React

TABLICE INFORMATYCZNE • Marcin Felczerek

Helion

PODSTAWOWE ZAŁOŻENIA

React jest *biblioteką* napisaną w ECMAScript (lecz możliwą do wykorzystania również w innych technologiach WWW), *służącą do budowania interfejsów użytkownika*.

Twórcy React definiują następujące cechy, wyróżniające tę bibliotekę:

- **Deklaratywna:** Budując interfejs w React, programista skupia się na jego deklarowanej strukturze, a nie przejściach czy aktualizacji elementów, czym zajmuje się sama biblioteka.
- **Komponentowa:** Interfejsy w React budowane są z deklaratywnych bloków zwanych komponentami, które mogą być powtórnie używane w innych miejscach interfejsu.
- **Niezależna od platformy:** React nie robi żadnych założeń na temat stosu technologii, które są wykorzystywane, zajmując się wyłącznie renderowaniem interfejsu i dając programiście dowolność w całej reszcie.

Dzięki tym cechom interfejsy zbudowane w React są przejrzyste, proste do tworzenia i łatwe w utrzymaniu. Programista może wykorzystać dowolne biblioteki i organizacje warstwy biznesowej, routingu czy interakcji z użytkownikiem.

Dołączanie React do projektu

Globalny plik JS

W tym podejściu React jest wczytywany niezależnymi znacznikami `<script>`, z CDN lub z lokalnego serwera, i instalowany jako globalna zależność. W wersji deweloperskiej możliwe jest również dodanie w ten sposób kompilatora JSX i kompilowanie kodu „w locie", jednak na produkcji kod strony powinien być skompilowany statycznie ze względu na czas ładowania i kwestie bezpieczeństwa. Kompilacja jest zbędna, jeśli projekt nie wykorzystuje składni JSX (rzadko spotykane).

```
<html>
  <head>
    ...
    <script src="https://unpkg.com/react@16/umd/react.
development.js"></script>
    <script src="https://unpkg.com/react-dom@16/umd/
react-dom.development.js"></script>
    <!-- Nie używać na produkcji: -->
    <script src="https://unpkg.com/babel-standa-
lone@6.15.0/babel.min.js"></script>
  </head>
  <body>
  ...
    <script type="text/babel">
    ... // kod React wstawić tutaj
        // lub za pomocą atrybutu src.
        // w trybie produkcyjnym kod powinien być
        // skompilowany na typ „application/javascript"
    </script>
  </body>
</html>
```

W wersji produkcyjnej skrypty React powinny wykorzystywać wersję *.production.*, a nie *.development.*. Skrypty mogą być również serwowane z lokalnego serwera zamiast z CDN, jeśli takie są wymogi projektu.

Użycie modułów i/lub bundlera

W tym scenariuszu React najczęściej instalowany jest jako lokalny moduł NPM:

```
> yarn add react react-dom
```

Wskazane jest także zainstalowanie transpilatora Babel i ustawień dla JSX:

```
> yarn add -D @babel/core @babel/cli @babel/preset-env
@babel/preset-react
```

… i skonfigurowanie go przez dodanie pliku *.babelrc* w projekcie:

```
{
  "presets": ["@babel/env", "@babel/preset-react"]
}
```

Wykorzystanie narzędzia create

Narzędzie `create-react-app` pozwala na szybkie skonfigurowanie projektu React z wykorzystaniem kompilatora Babel lub TypeScript oraz bundlera Webpack:

```
> npx create-react-app <nazwa aplikacji>
```

lub (dla managera Yarn):

```
> yarn create react-app <nazwa aplikacji>
```

Projekt posiada własny serwer deweloperski, automatycznie kompilujący pliki i wymuszający odświeżenie przeglądarki. Serwer uruchamiany jest poleceniem

```
> npm start
```

lub

```
> yarn start
```

w katalogu projektu.

Szczegóły działania narzędzia można znaleźć w dokumentacji: *https://facebook.github.io/create-react-app/*.

Podstawy

Element dokujący i komponent główny

Do działania React potrzebuje elementu, w którym zostanie wyświetlona zawartość stworzona przez bibliotekę. W przypadku strony WWW jest to element drzewa DOM. Element ten powinien być blokowy, pusty (jego zawartość i tak zostanie zastąpiona przez drzewo komponentów React) i posiadać atrybut ID:

```
<div id="react-korzen"></div>
```

Na stronie może być więcej niż jeden element dokujący, każdy będzie punktem dokowania niezależnie działającego drzewa komponentów React, jednak nie jest to często stosowane rozwiązanie.

React zakłada, że cały wyświetlany interfejs użytkownika jest zamknięty w jednym komponencie i ten komponent zostaje zamontowany w elemencie dokującym. Do wyświetlenia komponentu w drzewie DOM służy funkcja `ReactDOM.render(<element>, <węzeł DOM>)`:

```
let element = ....;//stwórz element React
let korzen = document.getElementById('react-korzen');
ReactDOM.render(element, korzen);
```

Węzeł DOM można dostarczyć na wiele sposobów, lecz najprostszym jest stworzenie elementu dokującego na stronie, nadanie mu atrybutu `id` i odnalezienie go funkcją `Document.prototype.getElementById`. Element musi być natomiast węzłem wirtualnego drzewa React, który może zostać stworzony na wiele różnych sposobów (opisanych dalej).

Drzewo elementów

Interfejs zbudowany w React składa się z **komponentów** — obiektów lub funkcji odpowiedzialnych za wyświetlenie jakiegoś fragmentu interfejsu. Każdy komponent może składać się z innych komponentów, a te ostatecznie (w przypadku strony WWW) składają się ze znaczników HTML, które z punktu widzenia React też są komponentami.

Aby komponent stał się częścią strony, musi zostać przekształcony w **element**, czyli węzeł *wirtualnego drzewa DOM*. Drzewo to jest niezmienne, stworzenie nowego stanu wymaga stworzenia nowego drzewa i zastąpienia nim starego (React odpowiada za to, by owa zmiana odbyła się wydajnie).

Aby utworzyć element (wraz z drzewem elementów potomnych), należy użyć funkcji `React.createElement(<komponent>, <atrybuty>, <elementy potomne>)`. Jej argumenty to:

- **Komponent**: Funkcja komponentu React **lub** nazwa (jako tekst). W drugim przypadku React stworzy znacznik HTML o podanej nazwie.
- **Atrybuty**: Obiekt, którego kluczami są nazwy atrybutów komponentu, a wartościami ich wartości. Wartości atrybutów komponentów React nie muszą być tekstem, chyba że tworzony jest element HTML.
- **Elementy potomne**: Tablica elementów React (wykreowanych tą samą funkcją, `React.createElement`) zagnieżdżonych w tworzonym elemencie. Nie wszystkie komponenty React pozwalają na elementy potomne; jeśli ich nie ma, można przekazać wartość `null`.

Stworzony element może być użyty jako element potomny innego elementu React lub wyświetlony w dokumencie HTML za pomocą funkcji `ReactDOM.render`, opisanej powyżej.

```
let element = React.createElement(Interfejs, { kolor:
'black' }, null);
```

JSX

JSX jest popularną alternatywą dla tworzenia drzew elementów za pomocą funkcji `React.createElement`. Jest to rozszerzenie składni ECMAScript (dostępne również w językach TypeScript, Flow i Reason), pozwalające na osadzenie w kodzie programu drzewa elementów XML:

```
let element = <Interfejs kolor="black" />;
```

- JSX wykorzystuje składnię języka XML (nie HTML!), jest wrażliwa na wielkość liter, wymaga cudzysłowów wokół wartości atrybutów i znaczników zamykających (lub skróconego zapisu `/>`).
- Kod JSX jest **wyrażeniem**, którego wartością jest `React.Element` (istotne w wypadku języków statycznie typowanych, jak Flow czy TypeScript). Może być użyte wszędzie tam, gdzie wymagany jest obiekt tego typu. W przypadku niejasności czy problemów z kompilacją należy umieścić wyrażenie JSX w nawiasach.
- **Nazwa** znacznika JSX jest interpretowana jako *literał ECMAScript*, jeśli *jest pisana wielką literą*. Taki literał (nazwa funkcji, nazwa klasy lub zmienna wskazująca na funkcję lub klasę) musi istnieć w miejscu wyrażenia, stworzony wcześniej lub zaimportowany z innego modułu.
- Nazwa znacznika JSX zapisana *małą literą* jest traktowana jako łańcuch tekstowy.
- Atrybuty JSX z reguły zapisywane są składnią *camelCase* zamiast składni z kreskami bądź małymi literami. Atrybut `class` musi być zapisywany jako `className` ze względu na kolizję ze słowem kluczowym ECMAScript.
- Atrybuty `key` i `ref` mają specjalne znaczenie (opisane dalej) i nie mogą być używane do innych celów.
- Każdy znacznik JSX jest przekształcany przez kompilator w wywołanie funkcji `React.createElement`.

```
// JSX
const element = <Container layout="vertical">
                    <Button type="round" color="red"/>
                    <span class="comment">Press
button</span>
                </Container>;

// „czysty" JS
const element = React.createElement(
    Container, // wielka litera - obiekt
    {layout: 'vertical'},
    [
        React.createElement(
            Button, // wielka litera - obiekt
            {
                type: 'round',
                color: 'red'
            }, null),
        React.createElement(
            'span', // mała litera - tekst
            {class: 'comment'},
            ['Press button'])
    ]
);
```

- JSX może zawierać nie tylko statyczny tekst, lecz również osadzone wyrażenia. Wyrażenie ma postać kodu ECMAScript osadzonego w nawiasach klamrowych. Wyrażenie może być częścią treści JSX lub wartością atrybutu (w tym wypadku *nie może* być ujęte w cudzysłowy). Jeśli jest to wartość atrybutu, wyrażenie musi mieć odpowiednią dla atrybutu wartość. Jeśli jest to treść, musi być ono typu `React.Element` **lub** tekstowe (w drugim wypadku nie powinno zawierać elementów HTML, gdyż zostaną one unieszkodliwione).

```
const numbers = ['jeden', 'dwa', 'trzy'];
const jsx = <select name="numery>
                { numbers.map((wart, ind) => (
                    <option value={ind + 1}>{wart}</
option>
                ) }
            </select>;
```

W powyższym przykładzie mamy wykorzystane zagnieżdżone wyrażenie JSX w funkcji `Array.prototype.map`, które tworzy nam listę elementów `<option>`.

Komponenty

Komponent React to obiekt odpowiedzialny za wyświetlenie części (bądź całości) interfejsu użytkownika. W JSX jest on reprezentowany przez znacznik XML o nazwie pisanej wielką literą.

Komponent React może być reprezentowany przez **funkcję** lub przez **klasę komponentu**.

Komponenty funkcyjne

Komponent funkcyjny jest zwykłą funkcją (czyli obiektem typu `Function`) ECMAScript o zdefiniowanej sygnaturze. Funkcja ta otrzymuje jeden parametr typu obiektowego, a zwracać powinna fragment drzewa elementów React.

```
function LabeledInput(props) {
    return <label className="labeledButton">
            {props.label}
            <input type="text" name={props.name}
                   value={props.value}
                   onChange={props.onChange}/>
           </label>;
}

let rnd = <LabeledInput label="Imie"
                        name="name" value={name}
                        onChange={onChangeHandler}/>;
```

Parametr funkcji komponentu jest tradycyjnie nazywany `props` i zawiera obiekt, którego kluczami są nazwy atrybutów komponentu, a wartościami ich wartości (jest to ten sam obiekt, który przekazywany jest jako drugi argument funkcji `React.createElement`). W nowych wersjach ECMAScript możliwe jest wykorzystanie destrukturyzacji:

```
function LabeledInput({label, name, value, onChange}) {
    return <label className="labeledButton">
            {label}
            <input type="text" name={name}
                   value={value}
                   onChange={onChange}/>
           </label>;
}
```

W językach statycznie typowanych (TypeScript, Flow, Reason) typ argumentu `props` może być statycznie sprawdzany na etapie kompilacji, zarówno w składni tradycyjnej, jak i JSX.

```
// @flow

interface LabeledInputProps {
    label: string,
    name: string,
    value?: string,
```