O'REILLY®

Wydanie II

React w działaniu

Tworzenie aplikacji internetowych

Helion

Stoyan Stefanov

Tytuł oryginału: React: Up & Running: Building Web Applications, 2nd Edition

Tłumaczenie: Joanna Zatorska

ISBN: 978-83-8322-039-0

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/reacw2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 2.

Życie komponentu

Wiesz już, jak używać gotowych komponentów DOM, dlatego możesz się zabrać za tworzenie własnych.

Własny komponent można zdefiniować na dwa sposoby. W obydwu przypadkach uzyskasz taki sam rezultat, ale za pomocą innej składni:

- korzystając z funkcji (są to *komponenty funkcyjne*);
- korzystając z klasy dziedziczącej po klasie `React.Component` (są to *komponenty klasowe*).

Niestandardowy komponent funkcyjny

Oto przykład komponentu funkcyjnego:

```
const MyComponent = function() {
   return 'Jestem zupełnie niestandardowy';
};
```

Ale przecież to jest zwykła funkcja! To prawda, niestandardowy komponent jest funkcją, która zwraca potrzebny element interfejsu. W tym przypadku tym elementem jest jedynie tekst, ale często potrzebne są bardziej złożone elementy, a najczęściej kompozycja komponentów. Oto przykład użycia elementu span zawierającego tekst:

```
const MyComponent = function() {
   return React.createElement('span', null, ' Jestem zupełnie niestandardowy');
};
```

Z nowego komponentu można korzystać w aplikacji tak samo jak z komponentów DOM w rozdziale 1., ale w tym przypadku trzeba *wywołać* funkcję definiującą komponent:

```
ReactDOM.render(
   MyComponent(),
   document.getElementById('app')
);
```

Wynik renderowania niestandardowego komponentu jest przedstawiony na rysunku 2.1.

Rysunek 2.1. Pierwszy niestandardowy komponent (02.01.custom-functional.html w pakiecie dołączonym do książki)

Wersja JSX

Ten sam przykład z użyciem składni JSX jest nieco czytelniejszy. Tak wygląda definiowanie komponentu:

```
const MyComponent = function() {
   return <span>Jestem zupełnie niestandardowy</span>;
};
```

Oto jak można skorzystać z komponentu za pomocą składni JSX, niezależnie od tego, jak został zdefiniowany (za pomocą składni JSX czy standardowo):

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

Zwróć uwagę na to, że odwrotny ukośnik w samozamykającym znaczniku `<MyComponent />` nie jest opcjonalny. Dotyczy to również elementów HTML używanych w składni JSX. Znaczniki `<br>` i `<img>` nie zadziałają, musisz je zamknąć: `<br/>` i `<img/>`.

Niestandardowy komponent klasowy

Komponent można też utworzyć, definiując klasę dziedziczącą po klasie `React.Component` oraz implementującą funkcję `render()`:

```
class MyComponent extends React.Component {
   render() {
      return React.createElement('span', null, 'Jestem zupełnie niestandardowy');
         // lub za pomocą składni JSX:
         // return <span>Jestem zupełnie niestandardowy</span>;
   }
}
```

Renderowanie komponentu na stronie odbywa się następująco:

```
ReactDOM.render(
   React.createElement(MyComponent),
   document.getElementById('app')
);
```

Jeśli korzystasz ze składni JSX, nie musisz wiedzieć, jak został zdefiniowany komponent (za pomocą klasy czy funkcji). W obydwu przypadkach komponentu używa się tak samo:

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

Którą składnię wybrać?

Być może się zastanawiasz, którą opcję (JSX, czysty JavaScript, komponent klasowy czy funkcyjny) wybrać. Składnia JSX jest najpopularniejsza. O ile używanie kodu XML w JavaScripcie nie budzi w Tobie oporów, najłatwiejszą opcją, wymagającą najmniej kodu, jest składnia JSX. Będziemy z niej korzystać już do końca książki, chyba że trzeba będzie wyjaśnić niektóre koncepcje. Dlaczego zatem omawiam inne opcje? Otóż warto pamiętać, że *istnieją* inne możliwości i że JSX nie jest żadnym magicznym narzędziem, lecz nową warstwą składniową, która przekształca XML w czyste wywołania funkcji JavaScript, takie jak `React.createElement()`, przed wysłaniem kodu do przeglądarki.

Jak wybrać między komponentami *klasowym* a *funkcyjnymi*? To kwestia preferencji. Jeśli wolisz programowanie zorientowane obiektowo (ang. *object-oriented programming* — OOP) i lubisz korzystać z klas, wybierz tę opcję. Komponenty funkcyjne wymagają nieco mniejszej mocy obliczeniowej CPU i mniej kodu. Sprawiają również wrażenie bardziej natywnego kodu javascriptowego. *Klasy* nie istniały we wczesnych wersjach języka JavaScript, wprowadzono je później i obecnie są lukrem składniowym, wykorzystującym funkcje i prototypy.

Jeśli chodzi o Reacta, za pomocą komponentów funkcyjnych nie można było osiągnąć dawniej tyle, ile za pomocą klas. To się zmieniło po wprowadzeniu koncepcji *hooków*, które niebawem poznasz. Przyszłości nie da się przewidzieć, ale można podejrzewać, że rozwój Reacta będzie zmierzał w kierunku komponentów funkcyjnych. Jednak najprawdopodobniej komponenty klasowe będą nadal dość długo w użyciu. W tej książce używam obydwu metod, chociaż może zauważysz nieco większą skłonność ku komponentom funkcyjnym. Po co zatem zajmować się klasami (takie pytanie zadała mi większość korektorów merytorycznych tej książki)?

Otóż w wielu istniejących aplikacjach i materiałach szkoleniowych korzysta się z klas. W czasie gdy piszę tę książkę, klasy są używane w większości przykładów w oficjalnej dokumentacji Reacta. Zatem uważam, że czytelnicy powinni poznać obydwa sposoby, aby móc zrozumieć każdy kod, z którym się zetkną, również składnię komponentów klasowych.

Właściwości

Renderowanie interfejsu *zakodowanego na sztywno* w komponentach niestandardowych jest poprawną metodą, która znajduje zastosowanie. Jednak komponenty mogą przyjmować *właściwości* i na podstawie ich wartości modyfikować sposób renderowania lub swoje zachowanie. Przypomnij

sobie element <a> w składni HTML-a i jego zachowanie w zależności od wartości atrybutu href. Podobnie działają właściwości w Reakcie (oraz w składni JSX).

W przypadku komponentów klasowych wszystkie właściwości są dostępne poprzez obiekt this.props. Spójrzmy na taki przykład:

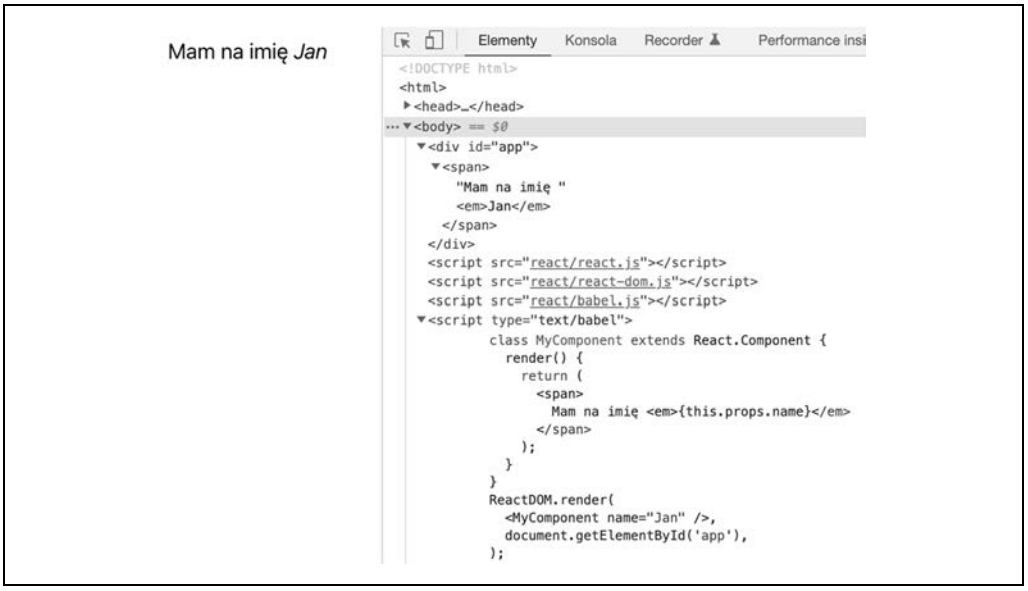
```
class MyComponent extends React.Component {
   render() {
      return <span>Mam na imię <em>{this.props.name}</em></span>;
   }
}
```

Jak widać w tym przykładzie, w składni JSX w nawiasach klamrowych możesz umieścić wartości (i wyrażenia) JavaScriptu. Rozwinę tę koncepcję w dalszej części książki.

Przekazywanie wartości właściwości name podczas renderowania komponentu wygląda następująco:

```
ReactDOM.render(
   <MyComponent name="Jan" />,
   document.getElementById('app')
);
```

Wynik działania powyższego kodu możesz zobaczyć na rysunku 2.2.



Rysunek 2.2. Używanie właściwości komponentu (02.05.this.props.html)

Trzeba pamiętać, że obiekt this.props jest tylko do odczytu. Właściwości są przydatne do przenoszenia konfiguracji od komponentu nadrzędnego do potomnego, ale nie powinny służyć do przechowywania wartości. Jeśli czujesz pokusę, aby ustawić właściwość w obiekcie this.props, skorzystaj z dodatkowych zmiennych lub z właściwości obiektu specyfikacji komponentu (na przykład this.thing zamiast this.props.thing).

Właściwości w komponentach funkcyjnych

W komponentach funkcyjnych nie ma obiektu `this` (w przypadku *czystego* JavaScriptu) albo `this` odnosi się do globalnego obiektu (powiedzmy, w *niechlujnym* trybie). W tym przypadku zamiast obiektu `this.prop` mamy do dyspozycji obiekt `props`, który jest przekazywany jako pierwszy argument funkcji:

```
const MyComponent = function(props) {
   return <span>Mam na imię <em>{props.name}</em></span>;
};
```

Typowym wzorcem jest użycie *przypisania destrukturyzacyjnego* JavaScriptu i przypisanie wartości właściwości do lokalnych zmiennych. Innymi słowy, poprzedni przykład można zapisać następująco:

```
// 02.07.props.destructuring.html w plikach dołączonych do książki
const MyComponent = function({name}) {
   return <span>Mam na imię <em>{name}</em></span>;
};
```

Możesz korzystać z dowolnej liczby właściwości. Jeśli na przykład potrzebujesz dwóch właściwości (`name` i `job`), możesz ich użyć w następujący sposób:

```
// 02.08.props.destruct.multi.html w plikach dołączonych do książki
const MyComponent = function({name, job}) {
   return <span> Mam na imię <em>{name}</em>, pracuję jako {job}</span>;
};
ReactDOM.render(
   <MyComponent name="Jan" job="inżynier"/>,
   document.getElementById('app')
);
```

Domyślne właściwości

Komponent może korzystać z wielu właściwości, ale czasem niektóre z nich mogą mieć domyślne wartości, które sprawdzą się w najpopularniejszych przypadkach. Domyślne wartości właściwości, zarówno w komponentach funkcyjnych, jak i klasowych, można określić za pomocą właściwości `defaultProps`.

Komponent funkcyjny:

```
const MyComponent = function({name, job}) {
   return <span>Mam na imię <em>{name}</em>, pracuję jako {job}</span>;
};
   MyComponent.defaultProps = {
   job: 'inżynier',
};
ReactDOM.render(
   <MyComponent name="Jan" />,
   document.getElementById('app')
);
```

Komponent klasowy:

```
class MyComponent extends React.Component {
   render() {
      return (
```

```
            <span> Mam na imię <em>{this.props.name}</em>,
            pracuję jako {this.props.job}</span>
        );
    }
}
MyComponent.defaultProps = {
    job: 'inżynier',
};
ReactDOM.render(
    <MyComponent name="Jan" />,
    document.getElementById('app')
);
```

Oto wynik w obydwu przypadkach:

```
Mam na imię Jan, pracuję jako inżynier
```

Zauważ, że wartość zwracana w instrukcji `return` w metodzie `render()` znajduje się w nawiasach. Wynika to z mechanizmu automatycznego wstawiania średnika (ang. *automatic semi-colon insertion* — ASI) w JavaScripcie. Instrukcja `return`, po której następuje nowy wiersz, odpowiada instrukcji `return;`, czego chcemy uniknąć. Po umieszczeniu zwracanych wartości w nawiasach kod będzie lepiej sformatowany i nadal poprawny.

Stan

Dotychczasowe przykłady były dość statyczne (inaczej: bezstanowe). Miały na celu jedynie zaprezentować ideę bloków budulcowych, z których można skomponować własny interfejs użytkownika. React jednak objawia swoją moc dopiero w przypadku zmian dotyczących danych aplikacji (czyli w sytuacjach problematycznych dla przeglądarek stosujących standardowe sposoby obsługi i utrzymania hierarchii DOM). W bibliotece React wykorzystywana jest koncepcja *stanu*, czyli danych, na podstawie których komponent renderuje sam siebie. Gdy stan ulega zmianie, React przebudowuje interfejs użytkownika bez konieczności Twojej interwencji. Dzięki temu, po utworzeniu interfejsu poprzez metodę `render()` (lub w funkcji renderującej w przypadku komponentu funkcyjnego), musisz się zająć jedynie aktualizacją danych. Nie musisz się wcale przejmować zmianami w interfejsie użytkownika. W końcu metoda lub funkcja renderująca otrzymała już projekt wyglądu komponentu.

Nie należy się obawiać słowa „bezstanowy”. Komponentami bezstanowymi znacznie łatwiej się zarządza. Chociaż zwykle zaleca się korzystanie z komponentów bezstanowych, aplikacje stają się tak skomplikowane, że wprowadzenie stanu jest konieczne.

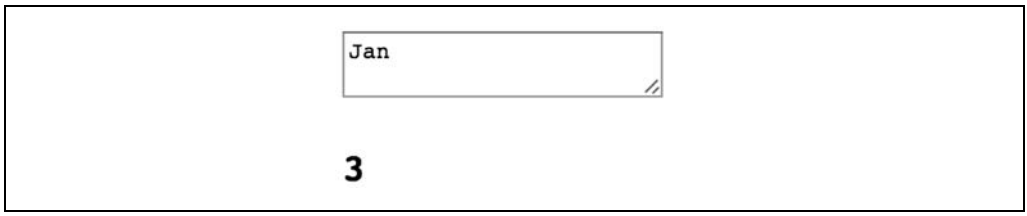
Podobnie jak właściwości dostępne są poprzez obiekt `this.props`, tak *odczyt* stanu jest możliwy za pośrednictwem obiektu `this.state`. Aby *uaktualnić* stan, skorzystaj z metody `this.setState()`. Po wywołaniu `this.setState()` React wywoła metodę `render()` komponentu (i wszystkich komponentów potomnych) i uaktualni interfejs.

Po wywołaniu metody `this.setState()` aktualizacje interfejsu użytkownika są wykonywane poprzez mechanizm kolejkowania, który efektywnie grupuje zmiany. Dlatego bezpośrednie uaktualnianie obiektu `this.state` może wywoływać nieoczekiwane skutki i nie powinno mieć miejsca.

Podobnie jak w przypadku obiektu `this.props`, traktuj `this.state` jako obiekt tylko do odczytu, ze względów semantycznych, ale także ze względu na nieoczekiwane skutki innego podejścia. Nie należy również samodzielnie wywoływać metody `this.render()`. Pozostaw to w gestii biblioteki React, która pogrupuje zmiany, wyznaczy ich niezbędne minimum i w odpowiednim momencie wywoła metodę `render()`.

Komponent obszaru tekstowego

Zbuduj teraz nowy komponent — obszar tekstowy, który będzie zliczać wpisane w nim znaki (rysunek 2.3).



Rysunek 2.3. Wynik działania niestandardowego komponentu obszaru tekstowego

Nowy komponent możesz (podobnie jak inni użytkownicy tego komponentu) wykorzystać w następujący sposób:

```
ReactDOM.render(
   <TextAreaCounter text="Jan" />,
   document.getElementById('app')
);
```

Teraz możesz zaimplementować komponent. Zacznij od utworzenia bezstanowej wersji, która nie będzie obsługiwać aktualizacji, ponieważ nie różni się ona zbytnio od wcześniejszych przykładów:

```
class TextAreaCounter extends React.Component {
   render() {
      const text = this.props.text;
      return (
         <div>
            <textarea defaultValue={text}/>
            <h3>{text.length}</h3>
         </div>
      );
   }
}
TextAreaCounter.defaultProps = {
   text: 'Policz, ile razy naciskam klawisz',
};
```

Być może nie uszło Twojej uwadze, że obszar tekstowy `<textarea>` w poprzednim listingu przyjmuje właściwość `defaultProperty`, w przeciwieństwie do tekstowego obiektu potomnego w zwykłym kodzie HTML. Otóż jeśli chodzi o formularze, między Reactem a kodem HTML istnieją pewne różnice. Zostaną one omówione w dalszej części książki, a na razie nie musisz się przejmować, ponieważ jest ich niewiele. Zapewne zauważysz, że mają one sens i są ułatwieniem w pracy programistów.

Jak widać, komponent `TextAreaCounter` przyjmuje opcjonalną wartość tekstową `text`, a następnie renderuje element `textarea` z podaną wartością oraz element `<h3>` wyświetlający długość tekstu, czyli wartość właściwości `length`. Jeśli nie podamy wartości `text`, zostanie użyta wartość `Policz, ile razy naciskam klawisz`.

Komponent ze stanem

Kolejny krok polega na przekształceniu tego *bezstanowego* komponentu w komponent *ze stanem*. Innymi słowy: potrzebny jest nam komponent, który będzie utrzymywał pewne dane (stan) i używał ich podczas początkowego renderowania i podczas uaktualniania (czyli ponownego renderowania) w wyniku zmiany danych.

Najpierw musisz ustawić początkowy stan w konstruktorze klasy za pomocą składni `this.state`. Pamiętaj, że stan można ustawiać bezpośrednio tylko w konstruktorze. W pozostałych przypadkach można go ustawiać jedynie za pomocą metody `this.setState()`.

Inicjalizacja zmiennej `this.state` jest niezbędna, gdyż w przeciwnym razie późniejsze odczyty wartości `this.state` w metodzie `render()` się nie powiodą.

W tym przypadku nie ma potrzeby inicjalizowania zmiennej `this.state.text`, ponieważ zawsze można skorzystać z właściwości `this.prop.text` (zajrzyj do pliku *02.12.this.state.html* dołączonego do książki):

```
class TextAreaCounter extends React.Component {
   constructor() {
      super();
      this.state = {};
   }
   render() {
      const text = 'text' in this.state ? this.state.text : this.props.text;
      return (
         <div>
           <textarea defaultValue={text} />
           <h3>{text.length}</h3>
        </div>
      );
   }
}
```

Zanim wywołasz `this` w konstruktorze, musisz wywołać `super()`.

Dane utrzymywane przez ten komponent dotyczą jedynie tekstu znajdującego się w obszarze tekstowym, dlatego stan ma tylko jedną właściwość, o nazwie `text`, która jest dostępna poprzez wywołanie `this.state.text`. Później trzeba uaktualnić stan. W tym celu można skorzystać z metody pomocniczej:

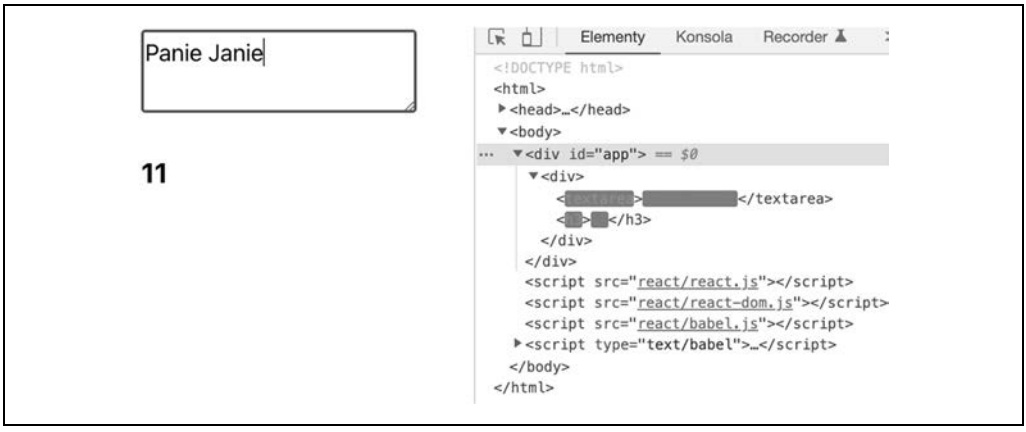
```
onTextChange(event) {
   this.setState({
      text: event.target.value,
   });
},
```

Stan zawsze należy uaktualnić, wywołując metodę `this.setState()`, która pobiera obiekt i scala go z istniejącymi danymi w obiekcie `this.state`. Jak można się domyślić, `onTextChange()` nasłuchuje zdarzeń i przyjmuje obiekt zdarzenia `event`, a następnie pobiera z niego tekst wpisany w obszarze tekstowym.

Pozostało nam jeszcze uaktualnienie metody `render()`, aby skonfigurować metodę obsługi zdarzeń:

```
render() {
   const text = 'text' in this.state ? this.state.text : this.props.text;
   return (
      <div>
         <textarea
            value={text}
            onChange={event => this.onTextChange(event)}
          />
          h3>{text.length}</h3>
      </div>
   );
}
```

Gdy użytkownik wpisze tekst w obszarze tekstowym, wartość licznika zostanie uaktualniona zgodnie z wpisanym tekstem (rysunek 2.4).



Rysunek 2.4. Wpisywanie w polu textarea (02.12.this.state.html)

Zwróć uwagę na to, że element `<textarea defaultValue...>` z wcześniejszego fragmentu kodu ma teraz postać `<textarea value...>`. Ta zmiana wynika ze sposobu obsługi pól wejściowych w języku HTML, których stan jest utrzymywany przez przeglądarkę. Jednak React radzi sobie z tym lepiej.

W tym przykładzie implementacja metody `onChange` oznacza, że element `textarea` jest teraz *kontrolowany* przez Reacta. Więcej informacji o *kontrolowanych komponentach* znajdziesz w dalszej części książki.

Uwaga na temat zdarzeń DOM

Aby uniknąć wszelkich nieporozumień, należy wyjaśnić kilka aspektów dotyczących następującego wiersza:

```
onChange={event => this.onTextChange(event)}
```

React wykorzystuje własny system *zdarzeń syntetycznych*, ze względu na wydajność, a także wygodę i bezpieczeństwo. Łatwiej Ci będzie zrozumieć to podejście, jeśli się zastanowisz, jak odbywa się to w świecie DOM.

Obsługa zdarzeń w dawnych czasach

Bardzo wygodnie jest obsługiwać *zdarzenia w wierszu*, jak poniżej:

```
<button onclick="doStuff">
```

Chociaż jest to wygodne i łatwe do odczytu (nasłuchiwanie zdarzeń odbywa się w tym samym miejscu, w którym znajduje się kod interfejsu użytkownika), takie rozproszenie obsługi zdarzeń w kodzie nie jest zbyt wydajne. Trudno jest też obsłużyć kilka zdarzeń w przycisku, szczególnie jeśli przycisk jest „komponentem" zdefiniowanym przez kogoś innego lub wchodzi w skład biblioteki i nie chcesz „naprawiać" ani tworzyć drugiej wersji tego kodu. To dlatego w świecie DOM używa się metody `element.addEventListener` do definiowania metod nasłuchujących zdarzeń (co prowadzi do utworzenia potrzebnego kodu w co najmniej dwóch miejscach) oraz *delegowania zdarzeń* (aby poradzić sobie z problemami wydajnościowymi). Delegowanie zdarzeń oznacza nasłuchiwanie zdarzeń w węźle nadrzędnym, na przykład w `<div>`, który zawiera wiele przycisków, i konfigurowanie obsługi zdarzeń dla wszystkich przycisków, a nie osobnej obsługi dla poszczególnych przycisków. Zatem *delegujemy* obsługę zdarzeń do elementu nadrzędnego.

Korzystając z techniki delegowania zdarzeń, można na przykład napisać taki kod:

```
<div id="parent">
   <button id="ok">OK</button>
   <button id="cancel">Anuluj</button>
</div>

<script>
document.getElementById('parent').addEventListener('click', function(event) {
   var button = event.target;

   // wykonaj różne zadania, w zależności od kliknięty przycisku
   switch (button.id) {
      case 'ok':
         console.log('OK!');
         break;
      case 'cancel':
         console.log('Anuluj');
```

```
            break;
        default:
            new Error('Przycisk o nieznanym ID');
    };
});
</script>
```

Powyższy kod działa poprawnie i jest wystarczająco wydajny, ale ma pewne niedogodności:

- Deklarowanie metody nasłuchującej jest oddzielone od komponentu interfejsu użytkownika, co sprawia, że kod jest trudniejszy do odnalezienia i debugowania.
- Użycie delegacji i konstrukcji `switch` tworzy niepotrzebnie skomplikowany kod, jeszcze zanim przejdziesz do wykonania właściwego zadania (w tym przypadku reagowania na kliknięcie przycisku).
- Obsługa niespójności między różnymi przeglądarkami (tutaj pominięta) wymaga wydłużenia tego kodu.

Niestety, zanim udostępnisz ten kod w produkcyjnej wersji dla użytkowników, musisz zadbać o kilka dodatków, które zapewnią wsparcie wszystkich przeglądarek:

- Oprócz `addEventListener` potrzebujesz też `attachEvent`.
- Na początku funkcji obsługi zdarzeń musisz użyć wiersza `const event = event || window.event;`.
- Musisz użyć wiersza `const button = event.target || event.srcElement;`.

Wszystko to jest niezbędne i wystarczająco irytujące, by ostatecznie zdecydować się na skorzystanie z gotowej biblioteki do obsługi zdarzeń. Ale po co dodawać kolejną bibliotekę (i uczyć się kolejnych API), skoro React umożliwia rozwiązanie problemów związanych z obsługą zdarzeń?

Obsługa zdarzeń w bibliotece React

React wykorzystuje *zdarzenia syntetyczne*, aby opakować i znormalizować zdarzenia przeglądarki, a w rezultacie zapewnić spójność zachowań we wszystkich przeglądarkach. Zawsze możesz wykorzystać to, że `event.target` jest dostępny we wszystkich przeglądarkach. Dzięki temu fragment kodu `TextAreaCounter` wykorzystuje jedynie wartość `event.target.value`, a aplikacja działa poprawnie. Oznacza to również, że API potrzebne do anulowania zdarzeń jest takie samo we wszystkich przeglądarkach. Innymi słowy: `event.stopPropagation()` i `event.preventDefault()` zadziałają także w starszych wersjach przeglądarki Internet Explorer.

Wspomniana składnia ułatwia utrzymywanie kodu interfejsu użytkownika i obsługi zdarzeń w jednym miejscu. Zapis ten przypomina dawne metody obsługi zdarzeń w wierszu, lecz mechanizm działania w bibliotece React jest inny i ze względu na wydajność opiera się na delegowaniu zdarzeń.

W bibliotece React nazwy funkcji obsługi zdarzeń są formatowane w notacji camelCase, dlatego zamiast nazwy `onclick` występuje `onClick`.

Jeśli z jakiegoś powodu potrzebujesz dostępu do oryginalnego zdarzenia przeglądarki, możesz go uzyskać, wywołując `event.nativeEvent`, lecz prawdopodobnie nigdy do tego nie dojdzie.

Jeszcze jedna uwaga: zdarzenie `onChange` (użyte w przykładzie z obszarem tekstowym) zachowuje się zgodnie z oczekiwaniami — jest wywoływane, gdy użytkownik coś wpisze, w przeciwieństwie do zakończenia wpisywania i opuszczenia pola, co jest zwykłym zdarzeniem w DOM.

Składnia obsługi zdarzeń

W ostatnim przykładzie użyliśmy funkcji strzałkowej do wywołania funkcji pomocniczej `onTextChange`, służącej do obsługi zdarzenia:

```
onChange={event => this.onTextChange(event)}
```

Jest to konieczne, ponieważ krótsza wersja `onChange={this.onTextChange}` nie zadziała. Inny sposób polega na zastosowaniu metody `bind`:

```
onChange={this.onTextChange.bind(this)}
```

Istnieje jeszcze jeden popularny sposób, polegający na przypisaniu wszystkich metod obsługi zdarzeń w konstruktorze:

```
constructor() {
   super();
   this.state = {};
   this.onTextChange = this.onTextChange.bind(this);
}
// ....
<textarea
   value={text}
   onChange={this.onTextChange}
/>
```

Widać tu sporo nadmiarowego kodu, ale dzięki temu funkcję obsługi zdarzeń przypisujemy tylko raz, a nie podczas każdego wywołania metody `render()`. W ten sposób zmniejszamy zużycie pamięci przez aplikację.

Ten popularny wzorzec porzucono, gdy tylko stało się możliwe używanie funkcji jako właściwości klas w JavaScripcie.

Wcześniej:

```
class TextAreaCounter extends React.Component {
   constructor() {
      // ...
      this.onTextChange = this.onTextChange.bind(this);
   }

   onTextChange(event) {
      // ...
   }
}
```

Później:

```
class TextAreaCounter extends React.Component {
   constructor() {
      // ...
   }
```

```
        onTextChange = (event) => {
            // …
        };
    }
```

Kompletny przykład znajduje się w pliku *02.12.this.state2.html* dołączonym do książki.

Props kontra state

Wiesz już, że w metodzie `render()`, która służy do wyświetlania komponentu, masz dostęp do obiektów `this.props` i `this.state`. Być może zastanawiasz się, kiedy należy użyć pierwszego, a kiedy drugiego.

Właściwości są mechanizmem przeznaczonym dla świata zewnętrznego (użytkowników komponentu), pozwalającym skonfigurować komponent. Stan umożliwia utrzymywanie danych wewnętrznych. Posługując się analogią ze świata zorientowanego obiektowo: `this.props` przypomina argumenty przekazywane do konstruktora klasy, a `this.state` jest zestawem właściwości prywatnych.

Należy dzielić kod aplikacji w taki sposób, aby uzyskać mniej komponentów *ze stanem* i więcej komponentów *bezstanowych*.

Props w stanie początkowym: antywzorzec

Wcześniej przedstawiłem przykład użycia `this.props` w celu ustawienia początkowej wartości obiektu `this.state`:

```
// ostrzeżenie: antywzorzec
this.state = {
    text: props.text,
};
```

Jednak takie użycie jest uważane za antywzorzec. Najlepiej stosować taką kombinację `this.state` i `this.props`, która najlepiej nadaje się do zbudowania interfejsu w metodzie `render()`. Czasem jednak chcesz pobrać wartość przekazaną do komponentu i na jej podstawie skonstruować stan początkowy. Jest to poprawne, ale programista wywołujący Twój komponent może oczekiwać, że właściwość (`text` w poprzednim przykładzie) będzie zawsze mieć najnowszą wartość, a wspomniany sposób narusza to oczekiwanie. Aby jasno zdefiniować oczekiwania, wystarczy prosta zmiana nazewnictwa — można na przykład zastosować właściwość o nazwie `defaultText` lub `initialValue` zamiast `text`.

W rozdziale 4. opisuję sposób implementacji wejścia i obszarów tekstowych w bibiotece React, w którym użytkownicy mogą mieć oczekiwania wynikające z wcześniejszych doświadczeń z HTML.

Dostęp do komponentu z zewnątrz

Nie zawsze mamy ten luksus, że możemy rozpoczynać tworzenie nowej aplikacji za pomocą Reacta od podstaw. Czasem musisz edytować kod istniejącej aplikacji lub strony WWW i zmigrować kolejne elementy Reacta. Na szczęście React został tak zaprojektowany, aby działać z dowolnym

istniejącym kodem. Twórcy Reacta nie mogli przecież zatrzymać świata i przepisać całej ogromnej aplikacji (Facebook) zupełnie od podstaw, szczególnie na wczesnym etapie rozwoju tej biblioteki.

Jednym ze sposobów na komunikację aplikacji utworzonej z użyciem Reacta ze światem zewnętrznym jest uzyskanie referencji do komponentu renderowanego w metodzie `ReactDOM.render()` i jej użycie na zewnątrz komponentu:

```
const myTextAreaCounter = ReactDOM.render(
   <TextAreaCounter text="Jan" />,
   document.getElementById('app')
);
```

Teraz za pomocą zmiennej `myTextAreaCounter` możesz skorzystać z tych samych metod i właściwości, do których masz zwykły dostęp poprzez `this` wewnątrz komponentu. Możesz nawet używać komponentu w konsoli JavaScriptu (rysunek 2.5).

Rysunek 2.5. Dostęp do wyrenderowanego komponentu poprzez referencję

W tym przykładzie `myTextAreaCounter.state` sprawdza bieżący stan (początkowo pusty), `myTextAreaCounter.props` sprawdza właściwości, a poniższy wiersz ustawia nowy stan:

```
myTextAreaCounter.setState({text: "Witaj, zewnętrzny świecie!"});
```

Ten wiersz pozwala uzyskać referencję do głównego węzła macierzystego DOM, utworzonego przez Reacta:

```
const reactAppNode = ReactDOM.findDOMNode(myTextAreaCounter);
```

Jest to pierwszy potomek elementu `<div id="app">`, czyli miejsce, w którym React będzie czynić swoją magię.

Spoza komponentu masz dostęp do całego API komponentu. Korzystaj jednak ostrożnie z tego przywileju, a najlepiej nie korzystaj wcale. Być może chcesz zmodyfikować stan komponentów napisanych przez innych programistów, aby je w ten sposób „naprawić", ale takie postępowanie narusza oczekiwania i ostatecznie prowadzi do błędów, ponieważ komponent nie przewiduje takich interwencji.

Metody cyklu życia

W bibliotece React dostępnych jest wiele metod *cyklu życia*. Można ich używać do nasłuchiwania zmian w komponencie, jeśli mamy do czynienia z modyfikacjami hierarchii DOM. Oto etapy cyklu życia komponentu:

Montowanie

Komponent jest dodawany do hierarchii DOM.

Aktualizowanie

Komponent jest akutalizowany w wyniku wywołania metody `setState()` i (lub) zmiany właściwości przekazanych do komponentu.

Odmontowywanie

Komponent jest usuwany z hierarchii DOM.

React wykonuje swoje zadanie przed aktualizacją hierarchii DOM. Jest to *etap renderowania*. Kolejnym etapem jest *zatwierdzanie*, podczas którego jest aktualizowana hierarchia DOM. Po tym wprowadzeniu możesz się zapoznać z metodami cyklu życia. Oto one:

- Po wstawieniu komponentu do hierarchii DOM i zatwierdzeniu tej zmiany wywoływana jest metoda komponentu `componentDidMount()`, o ile została zaimplementowana. W tej metodzie można zainicjalizować wszystko to, co wymaga użycia hierarchii DOM. Wszystkie zadania inicjalizacji, które *nie* wymagają hierarchii DOM, należy wykonać w konstruktorze. Większość zadań inicjalizacji nie powinna korzystać z hierarchii DOM. Jednak w tej metodzie można na przykład zmierzyć wysokość wyrenderowanego komponentu, dodać nasłuchiwanie zdarzeń (np. `addEventListener('resize')`) lub pobrać dane z serwera.
- Tuż przed usunięciem komponentu z hierarchii DOM wywoływana jest metoda `component↪WillUnmount()`. W tym miejscu można wykonać wszystkie niezbędne zadania porządkowe. Należy usunąć wszystkie funkcje obsługi zdarzeń i inne elementy, które mogą prowadzić do wycieków pamięci. Po wywołaniu tej funkcji komponent zostanie usunięty na zawsze.
- Przed aktualizacją komponentu (np. po wywołaniu metody `setState()`) można wywołać metodę `getSnapshotBeforeUpdate()`. Ta metoda przyjmuje argumenty w postaci poprzednich właściwości i stanu. Może zwrócić „migawkę”, czyli wartość, którą można przekazać do następnej metody cyklu życia `componentDidUpdate()`.
- Metoda `componentDidUpdate(previousProps, previousState, snapshot)` jest wywoływana po aktualizacji komponentu. Ponieważ na tym etapie obiekty `this.props` i `this.state` mają uaktualnione wartości, otrzymujemy kopie poprzednich wartości. Za pomocą tych informacji można porównać wcześniejszy stan z nowym i, jeśli to konieczne, wykonać więcej żądań sieciowych.
- Kolejną metodą jest `shouldComponentUpdate(newProps, newState)`, w której można dokonać pewnej optymalizacji. Otrzymujesz wartość nowego stanu, którą możesz porównać z obecną i pominąć aktualizowanie komponentu oraz wywołanie metody `render()`.

Z wspomnianych metod najczęściej wykorzystywane są `componentDidMount()` i `componentDidUpdate()`.

Przykład cyklu życia: zaloguj wszystko

Aby lepiej zrozumieć cykl życia komponentu, zdefiniujmy logowanie w komponencie `TextAreaCounter`. Po prostu zaimplementujemy wszystkie metody cyklu życia, aby zalogować w konsoli informację o ich wywołaniu, wraz z wszelkimi argumentami:

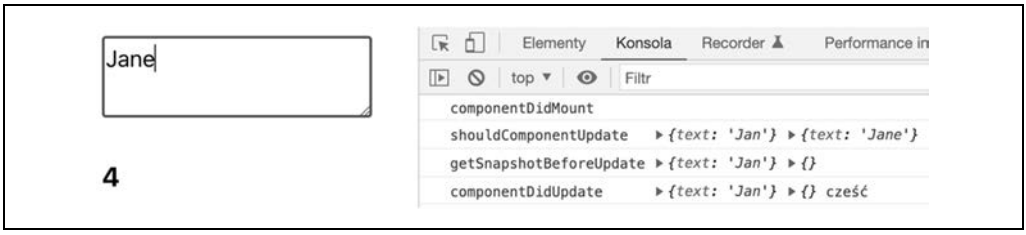
```
class TextAreaCounter extends React.Component {
   // ...

   componentDidMount() {
      console.log('componentDidMount');
   }
   componentWillUnmount() {
      console.log('componentWillUnmount');
   }
   componentDidUpdate(prevProps, prevState, snapshot) {
      console.log('componentDidUpdate     ', prevProps, prevState, snapshot);
   }
   getSnapshotBeforeUpdate(prevProps, prevState) {
      console.log('getSnapshotBeforeUpdate', prevProps, prevState);
      return 'hello';
   }
   shouldComponentUpdate(newProps, newState) {
      console.log('shouldComponentUpdate  ', newProps, newState);
      return true;
   }

   // ...
}
```

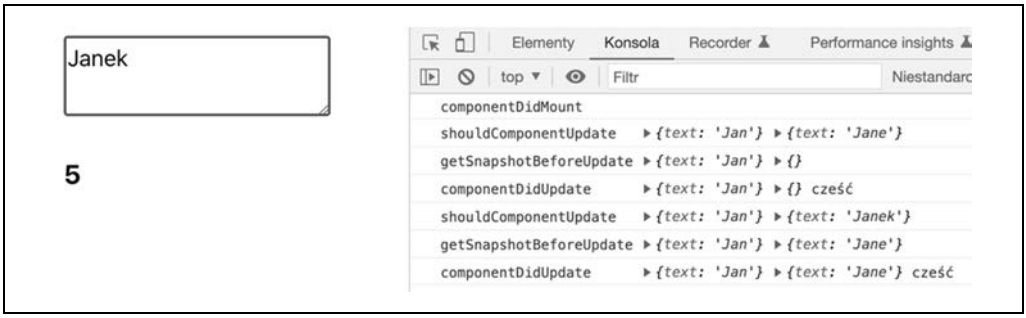
Po wczytaniu strony w konsoli będzie widoczny tylko komunikat `componentDidMount`.

Co się teraz stanie, gdy wpiszesz „e”, aby tekst miał postać „Jane”? (Zobacz rysunek 2.6). Zostanie wywołana metoda `shouldComponentUpdate()` z nowymi właściwościami (o takiej samej wartości jak poprzednio) i z nowym stanem. Ponieważ ta metoda zwróci wartość `true`, React wywoła metodę `getSnapshotBeforeUpdate()`, przekazując do niej dawne właściwości i stan. Na tym etapie możesz wykonać pewne zadanie na ich podstawie oraz na podstawie wcześniejszej hierarchii DOM. Uzyskany wynik możesz przekazać jako migawkę do następnej metody. Można na przykład zmierzyć pewne elementy lub sprawdzić pozycję przewijania i utworzyć na ich podstawie migawkę, aby sprawdzić, czy się zmienią po aktualizacji. Na koniec wywoływana jest metoda `componentDidUpdate()` z wcześniejszymi danymi (nowe są dostępne w obiektach `this.state` i `this.props`) oraz z migawką zdefiniowaną w poprzedniej metodzie.



Rysunek 2.6. Aktualizowanie komponentu

Uaktualnij element `textarea` jeszcze jeden raz, tym razem wpisując „k”. Wynik jest pokazany na rysunku 2.7.

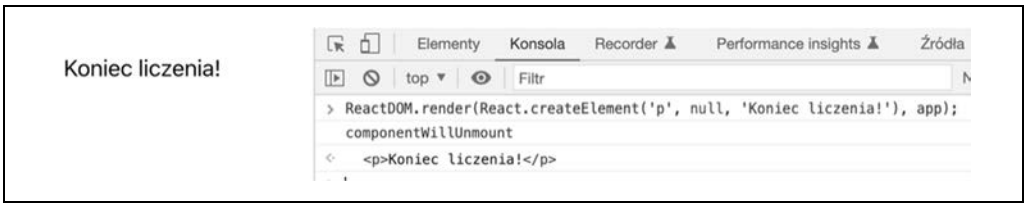


Rysunek 2.7. Jeszcze jedna aktualizacja komponentu

Aby zobaczyć działanie metody `componentWillUnmount()` (korzystając z *pliku 02.14.lifecycle.html* dołączonego do książki), możesz wpisać w konsoli:

```
ReactDOM.render(React.createElement('p', null, 'Koniec liczenia!'), app);
```

W ten sposób zastąpisz cały komponent `textarea` nowym elementem `<p>`. Następnie w konsoli zostanie wyświetlony komunikat `componentWillUnmount` (pokazany na rysunku 2.8).



Rysunek 2.8. Usuwanie komponentu z hierarchii DOM

Paranoiczna ochrona stanu

Załóżmy, że chcemy ograniczyć liczbę znaków w elemencie `textarea`. W tym celu powinniśmy użyć metody obsługi zdarzeń `onTextChanged()`, która jest wywoływana podczas pisania tekstu przez użytkownika. A co się stanie, gdy ktoś (młodszy i bardziej naiwny?) wywoła funkcję `setState()` spoza komponentu (co, jak już wspomniałem, jest błędne)? Czy nadal możesz zagwarantować spójność i poprawność działania komponentu? Oczywiście. Możesz dokonać walidacji w metodzie `componentDidUpdate()`, a jeśli liczba znaków przekracza limit, możesz przywrócić stan do poprzedniej postaci. Oto przykład:

```
componentDidUpdate(prevProps, prevState) {
   if (this.state.text.length > 3) {
      this.setState({
         text: prevState.text || this.props.text,
      });
   }
}
```

Warunek `prevState.text || this.props.text` jest potrzebny podczas pierwszej aktualizacji, gdy nie dysponujemy jeszcze poprzednim stanem.

Być może uznasz to podejście za zbyt paranoiczne, ale jest ono możliwe. Ten sam poziom ochrony można osiągnąć za pomocą metody `shouldComponentUpdate()`:

```
shouldComponentUpdate(_, newState) {
   return newState.text.length > 3 ? false : true;
}
```

Aby przećwiczyć tę koncepcję, zajrzyj do pliku *02.15.paranoid.html* dołączonego do tej książki.

W poprzednim fragmencie kodu element _ używany jako nazwa argumentu funkcji jest konwencją, którą przyszły czytelnik kodu zinterpretuje tak: „Wiem, że w sygnaturze funkcji zdefiniowany jest jeszcze jeden argument, ale nie będę go używać”.

Przykład cyklu życia: użycie komponentu potomnego

Wiesz już, że w razie potrzeby można dowolnie zagnieżdżać komponenty Reacta. Z komponentami `ReactDOM` (w przeciwieństwie do komponentów niestandardowych) miałeś dotychczas do czynienia tylko w metodach `render()`. Przyjrzyjmy się prostemu niestandardowemu komponentowi, którego użyjemy jako komponentu potomnego.

Definicję licznika znaków możesz wyizolować do osobnego komponentu. Zależy nam przecież na strategii „dziel i rządź”.

Najpierw wyizolujemy wyświetlanie komunikatów cyklu życia do osobnej klasy, po której będą dziedziczyć dwa komponenty. W przypadku Reacta dziedziczenie niemal nigdy nie ma uzasadnienia, ponieważ tworząc interfejs, preferuje się *kompozycję*, a w pozostałych przypadkach można użyć zwykłego modułu w JavaScripcie. Jednak warto wiedzieć, jak działa ten mechanizm, a jednocześnie uniknąć kopiowania i wklejania metod logowania.

Oto klasa bazowa:

```
class LifecycleLoggerComponent extends React.Component {
   static getName() {}
   componentDidMount() {
      console.log(this.constructor.getName() + '::componentDidMount');
   }
   componentWillUnmount() {
      console.log(this.constructor.getName() + '::componentWillUnmount');
   }
   componentDidUpdate(prevProps, prevState, snapshot) {
      console.log(this.constructor.getName() + '::componentDidUpdate');
   }
}
```

Nowy komponent `Counter` tylko pokazuje wartość licznika. Nie utrzymuje stanu, lecz jedynie wyświetla wartość właściwości `count`, nadaną przez element nadrzędny.

```
class Counter extends LifecycleLoggerComponent {
   static getName() {
      return 'Counter';
```

```
    }
    render() {
      return <h3>{this.props.count}</h3>;
    }
}
Counter.defaultProps = {
    count: 0,
};
```

W komponencie `textarea` zdefiniowana jest metoda statyczna `getName()`:

```
class TextAreaCounter extends LifecycleLoggerComponent {
    static getName() {
      return 'TextAreaCounter';
    }
    // ....
}
```

Na koniec metoda `render()` komponentu `textarea` używa warunkowo komponentu `<Counter/>`; jeśli licznik ma wartość 0, na ekranie nie zostanie nic wyświetlone:

```
render() {
    const text = 'text' in this.state ? this.state.text : this.props.text;
      return (
        <div>
          <textarea
            value={text}
            onChange={this.onTextChange}
          />
         {text.length > 0
          ? <Counter count={text.length} />
          : null
         }
      </div>
    );
}
```

Zauważ, że w wyrażeniu warunkowym użyłem składni JSX. Wyrażenie znajduje się w nawiasach {} i warunkowo renderuje komponent `<Counter/>` lub nic (`null`). Do celów demonstracyjnych pokażę też inny sposób przeniesienia warunku poza instrukcję `return`. Doskonale sprawdzi się przypisanie wyniku wyrażenia JSX do zmiennej:

```
render() {
    const text = 'text' in this.state
      ? this.state.text
      : this.props.text;
    let counter = null;
    if (text.length > 0) {
      counter = <Counter count={text.length} />;
    }
    return (
      <div>
        <textarea
          value={text}
          onChange={this.onTextChange}
        />
```

```
                {counter}
            </div>
        );
    }
```

Teraz możesz już zaobserwować metody cyklu życia logowane dla obydwu komponentów. Otwórz w przeglądarce plik *02.16.child.html* dołączony do książki, aby sprawdzić, co się stanie, gdy wczytasz stronę, a następnie zmienisz zawartość obszaru tekstowego.

Podczas początkowego ładowania komponent potomny jest montowany i aktualizowany przed komponentem macierzystym. W konsoli zostaną wyświetlone następujące komunikaty:

```
Counter::componentDidMount
TextAreaCounter::componentDidMount
```

Po usunięciu dwóch znaków zobaczysz, że najpierw aktualizowany jest element potomny, a następnie macierzysty:

```
Counter::componentDidUpdate
TextAreaCounter::componentDidUpdate
Counter::componentDidUpdate
TextAreaCounter::componentDidUpdate
```

Po usunięciu ostatniego znaku element potomny zostanie zupełnie usunięty z hierarchii DOM:

```
Counter::componentWillUnmount
TextAreaCounter::componentDidUpdate
```

Na koniec, po wpisaniu znaku, element licznika zostanie znów dodany do hierarchii DOM:

```
Counter::componentDidMount
TextAreaCounter::componentDidUpdate
```

Zysk wydajnościowy: zapobieganie aktualizacjom komponentów

Znasz już metodę `shouldComponentUpdate()` i mogłeś zaobserwować jej działanie. Przydaje się ona szczególnie podczas tworzenia elementów aplikacji krytycznych pod względem wydajnościowym. Jest ona wywoływana przed metodą `componentWillUpdate()` i daje nam szansę na anulowanie aktualizacji, o ile uważamy, że nie jest ona potrzebna.

Istnieje klasa komponentów, które w metodzie `render()` wykorzystują tylko obiekty `this.props` i `this.state` i nie wywołują żadnej innej funkcji. Te komponenty są zwane „czystymi" komponentami. Mogą implementować metodę `shouldComponentUpdate()`, w której porównywany jest stan oraz właściwości przed i po. W przypadku braku zmian metoda ta może zwracać wartość `false`, co pozwala zachować nieco mocy obliczeniowych. Ponadto można zdefiniować czyste statyczne komponenty, które nie używają ani obiektu `props`, ani `state`. Mogą od razu zwracać `false` we wspomnianej metodzie.

React może ułatwiać użycie typowych (i generycznych) metod sprawdzania wszystkich właściwości i stanu w metodzie `shouldComponentUpdate()`. Zamiast powtarzać wykonywanie tego zadania komponenty mogą dziedziczyć po klasie `React.PureComponent`, a nie po `React.Component`. Dzięki

temu nie musisz implementować metody `shouldComponentUpdate()` — React Cię w tym wyręczy. Wykorzystajmy to i poprawmy poprzedni przykład.

Ponieważ obydwa komponenty dziedziczą po klasie logującej, wystarczy, że zdefiniujesz ją następująco:

```
class LifecycleLoggerComponent extends React.PureComponent {
   // ... brak innych zmian
}
```

Teraz obydwa komponenty są *czyste*. Zdefiniujmy też wyświetlanie komunikatu w konsoli w metodzie `render()`:

```
render() {
   console.log(this.constructor.getName() + '::render');
   // ... brak innych zmian
}
```

Teraz po wczytaniu strony (plik *02.17.pure.html* dołączony do książki) w konsoli zostaną wyświetlone następujące komunikaty:

```
TextAreaCounter::render
Counter::render
Counter::componentDidMount
TextAreaCounter::componentDidMount
```

Po zmianie tekstu „Jan” na „Jann” uzyskamy oczekiwane komunikaty informujące o renderowaniu i aktualizacji:

```
TextAreaCounter::render
Counter::render
Counter::componentDidUpdate
TextAreaCounter::componentDidUpdate
```

Gdy teraz *wkleisz* tekst „LOLz” zamiast „Jann” (lub dowolny ciąg składający się z czterech znaków), ujrzysz następujący efekt:

```
TextAreaCounter::render
TextAreaCounter::componentDidUpdate
```

Jak widać nie ma potrzeby ponownego renderowania komponentu `<Counter>`, ponieważ jego właściwości się nie zmieniły. Nowy ciąg tekstowy ma tyle samo znaków co poprzedni.

Co się stało z komponentami funkcyjnymi?

Być może zauważyłeś, że w tym rozdziale przestaliśmy się zajmować komponentami funkcyjnymi, gdy zacząłem omawiać obiekt `this.state`. Powrócimy do nich w dalszej części książki, podczas omawiania koncepcji *hooków*. Ponieważ w funkcjach nie mamy do dyspozycji obiektu `this`, musi istnieć inny sposób zarządzania stanem komponentu. Jednak gdy tylko zrozumiesz koncepcje stanu i właściwości, zauważysz, że różnice w korzystaniu z nich w komponentach funkcyjnych dotyczą tylko składni.

Mimo że czas spędzony z elementem `textarea` był bardzo ciekawy, warto się zająć czymś bardziej atrakcyjnym. W następnym rozdziale poznasz korzyści wynikające z używania Reacta. Skupimy się na swoich *danych* i wykorzystamy Reacta do aktualizacji interfejsu użytkownika po ich zmianie.

React: naucz się raz, używaj zawsze!

React jest biblioteką służącą do tworzenia interfejsów użytkownika. Ułatwia pisanie aplikacji internetowych, natywnych aplikacji dla iOS i Androida, oprogramowania dla telewizorów czy też natywnych aplikacji dla komputerów stacjonarnych. Dzięki tej bibliotece można szybko zbudować interfejs, który automatycznie będzie reagował na zmiany stanu aplikacji. Idea polega na użyciu małych, zarządzalnych komponentów do budowy nawet dużych i bardzo złożonych aplikacji. Aby zacząć pracę z Reactem, wystarczy znajomość składni JavaScriptu — i lektura tej książki.

Z tym przewodnikiem nauczysz się stosowania Reacta w praktyce. Dowiesz się, w jaki sposób zbudować jednostronicową, złożoną aplikację internetową, i zdobędziesz wiedzę umożliwiającą używanie tej biblioteki w codziennej pracy. Pokazano tu, jak rozpocząć projekt i rozwijać rzeczywistą aplikację. Zaprezentowano także technologie, które znakomicie uzupełniają możliwości Reacta: JSX i narzędzie create-react-app. Omówiono również zagadnienia dotyczące komponentów funkcyjnych i klasowych i szczegółowo przedstawiono proces budowy aplikacji z tych komponentów. Szybko się przekonasz, jak duży potencjał tkwi w bibliotece React i jak bardzo ułatwia ona tworzenie łatwych w utrzymaniu, wielkoskalowych, atrakcyjnych aplikacji!

W książce między innymi:

- przygotowanie Reacta do pracy
- tworzenie komponentów Reacta i łączenie ich z komponentami DOM
- składnia JSX i hooki
- przepływ danych w aplikacji
- tworzenie aplikacji zapisujących dane po stronie klienta

Stoyan Stefanov jest przedsiębiorcą, inżynierem i autorem książek technicznych. Pracował w takich firmach jak Facebook i Yahoo! Opracował kilka przydatnych narzędzi programistycznych, w tym smush.it i YSlow 2.0. Często występuje jako prelegent na prestiżowych konferencjach, takich jak Velocity, JSConf, Fronteers i inne.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-8322-039-0
9 788383 220390
Cena: 59,00 zł