ARCHITEKTURA ZORIENTOWANA NA USŁUGI

Waldemar Korłub

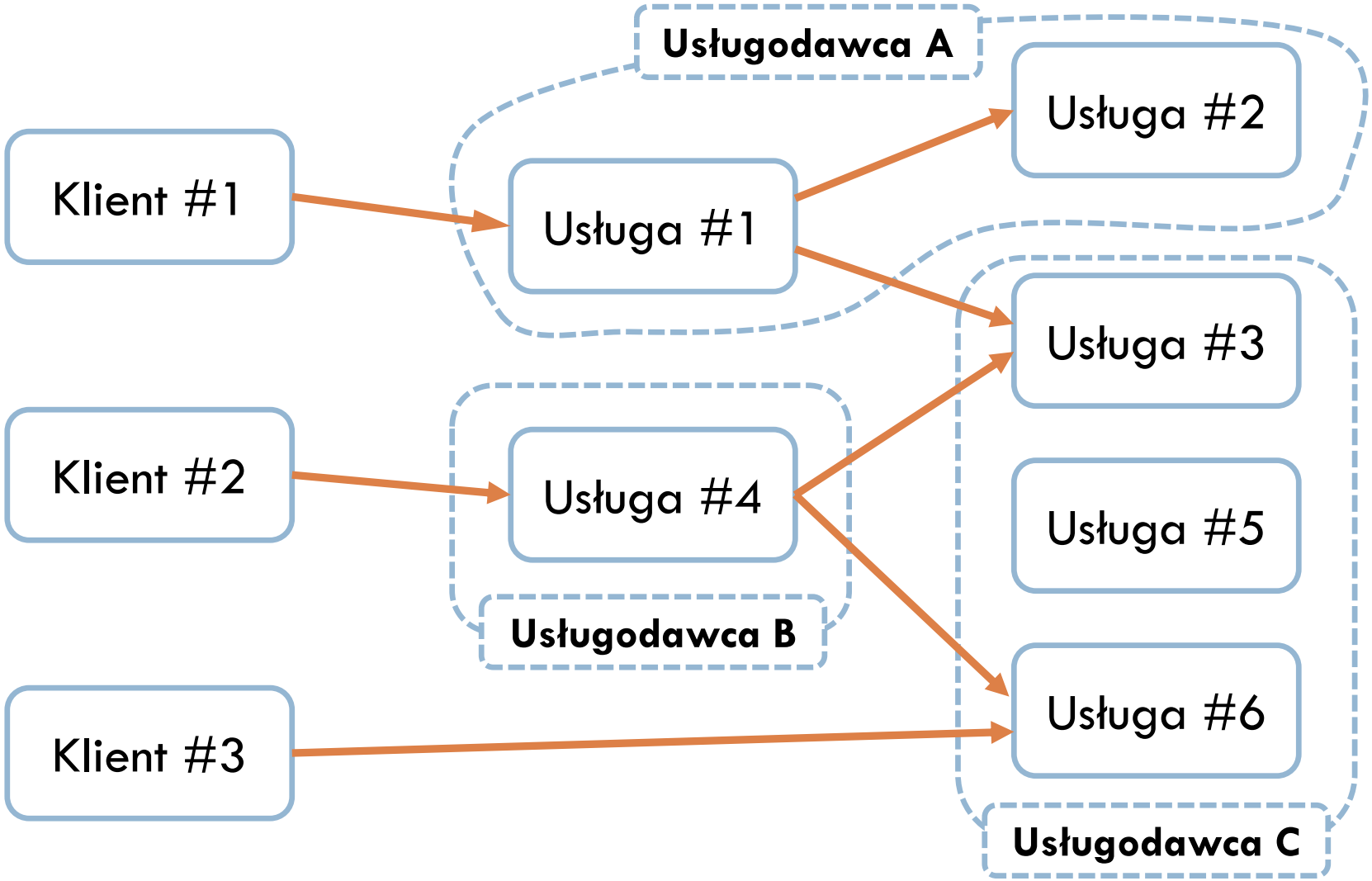
Architektury Usług Internetowych
KASK ETI Politechnika Gdańska

SOA: Service Oriented Architecture

Architektura zorientowana na usługi *Service Oriented Architecture* (SOA)

- Usługodawcy
  - Usługa prognozy pogody
  - Usługa płatności (np. kartą kredytową)
  - Usługa rozsyłania wiadomości SMS
- Klienci – aplikacje klienckie
  - Sklep internetowy
    - wykorzystuje płatności
  - Aplikacja na urządzenie mobilne (np. smartphone)
    - wykorzystuje serwis pogodowy
  - Aplikacja okienkowa w przychodni
    - wykorzystuje usługę SMSową (np. przypomnienia o wizytach)
  - Komponent w środowisku rozproszonym

Service Oriented Architecture (SOA)



Service Oriented Architecture (SOA)

- Klient nie musi znać szczegółów działania usługi – ani implementacyjnych ani biznesowych/branżowych
  - Usługa pogodowa
    - Szczegóły branżowe: skąd pochodzą dane meteorologiczne (stacje, czujniki), jaki model prognozowania został zastosowany
    - Szczegóły implementacyjne: jak dane są składowane
    - Klient chce tylko wiedzieć, czy będzie padać!
  - Usługa płatności
    - Szczegóły biznesowe: umowy z bankami, sesje rozliczeniowe
    - Szczegóły implementacyjne: komunikacja z systemem bankowym
    - Klient – np. sklep internetowy – chce tylko wiedzieć, czy zamówienie zostało opłacone!

Service Oriented Architecture (SOA)

- Architektura SOA pozwala klientom usług skupić się na własnych działaniach biznesowych
- Operacje niezwiązane bezpośrednio z biznesem klienta zostają oddelegowane do usługodawców, którzy się w nich specjalizują
  - Forma *outsourcingu*
- Usługa może wywoływać inne usługi (orkiestracja) w celu zaspokojenie potrzeb klienta
  - Usługa staje się klientem innych usług, które mogą pochodzić od innego usługodawcy

Manifest SOA (ang. *SOA Manifesto*)

- Business value over technical strategy
  - Wartość biznesowa ponad aspekty technologiczne
- Strategic goals over project-specific benefits
  - Strategiczne cele ponad korzyści dla pojedynczego projektu
- Intrinsic interoperability over custom integration
  - Interoperacyjność ponad integrację z własnościowymi rozwiązaniami

Manifest SOA (ang. *SOA Manifesto*)

- Shared services over specific-purpose implementations
  - Współdzielone usługi ponad dedykowane implementacje
- Flexibility over optimization
  - Elastyczność ponad optymalizację
- Evolutionary refinement over pursuit of initial perfection
  - Ewolucyjne udoskonalenia przed pogonią za początkową doskonałością

Architektura zorientowana na usługi

- Usługodawca i klient to często dwa całkowicie odrębne podmioty
  - Odrębne aplikacje, odrębne firmy
  - Różne języki implementacji i platformy technologiczne
    - Java, C#/.NET, PHP, JavaScript, Objective-C, Swift
  - Różne środowiska działania aplikacji
    - Windows, Linux, OS X, Android, iOS
    - 32-bitowe/64-bitowe, LittleEndian/BigEndian
- **Heterogeniczne środowisko**
  - Różnorodne komponenty wchodzące w interakcje

Komunikacja

Jak połączyć ze sobą heterogeniczne komponenty?

Komunikacja w usługach sieciowych

Protokoły komunikacyjne

- Historyczne
  - CORBA
  - Remote EJB
  - SOAP
- Aktualnie dominujące
  - REST
    - Swagger, RAML, API Blueprints
- Pretendenci
  - GraphQL (Facebook)
  - gRPC (Google)
  - Dubbo (Alibaba)

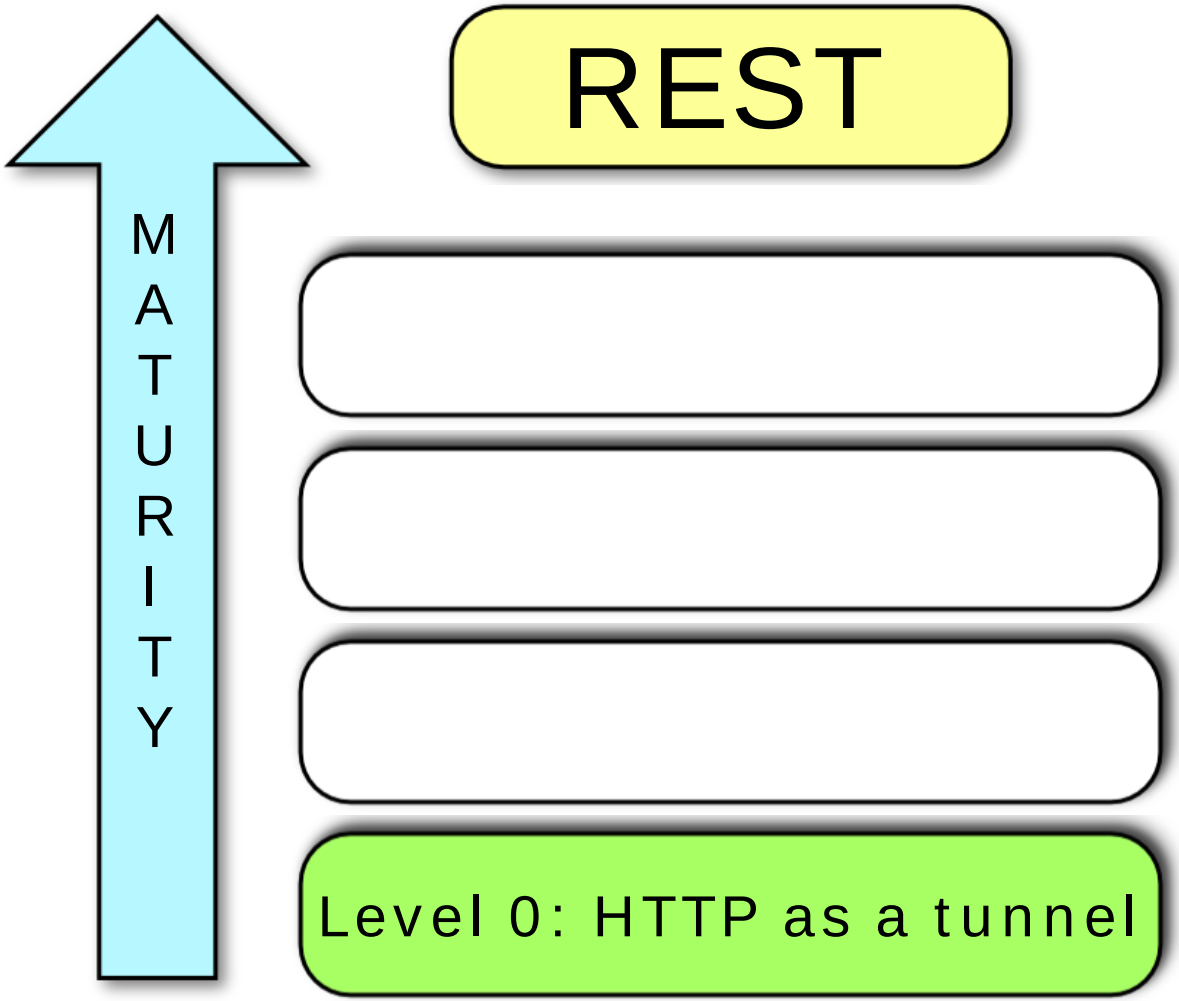
Formaty danych

- Historyczne
  - Własnościowe formaty binarne
  - XML
- Aktualnie popularne
  - JSON – dominujący
  - YAML
- Pretendenci
  - Protocol Buffers (Google)

Oczekiwania deweloperów

- Łatwość użycia w aplikacjach mobilnych
- Łatwość użycia w językach skryptowych (m.in. JavaScript w środowisku przeglądarki)
- Proste protokoły
  - Najlepiej takie, które już znamy
  - Łatwe w obsłudze nawet gdy nie mamy dedykowanych narzędzi
    - …a jeśli mamy to też chętnie ich użyjemy
- Proste i szybkie w przetwarzaniu formaty danych

Model dojrzałości usług wg Richardsona



SOAP jako przykład *poziomu 0*

- HTTP jako transport
- Pojedynczy punkt wejścia:
  http://example.com/Service
- Zawsze żądania POST
  - Operacje i ich parametry w ciele żądania (koperty SOAP)
- Dane uwierzytelniające również w ciele żądania
  - WS-Security

Co można poprawić?

- *Divide et impera*
  - Rozbicie pojedynczego punktu wejścia (czarnej skrzynki) na zbiór mniejszych i łatwiejszych w zarządzaniu zasobów
  - Łatwiejsze implementowanie i konsumowanie usługi
- Pojedynczy URI identyfikuje pojedynczy zasób
  - Hierarchicznie uporządkowane adresy dla zagnieżdżonych zasobów
  - Zasób to *rzeczownik*, a nie *czasownik*
    - Reprezentuje obiekt, na którym wykonujemy akcje (poziom 2)
    - …nie reprezentuje samej akcji

Struktura adresów – popularna konwencja

- Kolekcja elementów (użytkownicy):
  http://example.com/users
- Pojedynczy element kolekcji (użytkownik) :
  http://example.com/users/gandalf
- Kolekcja zagnieżdżona w pojedynczym elemencie:
  http://example.com/users/gandalf/friends
- Inna zagnieżdżona kolekcja:
  http://example.com/users/gandalf/msgs
- *We need to go deeper*:
  http://example.com/users/gandalf/msgs/17/replys

Struktura adresów

- Który jest lepszy **w kontekście API**?

/users/1234/comments/?sort=latest

czy

/users/1234/comments/latest/

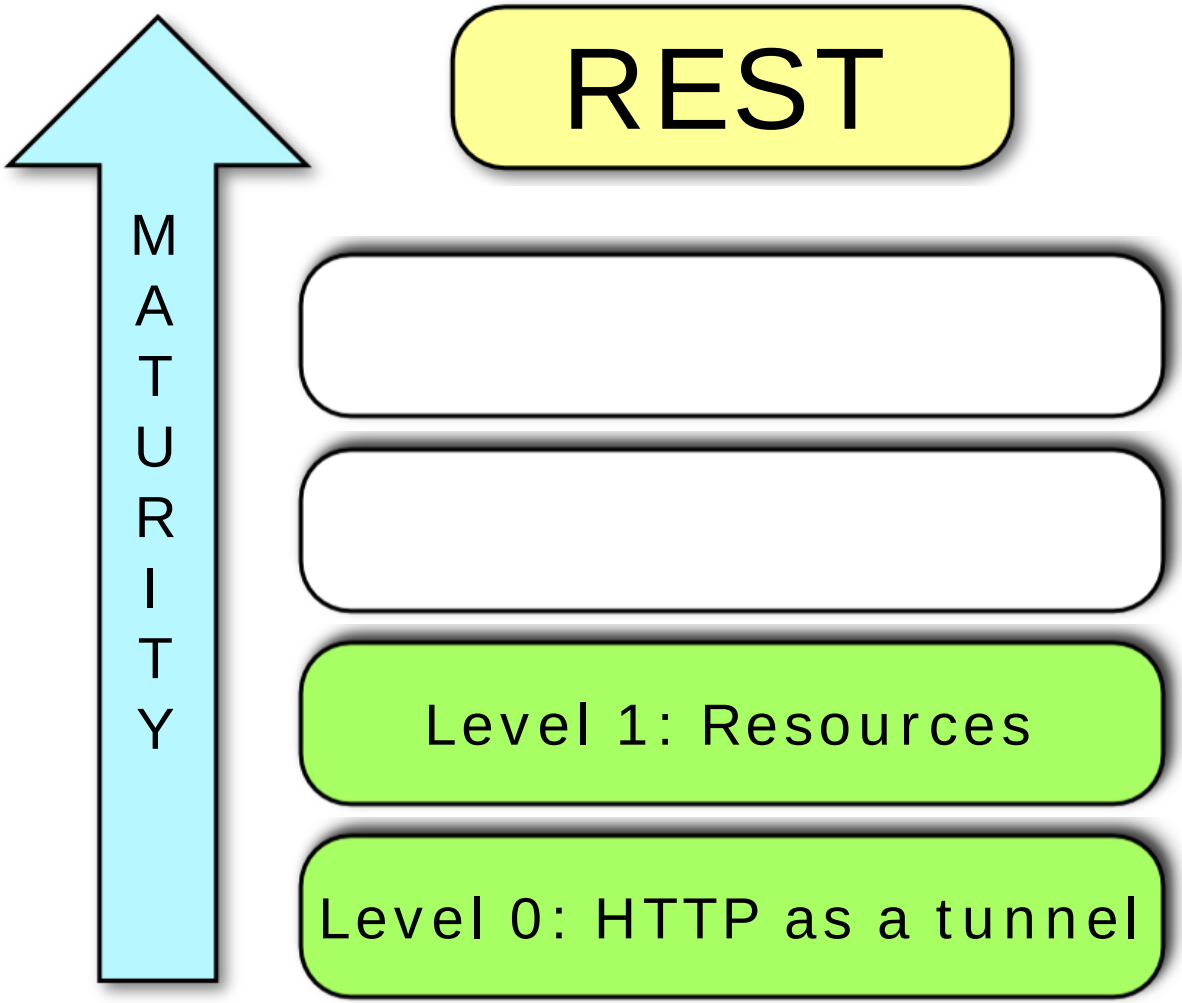
- Osobne URI mają reprezentować osobne zasoby
  - Kolekcja komentarzy
  - Kolekcja komentarzy w kolejności odwrotnie chronologicznej

} Inne zasoby, czy *inne reprezentacje* tego samego zasobu?

Struktura adresów

- Jeden URI reprezentuje jeden zasób:
  /users/1234/comments/
- Parametry doprecyzowują zapytanie użytkownika (aplikacji klienckiej), np.:
  - Sortowanie:
    ?sort=latest
  - Filtrowanie:
    ?unread=true
  - Stronicowanie:
    ?offset=20&limit=10
- Parametry umożliwiają uzyskanie różnych reprezentacji tego samego zasobu

Poziom 1



Gdy dojdziemy do poziomu 3…

…adresy nie będą miały znaczenia!

Co dalej?

- SOAP: wyłącznie zapytania POST
  - Cacheowanie w warstwie aplikacji
  - Definicje operacji w warstwie aplikacji
    - W ciele żądania – koperta SOAP
      - Wymaga głębokiej inspekcji żądań
    - W nagłówkach HTTP
      - Wymaga głębokiej inspekcji żądań
    - W adresie żądania, np.: http://example.com/posts/1234/comments/5678/**edit**
      - Nie wymaga głębokiej inspekcji
      - Ale nie to nie jest osobny zasób!

Poziom 2: HTTP verbs

- Wykorzystanie operacji zdefiniowanych w protokole HTTP
  - GET
  - POST
  - PUT
  - DELETE
- Każda operacja ma zdefiniowaną semantykę, wymagania i ograniczenia
- Gdy ograniczenia są dobrze zdefiniowane możemy optymalizować w oparciu o nie

Operacje na zasobach

- Operacje na zasobach wykonywane przy użyciu żądań HTTP: GET/POST/PUT/DELETE

	Kolekcja elementów, np. /books	**Pojedynczy element, np. /books/17**
GET	Pobranie kolekcji	Pobranie elementu
POST	Dodanie elementu do kolekcji	✗ Nie stosuje się
PUT	✗ Nie stosuje się (lub zastąpienie całej kolekcji)	Aktualizacja elementu
DELETE	Usunięcie całej kolekcji	Usunięcie elementu

Wymagania i ograniczenia

- GET
  - Bezpieczne
    (nie zmienia stanu na serwerze)
  - Idempotentne
  - *Cachowalne!*
- POST
  - Może zmieniać stan
  - Nie musi być idempotentne
- PUT
  - Może zmieniać stan
  - Idempotentne
- DELETE
  - Może zmieniać stan
  - Idempotentne

Kody odpowiedzi HTTP

- Równie ważne jak określenie operacji do wykonania jest określenie jej rezultatu
- Rezultat można opisać w ciele odpowiedzi
- …ale wymaga to od klienta parsowania ciała odpowiedzi i interpretacji zawartości
- Wiele typowych rezultatów operacji można wyrazić za pomocą samego kodu odpowiedzi HTTP

Kody odpowiedzi HTTP

- Sukces operacji (2xx):
  - 200 OK
    - Po prostu OK…
    - W odpowiedzi na żądania GET/PUT/DELETE
  - 201 Created
    - W odpowiedzi na żądanie POST
    - np. komentarz zapisany prawidłowo
  - 202 Accepted
    - W odpowiedzi na żądania POST/PUT
    - np. przyjęte zlecenie długotrwałej operacji
      - Nie ma pewności czy operacja się powiedzie

Kody odpowiedzi HTTP

- Błąd operacji (4xx):
  - 404 Not Found
  - Błędy na etapie kontroli dostępu:
    - 401 Unauthorized
      - konieczne uwierzytelnienie
    - 403 Forbidden
      - brak dostępu
  - 409 Conflict
    - Najczęściej w odpowiedzi na żądanie PUT
    - np. próba nadpisania wersjonowanego zasobu starszą wersją

Kody odpowiedzi HTTP

- Błąd operacji (4xx):
  - 417 Expectation Failed
    - Niespełnione oczekiwania…
    - np. nagłówek `Expect: 100-continue`

http://www.thedailystar.net/lifestyle/how-deal-parents-unrealistic-expectations-your-20s-1310869

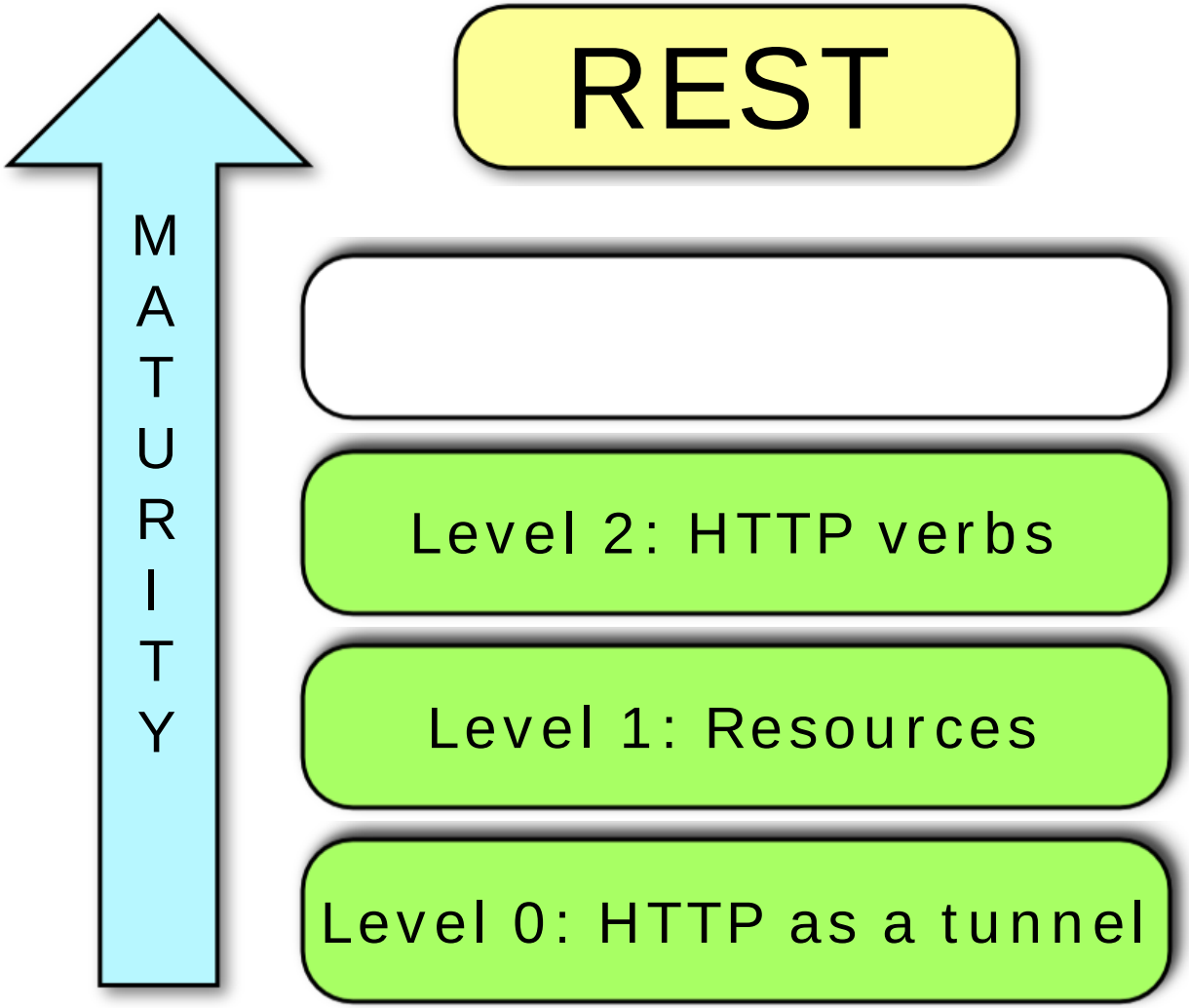
Kody odpowiedzi HTTP

- Błąd operacji (4xx):
  - Problem z przetworzeniem żądania:
    - 415 Unsupported Media Type
      - Format danych w żądaniu nie jest obsługiwany przez serwer
      - np. serwer oczekuje formatu JSON, a klient wysłał XML
    - 400 Bad Request
      - Odrzucenie żądania ze względu na błąd klienta
      - np. format danych (nagłówek Content-Type) jest poprawny, ale składnia jest błędna
    - 422 Unprocessable Entity
      - Format jest poprawny (~~415~~), składnia jest poprawna (~~400~~), ale semantyka żądania nie jest poprawna
      - np. próba zamówienia nieistniejącego produktu w sklepie

Kody odpowiedzi HTTP

- Błąd po stronie serwera – 5xx
  - Deweloper aplikacji klienckiej wchodzi na naszego *bug trackera* i zgłasza błąd

Poziom 2


REST
MATURITY
Level 2: HTTP verbs
Level 1: Resources
Level 0: HTTP as a tunnel

Wyślijmy JSONa po HTTP…

GET http://example.com/users/1234

```
{
  "id": 1234,
  "username": "gandalf",
  "avatar_id": 5678,
  "firstname": "Michal",
  "lastname": "Kowalski",
  "group_id": 9012
}
```

Wyślijmy JSONa po HTTP…

- Prosty protokół komunikacyjny
  - Nie wymaga dedykowanych narzędzi – wystarczy klient HTTP
  - Proste wywoływanie usług w każdym języku i na każdej platformie
    - Również na urządzeniach mobilnych
- Łatwy do parsowania format danych
  - Oszczędzamy baterię urządzenia mobilnego i czas użytkownika
- Zwięzłe żądania i odpowiedzi
  - Małe narzuty na dane

Jak bardzo REST-owa jest nasza usługa?

- REST architectural constraints:
  - **client-server**
    (klient-serwer)
  - **stateless**
    (bezstanowość)
  - **cacheable**
  - **layered**
    (warstwowość)
  - **uniform interface**
    (jednorodny interfejs)

Jednorodny interfejs

- **identification of resources**
  (identyfikacja zasobów)
- **manipulation through representations**
  (operacje na reprezentacjach zasobów)
- **self-descriptive messages, e.g. format, cacheability**
  (samoopisujące się żądania)
- **HATEOAS**
  Hypermedia As The Engine Of Application State
  (system sterowany relacjami hipertekstowymi)

Co jest nie tak z tym zasobem?

- Żądanie:

  GET http://example.com/users/1234

- Ciało odpowiedzi:

```
{
  "id": 1234,
  "username": "gandalf",
  "avatar_id": 5678,
  "firstname": "Michal",
  "lastname": "Kowalski",
  "group_id": 9012
}
```

Co jest nie tak z tym zasobem?

GET http://example.com/users/1234

```
{
  "id": 1234,
  "username": "gandalf",
  "avatar_id": 5678,
  "firstname": "Michal",
  "lastname": "Kowalski",
  "group_id": 9012
}
```

- Płaska reprezentacja
  - Klient często potrzebuje hierarchicznej reprezentacji danych – redukcja liczby żądań

Co jest nie tak z tym zasobem?

GET http://example.com/users/1234

```
{
  "id": 1234,
  "username": "gandalf",
  "avatar_id": 5678,
  "firstname": "Michal",
  "lastname": "Kowalski",
  "group_id": 9012
}
```

- Magiczne numerki:
  - 9012
  - 5678
- Jak zrobić z nich użytek?
  - Klient potrzebuje dodatkowej wiedzy, aby je wykorzystać

HATEOAS

- Relacje między zasobami w Internecie wyrażane są przy pomocy hipertekstu
- Ludzie podążają według relacji hipertekstowych aby *zrealizować swoje cele biznesowe*
  - „Do koszyka”
  - „Złóż zamówienie”
  - Semantyka rozpoznawana na podstawie etykiet
- Zautomatyzowane klienty powinny działać tak samo!
  - …ale na jakiej podstawie mają one rozpoznawać semantykę relacji?

Relacje między zasobami

GET http://example.com/users/1234

```
{
  "id": 1234,
  "username": "gandalf",
  "avatar": "/users/1234/avatar",
  "firstname": "Imiesław",
  "lastname": "Nazwiskowy",
  "group": { ... }
  "friends": "/users/1234/friends/",
}
```

Relacja hipertekstowa między zasobem użytkownika i kolekcją jego znajomych

HATEOAS

- Hipertekst to więcej niż linkowanie jednego zasobu w innym zasobie
- Hipertekst wskazuje też operacje możliwe do wykonania na zasobach
  - Które zwracają kolejne zasoby ze swoimi operacjami
  - Jak metody w obiektowym języku programowania
  - …albo akcje w procesie biznesowym

Relacje między zasobami

GET http://example.com/users/1234

```
{
  "id": 1234,
  "username": "gandalf",
  "avatar": "/users/1234/avatar",
  "firstname": "Imiesław",
  "lastname": "Nazwiskowy",
  "friends": "/users/1234/friends/",
  "_links": [
    {
      "uri": "/users/1234/friends/",
      "method": "POST",
      "rel": "like"
    }
  ]
}
```

Obiektowa reprezentacja zasobu

```
public class User {
    Integer id;
    String firstname;
    /*...*/
    public Avatar getAvatar() {
    /* GET /users/1234/avatar */
    }
    public User getFriends() {
    /* GET /users/1234/friends/ */
    }
    public User like() {
    /* POST /users/1234/friends/ */
    }
}
```

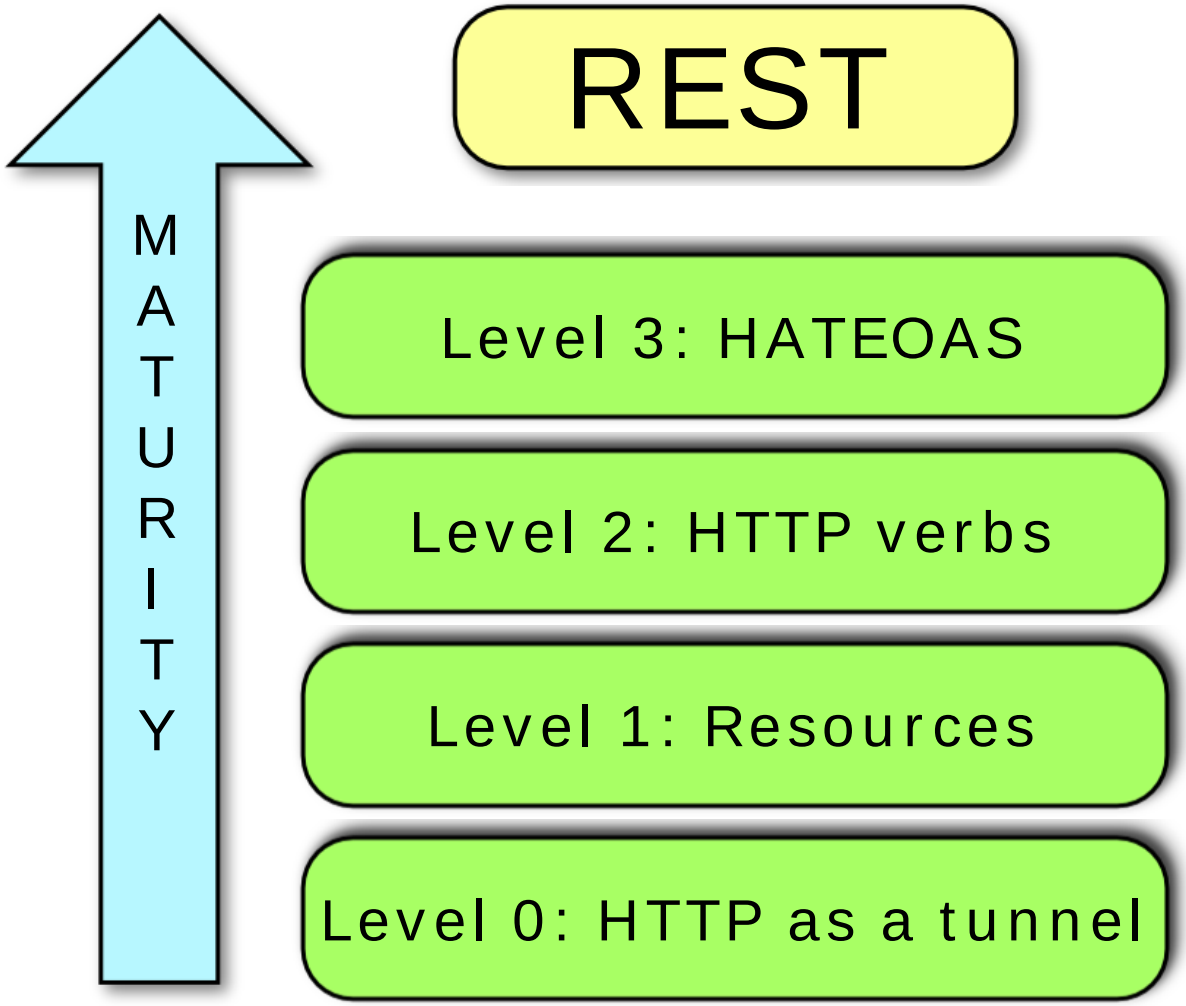
Definiowanie relacji hipertekstowych

- Relacje w obrębie zasobów mogą być definiowane jakkolwiek
  - REST to nie SOAP (specyfikacje, protokoły, standardy…)
- …ale w architekturze SOA potrzebujemy standaryzacji!
  - Łatwiejsze konsumowanie API
  - Możliwość budowania narzędzi automatyzujących typowe czynności

Standardowe relacji

- Typowe relacje są zdefiniowane w specyfikacjach takich jak AtomPub lub Web Linking (RFC 5988):
  - self – odnośnik na ten sam zasób
  - current – zasób reprezentujący najnowsze elementy w kolekcji
  - next/prev – następny poprzedni zasób w kolekcji
  - start/end – pierwszy ostatni zasób w kolekcji
  - up – zasób nadrzędny w hierarchii

Poziom 3


REST
MATURITY
Level 3: HATEOAS
Level 2: HTTP verbs
Level 1: Resources
Level 0: HTTP as a tunnel

Richardson Maturity Model

- Level 1: zarządzanie złożonością usługi,
  dziel i rządź!
- Level 2: standardowe operacje,
  podobne sytuacje obsługiwane w podobny sposób
- Level 3: eksplorowalność usługi

HATEOAS – zalety

- Łatwiejsze rozbudowywanie API (upgradeability)
  - Szczególnie gdy jest wiele aplikacji klienckich…
  - …rozwijanych przez wielu dostawców
  - Struktura adresów jest pod kontrolą serwera
- Solidniejsze aplikacje klienckie
  - Operacje wywoływane przez relacje hipertekstowe
  - Brak hard-kodowanych adresów
  - Jeśli adres się zmienił, serwer wyśle go w kolejnej odpowiedzi
    - …a klient nawet tego nie zauważy
  - **Adresy nie mają znaczenia dla klientów!**

HATEOAS – zalety

- Eksplorowalność API
  - Każda odpowiedź zawiera informację o możliwych dalszych krokach
  - Niczym metody obiektów, zwracające inne obiekty, na których można wykonywać kolejne operacje
- Elastyczność
  - np. przeniesienie zasobów na inny serwer bez wpływu na aplikacje klienckie

HATEOAS – nie zawsze konieczny

- Tylko jedna, własna aplikacja kliencka
  - Jeśli coś się popsuje, naprawimy to
- Brak potrzeby elastyczności
  - np. aplikacje do użytku wewnętrznego
- Brak potrzeby eksplorowalności
  - np. API i aplikacja kliencka wytwarzane przez ten sam zespół lub nawet tego samego dewelopera
- Czasami łatwiej naprawić aplikacje klienckie niż starać się uczynić je odpornymi na zmiany

HATEOAS – kto tego używa?

- Dostawcy publicznych API
  - Wiele aplikacji klienckich, nad którymi autorzy API nie mają kontroli
- Przykładowo:
  - Netflix
  - GitHub
  - PayPal

REST

To nie tylko JSON wysłany po HTTP!

Usługi sieciowe

- Projektując usługi należy uwzględnić wymagania, takie jak:
  - Długoterminowe utrzymywanie API
  - Długoterminowy rozwój API
  - Niezależna ewolucja komponentów
    - Aplikacji klienckich i API po stronie serwera
  - Elastyczność API
- Model Richardsona określa wytyczne w celu osiągnięcia powyższych wymagań

Narzędzia do pracy z RESTowym API

- Do pracy z API konieczne są narzędzia do budowania żądań HTTP (GET/POST/PUT/DELETE)
  - Back-end deweloper chce rozwijać i testować API niezależnie od pracy front-end deweloperów
- Popularne narzędzia:
  - Postman
    - Aplikacja Chrome (do pobrania z Chrome Web Store)

      lub
    - Aplikacja *standalone* (do pobrania ze strony producenta)
  - SoapUI
    - *Standalone*

Pytania?