Wzorce dla aplikacji działających w chmurze

(Cloud-Native Application)

Autorzy:
Sylwia Rusek
Kinga Wrona
Klaudia Kromołowska
Marcin Teofil Poręba
Mateusz Sitek
Mateusz Krasiński
Robert Kozik

Zamysł aplikacji w chmurze

- Skalowalność i elastyczność przy mniejszej złożoności
- Aplikacje są
  - pakowane w kontenery
  - zarządzane dynamicznie
  - zorientowane na mikrousługi
- Silnie rozproszony charakter

Główne cele aplikacji chmurowych

- Dostępność
- Zarządzanie danymi
- Komunikacja
- Zarządzanie i monitorowanie
- Wydajność i skalowalność
- Odporność na awarie
- Bezpieczeństwo

Fałszywe założenia przy projektowaniu aplikacji chmurowych

- Sieć jest niezawodna
- Opóźnienia nie istnieją
- Nieskończona przepustowość
- Sieć jest bezpieczna
- Topologia sieci się nie zmienia
- Istnieje jeden administrator
- Koszt przesyłu danych = 0
- Sieć jest jednorodna

Wzorce projektowe aplikacji działających w chmurze

Composite application (microservices)

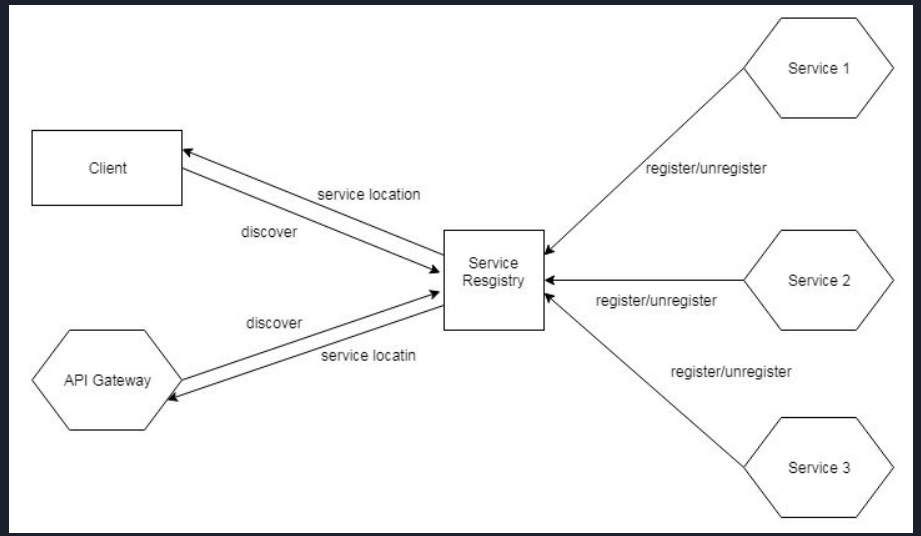
Zalety:

- pozwalają niezależnie rozwijać każdą usługę przez zespół, który dzięki temu koncentruje się tylko na funkcjonalności tej usługi
- umożliwia swobodny wybór technologii, usługi mogą być pisane w różnych językach
- wdrażanie jest szybsze, a integracja odbywa się w sposób bardziej automatyczny
- większa odporność na błędy

Wady:

- złożoność
- wzrost liczby mikrousług powoduje, że kontrola i zarządzanie tymi usługami staje się bardziej skomplikowane
- komunikacja pomiędzy różnymi usługami jest bardziej skomplikowana
- większe zużycie zasobów
- testy integracyjne wymagają więcej konfiguracji

Service registry



Źródło: Java EE 8 Design Patterns and Best Practices, Autor: Rhuan Rocha

Adresy IP kontenerów i maszyn wirtualnych są dynamiczne i mogą się często zmieniać. W związku z tym lokalizację usług rezydujących w tych kontenerach również podlegają zmianie. Instancje mikroserwisów są tworzone i usuwane na bieżąco.

Pojawia się pytanie, jak klient mikroserwisu może sobie z tym poradzić?

Service registry jest bazą danych zarejestrowanych usług. Po utworzeniu mikroserwisu automatycznie jest on zapisywany do tej bazy. Usunięty mikroserwis jest wyrejestrowany z tej bazy.

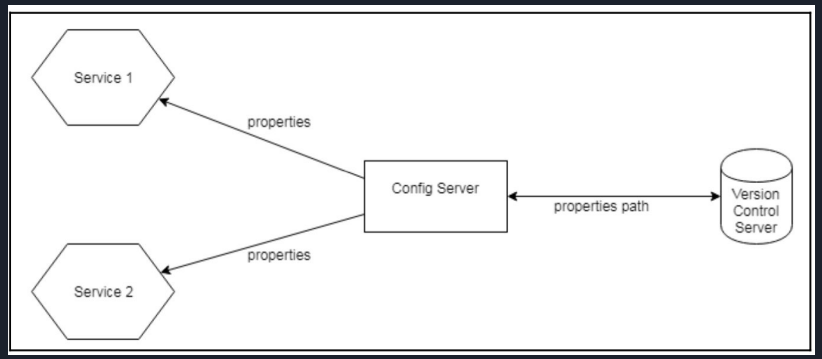
Klient mikroserwisu uzyskuje dostęp do rejestru usług, który odpowiada za to, czy dany mikroserwis jest dostępny, a także za dostarczenie klientowi jego lokalizacji.

Abstraction

Ten wzorzec mówi, że należy skoncentrować się na potrzebach klienta, a nie na istniejącej strukturze sprzętowej. W tym sensie zasoby obliczeniowe chmury są wykorzystywane na żądanie, co charakteryzuje elastyczną skalowalność. W ten sposób zasoby postrzegane są w sposób abstrakcyjny i zmieniane są zgodnie z potrzebami klienta.

Config server

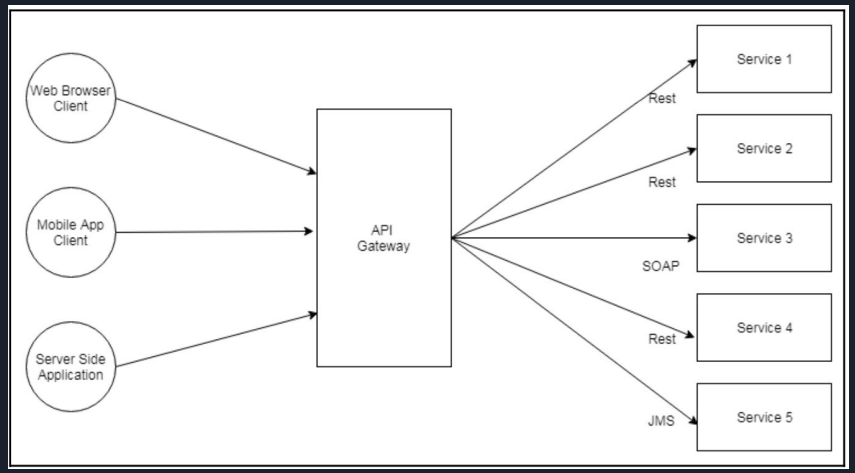
Serwer konfiguracji jest odpowiedzialny za dostarczanie konfiguracji dla każdej zarejestrowanej mikrousługi, a następnie przechowywane jej w pamięci. Za każdym razem gdy którakolwiek z właściwości zostanie zmieniona, zmiana ta zostanie odzwierciedlona w mikroserwisie bez konieczności ponownego kompilowania lub ponownego uruchamiania usługi. Dane konfiguracyjne pobierane są ze ścieżki określonej w systemie kontroli wersji.



Źródło: Java EE 8 Design Patterns and Best Practices, Autor: Rhuan Rocha

Api Gateway

API Gateway służy jako “frontend” dla klientów aplikacji w chmurze. Za pomocą jednego zapytania pozwala wykonać bardziej skomplikowane operacje wymagające komunikacji z różnymi aplikacjami. W zależności od typu klienta, funkcjonalność danego systemu może czasami działać inaczej, a aplikacja może odpowiadać różnymi informacjami

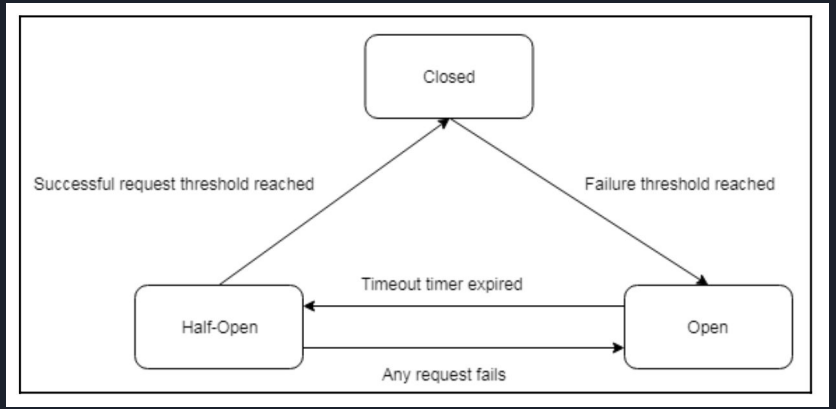


Źródło: Java EE 8 Design Patterns and Best Practices, Autor: Rhuan Rocha

Circuit-Breaker

Mechanizm działa jako serwer proxy, kierując żądanie do serwera lub natychmiast zwracając wyjątek. Działanie wzorca jest podobne do obwodu elektrycznego. Serwer ma trzy różne stany, jak pokazano na diagramie:

- Closed - obwód działa prawidłowo, przekazując requesty do serwera. W przypadku timeoutu licznik niepowodzeń zostaje zwiększony. Po przekroczeniu limitu przez licznik aplikacja przechodzi w stan half-open
- Open - na każde zapytanie zwracany jest wyjątek
- Half-open - ograniczona liczba żądań do serwisu. W przypadku sukcesu licznik zeruje się, a obwód przechodzi w stan closed

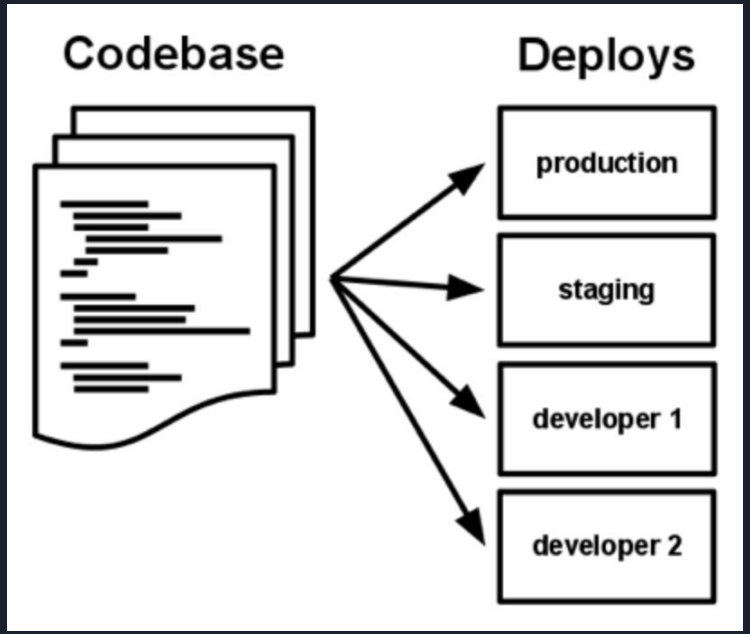


Źródło: Java EE 8 Design Patterns and Best Practices, Autor: Rhuan Rocha

Twelve Factor

Code Base

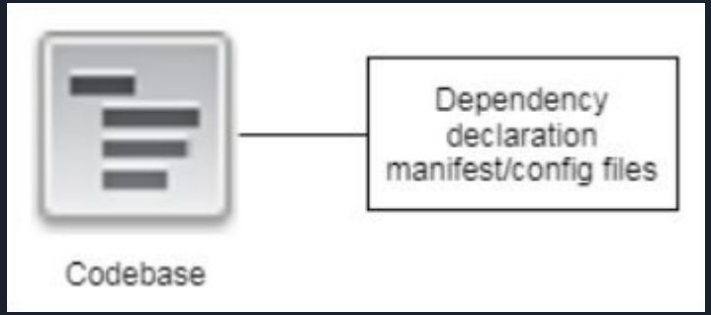
- Jedno repozytorium → wiele wdrożeń,
- Występuje tylko jedno źródło kodu umożliwiające budowanie różnych wersji aplikacji,
- Repozytorium jest zarządzane przez system kontroli wersji (np. GIT),
- Umożliwia to proste budowanie aplikacji w wersji deweloperskiej czy produkcyjnej,



Źródło: https://12factor.net/codebase

Dependencies

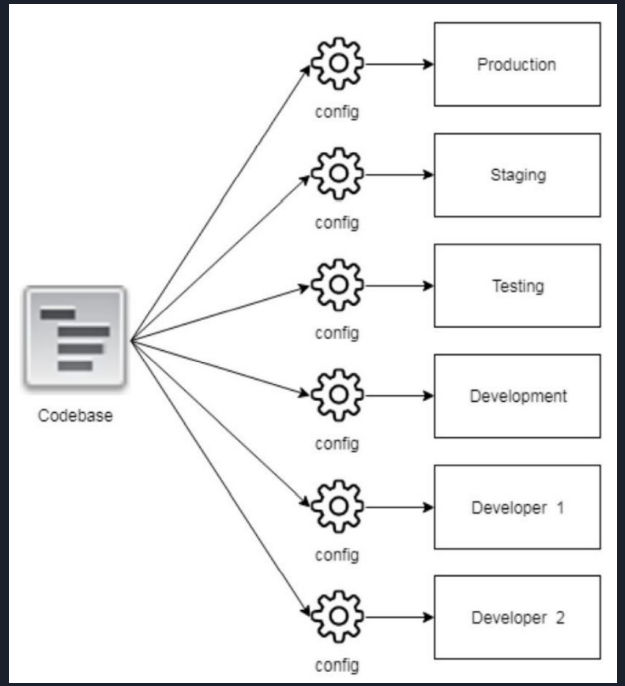
- Aplikacja 12factor nigdy nie jest zależna od bibliotek zainstalowanych dla całego systemu
- w praktyce wiąże się to z korzystaniem z narzędzi do zarządzania projektem np. Maven, Gradle, SBT
- Uproszczenie początkowej konfiguracji aplikacji dla developera



Źródło: Java EE 8 Design Patterns and Best Practices, Autor: Rhuan Rocha

Config

- Jedyny element, który może się różnić pomiędzy wdrożeniami aplikacji
- Konfiguracja jest ściśle oddzielona od kodu aplikacji
- Aplikacja 12factor przechowuje konfigurację w zmiennych środowiskowych



Źródło: Java EE 8 Design Patterns and Best Practices, Autor: Rhuan Rocha

Backing services

- Są to zewnętrzne usługi (bazy danych, serwisy komunikatów, repozytorium plików, email)
- Każda usługa jest traktowana przez twelve-factor jako zasób
- Do każdego zasobu uzyskuje się dostęp poprzez dane podane w konfiguracji - adres URL lub lokalizację
- Zmiany lokalizacji usługi nie wpływają na kod

Build, release, run

Wyróżniamy trzy etapy przekształcenia bazy kodu do danego środowiska.

- BUILD - kompilowanie i generowanie wykonywalnego projektu
- RELEASE - stosowanie konfiguracji w pakiecie, rezultatem jest wykonywalna aplikacja z konfiguracją dla danego środowiska
- RUN - inicjalizowanie aplikacji w środowisku

Rozdzielnie tych kroków pomaga w utrzymaniu systemu i poprawianiu automatyzacji.
Korzystamy z narzędzi do ciągłej integracji, takich jak Maven czy Jenkins.

Processes

- Procesy i składniki aplikacji powinny być bezstanowe i nie powinny przechowywać informacji
- Jeżeli istnieje potrzeba przechowywania danych czy stanów, powinno ię wykorzystać bazę danych
- W ten sposób minimalizujemy ryzyko problemów wynikających z przetwarzania zapytania przez inny proces

Port-binding

- całkowicie samowystarczalny, nie wymaga użycia zewnętrznego serwera, takiego jak Tomcat czy Apache do wyeksportowania jako usługi
- powiązanie portu z usługą HTTP, co oznacza, że komunikuje się za pośrednictwem adresu URL
- każdy serwis może działać jako serwis zastępczy

Concurrency

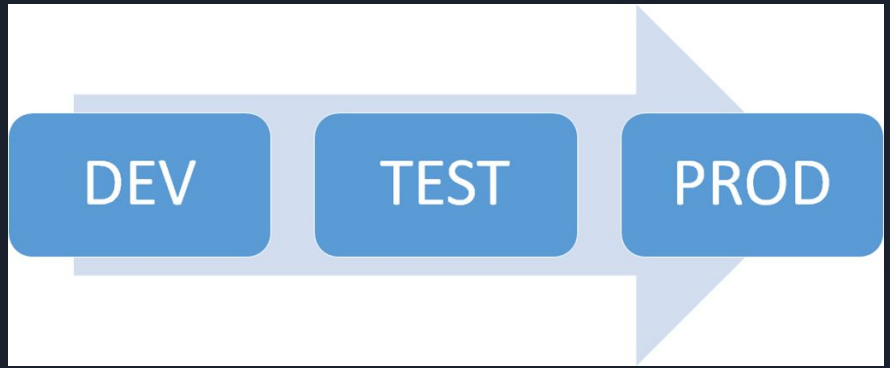
- skalowanie aplikacji w modelu równoległym (UNIX-owe deamony)
- wykonywanie długich zadań w tle
- aplikacja jest replikowana, tworząc kopie procesu zamiast wykonywania nowej instancji

Disposability

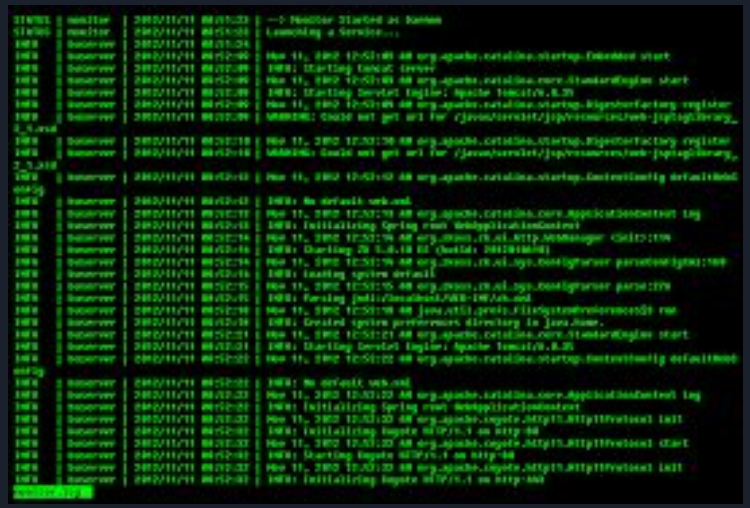
- procesy powinny zaczynać się i kończyć tak szybko jak to jest możliwe z jak najmniejszym wpływem
- procesy są jednorazowe, co oznacza, że mogą być zainicjowane i zatrzymane w dowolnym momencie, zatrzymanie procesu nie powinno mieć żadnego wpływu na działanie całej aplikacji, powinno zwolnić zasoby, w razie konieczności powinien być zapisany stan procesu

Dev/prod parity

Aplikacja napisana metodologią dwunastostopniową powinna mieć jak najbardziej zbliżone do siebie środowiska developerskie, testowe i produkcyjne. Dzięki takiemu podejściu unikniemy błędów podczas przenoszenia zmian pomiędzy środowiskami.



Logs



- Logi powinne być traktowane jako strumień zdarzeń
- Zapobiega możliwości utraty logów
- Umożliwienie przekierowania logów

Admin processes

Zadania polegające na utrzymaniu środowiska powinny być zautomatyzowane. Zadania takie jak migracja danych, inicjalizacja danych lub czyszczenie pamięci podręcznej powinny być wykonywane przy użyciu skryptów. Pozwala to na uniknięcie błędów związanych z różnorodnością środowisk.

Dziękujemy za uwagę!