14. ReactJS

Wyzwania:

- Dowiesz się, czym jest React, i dlaczego warto z niego korzystać.
- Stworzysz swoje pierwsze komponenty.
- Dowiesz się, czym jest JSX.

Zadanie: Poprawa istniejącej aplikacji

14.1. Czym jest React i dlaczego warto go znać

Zaczniemy od krótkiej historii internetu, przy czym skupimy się na tym, jak mniej-więcej wyglądało życie front-end developera.

Pierwsze strony internetowe budowano przy użyciu klasycznych technologii — HTML i CSS. Nie było w tych stronach żadnej dynamiki i w zasadzie serwer służył tylko do wysyłania odpowiednich plików do klienta (przeglądarki), a ta miała za zadania jedynie odczytać te pliki i odpowiednio je pokazać w swoim oknie.

Z biegiem czasu pojawiły się strony generowane dynamicznie. Zaczęto przeprowadzać walidację formularzy po stronie klienta — pojawienie się technologii AJAX do dziś umożliwia dynamiczne doładowywanie treści po stronie klienta bez konieczności przeładowania strony. Przeglądarek internetowych pojawiało się coraz więcej, a brak jednolitego standardu ich tworzenia zrodził problemy z kompatybilnością. W pewnym momencie powstała również bardzo popularna biblioteka jQuery, która ujednoliciła interfejs do manipulowania drzewem DOM, obsługę AJAXa, walidację i wiele innych rzeczy, z którymi musiał się zmagać programista po stronie frontu.

jQuery było (i jest) bardzo dobrą biblioteką do momentu, w którym zrodził się nowy termin: "aplikacja internetowa". Wraz z nim pojawił się szereg nowych problemów, którym jQuery nie mogło już sprostać. Programiści musieli wymyślić nowe rozwiązania, które umożliwiałyby tworzenie tych aplikacji internetowych.

Powstawało wiele rozwiązań, wśród których był między innymi Angular — framework, który jest kompletnym narzędziem do tworzenia aplikacji internetowych. Angular rozwiązuje wiele problemów, przy czym narzuca pewien styl pisania aplikacji.

Ze względu na szalony rozwój technologii front-endowych, duże firmy postanowiły wypuścić na światło dzienne również swoje zabawki. Jedną z tych firm był między innymi Facebook: jego bibliotekę o nazwie React poznamy w tym module.

React jest biblioteką, która rozwiązuje problem prezentowania danych. Dzięki Reactowi, aplikacje internetowe z dużą ilością danych (takie jak Facebook), ulegające ciągłym zmianom, mogą być poprawnie wyświetlane w przeglądarce. Nie trzeba się martwić o "ręczną" aktualizację — React wszystko robi za nas.

Dlaczego jeszcze warto znać Reacta?

React ma ogromną rzeszę zwolenników. Jego społeczność jest ciągle aktywna, a on sam jest obecnie narzędziem "na czasie". Sytuacja w każdej chwili może się oczywiście zmienić, co pokazuje historia front-endu, która jest często tematem żartów. Narzędzi z końcówką -JS jest cała masa i nie sposób wiedzieć wszystkiego. Pamiętaj jednak, że nie jesteś w tym sam, a znajomość czystego JavaScriptu bardzo pomoże Ci odnaleźć się w tym oceanie narzędzi.

Wracając do pytania: dlaczego React? Mimo że sytuacja "na froncie" ciągle ulega zmianie, w przypadku Reacta nic nie wskazuje na to, by miał stracić na znaczeniu. Zapewnił on sobie silną pozycję w sercach developerów, a ostatnie ankiety dotyczące JavaScriptu tylko to potwierdzają.

Ponadto jest cała masa ogłoszeń o pracę, które wymagają od nas znajomości Reacta. Jest to więc zdecydowanie idealny moment na naukę tego narzędzia!

Jeśli masz wątpliwości do powyższego materiału, to - zanim zatwierdzisz - zapytaj na czacie :)

✔Zapoznałe(a)m się!

14.2. Podstawy działania Reacta

Być może wiesz już, czym jest React, i masz za sobą próby jego użycia. No właśnie — próby. Wielu początkujących przy pierwszym kontakcie z Reactem zderza się z nim jak ze ścianą. Masa tutoriali, które można znaleźć w internecie, opisuje nie tylko samego, czystego Reacta, ale zajmuje się również całym procesem tworzenia środowiska developerskiego, nie wnosząc tak naprawdę niczego na temat samej biblioteki. Często początkujący wpadają w pętlę — żeby używać Reacta potrzebują narzędzi, a żeby wykorzystać narzędzia, potrzebują Reacta. ciężko jest wyjść z takiego impasu, ale my poradzimy sobie z nim w najprostszy możliwy sposób — po prostu zaczniemy od początku :)

Pierwsze kroki

Przejdźmy od razu do praktycznych rzeczy i stwórzmy pierwszy element przy pomocy Reacta. Zacznijmy od stworzenia plików *index.html* i *script.js*.

Plik *index.html* powinien wyglądać następująco:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Apka Reactowa!</title>
  </head>
  <body>
    <div id="app"></div>

    <script src="https://unpkg.com/react@15.3.2/dist/react.js"></scri
    <script src="https://unpkg.com/react-dom@15.3.2/dist/react-dom.js

    <script type="text/javascript" src="script.js"></script>
  </body>
</html>
```

Wstęp do Reacta

Do działania Reacta wystarczy dodać dwa skrypty:

```
<script src="https://unpkg.com/react@15.3.2/dist/react.js"></script>
<script src="https://unpkg.com/react-dom@15.3.2/dist/react-dom.js"></
```

Pierwszy z nich ładuje samą bibliotekę, natomiast drugi jest potrzebny do wyświetlenia elementów stworzonych za pomocą Reacta w oknie przeglądarki. Zauważ, że w kodzie HTML znajduje się również pusty div (`<div id="app"></div>`) — to w nim wyświetlona zostanie cała Reactowa aplikacja.

Plik *script.js* wypełnijmy następującą zawartością:

```
var element = React.createElement('div', {}, 'Hello world!');
ReactDOM.render(element, document.getElementById('app'));
```

Za pomocą metody createElement tworzymy obiekt, który w Reakcie nazywa się ReactElement. Metoda ta przyjmuje trzy parametry:

- nazwę klasy, na podstawie której ma zostać stworzony element,
- propsy, czyli właściwości danego elementu (wejścia),

- dzieci elementu, czyli to, co ma się znaleźć wewnątrz. Może to być zarówno tekst, jak i kolejny ReactElement.

Zatrzymajmy się jeszcze na chwilę, aby omówić nieco dokładniej propsy elementu. Propsy są dla Reacta tak samo ważne, jak atrybuty dla elementów HTML. Możesz traktować je jako rozszerzenie atrybutów HTML, które pozwala nam na przekazywanie danych między komponentami w prosty i uporządkowany sposób.

W powyższym przykładzie nasz element tworzymy na podstawie klasy div. Uwaga: nie należy mylić tego z elementem `<div>` znanym z HTML. Ten element, jak wiele innych, jest stworzony na potrzeby Reacta, ale o tym jeszcze wspomnimy.

Kiedy mamy już gotowy element, możemy wyrenderować go w odpowiednim miejscu w drzewie DOM. Tym właśnie zajmuje się metoda render, która jako parametry przyjmuje:

- ReactElement (np. ten który stworzyliśmy linijkę wyżej),
- węzeł drzewa DOM, do którego element ma się "wpiąć".

Do ReactDOM jeszcze sobie wrócimy. Teraz zobaczmy wynik działania tej operacji:

Jeśli wszystko zostało wykonane zgodnie z instrukcją, to po otworzeniu pliku *index.html* w naszej przeglądarce, powinniśmy zobaczyć nieśmiertelny napis "Hello world!" :)

Teraz stworzymy coś bardziej skomplikowanego, np. listę naszych ulubionych filmów:

```
var element =
  React.createElement('div', {},
    React.createElement('h1', {}, 'Lista filmów'),
    React.createElement('ul', {},
      React.createElement('li', {},
        React.createElement('h2', {}, 'Harry Potter'),
        React.createElement('p', {}, 'Film o czarodzieju')
      ),
      React.createElement('li', {},
        React.createElement('h2', {}, 'Król Lew'),
        React.createElement('p', {}, 'Film opowiadający historię król
      )
    )
  );
```

UWAGA: staraj się przepisać powyższy kod. Wiemy, że dużo się w nim powtarza i możesz myśleć, że jest to całkowita strata czasu, ale uwierz — dzięki temu więcej i szybciej się nauczysz! Jeśli wszystko wykonaliśmy prawidłowo, w przeglądarce powinniśmy zobaczyć podobny widok:

Refaktoryzacja utworzonego kodu

Oczywiście tych filmów może być cała masa i już możesz wyczuć pewien problem. Pisanie `React.createElement` i tworzenie kolejnych wpisów może być bardzo uciążliwe. Otóż powyższy kod to zwykły JavaScript, tak więc mając dane na temat filmów w formie tablicy możemy użyć odpowiedniej metody, aby zmapować listę filmów na listę Reactowych elementów, które następnie wyświetlimy użytkownikowi. Spróbujmy więc zmodyfikować nieco powyższy kod — zaczniemy od przeniesienia danych na temat naszych ulubionych filmów do osobnej tablicy.

```
var movies = [
  {
    title: 'Harry Potter',
    desc: 'film o czarodzieju'
  },
  {
    title: 'Król Lew',
    desc: 'Film o królu sawanny'
  }
];
```

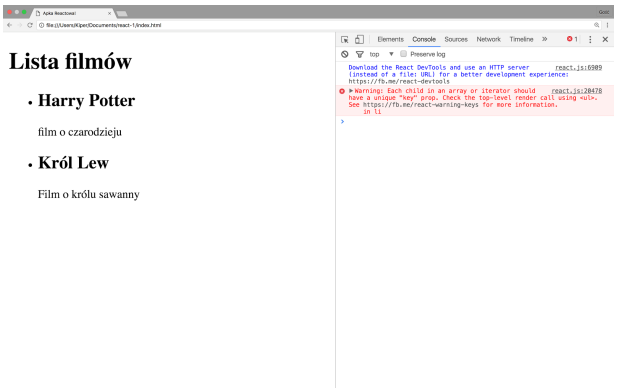
Kolejnym krokiem będzie zmapowanie powyższej listy na odpowiadające im Reactowe elementy. Do tego celu przyda się nam metoda `.map()`, która przyjmuje jako parametr funkcję, przez którą przechodzi każdy z elementów tablicy (czyli każdy film), który następnie jest mapowany do postaci ReactElementu korzystającego z informacji o filmie (`movie.title` i `movie.desc`):

```
var moviesElements = movies.map(function(movie) {
  return React.createElement('li', {},
      React.createElement('h2', {}, movie.title),
      React.createElement('p', {}, movie.desc)
    );
});
```

Ostatnim elementem będzie odpowiednie umieszczenie powyższych elementów w liście:

```
var element =
  React.createElement('div', {},
    React.createElement('h1', {}, 'Lista filmów'),
    React.createElement('ul', {}, moviesElements)
  );
```

Strona nie powinna ulec zmianie, ale kiedy zajrzymy w konsolę, powinniśmy zobaczyć pewien błąd:



Okazuje się, że React wykrył, że używamy tego samego elementu wielokrotnie i odsyła nas do swojej dokumentacji, żebyśmy rozwiązali problem. Sytuacja z dynamicznymi elementami jest dla Reacta o tyle skomplikowana, że bez wiedzy o tym, który element uległ zmianie, odświeżone zostaną wszystkie elementy, a nie tylko te, które powinny. Rodzi to oczywiście pewne problemy z optymalizacją naszej aplikacji, np. przy większej liczbie elementów do odświeżenia (weźmy za przykład około 1000) moglibyśmy odnotować spadki wydajnościowe, podczas gdy w rzeczywistości edytowaliśmy np. tylko jeden opis filmu. Stąd też należy używać specjalnego parametru `key`, aby pozwolić Reactowi zidentyfikować elementy z filmami ze względu na optymalizację :)

Dodanie parametru key

W celu dodania unikalności do naszej kolekcji filmów, musimy nadać im specjalne identyfikatory, czyli zmienić nieco tablicę z obiektami filmów:

```
var movies = [
  {
    id: 1,
    title: 'Harry Potter',
    desc: 'film o czarodzieju'
  },
  {
    id: 2,
    title: 'Król Lew',
    desc: 'Film o królu sawanny'
  }
];
```

Drugim miejscem zmiany jest dopisanie parametru key do odpowiedniego miejsca w listę filmów:

```
var moviesElements = movies.map(function(movie) {
  return React.createElement('li', {key: movie.id},
      React.createElement('h2', {}, movie.title),
      React.createElement('p', {}, movie.desc)
    );
});
```

Po zastosowaniu tych zmian, problem powinien zniknąć :)

Zadanie: Poprawa istniejącej aplikacji

W ramach ćwiczenia, spróbuj dopisać do swojej aplikacji kilka nowych obiektów z informacjami o filmach, a także dodaj do każdego z nich obrazek :)

1. Spróbuj dopisać kilka obiektów z filmami.
2. Dodaj do elementu zdjęcie z plakatem filmu.

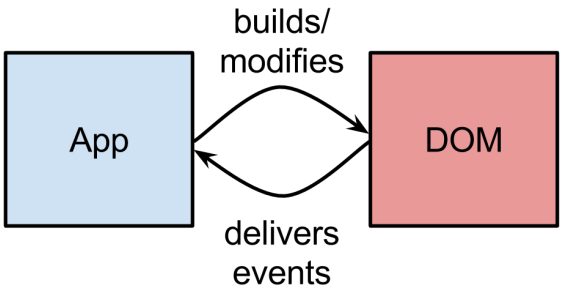
Podgląd zadania

https://github.com/0na/

Wyślij link ✔

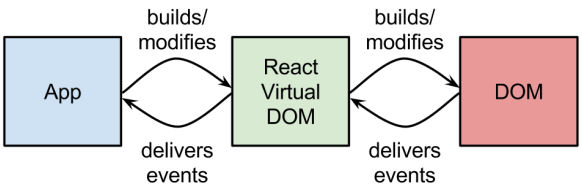
14.3. Zasada działania ReactDOM.render i słynna optymalizacja drzewa DOM

W tradycyjnych aplikacjach internetowych zmiana widoku na podstawie danych zachodzi bezpośrednio na drzewie DOM. Taka sytuacja ma miejsce np. w jQuery i wygląda to mniej więcej tak:



DOM jest zaznaczony na czerwono z powodu swojej kosztowności.

React rozwiązuje sprawę nieco inaczej dzięki popularnemu *VirtualDOM*. Cała zabawa polega na tym, że React tworzy swoją, wirtualną warstwę DOM i odpowiada za aktualizację prawdziwego DOM. Specjalne algorytmy optymalizacyjne, o których możesz poczytać nieco tutaj, sprawiają, że zmianie ulegają tylko te elementy, które zmianie faktycznie muszą ulec. Każdorazowa zmiana stanu aplikacji, o którym opowiemy sobie nieco później, odpala mechanizm porównywania (ang. *diffing*) i jeśli poprzedni stan aplikacji różni się od obecnego, to React odpowiednio modyfikuje prawdziwe drzewo DOM.



Oczywiście moglibyśmy robić to sami, ale jest to jeden z trudniejszych elementów pisania aplikacji internetowych i na pewno sprawiłby nam dużo problemów. Programiści z Facebooka wykonali kawał dobrej roboty tworząc Reacta, więc czemu nie skorzystać z ich rozwiązania? :)

W poprzednim submodule skorzystaliśmy z atrybutu key do określenia kolejnych, niezależnych elementów listy. Właśnie dzięki niemu zachodzi optymalizacja szybkości działania algorytmu diffującego i React nie musi przy każdej operacji `.render()` odświeżać całej listy. Wystarczy, że sprawdzi brakujące klucze i odpowiednio dopasuje poprawioną strukturę do istniejącego drzewa DOM.

Jeśli masz wątpliwości do powyższego materiału, to - zanim zatwierdzisz - zapytaj na czacie :)

✔Zapoznałe(a)m się!

14.4. Ćwiczenie: Pierwszy komponent - galeria

Do tej pory tworzyliśmy elementy na podstawie klas udostępnianych przez Reacta, takich jak div, h2, p, ul, li. Są to klasy, które mają nazwy znanych elementów HTML, przy czym należy pamiętać, że nie jest to stricte ani HTML, ani DOM. Przy tworzeniu elementów React idzie o krok dalej i pozwala na definiowanie naszych własnych komponentów, czyli klas/szablonów, za pomocą których w prostszy sposób stworzymy swoje własne elementy. Nie przedłużając, spróbujmy napisać nasz pierwszy komponent, którego zadaniem będzie wyświetlenie zdjęcia wraz z podpisem:

Zaczniemy od stworzenia klasy naszego komponentu. Możemy to zrobić dzięki metodzie `React.createClass`:

```
var GalleryItem = React.createClass({
    render: function() {
    return React.createElement('h2', {}, 'Pierwszy komponent');
}
});
```

Metoda ta przyjmuje na wejściu jeden parametr — obiekt, który określa zachowanie komponentu. Obiekt ten musi zawierać przynajmniej jedną metodę, która wykona się przy próbie stworzenia elementu — jest nią `render`, która, jak sama nazwa wskazuje, zajmuje się jego renderowaniem. Pamiętaj, że metoda `render` musi zwracać obiekt ReactElement!

Kolejnym atrybutem, który można umieścić w obiekcie określającym zachowanie komponentu, jest `propTypes`. Dzięki niemu możemy określić typ danych, które dostarczane są w postaci właściwości (ang. *props*) do ReactElement w drugim parametrze metody createElement w następujący sposób: `React.createElement('nazwa klasy elementu', {atrybuty}, elementy)`. Typowanie propsów komponentu wygląda następująco:

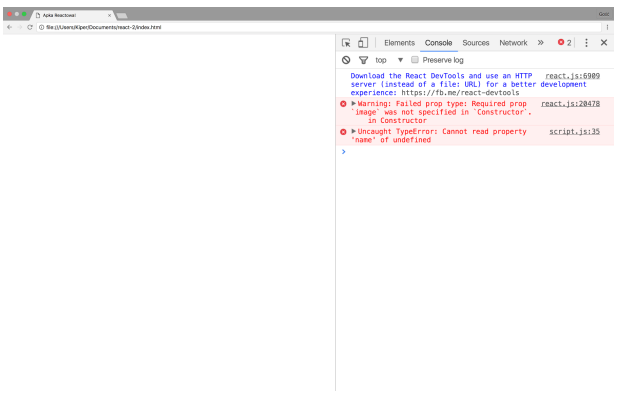
```
var GalleryItem = React.createClass({
  propTypes: {
    image: React.PropTypes.object.isRequired,
  },

  render: function() {
    return (
      React.createElement('div', {},
        React.createElement('h2', {}, this.props.image.name),
        React.createElement('img', {src: this.props.image.src})
      )
    )
  },
});
```

`propTypes` to obiekt, który określa, jakich właściwości możemy się spodziewać, i jakiego będą typu. Powyższy przykład zakłada, że będzie to jedna właściwość (`image`) i musi być to obiekt. Jest to swego rodzaju walidacja przeprowadzana na właściwościach. Jeśli spróbujemy więc przekazać elementowi zły typ danych w propsach, w konsoli zobaczymy błąd. Jeśli chcesz zobaczyć, w jaki sposób dokładniej walidować komponenty, zajrzyj na tę stronę.

Kolejną rzeczą, na którą warto zwrócić uwagę, jest konwencja zapisu zmiennej z nowo utworzoną klasą (`GalleryItem`). Przyjmuje się, że do zapisu klas zawsze stosujemy format `UpperCase` — dzięki temu prostemu zabiegowi jesteśmy w stanie odróżnić klasę od jej instancji.

Teraz zobaczymy, jak utworzyć element na podstawie klasy `GalleryItem`, i czym skutkuje brak przekazania odpowiedniego parametru `image`.

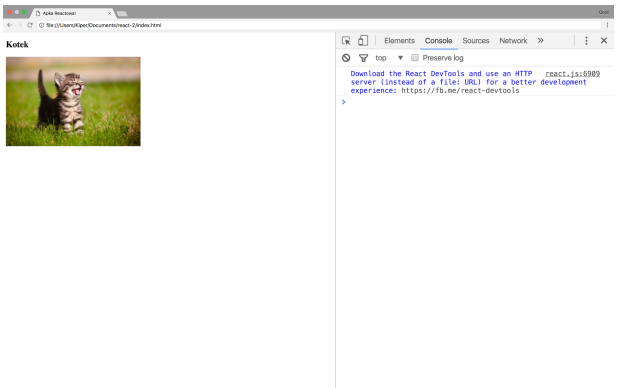
```
var element = React.createElement(GalleryItem);
ReactDOM.render(element, document.getElementById('app'));
```



Jak widzisz, nasz komponent nie został poprawnie wyświetlony, a w konsoli pojawił się komunikat o błędzie. Poprawmy to, dodając odpowiedni obiekt i przekazując go do komponentu przez propsy:

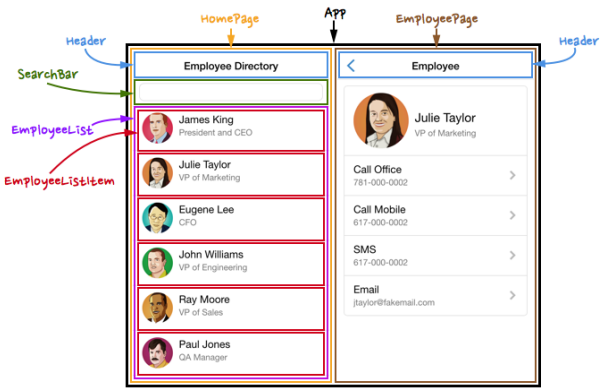
```
var image = {
  name: 'Kotek',
  src: 'http://imgur.com/n8OYCzR.png'
};

var element = React.createElement(GalleryItem, {image: image});
```

Teraz nasz komponent powinien wyświetlić się prawidłowo:



Zauważ, że komponent działa trochę na zasadzie czarnej skrzynki, której wejściem są propsy, a wyjściem jest wyrenderowany element.

Pamiętaj również, żeby tworząc swoje aplikacje rozkładać je na jak najmniejsze części pierwsze. Możesz nawet rozrysować sobie swój projekt i wydzielić z niego najmniejsze komponenty, tak jak np. tutaj:



Na razie to wszystko, co musisz wiedzieć na temat komponentów.

Zadanie: Wydzielamy komponenty!

Teraz Twoim zadaniem będzie podzielenie kodu, który napisaliśmy wcześniej, na jak najmniejsze komponenty.

1. Najpierw stwórz klasę `Movie` za pomocą `var Movie = React.createClass({...});`. Następnie dodaj do niej metodę `.render()` i przenieś do niej odpowiedni fragment kodu obsługujący wyświetlanie informacji na temat filmu.
2. Po napisaniu metody `.render()` dokonaj walidacji propsów komponentu, dodając parametr propTypes. Następnie stwórz instancję (ReactElement) na podstawie klasy `Movie` za pomocą `React.createElement(Movie, {key: movie_id, ...więcej_propsow},);`
3. Pamiętaj, żeby wyciągnąć klucz z elementu `li` na wysokość deklaracji elementu Movie (tak jak powyżej).
4. Kroki od od 1 do 3 należy wykonać również dla klas: `MovieTitle`, `MovieDescription`, `MoviesList` :)

Podgląd zadania

https://github.com/0na/

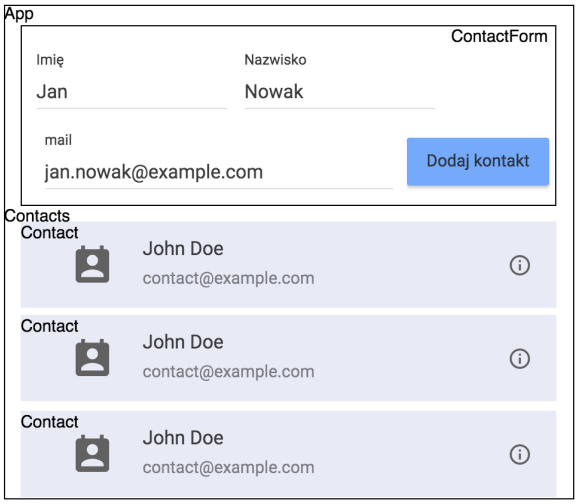
Wyślij link

14.5. Ćwiczenie: Komponenty “kontakt” i “formularz”

W tym submodule przećwiczymy tworzenie swoich własnych elementów, budując prostą aplikację do wyświetlania listy kontaktów i dodawania do niej nowych osób za pomocą formularza.

Ustalenie struktury aplikacji

Zacznijmy od rozrysowania naszej aplikacji i wyjaśnienia, z jakich elementów będzie zbudowana.



Końcowa aplikacja będzie nieco się różnić od powyższej makiety, ale chodzi bardziej o rozrysowanie i wydzielenie podstawowych komponentów naszej aplikacji. Jak widzisz, będą to cztery rodzaje:

- *Contact* — czyli jeden element listy kontaktów
- *Contacts* — lista kontaktów
- *ContactForm* — komponent, za pomocą którego będziemy dodawali kolejne kontakty
- *App* — komponent, który będzie "kontenerem" na wszystkie inne komponenty i będzie podłączony bezpośrednio do drzewa DOM

RADA: w celu stworzenia podobnych makiet (ang. *mockup*) możesz skorzystać np z: Moqups, Balsamiq czy też UXPin (znanego na całym świecie polskiego, profesjonalnego narzędzia do tworzenia makiet).

Ok, wiemy już z jakich komponentów będzie składać się nasza aplikacja. Stwórzmy teraz osobne pliki dla każdego z nich, tak aby struktura katalogów wyglądała następująco:

Jak widzisz, komponenty będą znajdować się w specjalnie wydzielonym katalogu `components`.

Napisanie podstawowego kodu HTML

Kolejnym krokiem będzie napisanie prostego pliku *index.html* z następującą treścią:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Apka Reactowa!</title>
    <link rel="stylesheet" type="text/css" href="styles.css">
  </head>
  <body>
    <div id="app"></div>

    <script src="https://unpkg.com/react@15.3.2/dist/react.js"></scri
    <script src="https://unpkg.com/react-dom@15.3.2/dist/react-dom.js

    <script type="text/javascript" src="components/ContactForm.js"></
    <script type="text/javascript" src="components/Contact.js"></scri
    <script type="text/javascript" src="components/Contacts.js"></scr
    <script type="text/javascript" src="components/App.js"></script>
    <script type="text/javascript" src="script.js"></script>
  </body>
</html>
```

Nie różni się on prawie niczym w porównaniu do tych napisanych w poprzednich zadaniach. Należy jedynie dodać każdy skrypt z każdym komponentem i dodatkowy plik ze stylami, które przygotujesz później samodzielnie.

Kod JavaScript aplikacji

Następnie, w pliku *script.js* umieśćmy prosty skrypt przypinający komponent `App` do drzewa DOM:

```
var app = React.createElement(App);
ReactDOM.render(app, document.getElementById('app'));
```

Wprowadzenie do atrybutów Reacta

Skoro korzystamy w tym miejscu z komponentu App, to zmodyfikujmy jego logikę tak, aby można było na jej podstawie tworzyć nowe `ReactElements`. Plik *App.js* powinien więc wyglądać następująco:

```
var App = React.createClass({
  render: function() {
    return (
      React.createElement('div', {className: 'app'},
        React.createElement(ContactForm, {contact: contactForm}),
        React.createElement(Contacts, {items: contacts}, {})
      )
    );
  }
});
```

W tym pliku wykorzystujemy dwa inne komponenty: `ContactForm` i `Contacts`. Do każdego z nich przekazujemy odpowiednie obiekty przez ich propsy (do komponentu `ContactForm` obiekt `contactForm`, a do komponentu `Contacts` obiekt `contacts`). Zanim zobaczymy, jak powinny wyglądać te obiekty, przyjrzyjmy się jeszcze dokładniej pierwszemu elementowi `div`.

Jak widzisz, nasz komponent przyjmuje dziwny parametr o nazwie `className`. Jest to odpowiednik class na elemencie HTML (np. `<div class="app"></div>`).

Dlaczego więc nie wpisaliśmy w tym miejscu samego słówka class? Cóż — powód jest bardzo prosty. Nowa implementacja JavaScriptu (ES6) posiada w sobie słowo kluczowe class, które jest zarezerwowane i najzwyczajniej nie można go używać przy tworzeniu `ReactElement`. O class w JavaScripcie dowiesz się jeszcze w module opisującym ES6. To samo tyczy się właściwości htmlFor, której nie nazwano po prostu `for`, gdyż w JS jest to słowo kluczowe dla pętli `for`.

Przygotowanie i wyświetlenie danych aplikacji

Następnym krokiem do ukończenia naszej aplikacji będzie stworzenie odpowiednich danych, które przekazujemy do środka komponentów. Będą to: lista z kontaktami `contacts` i obiekt z kontaktem (`contactForm`), który jest połączony z formularzem:

```
var contacts = [
  {
    id: 1,
    firstName: 'Jan',
    lastName: 'Nowak',
    email: 'jan.nowak@example.com',
  },
  {
    id: 2,
    firstName: 'Adam',
    lastName: 'Kowalski',
    email: 'adam.kowalski@example.com',
  },
  {
    id: 3,
    firstName: 'Zbigniew',
    lastName: 'Koziol',
    email: 'zbigniew.koziol@example.com',
  }
];

var contactForm = {
  firstName: '',
  lastName: '',
  email: ''
};

var App = React.createClass({
  render: function() {
    return (
      React.createElement('div', {className: 'app'},
        React.createElement(ContactForm, {contact: contactForm}),
        React.createElement(Contacts, {items: contacts}, {})
      )
    );
  }
});
```

Komponent formularza

Stwórzmy teraz komponent ContactForm. Na razie będzie to tylko komponent, który będzie wyświetlał widok formularza bez powiązanej logiki, ale nie martw się — jeszc do niej wrócimy:

```
var ContactForm = React.createClass({
  propTypes: {
    contact: React.PropTypes.object.isRequired
  },

  render: function() {
    return (
      React.createElement('form', {className: 'contactForm'},
        React.createElement('input', {
          type: 'text',
          placeholder: 'Imię',
          value: this.props.contact.firstName,
        }),
        React.createElement('input', {
          type: 'text',
          placeholder: 'Nazwisko',
          value: this.props.contact.lastName,
        }),
        React.createElement('input', {
          type: 'email',
          placeholder: 'Email',
          value: this.props.contact.email,
        }),
        React.createElement('button', {type: 'submit'}, "Dodaj kontak
      )
    )
  },
})
```

Jak widzisz, stworzyliśmy prosty formularz, który składa się z trzech inputów i jednego przycisku. Warto zwrócić uwagę na właściwości inputów. Zauważ, że zachowują one podobieństwo 1 do 1 z atrybutami elementu `<input>` (`type`, `placeholder` i `value` są atrybutami tego elementu).

Pamiętaj, że każdy `ReactElement` jest instancją pewnej klasy, a do każdego z nich przekazywane są obiekty za pomocą właściwości (ang. *props*). Stąd też, aby odwołać się do właściwości konkretnej instancji, należy skorzystać z `this.props`.

Kolejnym komponentem, który musimy stworzyć, jest komponent `Contacts`. Ma on za zadanie wyświetlić całą listę kontaktów:

```
var Contacts = React.createClass({
  propTypes: {
    items: React.PropTypes.array.isRequired,
  },

  render: function() {
    var contacts = this.props.items.map(function(contact) {
        return React.createElement(Contact, {item: contact, key: cont
    });

    return (
      React.createElement('ul', {className: 'contactsList'}, contacts
    );
  }
});
```

Opis metody map()

Tutaj, podobnie jak w komponencie `ContactForm`, również następuje walidacja właściwości, ale oprócz tego obserwujemy jeszcze jedną ciekawą rzecz — w metodzie `render` tworzymy listę kontaktów na podstawie przekazanych parametrów. Robimy to przy użyciu metody `map`, która bierze każdy element z tablicy `this.props.items` i przekształca je odpowiednią funkcją, tworząc nową listę elementów.

Jest to nasza pierwsza styczność z **programowaniem funkcyjnym w JavaScripcie**. Podczas tego kursu poznasz kilka technik związanych z tym paradygmatem. Za każdym razem będziemy sygnalizować to odpowiednią uwagą. Dlaczego programowanie funkcyjne? W ostatnim roku, FP (ang. *functional programming*) jako paradygmat zdobyło spore uznanie w środowisku front-end developerów i obecnie jest swego rodzaju trendem, który najzwyczajniej w świecie warto znać. Nie jest to technika ani lepsza ani gorsza od OOP (ang. *Object Oriented Programming*). Jest po prostu… inna :)

Wróćmy do konkretów — `map` jest jedną z charakterystycznych metod, którą możemy wywołać w kontekście dowolnej tablicy. W efekcie możemy stworzyć nową tablicę, modyfikując dowolnie każdy z jej elementów. Po więcej informacji zerknij do dokumentacji.

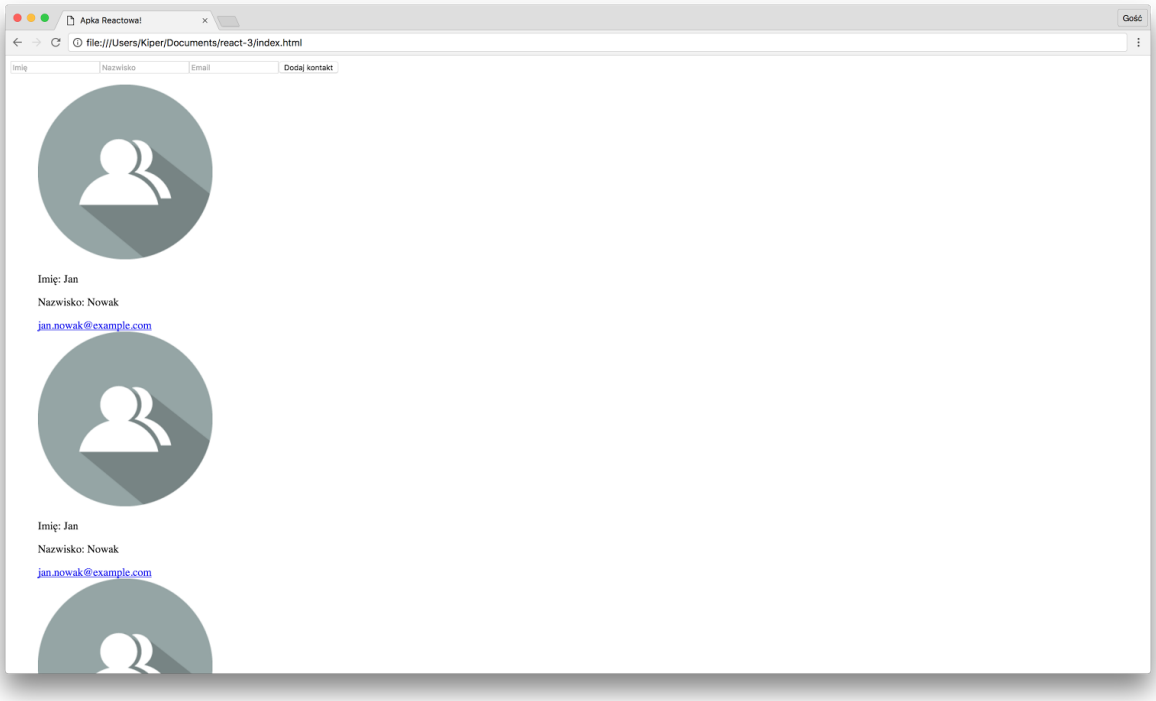
Ostatni komponent — Contact

Wracając do naszej aplikacji — przygotowaliśmy już trzy z czterech potrzebnych nam komponentów: `App`, `Contacts` i `ContactForm`. Został nam jeszcze jeden komponent, z którego korzystamy w komponencie `Contacts`, a mianowicie pojedynczy element kontaktowy (`Contact`):

```
var Contact = React.createClass({
  propTypes: {
    item: React.PropTypes.object.isRequired,
  },

  render: function() {
    return (
      React.createElement('div', {className: 'contactItem'},
        React.createElement('img', {
          className: 'contactImage',
          src: 'http://icons.veryicon.com/ico/System/100%20Flat%20Vol
        }),
        React.createElement('p', {className: 'contactLabel'}, 'Imię:
        React.createElement('p', {className: 'contactLabel'}, 'Nazwis
        React.createElement('a', {className: 'contactEmail', href: 'm
          this.props.item.email
        )
      )
    )
  },
});
```

Nie ma tu w zasadzie niczego nowego, o czym byśmy już sobie wcześniej nie powiedzieli. Na samym początku występuje walidacja właściwości, które wchodzą w skład komponentu (tutaj tylko jedna właściwość `item`), a następnie możemy odwołać się do tej właściwości dzięki `this.props.item`.

Jeśli wszystko wykonaliśmy prawidłowo, nasza aplikacja powinna wyglądać następująco:



Ostatnim krokiem będzie oczywiście prawidłowe ostylowanie, które będzie już Twoim zadaniem :)

Zadanie: Aplikacja z listą kontaktów

Wykonaj aplikację z listą kontaktów na podstawie wskazówek z tego submodułu, a następnie ostyluj aplikację, dodaj zmiany do repozytorium i wyślij link swojemu mentorowi.

Podgląd zadania

https://github.com/0na/

Wyślij link

14.6. Stan i cykle życia komponentu

Poznaliśmy już sposób na tworzenie własnych komponentów w React. Wiemy też, że komponenty można stworzyć za pomocą metody `createClass`, a instancje tych komponentów nazywamy `ReactElement`. Wiemy również, że na wejściu nowego elementu musimy określić jego właściwości (propsy), których typy deklarujemy podczas tworzenia klasy. Zostały nam jeszcze dwa elementy, które trzeba znać w kontekście komponentów.

Stan (state)

Każdy komponent może posiadać swój wewnętrzny stan, który będzie funkcjonował wewnątrz danego komponentu. Stan (ang. *state*), tak samo jak właściwości (propsy), określa zachowanie i sposób wyświetlania komponentu. Różnica między stanem a właściwościami polega na tym, że właściwości można określić podczas tworzenia elementu i mogą być one widoczne również poza nim, natomiast stan jest widoczny **jedynie** wewnątrz komponentu.

Kiedy mówimy o właściwościach, mamy na myśli inicjalizację (ustawianie wartości) komponentu. Stan możemy rozumieć jako wewnętrzny zestaw danych.

Nie istnieje żaden mechanizm, który mógłby przeprowadzać walidację typu stanu w przeciwieństwie do właściwości, których rodzaj określaliśmy w poprzednich zadaniach. Nie ma sensu walidować stanu, gdyż tylko Ty (jako programista) z niego korzystasz. Gdy tworzysz komponenty, mogą być one używane przez innych użytkowników, którzy jako interfejs widzą jedynie właściwości, dlatego te trzeba walidować.

Komponent wykorzystujący stan

Stwórzmy sobie prosty komponent, którego zadaniem będzie zwiększanie/zmniejszanie licznika, który to będzie wewnętrznym stanem komponentu:

```
var Counter = React.createClass({
    getInitialState: function() {
        return {
            counter: 0
        };
    },

    render: function() {
        return React.createElement('div', {},
            React.createElement('span', {}, 'Licznik ' + this.state.c
        );
    }
});

var element = React.createElement(Counter);
ReactDOM.render(element, document.getElementById('app'));
```

Jak widzisz, pojawiła się metoda `getInitialState`, której do tej pory nie znaliśmy. Dzięki niej jesteśmy w stanie określić początkowy stan naszego komponentu. Metoda ta powinna zwracać obiekt, do którego możemy się dostać, odwołując się do parametru `this.state`. W tym konkretnym przykładzie chcemy odwołać się do wartości, która znajduje się pod kluczem `counter`, stąd `this.state.counter`.

UWAGA: nie wolno odwoływać się do stanu, który nie ma wartości początkowej. Może to skutkować pojawieniem się błędów.

Występowanie stanu ma sens tylko wtedy, kiedy jego wartość będzie się zmieniała. Dopiszmy więc logikę, która będzie zajmować się inkrementacją (zwiększaniem wartości o 1) naszego licznika:

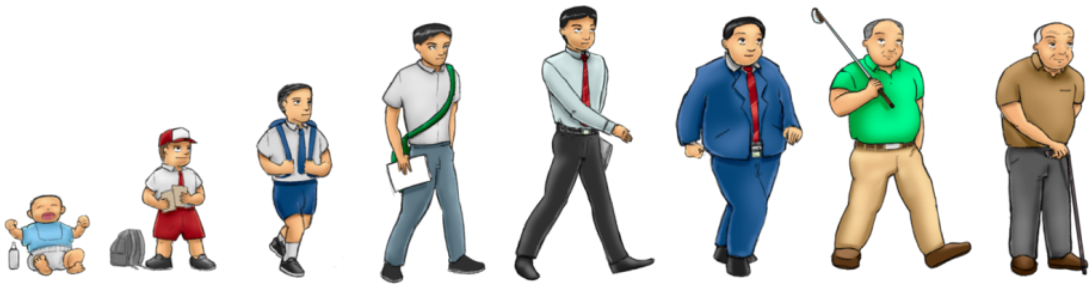
```
var Counter = React.createClass({
    getInitialState: function() {
        return {
            counter: 0
        };
    },

    increment: function() {
        this.setState({
            counter: this.state.counter + 1
        });
    },

    render: function() {
        return React.createElement('div', {onClick: this.increment},
            React.createElement('span', {}, 'Licznik ' + this.state.c
        );
    }
});
```

Do tego celu napisaliśmy funkcję `increment`, która zajmuje się odpowiednią zmianą stanu. Stan ustawiamy za pomocą metody `setState`. Aby jej użyć, przekazujemy do niej obiekt, w którym zawieramy klucze i ich wartości mające ulec zmianie. W tym wypadku chcemy, aby stan ustawił się na taki, jaki był, plus jeden. Następnie, za pomocą właściwości `onClick` ustawiamy, jaka funkcja ma się wykonać po wystąpieniu zdarzenia kliknięcia. Więcej na temat wspieranych zdarzeń dowiesz się z tego źródła.

UWAGA: stanu należy używać tak rzadko, jak to tylko możliwe. Jeśli nie musisz — nie twórz stanu. Znaczna większość komponentów to tzw. komponenty statyczne (ang. *static components*, *dummy components*), ale o tym napiszemy jeszcze w następnych rozdziałach.

Przejdźmy teraz do kolejnego zagadnienia, jakim są cykle życia komponentu.

Cykle życia



Każdy komponent, który jest tworzony przy użyciu danej klasy, ma swój cykl życia (ang. *component lifecycle*). Klasy udostępniają nie tylko metodę react, ale także cały zestaw innych funkcji, które wykonywane są w odpowiednich momentach życia komponentu. Zrozumienie ich pomoże Ci w wykonywaniu pewnych operacji w trakcie powstawania i usuwania komponentów, które będziemy tworzyć.

Cały cykl życia komponentu pokazuje poniższy diagram :)

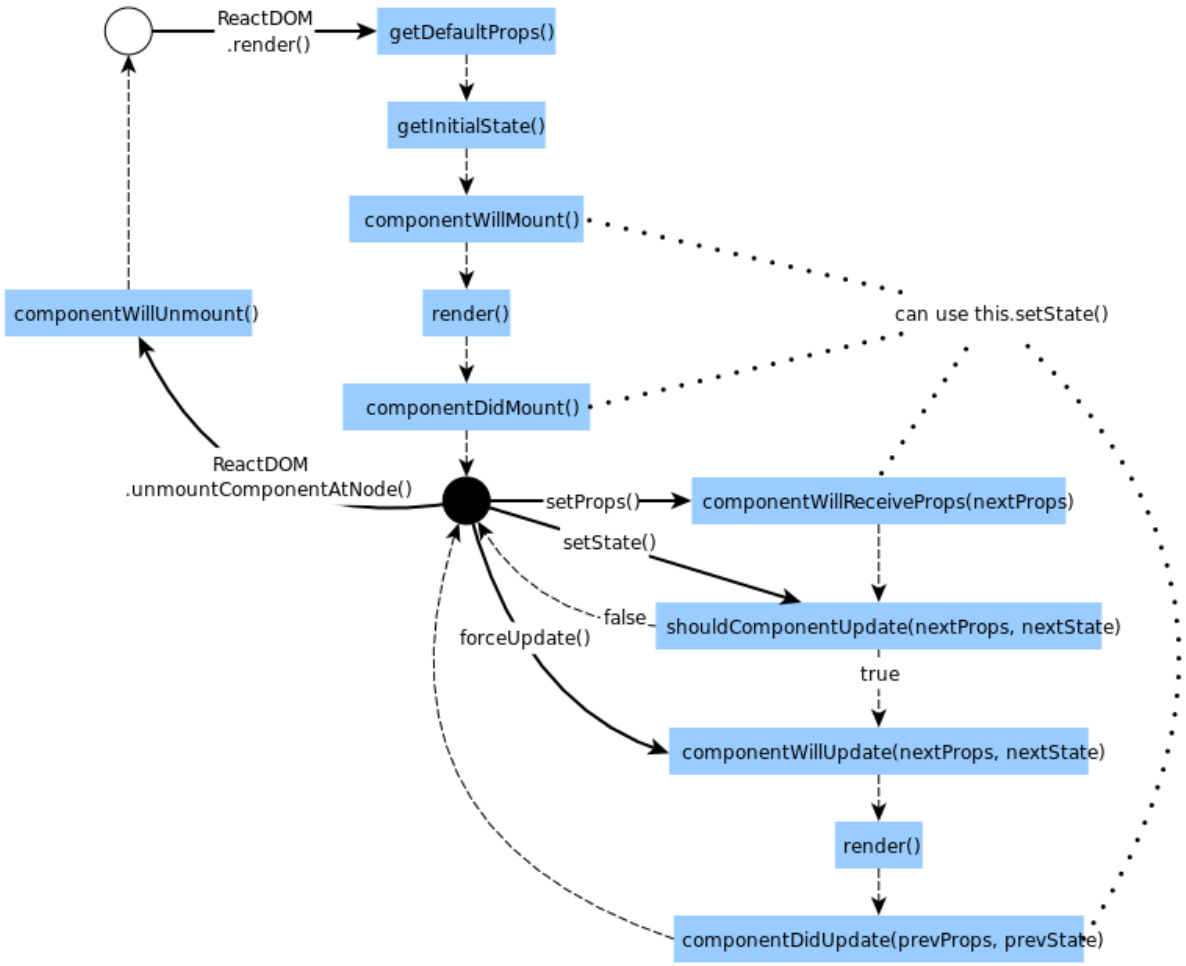
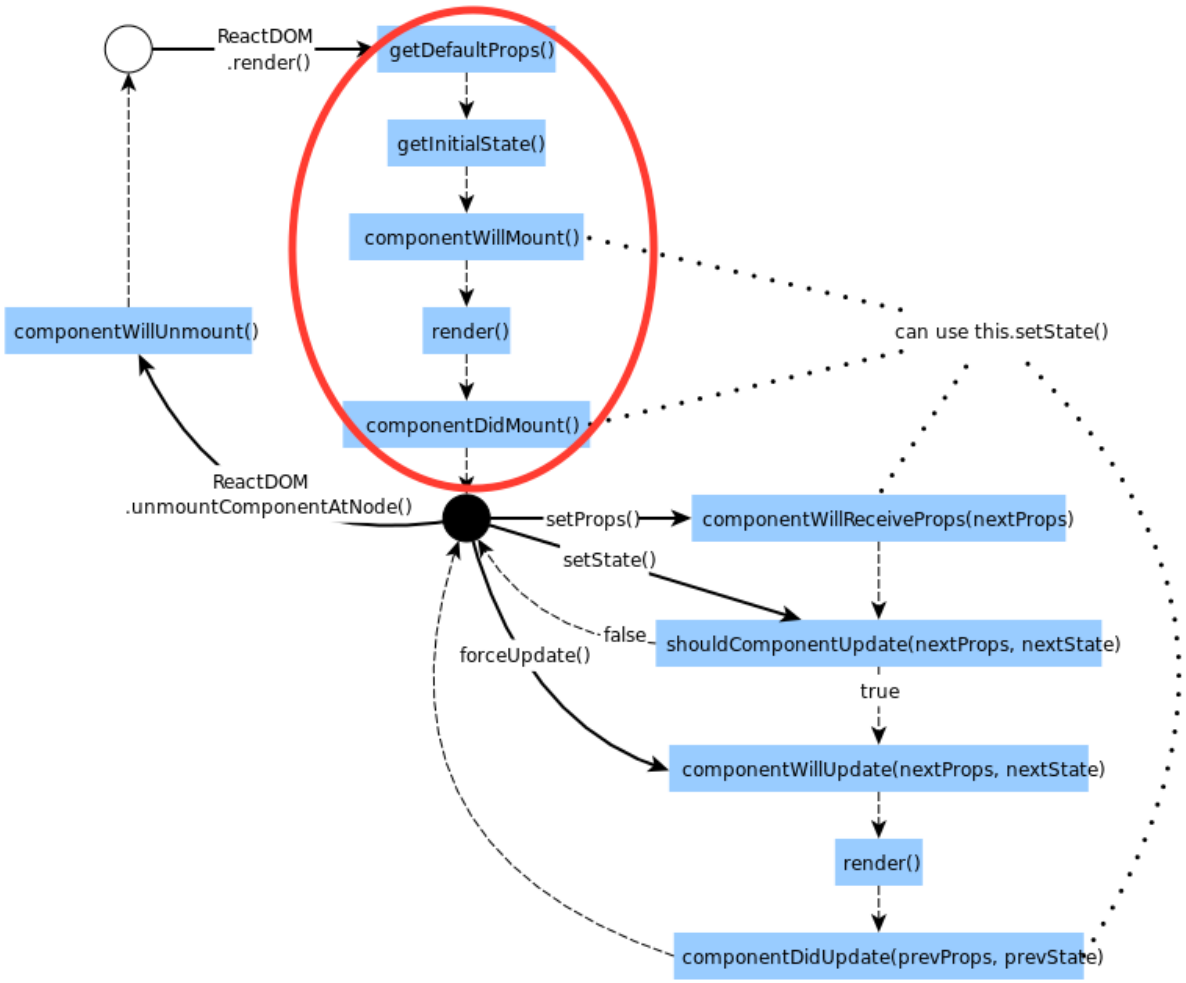


Diagram wydaje się nieco skomplikowany, ale spokojnie — omówimy sobie jego wszystkie elementy, dzięki czemu będziemy w stanie w pełni świadomie korzystać z komponentów.

Zacznijmy może od tego, że cykl życia można podzielić na trzy wyraźne etapy:

- inicjalizacji (montowania) — kiedy komponent jest tworzony
- aktualizacji — kiedy w komponencie zachodzą zmiany
- usuwania (odmontowywania) — kiedy komponent jest usuwany

Faza inicjalizacji



Na powyższym diagramie zaznaczone są bloczki, które można zaliczyć do fazy inicjalizacji, czyli tej, w której tworzymy nowy komponent. Pierwsze dwie metody, które zostają wywołane, to `getDefaultProps` i `getInitialState`. Metodę `getInitialState` poznaliśmy już w trakcie tworzenia licznika. Metoda `getDefaultProps` jest analogiczna do poprzedniej, tyle że ustawia domyślne wartości propsów (które nie zostały przekazane do komponentu).

Kolejne dwie metody wywoływane tylko w momencie inicjalizacji komponentu to `componentWillMount` i `componentDidMount`.

ComponentWillMount jest wywoływana zaraz przed wykonaniem metody `render`. Warto zaznaczyć, że ustawienie stanu w tej metodzie nie spowoduje przerenderowania komponentu.

Metoda `render` zwraca potrzebny do wyświetlenia `ReactElement`. Pamiętaj, że nie można zwrócić więcej niż jednego elementu! Jeśli chcemy zwrócić kilka elementów, możemy opakować je w np. element div — tak, jak zrobiliśmy to np. w komponencie `Contact`:

```
render: function() {
    return (
// zauważ, że metoda zwraca jeden element!
      React.createElement('div', {className: 'contactItem'},
        React.createElement('img', {
          className: 'contactImage',
          src: 'http://icons.veryicon.com/ico/System/100%20Flat%20Vol
        }),
        React.createElement('p', {className: 'contactLabel'}, 'Imię:
        React.createElement('p', {className: 'contactLabel'}, 'Nazwis
        React.createElement('a', {className: 'contactEmail', href: 'm
          this.props.item.email
        )
      )
    )
  }
```

Jeśli nie chcemy, aby komponent zwracał jakikolwiek elementy, możemy użyć wartości `null` lub `false`.

Należy pamiętać, że w metodzie render nie wolno modyfikować ani stanu, ani właściwości. Metoda ta ma być metodą **czystą**, o czym powiemy sobie niżej.

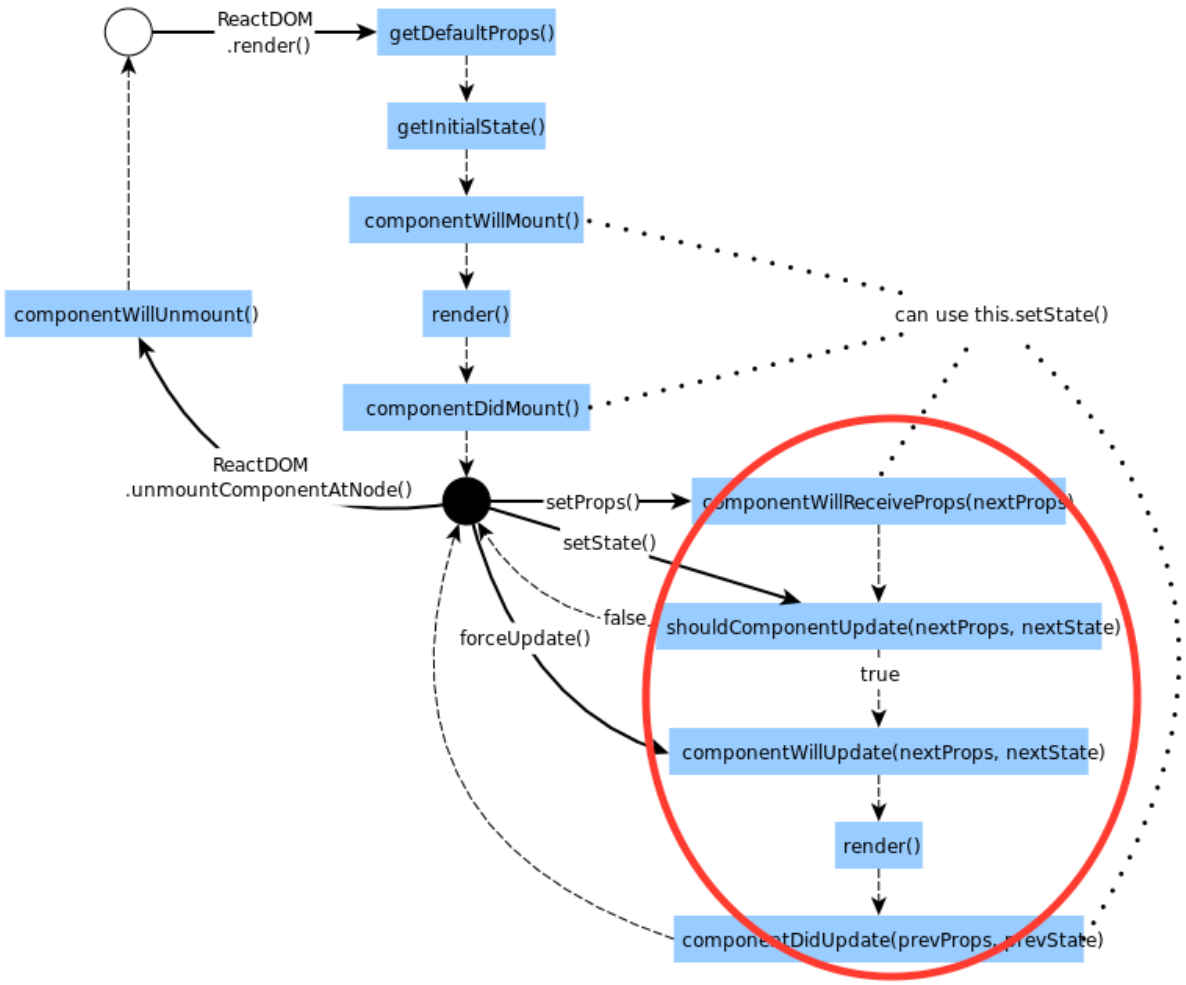
Programowanie funkcyjne

W paradygmacie funkcyjnym funkcja czysta (ang. *pure function*) to taka, która nie ma stanu wewnętrznego, przez co jeśli podamy jej n razy takie samo wejście, to n razy otrzymamy taki sam wynik. Funkcja ta za każdym razem zachowuje się tak samo. Można też powiedzieć, że nie ma pamięci wewnętrznej.

Jak tylko metoda render zostanie wywołana, od razu wywoływana jest metoda `componentDidMount`. W chwili wykonywania tej metody, nasz komponent widnieje już na stronie (jest zamontowany w drzewie DOM). Możemy wykonywać na nim różne operacje manipulacji, używać jQuery albo też pobrać dane. Dlaczego pobieramy dan

dopiero w tym miejscu? Dlatego, że chcemy jak najszybciej pokazać nasze komponenty użytkownikowi. Nie możemy sobie pozwolić na wstrzymanie działania aplikacji tylko dlatego, że czekamy na odpowiedź z serwera.

Faza aktualizacji



Faza aktualizacji (zmiana stanu albo otrzymanie nowych propsów) również wiąże się z wywołaniem pewnych metod.

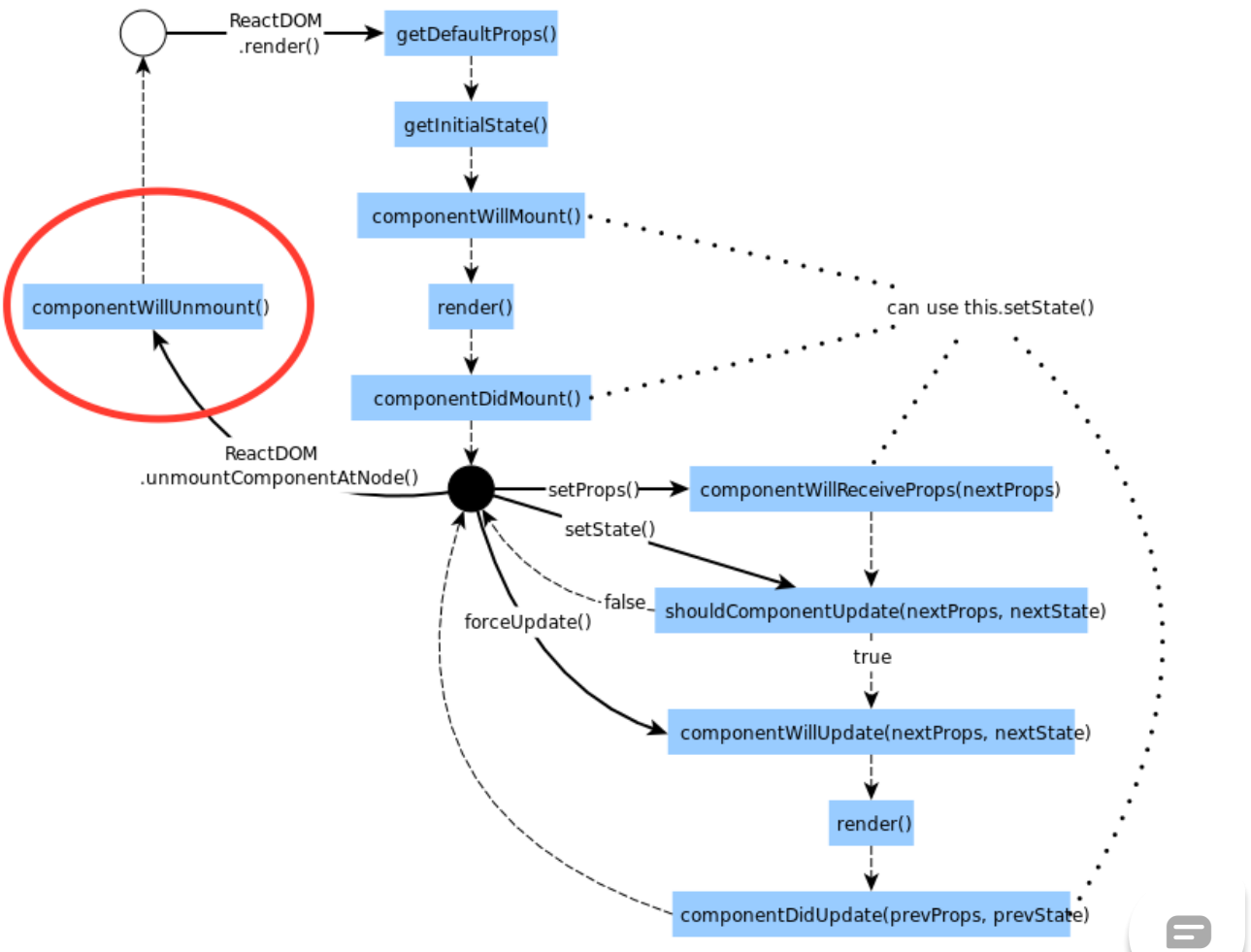
Zacznijmy od metody `componentWillReceiveProps`. Metoda ta, jak sama nazwa wskazuje, zostanie wywołana tylko wtedy, gdy komponent otrzyma nowe właściwości i nie jest to faza pierwszego renderowania (montowanie komponentu). Metoda ta pozwala aktualizować stan na podstawie nadchodzących właściwości. Warto nadmienić, że nie istnieje równoważna metoda dla stanu (stan nie może przyjść "z góry", czyli od rodzica komponentu).

Następna w fazie aktualizacji jest metoda `shouldComponentUpdate`. Metoda ta jest wywoływana tuż przed wywołaniem metody `render` i pozwala sprawdzić, czy faktycznie trzeba jeszcze raz przerysować komponent. Zwracana jest tutaj wartość true/false, czyli typ boolean. Metodę tę możemy zastosować, kiedy bardzo zależy nam na optymalizacji naszej aplikacji (pamiętaj, że operacje na DOM są bardzo kosztowne, a przerysowanie do nich należy).

Jeśli metoda `shouldComponentUpdate` zwróci wartość true, to natychmiast zostanie wywołana kolejna metoda: `componentWillUpdate`. Powinna zostać wywoływana tylko do przygotowania na nadchodzące zmiany. Nie wolno w tym miejscu modyfikować stanu (metoda `setState` tutaj nie działa).

Jako ostatnia zaraz po przerysowaniu komponentu wywołuje się metoda `componentDidUpdate` — w niej możemy wykonać np. jakieś manipulacje DOM (analogicznie do metody `componentDidMount`).

Faza usuwania



Życie naszego komponentu dobiega końca, służył nam dobrze, ale nie jest nam już potrzebny. Zanim jednak się go pozbędziemy może się zdarzyć, że będziemy musieli załatwić jeszcze pewne sprawy, posprzątać. Do tego celu służy metoda `componentWillUnmount`, w której możemy wykonywać wszystkie rzeczy związane z odpinaniem timerów czy nasłuchiwania zdarzeń na elementach DOM.

To by było na tyle, jeśli chodzi o komponenty. Jeśli masz jeszcze jakieś wątpliwości, zajrzyj do dokumentacji.

Zadanie: Niezależne liczniki

Napisz metodę `decrement`, która będzie dekrementowała (odejmowała 1) od stanu licznika. Twoja aplikacja powinna mieć dwa przyciski, aby działały obie funkcjonalności (dodawania i odejmowania).

Aby dodatkowo przećwiczyć koncepty poznane w tym module, stwórz kilka liczników, aby przekonać się, że stany między komponentami są niezależne.

Kiedy skończysz, dodaj do komponentu `Counter` wszystkie metody cyklu życia komponentu. W każdej z nich wypisz `console.log` ze swoimi pomysłami na ich użycie, np. kiedy wczytałbyś coś z serwera albo kiedy zależy nam na zoptymalizowaniu działania komponentu itp.

Na koniec wrzuć swój kod na GitHub i przekaż link do repozytorium mentorowi :)

Podgląd zadania

Wyślij link

14.7. JSX - lekarstwo na problematyczną składnię tworzenia nowych elementów

React to naprawdę świetne narzędzie i mamy nadzieję, że i Ty po tych paru submodułach zaczynasz dostrzegać jego potęgę. Jest jednak pewien drobny szczegół, który bardzo utrudnia życie. Spójrzmy jeszcze raz na przykład komponentu `Contact`:

```
React.createElement('div', {className: 'contactItem'},
    React.createElement('img', {
      className: 'contactImage',
      src: 'http://icons.veryicon.com/ico/System/100%20Flat%20Vol.%20
    }),
    React.createElement('p', {className: 'contactLabel'}, 'Imię: ' +
    React.createElement('p', {className: 'contactLabel'}, 'Nazwisko:
    React.createElement('a', {className: 'contactEmail', href: 'mailt
      this.props.item.email
    )
  )
```

Za każdym razem, gdy tworzymy nowy element, robimy to za pomocą metody `React.createElement`. Jest to bardzo uciążliwa rzecz, szczególnie, że widać w niej pewne podobieństwa i bardzo ciężko wyłapać jest różnice między poszczególnymi fragmentami kodu. Brak czytelności wiąże się z dłuższym czasem analizy kodu i większą szansą na wystąpienie błędu. Na szczęście okazuje się, że można temu w prosty sposób zaradzić.

Istnieje pewne rozwiązanie, które nazywa się *JSX*. Jest to preprocesor, który nakłada na JavaScript składnię XML (*eXtensible Markup Language*) znaną między innymi z HTML. Jak to wygląda w praktyce? Przekształćmy sobie kod napisany powyżej na postać JSX'ową:

```
<div className={'contactItem'}>
    <img className={'contactImage'} src={'link-do-obrazka.png'}/>
    <p className={'contactLabel'}>
        Imię: {this.props.contact.firstName}
    </p>
    <a href={'mailto:' + this.props.item.email}>
        {this.props.item.email}
    </a>
</div>
```

Prawda, że wygląda bardziej elegancko?

Gdyby porównać część z tego zapisu to:

```
<div>Hello!</div> jest równoznaczny React.createElement('div', {}, 'H
```

Teraz pojawi się najważniejsze zdanie, jeśli chodzi o ten submoduł. **JSX to nie jest HTML**. Pamiętaj, że JSX jest tylko upiększeniem składni (*ang. syntatic sugar*) i element `<div>` będzie kompilowany do `React.createElement('div', {}, {});`

Jest to również ukłon w stronę designerów, którzy znając HTMLa również mogą tworzyć własne komponenty.

Pewnie gdzieś z tyłu głowy zdajesz sobie pytania: dlaczego JSX nie został przedstawiony na samym początku? Dlaczego ciągle używaliśmy `React.createElement` zamiast użyć tej składni?

Po pierwsze: bardzo zależy nam na tym, aby podkreślić, że JSX to nie HTML w JavaScripcie. To tylko składania, którą posługujemy się do tworzenia komponentów.

Po drugie: jeśli chcesz dobrze poznać Reacta i uniknąć tzw. zmęczenia JavaScriptem (ang. JavaScript fatigue), warto poznawać rzeczy w chronologicznej kolejności i uzupełniać układankę o kolejne puzzle, które poruszymy w kolejnych modułach

To by było na tyle, jeśli chodzi o samego Reacta. Oczywiście istnieje sporo różnych smaczków, które na pewno napotkasz w swojej karierze, ale pamiętaj, że zawsze możesz skorzystać z dokumentacji. Pozostał Ci jeszcze do zrobienia mały projekt, o którym powiemy sobie w następnym submodule.

Ważne

JSX nie zadziała w naszej przeglądarce jeśli Babel nie będzie załączony. Istnieje kilka sposobów aby dodać go do naczego projektu. Najczęściej używane to:

- Załączenie go jako skryptu: `<script src="https://github.com/babel/babel-standalone/releases/download/release-6.26.0/babel.min.js"></script>`
- Instalacja poprzez NPM: `npm install --save babel-standalone`

Ponadto w strukturze HTML powinniśmy załączyć nasze skrypty z odpowiednim typem: `<script type="text/babel" src="scripts.js"></script>`

Nasz projekt powinien być na serwerze — powyższe rozwiązanie nie zadziała, jeśli strona będzie miała protokół `file:///`

Zadanie: Wykorzystujemy JSX

Przerób na JSX komponenty z zadania w submodule 5 — Ćwiczenie: Komponenty "kontakt" i "formularz". Następnie wyślij mentorowi link do repozytorium.

Podgląd zadania

Wyślij link

14.8. PROJEKT: Wyszukiwarka gifów

W tym projekcie stworzymy sobie dynamiczną wyszukiwarkę gifów. W polu do wyszukiwania będzie można wpisać wybraną frazę, a gify będą pobierane automatycznie. Do dzieła!

Przedstawienie aplikacji

Pierwszym krokiem będzie stworzenie wizji, jak taka aplikacja miałaby wyglądać. Po przeczytaniu powyższych wymagań (od np. potencjalnego klienta), nasza aplikacja powinna w efekcie końcowym wyglądać mniej więcej tak:

Tutaj widzimy gotową już aplikację. Normalnie, dla bardziej zaawansowanych aplikacji, tworzy się makiety (o których mówiliśmy w poprzednich submodułach), ale ta apka jest na tyle drobna, że nie ma potrzeby ich konstruować.

Wydzielenie potrzebnych komponentów

Nasza aplikacja będzie się składać z 3 komponentów:

- App — tutaj tradycyjnie wszystko będziemy wiązać w jeden byt. W tym komponencie będziemy odbierać wiadomość z komponentu zajmującego się wyszukiwaniem i przekazywać do komponentu, który wyświetli odpowiedniego gifa.
- Search — nie trzeba tutaj zbyt wiele wyjaśniać. Ten komponent to pasek wyszukiwania. Napiszemy w nim drobną logikę, która rozpocznie wyszukiwanie, kiedy przyciśniemy Enter, albo kiedy długość wpisywanego tekstu będzie większa niż 2 litery.
- Gif — zajmie się wyświetleniem gifa albo loadera (czyli kręcącego się kółka oznajmującego ładowanie obrazka) w przypadku kiedy nawiązywane jest połączenie z serwerem.

Struktura plików oraz inicjalizacja projektu

Mamy już pogląd na nasze komponenty. Zobaczmy teraz, jak powinna wyglądać struktura plików:

Zacznijmy od stworzenia projektu za pomocą `npm init`. Jak widzisz, `npm` nie jest używany tylko w projektach tworzonych w Node. `npm` można z powodzeniem używać też na froncie, o czym jeszcze więcej powiemy w następnych modułach.

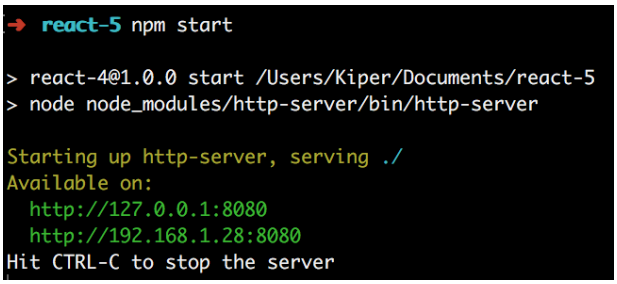
Kolejnym krokiem będzie zainstalowanie lokalnego serwera http, którego zadaniem będzie... serwowanie plików statycznych. Znamy już troszkę Node i moglibyśmy napisać taki serwer sami, ale nie będziemy wymyślać koła na nowo i pobierzemy sobie gotowe rozwiązanie:

```
npm install --save-dev http-server
```

Zauważ, że zainstalowaliśmy tę paczkę z flagą `--save-dev`. Różni się ona od `--save` tylko tym, że sygnalizuje komuś, kto ogląda plik package.json, że zainstalowana paczka służy tylko do celów developerskich. Może to być narzędzie do kompilacji albo tak jak w tym przypadku, zwykły serwer potrzebny do serwowania plików. Ta paczka nie będzie "do produkcji".

Napiszmy dodatkowo skrypt w pliku package.json, który będzie nam odpalał ten serwer:

```
"start": "http-server"
```



Przygotowanie struktury HTML

Serwer odpalamy komendą `npm start`, na razie jednak nie mamy nawet czego serwować. Stwórzmy wszystkie pliki, które zostały wymienione wyżej (wraz z katalogami) i rozpocznijmy od napisania pliku *index.html*:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Apka Reactowa!</title>
    <link rel="stylesheet" type="text/css" href="styles.css">
  </head>
  <body>
    <div id="app"></div>

      <script src="https://unpkg.com/react@15.3.2/dist/react.js"></sc
      <script src="https://unpkg.com/react-dom@15.3.2/dist/react-dom.
      <script src="https://unpkg.com/babel-core@5.8.38/browser.min.js

      <script>
        var App, Search, Gif
      </script>
      <script type="text/babel" src="components/Search.js"></script>
      <script type="text/babel" src="components/Gif.js"></script>
      <script type="text/babel" src="components/App.js"></script>
      <script type="text/babel" src="script.js"></script>
  </body>
</html>
```

Do "klasycznego" zestawu dodaliśmy na pierwszy rzut oka zbędny `<script>var App, Search, Gif</script>`. Jest w tym jednak pewien zamysł: zauważ, że w tym miejscu nie korzystamy z modułów tak, jak miało to miejsce w Node. Nie możemy więc zaimportować modułu Search lub Gif do modułu App. Z tego względu nasze komponenty muszą być zdefiniowane globalnie, dlatego też ich deklaracja została wysunięta na zewnątrz modułów.

Przygotowanie komponentów aplikacji

Jeśli chodzi o samą aplikację, to zaczniemy od góry i będziemy schodzić stopniowo w dół: najpierw napiszemy komponent App, żeby potem zejść do komponentów Gif i Search. Facebook dla prostych aplikacji zaleca stosowanie podejścia top-bottom, a dla bardziej skomplikowanych — bottom-top. Więcej informacji na ten temat możesz znaleźć tutaj. Zobaczmy, jak powinien wyglądać taki komponent.

Komponent App

```
App = React.createClass({
    render: function() {

        var styles = {
            margin: '0 auto',
            textAlign: 'center',
            width: '90%'
        };

        return (
          <div style={styles}>
                <h1>Wyszukiwarka GIFow!</h1>
                <p>Znajdź gifa na <a href='http://giphy.com'>giphy</a
                <Search />
            <Gif />
          </div>
        );
    }
});
```

Tak początkowo powinien wyglądać nasz komponent. Jak widzisz, dodaliśmy do niego również style, ale w inny niż dotychczas sposób (do tej pory używaliśmy `className`), a teraz użyliśmy stylów `inline`. Robimy to tworząc obiekt o dowolnej nazwie (tutaj również style) i przypisujemy go do dowolnego `ReactElement` do właściwości (również) `style`. Obiekt ten jest oczywiście zwykłym JavaScriptowym obiektem i nie ma tutaj żadnej specjalnej składni. Na szczególną uwagę zasługuje brak minusa we właściwości `textAlign: 'center'`. Nie mamy go z błahego powodu: minus to w JSie znak odejmowania :) Z tego powodu stosuje się konwencję camelCase.

Komponent Search

Ok, przejdźmy teraz do tworzenia kolejnych komponentów. Zaczniemy od komponentu `Search`:

```
Search = React.createClass({
    render: function() {
        var styles = {
            fontSize: '1.5em',
            width: '90%',
            maxWidth: '350px'
        };

    return <input
             type="text"
             onChange={this.handleChange}
             placeholder="Tutaj wpisz wyszukiwaną frazę"
             style={styles}
             value={this.state.searchTerm}
            />
  }
});
```

Tak powinien wyglądać komponent odpowiedzialny za wyszukiwanie. Tak naprawdę będzie to zwykły `input` (pamiętaj, że to nie jest HTML! Input to klasa stworzona przez zespół pracujący nad Reactem), który będzie wspomagany pewną logiką, która będzie obserwowała zmiany zachodzące w inpucie i wykonywać na nim pewne akcje, stąd `onChange={this.handleChange}`. Zaraz zajmiemy się implementacją tej funkcji, ale najpierw wyjaśnijmy sobie `value`.

Do tej właściwości przypisaliśmy wartość wewnętrznego stanu, a konkretnie parametru `searchTerm`. Dopiszmy sobie metodę ustawiającą wartość początkową stanu. Z poprzedniego submodułu o cyklach życia komponentu wiemy, że musi to być metoda `getInitialState`:

```
getInitialState() {
  return {
    searchingText: ''
  };
},
```

Ok, sprawa ze stanem załatwiona, wróćmy sobie teraz na chwilę do metody `handleChange`. Po co właściwie jest nam potrzebna ta metoda? Oczywiście do tego, aby na zmianę wartości (`value`) `inputa` ładować na nowo odpowiednie obrazki, ale nie tylko! Okazuje się, że gdybyśmy chcieli spróbować zmienić wartość inputa na stronie, to nic by się nie stało. W zwykłym HTMLu mogliśmy dowolnie zmieniać tę wartość, ale nie tutaj. Dlaczego? Otóż wartość `value` trzyma wartość stanu, a nie te co wpisuje użytkownik. Nasze wpisywanie tekstu do inputa nie zmienia stanu. Musin napisać to sami! Spójrzmy więc na implementację tej funkcji:

```
handleChange: function(event) {
    var searchingText = event.target.value;
    this.setState({
        searchingText: searchingText
    });
},
```

Każda zmiana generuje pewne zdarzenie. Wciskanie klawiszy na klawiaturze to właśnie przykład takiego zdarzenia (`event`). Do wartości tego zdarzenia możemy się dostać wchodząc do klucza `target`, a następnie do klucza `value`. Mając tę wartość możemy zaktualizować stan naszego komponentu za pomocą znanej już metody `setState`.

Komponent Gif

Pozostał nam jeszcze jeden komponent — Gif.

```
var GIPHY_LOADING_URL = 'http://www.ifmo.ru/images/loader.gif';
var styles = {
  minHeight: '310px',
  margin: '0.5em'
};

Gif = React.createClass({
  getUrl: function() {
    return this.props.sourceUrl || GIPHY_LOADING_URL;
  },
  render: function() {
    var url = this.props.loading ? GIPHY_LOADING_URL : this.props.url

    return (
      <div style={styles}>
        <a href={this.getUrl()} title='view this on giphy' target='ne
          <img id='gif' src={url} style={{width: '100%', maxWidth: '3
        </a>
      </div>
    );
  }
});
```

Zacznijmy klasycznie od omówienia metody `render`. Do wyświetlenia gifa potrzebujemy jego adresu URL. Skąd go pobrać? Tą kwestią będzie się zajmować komponent `App` (rodzic komponenta `Gif`). Wynik pobierania będzie przekazywany do dziecka (`Gif`) przez właściwości (`this.props.url`), ale nie tak prędko! Skąd wiemy, czy gif nie jest przypadkiem właśnie pobierany? O tym też wie komponent `App`. Dlatego linijka:

```
var url = this.props.loading ? GIPHY_LOADING_URL : this.props.url;
```

ustawia URL tylko, jeśli `this.props.loading` ma wartość `false`. W przeciwnym wypadku zostanie użyty URL ze zmiennej `GIPHY_LOADING_URL`, która została zadeklarowana na samej górze (adres ten można zmienić dowolnie na taki, jaki nam odpowiada). Poniżej została przedstawiona sytuacja ładowania:

Ostatnim ważnym elementem komponentu jest pobranie adresu obrazka, abyśmy mogli wejść na stronę Giphy i zobaczyć np. jego komentarze. Robimy to za pomocą metody `this.getUrl`. Zwraca ona adres do strony z pobranym obrazka albo do gifa sygnalizującego ładowanie (jeśli nie znajdzie tego pierwszego):

```
getUrl: function() {
    return this.props.sourceUrl || GIPHY_LOADING_URL;
},
```

Uzupełnienie komponentów brakującym kodem

Komponent Gif mamy z głowy. Wróćmy teraz do komponentu `App` i dopiszmy brakujące funkcjonalności. Zacznijmy od uzupełniania komponentu `Gif` o jego brakujące właściwości, których używamy w jego wnętrzu:

```
<Gif
    loading={this.state.loading}
    url={this.state.gif.url}
    sourceUrl={this.state.gif.sourceUrl}
/>
```

Jak widzisz, komponent App posiada swój wewnętrzny stan, który zawiera informacje o tym, czy gif jest ładowany, jaki jest jego bezpośredni adres URL (który używamy w `<img src={url}… />`) oraz jaki ma adres do strony z gifem (używany w `<a href={srcUrl}>...</a>`).

Zacznijmy od ustawienia wartości początkowych stanu, a następnie przejdziemy do implementacji metod, które będą z nim powiązane:

```
getInitialState() {
    return {
        loading: false,
        searchingText: '',
        gif: {}
    };
},
```

Jest tutaj jedna wartość, o której wcześniej nie wspomnieliśmy. Skoro komponent `App` zajmuje się pobieraniem gifa, to potrzebuje informacji na temat tego, jaką frazę ma wyszukiwać — stąd też klucz `searchingText`, który musimy odbierać od komponentu `Search`. Wiemy, w jaki sposób komunikować się w dół (od rodzica do dziecka), używaliśmy do tego właściwości (`props`), ale co z sytuacją odwrotną? W jaki sposób dziecko ma się dostać do swojego rodzica?

Okazuje się, że można to zrobić wywołując `callback`, czyli funkcję, którą rodzic przekazuje mu w propsach. Dzięki niej rodzic zobaczy, że jego dziecko ma dla niego jakąś wiadomość. Można też użyć bąbelkowania zdarzeń (podobne do emitera zdarzeń w Node), ale jest to nieco inny temat. Wróćmy na moment do komponentu `Search` i zaimplementujmy komunikację dziecko-rodzic:

```
handleChange: function(event) {
    var searchingText = event.target.value;
    this.setState({searchingText: searchingText});

    if (searchingText.length > 2) {
      this.props.onSearch(searchingText);
    }
  },

  handleKeyUp: function(event) {
    if (event.keyCode === 13) {
      this.props.onSearch(this.state.searchingText);
    }
  },

render: function() {
    var styles = {fontSize: '1.5em', width: '90%', maxWidth: '350px'};

    return <input
             type="text"
             onChange={this.handleChange}
             onKeyUp={this.handleKeyUp}
             placeholder="Tutaj wpisz wyszukiwaną frazę"
             style={styles}
             value={this.state.searchTerm}
            />
  }
```

Jak widzisz, do metody **handleChange** dodaliśmy warunek, który sprawdza, czy wpisywany przez nas tekst ma więcej niż 2 litery. Nie chcemy po prostu za każdym razem wysyłać zapytania o gif, które ma 2 znaki, bo i tak nie dostaniemy poprawnej odpowiedzi... zresztą, co miałby zwrócić nam serwer na zapytanie "ca"? Fraza "cat" byłaby już bardziej konkretną wiadomością :) Wróćmy do kodu. Jeśli warunek został spełniony, odpalamy funkcję przekazaną do właściwości:

```
this.props.onSearch(searchingText).
```

Wartość, którą wysyłamy, to oczywiście wpisany przez nas tekst. Dodaliśmy również nasłuchiwanie na wciśnięcie, a właściwie na "odciśnięcie" klawisza Enter (**onKeyUp**):

```
onKeyUp={this.handleKeyUp}
```

Dopisaliśmy też metodę, która będzie rozpoznawała, że wciśnięty klawisz to Enter i również wyśle wiadomość do rodzica, aby ten jeszcze raz uruchomił funkcję wysyłającą zapytanie po gifa:

```
handleKeyUp: function(event) {
    if (event.keyCode === 13) {
        this.props.onSearch(this.state.searchingText);
    }
},
```

Informacja o klawiszu jest ukryta w obiekcie zdarzenia (`event`) pod kluczem `keyCode`. Kod klawisza Enter to 13. Jeśli wciśniemy więc Enter, to wywołana zostanie funkcja:

```
this.props.onSearch(this.state.searchingText);
```

Ok, sprawę wysyłania wiadomości od dziecka do rodzica mamy z głowy. Wróćmy teraz do komponentu App i zaimplementujmy ostatnie fragmenty układanki — pobieranie gifa i obsługę wyszukiwania.

Pierwszym krokiem będzie dodanie właściwości dla komponentu `Search`:

```
<Search onSearch={this.handleSearch}/>
```

Kolejnym etapem będzie napisanie metody `handleSearch`:

```
handleSearch: function(searchingText) {  // 1.
    this.setState({
      loading: true  // 2.
    });
    this.getGif(searchingText, function(gif) {  // 3.
      this.setState({  // 4
        loading: false,  // a
        gif: gif,  // b
        searchingText: searchingText  // c
      });
    }.bind(this));
  },
```

Algorytm postępowania dla tej metody jest następujący:

1. Pobierz na wejściu wpisywany tekst.
2. Zasygnalizuj, że zaczął się proces ładowania.

3. Rozpocznij pobieranie gifa.
4. Na zakończenie pobierania:
   - przestań sygnalizować ładowanie,
   - ustaw nowego gifa z wyniku pobierania,
   - ustaw nowy stan dla wyszukiwanego tekstu.

Odpowiednie sekwencje oznaczyliśmy komentarzem w kodzie. Jest jednak jeszcze jedna rzecz wymagająca wyjaśnienia — dlaczego ustawiamy `this.getGif().bind(this)`?

Odpowiedź jest bardzo prosta: aby zachować kontekst. Przekazywana do metody `getGif` funkcja wskazuje na coś innego niż komponent App, dlatego trzeba posłużyć się pewnym obejściem (metoda `bind`), które zachowa odpowiedni kontekst.

Ostatni fragment wiążący całą aplikację to metoda `getGif`. Spójrzmy na jej implementację:

```
getGif: function(searchingText, callback) {  // 1.
    var url = GIPHY_API_URL + '/v1/gifs/random?api_key=' + GIPHY_PUB_
    var xhr = new XMLHttpRequest();  // 3.
    xhr.open('GET', url);
    xhr.onload = function() {
        if (xhr.status === 200) {
           var data = JSON.parse(xhr.responseText).data; // 4.
            var gif = {  // 5.
                url: data.fixed_width_downsampled_url,
                sourceUrl: data.url
            };
            callback(gif);  // 6.
        }
    };
    xhr.send();
},
```

Tak jak w poprzednim podpunkcie, rozpiszemy algorytm postępowania w punktach, zgodnie z odpowiadającymi im komentarzami w kodzie:

1. Na wejście metody `getGif` przyjmujemy dwa parametry: wpisywany tekst (`searchingText`) i funkcję, która ma się wykonać po pobraniu gifa (`callback`).
2. Konstruujemy adres URL dla API Giphy (pełną dokumentację znajdziesz pod tym adresem).
3. Wywołujemy całą sekwencję tworzenia zapytania XHR do serwera i wysyłamy
4. W obiekcie odpowiedzi mamy obiekt z danymi. W tym miejscu rozpakowujemy sobie do zmiennej `data`, aby nie pisać za każdym razem `response.data`.

5. Układamy obiekt gif na podstawie tego, co otrzymaliśmy z serwera
6. Przekazujemy obiekt do funkcji `callback`, którą przekazaliśmy jako drugi parametr metody `getGif`.

W kodzie wykorzystujemy dwie stałe wartości — `GIPHY_API_URL` oraz `GIPHY_PUB_KEY`. Obie powinny zostać utworzone w pliku z tym komponentem. Wartością dla `GIPHY_API_URL` będzie adres url API, który można znaleźć w dokumentacji — `https://api.giphy.com`. Klucz (wartość dla drugiej zmiennej) można uzyskać tworząc aplikację Giphy poprzez kliknięcie `Create an App` w prawym górnym rogu strony przeznaczonej dla developerów.

Jeśli wszystko dobrze zrobiliśmy, nasza aplikacja powinna spełniać swoje zadanie i będzie wyglądać mniej więcej tak:

Teraz możemy całymi dniami szukać gifów z kotami :)

Zadanie: Działająca wyszukiwarka gifów

Jeśli Twój projekt wyszukiwarki gifów jeszcze nie znalazł się na GitHubie, załóż teraz repozytorium i wyślij na nie kod projektu.

Link do repozytorium z ukończonym projektem wyślij do mentora.

Podgląd zadania

Wyślij link

Regulamin

Polityka prywatności