick
        title: "Send!"
}
```

Props do inicjalizacji state

Właściwość z props możemy wykorzystać do inicjalizacji state.

```
state = {
  title: 'Świat to symulacja komputerowa',
  date: new Date(),
  author: this.props.name
}
```

Jeśli jednak state nie będzie sie zmieniać, to nie ma sensu takie przypisywanie do właściwości state właściwości props, tylko bezpośrednio skorzystajmy z właściwości props.

Props - wartości domyślne (default props)

Props to dane przekazane z komponentu rodzica. Ale nie tylko. Możemy też przekazać wartości domyślne dla props.

Pamiętaj, jeśli chcesz skorzystać z właściwości props, która nie istnieje, np.

```
<p>{this.props.name}</p> (komponent klasowy)
<p>{props.name}</p> (komponent funkcyjny)
```

otrzymasz undefined, w tym wypadku React zingnoruje takie wyrażenie (bo ignoruje undefined czy null)

```
<p></p>
```

Props - wartości domyślne (default props)

Możemy się zabezpieczyć, wskazując wartość alternatywną za pomocą sumy logicznej

```
<p>{this.props.name || "brak"}</p> (komponent klasowy)
<p>{props.name || "brak"}</p> (komponent funkcyjny)
```

Jeśli name w props nie będzie istniał, to dostaniemy wartość znajdującą się jako druga.

```
<p>brak</p>
```

Props - wartości domyślne (default props)

Możemy też użyć wartości domyślnej w postaci wartości statycznej w klasie, którą nazwiemy defaultProps. Jeśli taka wartość będzie istniała w props, to wartość z defaultProps zostanie zignorowana.

```
class App extends React.Component {

   render() {
      console.log(this.props.name);
      return (
        <div>{this.props.name}</div>
      );
   }

   static defaultProps = {
      name: "jestem!"
   }
}
```

Props - wartości domyślne (default props)

Można też to zrobić poza klasą, przypisując do niej właściwość defaultProps.

```
class App extends React.Component {

   render() {
      console.log(this.props.name);
      return (
        <div>{this.props.name}</div>
      );
   }
}

App.defaultProps = {
   name: "jestem!"
}
```

Props - wartości domyślne (default props)

W przypadku komponentu funkcyjnego przypisujemy również właściwość poza funkcją.

```
const App =  (props) => {
      console.log(props.name);
      return (
        <div>{props.name}</div>
      );
}

App.defaultProps = {
      name: "jestem!"
}
```

Komponenty - stanowy czy bezstanowy?

Komponent stanowy (komponent klasowy) tworzymy, gdy:

- potrzebujemy obiektu stanu (ale naprawdę potrzebujemy)
- chcemy wykorzystać cykl życia komponentu (metody cyklu życia).
- potrzebujemy funkcji do obsługi zdarzeń bezpośrednio w komponencie

W pozostałych wypadkach używamy komponentu bezstanowego (komponent tworzony za pomocą funkcji).

Data flow, czyli przepływ danych w React

Przepływ danych w React jest jak rzeka. Jednokierunkowy od góry (najwyższy komponent) do końca (ostatni zagnieżdżony komponent).

Rzeką jest mechanizm props. Łódki na rzece spływające w dół są obiektami props na każdym etapie, czyli między komponentami to mogą być inne łódki, ale też możliwe jest, by jakąś daną przesłać od pierwszego komponentu do ostatniego.

Data flow czyli przepływ danych w React

State można potraktować jako dopływy tej rzeki.

Dopływy są oddzielone tamą (są hermetyczne), ale można tamę otworzyć i część danych w formie łódek (props) wpuścić do rzeki (pamiętajmy, że rzeka płynie cały czas w dół, od najwyższego komponentu do ostatniego).

State może być źródłem nowych danych (zawartością nowych łódek), ale przede wszystkim jest on odpowiedzialny za to, że na danym etapie zawartość łódek może zostać zmieniona (mechanizm re-renderowania przy zmianie stanu, odświeża też props).

Data flow czyli przepływ danych w React

Te same dane mogą pojawić się w wielu miejscach. Te same dane, jeśli pojawią się w wielu komponentach, dobrze jest podnieść (przenieść) do pierwszego wspólnego przodka dla wszystkich elementów, na przykład w aplikacji kalkulator (jak niżej) state będzie w najwyższym komponencie:

```
kalkulator (state) - > panel (props zawiera niektóre elementy state)
                   - > klawiatura (props zawiera niektóre elementy state)
                       Czy może teraz dać liczbę czy działanie matematyczne
                            -> klawisz (wciśnięcie aktualizuje state w kalkulatorze)
```

SyntheticEvent (syntetyczny event)

React tworzy przy każdym zdarzeniu, które obsługuje (kliknięcie, ruch myszką itd), obiekt zdarzenia, który nazywa się syntetycznym eventem. Jest od odpowiednikiem obiektu event w DOM.

```
SyntheticEvent {dispatchConfig: {…}, _targetInst: FiberNode,
nativeEvent: InputEvent, type: "change", target: input, …}
  bubbles: (...)
  cancelable: (...)
  currentTarget: (...)
  defaultPrevented: (...)
  dispatchConfig: null
  eventPhase: (...)
  isDefaultPrevented: (...)
  isPropagationStopped: (...)
  isTrusted: (...)
  nativeEvent: (...)
  target: (...)
  timeStamp: (...)
  type: (...)
```

Wspierane eventy

Clipboard Events
Composition Events
Keyboard Events
Focus Events
Form Events
Mouse Events
Pointer Events
Selection Events
Touch Events
UI Events
Wheel Events
Media Events
Image Events
Animation Events
Transition Events
Other Events

Przykłady:

`onKeyDown, onKeyUp`

`onFocus, onBlur`

`onChange, onSubmit`

`onSelect, onMouseMove`

`onClick, onTouchMove,`

`onScroll,`
`onAnimationStart`

Obsługa zdarzeń (handling events)

- nazwy zdarzeń w React używają notacji wielbłądziej, np. onClick, onChange.

w HTML:

```
<button onclick="nazwafunkcji()">Przycisk</button>
```

w React

```
<button onClick={nazwafunkcji}>Przycisk</button>
```

Obsługa zdarzeń (handling events)

- ustawiasz nasłuchiwanie w chwili tworzenia elementu

```
return (
    <input onChange={nazwafunkcji} />
)
```

Problem z this

Problem polega na tym, że this w tym wypadku będzie undefined (nie będzie wiązany z komponentem), nie ustawimy więc this.setState.

```
handleClick(){
    console.log(this)
}

return (
    <input onChange={this.handleClick} />
)
```

Pamiętaj o wiązaniu this

Wersja 1 - bind bezpośrednio użyte w deklaracji eventu.

```
handleClick(){
    console.log(this)
}

return (
    <input onChange={this.handleClick.bind(this)} />
)
```

Pamiętaj o wiązaniu this

Wersja 2 - przypisanie nowej związanej metody w konstruktorze. W evencie zostaje metoda z konstruktura (bo jest właściwością instancji) a nie z prototypu.

```
constructor(props) {
   super(props);
   this.handleClick = this.handleClick.bind(this)
}

handleClick(){
   console.log(this)
}

return (
   <input onChange={this.handleClick} />
)
```

Pamiętaj o wiązaniu this

Wersja 3 - funkcja strzałkowa (bo korzysta z wiązania odziedziczonego, nie tworzy własnego).

```
handleClick = () => {
    console.log(this)
}

return (
    <input onChange={this.handleClick} />
)
```

Pamiętaj o wiązaniu this

Wersja 4 - anonimowa funkcja strzałkowa przekazana w evencie a w niej metoda właściwa

```
handleClick(){
    console.log(this)
}

return (
    <input onChange={() => this.handleClick()} />
)
```

Pamiętaj o wiązaniu this

Wersja 5 - deklaracja metody a potem jej przypisanie do właściwości

```
handleClick(){
 console.log(this)
}

handleClick = this.handleClick.bind(this)

return (
   <input onChange={this.handleClick} />
)
```

Przekazanie argumentów do funkcji

Wersja 1 - anonimowa funkcja strzałkowa przekazana w evencie a w niej metoda właściwa

```
handleClick(x, y) {
 console.log(x, y)
}

return (
 <input onChange={() => this.handleClick(arg1, arg2)} />
)
```

Przekazanie argumentów do funkcji

Wersja 2 - bind, w którym przekazujemy opócz this także argumenty (pierwszym argumentem musi być this)

```
handleClick(x, y) {
 console.log(x, y)
}

return (
 <input onChange={this.handleClick.bind(this, arg1, arg2)}/>
)
```

Przekazanie eventu i argumentów do funkcji

Wersja 1 - anonimowa funkcja strzałkowa, której argumentem jest event.
W funkcji przekazana właściwa metoda z argumentami a także z eventem.

```
handleClick(x, y, event){
 console.log(x, y, event)
}

return (
 <input onChange={e => this.handleClick(arg1, arg2, e)} />
)
```

Przekazanie argumentów do funkcji

Wersja 2 - bind, w którym przekazujemy oprócz this także argumenty (pierwszym argumentem musi być this). Przekazywany jest też event, jako ostatni

```
handleClick(x, y, event) {
 console.log(x, y, event)
}

return (
 <input onChange={this.handleClick.bind(this, arg1, arg2)}/>
)
```

Nazwy metod w React - przykładowe konwencje

handleClick, handleSubmit - w evencie przedrostek on, tutaj handle

```
handleSubmit() {
        ...
}

render() {
    return (
        <form onSubmit={this.handleSubmit}>...</form>
    )
```

Nazwy metod w React - przykładowe konwencje

Bardziej rozbudowana nazwa z handle. Obsługa kliknięcia (handleClick) staje się obsługą kliknięcia logowania (handleLoginClick), obsługa zmiany pola input czy potwierdzenia formularza.

```
handleLoginClick() {}

handleInputChange() {}

handleFormSubmit() {}
```

Nazwy metod w React - przykładowe konwencje

Część osób tworzy nazwy z przedrostkiem handle, ale w drugiej kolejności wskazuje efekt (cel) działania metody.

```
handleCreateProduct() {}

handleIncreaseItems() {}

handleLogoutUser() {}

handleDeleteOption() {}

handleAdd() {}

handleReset() {}
```

Nazwy metod w React - przykładowe konwencje

onClick, onChange - tak samo jak nazwa eventu

```
onClick() {
        ...
}

render() {
    return (
        <button onClick={this.onClick}>Kliknij!</button>
    )
```

Nazwy metod w React - przykładowe konwencje

Część osób tworzy nazwy z końcówką Handler.

```
inputChangeHandler() {}

formSubmitHandler() {}
```

//tak jak w poprzednich przykładach, ale w odwrotnej kolejności

handleInputChange => inputChangeHandler

Nazwy metod w React - przykładowe konwencje

Inne konwencje, bez słowa handle - określenie przedmiotu i czynności zdarzenia

`buttonClicked() {} - przycisk kliknięty`

`inputChanged() {} - input zmieniony`

Określenia dla metod, które nie są wywoływane przez zdarzenie

`deleteElement() {} - skasowanie elementu`

`addUser() {} - dodanie użytkownika`

`getWeather() {} - pobranie pogody`

Nazwy props przy przekazywaniu metod

```
handleClick = () => {
  ...
}

return (
 <LoginButton
    onClick={this.handleClick}
    className="btn__login"
 />
)
```

Nazwy props przy przekazywaniu metod

```
handleClick = () => {
  ...
}

return (
 <LoginButton
    click={this.handleClick}
    className="btn__login"
 />
)
```