15. ES6 - nowe standardy JavaScript

Wyzwania:

- poznasz kolejne oblicze JavaScriptu, czyli ECMAScript
- zaczniesz pisać kod zgodnie z najnowszymi standardami
- dowiesz się czym są promise'y i jak z nich korzystać

Szablony

15.1. Czym jest ES6

ES6 jest akronimem od ang. *EcmaScript 6* (oficjalnie ES2015). Możesz też się spotkać z określeniem *Harmony* (szczególnie w środowisku Node). Jest to nic innego jak kolejna wersja specyfikacji JavaScriptu, czyli zbioru zasad, na podstawie których pisane są silniki obsługujące JS. W tym module zakładam, że znasz poprzednią specyfikację (ES5) chociaż w podstawowym stopniu.

Poprzednia specyfikacja JavaScriptu ES5 powstała w 2009 roku. Od tego czasu minęło już sporo czasu, a JavaScript jest dalej jednym z najpopularniejszych języków na świecie. Społeczność JSa, która musi pracować z tym językiem na co dzień, zaczęła się zastanawiać, czy można jakoś poprawić funkcjonowanie języka tak, aby spełniał współczesne potrzeby i w prostszy sposób rozwiązywał najczęściej spotykane problemy programistów.

Zespół najlepszych programistów uformował więc komitet TC39, który zajmuje się specyfikacją EcmaScript. Co jakiś czas członkowie tego komitetu organizują spotkania, na których omawiane są proponowane funkcjonalności, które mogłyby wchodzić w

kolejną wersję specyfikacji. Każda propozycja musi przejść przez proces ustalania m.in. zalet/wad danego rozwiązania. Jeśli pomysł zda egzamin, zostaje włączony do standardu.

Proces wdrożenia nowego standardu w życie to bardzo złożone przedsięwzięcie. Proces przejścia na nowy standard powinien odbyć się bez większego echa. Nie mam tu na myśli developerów, ale użytkowników przeglądarek, którzy korzystają ze stron, które zostały zaimplementowane w starym standardzie. W związku z tym, po wprowadzeniu zmian w zachowaniu silnika powinien on działać tak samo jak przed nimi. Nie jest to proste zadanie, a za każdym nowym elementem kryje się mnóstwo linii kodu testującego, czy ta kompatybilność wsteczna została zachowana - z tego względu proces zmian jest długi. Nie ma tu miejsca na żadną rewolucję, więc ES6 (i każdy kolejny standard EcmaScript) to po prostu nadzbiór jego poprzedniej wersji.

Pewnym problemem jest dostosowanie środowisk uruchomieniowych JSa do najnowszego standardu. Każde z nich pracuje nieco inaczej co my jako programiści musimy brać pod uwagę implementując kolejne funkcjonalności. W związku z tym możemy wziąć pod uwagę dwa scenariusze:

1. użytkownicy naszych aplikacji pracują wyłącznie w środowiskach obsługujących ES6,
2. kompilować kod do starszego standardu, o czym powiemy sobie jeszcze w kolejnych submodułach.

ES6 jako specyfikacja posiada pewne założenia do którego dąży:

1. być lepszym językiem do pisania bardziej złożonych aplikacji, bibliotek czy też generatorów kodu
2. poprawa współpracy z istniejącymi rozwiązaniami, przykładowo klasy mają zastąpić funkcje konstruujące
3. Poprawa wersjonowania kodu. Polega to głównie na tym, że nie istnieje coś takiego jak kod niekompatybilny z ES6 jako że jest to tylko nadzbiór poprzedniej wersji

Po tych wszystkich problemach, w 2015 roku, a dokładniej 17 czerwca, ES2015 został ogłoszony oficjalnym standardem pisania aplikacji. W jego skład wchodzi cała masa przeróżnych funkcjonalności, których nie sposób omówić w jednym module. My przedstawimy sobie te najczęściej używane i faktycznie wpływające na naszą pracę.

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

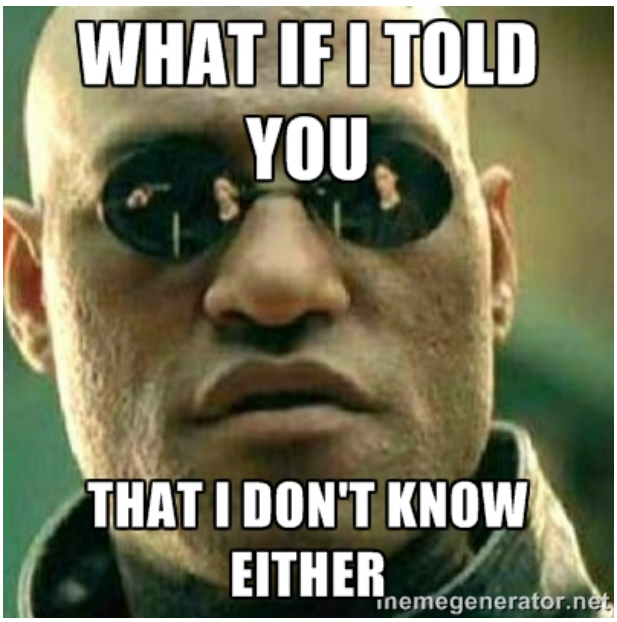
Zapoznałe(a)m się!

15.2. W jakim kierunku idzie nasza branża

Języki programowania dzielą się na dwie grupy: te, których nikt nie używa i te, na które wszyscy narzekają. Przez ostatnie kilka lat JavaScript miał podzielone opinie wśród programistów, ale czy tego chcemy czy nie, jest to język, którego używa się w przeglądarkach, bez których nikt nie wyobraża sobie dzisiaj życia.

Programiści, którzy muszą korzystać z tego momentami niewygodnego języka, tworzą na swoje potrzeby różne rozwiązania. Czasami znajdują one aprobatę wśród społeczności, która przyjmuje te rozwiązania jako coś, co warto znać i umieć. Takich pomysłów w postaci bibliotek i frameworków powstaje codziennie cała masa, przez co mamy wrażenie, że zostajemy ciągle w tyle.

To prawda, nasza branża nie spowalnia i nie sposób znać wszystkiego, ale musimy zadać sobie pytanie - czy faktycznie trzeba? Oczywiście, że nie. Biblioteka czy framework mają za zadanie rozwiązać pewien problem. React jest narzędziem, które rozwiązuje problem pisania złożonych aplikacji i robi to bardzo dobrze. Jest popularny i w tym momencie warto go znać, ale czy za parę lat dalej tak będzie?



To, co jest ponad bibliotekami i frameworkami to język, który jest mniej podatny na zmiany. Poza tym, te wszystkie narzędzia są w większości pisanie przy użyciu JavaScriptu i ten język z całą pewnością warto znać i szlifować.

Czy muszę uczyć się wszystkiego od nowa?

Oczywiście, że nie - jak już wcześniej wspomniałem ES2015 jest nadzbiorem poprzedniej wersji, więc kod, który pisaliśmy do tej pory jest teraz również jego częścią. To wszystko co wiemy na temat ES5 jest wciąż aktualne, nie musimy więc niczego uczyć się od nowa. Nie musimy także migrować naszego dotychczas napisanego kodu. Jest kilka rzeczy, które w pewien sposób zastępują funkcjonalności ES5, ale robią to bazując na dotychczasowych rozwiązaniach. Mimo wszystko, jeśli zależy nam na korzystaniu z ES2015 już dziś, musimy się przygotować na kompilowanie pisanego przez nas kodu do poprzedniej wersji EcmaScript, o czym opowiem już kolejnym submodule.

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

Zapoznałe(a)m się!

15.3. Babel: konfiguracja środowiska

Jak już wiesz z funkcjonalności ES6 możemy korzystać już dzisiaj, ale musimy brać pod uwagę środowiska, które nie mają jeszcze wbudowanej obsługi funkcjonalności tego standardu. Tutaj można podejrzeć które z nich działają, a które nie.

Inną, bardziej popularną metodą jest transpilowanie kodu z ES6 do ES5. Jednym z najbardziej popularnych narzędzi służących do tego celu jest właśnie Babel. Korzystaliśmy z niego przy okazji pisania komponentów za pomocą składni JSX. Tam należało dodać linka do zewnętrznego kompilatora, który pomagał nam pisać kod.

Pisząc kod ES6 będziemy korzystać z Babela, warto więc poznać sposoby jego konfiguracji. Zacznijmy od stworzenia w nowym katalogu pliku *package.json*. Robimy to za pomocą komendy `npm init`. Kolejnym krokiem będzie zainstalowanie potrzebnych nam zależności za pomocą komendy `npm install --save-dev babel-cli babel-preset-latest`

Przypominam, że flaga `--save-dev` odróżnia zależności, które będą wykorzystywane w aplikacji produkcyjnej od zależności developerskich, których potrzebujemy jedynie przy tworzeniu projektu.

Omówmy sobie pojedynczo paczki, które właśnie zainstalowaliśmy.

babel-cli - to wiersz poleceń dla Babela (ang. *command line interface*). Dzięki niemu w projekcie możemy korzystać z polecenia `babel`, które zajmuje się kompilacją pliku wejściowego (ze składnią ES6) na postać ES5 w pliku wyjściowym.

babel-preset-latest - to specjalne ustawienia w najnowszej wersji, które są wymagane do działania Babela.

Kolejnym krokiem będzie stworzenie pliku *.babelrc* albo rozpisanie konfiguracji w pliku *package.json*. Zdecydowanie lepiej jest jednak trzymać konfigurację w osobnym pliku. Konfiguracja odpowiada za zachowanie kompilatora. Można powiedzieć, że jest dla niego słownikiem, dzięki któremu wie jakie części kodu ma tłumaczyć. Pierwszym zbiorem pojęć, jakie mu dostarczymy, będą wcześniej zainstalowane presety (*babel-preset-latest*). W pliku powinno to wyglądać następująco:

```
{
  "presets": ["latest"]
}
```

Jak widzisz, korzystamy na razie z jednego presetu, ale jest ich znacznie więcej. Presety to zbiory wtyczek (ang. *plugin*), które bezpośrednio wpływają na tłumaczenie kodu. Chodzi tutaj głównie o to, żeby zamiast dołączania poszczególnych wtyczek dodać cały pakiet funkcjonalności w postaci właśnie presetu. *Latest* to najnowsza wersja posiadająca wszystkie najnowsze funkcjonalności. W kodzie produkcyjnym lepiej jest korzystać z konkretnych presetów takich jak: *es2015*, *es2016*, *es2017*, *stage-0*, *stage-1*, *stage-2*, *stage-3*, *stage-4*, czy *react*, które instalujemy odpowiednio komendą `npm install --save-dev babel-preset-<nazwa-presetu>`

Zauważ, że wspomniałem na temat presetów *es2016*, *es2017* - na razie się nimi nie przejmuj. Opowiem jeszcze o nich na koniec tego modułu.

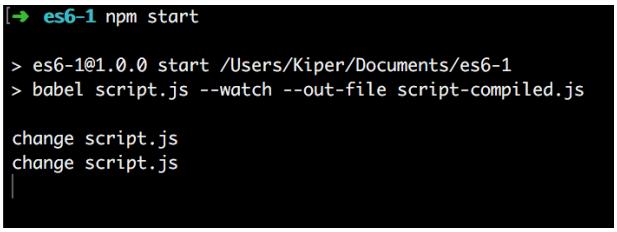
O co chodzi z presetami *stage-x*? Wszystkie funkcjonalności znajdujące się w nich to te, które nie dostały się do oficjalnego standardu ES2016. Wszystkie pluginy znajdujące się w presetach stadiów poniżej trzeciego są w fazie eksperymentalnej i należy korzystać z nich bardzo ostrożnie - oprócz ewentualnych błędów, które mogą wystąpić, ich funkcjonalności mogą się zmieniać. Okej, ale czym dokładnie są te stage-x? Jest to nic innego jak sposób podziału funkcjonalności w EcmaScript według stopnia 'dojrzałości' propozycji wprowadzony przez komitet TC39. Wyjaśnijmy sobie poszczególne stadia rozwoju funkcjonalności:

- *stage-0* - tylko pomysł, możliwe że stanie się kiedyś propozycją
- *stage-1* - propozycja, którą warto rozwijać
- *stage-2* - szkic posiadający wstępną specyfikację
- *stage-3* - kandydat posiadający kompletną specyfikację i pierwsze przeglądarkowe implementacje
- *stage-4* - zaadoptowane rozwiązania, które wejdą do kolejnej oficjalnej wersji języka

Póki co, najbardziej interesuje nas preset *es2015* (który zawiera się w presecie *latest*) i to właśnie nim się teraz zajmiemy. Ostatnią rzeczą, która wymaga konfiguracji, jest napisanie skryptu, który będzie obserwował nasze pliki i po zarejestrowaniu zmian dokona kompilacji plików dla naszej przeglądarki. W tym celu w pliku *package.json* musimy dołączyć do sekcji *scripts* następującą linijkę:

```
start: babel script.js --watch --out-file script-compiled.js
```

Po uruchomieniu polecenia i zmianie pliku *script.js* powinniśmy zobaczyć coś takiego:



Napiszmy sobie prosty program, aby potwierdzić, że nasz skrypt działa prawidłowo. W pliku *script.js* dodaj następujące linijki:

```
const sayHello = () => alert('Hello world!');
sayHello();
```

Póki co nie przejmuj się, jeśli nie rozumiesz tej konstrukcji - o tych i o wielu innych funkcjonalnościach opowiemy sobie za momencik. Jeśli wszystko napisaliśmy prawidłowo, to w naszym katalogu powinien powstać plik *script-compiled.js* z następującą treścią:

```
'use strict';

var sayHello = function sayHello() {
  return alert('Hello world!');
};

sayHello();
```

Taki widok oznacza, że proces konfiguracji babel.js mamy za sobą i możemy przystąpić do pisania kodu w stylu ES6!

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

Zapoznałe(a)m się!

15.4. Co nowego: składnia

W tym submodule opiszemy sobie najczęściej spotykane fragmenty składni (od ang. *syntax*) dzięki którym możemy osiągnąć ten sam efekt co przy użyciu starej składni, ale pisząc mniej kodu, przez co projekt staje się bardziej czytelny.

Let i const

W ES5 deklarowaliśmy zmienne przy użyciu słowa kluczowego `var`. Takie zmienne są powiązane tzw. zasięgiem funkcji (ang. *function-scope*). Początkującym programistom zasięg funkcji sprawia dużo kłopotów. Spójrzmy na przykład:

```
var greeting = 'Hello User!';
function greetWorld(isGreeting) {
    if (isGreeting) { // kod wykona się jeśli flaga isGreeting = true
        var greeting = 'Hello World!'; // (A) zasięg: cała funkcja
        return greeting;
    }
    return greeting; // zadziała hoisting i greeting nie będzie tym c
}
greetWorld(false); // undefined
```

Jak widzisz, przez hoisting zadziało się coś, co dla niektórych może być zaskakujące. Deklaracja zmiennej `greeting` wewnątrz funkcji `greetWorld` została przeniesiona na samą górę kodu, ale sama zmienna jest niezdefiniowana. Próba odwołania się do zmiennej `greeting` zwróci więc wartość `undefined`. Nie jest to coś, czego się spodziewaliśmy i nie wiedząc o *hoistingu*, możemy zastanawiać się co poszło nie tak.

W ES6 możemy używać zmiennych przy użyciu *let* lub *const*, które mają zasięg blokowy (ang. *block-scoped*). Blok to każda para nawiasów klamrowych. *let* to odpowiednik *var* tylko w innym zasięgu (blokowym), natomiast *const* działa w podobny sposób co *let*, tylko przypisana mu wartość nie może zostać zmieniona, czyli zmutowana.

Teraz pojawia się pytanie - kiedy używać słowa kluczowego *let*, a kiedy *const*?

Zasada jest taka, że należy używać *const* tam, gdzie jest to możliwe, ale w sytuacji, kiedy zmienna musi być przypisana jeszcze raz, trzeba używać *let*. Sytuacja taka może się np. przytrafić przy tworzeniu pętli *for*.

```
for (let i = 0; i < length; i++) {
    console.log(i);
}
```

const używamy, kiedy nie będziemy mieli zamiaru przypisywać zmiennej ponownie (czyli w większości przypadków). Jeśli będziesz miał napisać *let* zamiast *const*, dobrze się zastanów czy faktycznie będziesz zmieniał wartość zmiennej.

INFO: Programowanie funkcyjne

W tym miejscu pojawiło się kolejne pojęcie bardzo często używane w programowaniu funkcyjnym, czyli pojęcie *mutowalności*. Obiekt, który jest niemutowalny to taki, którego stan po inicjalizacji nie może ulec zmianie. W programowaniu każda zmiana stanu to tzw. skutek uboczny (ang. *side effect*). Jedna z reguł pisania programów w stylu funkcyjnym zakłada brak skutków ubocznych, co zapewnia jasny wgląd w stan naszej aplikacji. Dzięki temu nie musimy się martwić, że jakaś część aplikacji wpływa na stan w jej innej części.

Spróbujmy w przykładzie przywołanym przed chwilą zastąpić *var* słowem *let*:

```
let greeting = 'Hello User!';
function greetWorld(isGreeting) {
    if (isGreeting) {
        let greeting = 'Hello World!';
        return greeting;
    }
    return greeting;
}
greetWorld(false); // 'Hello User!'
```

Jak widzisz, *let* w powyższym przykładzie działa tylko w zasięgu blokowym. Oznacza to, że funkcja `greetWorld` traktuje inaczej zmienną `greeting` w instrukcji warunkowej i inaczej poza nią.

Dobre praktyki odnośnie używania *let* / *const* to między innymi:

- używać *const* wszędzie tam, gdzie zmienna nie zmienia wartości
- dla zmiennych, których wartość się zmienia używać *let* (np. podczas tworzenia pętli *for*)
- unikać słowa kluczowego *var*

Blok zamiast funkcji wywoływanej natychmiastowo

Kolejnym przykładem poprawy składni jest użycie bloku zamiast IIFE (ang. *Immediately-invoked function expression*), dzięki której mogliśmy zdefiniować funkcję i od razu ją wywołać. Wyglądało to w następujący sposób:

```
(function () {  // rozpoczęcie IIFE
    var hello = ...;
    ...
}());  // zakończenie IIFE
```

Powyższy zapis jest równy temu:

```
{  // rozpoczęcie bloku
    let hello = ...;
    ...
}  // zakończenie bloku
```

Szablony

Bardzo często korzysta się ze składania łańcucha znaków do konstruowania szablonów. Do tej pory używaliśmy następującej składni:

```
function sayHelloTo(person) {
    console.log('Hello, ' + person + ', nice to meet you!');
}
```

ES6 daje nam możliwość tworzenia szablonów (ang. *template literals*)

```
function sayHelloTo(person) {
    console.log(`Hello, ${person}, nice to meet you!`);
}
```

Należy zwrócić uwagę na dwie rzeczy. Po pierwsze zamiast rozpoczynać i kończyć stringi cudzysłowem lub apostrofem, używamy tzw. *backquote* lub *backtick*, czyli znaku `. *Backtick* zazwyczaj znajduje się tam gdzie tylda (~), czyli zazwyczaj tuż pod klawiszem esc, na lewo od cyfry 1. Po drugie, aby umieścić wartość zmiennej w szablonie używamy składni `${nazwaZmiennej}`.

Pisanie w `${}` nazywane jest "profesjonalnie" interpolacją. W tym miejscu możemy nie tylko odwoływać się do zmiennych, ale również wykonywać pewne wyrażenia takie jak np. dodawanie czy inne operacje na zmiennych jak np. metoda `join`, która łączy elementy tablicy za pomocą podanego separatora.

Dzięki temu zyskujemy też możliwość pisania szablonów wielolinijkowych, np.:

```
const HTML5_TEMPLATE = `
    <!DOCTYPE HTML>
    <html>
    <head>
        <meta charset="UTF-8">
        <title></title>
    </head>
    <body>
    </body>
    </html>`;
```

Arrow function

Jest to krótszy odpowiednik wyrażenia funkcyjnego. Przykładowy program, który napisaliśmy sobie w celu przetestowania konfiguracji Babela, to właśnie przykład *arr function* (`=>`). Składnia ta wnosi do naszego kodu dwie widoczne korzyści.

Po pierwsze, *arrow functions* są bardziej zwięzłe od tradycyjnych wyrażeń funkcyjnych, co widać na przykładzie poniżej.

```
// ES6
const numbers = [1, 2, 3];
const numbersPlusOne = numbers.map( x => x + 1 );
//ES5
var numbers = [1, 2, 3];
var numbersPlusOne = numbers.map(function (x) { return x + 1 });
```

Kolejną zaletą jest zmienione zachowanie *this* - to na co wskazuje słowo kluczowe *this* jest brane z otoczenia funkcji, a nie z kontekstu, w którym została użyta, dlatego nie trzeba już więcej robić sztuczek w stylu `var self = this` lub używać metody `bind`, aby poprawić kontekst.

To są główne zalety *arrow functions* - pora zapoznać się z ich konstrukcją. Możemy się spotkać z kilkoma zapisami funkcji tego typu:

Parametry możemy przekazywać w następujący sposób:

```
() => { ... } // bez parametru
x => { ... } // jeden parametr
(x, y) => { ... } // kilka parametrów
```

Jeśli nasza funkcja ma tylko jeden parametr, nie musimy otaczać go nawiasami - w przypadku większej ilości parametrów lub ich braku, należy użyć nawiasów.

Ciało funkcji może wyglądać następująco:

```
x => { return x * x }  // blok
x => x * x  // wyrażenie, takie samo jak powyższy kod
```

Kod otoczony nawiasami klamrowymi (blokiem) zachowuje się tutaj tak samo, jak w przypadku zwykłej funkcji i jeśli chcemy zwrócić jakąś wartość, musimy użyć słowa kluczowego *return*.

Destrukturyzacja

Czasami zdarza się, że funkcja zwraca obiekt albo tablicę i jeśli chcemy dostać się do poszczególnych fragmentów tych struktur, musimy tworzyć kilka linijek zmiennych np.:

```
function getCoords() {
    return {
        x: 2,
        y: 5
    };
}

const coords = getCoords();
const x = coords.x;
const y = coords.y;
```

Nie jest to specjalnie wygodne rozwiązanie, ale do tej pory nie mieliśmy innego wyjścia. ES6 przynosi nowe rozwiązanie, które pozwala na destrukturyzację obiektu/tablicy i wyciągnięcia z nich wartości, które nas interesują bez tworzenia masy zmiennych:

```
const {x, y} = getCoords();
```

Dzięki temu funkcja `getCoords`, która zwraca obiekt przypisze odpowiednio wartości kryjącą się pod `x` i `y` do odpowiadających im zmiennych. Trzy linijki kodu kontra jedna, czytelna linijka.

To samo można zastosować oczywiście w przypadku tablic:

```
const names = ['Jan', 'Zosia', 'Zbyszek', 'Kacper', 'Tomek', 'Magda']
const [first, second, , fourth] = names; // puste miejsce pomiędzy se
// first = 'Jan', second = 'Zosia', fourth = 'Kacper'
```

Zauważ, że podczas destrukturyzacji tablicy z imionami nie uwzględniliśmy trzeciej wartości, dlatego zostawiliśmy puste miejsce.

Domyślne wartości parametrów funkcji

Czasami, gdy wywołujemy funkcję, nie podajemy do niej pełnej ilości wymaganych parametrów. Co robiliśmy do tej pory? Używaliśmy znaku logicznego lub (`||`) co oznaczało, że jeśli nie podaliśmy wymaganego parametru, to ustawiała się jego

wartość domyślna.

```
function sayHello(name) {
  name = name || 'World';
  console.log('Hello ' + name);
}
```

W ES6 tę funkcję można zapisać w następujący sposób:

```
function sayHello(name = 'World') {
  console.log('Hello ' + name);
}
```

Korzystając z wcześniej poznanych elementów składni, można to zrobić nawet zwięźlej:

```
const sayHello = (name = 'World') => console.log(`Hello ${name}!`);
```

Domyślne parametry są ustawiane tylko wtedy, gdy wartość przekazywana do funkcji jest równa `undefined`. W ES5 każda wartość falsy ustawiała wartość podaną po znaku `||`, co mogło skutkować nieprzewidywanymi zachowaniami np.:

```
if ( [] ) {
  console.log('Ten kod wykona się, JS traktuje [] jako wartość true')
}
if ( [] == true ) {
  console.log('Niestety ten kod już się nie wykona, chociaż moglibyśm
}
if ( [] == false ) {
  console.log('Ten kod zostanie wykonany');
}
```

Rest params

Do tej pory, jeśli nasza funkcja przyjmowała nieokreśloną ilość parametrów, musieliśmy korzystać ze specjalnego parametru `arguments` - na przykład:

```
function logAllArguments() {
    for (var i=0; i < arguments.length; i++) {
        console.log(arguments[i]);
    }
}
```

Dzięki ES6, teraz możemy skorzystać z operatora *spread*, który zapisujemy jako trzy kropki - **...** . Zmodyfikujmy powyższy przykład tak, aby korzystał z tego operatora:

```
const logAllArguments = (...args) => args.forEach(arg => console.log(
```

Pierwszą rzeczą, na którą należy zwrócić uwagę jest ilość kodu potrzebna do zapisania tej samej funkcji - zredukowaliśmy ją z pięciu linijek do jednej! Aby to osiągnąć, zastosowaliśmy dwukrotnie *arrow function* i użyliśmy operatora *spread* do oznaczenia wszystkich argumentów funkcji.

Innym przykładem użycia tego operatora jest operacja na tablicach. Jeśli chcemy przykładowo zdestrukturyzować tablicę i wyciągnąć z niej dwa pierwsze elementy, ale chcemy przy okazji zachować resztę (ang. *rest*), możemy zrobić to w następujący sposób:

```
const names = ['Jan', 'Zosia', 'Zbyszek', 'Kacper', 'Tomek', 'Magda']
const [first, second, ...rest] = names;
```

Pod zmienną `rest` będzie przechowywana tablica imion `['Zbyszek, 'Kacper', 'Tomek', 'Magda'];`

Taki sam zabieg można również zastosować w przypadku deklaracji funkcji, gdzie przykładowo interesują nas dwa pierwsze elementy, a reszta jest dla nas mniej istotna:

```
function sayHello(first, second, ...rest) {
    // ciało funkcji
}
```

Kolejnym przydatnym zastosowaniem jest dołączanie nowych elementów do tablicy. Do tej pory korzystaliśmy z metody `push`, `pop`, `shift`, `unshift`. Teraz jeśli chcemy przykładowo dołączyć nowy element do tablicy możemy zrobić to w następujący sposób:

```
const newNames = [...names, 'Tadeusz'];
```

Połączenie kilku tablic działa dokładnie tak, jak prawdopodobnie już przewidujesz. Nie musimy już korzystać z metody concat, operator *spread* służy pomocą i w tym przypadku. Oto przykład łączenia kilku tablic w jedną:

```
const table = [...table1, ...table2, ...table3];
```

Zadanie: Ćwiczymy ES6

Zadanie pierwsze

Połączenie dwóch stringów przy użyciu operatora `+` jest bardzo łatwym zadaniem. Innym sposobem jest użycie metody `concat` albo `join`, ale co jeśli nie moglibyśmy skorzystać z żadnej z tych opcji? Twoim zadaniem będzie stworzenie dwóch zmiennych z wartościami `Hello` oraz `World`, a następnie połączenie ich metodą inną niż wymienione powyżej.

Zadanie drugie

Stwórz funkcję `multiply`, która ma zwracać wynik działania operacji mnożenia dwóch wartości `a` i `b`. Przykładowo:

```
multiply(2, 5) // 10
multiply(6, 6) // 36
```

Zadanie wydaje się być proste, ale co jeśli użytkownik poda na wejściu tylko jedną wartość? Przykładowo:

```
multiply(5) // ?
```

Chcemy, aby wynik takiego wywołania był również prawidłowy - możesz założyć, że jeśli użytkownik nie poda któregoś z parametrów, ma on zostać zastąpiony `1`. Nie wolno korzystać z instrukcji warunkowych! Funkcję stwórz za pomocą *arrow function*.

Zadanie trzecie

Napisz funkcję `average`, która obliczy średnią arytmetyczną wszystkich argumentów, które zostaną do niej przekazane. Załóż, że argumenty zawsze będą liczbami:

```
average(1) // 1
average(1, 3) // 2
average(1, 3, 6, 6) // 4
```

Skorzystaj z *rest parameters*! Funkcję stwórz za pomocą *arrow function*.

Zadanie czwarte

Stwórz tablicę z ocenami `const grades = [1, 5, 5, 5, 4, 3, 3, 2, 1]`, a następnie w umiejętny sposób przekaż oceny do funkcji `average` tak, aby otrzymać wynik. Skorzystaj z operatora `spread`!

Zadanie piąte

Podczas pracy nad projektem natknąłeś się na bardzo dziwną strukturę danych - `[1, 4, 'Iwona', false, 'Nowak']`. Twoim zadaniem jest skorzystanie z destrukturyzacji w celu wyciągnięcia z tablicy zmiennych `firstname` oraz `lastname`.

Po wykonaniu tych zadań, umieść odpowiednie pliki JavaScript na swoim Githubie i wyślij link do repozytorium mentorowi :)

Zadanie dodatkowe

Zarejestruj się na Codewars, a następnie rozwiąż poniższe zadania:

- Zadanie 1
- Zadanie 2
- Zadanie 3
- Zadanie 4

Podgląd zadania

Wyślij link

15.5. Co nowego: programowanie asynchroniczne

Środowiska JavaScript ze swojej natury są asynchroniczne - bazują na zdarzeniach, których nasłuchujemy, i wykonujemy pewien kod jako reakcję na nie. Dzieje się tak, ponieważ nie chcemy, aby wykonywanie kosztownej funkcji zawiesiło nasz interfejs. Znamy dwa sposoby na otrzymywanie rezultatu z funkcji asynchronicznej:

Zdarzenia

W tej metodzie tworzymy obiekt dla każdego żądania i przypinamy do niego zarejestrowane przez nas funkcje, które mają się wykonać na wystąpienie pewnego zdarzenia. W poniższym przykładzie są to zdarzenia udanego lub nieudanego zapytania:

```
var req = new XMLHttpRequest();
req.open('GET', url); //tutaj jakiś przykładowy endpoint

req.onload = function () {
    console.log('Załadowano dane');
};

req.onerror = function () {
    console.log('Ups, coś poszło nie tak!');
};

req.send(); // Dodaj zapytanie do kolejki zadań
```

Ostatnia linijka kodu dodaje zapytanie do kolejki zadań, nie jest wykonywana od razu. Z tego powodu mogliśmy umieścić tę linijkę również bezpośrednio po `req.open`.

Callbacki

Można również obsługiwać kod asynchroniczny za pomocą tzw. *callbacków*. Są to funkcje, które przypisuje się jako jeden z parametrów funkcji asynchronicznej. Tego typu rozwiązania stosowaliśmy w Node:

```
fs.readFile('myfile.txt', { encoding: 'utf8' },
    function (error, text) {
        console.log(text);
    }
);
```

Callbacki można zagnieżdżać jeden w drugim, otrzymując w ten sposób pewną kontrolę nad kolejnością wykonywanych zadań. Na przykład:

```
fs.readFile('myfile.txt', { encoding: 'utf8' },
    (error, text) => {
        console.log(text);
        fs.writeFile('myfile-copy.txt', text, () => {
          setTimeout(() => {
            console.log('Zapisano!');
          }, 1000);
        });
    }
);
```

Najpierw wykona się funkcja odczytująca plik *myfile.txt*. W kolejnym kroku metoda `writeFile` zapisze treść odczytanego pliku jako kopię do *myfile-copy.txt*. W momencie, gdy zapis się powiedzie, zarejestrowany zostanie *timeout*, który powiadomi nas o zakończonym zadaniu po odliczeniu jednej sekundy.

Taki sposób pisania kodu prowadzi do wielu zagnieżdżeń funkcji w funkcji, co jest określane jako *callback hell*. Wspominaliśmy o tym zjawisku w module o Node.

Można temu zapobiec, rozdrabniając callbacki na mniejsze funkcje:

```
const onTimeout = () => console.log('Zapisano!');

const onWriteSuccess = () => setTimeout(onTimeout, 1000);

const onReadSuccess(error, text) => {
  console.log(text);

  fs.writeFile('myfile-copy.txt', text, onWriteSuccess);
}

fs.readFile('myfile.txt', { encoding: 'utf8' }, onReadSuccess);
```

Zadanie wykona się dokładnie tak samo, ale jest tutaj jeden problem. Musimy dokładnie przeanalizować, jaki kod wykona się w pierwszej kolejności, i która funkcja wywoła kolejną.

Promise

Lekarstwem na powyższe problemy są tzw. obietnice (ang. *promise*). Są one nieco bardziej skomplikowane, jeśli chodzi o implementację, ale w zamian dają bardziej czytelny kod, który łatwiej się czyta i modyfikuje. Spójrzmy na poniższy kod, który zwraca wynik funkcji asynchronicznej przy użyciu *promise*:

```
function download() {
    return new Promise(
        function(resolve, reject) {
            ...
            resolve(result); //spełnienie obietnicy
            ...
            reject(error); //niespełnienie obietnicy
        });
}
```

Funkcja `download` zwraca nową obietnicę, która w konstruktorze przyjmuje funkcję anonimową z dwoma argumentami (`resolve` i `reject`). W środku "obietnicy" wykonywany jest kod asynchroniczny (w naszym przypadku będzie to pobieranie czegoś z serwera). Wynik tego pobierania, w zależności od tego, czy się powiedzie czy nie, przypisujemy do jednej ze zmiennych (w powyższym przykładzie są to odpowiednio zmienne `result` i `error`). Następnie każda zmienna jest przekazywana odpowiednio do funkcji `resolve` i `reject`, która obwieszcza spełnienie obietnicy lub jej odrzucenie.

Przykłady użycia *promise* pokażemy poniżej. Teraz zapamiętaj, w jaki sposób tworzony jest *promise* i jak zwraca wynik działania kodu asynchronicznego.

Powyższą funkcję wywołujemy w następujący sposób:

```
download()
  .then(result => …)
  .catch(error => …);
```

`.then()` to metoda, która wykona się w momencie, kiedy obietnica zwróci prawidłowy wynik (wywołana zostanie metoda `resolve`). Funkcja ta zawsze zwraca kolejnego promise'a, a zatem możemy łańcuchować nasze wywołania. Oto pseudokod oparty na poprzednim przykładzie z callbackami:

```
read()
  .then( result => write() )
  .then( result => timer() )
  .then( result => log() )
  .catch(error => …);
```

Funkcje `read`, `write`, `timer` i `log` to funkcje asynchroniczne opakowane w *promise*. Jak widzisz, w porównaniu do callbacków jest to o wiele czytelniejsza metoda. Wyraźnie widać, jaki kod wykonywany jest po spełnieniu kolejnych obietnic. Taki sposób wykonywania operacji asynchronicznych usuwa również problem zagnieżdżania i znacznie poprawia czytelność kodu. Nie spotkamy się tutaj więcej z problemem tzw. "callback hell", ponieważ struktura obsługi kolejnych wywołań funkcji asynchronicznych jest bradziej "płaska".

Promise'y były używane na długo przed dodaniem ich do specyfikacji ES6. Standard tej funkcjonalności jest opisany w specyfikacji Promise A+ i według niej zbudowane są takie biblioteki jak np. bluebird.js albo q. Na tym standardzie opiera się również implementacja *promise* w ES6.

Używanie promisów w naszym kodzie wymaga zainstalowania tzw. polyfilla, czyli pliku uzupełniającego brakujące funkcjonalności w przeglądarkach. Możesz to zrobić za pomocą komendy `npm install --save-dev babel-polyfill`.

Aby dodać *babel-polyfill* do naszego projektu, w pliku index.html (przed wszystkimi innymi skryptami js) powiniśmy dodać następujący skrypt:

```
<script type="text/javascript" src="<katalog-do-babel-polyfill>/dist/
```

Gdzie katalog-do-babel-polyfill to przeważnie folder *node_modules/babel-polyfill* w Twoim projekcie.

Tworzenie promise'ów

Promise tworzymy za pomocą konstruktora o takiej samej nazwie. Przyjmuje on jeden parametr - funkcję, która obejmuje kod asynchroniczny i może mieć dwa różne wyni rozwiązany (ang. *resolve*) - kiedy obietnica została “spełniona" i odrzucony (ang. *rejec*

kiedy obietnica “nie została spełniona". Spójrzmy na przykład:

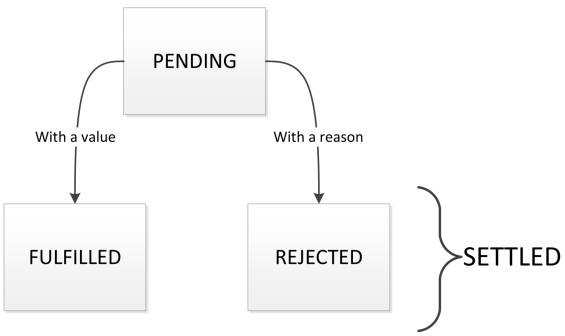
```
const p = new Promise(
  function (resolve, reject) {
      ...
      if (...) {
          resolve(value); // success
      } else {
          reject(reason); // failure
      }
  }
);
```

Stan promise'ów

Kiedy kod asynchroniczny został opakowany za pomocą promise'a, zostaje w nim zamknięty do czasu jego rozwiązania. Każdy z promise'ów ma trzy stany:

- oczekujący (ang. *pending*) - promise nie został jeszcze przeliczony, jest to stan początkowy każdego promise'a
- spełniony (ang. *fullfilled*) - rezultat został obliczony prawidłowo
- niespełniony (ang. *rejected*) - w trakcie wykonywania obliczeń coś poszło nie tak

Stany ustalone, które zwracają pewien wynik, to stan spełniony lub niespełniony. Pierwszy zwraca zawsze wartość, natomiast drugi zwraca powód, dla którego został odrzucony. Sytuację przedstawia poniższy diagram:



Jeśli stan jest ustalony, w zależności od wyniku wykonuje się jedna z dwóch funkcji: *resolve* albo *reject*.

Korzystanie z promise'ów

W momencie, w którym korzystamy z promise'a, jesteśmy powiadamiani o jego spełnieniu bądź niespełnieniu:

```
promise
  .then(value => { /* spełniony */ })
  .catch(error => { /* odrzucony */ });
```

Spróbujmy stworzyć kilka przykładowych promise'ów:

Promise dla fs.readFile

Poniższa funkcja to odpowiednik `fs.readFile` z zastosowaniem promise'ów. Pełni rolę opakowania dla kodu, który jest asynchroniczny. Funkcja zwraca nową instancję `Promise`, która jako parametr przyjmuje kolejną funkcje z dwoma parametrami *resolve* i *reject*. W środku wywołujemy metodę `readFile` tak jak robiliśmy to do tej pory i zależności od tego, czy uda nam się odczytać plik, odrzucamy bądź spełniamy promise'a:

```
var fs = require('fs');

function readFilePromisified(filename) {
    return new Promise(
        (resolve, reject) => {
            fs.readFile(filename, { encoding: 'utf8' },
                (error, data) => {
                    if (error) {
                        reject(error);
                    } else {
                        resolve(data);
                    }
                });
        });
}
```

Co prawda jest tutaj więcej zagnieżdżeń niż w przypadku użycia samej metody `fs.readFile`, ale jest to jednorazowe opakowanie, które możemy wydzielić do osobnego modułu i korzystać z niego w dowolnym miejscu w kodzie.

Teraz odczyt pliku wygląda następująco:

```
readFilePromisified('plik-do-odczytu.txt')
  .then(text => console.log(text)) //tutaj 'wyrzucamy' zawartość plik
  .catch(error => console.log(error));
```

Skorzystanie z zapytania AJAX

Ten przykład jest analogiczny do poprzedniego, z jedną drobną różnicą - teraz odrzucenie promise'a następuje w dwóch przypadkach: kiedy wysłanie zapytania do serwera nie powiedzie się, lub kiedy dostaniemy odpowiedź oznaczającą błąd serwera (np. 404):

```
function httpGet(url) {
    return new Promise(
        function(resolve, reject) {
            const request = new XMLHttpRequest();
            request.onload = function() {
                if (this.status === 200) {
                    resolve(this.response); // Sukces
                } else {
                    reject(new Error(this.statusText)); // Dostaliśmy
                }
            };
            request.onerror = function() {
                reject(new Error(
                   `XMLHttpRequest Error: ${this.statusText}`));
            };
            request.open('GET', url);
            request.send();
        });
}
```

Wywołanie takiej funkcji wygląda następująco:

```
httpGet('http://example.com/file.txt')
  .then(response => console.log('Contents: ' + response))
  .catch(error => console.error('Something went wrong', error));
```

Opóźnienie, czyli opakowanie dla setTimeout

Możemy również opakować promise'a na `setTimeout` w celu opóźnienia wykonywa
funkcji:

```
function delay(ms) {
    return new Promise(function(resolve, reject) {
        setTimeout(resolve, ms);
    });
}
```

Tak wygląda wywołanie tej funkcji:

```
delay(5000)
  .then(() => console.log('5 seconds have passed!'));
```

Promise.all()

Przy okazji omawiania promise warto wspomnieć o dodatkowych metodach, które służą do obsługi bardziej zaawansowanych sytuacji. Wyobraźmy sobie przypadek, w którym potrzebujemy ściągnąć informacje z różnych serwisów www i kod, który ma się wykonać wtedy i tylko wtedy gdy mamy dostęp do wszystkich tych danych. Wykonanie tego zadania okazuje się bardzo proste, jeżeli użyjemy funkcji `.all()`

Zadaniem funkcji `.all()` jest zwrócenie rozwiązanego promise'a wtedy i tylko wtedy, kiedy wszystkie inne obietnice (podane w tablicy jako argument) zostaną rozwiązane. Metoda zwróci stan `reject` w przypadku gdy chociaż jeden z promise'ów będących argumentem nie zostanie rozwiązany (zostanie wykonana metoda reject).

Przykład:

```
function getDummyData1() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve({
                status: 'OK',
                data: {
                    message: 'Testowa wiadomość dummy1',
                },
            });
        }, 1000);
    });
}

function getDummyData2() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve({
                status: 'OK',
                data: {
                    message: 'Testowa wiadomość dummy2',
                },
            });
        }, 2000);
    });
}

Promise.all([getDummyData1(), getDummyData2()])

    .then((resp) => {
        console.log(resp);
        console.log(resp[0]);
        console.log(resp[1]);
    });
```

Co tak naprawdę dzieje się w powyższym kodzie?

Najpierw definiujemy wymyśloną metodę `getDummyData1()`, która zwróci rozwiązanego promise'a po 1s. Druga z funkcji, `getDummyData2()`, zwróci rozwiązanego promise'a z wymyślonymi danymi po okresie 2s. Zadaniem `Promise.all()` w tym przykładzie jest wykonanie kodu zawartego w `.then()` wtedy i tylko wtedy, gdy obie obietnice zostaną rozwiązane z sukcesem.

Promise.race()

O ile `Promise.all()` będzie czekać na zakończenie wszystkich Promisów, o tyle metoda `.race()` zwróci pierwszy, który zostanie zakończony. Korzystając z powyższego przykładu, będzie to wyglądać tak:

```
function getDummyData1() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve({
                status: 'OK',
                data: {
                    message: 'Testowa wiadomość dummy1',
                },
            });
        }, 1000);
    });
}

function getDummyData2() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve({
                status: 'OK',
                data: {
                    message: 'Testowa wiadomość dummy2',
                },
            });
        }, 2000);
    });
}

Promise.race([getDummyData1(), getDummyData2()])

    .then((resp) => {
        console.log(resp);
    });
```

Kod jest bardzo podobny, różni się tylko ostatni fragment, gdzie zamiast `.all()` użyliśmy teraz `.race()`. Wynikiem będzie wyświetlenie obiektu z funkcji `getDummyData1()`, ponieważ ten promise zostanie rozwiązany jako pierwszy.

Zadanie: Refaktoryzacja

Skorzystaj z promise'ów w zadaniu z Reactem. Zadanie polega na opakowaniu kodu pobierającego gifa w promise'a, a następnie wywołanie metody then w celu obsłużenia wyniku zapytania. Pamiętaj o dodaniu skryptu z polyfillem przed wszystkimi innymi skryptami. Link do skryptu możesz znaleźć w tym miejscu - https://cdnjs.com/libraries/babel-polyfill

Po wykonaniu zadania zaktualizuj swoje repozytorium z wyszukiwarką gifów o nowe zmiany i przekaż link do niego mentorowi.

Podgląd zadania

Wyślij link

15.6. Co nowego: organizacja

Do tej pory JavaScript nie posiadał natywnego rozwiązania, jeżeli chodzi o moduły, tak więc i w tym wypadku programiści musieli wymyślać swoje rozwiązania w postaci bibliotek. ES6 jest pierwszym standardem, który posiada w sobie wbudowane rozwiązanie modularyzacji kodu. Rozwiązania te bazują na dotychczasowych implementacjach i działają na podobnej zasadzie. Szczególnym wzorem dla modułów ES6 były moduły CommonJS, które omawialiśmy sobie przy okazji Node'a.

Eksport modułu

Istnieją dwa rodzaje eksportowania modułu, czyli po prostu udostępnienia fragmentu kodu tak, aby inne pliki mogły z niego korzystać - niebawem opiszemy to na przykładach:

Eksport z nazwą - dający możliwość eksportu wielu części modułu

W poniższym przykładzie za pomocą słówka kluczowego `export` dokonujemy eksportu kilku fragmentów naszego modułu obsługującego funkcje matematyczne.

```
//------ math.js ------
export const sqrt = Math.sqrt;
export function square(x) {
    return x * x;
}
export function diag(x, y) {
    return sqrt(square(x) + square(y));
}
```

Eksport z nazwą jest bardzo prostym zabiegiem. Kod piszemy tak, jak pisaliśmy do tej pory, tylko do eksportowanej części dodajemy słówko `export`.

Tutaj widzimy przykład użycia kodu z napisanego przez nas modułu. Importujemy dwie z trzech funkcjonalności i korzystamy z nich do obliczenia kwadratu liczby i przeciwprostokątnej trójkąta:

```
//------ main.js ------
import { square, diag } from 'math';
console.log(square(11)); // 121
console.log(diag(4, 3)); // 5
```

Jeśli chcemy zaimportować wszystkie części modułu, możemy użyć następującej notacji:

```
import * as math from 'math';
console.log(math.square(11)); // 121
console.log(math.diag(4, 3)); // 5
```

Możemy przeczytać to w następujący sposób - zaimportuj wszystko (symbol *) jako (as) math (nazwa obiektu, który będzie trzymał w sobie wszystkie importowane funkcjonalności z *math.js*)

Eksport domyślny - jeden eksport per moduł

Moduły, które eksportują tylko jedną wartość, mogą używać składni eksportu domyślnego. We front-endzie bardzo często używa się składni domyślnej do eksportu komponentów (np. w React). Spójrzmy na przykład:

```
//------ log.js ------
export default function () {} // no semicolon!

//------ main1.js ------
import log from 'log';
log();
```

Zauważ, że eksportowana domyślnie funkcja nie musi posiadać nazwy, ponieważ i tak jest tylko ona jedna na cały moduł.

Import modułu

Powyższe przykłady pokazują również jak zaimportować moduły, ale warto zwrócić uwagę na pewne rzeczy z nimi związane:

- import musi znajdować się zawsze na samej górze modułu
- jeśli importowany moduł importuje inne moduły itd., to ES6 automatycznie wie, że musi się do nich odnieść

Istnieje kilka sposobów na importowanie modułu:

- Import domyślny - `import math from 'src/math';`
- Import nazwany - `import { sqrt, diag } from 'src/math';`
- Import w przestrzeni nazw - `import * as math from 'src/math';`
- Zmiana nazwy importowanego modułu nazwanego - `import { sqrt as sq, diag } from 'src/math';`
- Zmiana nazwy importowanego modułu domyślnego - `import { default as m } from 'src/math';`

Można również łączyć różne metody importowania modułów np.:

```
import math, { diag as diagonal } from 'src/math';
```

UWAGA: import i eksport modułów nie będzie działał w przypadku przeglądarek. Dzieję się tak dlatego, że Babel transformuje *import/export* na odpowiedniki w stylu CommonJS, czyli zapisie modułów w NodeJS. Na szczęście problem importowania modułów rozwiązuje między innymi Webpack, o którym powiemy sobie jeszcze w następnych modułach.

Klasy

Obiektowość JavaScript bazuje na prototypach i do tej pory nie było klas, tylko funkcje konstruujące. W ES6 mechanizm działania jest taki sam, ale zmieniła się składnia. Przykładowa klasa wygląda teraz tak:

```
class Point {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
    toString() {
        return `(${this.x}, ${this.y})`;
    }
}
```

Jak widzisz, pojawiła się specjalna funkcja konstruująca, która jest uruchamiana przy tworzeniu nowej instancji danej klasy. Deklaracje metod także uległy zmianie. Metodę `toString` w ES5 przypisalibyśmy do prototypu klasy `Point` - tutaj jest ona przypisana bezpośrednio w klasie.

Używanie klas nie uległo w żaden sposób zmianie i wygląda dokładnie tak samo jak w ES5:

```
var p = new Point(1,2);
p.toString(); // “1, 2"
```

UWAGA: Zauważ, że między kolejnymi metodami klasy nie ma ani średnika ani przecinka. Jeśli wkradnie Ci się średnik, nic złego się nie stanie - zostanie on zignorowany przez kompilator, natomiast przecinek wyrzuci błąd składni.

Metody statyczne

Klasy mogą posiadać również metody statyczne, czyli takie, które nie wymagają instancji - taka metoda nie jest wywoływana w kontekście żadnego obiektu:

```
class Point {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
    toString() {
        return `(${this.x}, ${this.y})`;
    }
    static print(text) {
        console.log(text);
    }
}
```

Powyższa metoda statyczna `print` nie wymaga tworzenia nowego obiektu, aby mogła zostać użyta - wystarczy, że ją wywołamy:

```
Point.print('Hello world!');
```

Pobieranie i ustawianie właściwości za pomocą gettera i settera

Domyślnie wartość ustawiamy za pomocą znaku równości np. `point.x = 2`, a pobieramy odwołując się do klucza pod którym wartość się znajduje, np. `point.x // 2`.

Możemy jednak spotkać się z koniecznością zmodyfikowania domyślnego działania przypisania lub pobrania elementu, np. jeśli chcielibyśmy logować każdą pobieraną / ustawianą wartość `x`, moglibyśmy napisać następujący kod:

```
class Point {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    get x() {
        console.log(this.x);
        return this.x;
    }
    set x(value) {
        console.log('setting x', value);
    }
}
```

Subklasy

Czasami zdarza się, że chcemy stworzyć klasę w oparciu o inną klasę. Do tej pory wiązało się to ze skomplikowanym procesem wiązania prototypów i funkcji konstruujących. Przy użyciu ES6 wystarczy, że użyjemy słowa kluczowego `extends`. Dzięki niemu możemy stworzyć klasę, która zostanie stworzona w oparciu o istniejący konstruktor:

```
class ColorPoint extends Point {
    constructor(x, y, color) {
        super(x, y);
        this.color = color;
    }
    toString() {
        return super.toString() + ' in ' + this.color; // (B)
    }
}
```

Korzystamy tutaj z konstruktora klasy `Point`, dlatego używamy specjalnej funkcji wbudowanej o nazwie `super` (od łac. *super* - ponad) - w niej przekazujemy argumenty dla konstruktora klasy rozszerzanej. Funkcję tę wywołujemy tylko w konstruktorze! Oprócz tego, nadpisujemy przy tym domyślne zachowanie metody `toString`, które było zdefiniowane w klasie `Point`.

Zadanie: Stoper

W ramach zadania wykonamy sobie prosty stoper przy użyciu dotychczas poznanych funkcjonalności ES6. Na początku stwórzmy prosty szablon HTML z dwoma przyciskami i miejscem na timer:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Timer!</title>
    <link rel="stylesheet" type="text/css" href="styles.css">
  </head>
  <body>
    <nav class="controls">
      <a href="#" class="button" id="start">Start</a>
      <a href="#" class="button" id="stop">Stop</a>
    </nav>
    <div class="stopwatch"></div>
    <ul class="results"></ul>
    <script type="text/javascript" src="script-compiled.js"></script>
  </body>
</html>
```

Jak widzisz, stworzyliśmy sobie prostego HTML'a z dwoma przyciskami (start i stop), które będą wykonywały metody odpowiadające ich zadaniom. Obie będą należeć do obiektu klasy `Stopwatch`, którą stworzymy sobie na potrzeby naszego projektu. Zacznijmy od zdefiniowania klasy w pliku *script.js*. Plik ten będzie kompilowany za pomocą Babela do pliku s*cript-compiled.js*, który podpięliśmy w pliku *index.html*. Przejdźmy do tworzenia klasy `Stopwatch`

```
class Stopwatch {
    constructor(display) {
        this.running = false;
        this.display = display;
        this.reset();
        this.print(this.times);
    }
}

const stopwatch = new Stopwatch(
document.querySelector('.stopwatch'));
```

Mamy już podstawowy szablon klasy oraz jej instancję - musimy teraz zarejestrowa metody, które będą wykonywane po kliknięciu w odpowiednie przyciski:

```
let startButton = document.getElementById('start');
startButton.addEventListener('click', () => stopwatch.start());

let stopButton = document.getElementById('stop');
stopButton.addEventListener('click', () => stopwatch.stop());
```

Nasza klasa zawiera na razie tylko konstruktor, któremu przekazujemy jeden parametr - pole, w którym ma się pojawić nasz timer. Konstruktor zajmuje się również ustawianiem wartości początkowych niektórych fragmentów instancji takich jak: to czy stoper pracuje (`running`), przechowuje również element DOM, pod którym znajduje się stoper (`display`), resetuje licznik i drukuje czasy. W tym momencie wiemy, że musimy dokonać implementacji czterech metod: `start`, `stop`, `reset` i `print`. Najpierw zajmiemy się metodami, które wykonują się od razu po stworzeniu nowej instancji `Stopwatch`, czyli `reset` i `print`:

```
class Stopwatch {
    constructor(display) {
        this.running = false;
        this.display = display;
        this.reset();
        this.print(this.times);
    }

    reset() {
        this.times = {
            minutes: 0,
            seconds: 0,
            miliseconds: 0
        };
    }
}
```

Zauważ, że po metodzie `constructor` nie ma średnika ani przecinka. Od razu przechodzimy do nowej metody `reset`, która zeruje stoper. Jest to prosty obiekt zawierający minuty, sekundy i milisekundy. Kolejnym krokiem będzie implementacja metody `print` (zaraz po metodzie `reset`):

```
print() {
        this.display.innerText = this.format(this.times);
}
```

Metoda ta ustawia wewnętrzny tekst elementu DOM, który znajduje się pod atrybutem `display`. Dzieje się to przy użyciu kolejnej, specjalnej metody `format`, która będzie zajmowała się przygotowaniem tekstu do wyświetlenia:

```
format(times) {
        return `${pad0(times.minutes)}:${pad0(times.seconds)}:${pad0(
}
```

Metoda `format` zwraca szablon (ang. *template*), który wykorzystuje obiekt (`times`) podany do metody. Korzystamy w tym miejscu ze znajomej konstrukcji `${nazwa_zmiennej}`, która umożliwia nam przekazanie wyniku kolejnej funkcji (`pad0`) jako jeden z elementu szablonu, który ma wyglądać tak:

```
02:04:23 (2 min, 4 s, 10 ms)
```

Funkcja `pad0` ma za zadanie dodać zero do liczb jednocyfrowych. Spójrzmy na implementację tej funkcji:

```
function pad0(value) {
    let result = value.toString();
    if (result.length < 2) {
        result = '0' + result;
    }
    return result;
}
```

Funkcja `pad0` przyjmuje na wejście wartość liczbową, przekształca ją na stringa, a następnie sprawdza czy długość tego przekształcenia jest mniejsza od 2 dodając tym samym zero przed tę liczbę.

Jeśli wszystko zaimplementowaliśmy prawidłowo nasz stoper po uruchomieniu powinien wyglądać tak:

Dorzucając parę stylów, można poprawić wygląd aplikacji przykładowo w ten sposób:

Nasza aplikacja posiada już swój stan początkowy. Brakuje nam jedynie implementacji funkcji `start` i `stop`. Zacznijmy od metody `start`:

```
start() {
    if (!this.running) {
        this.running = true;
        this.watch = setInterval(() => this.step(), 10);

    }
}
```

Na samym początku działania stopera musimy sprawdzić czy przypadkiem nasz timer już nie chodzi (stąd flaga `running`). Jeśli stoper był zatrzymany należy go uruchomić ustawiając tym samym flagę `running` na `true`. Stoper działa w oparciu o interwał, który odpala co 10 ms metodę `step` (która jest po prostu kolejnym tikiem stopera). Funkcja `setInterval` przyjmuje jako pierwszy argument `callback`, stąd arrow function. Przyjrzyjmy się teraz metodzie `step`:

```
step() {
    if (!this.running) return;
    this.calculate();
    this.print();
}
```

Metoda ta sprawdza, czy nasz timer jest uruchomiony. Jeśli tak, należy metodą calculate przeliczyć odpowiednio minuty, sekundy i milisekundy, a następnie wydrukować wynik za pomocą metody `print`. Spójrzmy na metodę `calculate`:

```
calculate() {
    this.times.miliseconds += 1;
    if (this.times.miliseconds >= 100) {
        this.times.seconds += 1;
        this.times.miliseconds = 0;
    }
    if (this.times.seconds >= 60) {
        this.times.minutes += 1;
        this.times.seconds = 0;
    }
}
```

Metoda ta ma na celu odpowiednie zerowanie wartości milisekund i sekund, jeśli te przekroczą pewną wartość i odpowiednie zwiększanie sekund i minut. Zawiera ona jednak pewną sztuczkę. Ze względu to, że milisekund w sekundzie jest tysiąc, a nasz interwał wykonuje się co 10ms, należało podzielić 1000 przez 10 - stąd warunek `this.times.miliseconds >= 100`.

Nasz stoper można już uruchomić, ale nie można go jeszcze zatrzymać - brakuje nam implementacji metody `stop`:

```
stop() {
    this.running = false;
    clearInterval(this.watch);
}
```

Metoda ta zatrzymuje stoper ustawiając flagę `running` na `false`, a następnie czyści interwał, który kryje się pod atrybutem `watch`. Jeśli wrócisz do metody `start`, to zauważysz że przypisujemy referencję interwału do tego właśnie atrybutu.

Dodatkowym zadaniem dla chętnych będzie dopisanie metod, które będą:

- resetowała licznik,
- zaznaczała wynik i przekazywała go do listy czasów,
- resetowała listę czasów.

Dla wykonanych zadań dodatkowych stwórz też odpowiednie przyciski na stronie, któ
będą uruchamiały poszczególne funkcjonalności.

Po wykonaniu zadania, umieść swój kod w repozytorium na Githubie, a następnie przekaż link do rozwiązania mentorowi.

Podgląd zadania

Wyślij link

15.7. ES6 a React

React jako biblioteka bardzo dobrze dostosowuje się do najnowszych standardów i można go używać stosując elementy składni ES6. Co prawda nie możemy korzystać w środowisku przeglądarki z modułów bez webpacka, ale wrócimy do nich jeszcze w następnym module.

Klasy

Najbardziej widoczną zmianą przy pisaniu komponentów w stylu ES6 jest użycie składni `class`. Zamiast pisać `React.createClass` w celu zdefiniowania nowego komponentu, możemy stworzyć klasę, która będzie rozszerzeniem klasy `React.Component`. Spójrzmy na przykład:

```
class Image extends React.Component {
  render() {
    return <img alt={this.props.caption} src={this.props.src} />;
  }
}
```

Od razu można zauważyć różnice w składni. W przypadku ES6 jest ona zdecydowani bardziej zwięzła:

```
// Sposób tworzenia klasy przy użyciu starej składni
var Image = React.createClass({
  handleOnClick: function(e) { … },
  render: function() { … },
});

// ES6
class Image extends React.Component {
  handleOnClick(e) { … }
  render() { … }
}
```

Warto zwrócić uwagę na metody. Zauważ, że pozbyliśmy się dwukropków, słowa kluczowego `function` oraz przecinków po każdej z metod. Wszystkie metody związane z cyklem życia komponentu pozostały bez zmian, oprócz jednej! Rolę `componentWillMount` pełni teraz konstruktor, który jest wywoływany podczas tworzenia komponentu:

```
// Sposób tworzenia klasy przy użyciu starej składni ES5
var EmbedModal = React.createClass({
  componentWillMount: function() { … },
});

// ES6
class Image extends React.Component {
  constructor(props) {
    super(props);
    // Tutaj wykonywane są wszystkie działania typowe
    // dla metody componentWillMount
  }
}
```

Ustawianie początkowego stanu komponentu

Do tej pory początkowy stan komponentu ustawialiśmy za pomocą specjalnej metody `getInitialState`. Przy zastosowaniu składni ES6 sprawa jest znacznie prostsza - wystarczy, że w konstruktorze przypiszemy do zmiennej `this.state` obiekt reprezentujący początkowy stan naszej aplikacji.

```
// Sposób tworzenia klasy przy użyciu starej składni
var Image = React.createClass({
    getInitialState: function() {
        return {
            viewed: false
        }
    }
});

// ES6
class Image extends React.Component {
    constructor(props) {
        super(props);
        this.state = {
            viewed: false
        };
    }
}
```

Ustawianie początkowych wartości właściwości

Inicjalizacja props przy użyciu ES6 odbywa się za pomocą właściwości statycznych. Dlaczego propsy deklarujemy jako statyczne? Ta informacja jest niezależna od komponentu, ponieważ przychodzi od rodzica i nie potrzebuje instancji. Dzięki temu React przeprowadza walidację właściwości komponentu zanim zostanie stworzona jego instancja.

```
// Sposób tworzenia klasy przy użyciu starej składni
var Image = React.createClass({
  getDefaultProps: function() {
    return {
      width: 400,
      height: 320
    };
  },
  propTypes: {
    width: React.PropTypes.number.isRequired,
    height: React.PropTypes.number.isRequired,
  },
});

// ES6
class Image extends React.Component {
  static defaultProps = {
    width: 400,
    height: 320
  }

  static propTypes = {
    width: React.PropTypes.number.isRequired,
    height: React.PropTypes.number.isRequired,
  }
}
```

Wyraźną poprawą w składni jest brak konieczności tworzenia osobnej metody do ustawiania domyślnych właściwości.

Arrow functions

Stary sposób tworzenia komponentów zapewniał, że słowo `this` używane w metodach naszych komponentów zawsze wskazywało na swoją instancję (danego komponentu). Sytuacja jest nieco inna w przypadku komponentów tworzonych przy użyciu ES6. Tutaj React nie robi niczego za nas i musimy ręcznie zmieniać kontekst wywoływanej funkcji, aby `this` wskazywało na dany komponent. Wygląda to tak:

```
// Ręczna zmiana kontekstu za pomocą metody bind
class Image extends React.Component {
  constructor(props) {
    super(props);
    // Zmieniamy this metody handleOnClick...
    this.handleOnClick = this.handleOnClick.bind(this);
  }
  handleOnClick(e) {
    // ...aby this używane w tym miejscu wskazywało na komponent.
    this.setState({loading: true});
  }
}
```

Metoda ta ma niestety jedną dużą wadę - za każdym razem musimy zmieniać kontekst wywoływanej metody tak, aby `this` wskazywało na komponent.

```
this.handleOnClick = this.handleOnClick.bind(this);
```

Przeglądając repozytoria z kodem napisanym w Reakcie możesz się natknąć na takie rozwiązania, ale istnieje bardziej elegancka metoda w postaci... *arrow function* :)

```
class Image extends React.Component {
  handleOnClick = (e) => {
    this.setState( {loading: true} );
  }
}
```

Arrow function zachowuje kontekst kodu, który go otacza - oznacza to, że `this` wewnątrz funkcji wskaże na... nasz komponent!

Destrukturyzacja i atrybut spread

Czasami zdarza się sytuacja, w której przekazujemy wszystkie propsy z rodzica do dziecka, ale chcemy wykorzystać tylko wybrane z nich.

```
class Image extends React.Component {
  render() {
    const {
      className,
      ...others,  // zawiera wszystko co this.props oprócz className
    } = this.props;
    return <img className={className} src={this.props.src} />;
  }
}
```

W tym fragmencie kodu następuje destrukturyzacja obiektu `this.props` i wyciągane są z niego dwie właściwości: `className` i `others` (który jest tablicą zawierającą wszystko oprócz `className`).

Możemy również użyć atrybutu *spread* do rozprzestrzenienia właściwości na całym *ReactElement* zapisanym przy pomocy JSX:

```
class Image extends React.Component {
  render() {
    return <img {...this.props}/>;
  }
}
```

Zakładając, że obiekt `this.props` zawiera właściwości `className`, `width`, `height` i `src` powyższa sytuacja przekształciłaby się (za pomocą Babela) w taką składnię:

```
class Image extends React.Component {
  render() {
    return <img className={this.props.className} width={this.props.wi
  }
}
```

Zadanie: Przekształcenie stopera na składnię ES6

W ramach ćwiczenia, przekształć swój kod z zadania ze stoperem na React. Użyj do tego składni ES6. Pamiętaj, żeby nie korzystać jeszcze z modułów (*import*/*export*). Omówimy je w następnym module.

Po wykonaniu zadania, zaktualizuj swoje poprzednie repozytorium i przekaż link do rozwiązania mentorowi.

Podgląd zadania

Wyślij link

15.8. Zmiany, zmiany, zmiany...

Niedawno został ogłoszony kolejny standard, czyli ES7 (lub też ES2016). Od tej chwili TC39 zamierza wypuszczać kolejne, mniejsze wersje standardu co rok. Nie omawiałem go, ponieważ tego standardu nie rozpatruje jeszcze prawie żadna przeglądarka. Warto jednak chociaż przez chwilę zastanowić się co nas czeka w przyszłości.

Dekoratory

Dekoratory są używane w wielu językach programowania i (jak sama nazwa wskazuje) służą do dekorowania, czyli dodawania funkcjonalności pojedynczym metodom czy też klasom. Jest to polepszenie składni dla tzw. funkcji wyższego rzędu (ang. *higher-order function*).

Programowanie funkcyjne

Pojawiło się kolejne pojęcie z programowania funkcyjnego. Funkcja wyższego rzędu, według Wikipedii, jest to taka funkcja, która spełnia chociaż jeden z tych warunków:

- Jako parametr przyjmuje jedną lub więcej funkcji
- Zwraca funkcję

Korzystaliśmy już z tej techniki przy wywołaniu zwrotnym (*callback*). W przypadku wszystkich zdarzeń, które zachodzą w przeglądarce lub w innym środowisku rejestrujemy funkcje, które mają się wykonać na pewne zdarzenie przekazując je jako parametry.

Korzyści płynące z dekoratorów to między innymi:

- wydobywanie pewnych wspólnych fragmentów kodu jako mniejsze fragmenty i składanie ich w razie potrzeby
- dekorowanie funkcji w celu rozszerzenia ich funkcjonalności
- pisanie funkcji, które tworzą inne dynamicznie
- kompozycja (kolejna fraza mocno związana z programowaniem funkcyjnym) serii funkcji w celu wygenerowania pewnego zachowania

Wracając do dekoratorów - przykładowy dekorator, z którym możesz spotkać się w wielu językach programowania (m. in. Python, Java) wygląda następująco:

```
@autobind
handleOnClick() {}
```

Powyższy dekorator jest przykładem bindowania (ustawiania kontekstu) metody w klasie. Do tej pory poznaliśmy już trzy metody zmiany kontekstu:

- `var self = this`
- *arrow function*
- metoda `bind`

Dekorator jest po prostu kolejną składnią, która ma spełniać to samo zadanie co powyższe metody.

Dekoratory można pisać jeden pod drugim, co jest przykładem kompozycji.

```
@log
@autobind
handleOnClick() {}
```

Powyższy kod już jest tylko koncepcją (chociaż może istnieć taki dekorator). Tak więc przykładowo, jeśli chcielibyśmy logować wywołania konkretnej metody, moglibyśmy napisać dekorator, który wystarczyłoby dodać nad odpowiednią metodą. Nie trzeba wchodzić do implementacji metody, aby dodać funkcjonalność logowania. Wystarczy dopisać dekorator.

Funkcje async/await

Do tej pory poznaliśmy dwie składnie wyrażania asynchronicznego kodu. Prymitywne *callbacki*, które, jak już zauważyliśmy, mają wiele wad oraz *promise'y*, które znacząco poprawiają czytelność i zrozumiałość naszego kodu.

Mimo zastosowania promise'ów, myślenie w sposób asynchroniczny wciąż nie jest prostym zadaniem i można się w nim nieco pogubić. Ten problem stara się właśnie rozwiązać *async/await*, które pozwalają na pisanie kodu asynchronicznego w sposób synchroniczny. Do tej pory przykładowy kod asynchroniczny wyglądał tak:

```
readFilePromisified('plik-do-odczytu.txt')
.then(text => console.log(text))
.catch(error => console.log(error));
```

Korzystając ze składni synchronicznej możemy zaznaczyć, że nasza funkcja ma działać asynchronicznie (*async*), a następnie zaczekać (*await*) na wykonanie pewnego działania:

```
async function readFile() {
    var text = await readFilePromisified();
    console.log(text);
}
```

Implementacja *async/await* opiera się dalej na promise'ach i ma na celu tylko poprawę czytelności kodu.

Operator **

Powyższy operator jest kolejnym uproszczeniem składni i służy do wyrażania potęgi. Do tej pory byliśmy zmuszeni do korzystania z metody `Math.pow(2, 3)`, która to dla tego przykładu zwracała 8 (2 do potęgi 3). ES2016 pozwala na zapis `2 ** 3`, który zwróci taki sam wynik.

Już za rogiem czai się kolejna wersja specyfikacji (ES8) i jak już wspomniałem, kolej wersje będą teraz wypuszczane z roku na rok. JavaScript to język przeglądarki i nic n wskazuje na to, żeby miało się to zmienić. Do kolejnego etapu tego języka jest jeszcze

daleko, ale warto być zawsze o jeden krok do przodu i interesować się takimi nowinkami. Praca programisty, tak samo jak specjalisty z każdej innej dziedziny, to ciągły rozwój. Nie przejmuj się, że w świecie JS'a tyle się dzieje i nie przywiązuj się do narzędzi.

Warto wyrobić w sobie umiejętność przystosowania się do zachodzących zmian i bardzo dobrze znać fundamenty na których wszystkie te rzeczy się opierają. Jak już wspomniałem, front-endowa triada (HTML, CSS, JS) ugruntowała sobie dobrą pozycję i znajomość tych trzech języków to prawdziwa miara dobrego front-end developera.

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

Zapoznałe(a)m się!

15.9. PROJEKT: Wyszukiwarka użytkowników na Githubie

W tym submodule zbudujemy sobie mały projekt korzystając z publicznego API Githuba. Dzięki niemu możemy zbudować własnego klienta bez zastanawiania się w jaki sposób działa część serwerowa aplikacji. Skorzystamy z endpointa umożliwiającego wyszukiwanie użytkowników. Do dzieła!

Zacznijmy od stworzenia katalogu projektu, a następnie w środku stwórzmy plik `.babelrc` z następującą treścią:

```
{
  "presets": ["es2015", "react"]
}
```

Kolejnym krokiem będzie zainicjalizowanie nowego projektu przy użyciu komendy npm i zainstalowanie niezbędnych paczek. Zacznijmy od zainicjalizowania projektu za pomocą `npm init`, a później zainstalujmy potrzebne pazki komendą `npm install --save babel-cli babel-preset-es2015 babel-preset-react`.

Następnie w wygenerowanym pliku *package.json* dodajmy nowy skrypt:

```
"build": "babel script.js --watch --out-file build.js"
```

Od tej pory, gdy uruchomimy komendę `npm run build`, Babel zacznie obserwować plik *script.js* i skompiluje go jak tylko pojawią się w nim jakiekolwiek zmiany. Zanim jednak zaczniemy pisać cokolwiek w pliku *script.js*, zacznijmy od stworzenia pliku *index.html*. Powinien się w nim zawierać następujący kod:

```
<!doctype html>
<html lang="pl">
<head>
    <title>Document</title>
</head>
<body>
    <div id="root"></div>
    <script src="https://unpkg.com/react@15/dist/react.min.js"></scri
    <script src="https://unpkg.com/react-dom@15/dist/react-dom.min.js
    <script src="./build.js"></script>
</body>
</html>
```

Element `<div>` z atrybutem `id` ustawionym na `root` jest punktem, w którym React zamontuje naszą aplikację. Skrypty w kolejnych linijkach to `react` i `react-dom` niezbędne do działania aplikacji z pliku *build.js*.

Stwórzmy teraz plik *script.js* i rozpocznijmy pisanie właściwej aplikacji. Zacznijmy od kodu, który zamontuje główny komponent `app` pod element o id `root`:

```
ReactDOM.render(
  <App />,
  document.getElementById('root')
);
```

Głównym komponentem aplikacji jest komponent `App`. Poniżej znajduje się jego kod:

```
class App extends React.Component {
  constructor() {
    super();
    this.state = {
      searchText: '',
      users: []
    };
  }

  onChangeHandle(event) {
    this.setState({searchText: event.target.value});
  }

  onSubmit(event) {
    event.preventDefault();
    const {searchText} = this.state;
    const url = `https://api.github.com/search/users?q=${searchText}`
    fetch(url)
      .then(response => response.json())
      .then(responseJson => this.setState({users: responseJson.items}
  }

  render() {
    return (
      <div>
        <form onSubmit={event => this.onSubmit(event)}>
          <label htmlFor="searchText">Search by user name</label>
          <input
            type="text"
            id="searchText"
            onChange={event => this.onChangeHandle(event)}
            value={this.state.searchText}/>
        </form>
        <UsersList users={this.state.users}/>
      </div>
    );
  }
}
```

Zacznijmy analizę kodu od metody `render`. Składa się ona z elementów `<form>` oraz `UsersList`. Sam formularz składa się z jednego pola, którego wartość jest wyszukiwaną frazą, po której chcemy odnaleźć użytkownika. Po zatwierdzeniu formularza (wciśnięciu klawisza enter) przez użytkownika, chcemy wysłać do API Github zapytanie, a zwrócone dane wyświetlić w komponencie `UsersList`.

W tym celu w konstruktorze inicjujemy stan komponentu:

```
this.state = {
      searchText: '',
      users: []
};
```

Część stanu `searchText` będzie wykorzystywana w adresie URL, który znajduje się w metodzie `onSubmit`. Zanim jednak do tego przejdziemy spójrzmy na metodę `onChangeHandle`:

```
onChangeHandle(event) {
    this.setState({searchText: event.target.value});
}
```

Ta krótka metoda ma za zadanie zmienić stan `searchText` na taki, jaki kryje się pod zdarzeniem zmiany inputa (`event.target.value`).

W metodzie onSubmit wykorzystujemy wiedzę na temat promise'ów. Po skonsturowaniu adresu URL dzięki szablonom ES6 (`` `https://api.github.com/search/users?q=${searchText}` ``), wywołujemy funkcję `fetch`, która zwraca Promise.

Funkcja `fetch` jest interfejsem dzięki któremu możemy pobierać różne zasoby z sieci. Dla osób, które korzystały wcześniej z obiektu XHR, bądź innego sposobu na wykorzystanie technologii AJAX, ten sposób komunikacji powinien być znany. Fetch jest po prostu natywną implementacją (co prawda nie wspieraną jeszcze przez wszystkie przeglądarki) zapytań ajaxowych korzystającą z promisów. Nie musimy więc martwić się o tworzenie obietnic opakowujących zapytanie XHR. Fetch dostarcza wszystkiego czego trzeba. Więcej na jego temat można przeczytać w tym miejscu!

Kiedy `fetch` dostanie odpowiedź z serwera (obietnica zostanie spełniona), do pierwszego `then` trafia obiekt typu `Response`, który musimy odpowiednio przekształcić na obiekt JSON (stąd metoda `response => response.json()`). Wykorzystujemy tu również *arrow function*, aby uprościć zapis callbacka. Po tym przekształceniu ustawiamy stan `users` na tablicę `items` znajdującą się w odpowiedzi od API Githuba.

Ten stan prowadzi nas do kolejnego komponentu - `UsersList`, który jako parametr przyjmuje tablicę userów i zajmuje się odpowiednim wyświetleniem każdego jednego użytkownika zwróconego z serwera:

```
class UsersList extends React.Component {
  get users() {
    return this.props.users.map(user => <User key={user.id} user={use
  }

  render() {
    return (
      <div>
        {this.users}
      </div>
    );
  }
}
```

Jest to bardzo prosty komponent, którego wyróżniającym się elementem jest *getter* `users`. W nim dokonujemy przekształcenia tablicy, którą otrzymujemy z komponentu `App` mapując każdy jej element na komponent `User` z propsami `key` oraz `user`. Wyświetleniem całej listy zajmie się oczywiście metoda `render`.

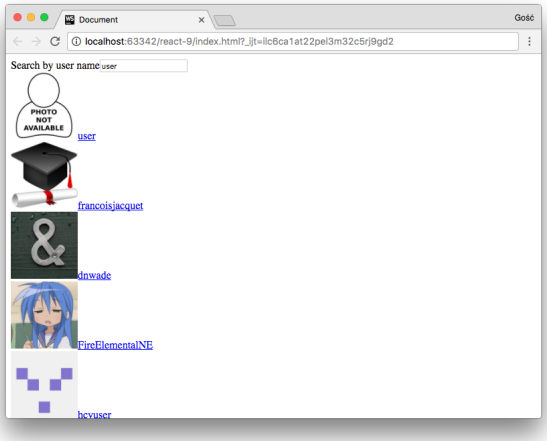
Ostatnim elementem aplikacji jest stworzenie komponentu odpowiedzialnego za wyświetlanie pojedynczego użytkownika:

```
class User extends React.Component {
  render() {
    return (
      <div>
        <img src={this.props.user.avatar_url} style={{maxWidth: '100p
        <a href={this.props.user.html_url} target="_blank">{this.prop
      </div>
    );
  }
}
```

W odebranych właściwościach znajduje się pojedynczy obiekt użytkownika (`this.props.user`), a w nim:

- adres URL avatara, który chcemy przekazać do elementu `<img>`,
- adres URL profilu, który podpinamy do elementu `<a>`,
- nazwa usera, którą wyświetlamy w podlinkowaniu

Po otworzeniu pliku *index.html*, cała aplikacja powinna prezentować się mniej-więcej tak:



Zadanie: Wyszukiwarka użytkowników

Po wykonaniu poleceń z tego submodułu, ostyluj swoją aplikację, a następnie umieść kod w repozytorium na Githubie i przekaż link do niego swojemu mentorowi.

Podgląd zadania

Wyślij link