REST API

Trener: Michał Michalczuk

Gdańsk, 28 października 2018 roku

Plan na dzisiaj

- API
- Czym jest REST API
- REST API a adresy URL / HTTP verbs
- Identyfikacja zasobów
- Co dostajemy w odpowiedzi
- Jak informować o błędach
- Bardziej złożone zapytania i customowe nagłówki
- Uwierzytelnianie
- Wersjonowanie API - czego możemy się spodziewać?

Po co testerowi REST API?

- to źródło danych dla aplikacji
  - mobilnych
  - webowych
  - desktopowych
- to nasz produkt
- to opcja na integrację naszego produktu (czyli też nasz produkt)

Jak testować REST API?

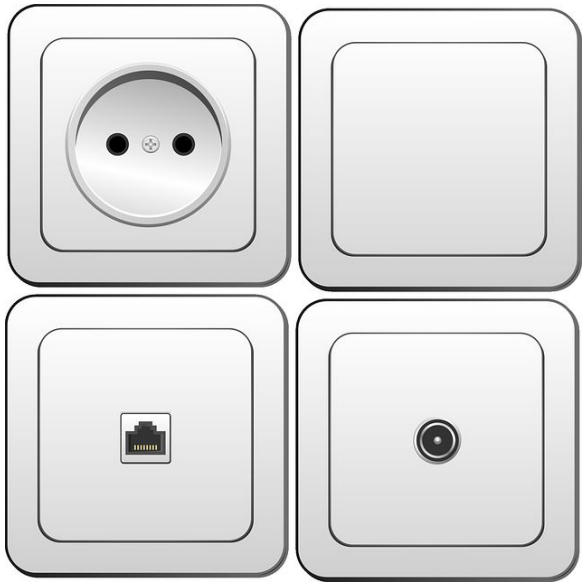
Narzędzia do testowania:

- REST Assured
- JMeter
- SOAP UI
- PostMan

Po co je testować?

- wczesne wyłapanie błędów ("przed" UI)

API i REST API

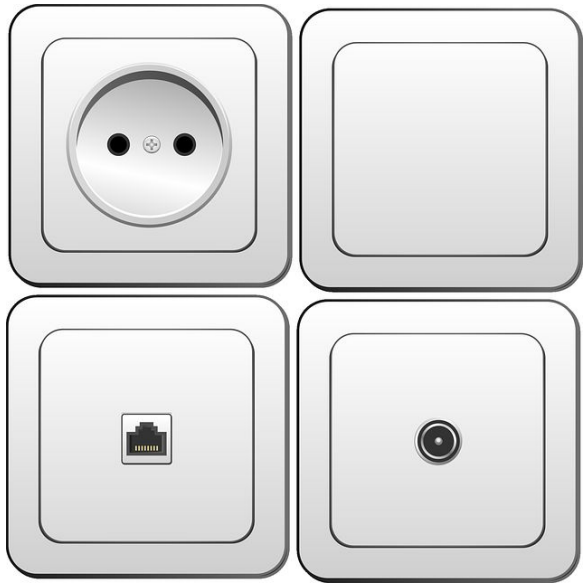


Application **P**rogramming **I**nterface

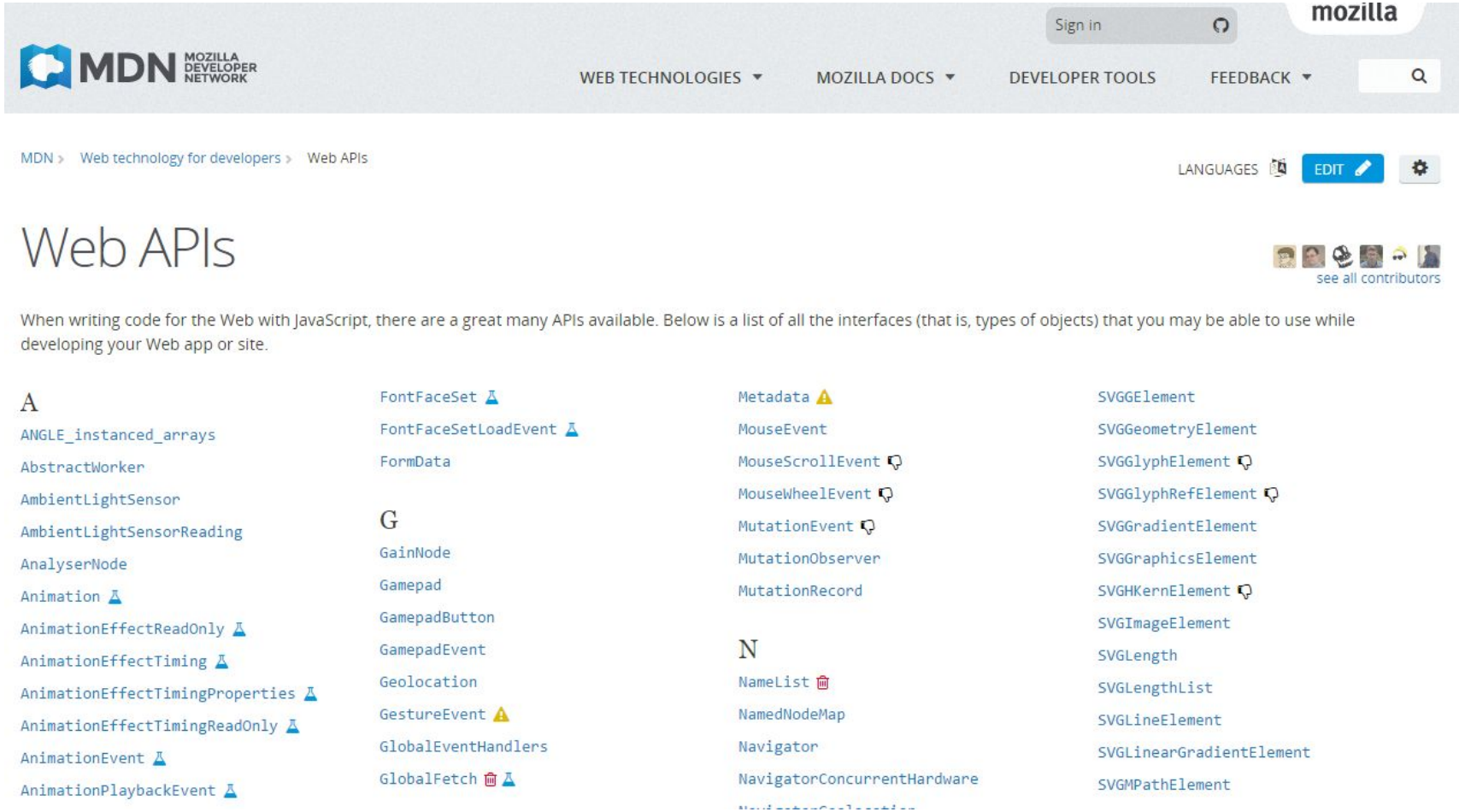
Zestaw

- reguł
- opisów
- zasad

Jak programy powinny się ze sobą komunikować.



API w JavaScript - https://developer.mozilla.org/en-US/docs/Web/API



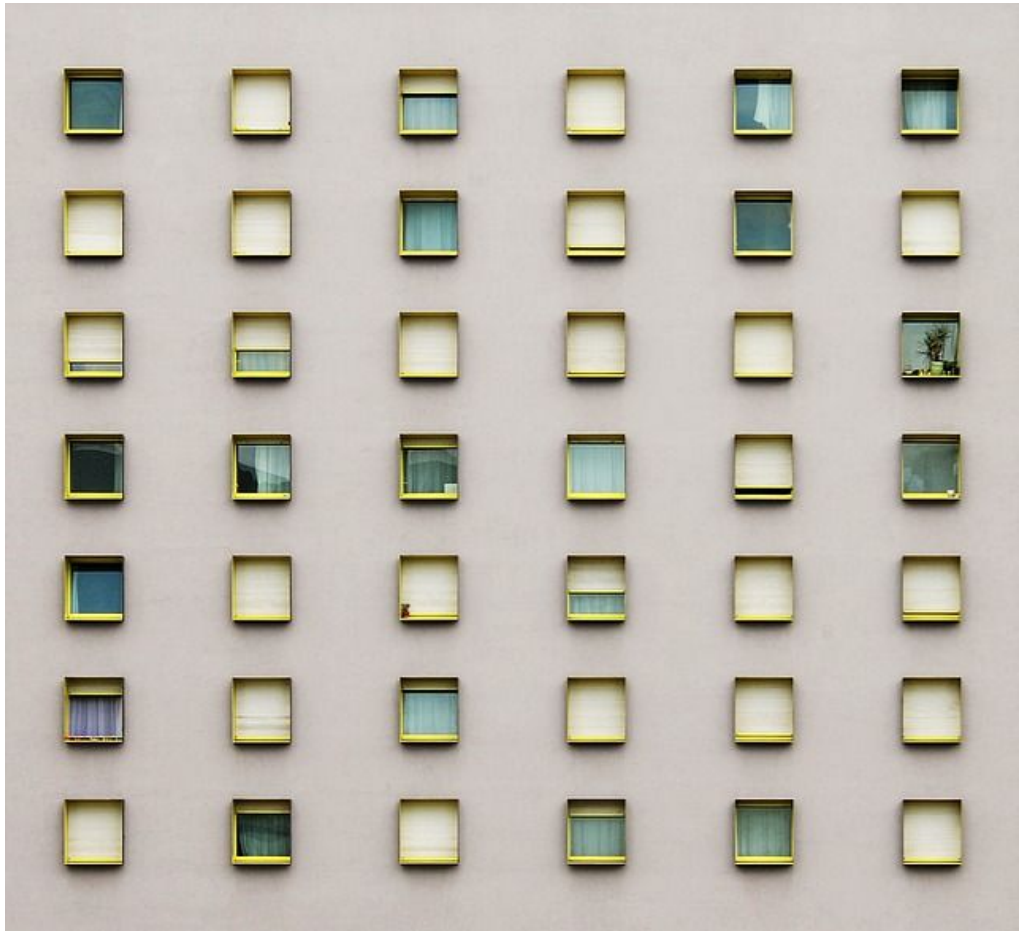
REST

Styl architektury oprogramowania.

REpresentational

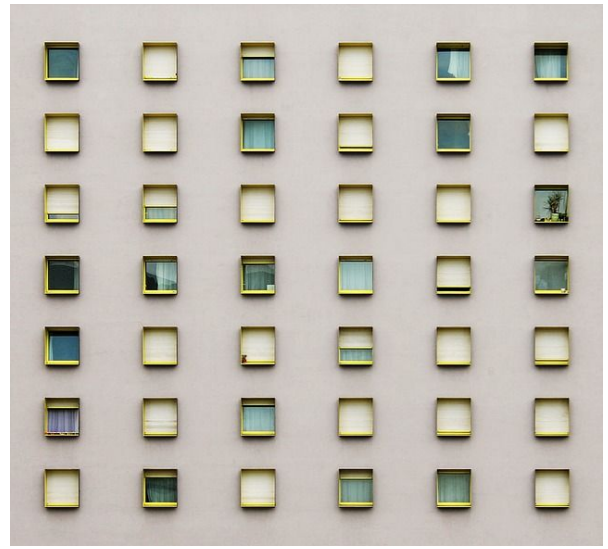
State

Transfer

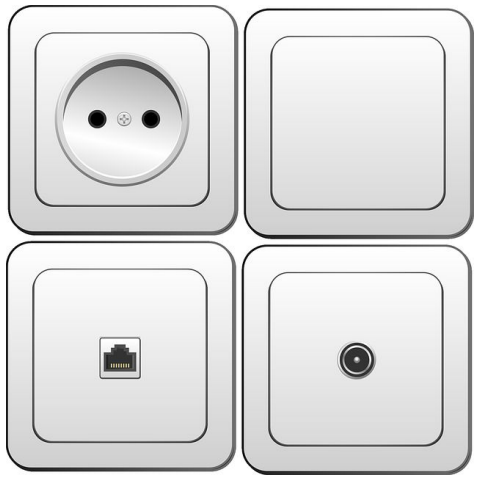


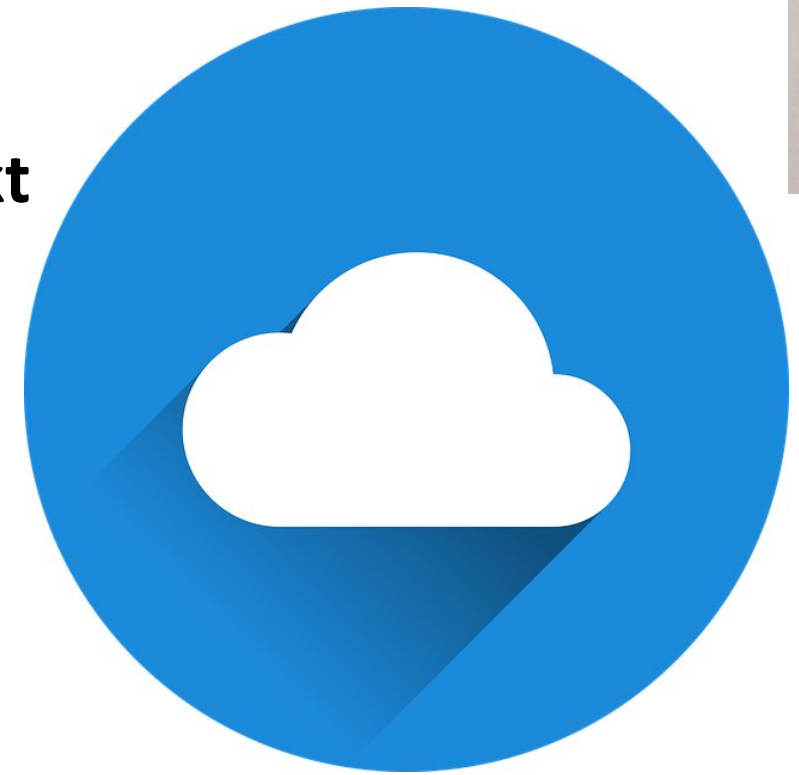
REST API

Api zaprojektowane w stylu REST



- **najczęściej - HTTP**
- **klient - serwer**
- **Zorientowane na zasoby**
- **URL - identyfikator**
- **Format danych - kontrakt**

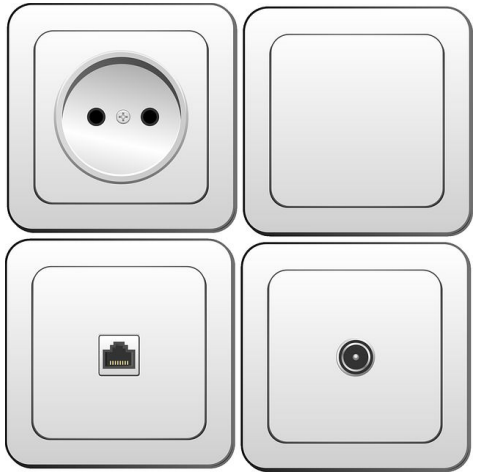




REST API - dostęp po HTTP to nie wszystko

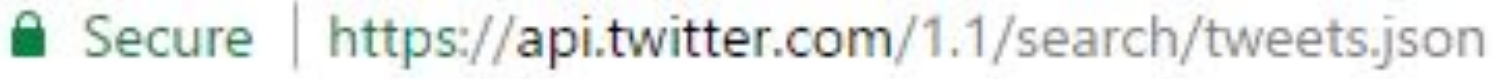
Jakie są reguły REST API?

- **Zdefiniowane zasoby**
- **Identyfikacja zasobów - po URL**
- **Manipulacja przez reprezentacje - format**
- **Opisowe wiadomości**
- **Kody HTTP**
- **Format URL**



Wszystko będzie się przewijać na zajęciach

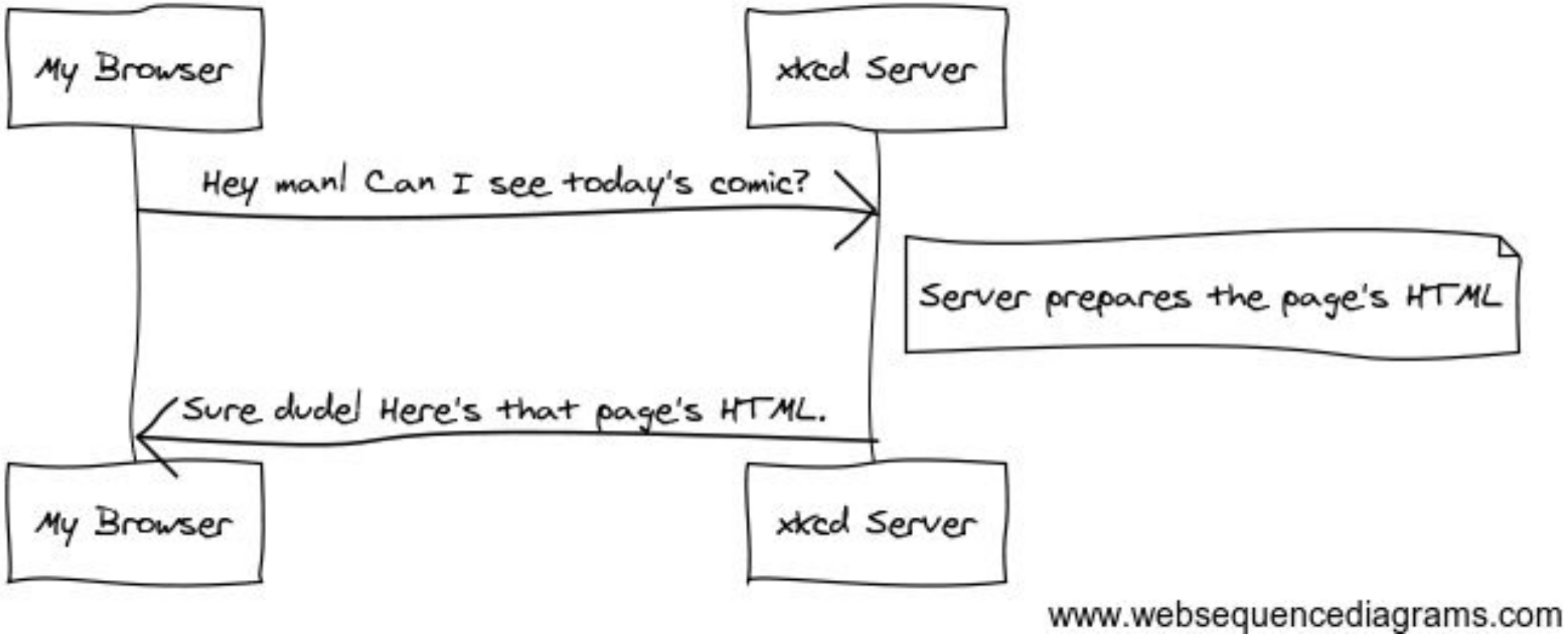
REST API - adresy URL



URL – jak go czytać? | Przypomnienie

Źródło: http://antezeta.com/news/campaign-tracking

Client - Server | Przypomnienie

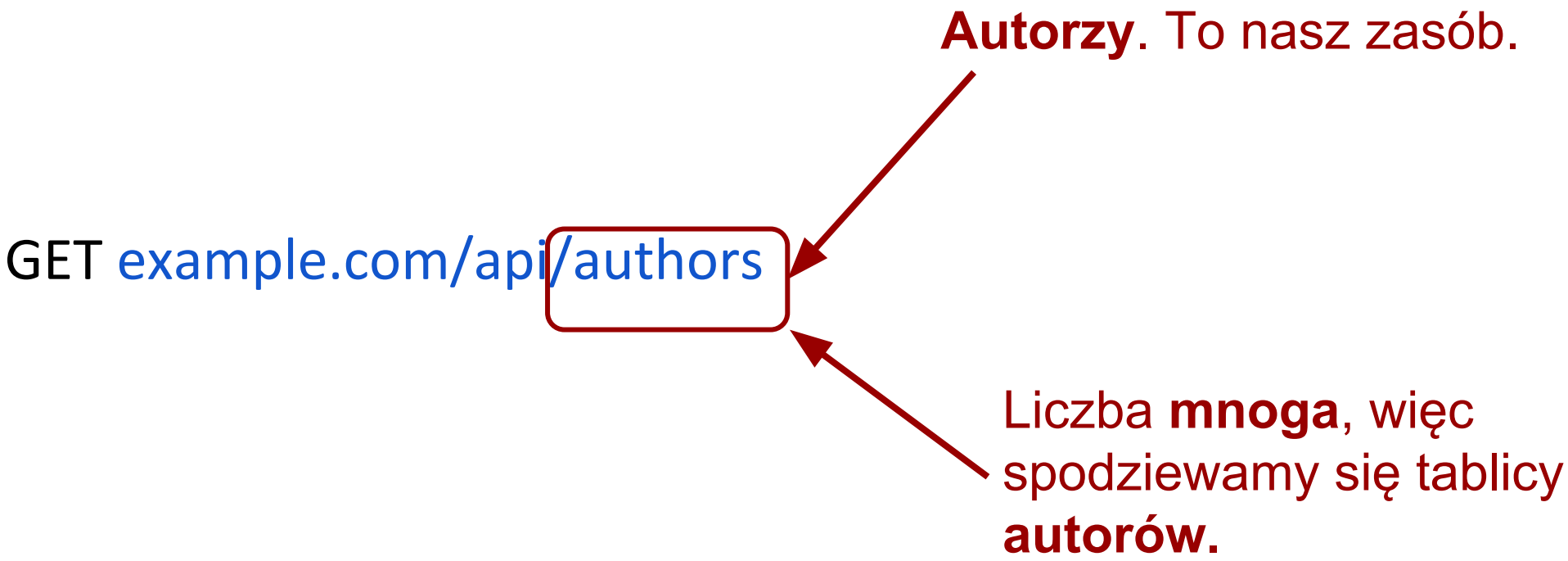


Źródło: http://symfony-docs.pl/_images/http-xkcd.png

Z czego składa się request? | Przypomnienie

- URL
- HTTP verb (method)
- headers

Zasób - czyli co chcemy dostać



Co jest tutaj zasobem i skąd to wiemy?

Czego możemy się spodziewać w odpowiedzi?

Zasób - czyli co chcemy dostać

GET example.com/api/authors

Nasza odpowiedź - zgodnie z przewidywaniami.

Choć ma tylko 1 element.

Zasób - czyli co chcemy dostać

GET example.com/api/blogs

GET example.com/api/authors

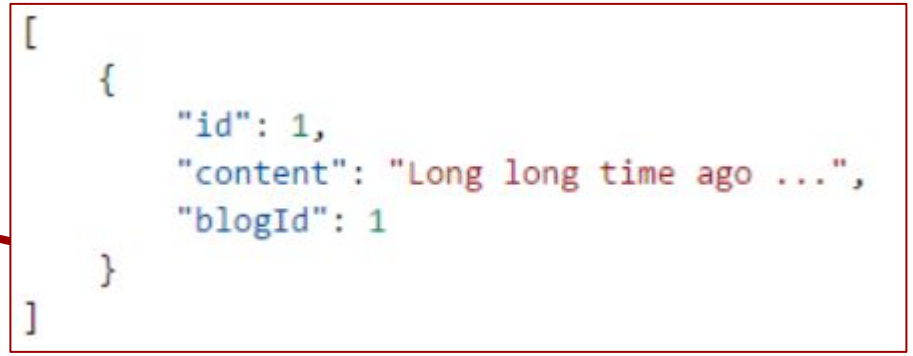
GET example.com/api/posts

Odpowiedzi API

GET example.com/api/blogs

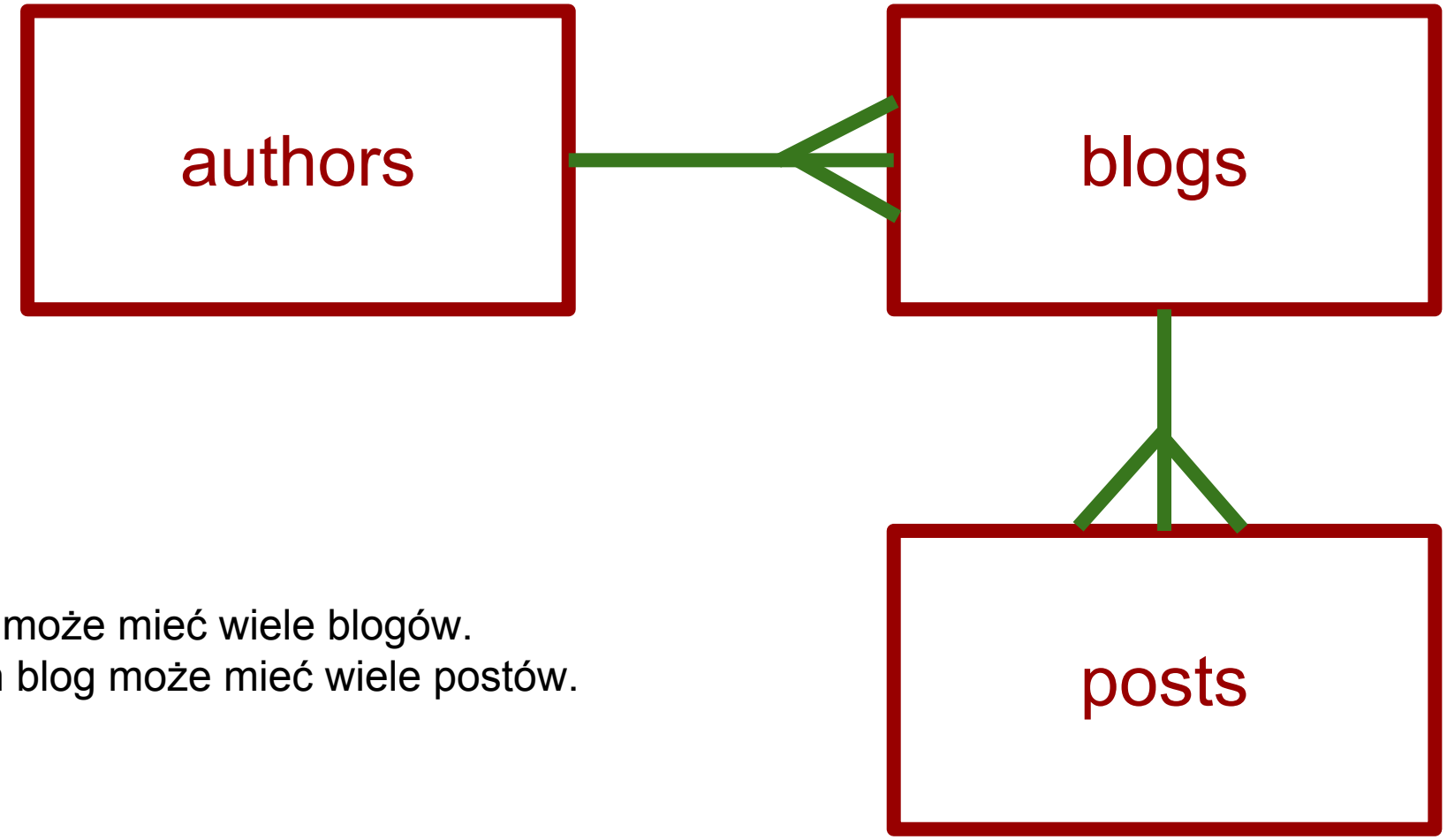
GET example.com/api/authors

GET example.com/api/posts



Hmm te dane wyglądają na **połączone** ze sobą

Modelujemy rzeczywistość



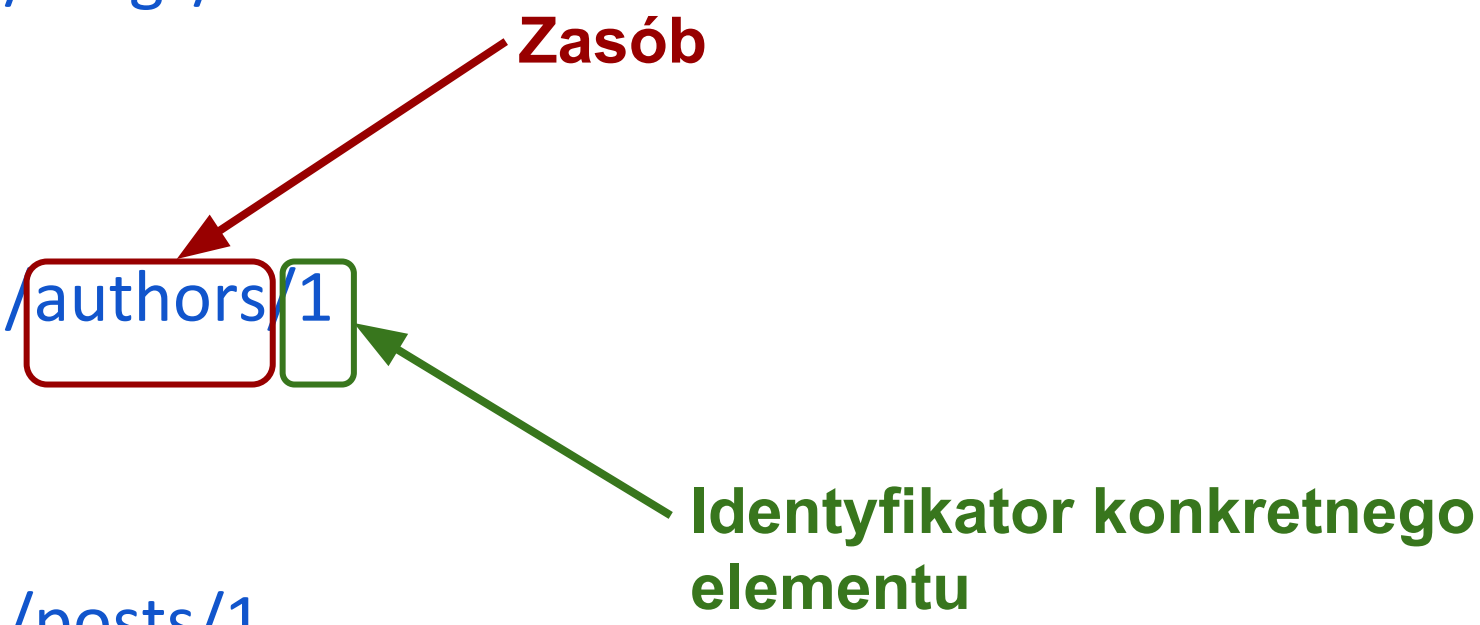
Autor może mieć wiele blogów.
Jeden blog może mieć wiele postów.

Bardzo konkretny element - dodajmy identyfikator

GET example.com/api/blogs/2

GET example.com/api/authors/1

GET example.com/api/posts/1



Bardzo konkretny element - dodajmy identyfikator

GET example.com/api/blogs/2


```
{
    "id": 2,
    "title": "Travel with code",
    "creationYear": "2017",
    "authorId": 1
}
```


GET example.com/api/authors/1


```
{
    "id": 1,
    "name": "Michal Michalczuk"
}
```


GET example.com/api/posts/1


```
{
    "id": 1,
    "content": "Long long time ago ...",
    "blogId": 1
}
```

A więc mamy **reprezentację** - to format/kształt danych oraz format w jakim przychodzą

JSON

```
[
   {
      "id": 1,
      "content": "First item"
   },
   {
      "id": 2,
      "content": "Another item"
   }
]
```

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <element>
    <content>First item</content>
    <id>1</id>
  </element>
  <element>
    <content>Another item</content>
    <id>2</id>
  </element>
</root>
```

Format danych pobranych dla listy jak i dla elementu powinien być taki sam

Ale różnie bywa.

Różne API powstają.

Czasem różne reguły REST API są łamane.

Wszystko zależy od projektu.

Ok, to chcę pobrać blogi danego autora

Idziemy głębiej

GET example.com/api/authors/1/blogs

- Zasób: **blogi** wybranego autora
- Bazujemy na relacji
- Spodziewamy się takiej odpowiedzi
- I tak wszystko zależy od serwera


```
[
    {
        "id": 1,
        "title": "My dogs blog",
        "creationYear": "2015",
        "authorId": 1
    },
    {
        "id": 2,
        "title": "Travel with code",
        "creationYear": "2017",
        "authorId": 1
    }
]
```

Jak inaczej mogłyby być dostępne blogi danego autora?

GET example.com/api/blogs?authorId=1

- Zasób: **blogi**
- Filtrujemy dane przez query params
- I tak wszystko zależy od serwera


```
[
    {
        "id": 1,
        "title": "My dogs blog",
        "creationYear": "2015",
        "authorId": 1
    },
    {
        "id": 2,
        "title": "Travel with code",
        "creationYear": "2017",
        "authorId": 1
    }
]
```

Więc do czego służą query params w REST API?

Do przekazywania extra informacji.

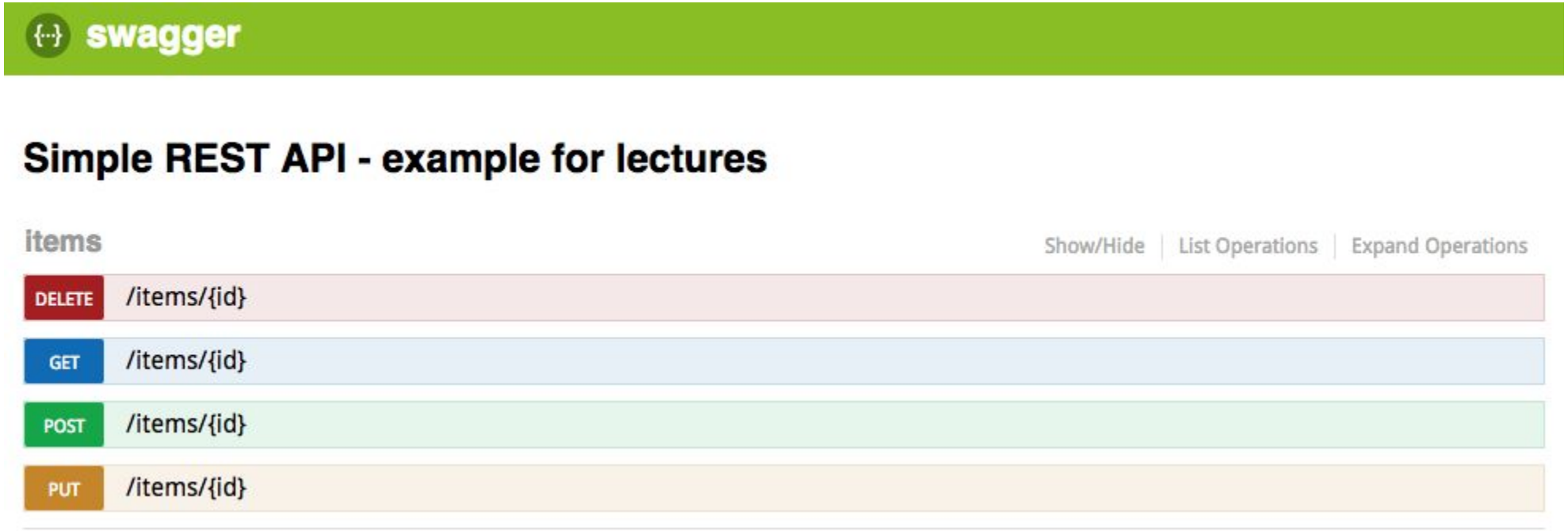
Najczęściej do filtrowania.

GET
example.com/api/blogs?tag=code&query=javascript

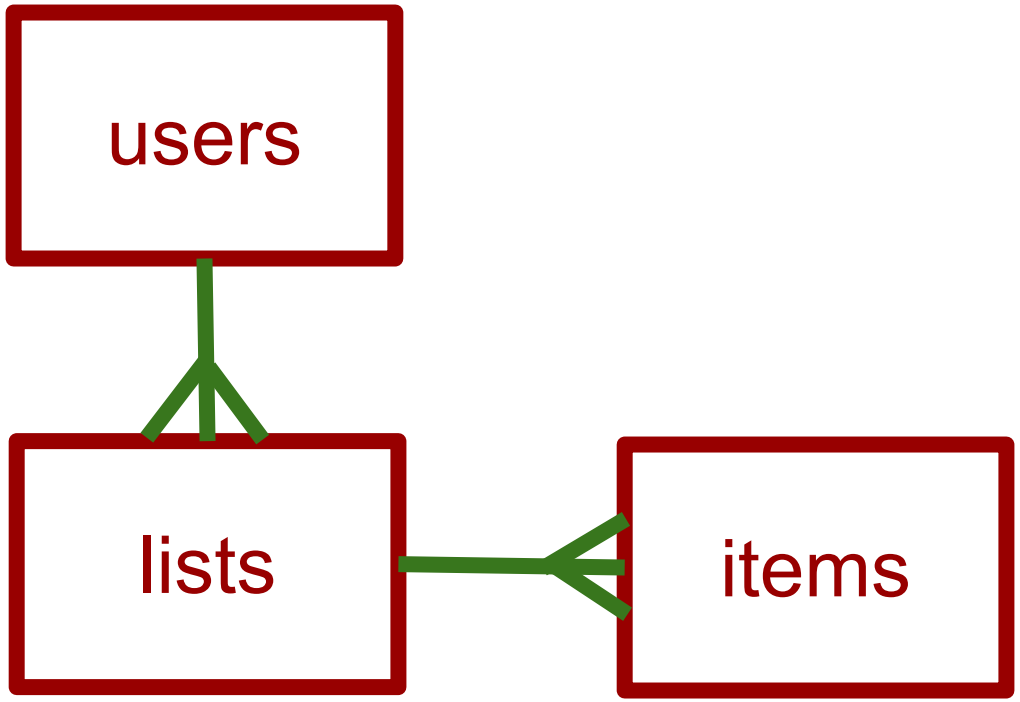
- Zasób: **blogi**
- Filtrujemy dane przez query params: wyszukujemy blogi po tagach `code` oraz zawierające `javascript` w opisie
- **To co jest obsługiwane zależy od serwera**

A co umożliwia mój serwer?

Dokumentacja (przyda się przez całe zajęcia):
https://isa-simple-rest-api.herokuapp.com/api/documentation



[EX 1] Pobierz dane przez POSTMAN/RestClient



adres API: https://isa-simple-rest-api.herokuapp.com/api

dokumentacja:

https://isa-simple-rest-api.herokuapp.com/api/documentation

Korzystając z POSTMAN lub Advanced REST Client pobierz:

- Listę userów
- Wszystkie listy usera o id=1 // na 2 sposoby
- Wszystkie itemy dla listy o id=3 // na 2 sposoby

Przydatne narzędzia:

Firefox: https://addons.mozilla.org/pl/firefox/addon/restclient/

Chrome: https://chrome.google.com/webstore/detail/postman/fhbjgbiflinjbdggehcddcbncdddomop

Co to znaczy, że wszystko zależy od serwera?

Spójrzmy w mój kod:

repo: https://github.com/michalczukm/simple-hapi-rest-api

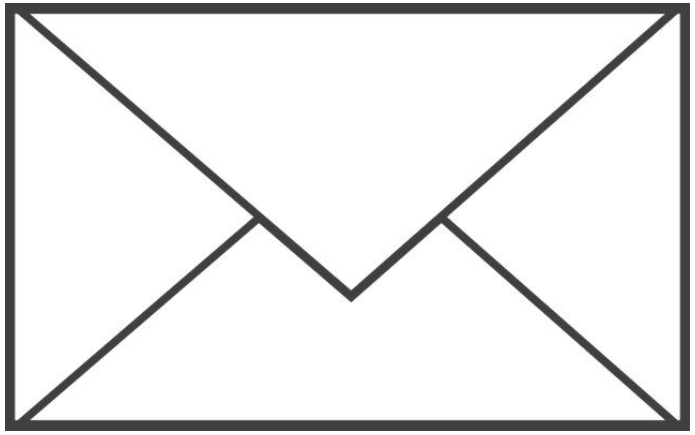
Poszczególne zasoby:

- **user**: https://github.com/michalczukm/simple-hapi-rest-api/blob/master/controllers/users.controller.js
- **list:** https://github.com/michalczukm/simple-hapi-rest-api/blob/master/controllers/lists.controller.js
- **item:** https://github.com/michalczukm/simple-hapi-rest-api/blob/master/controllers/items.controller.js

Klient API - kto może nim być?

- Postman
- cURL - narzędzie konsolowe
- Przeglądarka - przez wklejenie adresu
- Przeglądarka - JavaScript
- Inny serwer
- Aplikacja desktopowa
- Aplikacja mobilna
- Urządzenie IoT
- ...

REST API | HTTP- Odpowiedzi



Response – co dostaliście w odpowiedzi

- Kod http

Status: 200 OK

- Body (treść)

```
{
  "id": 1,
  "content": "First item"
},
```

- Headers (nagłówki)

Content-Type → application/json; charset=utf-8

Date → Sat, 17 Dec 2016 15:38:48 GMT

- Cookies (ciasteczka … nie tym razem)

Response – mogliście też dostać

- Kod http

Status: 404 Not Found

- Body (treść)

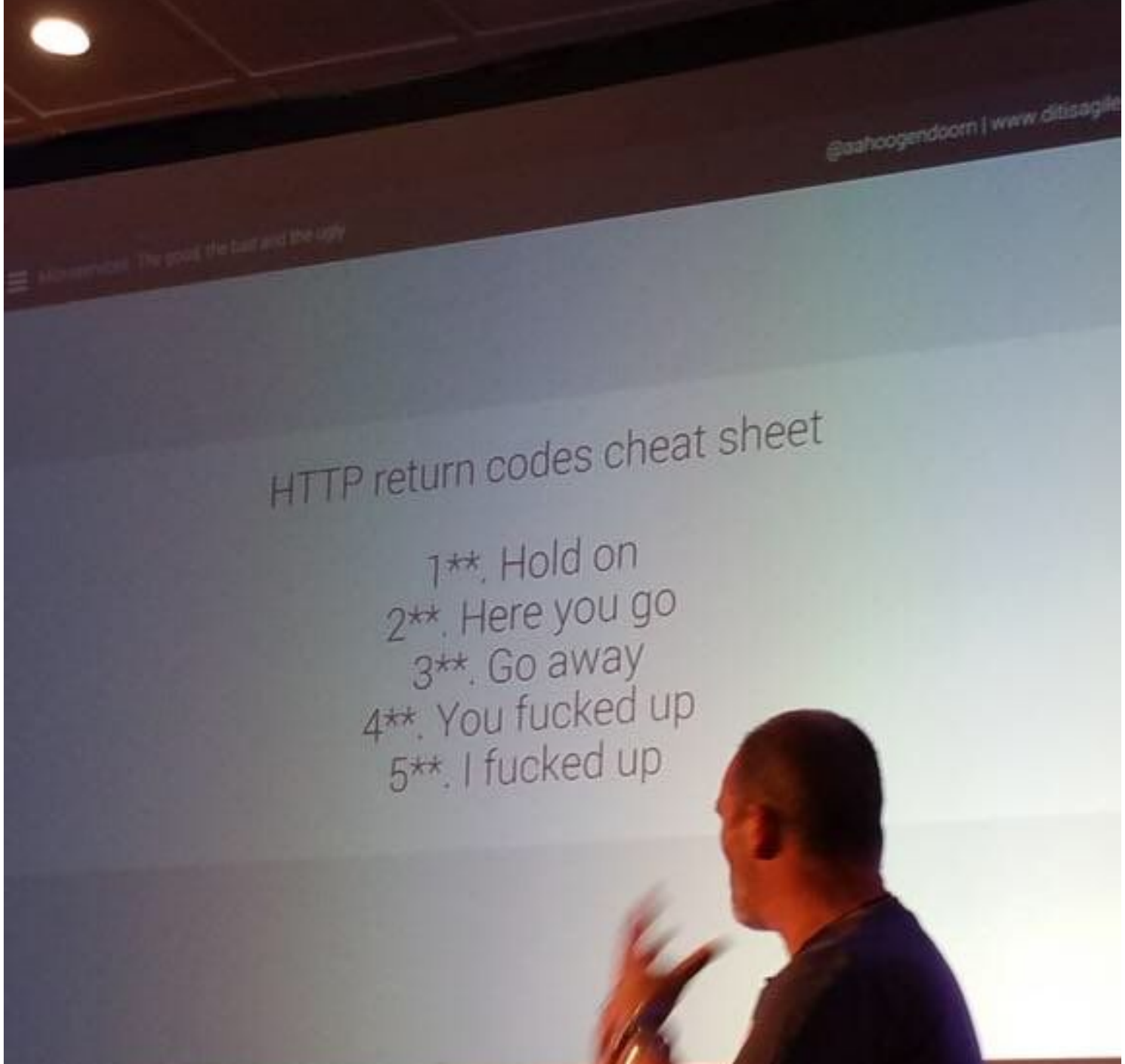
```
{
    "error": {
        "message": "Element doesn't exist",
        "code": "ELEMENT_NOT_EXIST"
    }
}
```

- Headers (nagłowki)

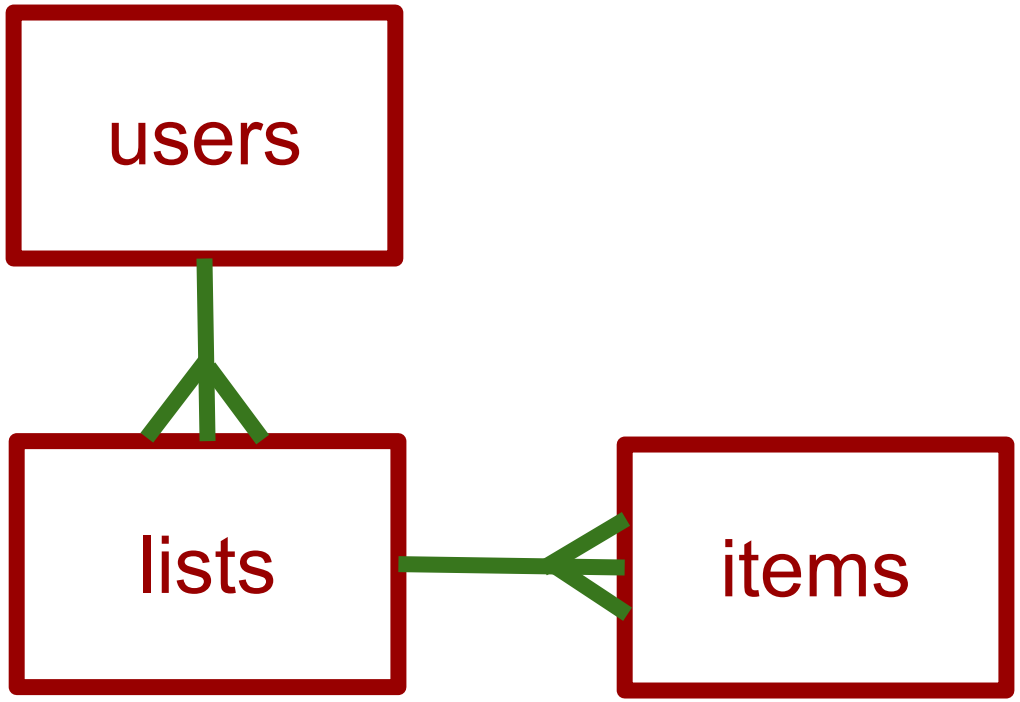
Content-Type → application/json; charset=utf-8

Date → Sat, 17 Dec 2016 15:38:48 GMT

- Cookies (ciasteczka … nie tym razem)


HTTP return codes cheat sheet
1**. Hold on
2**. Here you go
3**. Go away
4**. You fucked up
5**. I fucked up

[EX 2] Dodaj listy oraz itemy, tak aby zachować relacje



Pamiętacie o nagłówku **Content-Type**?

Zanotuj gdzieś na boku **ID list** które stworzyłeś oraz **ID użytkowników** których stwożyłeś.

Korzystając z POSTMAN lub Advanced REST Client:

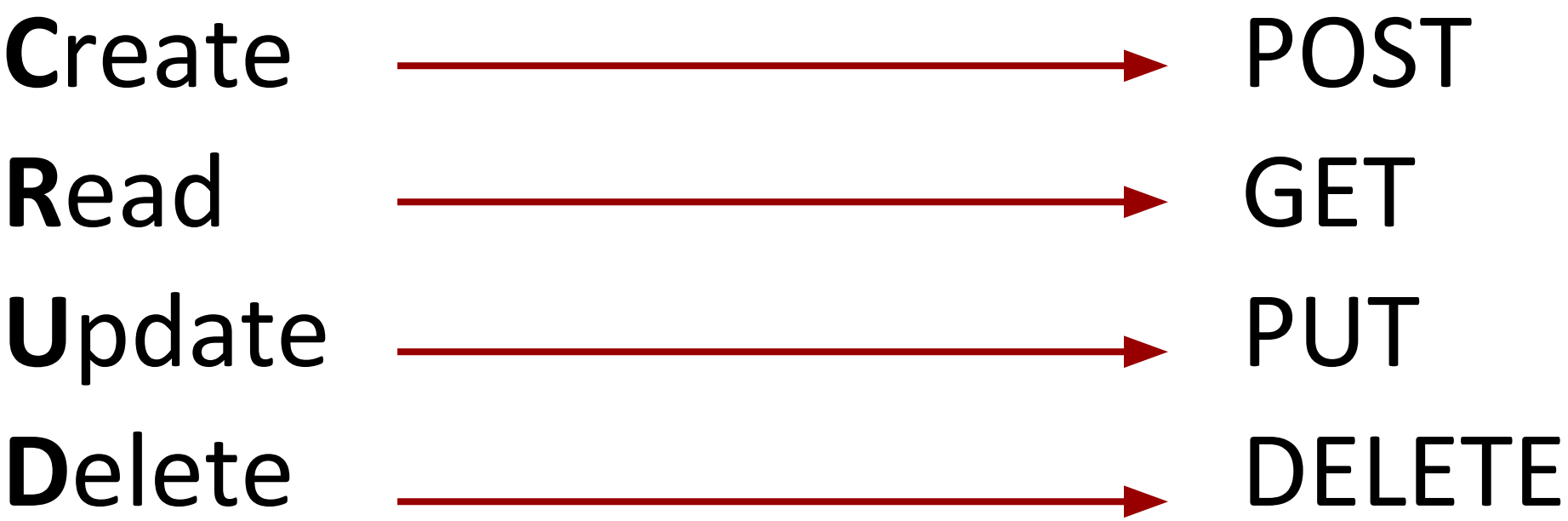
https://isa-simple-rest-api.herokuapp.com/api/

1. Dodaj listy do usera:
    a. Stwórz usera (POST /api/users)
    b. Dodaj 2 listy do usera (POST /api/lists
    c. Zweryfikuj czy use posiada poprawne listy (GET …)
2. Dodaj itemy do listy:
    a. Do stworzonej w punkcie 1. listy dodaj 3 itemy
    b. Zweryfikuj czy lista posiada odpowiednie itemy
3. Sprawdź jakie itemy na wszystkich listach ma user “kowalskim”

(zapisz jakie zapytania musiałeś wykonać)

REST API - HTTP verbs i CRUD

CRUD? Co to ma wspólnego z HTTP verbs?



HTTP verb (method)

Verb	Znaczenie	Przykład
GET	Pobierz dane	**GET** *example.com/api/authors*
POST	Wyślij dane. Utwórz nowy obiekt	**POST** *example.com/api/authors* body: *{ name: "Evil author" }*
PUT	Wyślij dane I stwórz albo uaktualnij obiekt	**PUT** *example.com/api/authors/2* body: *{ name: "Maybe better author" }*
DELETE	Usuń wskazany obiekt	**DELETE** *example.com/api/authors/2*

Headers (nagłówki) - w żądaniu (od klienta)

Meta informacje o żądaniu.

Content-Type (np. *application/json , application/xml*)

Ustaw Content-Type gdy wysyłasz dane.

Co można testować dla takich operacji?

- Status odpowiedzi
- Zawartość odpowiedzi
- Walidację: czy przy błędnych danych dostaniemy 400 oraz odpowiednią wiadomość
- Czy relacje działają poprawnie?

  Np: czy dodając item poprawnie zostanie przypisany do listy

Jak obsłużyć błędy w naszej aplikacji?

HTTP code: 500

body:

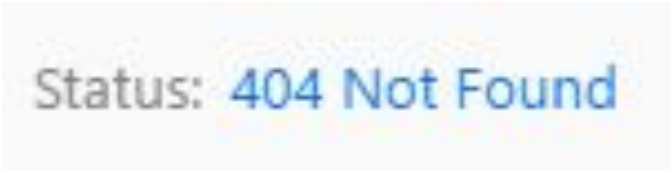
```
{
    error: {
        “message”: “Unexpected error”,
        “code”: “UNEXPECTED_ERROR”
    }
}
```

DOBRZE

ŹLE

Co określa błąd?

- Kod HTTP odpowiedzi



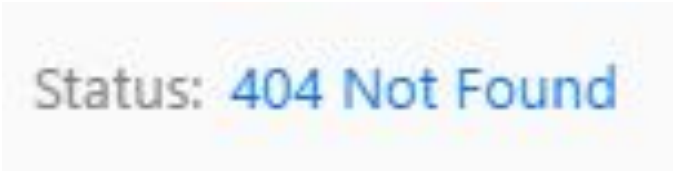
- Body odpowiedzi

```
{
    "error": {
        "message": "Element doesn't exist",
        "code": "ELEMENT_NOT_EXIST"
    }
}
```

Ale to zależy od API

- Kody - różnie mogą być używane
- Format odpowiedzi - nie istnieje jeden standard
- Jak sobie radzić?

Sprawdź dokumentację.




```
{
    "error": {
        "message": "Element doesn't exist",
        "code": "ELEMENT_NOT_EXIST"
    }
}
```

[EX 3] Wyślijmy dodania i zwróćmy uwagę na zwracane błędy

Pamiętacie o nagłówku **Content-Type**?

Korzystając z POSTMAN lub Advanced REST Client:

https://isa-simple-rest-api.herokuapp.com/api/

- POST ~/api/lists

  body: { "title": "some title", “userId”: “first user” }

- POST ~/api/lists

  body: { "title": "", “userId”: “1” }

- POST ~/api/lists

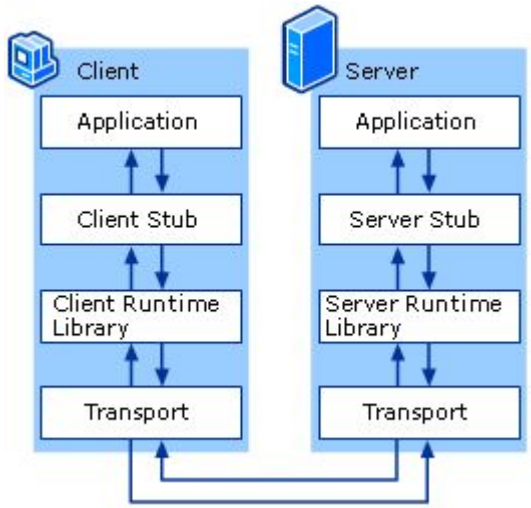
  body: { "title": "my list" }

- POST ~/api/lists

  body: { "title": "my list", “userId”: “1” }

Co możecie powiedzieć o tych odpowiedziach? Jak je testować?

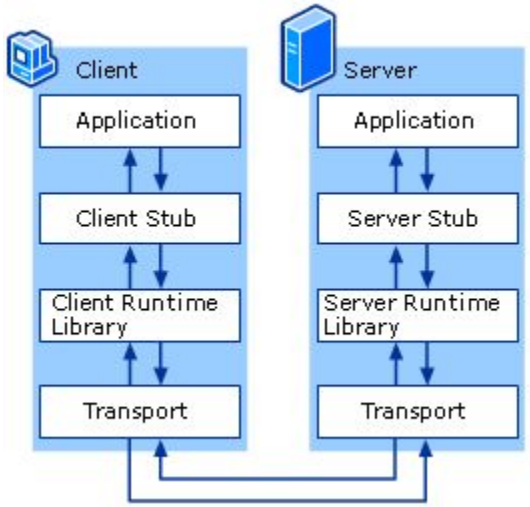
RPC calls - nie do końca REST



Remote **P**rocedure **C**all

```
PUT ~/api/lists/{id}/move
```

- Co jest zasobem?
- Co jest akcją?
- Co się tutaj wydarzy?



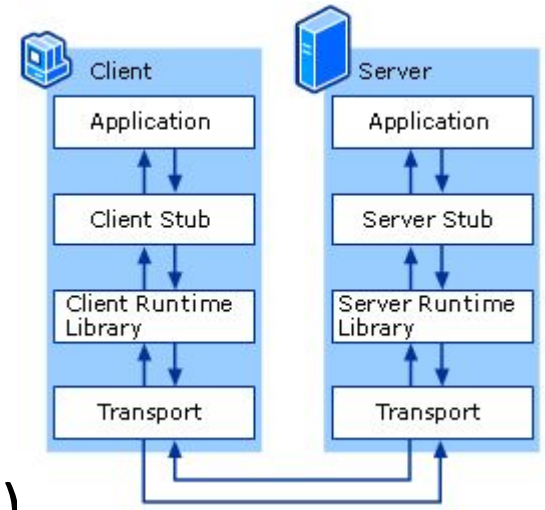
Remote **P**rocedure **C**all

Z reguły API **nie są tak proste** jak typowy CRUD.

Często potrzebujemy dedykowanych metod do wywołania.

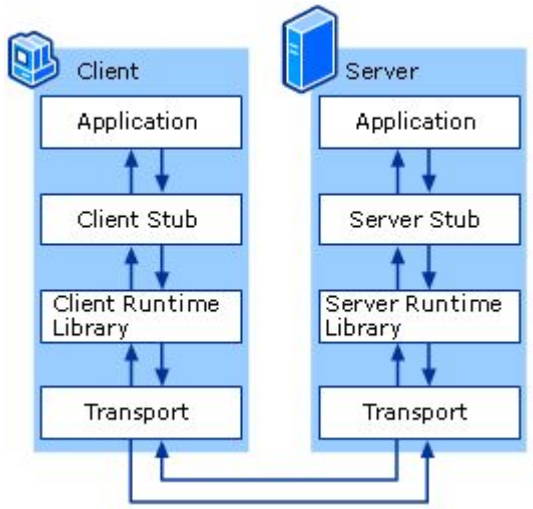
Np.: Zmiana właściciela listy

```
PUT ~/api/lists/{id}/move
```



(w oparciu o nasze REST API też można to zrobić. Jak?)

[EX 4] Wyślijmy dodania i zwróćmy uwagę na zwracane błędy



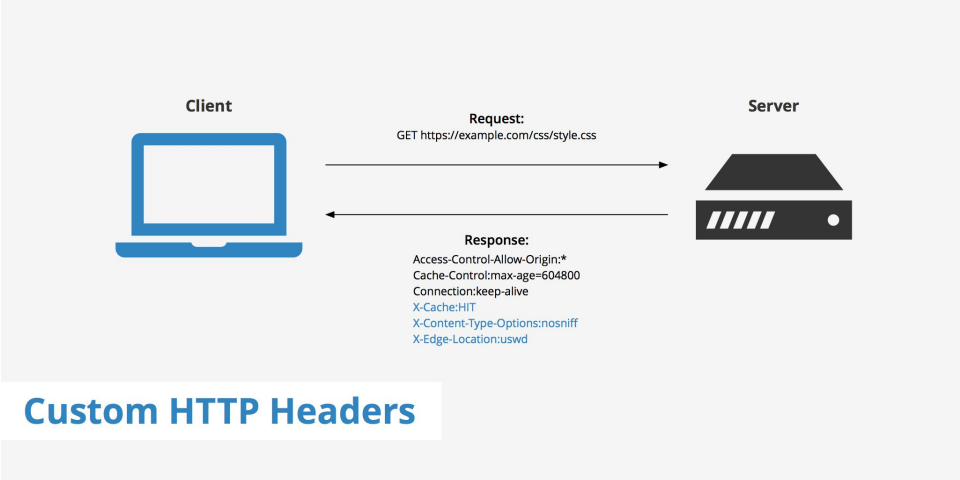
Pamiętacie o nagłówku **Content-Type**?

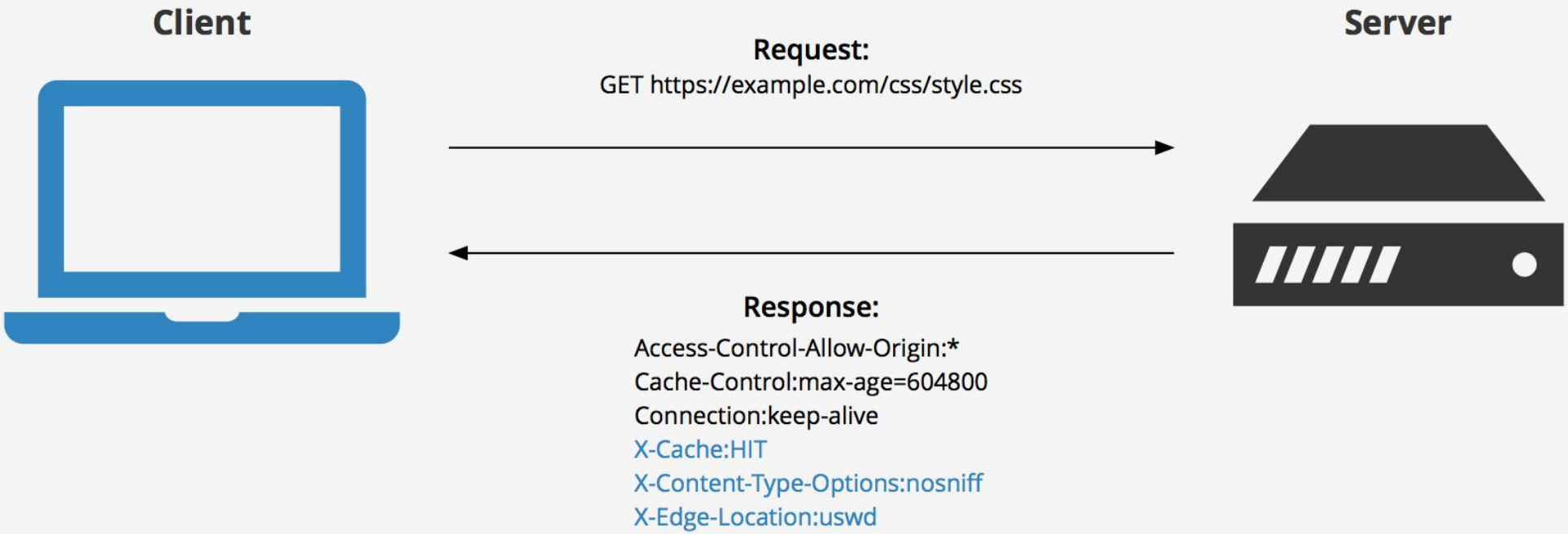
Korzystając z POSTMAN lub Advanced REST Client:

https://isa-simple-rest-api.herokuapp.com/api/

- Zamień właścicieli list z poprzedniego zadania:
  - Przypisz wszystkie listy które stworzyłeś w poprzednim zadaniu użytkownikowi “kowalskip”
  - Zweryfikuj jakie listy mają teraz twoi stworzeni użytkownicy

Request headers a pobieranie danych





Custom HTTP Headers

Paginacja na nagłówkach

Jakich danych potrzebujemy do wyświetlenia paginacji?

Zasób **“list”** można paginować w moim API.

Przez customowe nagłówki.

[EX 5] Paginacja zasobu “list”

Moje customowe nagłówki:
`X-pagination-page`
`X-pagination-per_page`

https://isa-simple-rest-api.herokuapp.com/api/

Dokumentacja:

https://isa-simple-rest-api.herokuapp.com/api/documentation#!/lists/getListsId

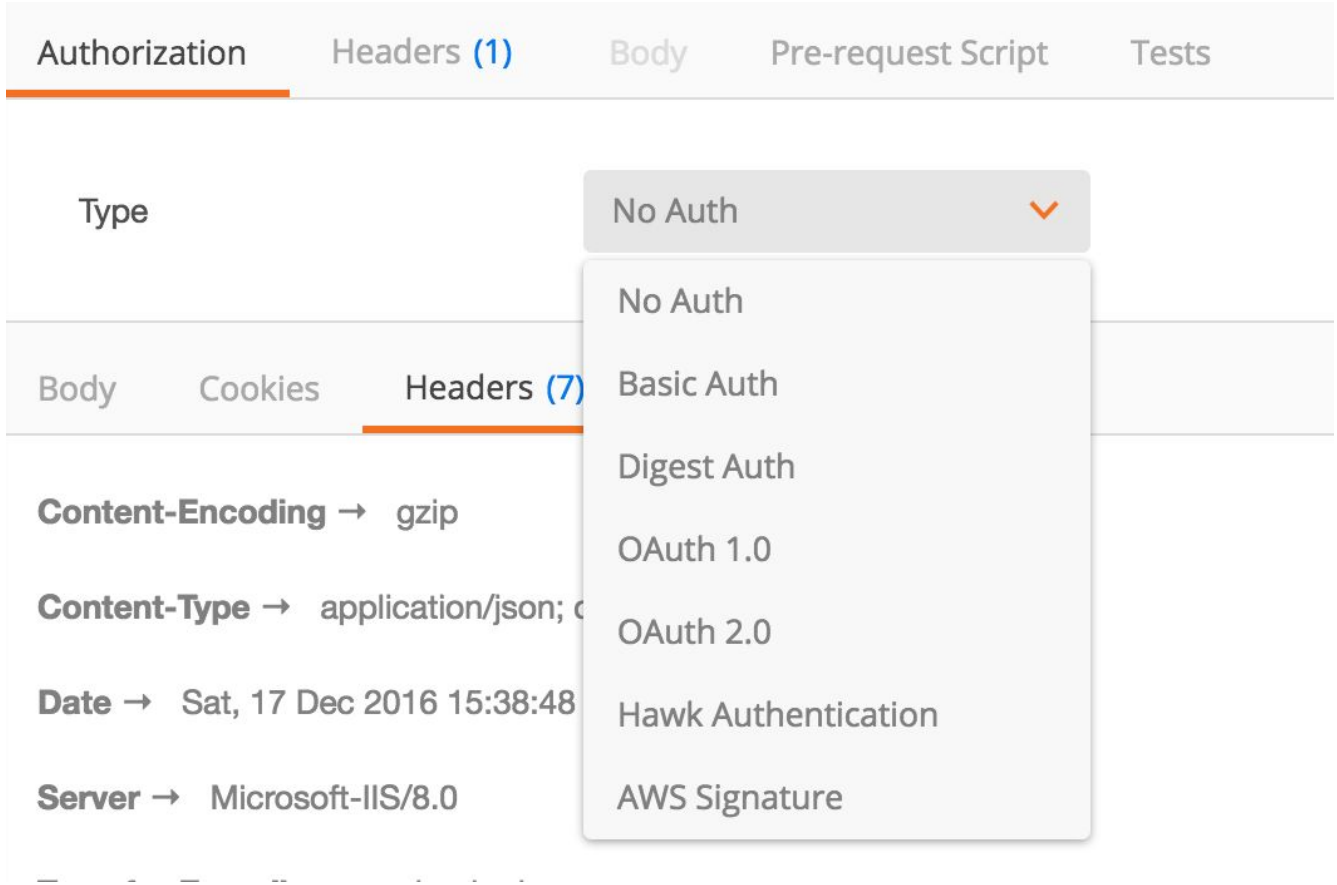
1. Wyślij GET z nagłówkami i pobierz dane dla wszystkich stron gdy:
   a. Nie ustawisz ilości elementów na stronie (`X-pagination-per_page`)
   b. 15 elementów na stronie
   c. 5 elementów na stronie
2. Zwróć uwagę na zwracane nagłówki - co z nich możesz odczytać?
3. Sprawdź i przetestuj przypadki brzegowe. Jakie to mogą być?
4. Pomyśl o testach które powinny być przeprowadzone aby pokryć całą funkcjonalność

W PostMan też można pisać testy!

```
// GET ~/api/lists
// Headers:
//    X-pagination-page: 1
//    X-pagination-per_page: 15

pm.test("Returns 15 elements", function () {
    const responseArrayLength = JSON.parse(pm.response.text()).length;
    pm.expect(responseArrayLength).to.eql(15);
});

pm.test("Content-Type is present", function () {
    pm.response.to.have.header("Content-Type");
});

pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});
```

Uwierzytelnianie do API

Gdzie, jak ?


Authorization
Headers (1)
Body
Pre-request Script
Tests
Type
No Auth
No Auth
Basic Auth
Digest Auth
OAuth 1.0
OAuth 2.0
Hawk Authentication
AWS Signature
Body
Cookies
Headers (7)
Content-Encoding → gzip
Content-Type → application/json;
Date → Sat, 17 Dec 2016 15:38:48
Server → Microsoft-IIS/8.0

Basic auth

Najprostsza opcja.

- Dodaj nagłówek “**Authorization”**
- Treść nagłówka: “Basic base64({user}:{password})”

Tylko po **HTTPS.**

Łatwo przechwycić tą informację.

OAuth 2

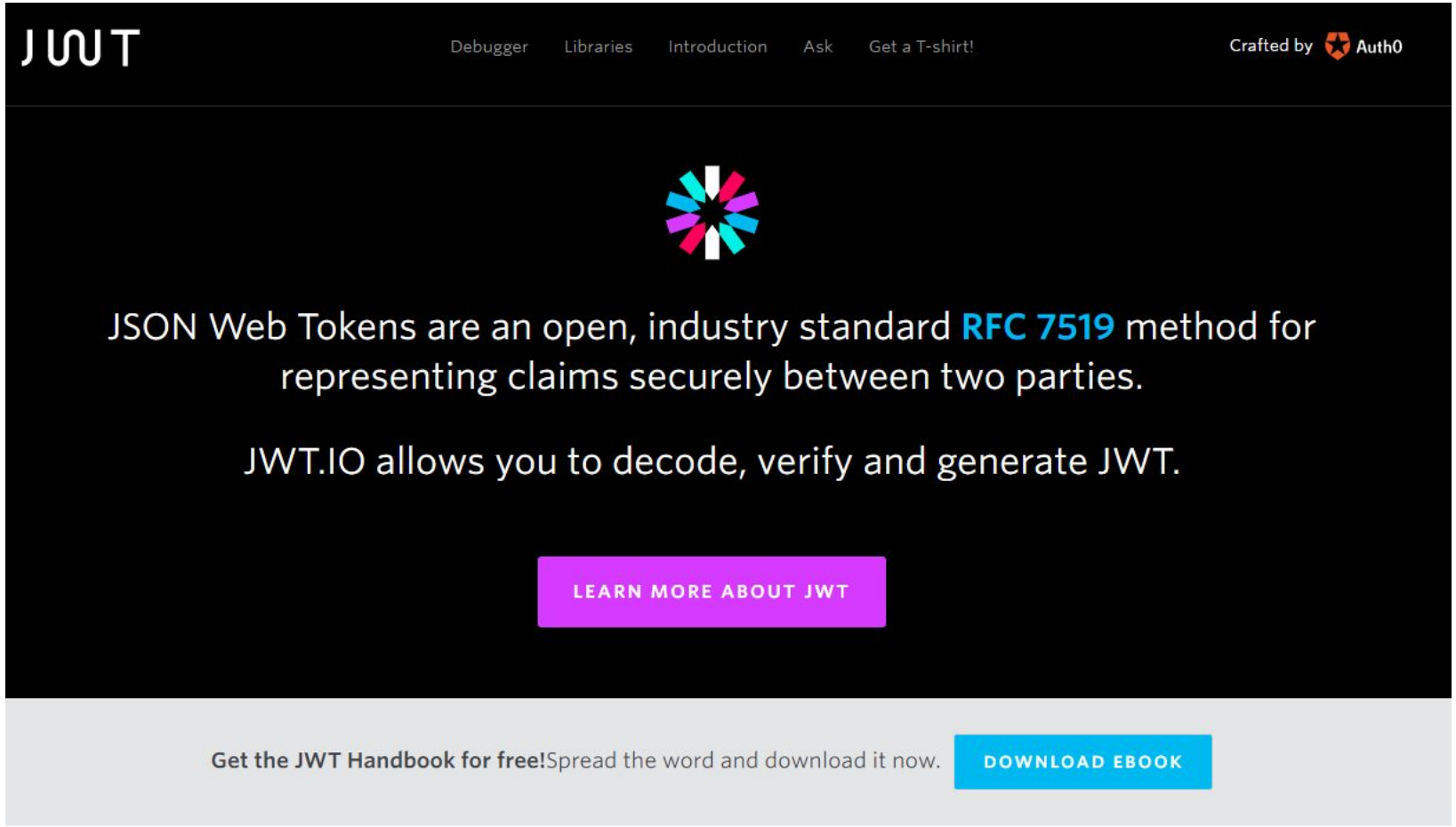
Aktualnie najpopularniejsza opcja. Bezpieczna.

Posługujemy się **Tokenem**, który ktoś **poświadczył**.

Główna różnica pomiędzy OAuth a np. Basic Auth:

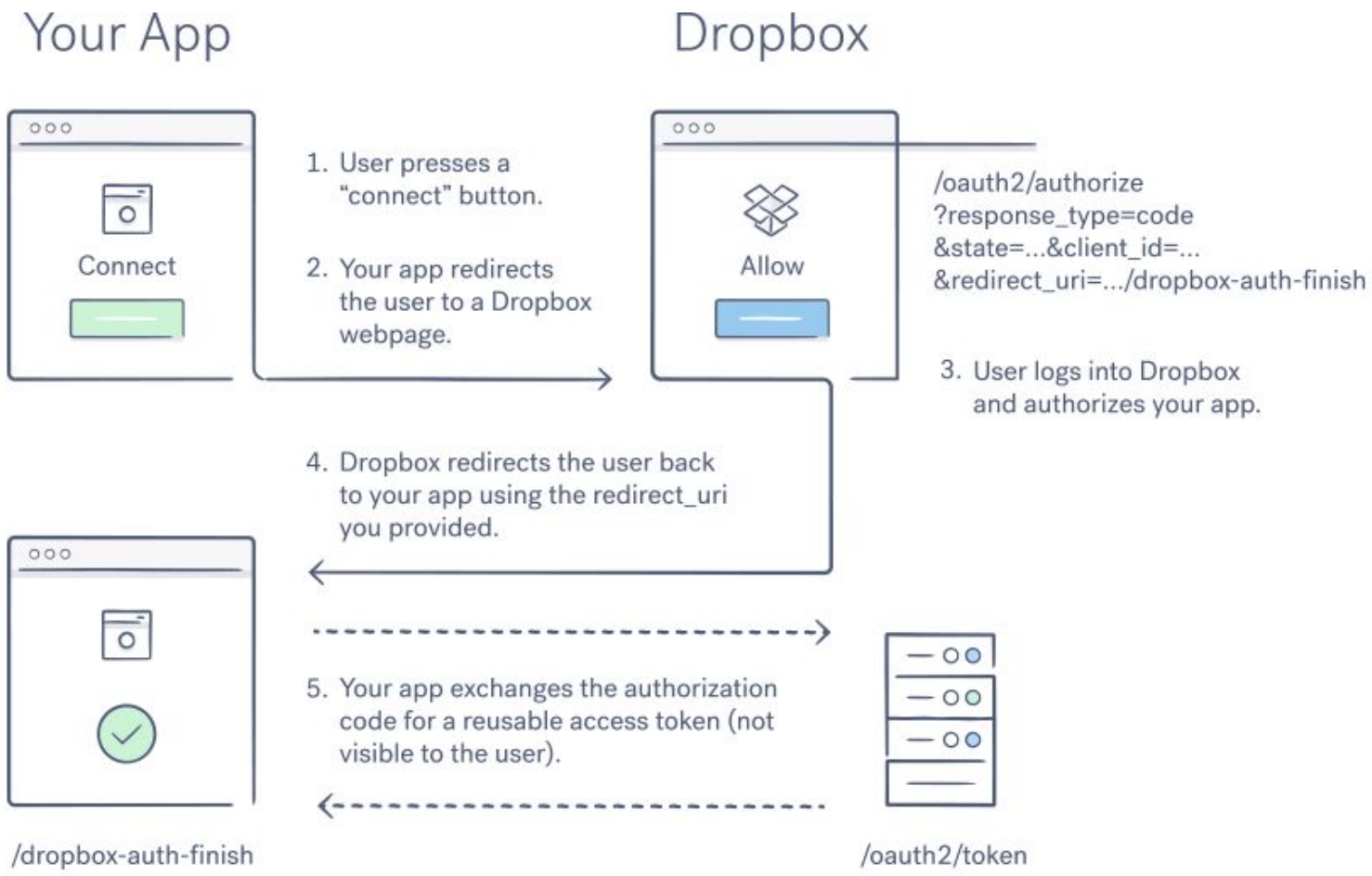
- token to upoważnienie do konkretnych akcji. W Basic Auth, dajemy pełne dane do logowania.
- danie komuś upoważnienia na piśmie VS danie komuś dowodu i karty

JSON Web Tokens - najpopularniejszy format tokenów



https://jwt.io/

OAuth 2 na przykładzie Dropbox



https://www.dropbox.com/developers/reference/oauth-guide

Wersjonowanie

Wersjonowanie przez URL

GET
https://api.twitter.com/1.1/search/tweets.json?q=superbowl

Plusy:

- Łatwo widoczne
- Link jest “klikalny”

Minusy:

- Jaka wersja ma być domyślna? Pierwsza?
- Wszyscy klienci muszą zmieniać URL-e w aplikacjach

Wersjonowanie przez customowy nagłówek

GET https://api.example.com/items

Custom header *api-version:2*

Plusy:

- Nie ma potrzeby zmiany adresów URL

Minusy:

- Link nie jest klikalny
- Nie semantyczne, powinno być w nagłówku “*accept*”

Wersjonowanie przez standardowy nagłówek ***Accept***

GET https://api.example.com/items

Accept header *Accept:application/example.com.v2+json*

Plusy:

- Nie ma potrzeby zmiany adresów URL
- Semantycznie poprawny

Minusy:

- Link nie jest klikalny
- Mało kto o tym wie - słaba wiedza wśród programistów

Wniosek?

Każde wersjonowanie API ma minusy.

Ale musicie być gotowi na każde z nich :)

Najpopularniejsze:

Przez URL.

Jaki model wersjonowania wykorzystuje moje API?

Podsumowanie

- API to rodzaj kontraktu, opisu metod i formatu danych
- W REST API identyfikatorami zasobów są adresy URL, które są jednoznaczne
- Zasoby powinny być nazwane liczbą mnogą
- Reprezentacja to format danych (np json, xml) oraz ich kształt (czyli jakie pola zawierają)
- HTTP verbs definiują akcje którą możemy zrobić
    - Create Read Update Delete → POST GET PUT DELETE
- Nie wszystkie metody HTTP dla zasobu muszą być zaimplementowane po stronie serwera
- Może być wiele "specjalnych" metod w API, warto zajrzeć do dokumentacji (np metody typu RPC, jak nasze `~/api/lists/{id}/move`
- Na odpowiedź i jej stan składa się : Kod HTTP i body
- Większość API jest zabezpieczona, aktualnie najpopularniejszą formą jest OAuth2
- Tokeny najczęściej są w formie JWT
- **To co opisywałem - to dobre praktyki.**

Nie każde API jest REST-owe, nie każde API trzyma się tych reguł.
Co nie oznacza, że jest złym API :)

Parę przydatnych linków

- HTTP Statuses cheat sheet: http://www.restapitutorial.com/httpstatuscodes.html
- Poziomy “dojrzałości” REST API: https://martinfowler.com/articles/richardsonMaturityModel.html
- REST API concepts: https://www.youtube.com/watch?v=7YcW25PHnAA
- REST - the short way: http://exyus.com/articles/rest-the-short-version/
- Go deep into REST: http://www.restapitutorial.com/index.html
- Dlaczego standardowe HTTP verbs są ważne:

https://dev.to/suhas_chatekar/why-should-you-use-standard-http-methods-while-designing-rest-apis

Dziękuję za uwagę

michalczukm

michalczukm@gmail.com