

12. Ajax i API



Wyzwania:

- Dowiesz się, czym jest Ajax, oraz dlaczego powstała ta technologia.
- Nauczysz się, jak pobrać dane z serwera za pomocą JavaScriptu.
- Zdobędziesz wiedzę na temat API, dowiesz się też, jaki związek ma API z Ajaxem.
- Zrobisz aplikację do wyświetlania podglądu danych otrzymywanych z serwera.

Ocena kursu

Stale podnosimy jakość naszych szkoleń. W związku z tym bylibyśmy wdzięczni za wypełnienie krótkiej ankiety. Zajmie Ci ona dosłownie chwilę.



Ocena bootcampa WebDeveloper

*Wymagane

W której edycji bierzesz udział? *

Wybierz ▼

Jaka jest Twoja dotychczasowa ocena bootcampa? *

1 2 3 4 5 6 7 8 9 10

Mega słaby ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ Świetny!

Twoja opinia na temat kursu (opcjonalnie)

Twoja odpowiedź

Imię i nazwisko/lub mail jakim logujesz się do bootcampa *
*

Twoja odpowiedź

PRZEŚLIJ

Nigdy nie podawaj w Formularzach Google swoich haseł.

Formularze Google

Ten formularz został utworzony w domenie Codemy S.A..



Dziękujemy!



12.1. Protokół HTTP



Zanim zaczniemy poznawać techniki Ajaksa, warto jest wyjaśnić sobie, w jaki sposób działa internet. Jest to szczególnie ważne dla kogoś, kto pracuje z nim na co dzień. Zatem co tak naprawdę dzieje się w momencie wpisania adresu w górnym pasku przeglądarki aż do wyświetlenia właściwej strony?

Internet

Internet najprościej wyobrazić sobie w formie kabla, do którego podłączone są inne komputery. Oczywiście jest to bardzo, ale to bardzo, duże uproszczenie i jeżeli wiesz coś na temat sieci komputerowych, możesz uznać je za śmieszne. Chodzi o to, że internet łączy ze sobą dwa lub więcej komputerów.



Część z nich to tzw. *serwery*, które zawierają specjalne oprogramowanie do świadczenia pewnych usług innym komputerom podłączonym do sieci.

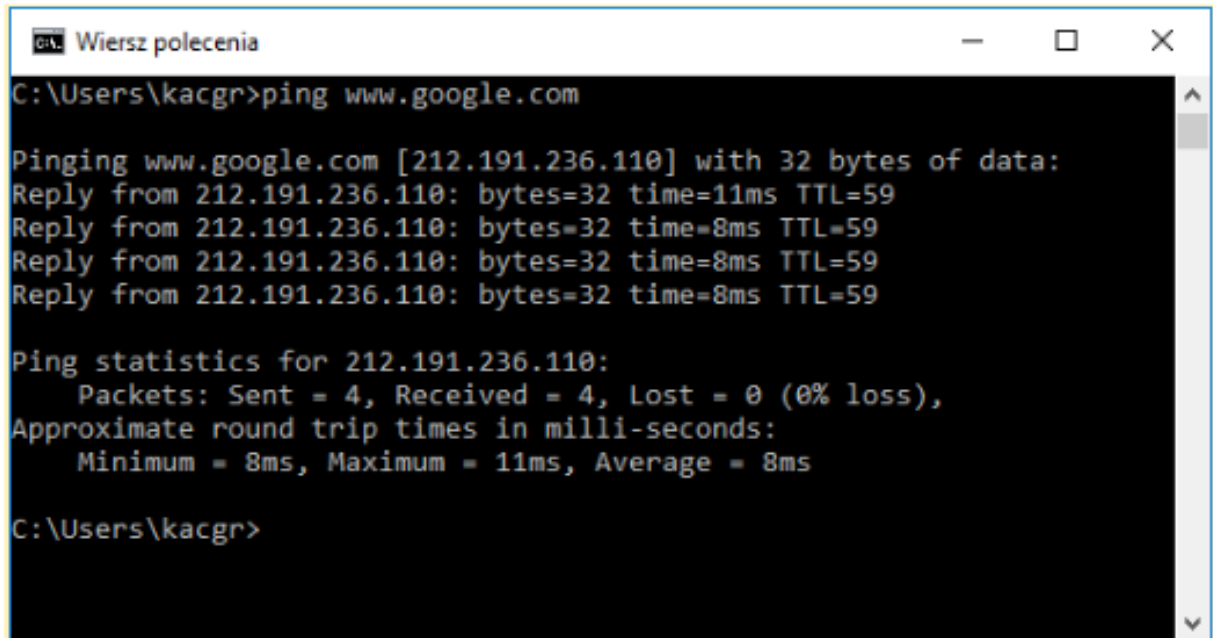
Oprogramowanie, które korzysta z takich usług, to klient. Przykładem klienta może być przeglądarka lub klient poczty. Dlaczego Twój komputer nie jest serwerem? Z prostej przyczyny. Jeśli na komputerze nie ma zainstalowanego żadnego oprogramowania serwerowego, to komputer "nie potrafi" po prostu odpowiadać na zapytania, czyli nie może pełnić roli serwera.

IP



Zapewne zdarzyło Ci się słyszeć o adresach IP. Przykładowy adres wygląda następująco: **212.191.236.110**. Jest to jeden z wielu adresów IP serwerów Google'a. Jeśli wpiszesz go w przeglądarkę, to Twoim oczom ukaże się znajoma witryna.

Skąd można się dowiedzieć, jaki adres IP ma strona Google? Można to w prosty sposób sprawdzić za pomocą komendy ping w wierszu poleceń/terminalu. Polecenie służy do sprawdzenia, czy komputer, z którym próbujemy się połączyć, jest osiągalny, tzn. czy możemy się z nim skomunikować:



```
C:\Users\kacgr>ping www.google.com

Pinging www.google.com [212.191.236.110] with 32 bytes of data:
Reply from 212.191.236.110: bytes=32 time=11ms TTL=59
Reply from 212.191.236.110: bytes=32 time=8ms TTL=59
Reply from 212.191.236.110: bytes=32 time=8ms TTL=59
Reply from 212.191.236.110: bytes=32 time=8ms TTL=59

Ping statistics for 212.191.236.110:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 8ms, Maximum = 11ms, Average = 8ms

C:\Users\kacgr>
```

IP (ang. *Internet Protocol*) to coś więcej niż tylko adresy. Protokół internetowy to zbiór ścisłych reguł, które są wykonywane za naszymi plecami w celu nawiązania łączności, a jednym z elementów protokołu jest właśnie adres IP.

Zapytanie/Odpowiedź

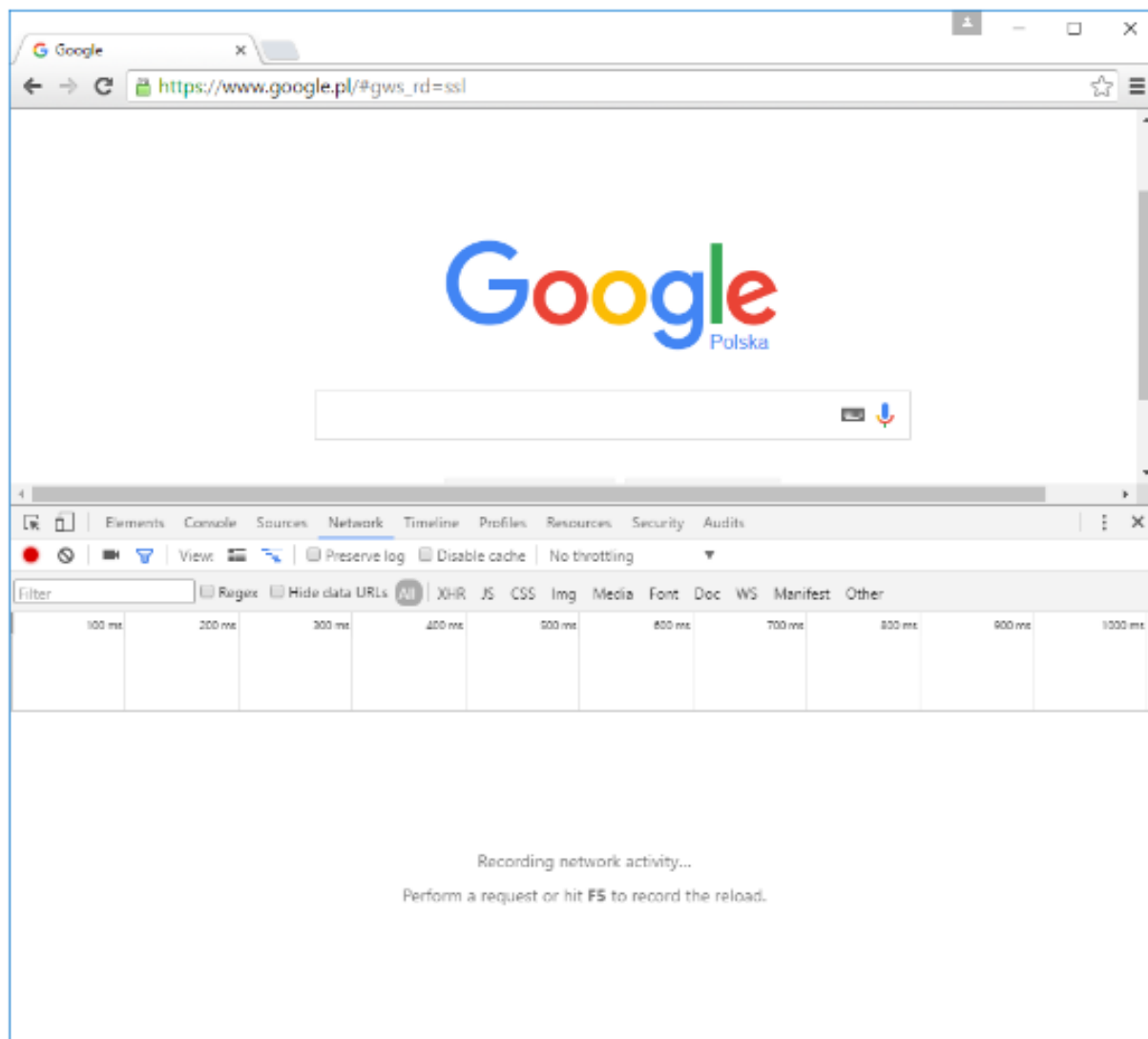
Weźmy jeszcze raz adres Google'a. Tym razem nie będzie to niezrozumiałe 212.191.236.110, lecz łatwa do zapamiętania nazwa: **www.google.com**. Po wpisaniu adresu w przeglądarkę klikamy enter. W tym momencie nasz komputer wysyła zapytanie do ISP (ang. Internet Service Provider), czyli naszego dostawcy internetu.

Nasz dostawca przesyła wpisany przez nas adres (www.google.com) do tzw. DNSa (ang. Domain Name System). Zadaniem DNSa jest przetłumaczenie adresu czytelnego dla użytkownika na adres czytelny dla urządzeń w sieci. W tym przypadku zamienia www.google.com na 212.191.236.110.



Jak to widzi przeglądarka?

Ok, wiemy już co nieco na temat tego, co dzieje się za naszymi plecami, jeśli chodzi o całe połączenia. Zobaczymy teraz, jak widzi to przeglądarka. W tym celu otworzymy sobie narzędzia deweloperskie Chrome i przejdźmy do zakładki Network.



Zanim przejdziemy do opisu okienka, odświeżmy okno przeglądarki, tak jak podpowiada nam Chrome. Rozpocznijemy tym samym nagrywanie aktywności sieciowej.

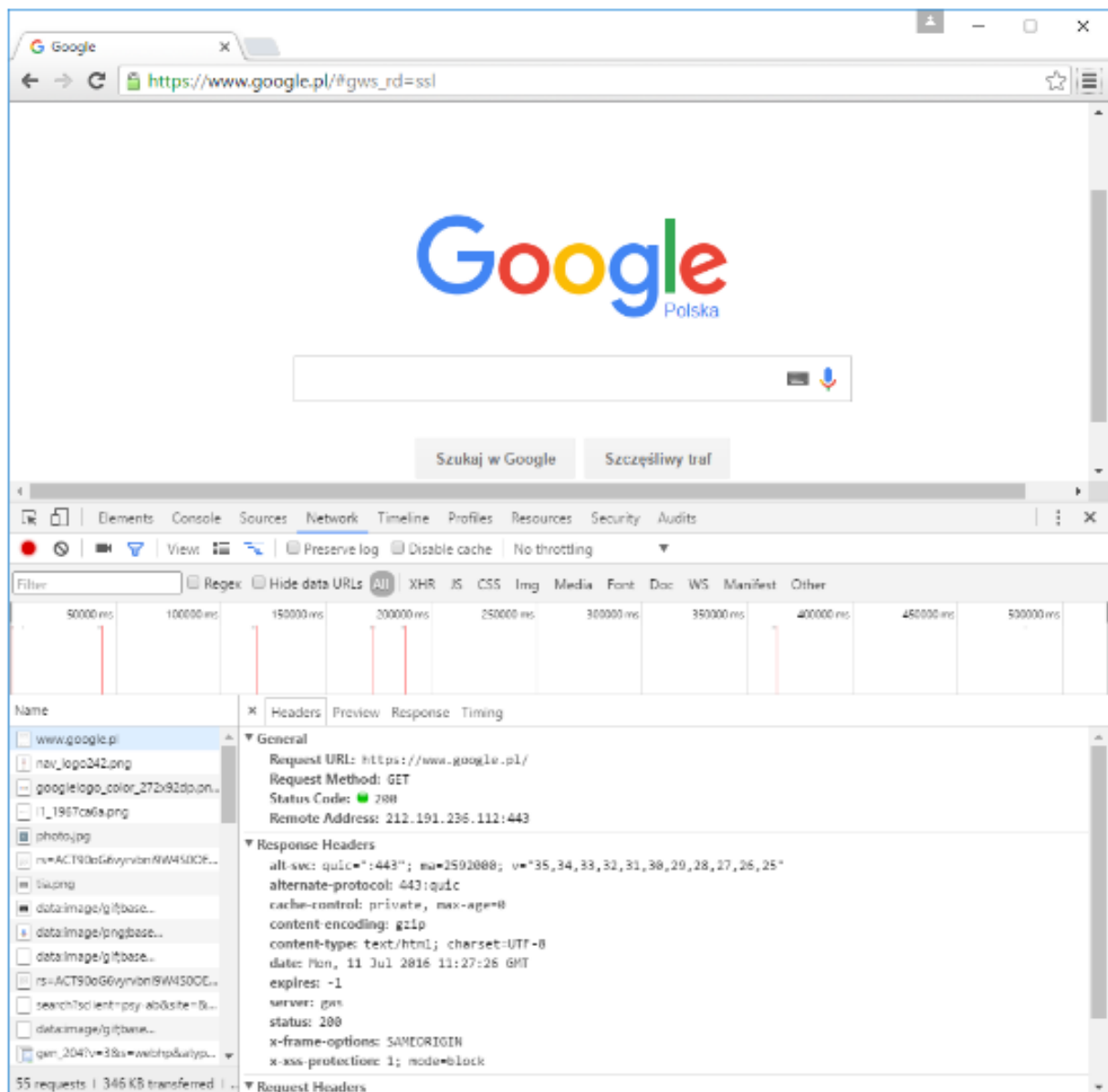
Po odświeżeniu strony przewińmy widok okna narzędzi deweloperskich na samą górę (za pomocą scrolla). Powinniśmy zobaczyć mniej więcej coś takiego:

The screenshot shows a Google Chrome browser window with the URL `https://www.google.pl/#gws_rd=ssl`. The page displays the Google Polska logo and a search bar. The Chrome DevTools Network tab is open, showing a list of network requests. The first request, `www.google.pl`, is selected. The table below represents the data shown in the Network tab.

Name	Status	Type	Initiator	Size	Time	Timeline - Start Time	3.00s	4.00s	5.00s
www.google.pl	200	document	Other	60.5 KB	259 ms				
nav_logo242.png	200	png	www.google.pl/032	(from cac..)	0 ms				
googlelogo_color_272x92dp.png	200	png	www.google.pl/032	(from cac..)	0 ms				
rl_1967c6fa.png	200	png	www.google.pl/032	(from cac..)	1 ms				
photo.jpg	200	png	www.google.pl/032	(from cac..)	0 ms				
rs=ACT90dG6yrvn8W450CERFs...	200	script	[index]:52	(from cac..)	5 ms				
5aapng	200	png	rs=ACT90dG6yrvn8W450CERFs...	(from cac..)	1 ms				
data:image/gif;base64...	200	gif	rs=ACT90dG6yrvn8W450CERFs...	(from cac..)	0 ms				
data:image/png;base64...	200	png	rs=ACT90dG6yrvn8W450CERFs...	(from cac..)	0 ms				
data:image/gif;base64...	200	gif	rs=ACT90dG6yrvn8W450CERFs...	(from cac..)	0 ms				
rs=ACT90dG6yrvn8W450CERFs...	200	script	rs=ACT90dG6yrvn8W450CERFs...	(from cac..)	2 ms				
search?client=psy-ab&site=830a...	200	xhr	rs=ACT90dG6yrvn8W450CERFs...	304 B	66 ms				
data:image/gif;base64...	200	gif	[index]:260	(from cac..)	0 ms				
gem_2047v=33s=awolpka/yprcs...	204	text/html	Other	18 B	50 ms				

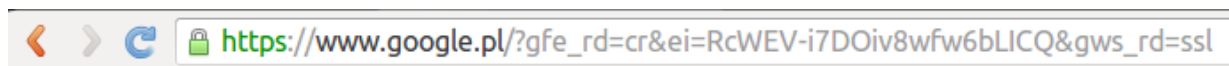
Summary: 23 requests | 61.0 KB transferred | Finish: 5.67 s | DOMContentLoaded: 367 ms | Load: 403 ms

Na samej górze tabeli w pierwszej kolumnie widzimy nasze zapytanie: `www.google.pl`. Kliknijmy w to zapytanie i przejdźmy do zakładki Headers.



Protokół HTTP

Czyli *Hypertext Transfer Protocol*. Używamy go na co dzień przy okazji korzystania z internetu. Jak zapewne bardzo dobrze wiesz, przeglądarka dodaje przed adresem strony tajemnicze `http://` lub `https://`



Wróćmy na chwilę do `https`. Czym jest ten dodatek "s" na końcu `http`? Literka "s" to skrót od *Secure*. `Https` jest szyfrowaną wersją protokołu `http`. Szyfrowanie informacji stosuje się zapobiegawczo w celu uniknięcia przechwycenia i zamiany przesyłanych danych.





Nagłówki HTTP

Dzięki nagłówkom klient albo serwer mają możliwość wysyłania dodatkowych informacji razem z zapytaniem lub z odpowiedzią. Typowa składnia nagłówka to:

`"header name : header value"`

W zakładce General znajdują się informacje ogólne na temat nawiązanego połączenia:

1. **Request URL** — wskazuje na adres, jaki został użyty w celu nawiązania połączenia.
2. **Request Method** — informuje na temat użytej metody HTTP (będzie o nich jeszcze mowa w następnym podrozdziale). GET to metoda, która służy do pobierania zasobów.
3. **Status Code** — służy do informowania o tym, czy zapytanie się powiodło, czy też nie. Więcej o kodach statusu powiemy sobie troszkę później. 200 oznacza, że z połączeniem jest wszystko OK.
4. **Remote Address** — adres, za pomocą którego przeglądarka nawiązuje połączenie. To jest właśnie adres po tłumaczeniu przez serwer DNS (patrz podrozdział Zapytanie/Odpowiedź powyżej).



Nagłówki można podzielić na nagłówki zapytania i odpowiedzi. Nie będziemy omawiać całej listy nagłówków, ale warto zwrócić szczególną uwagę na:

- **Content-Type** – to dzięki niemu przeglądarka wie, co zrobić z danym plikiem. W naszym przykładzie wartość tego nagłówka to `text/html; charset=UTF-8`. Dzięki temu plik zostanie potraktowany jako tekst html o kodowaniu utf-8.
- **Set-Cookie** – ten nagłówek jest zwracany w odpowiedzi serwera i ustawia w naszej przeglądarce tzw. ciasteczko, czyli informacje o sesji klienta z serwerem. Może służyć np. do zapamiętania stanu koszyka na stronie. Po więcej informacji na temat ciasteczek zajrzyj [tutaj](#).
- **Cookie** – dzięki niemu odsyłamy do serwera informacje o stanie naszej sesji.
- **User-Agent** – nagłówek, który zawiera informacje na temat klienta, za pomocą którego zostało wykonane zapytanie. Na podstawie screena można odczytać, że zapytanie zostało wykonane z przeglądarki Google Chrome v.51.0 zainstalowanej na systemie Win 10 w wersji 64-bitowej.

Metody HTTP

Zajmiemy się teraz popularnymi metodami protokołu HTTP i powiemy sobie, do czego te metody służą.

- **GET** – jest to najprostsza z metod. Jak już wspominaliśmy, służy ona do pobierania zasobów z serwera. Przykładowe zasoby to: pliki html, css, js, a także obrazki w różnych formatach. Dokładny opis treści, którą odsyła nam serwer, jest przechowywany w nagłówku Content-Type.
- **POST** – ta metoda służy do wysyłania danych do serwera. Dane te mogą być zapisane w formacie pary "klucz: wartość" lub w postaci binarnej, której używa się do przesłania plików (np. załączników w postaci zdjęć). Z metody tej korzystamy, kiedy chcemy zatwierdzić formularz lub wysłać jakieś duże pliki, takie jak zdjęcia, filmiki, pliki tekstowe.
- **PUT** – jest metodą, która działa bardzo podobnie do metody POST. Ograniczeniem tej metody jest wysyłanie tylko jednej porcji danych. Za jej pomocą nie można wysłać całego formularza jako zbioru danych "klucz: para". Można wysłać tylko jedną porcję. W przypadku formularza byłoby to jedno pole. W praktyce metoda PUT używana jest najczęściej do aktualizowania danych już istniejących.
- **DELETE** – jak się pewnie domyślasz, służy do usuwania z serwera danych, które zostały wskazane przez zapytanie.
- **HEAD** – ta metoda działa bardzo podobnie do metody GET z tą różnicą, że zwracane są jedynie metadane o zasobie. Działa ona troszkę jak element `<head>` w HTML. Dane te kryją się oczywiście pod postacią nagłówków zapytania.



Kodowanie odpowiedzi HTTP

Ostatnim tematem, który należy omówić w kontekście protokołu HTTP, są kody odpowiedzi, nazywane też statusami. Są prezentowane jako trzycyfrowe wartości i dzięki nim wiemy, jaki jest status odpowiedzi serwera. Spójrzmy na typowe oznaczenia tych kodów:

- 1xx — rzadko spotykane kody informacyjne.
- 2xx — kod, który oznacza, że zapytanie klienta zostało poprawnie odebrane, zrozumiane i zaakceptowane. Przykłady:
 - 200 — zapytanie się powiodło, a odpowiedź na zapytanie jest zależna od metody, której użyto do wysłania zapytania,
 - 201 — symbolizuje nowo utworzony zasób na serwerze. Serwer może odpowiedzieć za pomocą właśnie tego kodu np. po dodaniu komentarza.
- 3xx — kod przekierowania. Oznacza, że klient musi podjąć pewne akcje, aby dokończyć zapytanie. Po tym zapytaniu kolejne musi być HEAD albo GET w celu pobrania informacji o zasobie albo pobrania innego zasobu.
- 4xx — błąd spowodowany działaniami użytkownika. Najpopularniejszym kodem z tego zakresu jest błąd 404, który pojawia się, kiedy chcemy się odnieść do miejsca w sieci, które nie istnieje. Innymi przykładami mogą być:
 - 400 — serwer nie potrafi zrozumieć zapytania,
 - 401 — nieautoryzowane zapytanie. Pojawia się, gdy pominiemy odpowiednie nagłówki autoryzacji lub podane informacje są nieprawidłowe,
 - 403 — serwer zrozumiał zapytanie, ale nie zgadza się na wysłanie odpowiedzi.
- 5xx — błędy serwera. Pojawiają się, kiedy serwer nie potrafi przetworzyć informacji albo coś poszło nie tak po jego stronie. Przykładowe błędy:
 - 500 — wewnętrzny błąd serwera uniemożliwiający mu spełnienie zapytania,
 - 501 — funkcjonalność potrzebna do spełnienia zapytania nie jest jeszcze zaimplementowana,
 - 503 — serwis nie jest w stanie w tej chwili obsłużyć zapytania.





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

✓ Zapoznałem(a)m się!

12.2. Czym jest Ajax?



Nie mamy tutaj na myśli znanej marki środków czyszczących, ani klubu piłkarskiego. Ajax to akronim od Asynchronous JavaScript and XML, co w języku polskim oznacza: Asynchroniczny JavaScript i XML. Jest to technologia, za pomocą której tworzymy aplikacje internetowe. Dzięki niej interakcja klienta z serwerem odbywa się bez przeładowania całej strony.

Przykłady zastosowania z życia codziennego

Zastanówmy się nad przykładami z życia codziennego. Wcale nie trzeba daleko szukać z tej techniki korzysta np. czat znajdujący się na stronie [Kodilli](#). Inne przykłady:



- **Google:** podpowiadanie treści zapytania w trakcie wpisywania, doładowywanie zdjęć podczas przewijania, stronicowanie wyników wyszukiwań, cały klient poczty Gmail,
- **Facebook:** powiadomienia, dynamiczne dodawanie na tablicy, Messenger.

W jaki sposób działa Ajax?

Do tej pory kiedy wysyłaliśmy zapytania do serwera, nasza strona się przeładowywała.

Ajax działa w oparciu o obiekt XMLHttpRequest, którego działanie jest zaimplementowane w środowisku przeglądarki. Obiekt ten możemy wykorzystać w JavaScriptcie na przykład po to, aby doładować treść strony po np. kliknięciu przycisku.

O wspomnianym obiekcie opowiemy sobie w kolejnych rozdziałach.

Korzyści stosowania tego rozwiązania

Najważniejszą i chyba najbardziej oczywistą korzyścią jest brak potrzeby przeładowywania całej strony za każdym razem, gdy odpytujemy o coś serwer. Możemy za pomocą JavaScriptu podpiąć wywołania zwrotne pod pewne zdarzenia i generować treść dynamicznie.

Równie ważną korzyścią jest możliwość budowania aplikacji internetowych, które działają na zasadzie typowych aplikacji okienkowych. Z tą różnicą, że trzeba doładować treść aplikacji z serwera, a nie bezpośrednio z dysku.

Technologia ta z całą pewnością zrewolucjonizowała internet i każdy front-end developer powinien ją znać.

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

✓ Zapoznałem się!





12.3. Co to znaczy synchroniczne i asynchronicznie?

Omówimy sobie teraz nieco szerzej pojęcia asynchroniczności i synchroniczności, które są bardzo istotne przy pracy z JavaScriptem.

Synchroniczność

Kiedy mówimy o synchroniczności kodu, mamy na myśli, że nasz skrypt wykonuje się linijka po linijce. Problem pojawia się w momencie, w którym wykonanie fragmentu kodu zajmuje naszej przeglądarce dłużej niż kilka milisekund (np. podczas pobierania danych z serwera).

JavaScript jest synchronicznym językiem programowania, a mimo tego na stronach typu Facebook albo Google nie zauważamy zjawiska zawieszenia interfejsu.

Asynchroniczność

Funkcje takie jak `setTimeout` (która wykonuje się po określonym czasie) albo `setInterval` (która wykonuje się co określony odcinek czasu) jakoś sobie z tym radzą. Działają jakby nie po kolei. Na przykład funkcja:

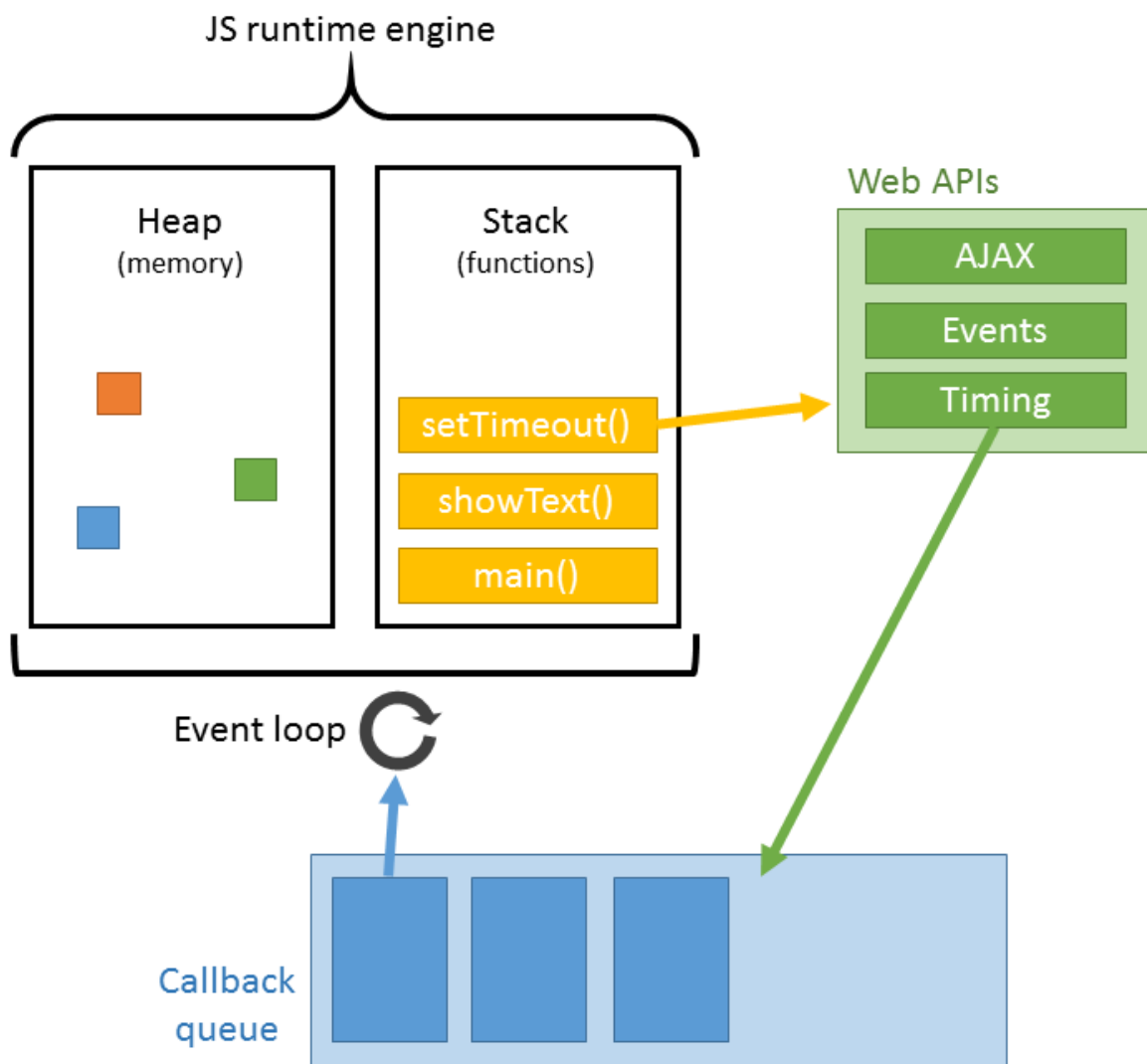
```
setTimeout(function() {  
    console.log('Hello');  
}, 5000);
```

która ma za zadanie wyświetlić w konsoli "Hello", nie zawiesza interfejsu na pięć sekund. W tym przypadku, mimo tego, że JavaScript jest synchroniczny, to funkcja wykonuje się asynchronicznie. Jak to się dzieje? Odpowiedzią na to pytanie jest tzw. pętla zdarzeń (ang. *event loop*).

Pętla zdarzeń



Pętlę zdarzeń w najprostszym przypadku można opisać na podstawie tego diagramu:



Wyjaśnijmy sobie bardzo krótko, czym są pojęcia użyte w tym obrazku:

Heap (pol. *sterta*) — jest to obszar w pamięci, który jest utworzony na potrzeby działania skryptu. JavaScript używa tego obszaru do alokowania pamięci na dane. Tej pamięci nie będziemy używać do opisu pętli zdarzeń.

Stack (pol. *stos*) — jest to, podobnie jak sterta, pamięć, która jest tworzona na potrzeby działania naszego programu. Na tej pamięci odkładane są każdorazowo wywołania funkcji. Na obrazku widzimy 3 elementy na stosie. To znaczy, że funkcja `main()` wywołała funkcję `showText()`, ta wywołała funkcję `setTimeout()`.

WebAPIs — co prawda jeszcze nie omawialiśmy pojęcia API, ale używaliśmy już go w module 10 do wstawienia mapy Google na stronę. WebAPI nie są elementami języka JavaScript, a środowiskami, w których język działa (w tym wypadku jest to przeglądarka).



Callback queue — czyli kolejka wywołań zwrotnych. Do tego miejsca trafiają wszystkie funkcje, które mają się wykonać po wystąpieniu pewnego zdarzenia (np. na kliknięcie, na załadowanie drzewa DOM, po skończeniu 5-sekundowego oczekiwania).

Event loop — to nasza pętla zdarzeń, która wrzuca na stos elementy stojące **w kolejce wywołań zwrotnych**.

Jak to działa?

1. Najpierw wykonywane są wszystkie funkcje, które znajdują się na stosie. Po kolei. Pamiętajmy, że JavaScript jest językiem synchronicznym.
2. W naszym skrypcie pojawia się `setTimeout(sayHello, 5000)`, która po 5 sekundach wykona funkcję zwrotną `sayHello`. Funkcja `setTimeout` jest zaimplementowana przez twórców przeglądarki i fundamentalnie działa troszkę inaczej, jeśli chodzi o wykonywanie zadań asynchronicznie. Z punktu widzenia napisanego przez nas skryptu funkcja wykona się synchronicznie, ale to przeglądarka przeniesie wywołanie zwrotne `sayHello` do kolejki wywołań dopiero po pięciu sekundach.
3. Jeśli na stosie nie będzie już żadnych funkcji, to wtedy pętla zdarzeń wrzuci na stos pierwszy element z kolejki wywołań. W naszym przykładzie będzie to funkcja `sayHello`, która w końcu zostanie wykonana!

Szczegółowy (co wcale nie oznacza, że skomplikowany) opis całego zjawiska asynchroniczności JS znajdziesz [tutaj](#). Bardzo polecamy się z nim zapoznać!

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

✓ Zapoznałem(a)m się!

12.4. Obiekt XMLHttpRequest



Przez trzy ostatnie submoduły bardzo dużo czytaliśmy na temat teorii. To jednak bardzo ważne, żeby każdy programista JavaScript pracujący w środowisku przeglądarki:

- znał zasady działania internetu i protokołów http (metody, nagłówki, kody odpowiedzi),
- wiedział, czym jest Ajax,



- wiedział, w jaki sposób działa JavaScript (synchroniczność, asynchroniczność, pętla zdarzeń).

Przejdźmy teraz do bardziej szczegółowego opisu obiektu *XMLHttpRequest*, za pomocą którego możemy korzystać z dobrodziejstw Ajaksa.

Po pierwsze, do pobierania danych z serwera obiekt korzysta z protokołu HTTP. XML to sposób prezentacji danych, ale jak się okazuje, możemy korzystać też z bardziej przyjaznych struktur, takich jak np. JSON, o którym jeszcze przeczytasz.

Aby skorzystać z obiektu *XMLHttpRequest*, musimy utworzyć instancję obiektu, otworzyć adres URL, a na koniec wysłać zapytanie.

Spójrzmy:

```
var request = new XMLHttpRequest(); // first step
request.open('GET', 'http://apis.is/concerts'); // second step
request.send(); // third step
```

Pierwszy krok po przejściu przez moduł o obiekowym JS na pewno jest dla Ciebie jasny. Tworzymy nową instancję *XMLHttpRequest*.

W drugim kroku na instancji *request* używamy metody *open*, która przyjmuje minimalnie dwa parametry. Maksymalnie tych parametrów może być aż pięć:

- pierwszy parametr to metoda HTTP, której użyjemy przy zapytaniu,
- drugi parametr to adres URL zapytania,
- trzeci parametr, którego nie ma w naszym przykładzie, określa, czy zapytanie jest asynchroniczne (*true*) czy synchroniczne (*false*). Domyślnie flaga ta jest ustawiona na *true*, czyli zapytanie wykonywane jest asynchroniczne (patrz pętla zdarzeń),
- czwarty parametr to opcjonalna nazwa użytkownika potrzebna w celach autoryzacji,
- piąty parametr to opcjonalnie hasło. Jest ono również potrzebne w celach autoryzacji.

Trzeci krok wysyła zapytanie :) Metoda ta może przyjąć jako parametr np. dane formularza, ale tylko jeśli zapytanie używa innej metody niż GET lub HEAD.

Jeśli zapytanie jest synchroniczne, to metoda wstrzymuje działanie skryptu (można to zauważyć przez zawieszenie interfejsu).

Natomiast jeśli działanie jest asynchroniczne, to metoda zwraca wynik natychmiastowo, a kiedy pojawi się odpowiedź z serwera, zostanie wysłane powiadomienie (zdarzenie), którego musimy nasłuchiwać w celu wychwycenia



odpowiedzi.

W tym przypadku nie nasłuchujemy żadnego zdarzenia, mimo że wywołujemy metodę asynchronicznie. To znaczy, że gdy przyjdzie odpowiedź, my nigdy się o niej nie dowiemy (chyba że podejrzemy ruch sieciowy w zakładce Network). Chcemy, żeby odpowiedź z serwera była dla nas użyteczna. Co należy zrobić?

Są dwa rozwiązania tego problemu:

Synchroniczne

```
var request = new XMLHttpRequest();
request.open('GET', 'http://apis.is/concerts', false); // async flag
request.send(); // interface suspension
if(request.status == 200) {
    console.log(request.response);
}
```

Asynchroniczne

```
var request = new XMLHttpRequest();
request.open('GET', 'http://apis.is/concerts'); // just like it's been
request.onload(function() {
    if (request.status == 200) {
        console.log(request.response);
    }
});
request.send();
```

Obie metody wykonują tę samą czynność, ale w inny sposób.

Pierwsza metoda działa synchronicznie, czyli zauważymy zawieszenie interfejsu trwające mniej-więcej tyle, ile czas połączenia z serwerem.

Druga metoda jest asynchroniczna, czyli kod po wykonaniu metody **send** będzie wykonywał się dalej, a w momencie odpowiedzi (**onload**), zgodnie z event loop, wykona callback, gdy tylko serwer dostarczy nam odpowiedzi.

Warunek:



```
if(request.status == 200) {  
    console.log(request.response);  
}
```

sprawdza, czy status odpowiedzi jest równy 200 (czyli wszystko OK). Jeśli tak, to wyświetla odpowiedź z serwera.

Atrybuty obiektu XMLHttpRequest

Do najważniejszych atrybutów obiektu XMLHttpRequest należą:

- **response** — atrybut trzyma w sobie ciało odpowiedzi. Może to być np. zwykły text, surowa wartość binarna (ArrayBuffer), dokument;
- **responseText** — podobnie jak odpowiedź, z tą różnicą, że odpowiedzią na żądanie może być jedynie text,
- **responseType** — określa, jakiego rodzaju jest odpowiedź otrzymana z serwera. Przykładowe wartości to: "arraybuffer", "text", "document", "json",
- **status** — zwraca wartość numeryczną kodu odpowiedzi. Przed wysłaniem zapytania wartość kodu wynosi 0,
- **statusText** — tutaj, w przeciwieństwie do poprzedniej wartości, trzymana jest wartość tekstowa kodu, np. "OK" lub "Not Found",
- **withCredentials** — ta wartość określa, czy zapytania powinny używać jakiegokolwiek metody autoryzacji, np. ciasteczek lub nagłówków.

Metody obiektu XMLHttpRequest

Ważniejsze metody XMLHttpRequest to:

- **open** — tworzy nowe zapytanie,
- **send** — wysyła zapytanie,
- **abort** — jeśli zapytanie zostało już wysłane, ta metoda je przerywa,
- **setRequestHeader** — metoda ustawia wartość konkretnego nagłówka przed wysłaniem zapytania (należy jej użyć po metodzie open, ale przed metodą **send**). Może być przydatna do ustawiania nagłówków przy autoryzacji;
- **getAllResponseHeaders** — zwraca wszystkie nagłówki,



- **onload** — metoda, która wykonana się, kiedy dostaniemy odpowiedź z serwera, niezależnie od kodu odpowiedzi,
- **onprogress** — ta metoda może być przydatna do pokazywania postępu ładowania np. w formie paska z wartościami od 0% do 100%,
- **onerror** — ta metoda zostanie wykonana, jeśli serwer nie zwróci nam użytecznej odpowiedzi, np. ktoś utnie nam kabel od internetu,
- **addEventListener** — to jest najbardziej ogólna metoda wykorzystywana do nasłuchiwanie zdarzeń występujących przy zapytaniu. Z tej metody korzystaliśmy też w kontekście elementów drzewa DOM.

Należy pamiętać, że wszystkie metody musimy wywołać. Przykładowo:

`request.open()`, `request.send()`.

Zadanie: Pierwsze zapytanie do serwera

Spróbujmy teraz wykonać proste zadanie, które będzie miało na celu pobieranie losowego żartu o Chucku Norrisie :)

Efekt prac umieścisz jak zazwyczaj na GitHubie i przekażesz mentorowi.

Do tego celu potrzebny nam będzie prosty szablon z przyciskiem do losowania kolejnego żartu i miejscem na żart.

```
<button id="get-joke">Random joke</button>
<p id="joke"></p>
```

Przejdźmy teraz do skryptu. Utwórz zmienną `url`, która będzie zawierała pełny adres do naszego dowcipu:

```
var url = 'http://api.icndb.com/jokes/random';
```

Trzeba też znaleźć odpowiedni przycisk na stronie i podpiąć nasłuchiwanie na kliknięcie tak, aby każdorazowo pobierał nam się losowy żart.



```
var button = document.getElementById('get-joke');
button.addEventListener('click', function(){
    getJoke();
});
```

Do zmiennej **paragraph** przypiszemy element DOM paragrafu, który odpowiada za wyświetlanie dowcipu.

```
var paragraph = document.getElementById('joke');
```

Ostatnim elementem, który należy wykonać, jest implementacja funkcji **getJoke()**:

```
function getJoke() {
    var xhr = new XMLHttpRequest();
    xhr.open('GET', url);
    xhr.addEventListener('load', function(){
        var response = JSON.parse(xhr.response);
        paragraph.innerHTML = response.value.joke;
    });
    xhr.send();
}
```

Najpierw tworzymy nową instancję obiektu XMLHttpRequest, potem otwieramy połączenie z wybranym adresem.

Do połączenia należy podpiąć nasłuchiwanie na odpowiedź z serwera. W callbacku tworzymy zmienną **response** i przypisujemy do niej tę dziwną linię:

```
var response = JSON.parse(xhr.response);
```

Czym jest JSON?

JSON to akronim od JavaScript Object Notation. Jest jednym z formatów służących do wymiany danych. Jego składnia jest bardzo podobna do obiektu JavaScript, z tą różnicą, że w formacie JSON klucz również musi być w postaci stringa.

Przykład:



```
// CORRECT
{
  "firstName": "Jan",
  "lastName": "Nowak"
}
```

```
// WRONG
{
  firstName: "Jan",
  lastName: "Nowak"
}
```

Tego formatu, ze względu na podobieństwo do obiektów JavaScriptowych, używa się bardzo łatwo przy żądaniach ajaksowych. Trzeba jednak używać specjalnych metod wbudowanego obiektu JSON, które umożliwiają prawidłowe wysłanie lub odbiór obiektu.

JSON.stringify(obiekt) — służy do przygotowania obiektu do wysłania, np. metodą POST.

JSON.parse(string) — służy do zamieniania wartości tekstowej w formacie JSON na zrozumiałą dla JS obiekt, którym można manipulować.

Tak więc ta linijka mówi nam, że dostajemy od serwera odpowiedź w formacie JSON i musimy ją "sparsować" na obiekt JavaScriptowy.

```
var response = JSON.parse(xhr.response);
```

Wejdźmy teraz do zakładki Network w narzędziach developerskich i wciśnijmy przycisk "Random joke". Zobaczmy, co się wydarzy.



Random joke

Since 1940, the year Chuck Norris was born, roundhouse-kick related deaths have increased 13,000 percent.

Collections ▾ Console Assets Comments Delete Shortcuts

⌕ | Elements Console Sources Network Timeline Profiles Resources Security Audits

⏹ | View: [List Icon] [Code Icon] [Table Icon] [Timeline Icon] | [x] Preserve log [x] Disable cache | No throttling ▾

Filter [] [x] Regex [x] Hide data URLs [x] All XHR JS CSS Img Media Font Doc WS Manifest Other

	50 ms	100 ms	150 ms	200 ms	250 ms	300 ms	350 ms	400 ms
Name				Status	Type		Initiator	
<input type="checkbox"/> random				200	xhr		<u>VM1616 pen.js:14</u>	

Powinniśmy zaobserwować ruch sieciowy w postaci zapytania. Kliknijmy w celu podejrzenia szczegółów zapytania.

Random joke

Since 1940, the year Chuck Norris was born, roundhouse-kick related deaths have increased 13,000 percent.

Collections ▾ Console Assets Comments Delete Shortcuts

⌕ | Elements Console Sources Network Timeline Profiles Resources Security Audits

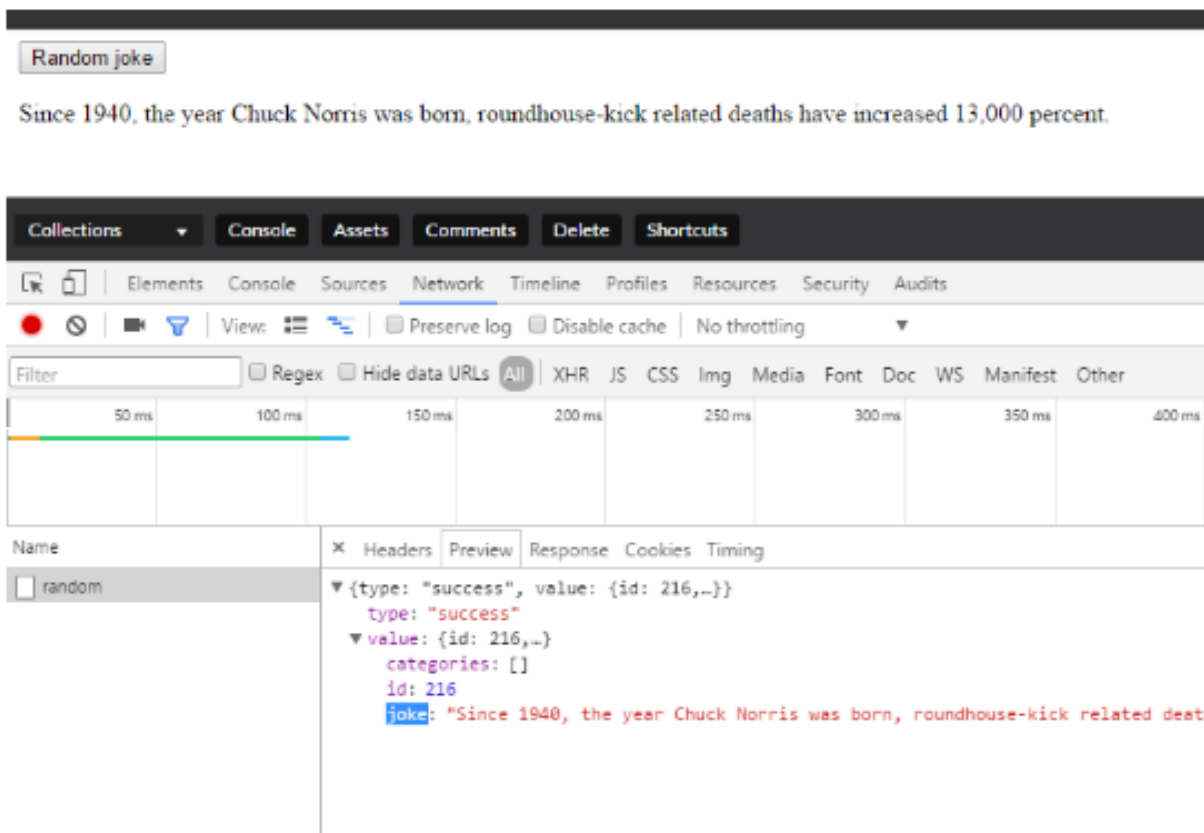
⏹ | View: [List Icon] [Code Icon] [Table Icon] [Timeline Icon] | [x] Preserve log [x] Disable cache | No throttling ▾

Filter [] [x] Regex [x] Hide data URLs [x] All XHR JS CSS Img Media Font Doc WS Manifest Other

	50 ms	100 ms	150 ms	200 ms	250 ms	300 ms	350 ms	400 ms
Name				✕ Headers Preview Response Cookies Timing				
<input type="checkbox"/> random				<div>▼ {type: "success", value: {id: 216,...}} type: "success" ▶ value: {id: 216,...}</div>				

Wejdźmy w zakładkę Preview. Zakładka Response pokazuje surową odpowiedź w formacie JSON. Przeglądarka potrafi za nas sparsować odpowiedź i pokazać nam ją właśnie w zakładce Preview. Widzimy, że obiekt ma dwa klucze: typ i value. Rozwińmy obiekt value.





W obiekcie znaleźliśmy treść żartu. Ok, czyli aby dostać się do samego dowcipu, należy dotrzeć do następującego klucza: **response.value.joke**

To wyjaśniałoby ostatnią linijkę w callbacku:

```
paragraph.innerHTML = response.value.joke;
```

Jako tekst paragrafu z dowcipem ustawiamy wartość, która siedzi w odpowiedzi z serwera, czyli w response.value.joke.

Przetestujmy, czy wszystko działa prawidłowo. Kliknij parę razy przycisk i sprawdź, czy generowane są nowe dowcipy.

Dla chętnych:

Zastanów się, jak zmodyfikować skrypt, aby przy pierwszym wejściu na stronę nie trzeba było wciskać przycisku pobierania dowcipu. Innymi słowy: zastajemy stronę z już wygenerowanym dowcipem. Podpowiedź: można to zrobić, dopisując jedną linijkę kodu.

Podgląd zadania



<https://github.com/0na/>

Wyślij link ✓

12.5. Fetch API



Do tej pory wykorzystywaliśmy starą metodę komunikacji z serwerem. Od jakiegoś czasu przeglądarki dają nam dostęp do nowego API, które jest prostsze i zgodne z nowymi standardami — **Fetch API**

Jedną z głównych zalet Fetch API jest zwracanie tzw. **Promises**, czyli obietnic, które w bardzo przyjemny sposób rozwiązują problem pracy z kodem asynchronicznym. Są bardzo dobrą odpowiedzią na stare i wysłużone callbacki.

Promise jest częścią stosunkowo nowego standardu JavaScript — ES6. Promise jest obiektem, który reprezentuje stan operacji asynchronicznej (ostateczne zakończenie lub niepowodzenie) oraz wynikową wartość zapytania.

Przykładowe wywołanie zapytania do serwera z użyciem **Fetch API**:

```
fetch("http://api.icndb.com/jokes/random")
  .then(function(resp) {
    return resp.json();
  })
  .then(function(result) {
    // code that will be executed when the server responds correctly
  })
  .catch(function(error) {
    // code that will be executed in the case of an incorrect response
  });
```

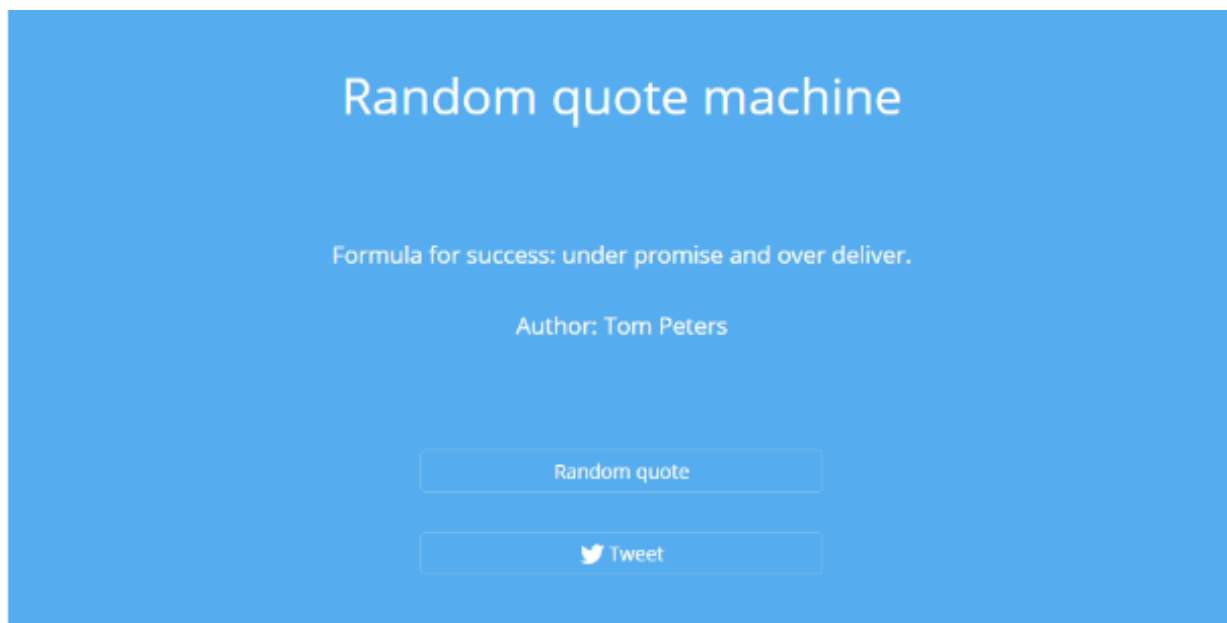
W funkcji **fetch** podajemy adres docelowy, z którego chcemy pobierać dane. Domyślną metodą komunikacji jest GET. Następnie otrzymujemy w odpowiedzi obiekt, tzw. **ReadableStream**. Aby zmienić go na format odpowiedzi typu JSON, musimy zastosować odpowiednią metodę, tutaj: **resp.json()**



Fetch API jest nowocześniejszym sposobem obsługi zapytań asynchronicznych do zasobów internetowych. Zapisy funkcji użytych w tym module nie dają pełnej korzyści (pod względem zapisu kodu), ponieważ staramy się nie używać jeszcze ES6, standardu, który poznamy w module 15. Stosowanie Fetch API jest prostsze względem wysłużonego już obiektu XMLHttpRequest. Dodatkowo zwracanie Promise w odpowiedzi daje możliwość tworzenia czytelniejszego kodu, ponieważ unikamy przesyłania i używania funkcji zwrotnych (callback).

Zadanie: Aplikacja losująca cytaty

W tym zadaniu zbudujemy prostą apkę do pobierania cytatów przez API, która – podobnie jak generator żartów o Chucku Norrisie – będzie wyświetlała na naszym ekranie losowo wybrany cytat. Oprócz tego dodamy możliwość wrzucania pobranego cytatu na Twittera. Całość powinna wyglądać mniej więcej tak:



Pamiętaj, aby po zakończeniu prac nad swoją aplikacją do losowania cytatów udostępnić swój kod na GitHubie i przekazać go mentorowi do oceny :)

Założenia

1. Twitter przyjmuje maksymalnie 140 znaków w wiadomości. Jeśli dane, które prześlemy Twitterowi, będą dłuższe, to zostaną przycięte do maksymalnej dozwolonej długości. Oczywiście cytaty straciłyby wtedy sens, więc musimy odrzucać te, które będą zbyt długie.
2. Cytat powinien pobierać się przy pierwszym załadowaniu strony oraz przy kliknięciu w przycisk z napisem "Random quote".



3. Przycisk "Tweet" powinien rzecz jasna udostępnić tweeta z cytatem.

W projekcie nie będziemy zajmować się wyglądem. Ustawimy sobie podstawową strukturę, a ostylewanie całości będzie dla Ciebie dodatkową okazją do przećwiczenia HTML i CSS :) My zajmiemy się logiką aplikacji.

Struktura

Zacznijmy od utworzenia struktury HTML:

```
<main>
  <header>
    <h1>Random quote machine</h1>
  </header>
  <div class="box">
    <h2 class="quote"></h2>
    <h3 class="author"></h3>
  </div>
  <div>
    <button class="trigger"> Random quote </button>
    <a href="#" target="_blank" class="tweet"><i class="fa fa-lg
  </div>
</main>
```

Adresy URL

Przejdźmy teraz do pisania naszych skryptów :) Będą one działały jedynie na serwerze (API wymaga protokołu http lub https), zatem kod należy testować hostując stronę na [GitHub Pages](#) lub innym serwerze.

W naszym skrypcie potrzebujemy tym razem dwóch adresów URL — pierwszego do wysyłania tweetów, drugiego do pobierania cytatów.

Dodaj poniższe zmienne do swojego pliku ze skryptami.

```
var tweetLink = "https://twitter.com/intent/tweet?text=";
var quoteUrl = "https://quotesondesign.com/wp-json/posts?filter[order
```

Pierwszy link jest standardowym linkiem do wysyłania tweetów na Twittera. Jedyne, czego mu brakuje, to samej treści tweeta, którą będziemy dodawać na końcu po znaku = za pomocą kodu JavaScript.



Drugi URL to link do API Quotes on Design, które pozwala nam pobierać losowe cytaty ze swojej bazy. Użyjemy go w metodzie, którą poznamy za chwilę.

Pobieranie cytatu

Kolejnym krokiem będzie napisanie logiki, która pobierze nam losowy cytat za pomocą API.

```
function getQuote() {  
  fetch(quoteUrl, { cache: "no-store" })  
    .then(function(resp) {  
      return resp.json();  
    })  
    .then(createTweet);  
}
```

Opiszemy teraz parametry, których użyliśmy:

- **quoteUrl** (pierwszy parametr) to adres zapytania (czyli nasz link do API),
- **{ cache: "no-store" }** – stosujemy to, aby wyłączyć możliwość zaglądania do HTTP Cache przez przeglądarkę. Innymi słowy – chcemy, aby przeglądarka za każdym razem pytała podany URL o dane.
- **createTweet** (drugi parametr) to po prostu funkcja, która zostanie wykonana przy pomyślnym wykonaniu zapytania.

Tworzenie tweeta

Funkcja **createTweet()** ma za zadanie tworzyć linki z tweetami i podpinąć je pod przycisk do tweetowania. Będziemy sukcesywnie dopisywać kolejne fragmenty tej funkcji. Zaczniemy od sprawdzenia autora cytatu:

```
function createTweet(input) {  
  var data = input[0];  
  
  var dataElement = document.createElement('div');  
  dataElement.innerHTML = data.content;  
  var quoteText = dataElement.innerText.trim();  
  var quoteAuthor = data.title;  
  
  if (!quoteAuthor.length) {  
    quoteAuthor = "Unknown author";  
  }  
}
```



Zwróć uwagę na to, co dzieje się w deklaracji zmiennej `quoteText`, a tak właściwie to nad nią. Jeśli sprawdzisz, co kryje się pod kluczem `data.content`, zobaczysz że jest to zwykły kod HTML paragrafu. Niestety, nie jest to ten format danych, o jaki nam chodzi. W związku z tym tworzymy nowy element HTML, następnie uzupełniamy jego właściwość `innerHTML` wartością z tego klucza, a potem wyciągamy z niego zawartość tekstową za pomocą właściwości `innerText` – sprytne, prawda? :) Oprócz tego, wykorzystaliśmy też metodę `.trim()`, która pozwoli nam "uciąć" niepotrzebne spacje na początku/końcu stringa.

Jeśli autor cytatu jest pustym *stringiem* (jego długość jest równa 0), to w pole autora należy wpisać *"Unknown author"*.

Wskazówka: konstrukcję `!quoteAuthor.length` JavaScript potraktuje jako `true`. Dzieje się to w następujący sposób:

1. `quoteAuthor.length` zwróci wartość 0 w przypadku, gdy autor cytatu będzie pusty – JavaScript interpretuje zerową długość jako po prostu `false`.
2. Wykrzyknik na początku (`!quoteAuthor.length`) zaneguje wartość fałszu i zrobi z niej prawdę, czyli jeśli autor cytatu jest pusty, to wejdziemy do treści warunku.

Kolejną rzeczą, którą trzeba dopisać wewnątrz funkcji `createTweet()`, jest wygenerowanie treści tweeta. Pamiętaj, aby koniecznie umieścić deklarację tej zmiennej poza warunkiem *if*, który pisaliśmy wcześniej, w przeciwnym wypadku otrzymasz błąd o niezdefiniowanej zmiennej. Kod, który musimy teraz wprowadzić, wygląda następująco:

```
var tweetText = "Quote of the day - " + quoteText + " Author: " + qu
```

Tak będzie wyglądał przykładowy tweet z cytatem:

Quote of the day - I never worry about action, but only inaction Author: Winston Churchill

Skoro mamy treść tweeta, to możemy sprawdzić, czy nie wykraczamy przypadkiem poza maksymalną długość 140 znaków (z której zresztą Twitter jest znany).

```
if (tweetText.length > 140) {  
    getQuote();  
}
```



Jeśli wykraczamy poza 140 symboli, należy jeszcze raz wygenerować tweeta. Jeśli długość tweeta jest prawidłowa, możemy pokazać cytaty użytkownikowi i podpiąć pod link, który zajmie się generowaniem tweeta. Aby upewnić się, że ten warunek jest spełniony, w funkcji `createTweet()` dodamy następujący fragment kodu:

```
if (tweetText.length > 140) {  
    getQuote();  
} else {  
    var tweet = tweetLink + encodeURIComponent(tweetText);  
    document.querySelector('.quote').innerText = quoteText;  
    document.querySelector('.author').innerText = "Author: " + quoteA  
    document.querySelector('.tweet').setAttribute('href', tweet);  
}
```

Zmienna `tweet` to złożenie dwóch elementów: linka do generowania nowych tweetów oraz samego tekstu tweeta.

`document.querySelector('.quote')` to element, w którym wyświetlamy treść naszego cytatu.

`document.querySelector('.author')` jest elementem, w którym pokazujemy autora cytatu.

Ostatnia linijka kodu w funkcji `createTweet()` to:

```
document.querySelector('.tweet').setAttribute('href', tweet);
```

Wybieramy w niej element z klasą `.tweet` i modyfikujemy zawartość atrybutu `href` na URL tweeta, który trzymany jest w zmiennej `tweet`.

Jest to kompletna implementacja funkcji `createTweet`. Przejdźmy teraz do ostatniego fragmentu, jakim jest generowanie nowego tweeta po wejściu na stronę oraz po kliknięciu w przycisk z napisem "Random quote".

Ostatnie szlify

Po załadowaniu strony należy:

1. Wygenerować cytaty
2. Podpiąć na element o klasie `.trigger` nasłuchiwanie na zdarzenie kliknięcia, po którym ma się wykonać funkcja generująca cytaty



Pomoże nam w tym następujący kod:

```
document.addEventListener('DOMContentLoaded', function() {  
  getQuote();  
  document.querySelector('.trigger').addEventListener('click', func  
    getQuote();  
  });  
});
```

Przetestujmy działanie naszego skryptu i sprawdźmy, czy wszystko działa jak należy. Jeśli Twój skrypt radzi sobie z pobieraniem losowego cytatu i wstawianiem go na Twittera, to znaczy, że wszystko poszło świetnie — **gratulacje!** :)

W niektórych przypadkach możemy napotkać się na problem z **CORS**, który nie pozwoli nam na poprawne wyświetlanie naszych cytatów.

Jest na to sposób! Przed definiowaniem naszych linków dodajmy prefix w postaci url:

```
var prefix = "https://cors-anywhere.herokuapp.com/";
```

W metodzie `getQuote()` wstaw `fetch(prefix + quoteUrl, { cache: "no-store" })`

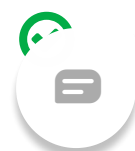
Powodzenia!

Podgląd zadania

<https://github.com/0na/>

Wyślij link ✓

12.6. Co kryje się pod akronimem API



API to akronim **A**pplication **P**rogramming **I**nterface, co tłumaczy się jako Interfejs Programistyczny Aplikacji. Oznacza to sposób komunikacji komponentów programistycznych między sobą.

Z API już korzystaliśmy!

API już stosowaliśmy, np. korzystając z obiektu **document**, którego używaliśmy do wybierania ze strony odpowiednich elementów np. za pomocą metody **getElementById()**. Ta metoda to interfejs, który został nam udostępniony przez innych programistów, abyśmy mogli wykonać określone zadanie bez znajomości implementacji metody.

Nie zastanawiamy się za bardzo, w jaki sposób zostało to napisane. Mamy API, które pozwala nam w prosty sposób komunikować się między obiektami.

API i Ajax

Ok, ale jak to wszystko ma się do komunikacji z serwerem? Otóż API w kontekście Ajaksa oznacza, że do komunikacji używamy tzw. *endpointów*.

Endpointy to URLe, które — odpowiednio użyte — w odpowiedzi zwracają developerom różne dane w formacie np. JSON.

Są zazwyczaj udostępniane przez back-end (czyli programistów zajmujących się częścią serwerową aplikacji). Przyjrzyjmy się prostemu przykładowi API serwisu *Imgur*.

Imgur to znany serwis, który zajmuje się hostingiem zdjęć. Jest to miejsce, gdzie użytkownicy mogą po prostu wrzucać swoje zdjęcia prywatnie lub publicznie.

Jak się okazuje, z serwisu możemy korzystać nie tylko wchodząc na jego stronę i bezpośrednio działając tylko w tamtym miejscu, ale możemy również wykorzystać jego funkcjonalności właśnie dzięki API, które Imgur nam udostępnia.

Jak korzystać z API?

Każdy, kto udostępnia nam interfejs programistyczny swojej aplikacji (API), musi dołączyć do tego jakąś "instrukcję obsługi", czyli dokumentację. Do tej pory każda biblioteka, z którą mieliśmy do czynienia, miała swoją dokumentację. Zresztą nikt, kto



odrobinę ceni swój czas, nie korzysta z API, jeśli nie wie, do czego służy!

Imgur jest profesjonalnym serwisem i oczywiście w parze z API dostajemy pełną dokumentację, którą możemy zobaczyć [tutaj](#).

RESTful API

Większość nowych aplikacji powstaje w oparciu o wzorzec RESTowy, który ma na celu narzucenie dobrych praktyk przy tworzeniu API. REST to skrót od *Representational State Transfer*. Wzorzec ten wykorzystuje protokół HTTP omawiany wcześniej.

Znajomość metod (GET, POST, PUT, DELETE) jest tutaj bardzo potrzebna. Do tej pory w większości przypadków wykorzystywaliśmy GET i POST, ale wzorzec RESTowy idzie o krok dalej. Zakłada istnienie tzw. zasobów (ang. *resource*).

Czym jest zasób?

Zasób to obiekt, który ma określony typ, powiązane z nim dane i może być w relacjach z innymi obiektami. Przykład Imgur rozjaśni, o czym mowa.

Zacznijmy od rodzajów zasobów:

- użytkownicy,
- albumy ze zdjęciami,
- zdjęcia.

Przykład relacji między zasobami:

- użytkownik może mieć wiele albumów,
- album może zawierać wiele zdjęć,
- dany album może być przypisany (może należeć) tylko do jednego użytkownika,
- dane zdjęcie może należeć tylko do jednego albumu.

Grupowanie zasobów

Zasoby mogą być powiązane w tzw. **kolekcje** (ang. *collections*). Każda kolekcja musi zawierać zasoby tego samego typu, np. człowiek. Typy zasobów w kolekcji nie mogą się mieszać. Kiedy zbieramy znaczki, a w naszym klastrze znajdziemy monetę, to ją stamtąd wyrzucamy, bo nie jest elementem naszej kolekcji!

Pojedynczy zasób bywa często określany modelem. Model oczywiście może być częścią kolekcji.

JSON sprawdza się idealnie do przesyłania informacji na temat zasobu, ponieważ kolekcję możemy przedstawić w prosty sposób za pomocą arraya (tablicy). Do reprezentacji pojedynczego modelu idealnie nadaje się Obiekt. Istnieją też inne formaty,



takie jak **XML**, czy **YAML**, jednak coraz rzadziej się je spotyka. Dominuje zdecydowanie JSON!

Przykłady

Weźmy kilka przykładowych **endpointów**, które oferuje nam Imgur i je omówmy. Będziemy używać następującej notacji: metoda **adres_url**

Imgur API — pobranie informacji o koncie użytkownika

```
GET https://api.imgur.com/3/account/{user_name}
```

W odpowiedzi dostajemy pojedynczy model użytkownika, który zawiera te same informacje, które możemy znaleźć na stronie Imgura. Należą do nich:

- identyfikator (*id*),
- opis konta użytkownika (*bio*),
- popularność konta (*reputation*),
- kiedy konto zostało utworzone (*created*).

Ok, ale to nic nowego. Robiliśmy już podobne rzeczy wcześniej. Metoda GET i adres URL do zasobu są nam znane choćby z przykładu z cytataми albo z dowcipami o Chucku Norrisie.

Uaktualnienie informacji o koncie użytkownika

```
PUT https://api.imgur.com/3/account/{user_name}/settings
```

Tym razem użyliśmy metody PUT. Zastanówmy się przez chwilę: czy możemy zmieniać ustawienia każdego konta dowolnie? Intuicja podpowiada, że nie! W tym miejscu poprawne zapytanie wymaga bycia zalogowanym na konto, którego ustawienia chcemy zmienić. Zaraz też troszkę o tym powiemy.

Teraz zobaczmy, co możemy uaktualnić:

- wszystko, co w poprzednim endpointzie (*oprócz ID, rzecz jasna!*),
- możliwość odbierania prywatnych wiadomości od innych użytkowników (*messaging_enabled*),
- nazwę użytkownika (*username*),
- zgodę na wyświetlanie treści dla dorosłych (*show_mature*),
- i kilka innych, które możemy znaleźć w dokumentacji.

Oczywiście nie chodzi tutaj o to, żeby wymieniać, jakie zostały nam udostępnione opcje. Skupmy się na samej zasadzie działania RESTowego API.



Działania na albumie

W tym miejscu skorzystamy z tego samego endpointa w troszkę inny sposób.

Metoda	Endpoint
GET	<code>https://api.imgur.com/3/album/{id}</code>
POST	<code>https://api.imgur.com/3/album</code>
PUT	<code>https://api.imgur.com/3/album/{id}</code>
DELETE	<code>https://api.imgur.com/3/album/{id}</code>

Widzimy cztery bardzo podobne adresy URL. Jednak przy użyciu różnych metod HTTP zapytania mają odmienny efekt:

- **GET** — służy do pobierania konkretnego albumu,
- **POST** — tworzy nowy album,
- **PUT** — uaktualnia informacje na temat albumu,
- **DELETE** — usuwa album.

Ok, ale gdzie są wykonywane są te operacje? Napisaliśmy sobie, że album jest w relacji z użytkownikiem, tzn. użytkownik może mieć wiele albumów. To znaczy, że podglądamy/dodajemy/uaktualniamy/usuwamy album konkretnego użytkownika, który musi być zalogowany do systemu. Gdzieś muszą być trzymane informacje na temat klienta. Tylko gdzie?

Uwierzytelnianie

Wrócimy jeszcze za moment do Imgura, ale najpierw wyjaśnijmy sobie, czym jest uwierzytelnianie.

Uwierzytelnianie ma na celu potwierdzenie zadeklarowanej przez nas tożsamości. Chodzi o uzyskanie pewności, że my to faktycznie my :) Każdy serwis internetowy, na który można się zalogować, ma jakąś formę uwierzytelniania. Dzięki temu wiemy na przykład, na jakie konto chcemy założyć nowy album za pomocą: **POST** `https://api.imgur.com/3/album`, a serwer ma pewność, że to my chcemy skorzystać z API, a nie ktoś inny.

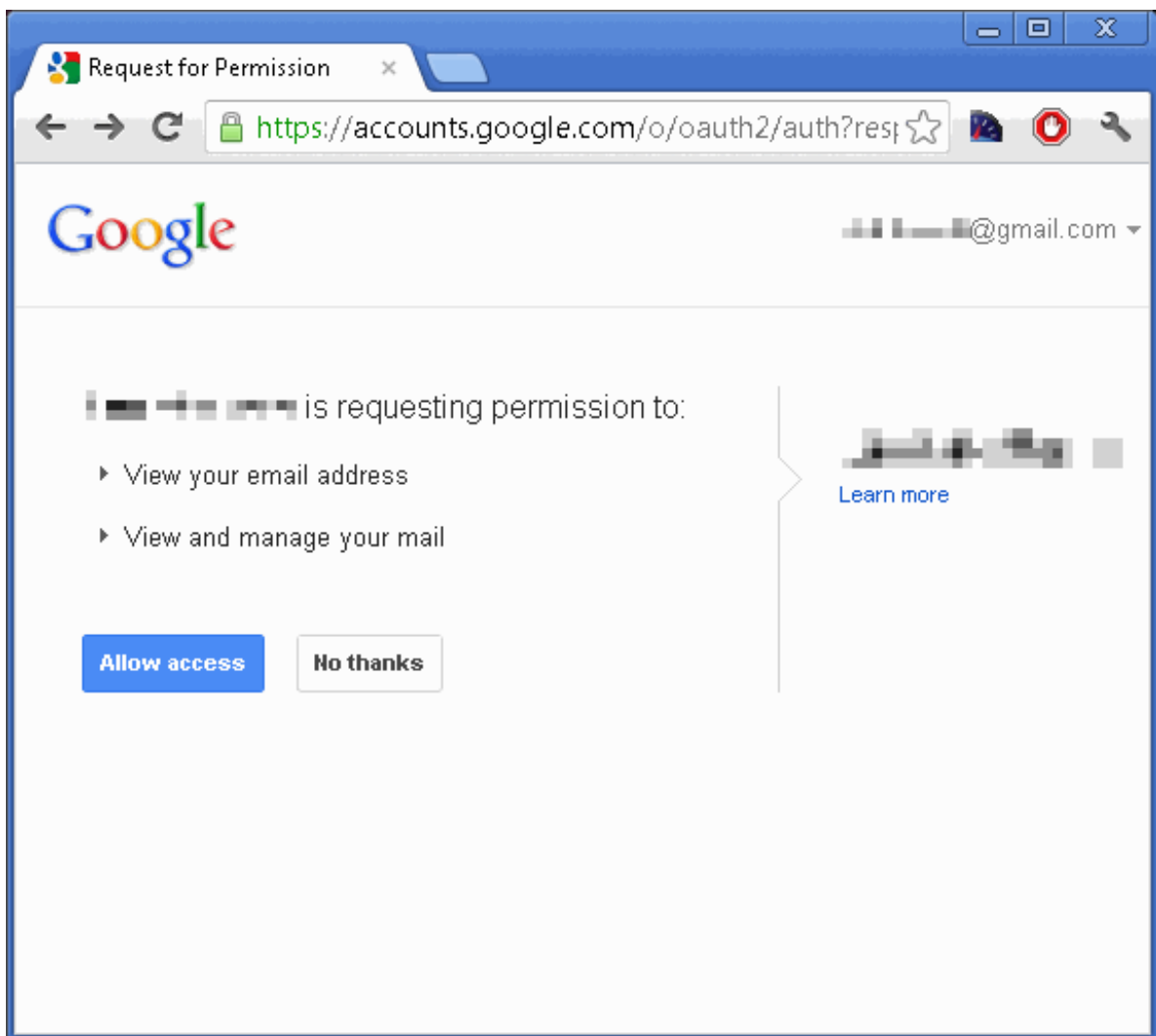


Metody uwierzytelniania

API, które służą do pobierania publicznych zasobów, nie wymagają zazwyczaj uwierzytelniania. Z takich wersji korzystaliśmy w poprzednich zadaniach. Tym razem mamy do czynienia z naszymi prywatnymi zasobami. W jaki sposób oświadczyć, że my to my? Istnieje kilka przykładowych sposobów:

- **Użycie protokołu HTTPS** — do nagłówków takiego zapytania dodajemy parametr, np. `Authorization: Basic 1Ad23DczJ8p`. Jest to najprostsza metoda, która wysyła do serwera nasze dane (login i hasło) w postaci zakodowanej. Problem przy wysyłaniu danych za pomocą zwykłego protokołu HTTP polega na tym, że kodowanie to jest znane i dane mogą wpaść w niepowołane ręce. Rozwiązaniem jest protokół HTTPS! :)
- **Użycie ciasteczek** — problem uwierzytelniania możemy rozwiązać również poprzez ciasteczka. Serwer wysyła nam w odpowiedzi na logowanie ciasteczka, które dodajemy do każdego kolejnego zapytania (w formie nagłówka `Cookie`). Ciasteczko trzyma informację o trwającej sesji. Po jakimś czasie, jeżeli nie wysyłamy żadnych zapytań, po prostu wygasa.
- **Token (OAuth 2.0)** — z tej metody korzystamy za każdym razem, kiedy logujemy się do systemu przy pomocy trzeciej strony (Google, Facebook, Twitter, GitHub). Logujemy się na swoje konto i potwierdzamy, że aplikacja, z której chcemy skorzystać, może wykorzystywać nasze dane. Wygląda to mniej więcej tak:





- **Uwierzytelnianie jako część URLa** – polega na dodaniu odpowiedniego klucza jako część adresu URL. Przykładowo: `https://api.cats.com/cats?key=aa786jhdb`

Nasz Imgur obsługuje uwierzytelnianie za pomocą metody OAuth 2.0. Więcej na temat tej metody można przeczytać w [tym](#) miejscu.

Publiczne API

Imgur nie jest jedynym publicznym API, z którego możemy skorzystać. Twórcy serwisów internetowych bardzo często udostępniają swoje API, a my, developerzy, możemy tych API śmiało używać. Przykładami mogą być:

- [Youtube](#)
- [Google Maps](#)
- [Twitter](#)
- [Facebook](#)

Tematyka

Każda aplikacja internetowa ma swoją domenę, czyli temat. Domenę można w najprostszy sposób określić słownictwem, którego w danej tematyce się używa. W aplikacji do zdjęć będą to: albumy, fotografie, zdjęcia, aparat, itp. Zauważ, że w endpointach Imgura pojawiają się słowa album czy photo.

Tematów jest oczywiście cała masa. Bardzo dużo interesujących API znajduje się w tym miejscu: <http://apis.io/>.

Zadanie: Wyszukiwarka krajów

Napišemy prostą wyszukiwarkę, która znajdzie nam kraje zawierające w nazwie wpisaną przez nas frazę.

Zadanie to wykonasz lokalnie, a następnie umieścisz kod w repozytorium na GitHubie i wyślesz link mentorowi.

HTML

Do tego celu przyda nam się prosty szablon w postaci jednego inputa do wpisywania nazwy kraju, jednego przycisku (*button*) i listy, która będzie prezentowała znalezione kraje. Może to wyglądać następująco:

```
<h1>Country search engine</h1>
<button id="search">Search a country</button>
<input id="country-name" type="text" />
<h2>List of countries</h2>
<ul id="countries">
  <li>No data</li>
</ul>
```

Każdy z elementów oznaczmy odpowiednim identyfikatorem. Przycisk — *search*, input — *country-name*, lista krajów — *countries*

Podstawowe zmienne

Zacznijmy od przypisania zmiennych z adresem URL oraz listy krajów:

```
var url = 'https://restcountries.eu/rest/v1/name/';
var countriesList = document.getElementById('countries');
```



Wyszukiwanie krajów

Na kliknięcie w "szukaj" należy uruchomić pewną funkcję. Nazwijmy ją `searchCountries()`. Do kodu tej funkcji przejdziemy za chwilę. Najpierw zerknijmy na podpięcie żądania pod przycisk z id `search`:

```
document.getElementById('search').addEventListener('click', searchCou
```

Przejdźmy do implementacji funkcji `searchCountries()`. Zanim wywołamy samo wyszukiwanie, musimy pobrać wartość wpisaną przez użytkownika i przypisać do zmiennej. Możemy tego dokonać odczytując właściwość `value`:

```
function searchCountries() {  
    var countryName = document.getElementById('country-name').value;  
}
```

Dodajmy do naszej funkcji warunek sprawdzający, czy pole tekstowe nie jest przypadkiem puste. Jeśli tak, ustawmy jego wartość na "Poland".

```
function searchCountries() {  
    var countryName = document.getElementById('country-name').value;  
    if(!countryName.length) countryName = 'Poland';  
}
```

Ostatnim krokiem jest napisanie samej logiki wyszukiwania. Do tego celu użyjemy Fetch API. Żądanie spróbujcie skonstruować sami na podstawie [dokumentacji](#).

Zauważ, że w kodzie poniżej użyliśmy parametru, który określa, jaka funkcja ma zostać wykonana w przypadku powodzenia zapytania — jest to `showCountriesList()`. Poniżej kod funkcji:

```
function searchCountries() {  
  var countryName = document.getElementById('country-name').value;  
  if(!countryName.length) countryName = 'Poland';  
  fetch(url + countryName)  
    .then(function(resp) {  
      return resp.json();  
    })  
    .then(showCountriesList);  
}
```

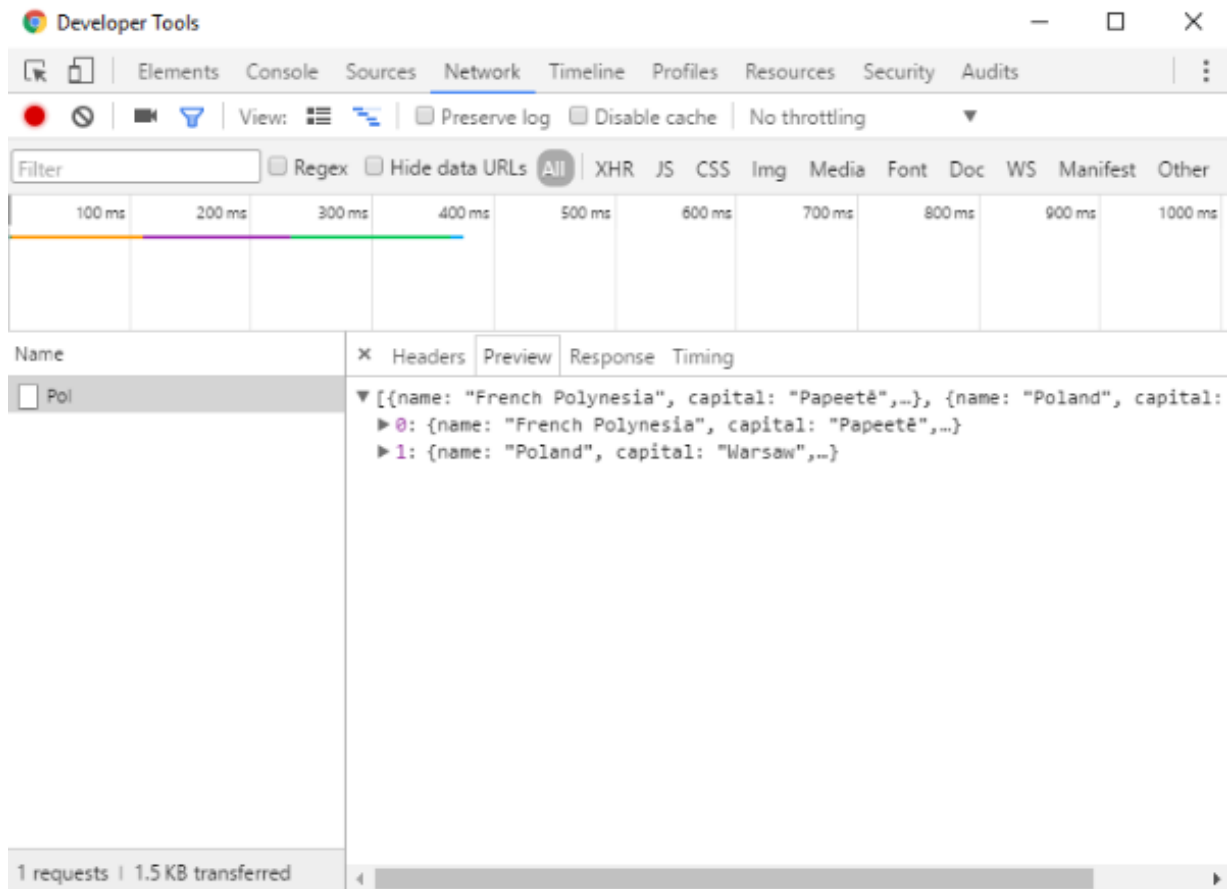
Pobieranie wyników

Weźmy się za implementację funkcji, która zajmie się logiką pokazywania listy krajów `showCountriesList()`. Pierwszą rzeczą, którą musimy zrobić, jest wyczyszczenie listy krajów po poprzednim zapytaniu. Nasza funkcja `showCountriesList()` ma w definicji parametr `resp`, czyli obiekt JSON, który przesyła do niej funkcja `fetch`. Możemy nazwać ten parametr dowolnie, nie istnieje żaden wzór, z którego trzeba skorzystać, ale w tym ćwiczeniu trzymajmy się tego nazewnictwa.

```
function showCountriesList(resp) {  
  countriesList.innerHTML = '';  
}
```

Zanim przejdziemy do kolejnego kroku, wywołajmy jakieś wyszukiwanie i sprawdźmy w zakładce Network, jak wygląda odpowiedź z serwera.

W przykładzie widzimy zapytanie, które miało postać 3 liter. Odpowiedź na nie wygląda następująco:



W odpowiedzi dostajemy kolekcję obiektów reprezentujących kraje.

Wyświetlanie wyników

Array udostępnia specjalną metodę `forEach()`, której działanie jest bardzo podobne do pętli `for`. Metoda `forEach()` ma za zadanie przeiterować po każdym elemencie tablicy `resp` (to tutaj jest ukryta nasza odpowiedź) i wykonać na każdym elemencie pewną funkcję.

Jako parametr takiej funkcji przyjmujemy każdy pojedynczy element z listy (parametr `item`) – wynika to z konstrukcji pętli `forEach()`

```
function showCountriesList(resp) {  
  countriesList.innerHTML = '';  
  resp.forEach(function(item) {  
    //Here is the code that will execute on each successive item  
  });  
}
```

Dopiszmy jeszcze do środka tej funkcji kod, za pomocą którego stworzymy nowy element listy, ustawimy jego tekst na taki, jaki kryje się w pojedynczym elemencie z kolekcji, którą dostajemy z serwera, a na koniec dodajmy element do listy krajów (`countriesList`).




```
resp.forEach(function(item){  
    var liEl = document.createElement('li');  
    liEl.innerText = item.name;  
    countriesList.appendChild(liEl);  
});
```

Dla chętnych

Odnajdź w pojedynczym obiekcie klucz, który trzyma informacje o stolicy kraju, a następnie wyświetl go na liście obok nazwy. Może uda Ci się skorzystać z większej ilości informacji i stworzyć na przykład coś takiego:



Przetestuj, czy wszystko działa prawidłowo i pomyśl, w jaki sposób można byłoby rozbudować aplikację o kolejne funkcjonalności. Może [dokumentacja API](#) udostępni jakieś ciekawe zasoby?

Podgląd zadania



<https://github.com/0na/>

Wyślij link ✓

12.7. Podsumowanie modułu



Mamy już całkiem sporą wiedzę na temat Ajaksa i API. Skorzystajmy z niej do rozbudowania aplikacji z poprzedniego modułu — Kanbana.

W obecnym wydaniu aplikacja jest co prawda użyteczna, ale tylko do momentu odświeżenia strony. Nasze dane nie są nigdzie przechowywane, co oznacza, że wszystkie kolumny i karteczki po odświeżeniu przepadają. Po dodaniu każdej kolumny czy też nowej karteczki lub przy innych podobnych akcjach powinniśmy komunikować się z serwerem.

Zastanówmy się przez moment nad tym, jakie mamy funkcjonalności i w jaki sposób możemy je powiązać z metodami HTTP:

- dodawanie kolumny — **POST**,
- usuwanie kolumny — **DELETE**,
- dodawanie karteczki — **POST**,
- usuwanie karteczki — **DELETE**,
- przenoszenie karteczki do innej kolumny — **PUT**,
- pokazywanie całości w formie tablicy — **GET**.

Mamy kilka funkcjonalności, którym śmiało można przypisać metody HTTP. Teraz wystarczy dogadać się z *back-endem* (częścią serwerową), żeby wykonali nam funkcjonalności pozwalające na zapis stanu naszej aplikacji do bazy. O samej implementacji nie musimy wiedzieć nic.

Jedyne, czego potrzebujemy to RESTowe API z małą instrukcją obsługi. Podstawowy adres URL wygląda tak (tak, mamy dla Ciebie API): <https://kodilla.com/pl/bootcamp-api>

Uwaga: jeśli otworzysz ten link w przeglądarce, zobaczysz stronę 404 — jest to tylko endpoint, na który wysyłamy zapytanie.



Zanim przejdziemy do omawiania szczegółów API, spójrzmy na przykładowe zapytanie wraz z odpowiedzią. Weźmy dla przykładu pokazywanie całej tablicy:

GET /board

Response:

```
{
  id: int,
  name: string,
  columns: [{
    id: int,
    name: string,
    cards: [{
      id: int,
      bootcamp_kanban_column_id: int,
      name: string
    }]
  }]
}
```

Pierwszym elementem jest wskazanie metody i endpointa, na który mamy wysłać zapytanie.

Całość powinna wyglądać tak:

GET <https://kodilla.com/pl/bootcamp-api/board>

Kolejnym fragmentem jest format odpowiedzi, jaką dostajemy z serwera (*Response*).

Czasami możemy również podejrzeć format samego zapytania (*Request*), czyli co musimy wysłać, żeby np. utworzyć nowy zasób (karteczkę lub kolumnę).

Ostatnia istotna kwestia w kontekście zapytań to uwierzytelnianie. Serwer musi przecież wiedzieć, czyją tablicę odesłać, albo na której umieścić kartę/kolumnę.

Na potrzeby tego zadania zostały stworzone dwie wartości, które musisz docelowo umieścić w nagłówku każdego zapytania:

X-Client-Id: 'X-Client-Id'
X-Auth-Token: 'X-Auth-Token'

Co z tym zrobić, dowiesz się niżej :)



Przygotowania do zadania

Nasz kod zaczyna się coraz bardziej rozrastać. Traci przez to na czytelności, a my za każdym razem musimy wpatrywać się w kod w poszukiwaniu najmniejszej zmiany.

Rozwiązaniem tego problemu jest podział kodu na mniejsze fragmenty. Zaczniemy więc od podziału całego pliku *script.js* na mniejsze kawałki.

Nie będziemy tego robić od zera — przygotowaliśmy kod, który będzie Twoim punktem wyjścia. Znajdziesz go [tutaj](#) — możesz *sforkować* lub wyeksportować projekt i zacząć pracę od tego momentu.



Modyfikowanie pliku App.js

Wyjaśniliśmy już sobie, że komunikacja z API będzie wymagała adresu URL i nagłówków (każdy użytkownik ma inne wartości nagłówków). Ustawmy więc zmienne, z których będziemy korzystać przy komunikacji z serwerem:

```
var baseUrl = 'https://kodilla.com/pl/bootcamp-api';
var myHeaders = {
  'X-Client-Id': 'X-Client-Id',
  'X-Auth-Token': 'X-Auth-Token'
};
```

baseUrl to po prostu podstawowy adres serwera, który wystawia nam endpointy, z których będziemy korzystać przy komunikacji.



Każde zapytanie, które wykonamy, będzie musiało mieć w sobie nagłówki (**myHeaders**).

Nagłówek **Content-Type** jest potrzebny do zakomunikowania serwerowi, że dane wysłane są w formacie JSON. Jeśli zabraknie takiej informacji, to serwer nie będzie potrafił obsłużyć naszego zapytania.

Wróćmy do dokumentacji pobierania tablicy:

GET `/board`

Response:

```
{
  id: int,
  name: string,
  columns: [{
    id: int,
    name: string,
    cards: [{
      id: int,
      bootcamp_kanban_column_id: int,
      name: string
    }]
  }]
}
```

Musimy użyć metody GET, a endpoint to `/board`. Ok, dodajmy więc funkcję odpytującą serwer o zasób tablicy. Powinno to wyglądać w następujący sposób:

```
fetch(baseUrl + '/board', { headers: myHeaders })
  .then(function(resp) {
    return resp.json();
  })
  .then(function(resp) {
    setupColumns(resp.columns);
  });
```

Po odebraniu odpowiedzi możemy przejść do tworzenia kolumn, stąd funkcja `setupColumns(response.columns)`.

UWAGA: skąd wiadomo, że `response.columns` kryje w sobie jakieś informacje? Po pierwsze, mamy dokumentację, a poza tym zawsze możemy skorzystać z narzędzi developerskich i sprawdzić, jak to wygląda w *Network*.



Tworzenie kolumn

Przejdźmy teraz do implementacji funkcji `setupColumns()`. To, co funkcja musi wykonać, to stworzenie tylu kolumn, ile dostaliśmy w odpowiedzi z serwera, następnie zaś każdą z nich musi przypiąć do tablicy (tej, którą widzimy na stronie).

Zacznijmy od stworzenia znajomej już konstrukcji `forEach()`.

```
function setupColumns(columns) {  
  columns.forEach(function(column) {  
  
    });  
}
```

Oznacza ona, że dla każdego elementu (kolumny) znajdującej się w tablicy `columns` ma się wykonać pewna funkcja. Ta funkcja, jak już sobie powiedzieliśmy, musi stworzyć kolumnę i dodać ją do tablicy:

```
function setupColumns(columns) {  
  columns.forEach(function(column) {  
    var col = new Column(column.id, column.name);  
    board.addColumn(col);  
  });  
}
```

Zauważ, że tworzenie naszej kolumny jest trochę inne, niż było do tej pory. Do jej stworzenia potrzebujemy dwóch parametrów: **ID** i **name**.

Potrzebujemy ID dlatego, że jest tworzone za nas przez serwer za każdym razem, gdy tworzymy nowy element. Nie musimy więc generować losowego ID tak jak do tej pory. Oznacza to, że możemy całkowicie pozbyć się funkcji `randomString()`. Do kosza!





Karty w kolumnie

Mamy utworzone kolumny, teraz potrzebujemy jeszcze kart. Informację do nich również dostajemy w odpowiedzi z endpointa `/board`. Dopiszmy funkcję, która w taki sam sposób, jak ustawiała kolumny, ustawi nam karty. Oczywiście w odpowiednie kolumny!

```
function setupColumns(columns) {  
  columns.forEach(function (column) {  
    var col = new Column(column.id, column.name);  
    board.addColumn(col);  
    setupCards(col, column.cards);  
  });  
}
```

Przejdźmy do implementacji funkcji `setupCards()`. Do funkcji przekazujemy kolumnę, do której mają zostać przyłączone karty, które należy stworzyć:

```
function setupCards(col, cards) {  
  
}
```

Teraz musimy, podobnie jak w przypadku funkcji `setupColumns()`, przeiterować po wszystkich kartach, utworzyć je i dodać do odpowiedniej kolumny:

```
function setupCards(col, cards) {  
  cards.forEach(function (card) {  
    var cardObj = new Card(card.id, card.name);  
    col.addCard(cardObj);  
  });  
}
```

Zauważ, że tworzenie nowej karty (`new Card()`) znowu jest rozszerzone o kolejne parametry. Wynika to z tego, że tworząc nowe obiekty musimy dopasować się do odpowiedzi serwera.

Ostatnim elementem jest dodanie karty do kolumny za pomocą metody `createCard()`, którą tworzyliśmy w module 11 (można ją znaleźć w pliku *Column.js*).

To koniec modyfikowania pliku *App.js*. Są tutaj jednak pewne zmiany, których trzeba dokonać w plikach *Card.js* i *Column.js*. Przejdźmy najpierw do pliku *Column.js*.

Modyfikowanie pliku Column.js

Zacznijmy od samej klasy *Column*. W funkcji konstruującej musimy dodać jeszcze jeden parametr przed `name` — parametr `id`.

```
function Column(id, name) {  
  
}
```

Jak już mówiliśmy, `id` nie będzie już generowane losowo, tylko przez serwer, więc musimy podmienić ustawianie wartości początkowej `id`. Ustawmy też wartość domyślną dla nazwy kolumny, jeśli będzie pusta:

```
function Column(id, name) {  
  this.id = id;  
  this.name = name || 'No name given';  
}
```

Operacje, które możemy wykonać w obrębie kolumny, to dodanie nowej karty lub usunięcie kolumny.



Usuwanie kolumny

Zacznijmy od usuwania kolumny i połączenia tego z serwerem.

Spójrzmy najpierw na dokumentację:

```
DELETE /column/{id}
-----
Request:
{id}: int - id column, we want to delete
-----
Response:
{
  id: int
}
```

W celu usunięcia kolumny należy wysłać zapytanie metodą **DELETE** na endpoint `/column/{id_kolumny}`.

Dodajmy tę funkcjonalność do prototypu Column do metody `removeColumn`:

```
removeColumn: function() {
  var self = this;
  fetch(baseUrl + '/column/' + self.id, { method: 'DELETE', headers:
    .then(function(resp) {
      return resp.json();
    })
    .then(function(resp) {
      self.element.parentNode.removeChild(self.element);
    });
}
```

Rzeczą, na którą należy zwrócić uwagę, jest dobranie adresu URL. Usuwamy kolumnę, która ma określony id. Użycie funkcji w bloku `.then()` spowodowało utratę kontekstu, stąd `self`. Dopiero po uzyskaniu potwierdzenia z serwera, że element został usunięty, możemy usunąć go również z widoku. Stąd użycie metody `self.element.parentNode.removeChild(self.element);`.

Dodawanie kolumny

Nieco trudniejszą sprawą jest dodawanie nowej kolumny. Do tej pory nasza funkcja wyglądała tak:



```
this.element.querySelector('.column').addEventListener('click', function(event) {
    if (event.target.classList.contains('btn-delete')) {
        self.removeColumn();
    }

    if (event.target.classList.contains('add-card')) {
        self.addCard(new Card(prompt("Enter the name of the card")));
    }
});
```

Nazwę karty musimy wydzielić do osobnej zmiennej. Będzie nam to potrzebne do wykonania zapytania do serwera przy tworzeniu nowego zasobu.

Zróbmy to w następujący sposób:

```
if (event.target.classList.contains('add-card')) {
    var cardName = prompt("Enter the name of the card");
    event.preventDefault();
    self.addCard(new Card(cardName));
}
```

Kolejnym krokiem będzie wykonanie zapytania do serwera. Spójrzmy na dokumentację:

POST /card

Request:

name: string - the name of the card we create

bootcamp_kanban_column_id: int - the id of the column to which the card is added

Response:

```
{
  id: int
}
```

Jak widzisz, musimy wykonać zapytanie POST na endpointa **/card**, do tego w ciele zapytania musimy wysłać dane potrzebne do stworzenia nowej karty: **name** i **bootcamp_kanban_column_id**. W odpowiedzi dostajemy **id** utworzonej kolumny.

Napišmy sobie to zapytanie:



```
if (event.target.classList.contains('add-card')) {  
  var cardName = prompt("Enter the name of the card");  
  event.preventDefault();  
  
  fetch(baseUrl + '/card', {  
    method: 'POST',  
    body: {  
      //body query  
    }  
  })  
  .then(function(res) {  
    return res.json();  
  })  
  
  .then(function() {  
    //create a new client side card  
  });  
  
  self.addCard(new Card(cardName));  
}
```

Mamy nie do końca sformułowane zapytanie. Musimy dodać do niego dane, które wysyłamy, i uzupełnić funkcję, która ma się wykonać po utworzeniu nowej kolumny na serwerze.

Spójrzmy:

```
var data = new FormData();  
data.append('name', cardName);  
data.append('bootcamp_kanban_column_id', self.id);  
  
fetch(baseUrl + '/card', {  
  method: 'POST',  
  headers: myHeaders,  
  body: data,  
})  
  .then(function(res) {  
    return res.json();  
  })  
  .then(function(resp) {  
    var card = new Card(resp.id, cardName);  
    self.addCard(card);  
  });
```



Dodaliśmy dwa parametry `name` i `bootcamp_kanban_column_id` (zgodnie z dokumentacją). Znowu tracimy kontekst przy użyciu funkcji w metodzie `.then()`, więc musimy skorzystać ze zmiennej `self`.

Warto zauważyć również pojawienie się nowej konstrukcji `var data = new FormData()`; . Jest to wbudowany w przeglądarkę interfejs dostarczający nam możliwość tworzenia par klucz-wartość, reprezentujących pola formularza. Tak przygotowany formularz wysyłamy do naszego serwera.

W funkcji, która ma się wykonać po poprawnym zapytaniu i utworzeniu nowej kolumny, tworzymy nową kartę i dodajemy ją do kolumny. Zwróć uwagę, że **ID** potrzebne do utworzenia karty pobieramy z odpowiedzi serwera.

Skończyliśmy modyfikację pliku *Column.js*. Możemy teraz zabrać się za plik *Card.js*, który też musimy nieco zmodyfikować.

Modyfikowanie pliku Card.js

Tutaj, podobnie jak w pliku *Column.js*, musimy trochę zmodyfikować funkcję konstruującą. Usuwamy `description` i wpisujemy `id` i `name`. Musimy nanieść te zmiany ze względu na odpowiedź, którą dostajemy z serwera:

```
function Card(id, name) {  
  
}
```

Zmodyfikujmy też ustawianie początkowych wartości atrybutów. Metoda `randomString()` jest już usunięta, więc nie potrzebujemy jej, a `this.description` podmieniamy na `name`:

```
this.id = id;  
this.name = name || 'No name given';
```

Jest jeszcze jedno miejsce, w którym musimy zmienić `description` na `name` — w funkcji `generateTemplate()`.

Należy zmienić:



```
generateTemplate('card-template', { description: this.description },
```

na

```
generateTemplate('card-template', { description: this.name }, 'li');
```

Usuwanie karty

Jedyną czynnością, którą możemy wykonać w kontekście karty, jest jej usunięcie. Zmodyfikujmy tę funkcję, aby mogła wysyłać zapytanie o usunięcie zasobu. Spójrzmy na dokumentację:

```
DELETE /card/{id}
-----
Request:
{id}: int - id card we want to remove
-----
Response:
{
  id: int
}
```

W celu usunięcia karty z serwera należy wysłać zapytanie metodą DELETE na endpointa `/card/{id_karty}`. Analogicznie do poprzednich metod użyjemy do tego fetcha:

```
removeCard: function() {
  var self = this;

  fetch(baseUrl + '/card/' + self.id, { method: 'DELETE', headers: my
    .then(function(resp) {
      return resp.json();
    })
    .then(function(resp) {
      self.element.parentNode.removeChild(self.element);
    })
  }
}
```

Chyba nie trzeba już tłumaczyć tego zapisu. W bardzo podobny sposób wyglądało usuwanie kolumny :)



Modyfikowanie pliku Board.js

Ostatnim elementem naszej układanki jest tablica. Tutaj nie musimy już niczego specjalnie zmieniać. Możemy przejść od razu do modyfikowania funkcjonalności dodawania nowej kolumny.

Dodawanie kolumny

Spójrzmy na prawidłowy sposób dodania nowej kolumny:

```
=====
POST /column
-----
Request:
name: string - name of the column to create
-----
Response:
{
  id: int
}
```

Nasze zapytanie powinno zostać wykonane przy użyciu metody POST na endpoint `/column`. Ciało żądania powinno zawierać jedynie nazwę kolumny, którą chcemy utworzyć. Dodajmy obsługę zapytania w miejscu, gdzie była tworzona nowa kolumna (w nasłuchiowaniu na kliknięcie przycisku *Dodaj kolumnę*).



```
document.querySelector('#board .create-column').addEventListener('click', function() {
  var name = prompt('Enter a column name');
  var data = new FormData();

  data.append('name', name);

  fetch(baseUrl + '/column', {
    method: 'POST',
    headers: myHeaders,
    body: data,
  })
  .then(function(resp) {
    return resp.json();
  })
  .then(function(resp) {
    var column = new Column(resp.id, name);
    board.addColumn(column);
  });
});
```

W taki sam sposób, jak tworzyliśmy nową kartę w kolumnie, musimy utworzyć nową kolumnę na tablicy. W danych, które wysyłamy do serwera, jest jedynie (jak już wspomnieliśmy) nazwa kolumny.

Samo jej tworzenie wykonujemy dopiero wtedy, gdy mamy pewność, że kolumna została poprawnie utworzona na serwerze. Dlatego musimy przenieść logikę odpowiadającą za tworzenie nowej kolumny do środka funkcji `.then()`

Zwróć uwagę, że tworzenie kolumny również rozbito na dwie części:

```
var column = new Column(response.id, columnName);
board.createColumn(column);
```

Najpierw tworzona jest kolumna przy użyciu id, które dostajemy w odpowiedzi z serwera oraz nazwy kolumny, o którą nas zapytano, a dopiero potem tworzymy kolumnę na tablicy.

Ostatnie szlify



Ostatnim krokiem jest usunięcie niepotrzebnego tworzenia elementów przy starcie aplikacji. Wchodzimy do pliku *App.js* i usuwamy:

```
// CREATING NEW COLUMN EXAMPLES
var todoColumn = new Column('TO DO');
var doingColumn = new Column('DOING');
var doneColumn = new Column('DONE');

// ADD COLUMN TO TABLES

board.createColumn(todoColumn);
board.createColumn(doingColumn);
board.createColumn(doneColumn);

// CREATING NEW CARDS
var card1 = new Card('New task');
var card2 = new Card('Create kanban boards');

// ADDING CARDS TO COLUMN
todoColumn.createCard(card1);
doingColumn.createCard(card2);
```

Nie było tak źle, prawda? Teraz możesz się cieszyć własnym w pełni funkcjonującym kanbanem :)

Zadanie: Ożywiamy kanban!

W tym zadaniu zajmiemy się modyfikacją naszego poprzedniego projektu — tablicy Kanban. Specjalnie na potrzeby tego zadania przygotowaliśmy API, do którego będziesz łączyć się, żeby zapisywać swoje dane z tablicy. Dzięki temu dane będą dostępne również po odświeżeniu strony, a Ty będziesz mieć swoje małe Trello, które będzie idealnym projektem do portfolio.

Pamiętaj, żeby po skończeniu prac nad projektem wstawić swój kod na GitHuba i udostępnić go mentorowi do oceny. Powodzenia! :)

Twoje ID oraz token na potrzeby komunikacji z naszym API:

```
X-Client-Id: 3851
X-Auth-Token: 5905e6ef19a34924ae6c9414a9eb4386
```



Dla chętnych

Jak się okazuje, serwer udostępnia nam również endpointy służące do aktualizowania np. bieżącej karty przy zmianie nazwy albo zmiany kolumny, do której karta przynależy. Dostępne są jedynie fragmenty dokumentacji. Spójrz na nie i spróbuj zaimplementować:

- zmianę nazwy karty/kolumny,
- przenoszenie karty do innej kolumny.

Modyfikowanie istniejącej karty

```
PUT /card/{id}
```

Request:

{id}: int - id card we want to edit

name: string - new name card

bootcamp_kanban_column_id: int - the column id to which we want to move card

Response:

```
{
```

```
  id: int
```

```
}
```

Modyfikowanie istniejącej kolumny

```
PUT /column/{id}
```

Request:

{id}: int - the column id we want to edit

name: string - new column name

Response:

```
{
```

```
  id: int
```

```
}
```

Podgląd zadania



<https://github.com/0na/>

Wyślij link ✓

12.9. Egzamin sprawdzający



Minęła już spora część kursu, po której warto sprawdzić swoją wiedzę. W związku z tym zapraszamy Cię do wzięcia udziału w egzaminie sprawdzającym.

Opis platformy egzaminacyjnej

Egzamin odbywa się na platformie rekrutacyjnej Devskiller. Pracodawcy często korzystają z tego rodzaju platform testowych w trakcie rekrutacji programistów. Wybraliśmy to rozwiązanie, aby lepiej przygotować Cię do procesów rekrutacyjnych z którymi spotkasz się już niedługo. Egzamin odbywa się przez internet i wykonujesz go samodzielnie (nie łączysz się z żadnym Mentorem ani Egzaminatorem).

Etapy egzaminu

1. Pytania testowe, część 1 - 7 pytań po 3 minuty (w sumie 21 minut),
2. **Przerwa - 10 minut,**
3. Pytania testowe, część 1 - 7 pytań po 3 minuty (w sumie 21 minut),
4. **Przerwa - 10 minut,**
5. Przykładowe zadanie praktyczne - 30 minut,
6. Zadanie praktyczne z CSS - 15 minut,
7. **Przerwa - 10 minut,**
8. Zadanie praktyczne z JS - 30 minut.

Wszystkie czasy podane powyżej są limitem czasu. Każde pytanie, zadanie czy prze
możesz zakończyć szybciej. Z tego względu, egzamin nie powinien zająć Ci więcej ni
1,5 godziny, ale na wszelki wypadek **zarezerwuj sobie co najmniej 2,5 godziny**.



Rodzaje zadań

1. Pytania testowe, w których należy zaznaczyć wszystkie poprawne odpowiedzi (jedną lub więcej).

Question#2

Select correct answers.

☐ option 2

☒ option 4

☐ option 3

☒ option 1

2. Zadania praktyczne, w których otrzymasz fragment kodu do poprawienia lub uzupełnienia. Pierwsze z zadań praktycznych jest przykładowe - nie otrzymasz za nie żadnych punktów, ale będziesz mieć okazję zaznajomić się z edytorem oraz strukturą projektu.

Save Revert Download Clone Synchronize Build

Project files

Right click on node for context menu

- src
 - main
 - java
 - com
 - devskiller
 - calculator
 - Calculator.java

test

README.md

devskiller.marker

pom.xml [Read only]

Calculator.java

```
1 package com.devskiller.calculator;
2
3 public class Calculator {
4
5     public int add(int a, int b) {
6         return 0;
7     }
8
9     public int subtract(int a, int b) {
10        return 0;
11    }
12
13    public int multiply(int a, int b) {
14        return 0;
15    }
16    public int divide(int a, int b) {
17        return 0;
18    }
19 }
20 }
```

Mode: Java Show keyboard shortcuts

> Build console Test results 0 of 4

shouldMultiplyTwoNumbers(com.devskiller.calculator.BasicTest): expected:<[6]> but was:<[0]>
shouldAddTwoNumbers(com.devskiller.calculator.BasicTest): expected:<[5]> but was:<[0]>

Tests run: 4, Failures: 4, Errors: 0, Skipped: 0

[ERROR] There are test failures.

Please refer to target/surefire-reports for the individual test results.

[INFO] -----

[INFO] BUILD SUCCESS

[INFO] -----

[INFO] Total time: 2.048 s

[INFO] Finished at: 2017-08-17T09:09:24+00:00

[INFO] Final Memory: 16M/203M

[INFO] -----

Przydatne informacje

1. Zadbaj o czas i miejsce, które pozwolą Ci się skupić na egzaminie. Jeśli to możliwe, wycisz swój telefon i uprzedź bliskich aby w tym czasie Ci nie przeszkadzali.
2. Nie można wracać do wcześniejszych zadań, więc uważnie czytaj polecenia i zastanów się nad odpowiedzią zanim przejdziesz do kolejnego zadania. Każde zadanie ma limit czasu.
3. W niektórych pytaniach testowych jest kilka poprawnych odpowiedzi - zaznacz wszystkie.
4. Nie przerywaj egzaminu - możesz do niego podejść tylko raz. W trakcie egzaminu możesz robić przerwy tylko pomiędzy etapami wymienionymi poniżej.
5. Jeśli przypadkiem zamkniesz egzamin, możesz ponownie go otworzyć za pomocą linka w mailu z zaproszeniem na egzamin.
6. Wyniki otrzymasz w ciągu tygodnia od podejścia do egzaminu.

Zapis na egzamin

Do przystąpienia do egzaminu niezbędne jest zapisanie się na egzamin.

Pamiętaj:

- na egzamin możesz zapisać się tylko raz,
- link z zaproszeniem otrzymasz najpóźniej następnego dnia roboczego.

Przed zapisaniem się na egzamin:

1. zaplanuj czas, w którym są najmniejsze szanse, że ktoś będzie Ci przeszkadzał,
2. zaplanuj miejsce, które pozwoli Ci skupić się na egzaminie,
3. zaplanuj sprzęt (laptop, internet), który nie będzie powodował problemów.

Najpóźniej **dzień przed planowanym podejściem do egzaminu** wypełnij poniższy formularz.



Zapis na egzamin z podstaw Web

W celu udziału w egzaminie podaj swój adres mailowy, powiązany z kontem Kodilla.com

***Wymagane**

Adres email *

Twoja odpowiedź

PRZEŚLIJ

Nigdy nie podawaj w Formularzach Google swoich haseł.

Formularze Google

Ten formularz został utworzony w domenie Codemy S.A..



Powodzenia!

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

✓ Zapoznałem(a)m się!



Regulamin

Polityka prywatności

© 2019 Kodilla

