**Microservices with Spring Cloud**

(na podstawie tutka z udemy -> <https://www.udemy.com/microservices-with-spring-boot-and-spring-cloud/>)

Ogólnie tutek spoko i w sumie tyle. Aaa no warto wspomnieć o tym, że jak coś było o niczym to zwyczajnie pomijałem w opracowaniu. Warto również wspomnieć, że każdy obrazek, który się pojawi w opracowaniu jest oczywiście nie mój 😊 Wszelkie uwagi po przeczytaniu wskazane ++

To jak było ogólnie to teraz skrobnę coś w miarę szczegółowo. Znaczy nie będzie to turbo szczegółowe bo wiem jak (nie)dobrze piszę, ale postaram się przybliżyć wam co fajnego oferuje nam Spring Cloud. No to zaczynając, jest nam to po to aby ułatwić pracę w architekturze mikroserwisów.

# Spring Cloud Config Server

Port Domyślny: 8888

Jest to serwer, który ma za zadanie wspierać zarządzanie konfiguracją mikroserwisów. Bez znajomości Spring Cloud Config zazwyczaj trzyma się nasze ustawienia razem z kodem w plikach typu properties. No i tu jest problem bo może zdarzyć się tak, że ktoś przez nieuwagę np. ja 😊 zacommituje do publicznego użytku coś co nie powinno być współdzielone np. hasła. Ponadto gdy nie wersjonujemy naszych properties to tracimy możliwość przeglądania historii zmian w tych plikach i może dojść do sytuacji gdzie załadujemy do naszej apki inne ustawienia (apka działa, po prostu ustawienia są inne niż ostatnio) i będziemy zachodzić w głowę czy i dlaczego, ktoś je zmienił, może faktycznie była taka potrzeba, ale po co itp. To może zająć sporo czasu. Kolejny minus properties per mikroserwis to brak możliwości ich współdzielenia. Bez serwera z konfiguracjami musimy kopiować zawartość plików, które można uwspólniać. Do tego dochodzi niebezpieczeństwo przechowywania haseł plain textem. Trochę to minusów ma ale z pomocą przychodzi nasz Config Serwer.

Obraz zawierający tekst

Opis wygenerowany przy bardzo wysokim poziomie pewności

No to tutaj wyżej widzimy jak to pięknie wszystko działa jeśli korzystamy z Config Serwera. To teraz opisze dlaczego to tak ładnie działa, przy okazji jak to zaimplementować i jeszcze pewnie kilka innych rzeczy na raz, żeby za prosto nie dało się tego wszystkiego skumać.

W dużych apkach spotkamy się bowiem z setkami mikroserwisów z których każdy z nich posiada wiele środowisk. Co więcej część z tych środowisk może mieć po kilka instancji tego samego mikroserwisu. Ręczna konfiguracja indywidualnych przypadków jest wtedy bardzo bardzo trudna. Noi tu nasz serwerek mówi – OK, ty mi włóż gdzieś (np. do gita) całą konfigurację twoich mikroserwisów a ja już zadbam o zarządzanie nimi. I tak np. jeden mikroserwis powie: „Hej, chcę być konfigurowany za pomocą ustawień produkcyjnych”, a inny „Hola, hola! Jeszcze coś zepsuję, yyy… mi wystarczy konfiguracja testowa”. W każdym z tych przypadków nasz Config Serwer zadba o to by każdy mikroserwis został właściwie skonfigurowany.

Według konwencji portem domyślnym dla tego serwera jest port nr 8888. Tworzymy sobie prosty mikroserwis: „limits-service”. Po ustawieniu będzie on zawierał swoje propertisy, które następnie zostaną nadpisane konfiguracją ze SpringCloudConfigServer-a. Plan jest taki:

Obraz zawierający obiekt

Opis wygenerowany przy bardzo wysokim poziomie pewności

Co to za CurrencyCalculation i CurrencyExchange będzie potem.

LimitsService propertiesFile na początku zawiera:

spring.application.name=limits-service

i defaultowy port = 8080

Tworzymy klasę, która będzie zawierała pola minimum i maximum reprezentujące przykładowe wartości obiektu LimitConfiguration:

Obraz zawierający zrzut ekranu

Opis wygenerowany przy bardzo wysokim poziomie pewności

Oraz przykładowy kontroler zwracający nam obiekt LimitConfiguration.

Obraz zawierający zrzut ekranu

Opis wygenerowany przy bardzo wysokim poziomie pewności

Na razie bez żadnych Rocket Science 😊

Feature: SpringBoot pozwala nam na odczyt danych konfiguracyjnych bezpośrednio z pliku properties i do tego właśnie dążymy, aby móc konfigurować mikroserwisy na podstawie danych z Config Serwera, które znajdują się właśnie w **plikach properties.**

**Dodajmy sobie dwie dane konfiguracyjne do pliku properties w mikroserwisie limits-service:**

limits-service.minimum=99

limits-service.maximum=9999

Stwórzmy nową klasę, gdzie za pomocą prostej adnotacji zadzieje się magia 😉

Obraz zawierający zrzut ekranu

Opis wygenerowany przy bardzo wysokim poziomie pewności

Adnotacja @ConfigurationProperties pozwala nam na wskazanie, który plik konfiguracyjny powinien być użyty. W nawiasie podajemy prefix z pomocą którego tworzyliśmy dane konfiguracyjne w pliku properties.

Wstrzyknijmy teraz naszą magiczną klasę do kontrolera obsługującego naszą konfigurację. Uzyskamy to za pomocą @Autowired, a następnie odwołując się do pól z tej klasy.

Obraz zawierający zrzut ekranu

Opis wygenerowany przy bardzo wysokim poziomie pewności

Hurra! Nasz serwis jest w pełni gotowy do czytania danych konfiguracyjnych bezpośrednio ze swojego pliku properties.

Teraz możemy postawić nasz Config Serwer na nogi 😊

Więc, tworzymy nowy mikroserwis, który potem dostanie skromny upgrade. Na razie zainicjujmy nowy projekt o nazwie spring-cloud-config-server.

PS. Dodajmy do mavena zależność, która potem nam się przyda:

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-config-server</artifactId>

</dependency>

I ustawmy mu nazwę oraz port:

spring.application.name=spring-cloud-config-server

server.port=8888

**TERAZ INSTALUJEMY GITA**

<https://git-scm.com/>

- Tworzymy lokalne repozytorium

- Tworzymy w nim pliki tekstowe, które posłużą jako konfiguracja (dajemy im rozszerzenie properties)

- Wycinamy z .properties serwisu limits-service dane konfiguracyjne (limits-service.minimum oraz limits-service.maximum) i przeklejamy je do nowo utworzonego w repozytorium pliku limits-service.properties (nazwa pliku – limits-service oznaczać będzie defaultowy plik konfiguracyjny). Ustawiamy je np. na minimum=8 oraz maximum=888 – tak było w tutku 😊

- Commitujemy zmiany w naszym lokalnym repozytorium

Teraz wypada wskazać naszemu serwerowi miejsce, w którym znajduje się cała nasza konfiguracja czyt. Ścieżkę do naszego lokalnego repo.

Możemy to uzyskać dodając do naszego pliku konfiguracyjnego

spring.cloud.config.server.git.uri=**file://**/ścieżka do lokalnego repo z plikami konfiguracyjnymi

U mnie:

**spring.cloud.config.server.git.uri=** <file:///C:/Users/181395/Desktop/microservices/spring-cloud-config-server/git-localconfig-repo>

* **file://** <- to składniowo jest ważne celem wskazania dobrej ścieżki!!!

Następnie, żeby zadziała się magia Spring Clouda należy uruchomić nasz mikroserwis w trybie Config Serwera. Możemy to uzyskać dodając do naszej klasy bootstrapującej serwer adnotację @EnableConfigServer.

Żeby sprawdzić poprawność tej części należy strzelić do localhost:8888/limits-service/default co oznacza localhost:{port na którym odpalony jest serwer}/{nazwa usługi}/default

**Teraz to na co wszyscy czekali czyli multikonfiguracja.**

Więc, stwórzmy sobie dwa dodatkowe pliki konfiguracyjne – niech będzie jak w tutku –

limits-service-dev.properties oraz limits-service-qa.properties i nadajmy im odpowiednio minimum=1 lub 2 oraz maximum = 111 lub 222

**ZACOMMITUJMY NOWE PLIKI .PROPERTIES**

Dla testu możemy się odwołać: **localhost:8888/limits-service/qa** , **localhost:8888/limits-service/dev**

Cel jest taki, aby mikroserwis limits-service był konfigurowany przez Spring Cloud Config Server z poziomu gita. Należy więc teraz zmienić nazwę pliku .properties dla limits-service na bootstrap.properties (bo tak 😊 ) oraz dodać do niego adres serwera poprzez dodanie:

**spring.cloud.config.uri=http://localhost:8888** (u nas jest to 8888)

Żeby ładnie móc konfigurować serwisy w zależności od np. środowiska należy w jakiś cudowny sposób wskazać im czym mają być konfigurowane. Uzyskać to możemy poprzez wyspecyfikowanie w propertiesach profilu dla jakiego mają zostać zaczytane ustawienia.

Otrzymamy to poprzez dodanie:

**spring.profiles.active=dev** lub **spring.profiles.active=qa**

Na tej podstawie określamy z którego pliku dane konfiguracyjne Spring Cloud Config Server powinien pobrać dla naszego limits-service.

**\*Żeby wprowadzać zmiany w plikach konfiguracyjnych znajdujących się na repo i widzieć je należy te zmiany commitować 😊**

JUPI!!! W TAKI OTO CUDOWNY SPOSÓB UDAŁO SIĘ NAM WSPÓLNIE STWORZYĆ MIKROSERWIS KONFIGUROWANY DZIĘKI SPRING CLOUD CONFIG SERVEROWI Z POZIOMU GITA. ALE PO CO NAM TO, TAK WŁAŚCIWIE?

Obraz zawierający obiekt

Opis wygenerowany przy bardzo wysokim poziomie pewności

W tym momencie wystarczy, że stworzymy sobie np. CurrencyExchangeService i wskażemy mu tak jak w limitsService:

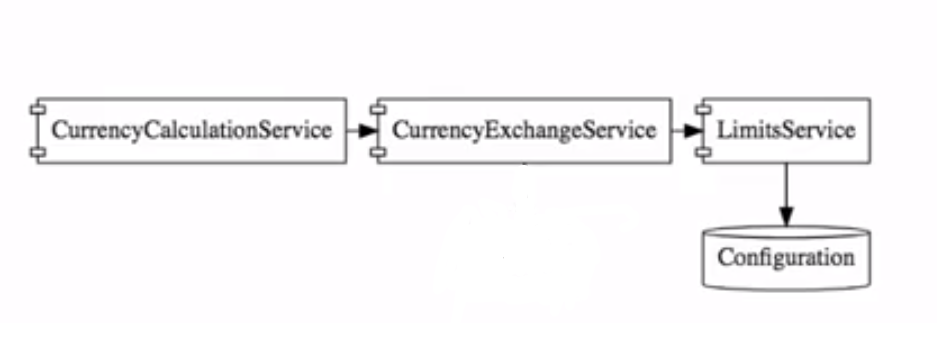
Obraz zawierający zrzut ekranu

Opis wygenerowany przy wysokim poziomie pewności

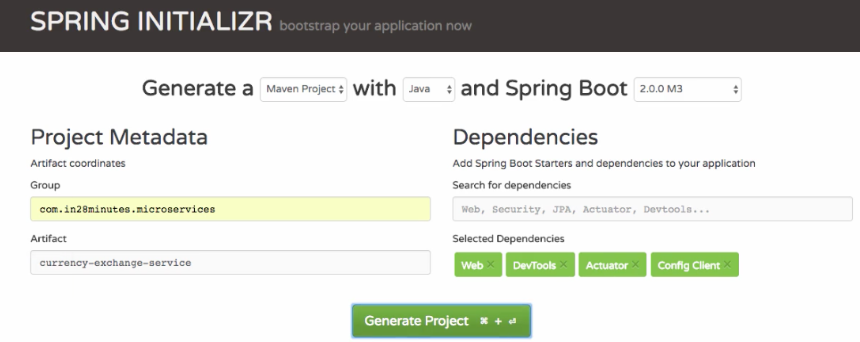
Nazwę, uri config serwera oraz profil z jakiego ma zczytywać sobie dane konfiguracyjne et voila – podłączyliśmy sobie kolejny mikroserwis do naszej zewnętrznej konfiguracji.

# Dynamiczna zmiana portów – a przynajmniej coś w tym stylu :/

Plan jest taki abyśmy stworzyli dwa mikroserwisy rozmawiające ze sobą i przy okazji konfigurowane wg mnie pseudo dynamicznie. Struktura jak poniżej:



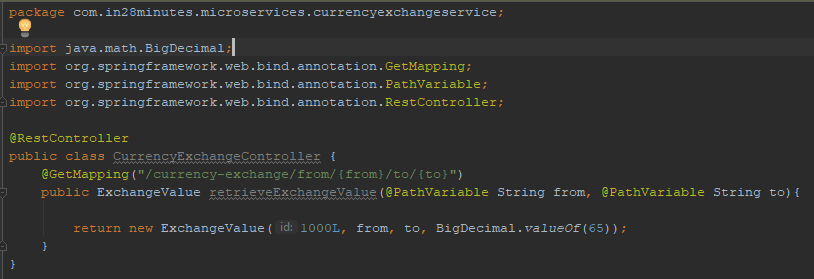
No to zacznijmy od stworzenia pierwszego z nich. Dajmy mu nazwę currencyExchangeService bo tak też podpatrzyłem w tutku. Po prostu tworzymy nowy mikroserwis czyt. Projekt spring bootowy i wsjo. Zależności poniżej. BTW strona nazywa się ***start.spring.io***



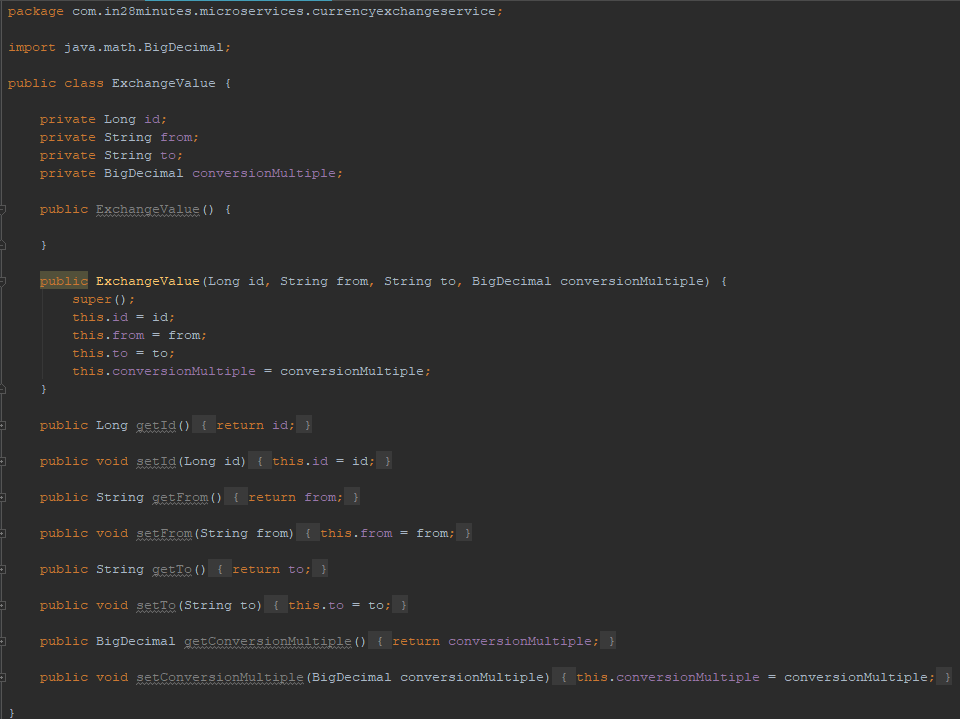
Nadajmy mu nazwę oraz port (w pliku .properties):



„Zahardkodujmy” ten mikroserwis.



I stwórzmy dodatkową klasę reprezentującą zwracany obiekt:



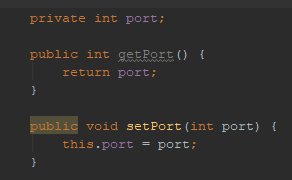
Rewelacji brak bo też zrobiliśmy tyle kodu w dwie minuty ==> 

Noi po co żeśmy to zrobili? A no po to żeby dalej zaprezentować w jaki sposób tworzyć multi-instancje konkretnego mikroserwisu. Bowiem może zdarzyć się tak, że będziemy chcieli niejako

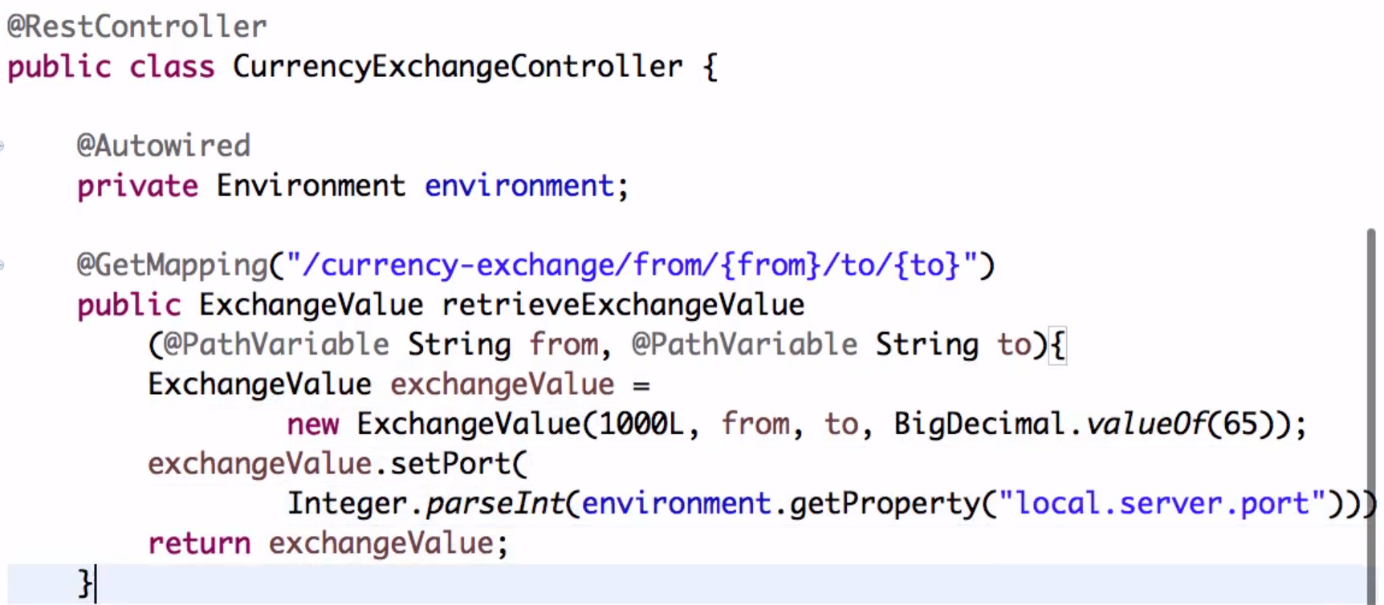
„w locie” *<- to chyba coś innego znaczy xD*

uruchamiać kolejną instancję danego mikroserwisu ALE NA INNYM PORCIE.

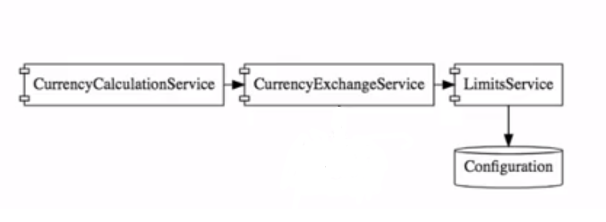
Dodajmy pole **port** w naszej klasie reprezentującej zwracany obiekt – ExchangeValue

wraz z getterem i seterem: 

OK, rozbudujmy nasz RestController poprzez dodanie kilku moim zdaniem fajnych opcji:



Jak widzimy zostało dodane kilka rzeczy i dzięki temu widzimy w zwracanym obiekcie numer portu na jakim działa mikroserwis. Pewnie że mogliśmy to zrobić bez dependency injection, ale coś jednak dzięki temu osiągniemy. W przyszłości tzn na następnej stronie ;) zadbamy o to aby odpalać kilka instancji exchangeService więc fajnie by było widzieć jaki (stojący na jakim porcie) mikroserwis zwrócił odpowiedź serwisowi CurrencyCalculationService przypominając, że docelowo w tym punkcie dążymy do tego aby było kilka instancji ExchangeService gadających z CurrencyCalculationService.

Dlaczego to osiągniemy? Ponieważ wstrzykując za pomocą @Autowired interfejs Environment możemy bezpośrednio „dobierać się” do zmiennych konfiguracyjnych obecnie odpalonej instancji.

getProperty() – żadne Rocket Science – po prostu wyciąga „własność” – wolę określenie „propertisa”

parseInt – metoda wyżej zwraca String-a więc zwyczajnie go parsujemy na Integera i wsjo.

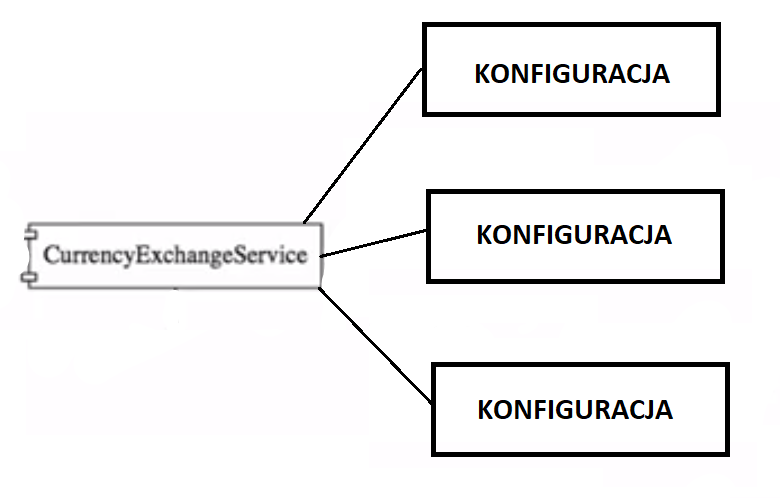
Czas na clue programu -> zmianę portu kolejnej instancji:

Możemy ręcznie wejść w .properties i zmienić np. w ExchangeService 8000 na np. 8069 ale nic tym nie osiągniemy, bo zmienimy port klikniemy run i cały czas będziemy mieć jedną instancję mikroserwisu, która działa po prostu na innym porcie. Chodzi nam o multi-instancyjność(!? – nie podkreśla więc chyba jest takie słowo).

Osiągniemy to poprzez podanie na tacy maszynie wirtualnej portu na jakim ma być odpalona aplikacja czyt. Mikroserwis. Gdy podamy wprost do JVM to port zadeklarowany w .properties zostanie nadpisany i aplikacja uruchomi się na wskazanym przez nas, a nie we właściwościach porcie.

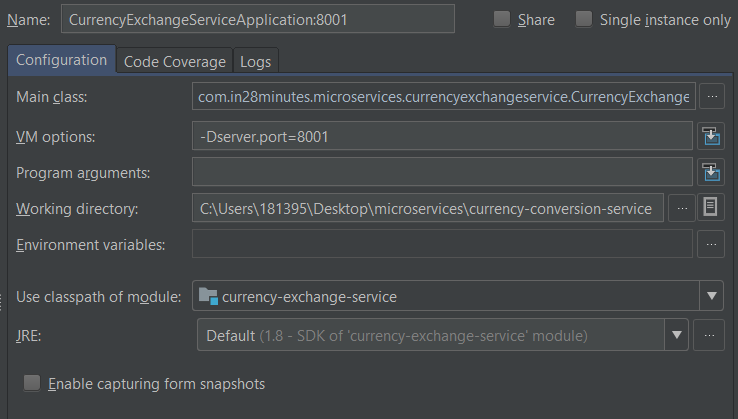
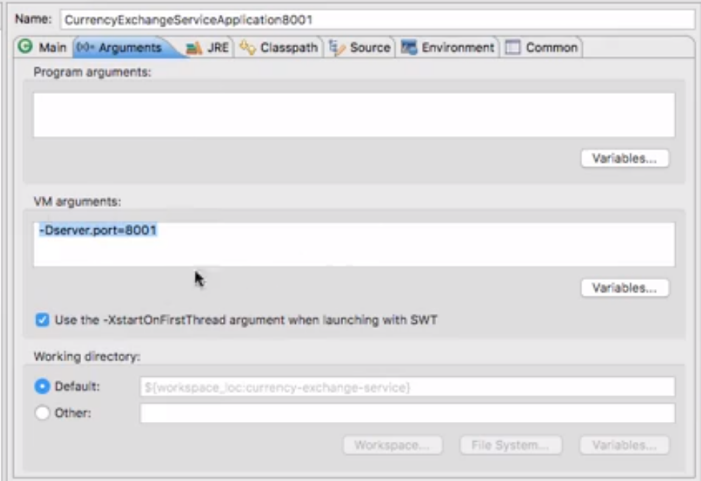
Jak to osiągnąć?

Należy przygotować nową konfigurację tego samego mikroserwisu tzn. mieć dwie konfigurację i jeden mikroserwis.



Więc – proste jak budowa cepa – jeśli odpalimy trzy konfiguracje na raz to odpalimy de facto trzy mikroserwisy czyli to o co nam chodzi -> trzy różne instancje tego samego mikroserwisu.

Jak to zrobić?



Eclipse IntelliJ

Wystarczy podać: **-Dserver.port={numer\_portu}**

Mamy już możliwość tworzenia wielu instancji tego samego mikroserwisu. Teraz stworzymy koleżkę dla naszych instancji aby miały z kim gadać. I tak tworzymy nowy projekt o nazwie CurrencyCalculationService, żebym nie musiał nowych obrazków szukać ☺ , pom mavena jak wyżej.

Ustawmy mu: 

oraz zaimplementujmy klasę reprezentującą obiekt CurrencyConversion wraz z getterami i seterami, które uciąłem żeby nie zająć zbyt dużo miejsca ;)



-----------

-> gettery + setery

Ponadto utwórzmy kontroler, który będzie ten zasób konsumował.



Nasz RestController ma za zadanie zwrócić obiekt reprezentujący klasę CurrencyConversionBean, ale po uprzednim pogadaniu sobie z mikroserwisem currency-exchange, którego instancja w naszym przykładzie śmiga sobie na porcie numer 8000. W przykładzie powyżej została użyta klasa RestTemplate, która ma za zadanie zwyczajnie uprościć z poziomu Javy wykonywanie żądań http czyli w naszym przypadku GET-a. Warto dodać, że metoda getForEntity() jako pierwszy parametr przyjmuje url spod jakiego pobierze odpowiedź, drugi to typ tej odpowiedzi czyli w naszym przypadku będzie to CurrencyConversionBean, a trzeci to dodanie uprzednio zadeklarowanego interfejsu Map przechowującego w naszym przypadku zmienne przekazane w żądaniu.

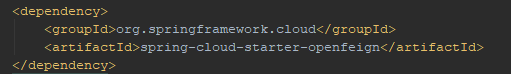
# Load Balancing

## Feign

Ziomek w tutku opisuje Feign jako „deklaratywny klient HTTP” i ja też nic z tego nie czaje :D

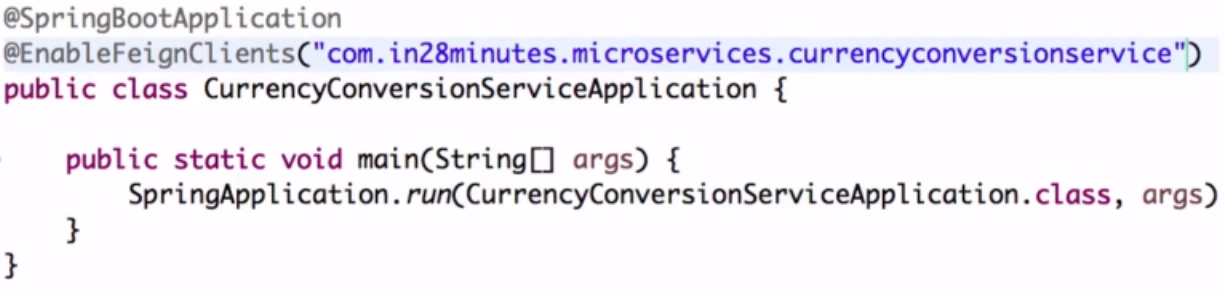
Po analizie wyczaiłem, że wszystko sprowadza się do tego, że użycie Feign-a pomaga nam oddzielić naszą logikę od naszego kodu biznesowego. „Pomaga nam zminimalizować ilość pisanego kodu, który za cel ma pobranie podobnie ustrukturyzowanych danych” – w myśl zasady, że im mniej kodu tym lepiej. Z pomocą Feign deklarujemy co nasz klient ma robić, a nie w jaki sposób ma to robić.

Użycie Feign jest bardzo intuicyjne. Do serwisu, który ma używać naszego deklaratywnego klienta (chciałem to napisać ☺ ) wystarczy dodać zależność:



lub **feign** zamiast **openfeign** jeśli używamy wersji niższej niż 2.0.0.M4

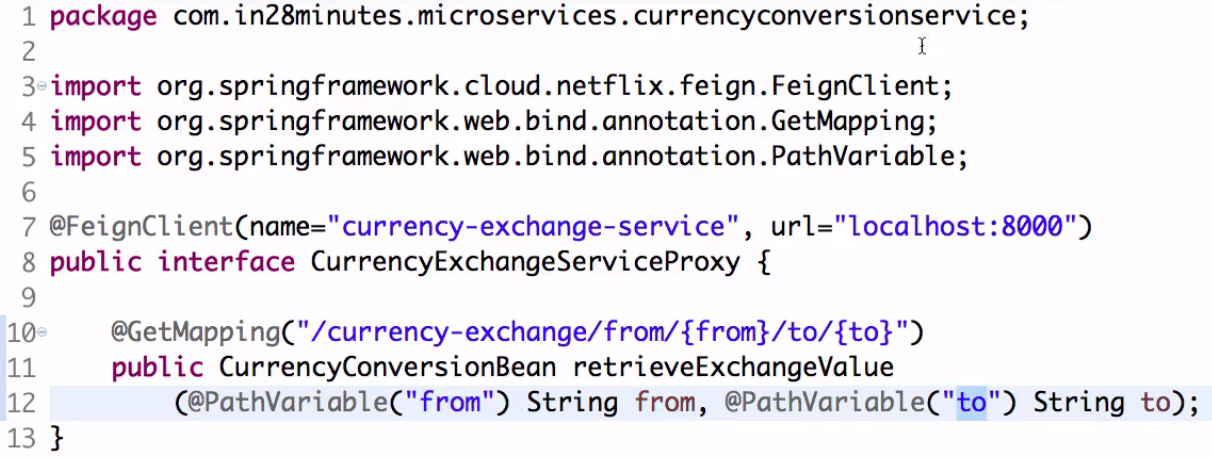
Używając SpringBoota wystarczy nam jedna adnotacja, żeby uruchomić autokonfigurację Feigna:



Jako argument podajemy pakiet do skanowania celem szukania Feign klientów.

Co musimy zrobić teraz, to utworzyć Feign proxy by móc komunikować się z innymi mikroserwisami.

Stwórzmy więc interfejs, który będzie za to odpowiadał. Jako atrybut **name** ważne jest aby podać nazwę mikroserwisu, z którym będziemy chcieli się komunikować, a w miejsce, url lokalizację tego serwisu. Teraz zadeklarujmy metodę, która będzie łączyła się z odpowiednim zasobem i dzięki której będziemy mogli faktycznie zeswatać dwa serwisy ze sobą i umożliwić im komunikację:

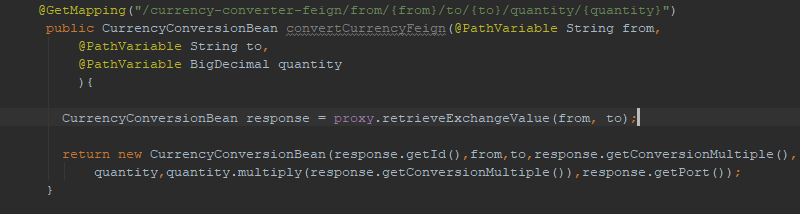


Dodatkowo ważne jest, że jeśli używamy Feign to musimy bezpośrednio wyspecyfikować ścieżkę do zmiennej podawanej jako parametr tzn. (”from”) i (”to”) jest niezbędne, żeby uniknąć błędu podczas kompilacji. Po utworzeniu takiego interfejsu z pełną świadomością możemy go reużywać i wstrzykiwać jak zwykłego beana. Użycie takiego klienta jest już bardzo proste i sprowadza się do wstrzyknięcia beana klienta oraz wywołania odpowiedniej metody, co upraszcza nam znacznie kod.

Wstrzyknijmy więc w nasz kontroler powstałego Feign klienta:



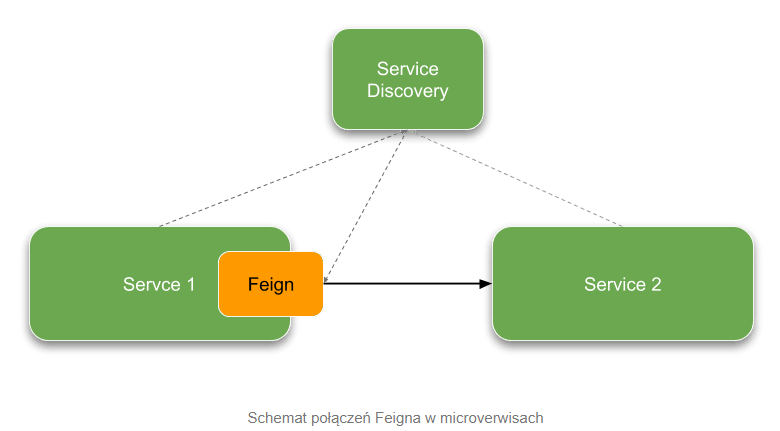
I przygotujmy pod niego usługę:



Widzimy tu jak z pomocą Feign możemy skrócić ilość kodu generowanego wcześniej z pomocą RestTemplate. Kod robi dokładnie to samo, endpointy różnią się tylko dostawioną wzmianką **–feign-**

a mimo tego nasz kod wygląda schludniej i jest go mniej.

Kończąc wspomnę i rozwinę w dalszych punktach, że Feign od początku był przeznaczony do pracy w środowisku mikroserwisów – tak przynajmniej przeczytałem, a więc potrafi korzystać z service discovery np. Eureka, której specyfikację przybliżę w dalszej części. Feign używa Ribbona (następny punkt) do łączenia z discovery i rozkładania ruchu na poszczególne instancje aplikacji i jest to ten tak zwany Load Balancing. Schemat poniżej:

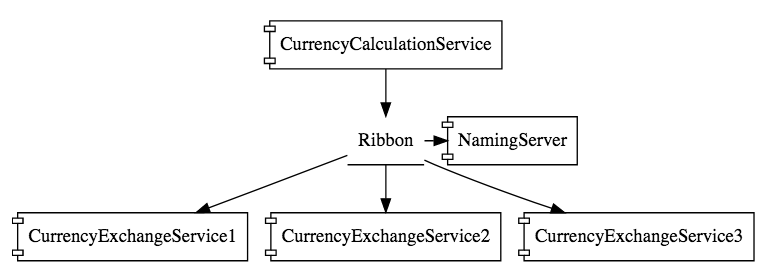


Instancje poszczególnych serwisów rejestrują się w serwis discovery, po czym Feign może pobrać listę zarejestrowanych serwisów. Zwykle serwisy rejestrują się pod nazwą np. Service1, Service2. Dodatkowo co zostanie opisane przy Eurece trzeba pamiętać o uruchomieniu autokonfiguracji klienta discovery w aplikacji bootstrapującej poprzez @EnableDiscoveryClient i możemy się cieszyć działającym połączeniem pomiędzy naszymi microserwisami.

OK, wracając ostatni raz do tematu Feign. Po co nam w ogóle ten deklaratywny klient i w czym on jest lepszy? A więc po zdefiniowaniu odpowiedniej konfiguracji jedyne co musimy wywołać w naszym kodzie to metoda, którą zdefiniowaliśmy i parametry jej wywołania. KROPKA. Nic więcej nie zaśmieca naszego kodu i nic więcej nie musimy ręcznie tworzyć.

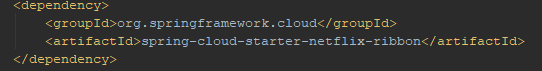
## Ribbon

Zacznijmy od obrazka ☺

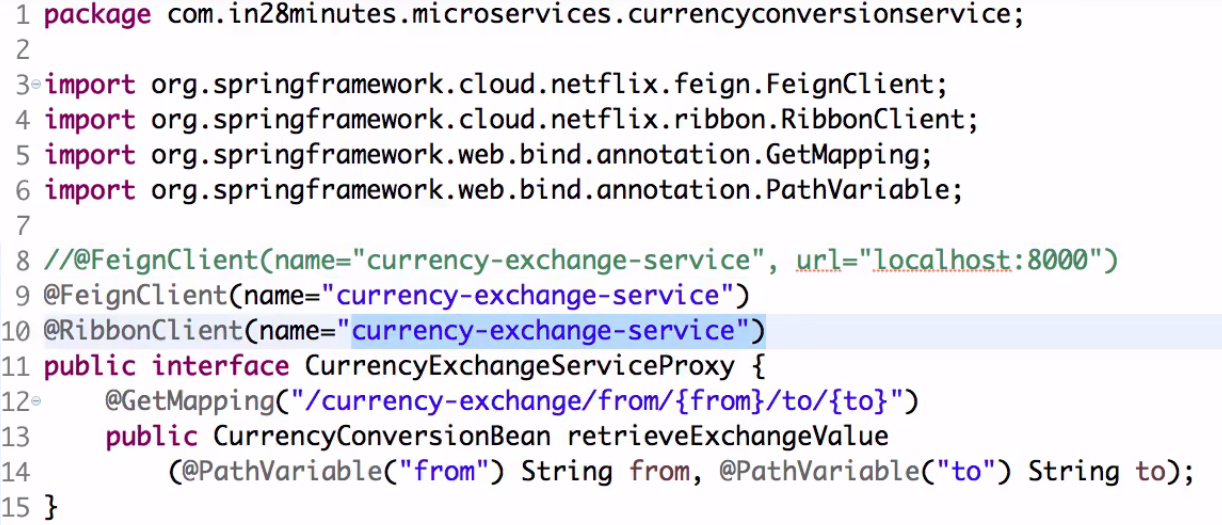


Obecnie potrafimy dynamicznie skonfigurować gadanie ze sobą dwóch serwisów. No i fajnie bo możemy to robić dynamicznie uruchamiając kilka instancji jednego serwisu. Załózmy hipotetyczną sytuację, że mamy uruchomione trzy instancje CurrencyExchangeService (1, 2 i 3) Ale pojawia się problem w postaci takiej, że nasz CurrencyCalculationService może gadać albo z uruchomionym 1 albo 2 albo 3 hmm… Przydałoby się aby mógł rozmawiać z nimi wszystkimi ☹

I tu wkracza Ribbon, który pomaga nam „rozdystrybuować” połączenia pomiędzy różnymi instancjami CurrentExchangeService. Podłączamy go poprzez dodanie mavenowej zależności



do serwisu, który ma mieć podłączenie do innych tzn. CurrencyExchangeService(1, 2, 3) będą konsumować zasoby z CurrencyCalculationService więc w przykładzie z obrazka to on właśnie ma być dostarczycielem zasobów. Uruchamiamy go, albo inaczej – deklarujemy klientów dla naszego CurrencyConversionService poprzez dodanie adnotacji (obrazek poniżej):



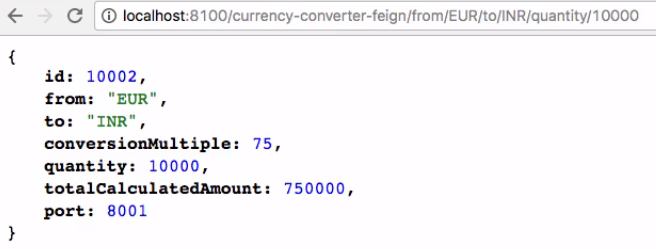
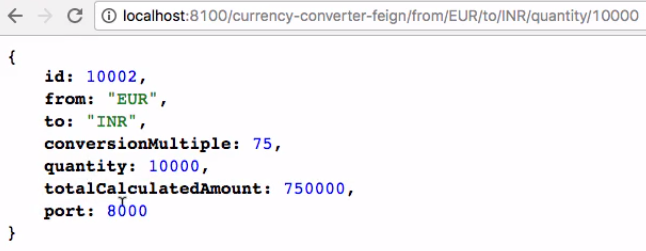
oraz skoro mamy wiele instancji tego samego mikroserwisu, to powinniśmy w jakimś miejscu mu (Ribbonowi xD) wskazać gdzie się znajdują te mikroserwisy.

Uzyskamy to poprzez dodanie w pliku konfiguracyjnym naszego CurrencyConversionService właściwości

**currency-exchange-service.ribbon.listOfServers=http://localhost:8000,http://localhost:8001**

co wydaje mi się w pełni zrozumiałe i nie wymaga nadmiernego komentarza. No może poza tym, że jedna instancja ExchangeService stoi na 8000 a druga na 8001 numerze portu.

Po uruchomieniu dwóch instancji możemy w fajny sposób zaobserwować jak Ribbon dystrybuuje żądania po dwóch instancjach co widzimy na dwóch obrazkach poniżej. Raz gada z jednym a raz z drugim: **\* Inny port ;)**



Dalej, pozmieniamy trochę, bo jak się dowiedziałem to w praktyce ilość aktywnych instancji zwracana jest przez Service Discovery np. Eureka. Następnie wybór która usługa zwrócona przez Service Discovery zostanie wywołana leży po stronie klienta.

# Naming Server – Eureka

Eureka Service Discovery umożliwia efektywne zarządzanie usługami bez konieczności deklarowania adresów kolejnych instancji na sztywno w kodzie źródłowym aplikacji. Zdecydowanie dobrą praktyką jest korzystanie z mechanizmu Service Discovery. Jeśli jednak jest uzasadnienie biznesowe do używania predefiniowanej listy serwerów to jak najbardziej jest możliwe używanie projektu Spring Cloud Ribbon bez konieczności stawiania serwera Eureki.

# API Gateway – Zuul

# Rozproszone śledzenie

## Zipkin

## Sleuth

# Odporność na uszkodzenia – Hystrix

Mój styl pisania pozostawia wiele do życzenia, ale uważam, że jeśli ktoś nie potrafi wytłumaczyć drugiej osobie tego co wie tak żeby zrozumiała to w rzeczywistości wcale tak dobrze tego nie zna ;) wobec tego jeśli pojawiły się jakieś pytania podczas czytania proszę śmiało o zadawanie ich. W sumie tyle.