Architektura baz danych dla systemów skalowalnych

Witold Rakoczy

2019/2020

1

Zagadnienia

- Architektura systemu
- Skalowanie i skalowalność systemów
- Skalowalność a proces tworzenia systemu
- Rodzaje skalowania
- REST - podstawowe idee i cechy podejścia
  - Laboratorium
- Mikroserwisy
- Partycjonowanie baz danych

2

Co rozumiemy przez architekturę systemu?

- Statyczne a dynamiczne spojrzenie na system oprogramowania
- Różnica pomiędzy kodem systemu a obiektami, które
  - z niego powstają - gdy system startuje,
  - trwają - póki on działa
- Struktura kodu
- Architektura systemu

3

Co rozumiemy przez architekturę systemu?

- Struktura kodu systemu
  - Jest statyczna
  - Na poziomie elementu składowego systemu jest zdefiniowana w określonym języku programowania
  - Na poziomie środowiska uruchomieniowego jest wyznaczona przez wskazany przez programistę sposób składania elementów składowych (komponentów, modułów) w jedną całość, w ramach ograniczeń narzuconych przez reguły środowiska uruchomieniowego (sprzęt + system operacyjny, chmurę obliczeniową itp.)

4

Co rozumiemy przez architekturę systemu?

- Architektura systemu
  - Jest dynamiczna
  - Jest związana z wykonywaniem kodu w środowisku (jednym lub wielu) interpretującym kod („runtime”)
  - Zmienia się zgodnie z cyklem działania systemu
    - dla małych systemów odpowiada to cyklowi "uruchomienie - działanie - wyłączenie"
    - dla wielkich systemów, pracujących nieustannie i wykorzystujących zwielokrotnienie (zwłaszcza systemów chmurowych) poszczególne *instancje systemu* w tym samym czasie mogą pracować w konfiguracjach odpowiadających różnym architekturom

5

Architektura systemu

- Klasyczny cykl zmian architektury systemu
  - Zaczyna się od aktywacji pojedynczej „aktywności wykonawczej”, która może powoływać do życia kolejne aktywności (procesy) ***[start systemu]***
    - Nieco inaczej jest, gdy system jest rozproszony, ale nawet wtedy istnieje określona *sekwencja uruchamiania* poszczególnych części systemu i synchronizacji ich stanów
  - To początkowe działanie doprowadza do stanu aktywności ustaloną z góry (albo dobieraną w zależności od parametrów konfiguracyjnych systemu) liczbę *aktywnych instancji* poszczególnych składowych systemu (odpowiadających określonym częściom kodu i działających zgodnie z treścią tego kodu) ***[faza inicjacji systemu]***

6

Architektura stanu pracy systemu

- Po zakończeniu fazy inicjacji systemu, jest on gotowy do pracy, tj. zdolny przyjmować i obsługiwać napływające do niego zlecenia wykonania operacji (w ramach repertuaru usług przez niego dostarczanych) ***[architektura fazy normalnego działania]***
- Czasami wskutek usterki: nieoczekiwanego działania środowiska wykonawczego, na ogół pewnej jego części, albo wskutek błędu w samym systemie, dochodzi do awarii, co przejawia się w niezdolności do pracy jakiejś instancji składowej systemu (jednej lub wielu), co powoduje zmianę architektury ***[architektura stanu awarii]***

7

Architektura systemu w stanie awarii

- Architektura stanu awaryjnego nie jest trwała, bo system nie jest zdolny do normalnej pracy; zwykle kończy on działanie w mniej (załamanie systemu) lub bardziej uporządkowany sposób (awaryjne zatrzymanie systemu)
  - Alternatywnie, system może - zamiast całkowicie zaprzestać działania - zmienić sposób działania, np. udostępniając tylko podzbiór swojego normalnego zestawu usług lub funkcji *[architektura stanu pracy ograniczonej]*;
  - Z tego stanu może w pewnych okolicznościach (interwencja obsługi, naprawa lub samoistne przywrócenie prawidłowego działania środowiska) wrócić do fazy normalnej pracy np. przez reaktywację potrzebnych instancji składowych (np. przez ponowne uruchomienie odpowiednich składowych) *[zmiana architektury]*

8

Zmiana architektury podczas pracy

- Jeśli zasada działania systemu dopuszcza zatrzymanie świadczenia przezeń usług, system może:
  - zacząć ograniczać obsługę zleceń (np. przez zaprzestanie akceptowania nowych zleceń) i stopniowo wyłączać/zatrzymywać zbędne instancje swoich składowych, aż do chwili, kiedy cała aktywność systemu zaniknie *[zamykanie systemu]*
  - zmienić zakres świadczonych usług, tak aby niesprawna część nie musiała być używana *[modyfikacja zakresu usług]*

9

Architektury systemu

- Cykl zmian stanów pracy definiuje ***zbiór architektur*** systemu
  - Nie wszystkie muszą być przewidziane przez projektantów ;-)
  - Potrzeby dostosowania systemu do zmieniających się z upływem czasu wymagań prowadzą zwykle do zmiany tego zbioru
- Mimo, że w istocie zawsze istnieje zbiór architektur, odpowiadających różnym stanom pracy, dla uproszczenia mówimy o architekturze systemu

10

Architektura systemu a struktura kodu

- Kiedy architektura systemu dokładnie odzwierciedla strukturę jego kodu?
  - Jeden wątek wykonania w kodzie systemu
  - Jedna uruchamiana składowa (system monolityczny)
  - Jedna instancja składowej
- Taka sytuacja jest raczej rzadko spotykana (nawet dość prosty skrypt w bashu prawie zawsze definiuje system o bardziej złożonej architekturze)

11

Skalowalność a proces tworzenia systemu [1]

Zagadnienia:

- Na czym polega skalowanie systemu?
- Rodzaje skalowalności
- Wady tradycyjnej architektury systemu ujawniające się w trakcie rozwoju aplikacji

12

Adaptacja systemu w okresie jego życia

- W miarę jak zmieniają się warunki pracy systemu (najczęściej: rośnie obciążenie) dochodzi do sytuacji, w której bieżąca architektura okazuje się niewystarczająca (nawet mimo udostępniania coraz wydajniejszych wersji środowiska)
- Celem skalowania jest dostosowanie systemu architektury (a często i struktury kodu systemu) do nowych warunków

13

Skalowanie systemu

- *Skalowanie* to czynność**,** polegająca na modyfikacji systemu, nastawionej na usunięcie:
  - „wąskich gardeł", ograniczających przepustowość/wydajność systemu lub
  - innych wad systemu, zmniejszających odporność na negatywne oddziaływanie czynników przypadkowych (błędów w nim samym i usterek/awarii środowiska wykonawczego)
- Są systemy dające się łatwo skalować (nawet wielokrotnie) i takie, które przeskalować trudno od samego początku, albo dla których każdy kolejny cykl skalowania jest coraz kosztowniejszy

14

Skalowalność systemu

- *Skalowalność* systemu to jego cecha, mówiąca o stopniu łatwości, z jaką można przeprowadzać (najlepiej systematycznie i wielokrotnie, cyklicznie) *operację skalowania*, w miarę możności bez wzrostu jej pracochłonności, czasochłonności i innych kosztów
- Nie ma bezpośredniego związku pomiędzy prostotą architektury a skalowalnością, ale
- Są architektury źle skalowalne i takie, które ułatwiają skalowanie

15

Na czym polega skalowanie systemu?

- Dostosowanie systemu (jego architektury i także struktury kodu) do zmienionych warunków obciążenia; w najprostszych przypadkach np.
  - zmiana liczby uruchomionych kopii składowej systemu np. demona httpd
  - wprowadzenie w programie asynchronicznego wykonania operacji wejścia/wyjścia
  - wyodrębnienie (w postaci niezależnie działających procesów) aktywności komponentów, które mogą przebiegać niezależnie od siebie (jak np. w skrypcie startowym systemu Linux, gdy uruchamia się wiele kopii programu fsck sprawdzającego stan systemów plików, po jednej dla każdego urządzenia pamięci masowej)

16

Skalowanie systemu

- Obejmuje zarówno modyfikacje na poziomie kodu (optymalizacja), jak i działania pozwalające na zwiększenie stopnia zrównoleglenia operacji, wykonywanych przez komponenty systemu
  - (Zwykle) Wymaga odpowiedniego podziału kodu aplikacji na niezależne od siebie komponenty – nie zawsze aplikacja ma tę właściwość od początku
  - (Zwykle) Wymaga odpowiedniej struktury bazy danych, dobranej pod kątem ograniczenia konkurencji komponentów w dostępie do danych

17

Trudności w procesie skalowania

- W rozbudowanym, skomplikowanym systemie nie daje się zapewnić „z góry” ustalonego stopnia poprawy wydajności w jednym kroku adaptacji, ponieważ:
  - Szeroki zakres zmian powoduje znaczne koszty
  - Wymagania niezawodnościowe wymuszają znaczny nakład czasu
  - Wąskie gardła ujawniają się stopniowo, w miarę jak likwidowane są te wcześniej wykryte, w związku z czym warto modyfikować system cyklicznie, usuwając w każdym cyklu źródła najbardziej dotkliwych ograniczeń

18

Skalowalność a proces tworzenia systemu ²

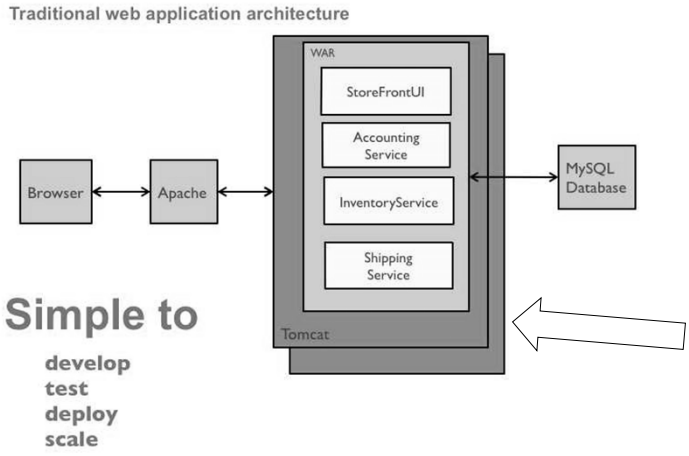
Zagadnienia:

- Na czym polega skalowanie systemu?
- Rodzaje skalowania
- Wady tradycyjnej architektury systemu ujawniające się w trakcie rozwoju aplikacji

19

Rodzaje skalowania - Przykłady

Przykład 1 – tradycyjna architektura aplikacji webowej



Sposób: zwielokrotnie całego oprogramowania transakcyjnego

20

Rodzaje skalowania - Przykłady

Przykład 1 (c.d.)

- Korzyści:
  - architektura prosta w budowie, testowaniu, wdrożeniu
  - łatwa do skalowania (ale w ograniczonym zakresie)
- Skalowanie poziome – zwielokrotnienie
  - ograniczony zakres
  - „gwarantowane” wąskie gardło – baza danych

21

Rodzaje skalowania - Przykłady

Przykład 2 – architektura systemu wykorzystującego zrównoleglenie działań i zwielokrotnioną bazę danych

- Założenia:
  - Znakomita większość danych jest „tylko do odczytu” (niezwykle rzadko/wcale nie modyfikowana)
  - Znaczna część zleceń wykorzystuje wyłącznie dane „tylko do odczytu”
  - Większość zleceń nie wymaga danych „z ostatniej chwili”
- Idea: system wykorzystuje zwielokrotnioną bazę danych w konfiguracji master-slave

22

Rodzaje skalowania - Przykłady

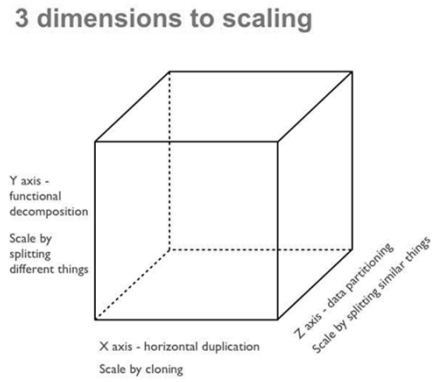
Przykład 2 (c.d.)

- Modyfikacje dokonywane na jednej kopii (*master*) są propagowane przez system bazodanowy do pozostałych kopii (*slaves*), które nie podlegają bezpośrednim modyfikacjom ze strony oprogramowania systemu
- Kopie bazy „tylko do odczytu” służą serwerom realizującym zlecenia nie wymagające modyfikacji zawartości bazy

23

Trzy rodzaje skalowania

- Pierwszy rodzaj skalowania - skalowanie poziome



- Skalowanie poziome – w osi X (skalowanie przez zwielokrotnienie - „klonowanie” aplikacji)
- Konieczny load balancer; każda kopia aplikacji obsługuje 1/N ruchu
- Problem z (potencjalną) potrzebą dostępu do wszystkich danych

24

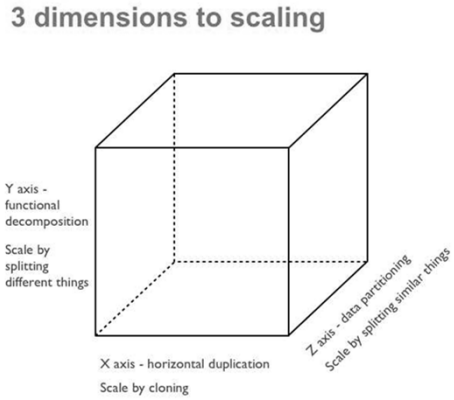
Trzy rodzaje skalowania

- Drugi rodzaj skalowania - skalowanie pionowe, czyli w osi Y (skalowanie przez dekompozycję funkcjonalną – rozdzielanie różnych rodzajów elementów pomiędzy odrębne składowe)
  - Aplikacja zostaje podzielona na liczne odrębne serwisy
  - Każdy serwis odpowiada za jedną lub kilka związanych ze sobą funkcji
  - Możliwe podziały:
    - według przypadków użycia (zorientowanie na czynności, np. wykonanie rezerwacji), albo
    - według obszaru aktywności (zorientowanie na obsługiwane jednostki/obiekty systemowe, np. zarządzanie klientami),
    - ewentualnie wariant mieszany

25

Trzy rodzaje skalowania

- Drugi rodzaj skalowania - skalowanie pionowe



- Skalowanie pionowe – w osi Y (skalowanie przez dekompozycję funkcjonalną – rozdzielanie na odrębne składowe różniących się elementów)

26

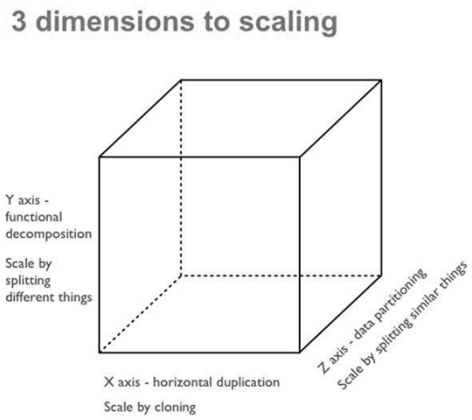
Trzy rodzaje skalowania

- Trzeci rodzaj skalowania – skalowanie w głąb czyli w osi Z (partycjonowanie danych)
  - Na każdym serwerze jest kopia systemu, ale dane obsługiwane/wykorzystywane przez nią są podzbiorem całego zestawu danych systemu.
  - Odpowiedni komponent systemu *(request router)* doprowadza zlecenie operacji do właściwego serwera, który jest wstanie je obsłużyć (w tym - posiada potrzebne dane dla jego obsługi)
    - przykładowo na podstawie zawartego w zleceniu klucza głównego obiektu, na którym ma być wykonana operacja, ustala się który z serwerów ma dane tego obiektu;
    - inny przykład – na podstawie atrybutu klienta wybiera się serwer, posiadający dane wszystkich klientów posiadających ten atrybut

27

Rodzaje skalowania

- Trzeci rodzaj skalowania – skalowanie w głąb



- Skalowanie w głąb – w osi Z (skalowanie przez partycjonowanie danych)

28

Rodzaje skalowania

Skalowanie w osi Z – zalety i wady

+ Ograniczenie objętości danych poprawia parametru techniczne (lepsze wykorzystanie pamięci cache serwera, redukcja łącznego wykorzystania jego pamięci głównej, zmniejszenie intensywności transmisji danych)
+ Poprawa skalowalności (zwłaszcza w połączeniu ze zwielokrotnieniem serwerów)
+ Poprawa stopnia izolowania usterek, ponieważ usterka jednego serwera ogranicza/blokuje dostęp do co najwyżej części danych
- Wzrost stopnia skomplikowania aplikacji
- Konieczność partycjonowania danych, co może być niełatwe, zwłaszcza, jeśli dopuścić (a zawsze trzeba…) konieczność zmiany sposobu partycjonowania danych w przyszłości
- Nie załatwia sprawy wzrostu utrudnień w miarę rozwoju aplikacji (to daje się zrobić tylko przez skalowani w osi Y)

29

Skalowalność a proces tworzenia systemu 3

Zagadnienia:

- Na czym polega skalowanie systemu?
- Rodzaje skalowalności
- Wady tradycyjnej architektury systemu ujawniające się w trakcie rozwoju aplikacji
  - Wady z punktu widzenia deweloperów
  - Wady z punktu widzenia ograniczeń procesu projektowego

30

Wady tradycyjnej architektury systemu

- Wady tradycyjnej, monolitycznej architektury systemu, ujawniające się w trakcie rozwoju aplikacji - *z punktu widzenia deweloperów*
  - Kod jest monolityczny i obszerny
    - przeraża/utrudnia zrozumienie i kolejne modyfikacje;
    - spowalnia proces rozbudowy/rozwoju;
    - nie ma wyraźnych granic pomiędzy składowymi, co prowadzi do osłabiania modularności (łatwo wprowadzać zmiany psujące ją)
  - Spowalnia działanie narzędzi deweloperskich (IDE)
  - Spowalnia uruchamianie (strata czasu przy starcie)
  - Utrudnia ciągły proces budowy (każda zmiana w komponencie wymaga rozmieszczenia całej aplikacji), co wydłuża okresy przestoju/niedostępności systemu przy dokonywaniu zmian.

31

Wady tradycyjnej architektury systemu

- Wady architektury systemu, ujawniające się w trakcie rozwoju aplikacji - *z punktu widzenia organizacji procesu projektowego*
  - Skalowanie staje się z czasem coraz trudniejsze (stosować można tylko skalowanie poziome)
  - Rosnąca wielkość aplikacji wymusza podział całego zespołu na współpracujące ze sobą mniejsze, ale nie mogą one pracować niezależnie, co powoduje pogorszenie sprawności prowadzenia projektu
  - Raz (na początku) wybrany stos technologiczny nie może być łatwo zmieniony, nawet gdy pojawiają się lepsze technologie/platformy; decyzja o podjęciu zmiany platformy jest z natury bardzo ryzykowna.

32

Co można zmienić, aby uniknąć wad architektury monolitycznej?

- Zagadnienia
  - Komunikacja z użyciem RESTful API
  - Zastosowanie architektury opartej o mikroserwisy
  - Partycjonowanie baz danych
  - Wzorce operowania na bazach danych

33

Komunikacja z użyciem RESTful API

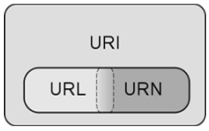
- REST
  - Obejmuje zbiór reguł i zaleceń odnośnie budowy API dla właściwie dowolnych usług
  - Implementacja tych wskazówek odpowiada realizacji zasady *Smart endpoints and dumb pipes*, ponieważ zakłada możliwie najprostsze środki realizacji komunikacji

34

Komunikacja z użyciem RESTful API

Reguły/założenia bazowe

- (Reguła 1) Wszystko jest zasobem
- (Reguła 2) Każdy zasób jest identyfikowany przez URI (Uniform Resource Identifier, Ujednolicony Identyfikator Zasobu) – w formie URL lub URN (w praktyce przeważnie URL)



- https://docs.iisg.agh.edu.pl/labs/config.html
- http://www.wikipedia.org:8080
- http://www.google.com/search?hl=pl&q=uri
- http://www.mydatastore.com/data/documents/balance?date=current

35

Komunikacja z użyciem RESTful API

Reguły/założenia bazowe (c.d.)

- (Reguła 3) Aplikacja do komunikacji z serwerem korzysta z (podzbioru) standardowych metod HTTP
- (Reguła 4) Zasoby mogą mieć zwielokrotnione reprezentacje.
- (Reguła 5) Komunikacja pomiędzy usługobiorcą (użytkownikiem usługi, zleceniodawcą) a usługą (dostawcą usługi, serwerem) musi być bezstanowa (serwer nie pamięta stanu zaawansowania konwersacji w ramach kroków realizacji zlecenia, zatem cała informacja potrzebna do tego celu musi być dostarczana w zleceniu i zwracana w wynikach)

36

Komunikacja z użyciem RESTful API

Zasób

- Idea zasobu danych odwołuje się do formatu danych jako podstawowego czynnika definiującego charakter zasobu; format danych mówi o typie zawartości zasobu
- Obrazy JPEG, video MPEG, dokumenty w html, xml, tekstowe, dane binarne - mają odpowiednie typy zawartości: image/jpeg, video/mpeg, text/html, text/xml, application/octet-stream

37

Komunikacja z użyciem RESTful API

Metody HTTP

- HEAD, OPTIONS, TRACE, CONNECT – nie są wykorzystywane
- GET, POST, PUT, DELETE – używane w naturalny sposób w kontekście operowania na zasobach; użyte właściwie pozwalają zlecić bazodanowe operacje CRUD

38

Komunikacja z użyciem RESTful API

Metody (c.d.)

- Semantyka operacji dla metod GET, POST, PUT, DELETE

HTTP verb	Action	Response status code
GET	Request an existing resource	"**200 OK**" if the resource exists, "**404 Not Found**" if it does not exist, and "**500 Internal Server Error**" for other errors
PUT	Create or update a resource	"**201 CREATED**" if a new resource is created, "**200 OK**" if updated, and "**500 Internal Server Error**" for other errors
POST	Update an existing resource	"**200 OK**" if the resource has been updated successfully, "**404 Not Found**" if the resource to be updated does not exist, and "**500 Internal Server Error**" for other errors
DELETE	Delete a resource	"**200 OK**" if the resource has been deleted successfully, "**404 Not Found**" if the resource to be deleted does not exist, and "**500 Internal Server Error**" for other errors

39

Komunikacja z użyciem RESTful API

Metody (c.d.)

- Przykład komunikacji – zapis zasobu (metoda PUT)

```
# Resource creation in specified location
PUT /data/documents/balance/22082014 HTTP/1.1
Content-Type: text/xml
Host: www.mydatastore.com
<?xml version="1.0" encoding="utf-8"?>
<balance date="22082014">
<Item>Sample item</Item>
    <price currency="EUR">100</price>
</balance>
HTTP/1.1 201 Created
Content-Type: text/xml
Location: /data/documents/balance/22082014
```

40

Komunikacja z użyciem RESTful API

Metody (c.d.)

- Istnieje szczególny przypadek (kreowanie zasobu bez wskazania miejsca, gdzie ten zasób ma zostać umieszczony, aby serwer podjął decyzję sam), w którym sensowne jest użycie POST zamiast PUT

```
# Resource creation in unspecified location
POST /data/documents/balance HTTP/1.1
Content-Type: text/xml
Host: www.mydatastore.com
<?xml version="1.0" encoding="utf-8"?>
<balance date="22082014">
<Item>Sample item</Item>
    <price currency="EUR">100</price>
</balance>
HTTP/1.1 201 Created
Content-Type: text/xml
Location: /data/documents/balance
```

41

Komunikacja z użyciem RESTful API

Zwielokrotniona reprezentacja zasobu

- Zasób może być dostarczony w dowolnym formacie akceptowanym (obsługiwanym) przez punkt końcowy (*Smart endpoints and dumb pipes*)

```
#Alternative representation data - JSON
POST /data/documents/balance HTTP/1.1
Content-Type: application/json
Host: www.mydatastore.com
{
  "balance": {
    "date": ""22082014"",
    "Item": "Sample item",
    "price": {
      "-currency": "EUR",
      "#text": "100"
    }
  }
}
HTTP/1.1 201 Created
Content-Type: application/json
Location: /data/documents/balance
```

Alternatywna reprezentacja danych z poprzedniego przykładu

42

Komunikacja z użyciem RESTful API

- Bezstanowa komunikacja - Założenia
  - Operacje manipulacji na zasobach poprzez żądania HTTP muszą być zawsze uważane za niepodzielne (atomowe).
  - Wszystkie modyfikacje jakiegoś zasobu powinny być przeprowadzane wewnątrz operacji HTTP w izolacji.
  - Po wykonaniu operacji HTTP zasób jest pozostawiany w jego końcowym stanie, co domyślnie oznacza, że częściowe („na raty”) modyfikacje zasobu nie są wspierane.
  - Zawsze należy wysyłać „cały” zasób (kompletny stan zasobu).

43

Komunikacja z użyciem RESTful API

- Bezstanowa komunikacja – Założenia (c.d.)
  - Udostępniane API aplikacji będzie wykorzystywane przy założeniu, że jest bezstanowe, zatem takie właśnie być musi
  - Oprogramowanie, realizujące poszczególne operacje w odpowiedzi na użycie metod HTTP, nie pamięta (=”nie może pamiętać”) niczego, co zdarzyło się wcześniej
    - w końcu nie wiadomo, do których kopii programu *load balancer* będzie kierować kolejne żądania klienta

44

REST – komunikacja bezstanowa

- Cechy podejścia opartego na idei REST
  - Separacja reprezentacji zasobu od samego zasobu
  - Obserwowalność (visibility)
  - Niezawodność (reliability)
  - Skalowalność (Scalability) i wydajność (Performance)
  - Wspomaganie tworzenia aplikacji RESTowych

45

REST – cechy podejścia

Separacja reprezentacji zasobu od samego zasobu

- Użytkownik podaje typ oczekiwanej zawartości zasobu w żądaniu (GET) dostarczenia zasobu (np:
  Accept: application/json
  zaś serwer albo potrafi poradzić sobie z takim typem zawartości (i odpowiada HTTP 200 OK), albo nie (i zgłasza błąd HTTP 400)

46

REST – cechy podejścia

Obserwowalność (visibility)

- Obserwowalność usługi oznacza - z punktu widzenia świata zewnętrznego – że monitorowanie działania tej usługi wymaga jedynie obserwowania komunikacji pomiędzy zlecającym operacje a serwerem
- Ponieważ mamy do czynienia z bezstanowością żądań i odpowiedzi, niczego więcej nie potrzeba, aby podążać za tokiem działań obu stron i ewentualnie ustalić, co poszło źle

47

REST – cechy podejścia

Niezawodność (reliability)

- Niezawodność zależy od
  - bezpieczeństwa (safety) [safe – bezpieczny w użyciu] i
  - idempotentności *) metod HTTP
- Metoda HTTP jest uważana za bezpieczną, jeśli jej wywołanie nie dokonuje modyfikacji zasobu, ani nie ma jakichkolwiek ubocznych efektów dla stanu tego zasobu
- Metoda HTTP jest uważana za idempotentną, jeśli udzielana przez nią odpowiedź jest zawsze taka sama, niezależnie od liczby wykonań (można operację powtórzyć bez zauważalnych skutków)
- Cechy metod HTTP

HTTP Method	Safe	Idempotent
GET	Yes	Yes
POST	No	No
PUT	No	Yes
DELETE	No	Yes

48

REST – cechy podejścia

Skalowalność (Scalability) i wydajność (Performance)

- Skalowalność pozioma jest zapewniona z definicji – dodanie kolejnego serwera za *load balancerem* nie powoduje zmiany sytuacji w działaniu systemu
- Wydajność – rozumiana jako nakład pracy systemu (mierzony czasem wykonania dla pojedynczego zlecenia) zależy od
  - efektywności algorytmów,
  - dysponowanej mocy obliczeniowej,
  - prędkości pamięci masowej,
  - efektywności systemu operacyjnego itp.
- Przynajmniej część powyższych właściwości pozostaje pod kontrolą deweloperów.

49

REST – cechy podejścia

Wspomaganie tworzenia aplikacji RESTowych

- RESTful API jest podejściem dojrzałym - w użyciu od kilkunastu lat
- Istnieje wiele narzędzi wspomagających deweloperów w procesie tworzenia webowych aplikacji RESTowych, łącznie z bibliotekami i platformami, pozwalających standaryzować i częściowo automatyzować ich budowę.
- Wspomaganie formalnego opisu aplikacji RESTowych jest możliwe (choć nieobowiązkowe) dzięki istnieniu języka definicji aplikacji webowych (WADL - Web Application Definition Language)

50

[Laboratorium]

W ramach zajęć laboratoryjnych uczestnicy:

- zbudują (korzystając z gotowych narzędzi) prymitywny serwer posiadający RESTful API
- wypróbują podstawowe operacje odwołując się do serwera za pośrednictwem metod - zgodnie z jego API
- będą mogli wypróbować modyfikacje API

Do zajęć potrzebne będą elementarne umiejętności w zakresie języka Javascript (przy modyfikacji API)

51

Mikroserwisy

Mikroserwisy jako fundamentalna koncepcja wykorzystywana dla zapewniania skalowalności (1)

- Podstawowa idea wykorzystuje następujące spostrzeżenia:
  - W zasadzie każdy sposób podziału systemu na równolegle działające części sprzyja poprawie przepustowości/wydajności tego systemu
  - Nie każdy sposób podziału jest równie dobry z punktu widzenia *organizacji procesu projektowego* i trudności w realizacji zadań stojących przed deweloperami

52

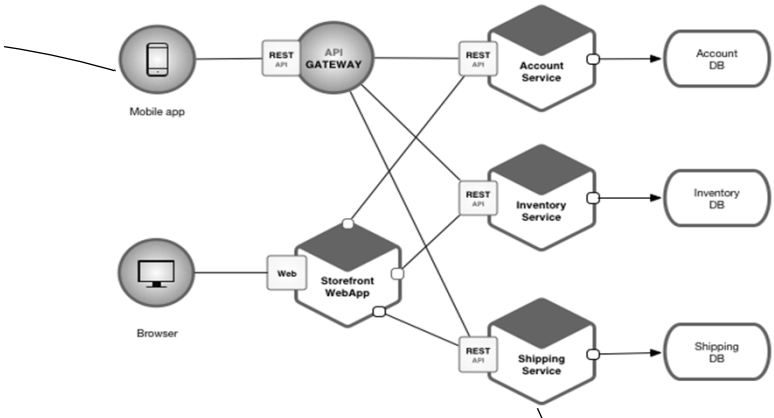
Mikroserwisy

Mikroserwisy jako fundamentalna koncepcja wykorzystywana dla zapewniania skalowalności (2)

- Mikroserwisowe podejście korzysta z połączenia skalowania w osi Z i skalowania w osi Y:
  - składniki oprogramowania działające na poszczególnych częściach podzielonej bazy danych nie są jednakowe;
  - są bowiem wyspecjalizowane pod kątem operacji, które realizują
- Następny przykład pokazuje w jaki sposób można podzielić na mikroserwisy aplikację e-sklepu, łącząc komponenty, wyspecjalizowane w realizacji usług składowych, za pomocą prostego i „lekkiego” mechanizmu komunikacyjnego.

53

Mikroserwisy



Dekompozycja struktury bazy danych zapewnia niezależność usług podstawowych. Natomiast usługi finalne w ogóle nie odwołują się bezpośrednio do pamiętanych danych – zależnie od sposobu komunikowania się z użytkownikiem udostępniają usługę systemu zbudowaną na bazie usług dostarczany przez pozostałe mikroserwisy.

54

Mikroserwisy – Zalety podejścia

+ Podejście mikroserwisowe pozwala organizować komponenty zorientowane na usługi

  System jest zbiorem serwisów, z których każdy dostarcza określonych usług, podczas gdy sam może korzystać z usług innych serwisów.

+ Sprzyja niezależności procesu tworzenia poszczególnych komponentów

  Likwiduje bowiem znaczną część wspomnianych wcześniej ograniczeń projektowych.

55

Mikroserwisy – Zalety podejścia

+ Komponenty realizujące funkcje mikroserwisów stanowią wymienialne niezależnie i aktualizowalne oddzielnie składniki systemu, umożliwiając podmianę/instalację tylko zmodyfikowanych komponentów, bez wszystkich pozostałych, które nie zostały one zmienione.

+ Dzięki niezależności redukcji ulega czas przestojów, potrzebnych przy adaptacji systemu (w szczególności jego skalowaniu), pozwalając potencjalnie na realizację polityki „(prawie) zero przestojów” (*close-to-zero downtime*)

56

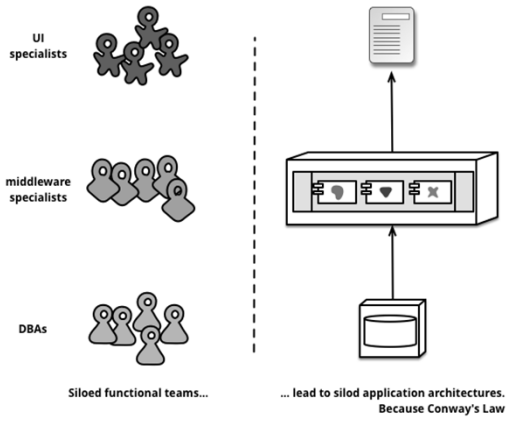
Mikroserwisy – Zalety podejścia

+ Mikroserwisy mają dobrze zdefiniowane zadania biznesowe i ich konstruowanie – wymagając zespołów obejmujących pełny przekrój specjalistów (UI, DB, PM) – sprzyja powstawaniu serwisów skupionych wokół realnych zadań. Pozwala to uniknąć pułapki prawa Convay'a:

57

Mikroserwisy – Zalety podejścia

Prawo Convay'a



58

Mikroserwisy – Zalety podejścia

+ Mikroserwisy pozwalają łączyć zespół realizatorów z ich produktem (mikroserwisem) nie na czas samego projektu, lecz na cały okres życia produktu

  unikanie podziału na dział rozwoju i dział eksploatacji; idea DevOps

59

Mikroserwisy – Zalety podejścia

+ Mikroserwisy nie wymagają zachowania spójności technologicznej

  Jedyny interfejs wykorzystuje komunikację opartą na bardzo prostych zasadach.

+ Zachowanie spójności technologicznej przestaje być ograniczeniem

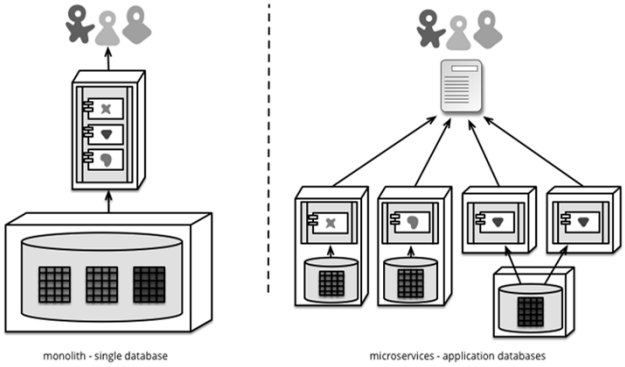
  Pozostaje jednak dalej celem, do którego warto dążyć, gdy przynosi korzyści (cecha zdecentralizowanego zarządzania, *decentralized governance*).

60

Mikroserwisy – Zalety podejścia

+ Pozwalają na zdecentralizowane zarządzanie danymi

Widok danych nie musi być taki sam dla wszystkich składników systemu, np. pewne elementy danych nie zawsze są widoczne, albo są prezentowane na wiele sposobów.



61

Mikroserwisy – Wady podejścia

- Za zdecentralizowane zarządzanie danymi płacimy koniecznością wprowadzania do systemu działań odpowiadających rozwiązywaniu sytuacji konfliktowych na poziomie transakcji biznesowych

Musimy być w stanie wycofać się ze „ślepego zaułka" operacji, ponieważ przy działaniu na wielu bazach danych nie mamy możliwości wykorzystania transakcji bazodanowych (ACID).
To jednak w przypadku wielu różnorodnych usług powodują znaczne spowolnienie działania systemu.

62

Mikroserwisy – Wady podejścia

- Aplikacje korzystające z mikroserwisów muszą liczyć się z ewentualności ich awarii i stosownie do tego reagować

  Reakcja powinna być możliwie mało kłopotliwa dla użytkownika, co na ogół nie musi być łatwe do uzyskania.

63

Mikroserwisy – Wady podejścia

- Dekompozycja usług końcowych i usług wewnętrznych systemu wymaga solidnej, sprawdzonej wiedzy o potrzebach w tym zakresie

  Dlatego lepiej nie próbować rozpoczynać projektowania nowego systemu z użyciem mikroserwisów: warto ich użyć dopiero, gdy monolityczna wersja systemu jest dojrzała lub staje się trudna w skalowaniu (*strangler pattern*, Martin Fowler).

64

Stosowanie mikroserwisów

Co jest potrzebne, aby z powodzeniem budować udane aplikacje w tej architekturze?

- Technologie gwarantujące wysokiej jakości rozwiązania w zakresie implementacji komunikacji ***[są dostępne]***
- Techniki przeprowadzania adaptacji baz danych od postaci monolitycznej, odpowiedniej dla systemów (prawie) monolitycznych, do podzielonej (metody partycjonowania baz danych) ***[są znane]***
- Dobrze rozpoznany obszar i zrealizowany projekt, dla którego pojawia się potrzeba skalowania i/albo dalszej rozbudowy systemu, a stopień skomplikowania zaczyna blokować możliwości efektywnego prowadzenia prac deweloperskich *[najlepiej zacząć od istniejącej aplikacji monolitycznej]*

65

Literatura do poprzedzającego materiału

- Przykładowa literatura i źródła

- Bojinov V., Design and implement comprehensive RESTful solutions in Node.js, Packt Publishing, 2015
- Newman S., Building Microservices. Designing Fine-Grained Systems, O'Reilly Media, 2015
- Yanaga E., Migrating to Microservice Databases. O'Reilly, 2017
- http://microservices.io/ , pobrano 2017-12-10

66

Koniec materiału podstawowego

Uwagi:

- Materiały do zajęć laboratoryjnych (i maszyna wirtualna przygotowana dla ułatwienia, która jednak nie jest niezbędna) będą dostępne przed zajęciami,*) tak aby każdy mógł się zapoznać z przebiegiem ćwiczenia i w razie potrzeby zapoznać się z Javascriptem w potrzebnym (minimalnym…) zakresie.
- Kartkówka obejmować będzie tematyczny zakres wykładu (ze szczególnym uwzględnieniem podstawowych pojęć i zagadnień dot. skalowalności i RESTful API)

*) co najmniej tydzień przed pierwszymi zajęciami

67

A co jest dalej?

Można zajrzeć ☺

68

Partycjonowanie baz danych

Zmiany struktury baz danych w procesie realizacji/rozwoju systemu

- Jednym z powodów, dla których zawsze korzystna jest dekompozycja systemu na (w miarę) niezależne części, jest zmniejszenie objętości części systemu zmienianej podczas cyklu modyfikacji: nowa wersja mniejszej części wnosi mniej błędów…
- Jednak zmieniana część nie może być dowolnie mała - musi obejmować wszystkie powiązane poprawki w kodzie, jak również modyfikacje środowiska, w tym struktury bazy danych.

69

Partycjonowanie baz danych

Zmiany struktury baz danych w procesie realizacji/rozwoju systemu (c.d.)

- Reguła: Należy redukować rozmiar wsadu modyfikacji (części zmienianej) do minimalnej dopuszczalnej z punktu widzenie jej skuteczności i możliwości oddzielnego wprowadzenia.
- Ostrzeżenie: różnice pomiędzy środowiskami: deweloperskim i produkcyjnym, są źródłem najgorszych, najtrudniejszych do usunięcia błędów. *[SOA#1 :-)]*

70

Partycjonowanie baz danych

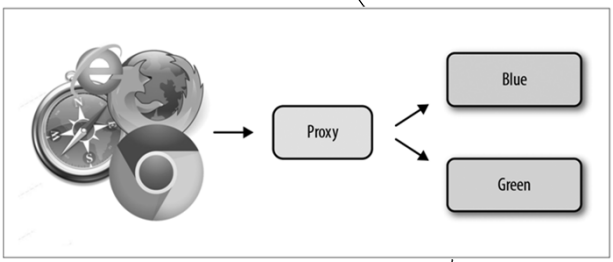
Zmiany struktury baz danych w procesie realizacji/rozwoju systemu (c.d.)

- Modyfikacje zmierzające do dekompozycji (np. wprowadzenie architektury mikroserwisowej) oprócz znacznych zmian w kodzie prowadzą do zmiany struktur danych, w tym podziału jednej, złożonej bazy danych na mniejsze, niezależne technicznie, choć dalej powiązane logicznie *[partycjonowanie]*
- Proces modyfikacji struktur baz danych obejmuje wiele operacji, które choć wydają się proste (jak np. przemianowanie kolumny w tabeli), gdy rozważać je jako element procesu wprowadzania zmian ***w systemie produkcyjnym***, okazują się kłopotliwe

71

Partycjonowanie baz danych

Proces wdrożenia zmian powinien być realizowany w sposób redukujący do minimum zakłócenia w pracy systemu produkcyjnego, stąd często stosowana architektura z systemem produkcyjnym zawierającym serwer proxy i dwa systemy wykonawcze: z bieżącą wersją i z nową wersją systemu i przełączaniem przekierowania pomiędzy systemami (w bieżącej wersji i w wersji nowej).



72

Partycjonowanie baz danych

Zmiany struktury baz danych w procesie realizacji/rozwoju systemu (c.d.)

- Ponieważ systemy - bieżący i nowy - nie są niezależne (muszą składować dane tak, aby oba mogły działać na tych samych strukturach danych), baza danych i obie wersje oprogramowania muszą być przygotowane do takiego sposobu operowania na bazie danych, aby zachować kompatybilność w obu kierunkach.
- Co więcej, proces wprowadzania zmian w kodzie musi być zsynchronizowany (przeplatać się) z modyfikacjami schematu bazy

73

Partycjonowanie baz danych

Migracja pomiędzy wersjami systemu

- Zakładając, że aplikacja jest zbudowana jako RESTowa, możemy dokonać zmiany oprogramowania „w locie”. Nie dotyczy to jednak danych.
- Aby zmiana bez przestojów była możliwa musimy migrację bazy danych przeprowadzać krokowo, modyfikując schemat bazy i dane w niej zawarte przy uwzględnieniu zachowania kompatybilności i bez (widocznego dla użytkowników) blokowania systemu na dłuższy czas
- Stosowny algorytm postępowania musi zatem uwzględniać konsekwencje danej operacji dla obciążenia SZBD

74

Migracja pomiędzy wersjami systemu [1]

Przykład: Zmiana nazwy kolumny tabeli

- **Chcemy przeprowadzić zmianę schematu bazy:**

```
ALTER TABLE customers RENAME COLUMN wrong TO correct;
```

- **Operacja jest destrukcyjna – tracimy informacje pamiętane w starej kolumnie – bieżąca wersja systemu nie będzie mogła po zmianie pracować**
- **Operacja wymaga dużo czasu – niektóre SZBD zablokują podczas jej wykonania całą tabelę, efektywnie zatrzymując działanie systemu**

75

Migracja pomiędzy wersjami systemu

Aby przyprowadzić migrację właściwie musimy wykonać ją w wielu krokach

```
ALTER TABLE customers ADD COLUMN correct VARCHAR(20);
UPDATE customers SET correct = wrong
   WHERE id BETWEEN 1 AND 100;
…
UPDATE customers SET correct = wrong
   WHERE id BETWEEN 99901 AND 100000;
ALTER TABLE customers DELETE COLUMN wrong;
```

Powyższy schemat wydaje się prosty, ale jego dosłowne użycie jest dalekie od praktyki: nie zapewnia on kompatybilności wstecz - skasowanie kolumny uniemożliwi powrót do poprzedniej wersji systemu

76

Migracja pomiędzy wersjami systemu 2

Przykład: Modyfikacja wartości danych kolumny

Chcemy przemnożyć wartość pola dla wszystkich rekordów tabeli:

```
UPDATE Account SET amount = amount * 1.1;
```

Aby zredukować czas zablokowania systemu stosujemy fragmentację operacji:

```
UPDATE Account SET amount = amount * 1.1
    WHERE number BETWEEN 1 AND 200000;
UPDATE Account SET amount = amount * 1.1
     WHERE number BETWEEN 200001 AND 400000;
UPDATE Account SET amount = amount * 1.1
     WHERE number BETWEEN 400001 AND 600000;
UPDATE Account SET amount = amount * 1.1
     WHERE number BETWEEN 600001 AND 800000;
UPDATE Account SET amount = amount * 1.1
     WHERE number BETWEEN 800001 AND 1000000;
```

…sprawdziwszy uprzednio, że czas wykonania operacji dla 200 tysięcy rekordów jest akceptowalny.

77

Migracja pomiędzy wersjami systemu

Schemat migracji - Dodanie kolumny

(Krok 1) ALTER TABLE ADD COLUMN… - Dodanie kolumny do tablicy

Uwaga: nie wolno w tym momencie dodać frazy NOT NULL nawet, gdy nowy model danych tego wymaga, ponieważ uniemożliwi to wykonanie operacji INSERT/UPDATE w dotychczasowej wersji systemu (która nic „nie wie” o nowej kolumnie)

Uwaga2: wprowadzenie w schemacie bazy frazy NOT NULL będzie możliwe po przeprowadzenia wszystkich potrzebnych zmian w bazie.

(Krok 2) Nowa wersja kodu może pobierać zawartość nowej kolumny, ale nie może zakładać, że ona istnieje (wtedy może dodać wartość domyślną, albo wyliczyć ją na podstawie informacji dostępnych dla aplikacji)

(Krok 3) Wykonanie operacji UPDATE dodającej wartości w nowej kolumnie, z wykorzystaniem fragmentacji operacji

(Krok 4) Kolejna wersja kodu może korzystać z wartości nowej kolumny.

78

Migracja pomiędzy wersjami systemu

Schemat migracji - przemianowanie kolumny

(Krok 1) ALTER TABLE ADD COLUMN
Dodanie nowej kolumny z docelową nazwą.
Uwaga: obowiązuje ograniczenie użycia frazy NOT NULL – jak dla Dodania kolumny

(Krok 2) Nowa wersja kodu pobiera wartości ze starej kolumny, natomiast gdy zapisuje, to pisze do obu (nowe rekordy będą zawsze miały obie wartości)

(Krok 3) Wykonanie kopiowania wartości ze starej kolumny do nowej, z wykorzystaniem fragmentacji; oprogramowanie dalej czyta wartość ze starej kolumny, ale po tej operacji wszystkie rekordy będą miały wartość w nowej kolumnie

79

Migracja pomiędzy wersjami systemu

Schemat migracji - przemianowanie kolumny (c.d.)

(Krok 4) Kolejna wersja kodu czyta wartość z nowej kolumny, ale zapisuje dalej do obu (to gwarantuje kompatybilność wsteczną – można wrócić do poprzedniej wersji kodu, jeśli coś pójdzie nie tak)

(Krok 5) Kolejna wersja kodu czyta i zapisuje wartości tylko w nowej kolumnie (wprowadzenie tej wersji zakłada, że poprzednia została uznana za dobrą – zakończono jej testowanie); w tym momencie operacja przemianowania została zakończona

(Krok 6) *(Dużo później)* Skasowanie starej kolumny (wymaga posiadania systemu pozwalającego odnotowywać dokonywane zmiany i zapisać potrzebę przypomnienia o konieczności ich wprowadzenia w przyszłości ;-))

80

Migracja pomiędzy wersjami systemu

Schemat migracji - Zmiana typu/formatu danych w kolumnie

Operację przeprowadza się według schematu dla przemianowania kolumny, biorąc pod uwagę sposób konwersji postaci/zawartości danych wykonywanej w Kroku 3 w stosunku do Kroku 2

81

Migracja pomiędzy wersjami systemu

Schemat migracji - Kasowanie kolumny

(Krok 1) Nowa wersja oprogramowania nie pobiera zawartości kolumny, ale ją zapisuje, tak jak to robiła poprzednia.

(Krok 2 – opcjonalny) Usunięcie frazy NOT NULL dla kolumny. Kolejna wersja oprogramowania przestaje zapisywać wartości w kolumnie.

(Krok 3) *(Dużo później)* Skasowanie kolumny.

Analogicznie jak dla Przemianowania kolumny operacja powinna być wykonana po dostatecznie długim czasie, ewentualnie podczas rutynowej pielęgnacji bazy (przy wyłączonym systemie) – razem z innymi podobnymi operacjami.

Operacja jest niszcząca, zatem musi być przeprowadzana tylko w razie konieczności i z dużą ostrożnością (posiadanie backupu nie oznacza, że można szybko przywrócić starą wersję schematu, a utrata danych i tak może nastąpić). NIGDY nie należy kasować kolumny od razu, gdy wprowadzany nową wersję oprogramowania, która jej nie potrzebuje.

82