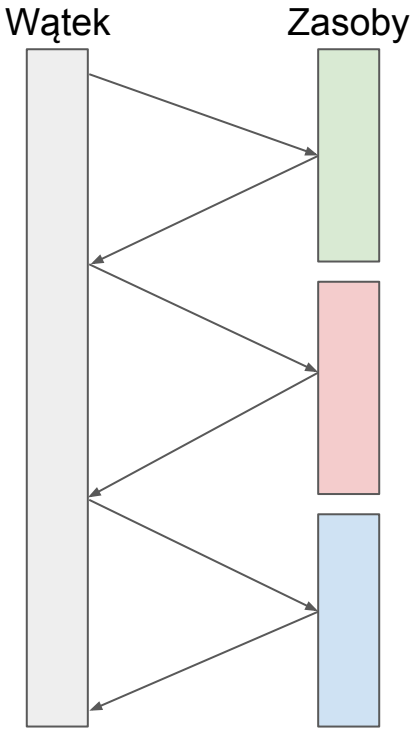
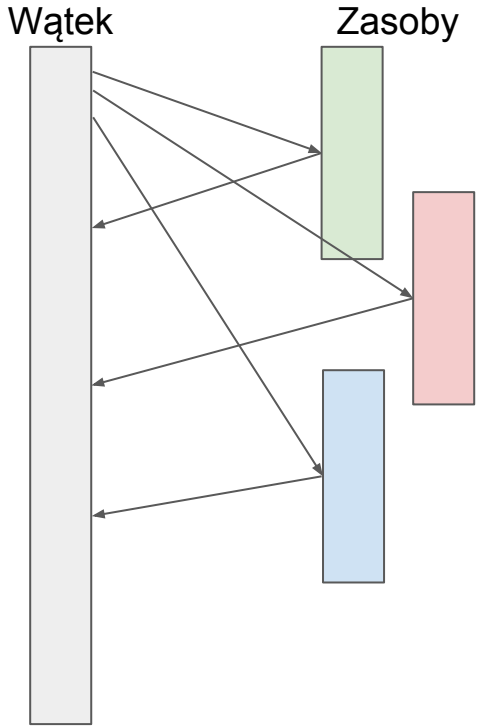
Patterny reaktywne i mikroserwisy

Maksym Blazhyievskyi
Michał Gabryś
Tetiana Zhvanko
Karol Gostek
Konrad Lempart

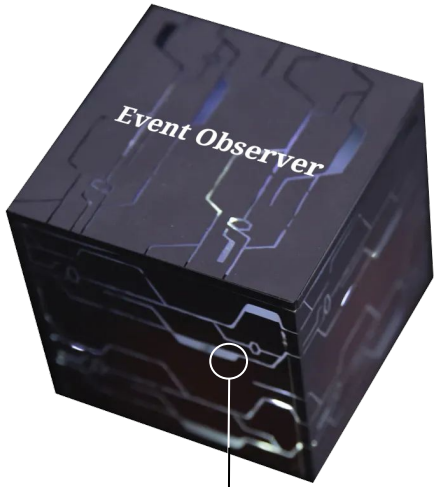
Podejście synchroniczne



Podejście asynchroniczne



Zdarzenia




```
if (event.type == "ScrollEvent") {
        handleScrollEvent(event.data)
} else if (event.type == "ButtonClickEvent") {
        handleButtonClickEvent(event.data)
} else {
        // ...
}
```


data: {
whichButton: "BigRed",
clickForce: 7,
userLoggedIn: true,
formInputValue: "abc123"
}

Zalety

- Asynchroniczność
- Wywołanie procedury z dowolnego miejsca
- Możliwość uporządkowania logiki
- Wyższy poziom abstrakcji

Wady

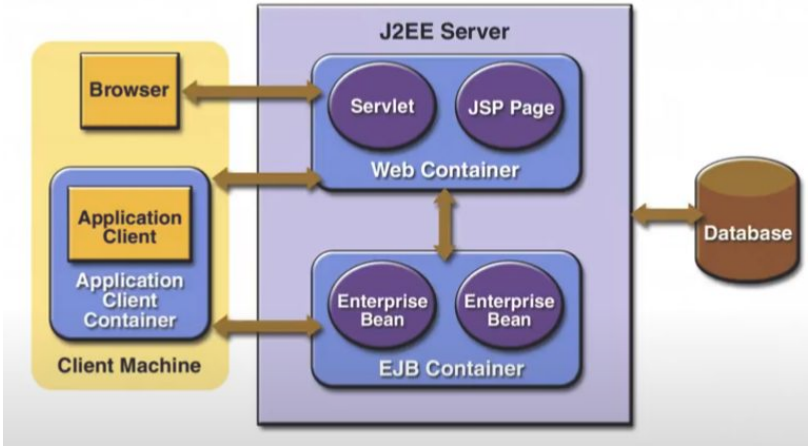
- Nie zwraca wartości
- Łatwo zrobić bałagan

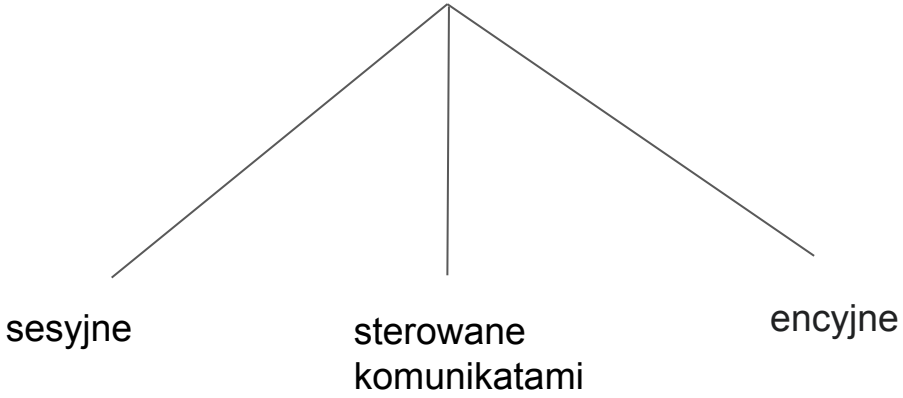
Asynchroniczne EJB

EJB - technologia działająca po stronie serwera, która "kapsułkuje" logikę biznesową aplikacji. Logika biznesowa to część kodu, która spełnia cel aplikacji. Nie zajmuje się prezentowaniem danych biznesowych czy bezpośrednią obsługą bazy danych. Zastosowanie zwalnia użytkownika EJB z konieczności opracowywania własnych metod obsługi komponentów.

Kontener EJB zarządza ziarenkami korporacyjnymi, które są w nim zawarte. Dla każdego ziarna, kontener jest odpowiedzialny za rejestrowanie obiektu, zapewnienie zdalnego interfejsu dla obiektu, kreacją i destrukcją instancji obiektu oraz zarządzaniem aktywnym stanem obiektu.

Główną zaletą EJB jest nakierowanie projektanta na pewne sprawdzone sposoby rozwiązania typowych problemów w systemie rozproszonym: zarządzanie połączeniami, transakcja rozproszona, mapowanie danych na model obiektowy itp.





- Ziarna sesyjne są używane do umieszczania w nich logiki aplikacji - kodu, który przetwarza dane.
- Encyjne EJB reprezentują w sposób obiektowy dane (np. dostarczają obiektowego spojrzenia na relacyjną bazę danych)
- Ziarna sterowane komunikatami znajdują zastosowanie w przetwarzaniu asynchronicznym i w zaawansowanych modelach współpracy oprogramowania.

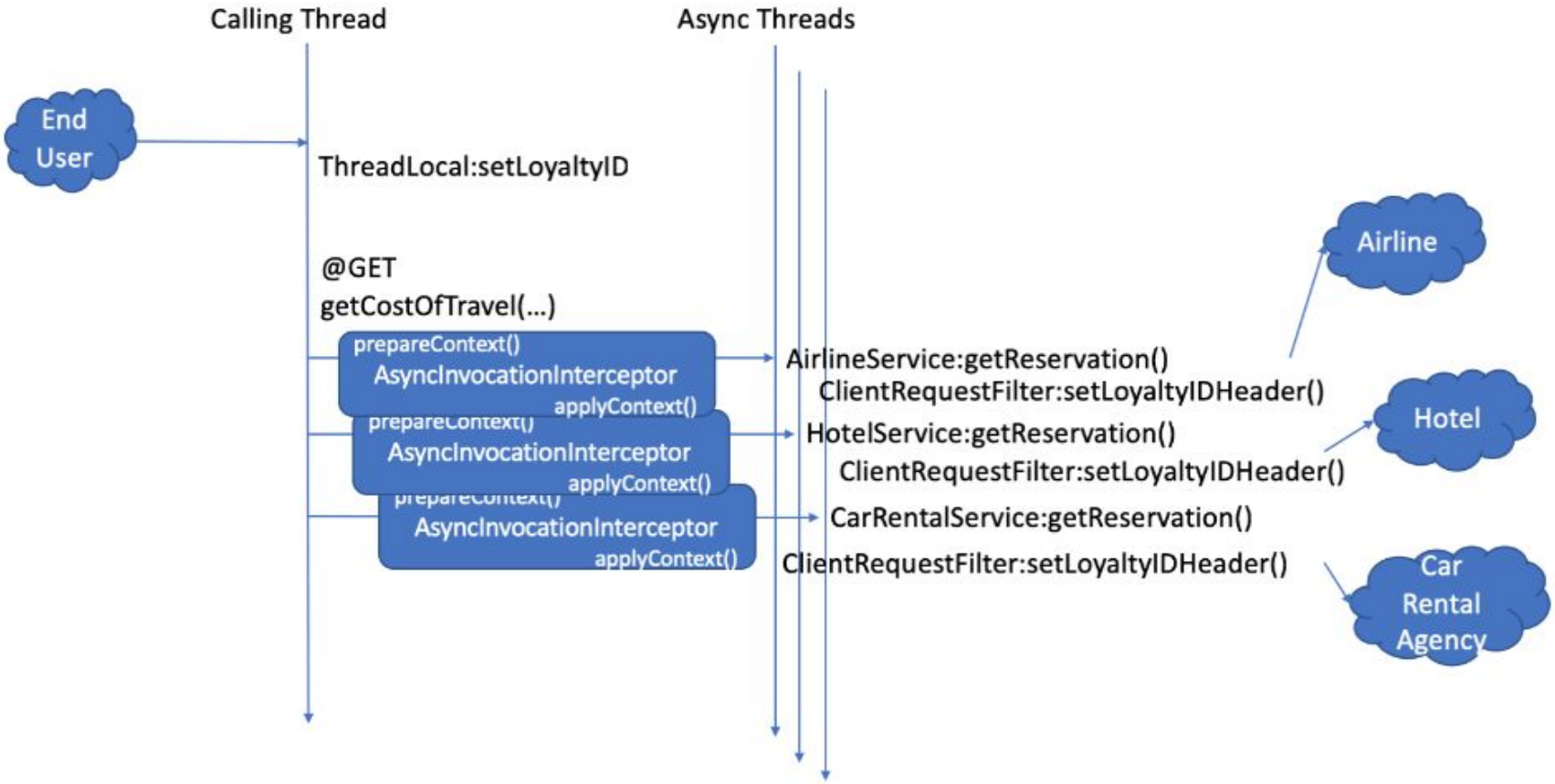
Ziarna sesyjne mogą implementować asynchroniczne metody, biznesowe metody gdzie kontrola jest zwracana klientowi przez ziarno przed przed inwokacją metody na ziarnie. Klienci mogą wtedy użyć API żeby otrzymać wynik. Asynchroniczne metody są używane dla długoterminowych operacji, zadań obciążających procesor lub do polepszenia czasu odpowiedzi aplikacji jeżeli wynik wywoływanej metody nie jest od razu potrzebny.

Zwrot kontroli przez ziarno pozwala klientowi na wykonywanie innych zadań podczas wykonywania się metody.

Asynchroniczny REST

Użycie serwisów asynchronicznych REST

- cenna w przypadkach, gdy dostępność usługi jest niska
- wymiana informacji niezależna od czasu
- nie zatrzymuje wszystkich innych operacji


Calling Thread
Async Threads
End User
ThreadLocal:setLoyaltyID
@GET
getCostOfTravel(...)
prepareContext()
AsyncInvocationInterceptor
applyContext()
AirlineService:getReservation()
ClientRequestFilter:setLoyaltyIDHeader()
Airline
HotelService:getReservation()
ClientRequestFilter:setLoyaltyIDHeader()
Hotel
CarRentalService:getReservation()
ClientRequestFilter:setLoyaltyIDHeader()
Car Rental Agency

Linki

- Artykuł z implementacją przykładu usług turystycznych:

  https://openliberty.io/blog/2019/01/24/async-rest-jaxrs-microprofile.html

- Kolejny przykład:

  https://blog.allegro.tech/2014/10/async-rest.html

Mikroserwisy

Jak działają mikroserwisy?

- Rozłożenie aplikacji na mniejsze komponenty
- Prostsze i inteligentniejsze przetwarzanie
- Zdecentralizowane zarządzanie bibliotekami i API
- Zasada pojedynczej odpowiedzialności
- Tolerancja błędów
- Systemy ewolucyjne
- Zdecentralizowane dane

Kiedy używać mikroserwisów?

Gdy system rośnie pod względem wymagań i funkcjonalności.

Kiedy musimy ponownie wykorzystać usługi.

Kiedy centralizacja interfejsów API zaczyna blokować ewolucję systemu.

Gdy istnieje potrzeba wdrożenia nowych funkcji, interfejsów API, bibliotek, bez przepisywania całego oprogramowania.

Zalety i wady mikroserwisów

Zalety

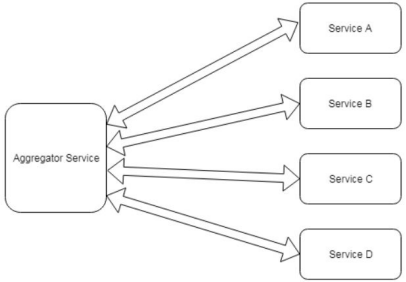
- Elastyczne skalowanie
- Niezależne wdrażanie
- Szybka eliminacja pojedynczej awarii
- Mniejsze ryzyko zepsucia aplikacji
- Minimalizacja przestojów po wydaniu nowej wersji
- Łatwiejsza aktualizacja
- Łatwiejsza praca z wieloma zespołami
- Łatwiejsze do zrozumienia
- Łatwiejsza rozbudowa
- Możliwość niezależnej zmiany
- Może mieć różne bazy danych
- Niezależny od technologii

Wady

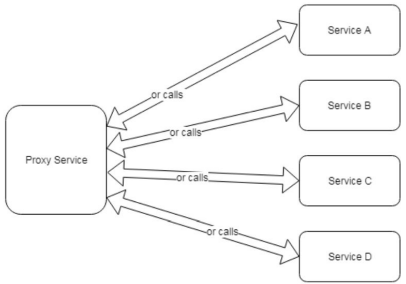
- Produktywność rozwoju
- Może być trudny do debugowania
- Komunikacja między mikroserwisami
- Postępowanie z błędami
- Aktualizacja udostępnionych danych
- Zautomatyzowane wdrażanie
- Trudność identyfikacji błędów
- Monitorowanie

Wzorce projektowe

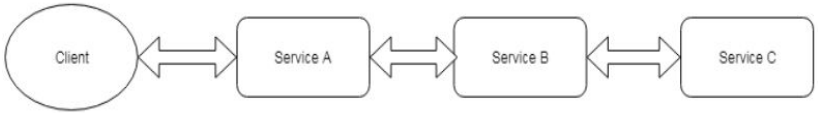
1) Aggregator pattern - wzorzec opiera się na stworzeniu kilku mikroserwisów, z którymi jeden - agregator - komunikowałby się i zbierał potrzebne odpowiedzi.



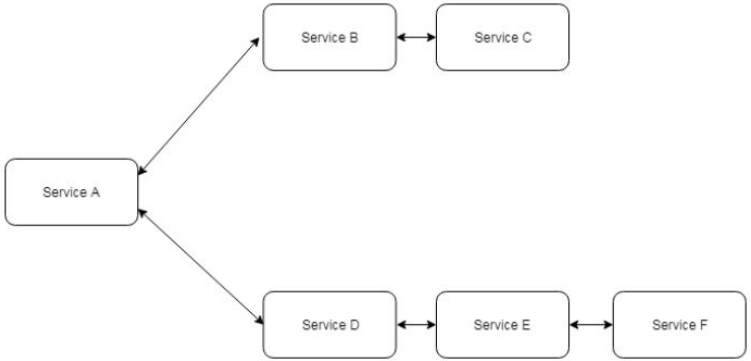
2) Proxy pattern - wariacja Aggregator pattern wyróżniająca się tym, że Proxy Service decyduje który mikroserwis zostanie wykorzystany w zależności od potrzeb



3) Chained pattern - opiera się na łańcuchu mikroserwisów, które komunikują się sekwencyjnie



4) Branch pattern - działa jako wariacja Aggregator Pattern, może wywoływać różne łańcuchy mikroserwisów równolegle



5) Asynchronous pattern - polega na asynchronicznej komunikacji mikroserwisów za pomocą brokera komunikacyjnego

Implementacja mikroserwisów

Wybór konkretnego wzorca w projekcie powinien być uzależniony od struktury i celu użycia aplikacji. Przykładowo: projektując aplikację, która zbierałaby potwierdzenie wpłaty w banku a następnie generowałaby potwierdzenie płatności dla klienta stworzylibyśmy dwa mikroserwisy - jeden czytający informacje o wpłacie i potwierdzający ją w banku, oraz drugi, wysyłający potrzebne informacje takie jak nowy kod kreskowy. Następnie zbudowalibyśmy trzeci, bardziej złożony mikroserwis używający powyższych. Byłoby to klasyczne użycie Aggregator Pattern.