Lekcja 46: Silnik DOM JSX, część 1

W tej lekcji dowiemy się o fabrykach JSX i JSX. W ciągu ostatnich kilku lat prawie nie było o technologii bardziej dyskutowanej niż React. Cóż, może TypeScript! React to biblioteka do tworzenia interfejsu użytkownika i obsługi stanu bezpośrednio w JavaScript - żadna inna technologia nie jest potrzebna. Z wyjątkiem jednego: JSX. JSX to funkcja składni, która umożliwia pisanie poleceń w formacie HTML lub XML bezpośrednio w JavaScript. Twoje komponenty wyglądają mniej więcej tak:

```
export function Button() {

return <button>Click me</button>

}
```

Jeśli nigdy nie widziałeś czegoś takiego, a nawet gdybyś to widział, pierwszą reakcją może być: „JSX miesza HTML z moim JavaScriptem - to brzydkie!" Uwierz mi, to była moja pierwsza reakcja. Zapewniamy, że JSX nie łączy w sobie HTML i JavaScript. Oto, czym JSX nie jest:

• JSX nie jest językiem do tworzenia szablonów

• JSX to nie HTML

• JSX to nie XML

JSX tak wygląda, ale to nic innego jak cukier syntaktyczny.

JSX to wywołania funkcji

JSX przekłada się na czyste, zagnieżdżone wywołania funkcji. Sygnatura metody React w JSX to (element, properties, ... children), przy czym element jest albo komponentem React, albo string, properties to obiekt JS z kluczami i wartościami, a children są puste lub tablica z większą liczbą funkcji wezwania.

Więc:

```
<Button onClick={() => alert('YES')}>Click me</Button>
```

translates to:

```
React.createElement(Button, { onClick: () => alert('YES') }, 'Click me');
```

With nested elements, it looks something like this:

```
<Button onClick={() => alert('YES')}><span>Click me</span></Button>
```

co przekłada się na:

```
React.createElement(Button, { onClick: () => alert('YES') },

React.createElement('span', {}, 'Click me'));
```

Jest jedna konwencja: elementy z wielkich liter są tłumaczone na komponenty, elementy z małych liter na łańcuchy. Ten ostatni jest używany dla standardowych elementów HTML. Jakie są konsekwencje, zwłaszcza w porównaniu z szablonami?

• Nie ma kompilacji i analizowania szablonów w czasie wykonywania. Wszystko trafia bezpośrednio do wirtualnego DOM lub silnika układu pod spodem.

• Brak wyrażeń do oceny. Wszystko dookoła to JavaScript.

• Każda właściwość komponentu jest tłumaczona na klucz obiektu JSX. To pozwala nam na wpisanie ich sprawdzić.

TypeScript działa tak dobrze z JSX, ponieważ pod spodem znajduje się JavaScript. Więc wszystko wygląda jak XML, z wyjątkiem tego, że są to funkcje JavaScript. Jedno pytanie do naszych doświadczonych programistów internetowych: czy kiedykolwiek chcieliście pisać bezpośrednio do DOM, ale zrezygnowaliście, ponieważ jest tak nieporęczny? document.createElement ma dość łatwe API, ale musimy wykonać mnóstwo wywołań API DOM, aby uzyskać to, co możemy tak łatwo osiągnąć, pisząc HTML. JSX to rozwiązuje. Dzięki JSX masz przyjemną i znajomą składnię pisania elementów bez HTML.

Pisanie DOM z JSX

Wpisz TypeScript! TypeScript to pełnowymiarowy kompilator JSX. Dzięki TypeScript mamy możliwość zmiany fabryki JSX. W ten sposób TypeScript jest w stanie skompilować JSX dla React, Vue.js, Dojo… dowolnego frameworka używającego JSX w taki czy inny sposób. Implementacje wirtualnego DOM poniżej mogą się różnić, ale interfejs jest taki sam:

```
/**
* element: string or component
* properties: object or null
* ...children: null or calls to the factory
*/
function factory(element, properties, ...children) {
//...
}
```

Możemy użyć tej samej fabrycznej sygnatury metody nie tylko do pracy z wirtualnym DOM, ale także do pracy z prawdziwym DOM, tylko po to, aby mieć ładne API na górze document.createElement. Spróbujmy! Oto funkcje, które chcemy wdrożyć:

1. Przeanalizuj JSX do węzłów DOM, w tym atrybutów.

2. Mają proste elementy funkcjonalne dla większej kompozycyjności i elastyczności.

TypeScript musi wiedzieć, jak skompilować dla nas JSX. Wystarczy ustawić dwie właściwości w tsconfig.json.

```
{
“compilerOptions”: {
...
“jsx”: "react",
“jsxFactory”: "DOMcreateElement",
“noImplicitAny”: false
```

```
}

}
```

Zostawiamy to wzorcowi React JSX (sygnatura metody, o której mówiliśmy wcześniej), ale powiedz TypeScript, aby użył do tego naszej wkrótce utworzonej funkcji DOMcreateElement. Ponadto ustawiliśmy na razie noImplicitAny na false. Dzieje się tak, abyśmy mogli skupić się na implementacji i na późniejszym etapie dokonać poprawnego wpisania. Jeśli chcemy używać JSX, musimy zmienić nazwy naszych plików .ts na .tsx. Najpierw wdrażamy naszą funkcję fabryczną. Jego specyfikacja:

1. Jeśli element jest funkcją, to jest składnikiem funkcjonalnym. Nazywamy tę funkcję (oczywiście przekazując properties i children) i zwracamy wynik. Oczekujemy wartości zwracanej typu Node.

2. Jeśli element jest łańcuchem, parsujemy zwykły węzeł.

```
function DOMcreateElement(

element, properties, ...children

) {

if(typeof element === 'function') {

return element({

...nonNull(properties, {}),

children

});

}

return DOMparseNode(

element,

properties,

children

);

}

/**

* A helper function that ensures we won't work with null values

*/

function nonNull(val, fallback) { #

return Boolean(val) ? val : fallback

};
```

Następnie analizujemy zwykłe węzły.

1. Tworzymy element i stosujemy wszystkie właściwości z JSX do tego węzła DOM. Oznacza to, że wszystkie właściwości, które możemy przekazać, są częścią HTMLElement.

2. Jeśli to możliwe, dołączamy wszystkie dzieci.

```
function DOMparseNode(element, properties, children) {

const el = Object.assign(

document.createElement(element),

properties

);

DOMparseChildren(children).forEach(child => {

el.appendChild(child);

});

return el;

}
```

Na koniec tworzymy funkcję obsługującą children. Dzieci mogą albo:

1. Wywoływać fabryczną funkcję DOMcreateElement, zwracając HTMLElement.

2. Treść tekstową, zwracającą tekst.

```
function DOMparseChildren(children) {

return children.map(child => {

if(typeof child === 'string') {

return document.createTextNode(child);

}

return child;

})

}
```

Podsumowując:

1. Funkcja fabryczna przyjmuje elementy. Elementy mogą być typu string lub funkcją.

2. Element funkcyjny to komponent. Wywołujemy tę funkcję, ponieważ spodziewamy się uzyskać z niej DOM Node. Jeśli komponent funkcji zawiera również więcej komponentów funkcyjnych, w pewnym momencie zostaną one przekształcone w węzeł DOM.

3.Jeśli element jest łańcuchem, tworzymy zwykły DOM Node. W tym celu nazywamy document.createElement.

4. Wszystkie właściwości są przekazywane do nowo utworzonego Node. Teraz możesz już zrozumieć, dlaczego React ma coś w rodzaju className zamiast class. Dzieje się tak, ponieważ interfejs DOM API

znajdujący się pod spodem to także className.onClick to jednak camelCase, co wydaje mi się trochę dziwne.

5.Nasza implementacja zezwala tylko na właściwości DOM Node w naszym JSX, ze względu na to proste przekazywanie właściwości.

6. Jeśli nasz komponent ma dzieci (zebrane razem w tablicy), usuwamy również elementy potomne i dołączamy je.

7.Children może być wywołaniem DOMcreateElement, ostatecznie rozwiązywaniem w węźle DOM, lub prostym ciągiem znaków.

8.Jeśli jest to ciąg, tworzymy tekst. Teksty mogą być również dołączane do DOM Node. To wszystko! Spójrz na następujący przykład kodu:

```
const Button = ({ msg }) => {

return <button onclick={() => alert(msg)}>

<strong>Click me</strong>

</button>

}

const el = (

<div>

<h1 className=”what”>Hello world</h1>

<p>

Lorem ipsum dolor sit, amet consectetur adipisicing elit. Quae sed consectetur placeat veritatis illo vitae quos aut unde doloribus, minima eveniet et eius voluptatibus minus aperiam sequi asperiores, odio ad?

</p>

<Button msg='Yay' />

<Button msg='Nay' />

</div>

)

document.body.appendChild(el);
```

Nasza implementacja JSX zwraca DOM Node ze wszystkimi jego elementami podrzędnymi. Możemy nawet użyć do tego komponentów funkcyjnych. Zamiast szablonów pracujemy bezpośrednio z DOM, ale API jest dużo przyjemniejsze! Więc czego brakuje? Typowanie właściwości!