**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. W sumie tyle.

# 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:

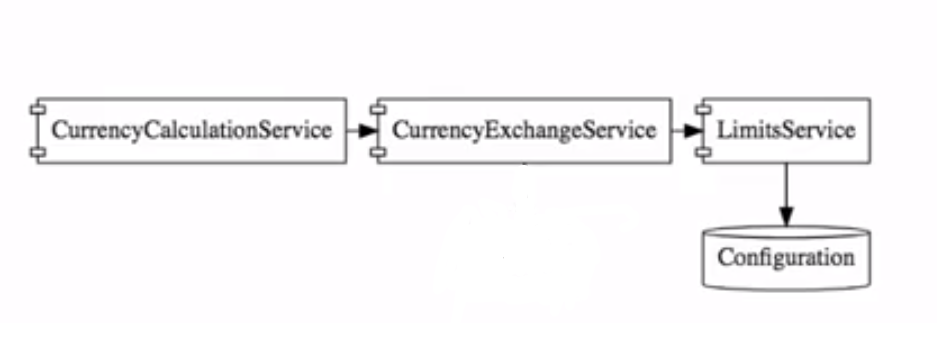
Obraz zawierający zrzut ekranu

Opis wygenerowany przy wysokim poziomie pewności

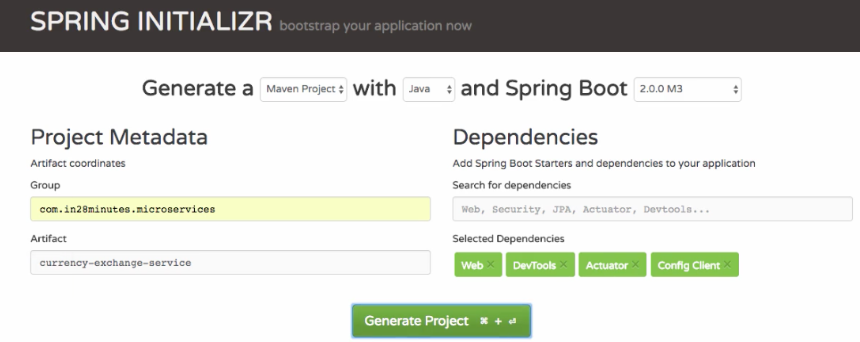
Nazwę, uri config serwera oraz profil z jakiego ma zczytywać sobie dane konfiguracyjne et voila – podłączyliśmy sobie kolejny mikroserwis do naszej zewnętrznej konfiguracji.

# 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:



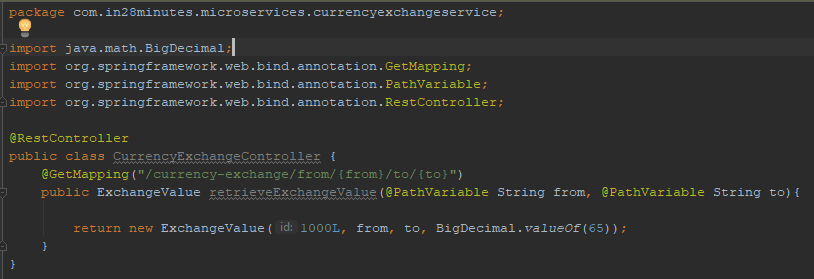
No to zacznijmy od stworzenia pierwszego z nich. Dajmy mu nazwę currencyExchangeService bo tak też podpatrzyłem w tutku. Po prostu tworzymy nowy mikroserwis czyt. Projekt spring bootowy i wsjo. Zależności poniżej. BTW strona nazywa się ***start.spring.io***



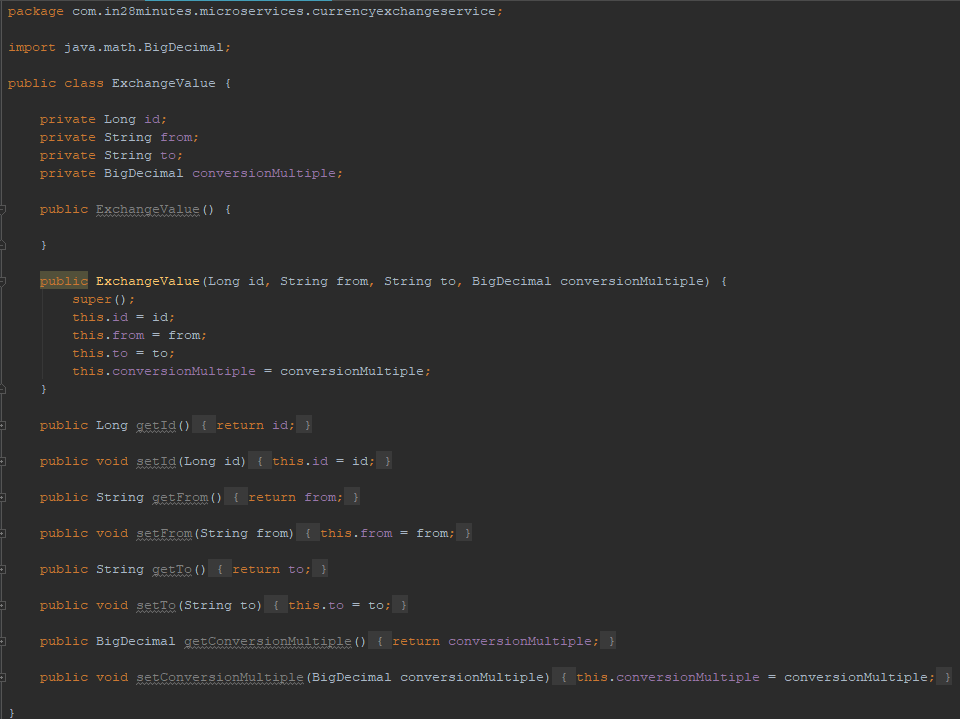
Nadajmy mu nazwę oraz port (w pliku .properties):



„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

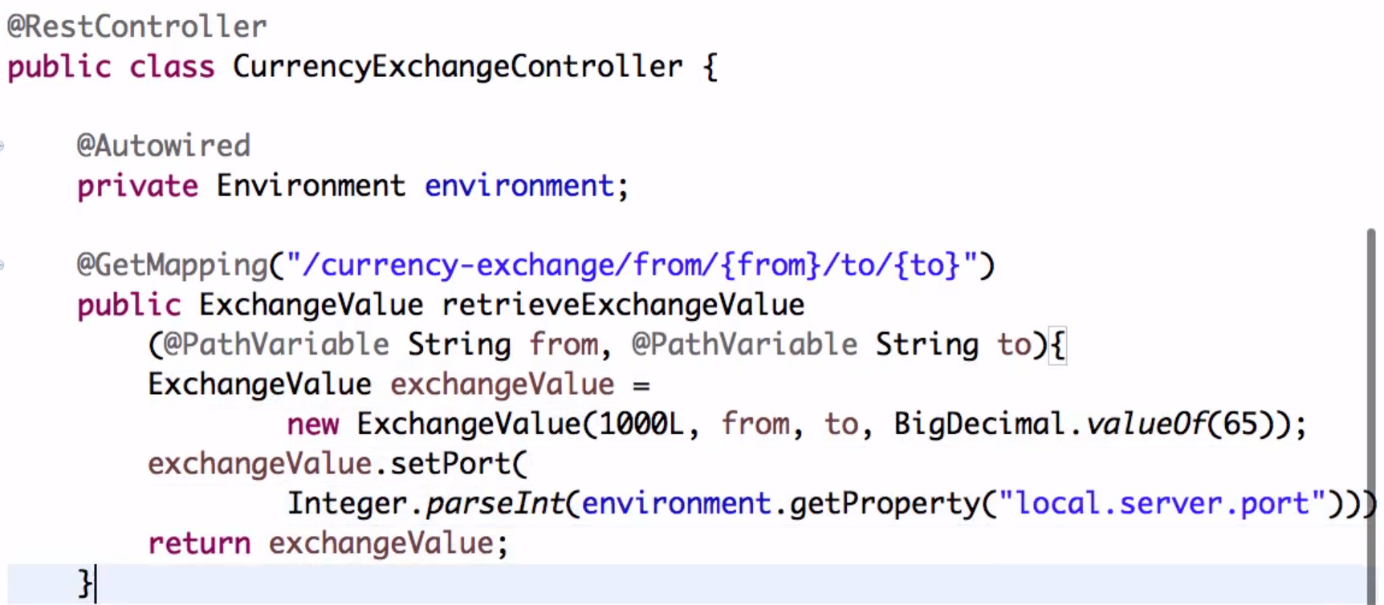
„w locie” *<- to chyba coś innego znaczy xD*

uruchamiać kolejną instancję danego mikroserwisu ALE NA INNYM PORCIE.

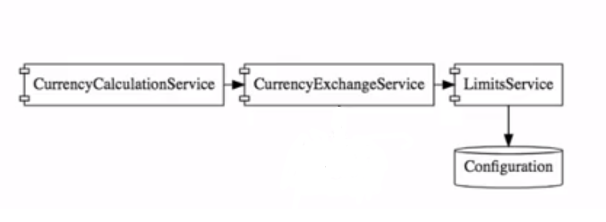
Dodajmy pole **port** w naszej klasie reprezentującej zwracany obiekt – ExchangeValue

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.

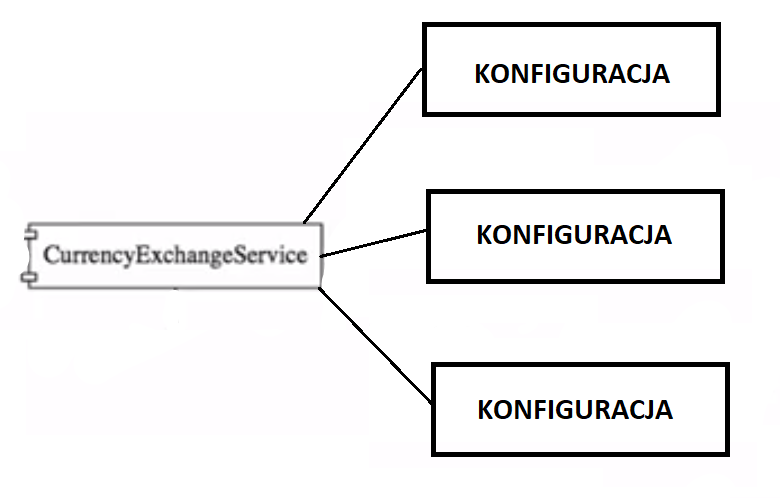
Czas na clue programu -> zmianę portu kolejnej instancji:

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.

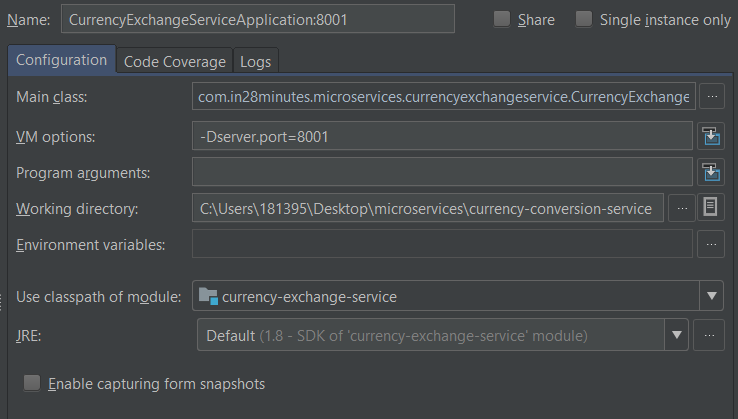
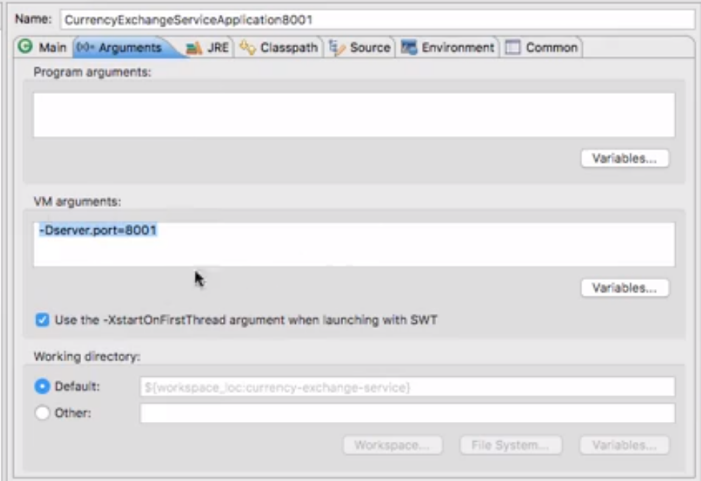
Jak to osiągnąć?

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.

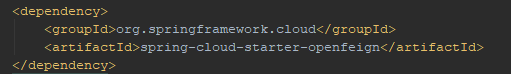
# Load Balancing

## Feign

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

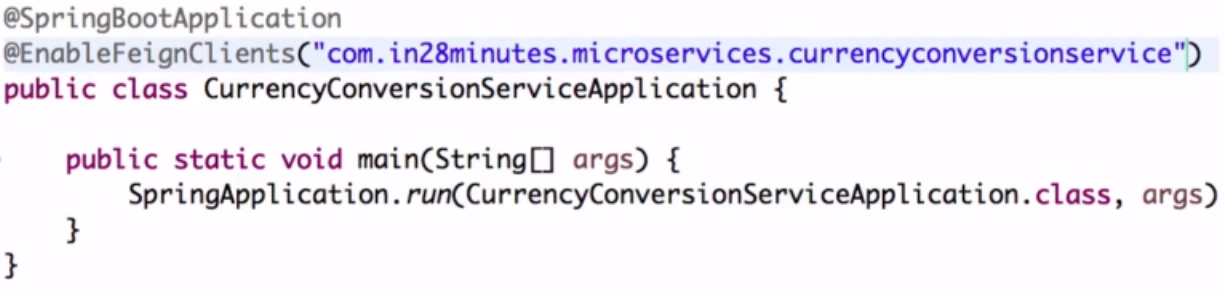
Po analizie wyczaiłem, że wszystko sprowadza się do tego, że użycie Feign-a pomaga nam oddzielić naszą logikę od naszego kodu biznesowego. „Pomaga nam zminimalizować ilość pisanego kodu, który za cel ma pobranie podobnie ustrukturyzowanych danych” – w myśl zasady, że im mniej kodu tym lepiej. Z pomocą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

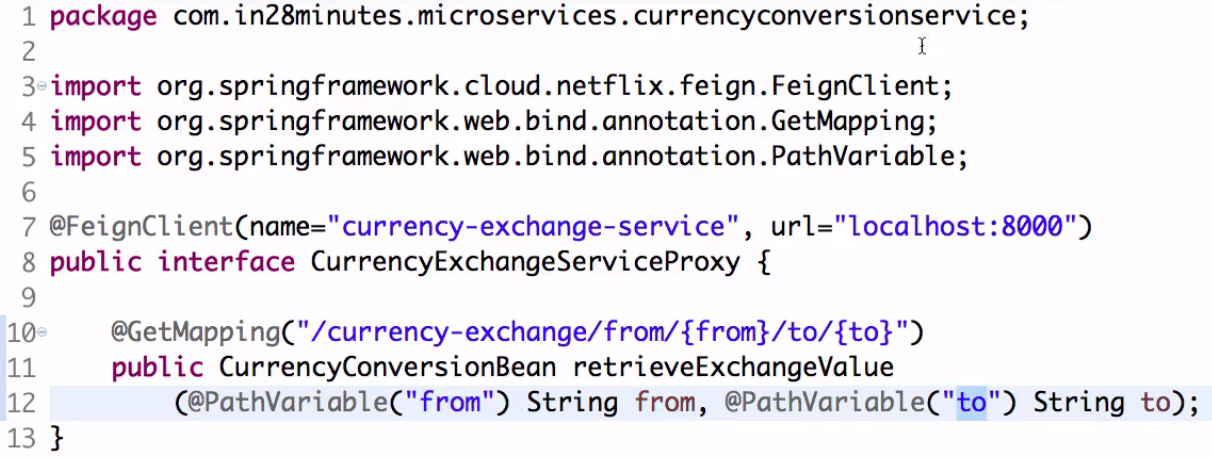
Używając SpringBoota wystarczy nam jedna adnotacja, żeby uruchomić autokonfigurację Feigna:



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:

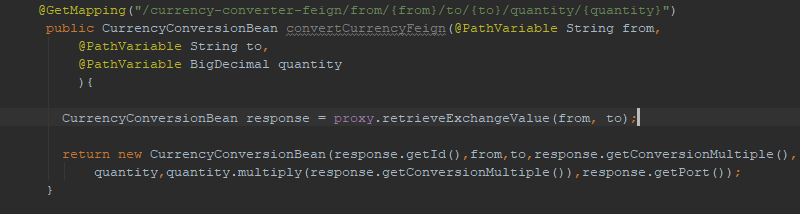


Dodatkowo ważne jest, że jeśli używamy Feign to musimy bezpośrednio wyspecyfikować ścieżkę do zmiennej podawanej jako parametr tzn. (”from”) i (”to”) jest niezbędne, żeby uniknąć błędu podczas kompilacji. Po utworzeniu takiego interfejsu z pełną świadomością możemy go reużywać i wstrzykiwać jak zwykłego 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:



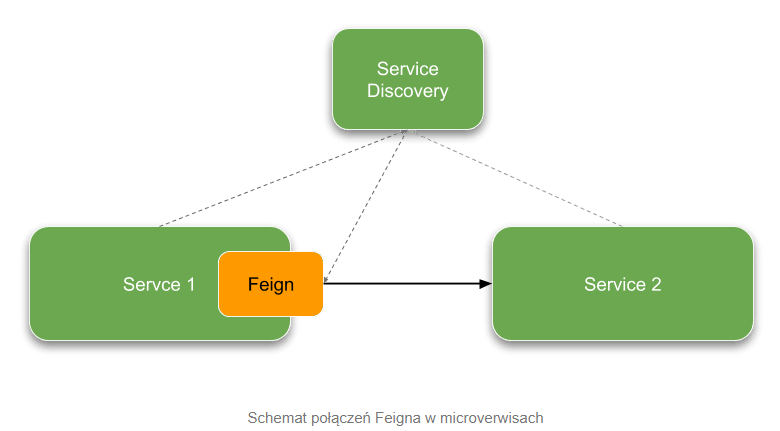
I przygotujmy pod niego usługę:



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:

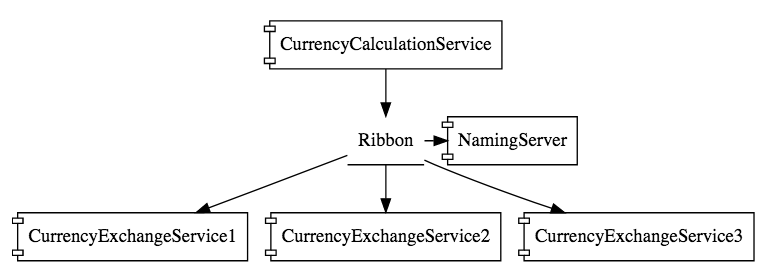


Instancje poszczególnych serwisów rejestrują się w serwis discovery, po czym Feign może pobrać listę zarejestrowanych serwisów. Zwykle serwisy rejestrują się pod nazwą np. Service1, Service2. Dodatkowo co zostanie opisane przy Eurece trzeba pamiętać o uruchomieniu autokonfiguracji klienta discovery w 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 ☺

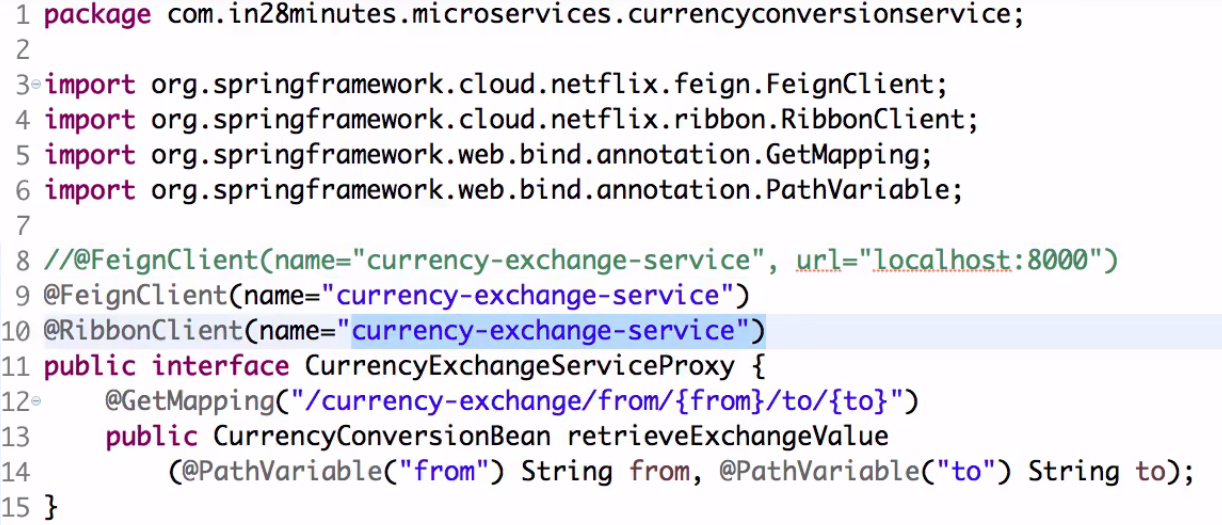


Obecnie potrafimy dynamicznie skonfigurować gadanie ze sobą dwóch serwisów. No i fajnie bo możemy to robić dynamicznie uruchamiając kilka instancji jednego serwisu. Załózmy hipotetyczną sytuację, że mamy uruchomione trzy instancje CurrencyExchangeService (1, 2 i 3) Ale pojawia się problem w postaci takiej, 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):



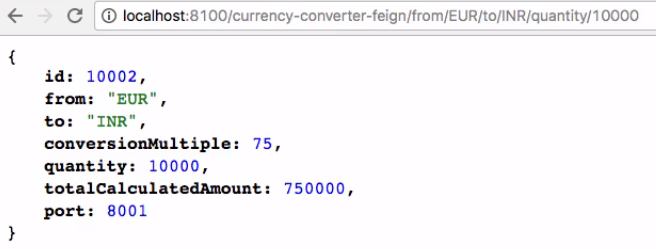
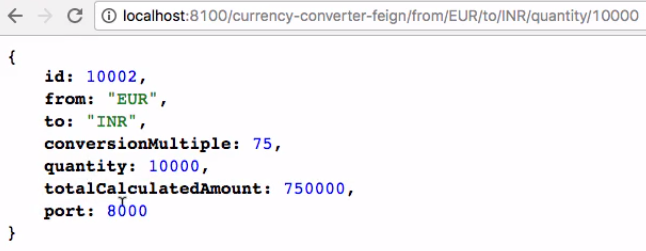
oraz skoro mamy wiele instancji tego samego mikroserwisu, to powinniśmy w jakimś miejscu mu (Ribbonowi xD) wskazać gdzie się znajdują te mikroserwisy.

Uzyskamy to poprzez dodanie w pliku konfiguracyjnym naszego CurrencyConversionService właściwości

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

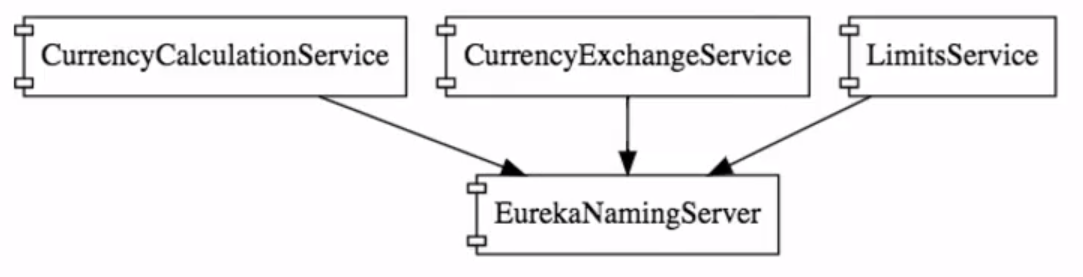
co wydaje mi się w pełni zrozumiałe i nie wymaga nadmiernego komentarza. No może poza tym, że jedna instancja ExchangeService stoi na 8000 a druga na 8001 numerze portu.

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 ;)**



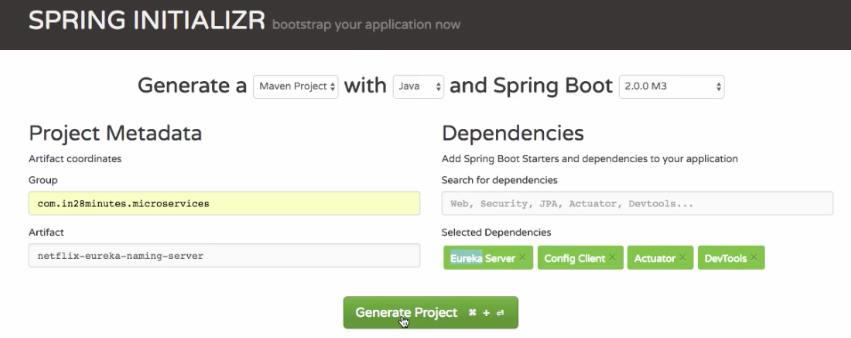
Dalej, pozmieniamy trochę, bo jak się dowiedziałem to w praktyce ilość aktywnych instancji zwracana jest przez Service Discovery np. Eureka. Następnie wybór która usługa zwrócona przez Service Discovery zostanie wywołana leży po stronie klienta.

# Naming Server – Eureka



Eureka Service Discovery umożliwia efektywne zarządzanie naszymi mikroserwisami 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. U nas się to przyda więc jadę zgodnie z tutkiem.

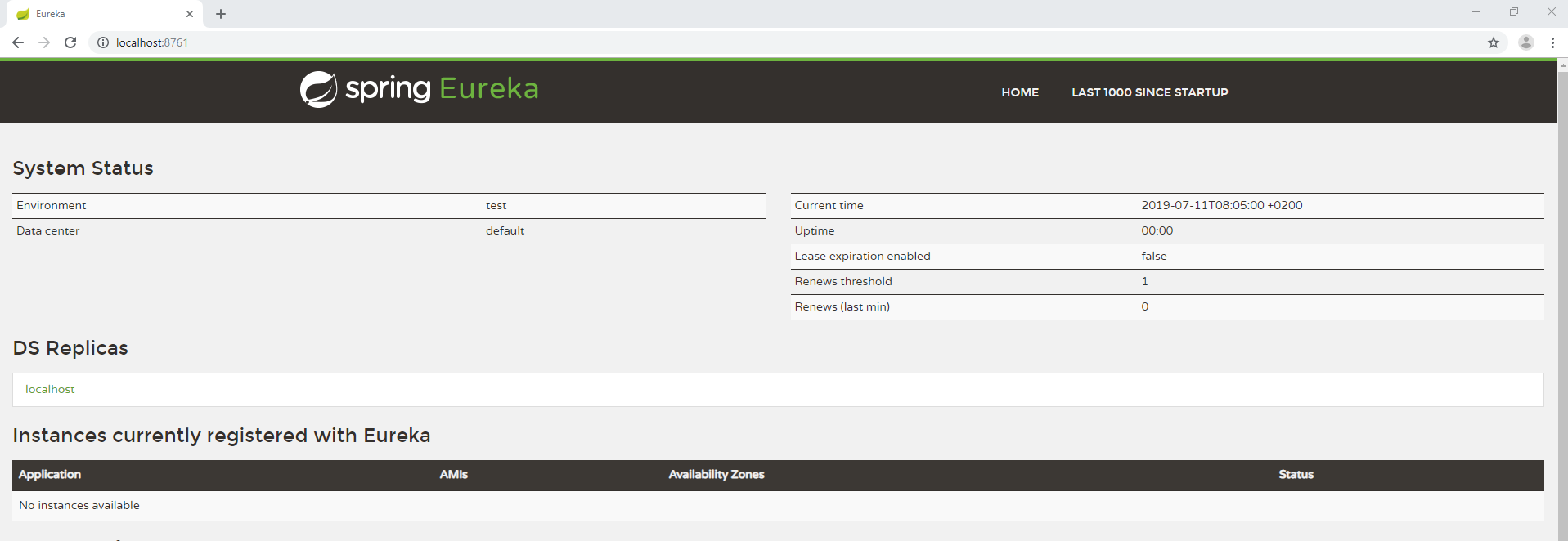
Postawmy więc nasz Naming Server. -> **start.spring.io** i wg obrazka poniżej:



Nasz nowy mikroserwis ma pełnić funkcję Naming Servera więc potrzebujemy to zakomunikować światu. Wobec tego należy dodać @EnableEurekaServer do klasy bootstrapującej nasz mikroserwis.

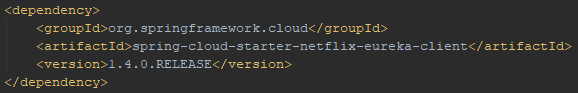
Ustawienia: 

Port numer 8761 jest typowym portem uruchomieniowym serwera Eureki. Warto również zadbać, aby nasz Naming Server nie traktował sam siebie jego klienta i wyłączyć tą opcję oraz wyłączyć opcję automatycznego szukania innych klientów ponieważ po włączenia naszych instancji ExchangeService oraz reszty z naszych serwisów, innych zwyczajnie nie ma prawa być w naszym systemie. Po przejściu do przeglądarki możemy zaobserwować, że nasza Eureka w pełni działa:



Tylko z aplikacjami bieda bo nic nie odpaliłem (No instances available), ale tu coś kiedyś będzie i to potem pokaże więc spokojnie.

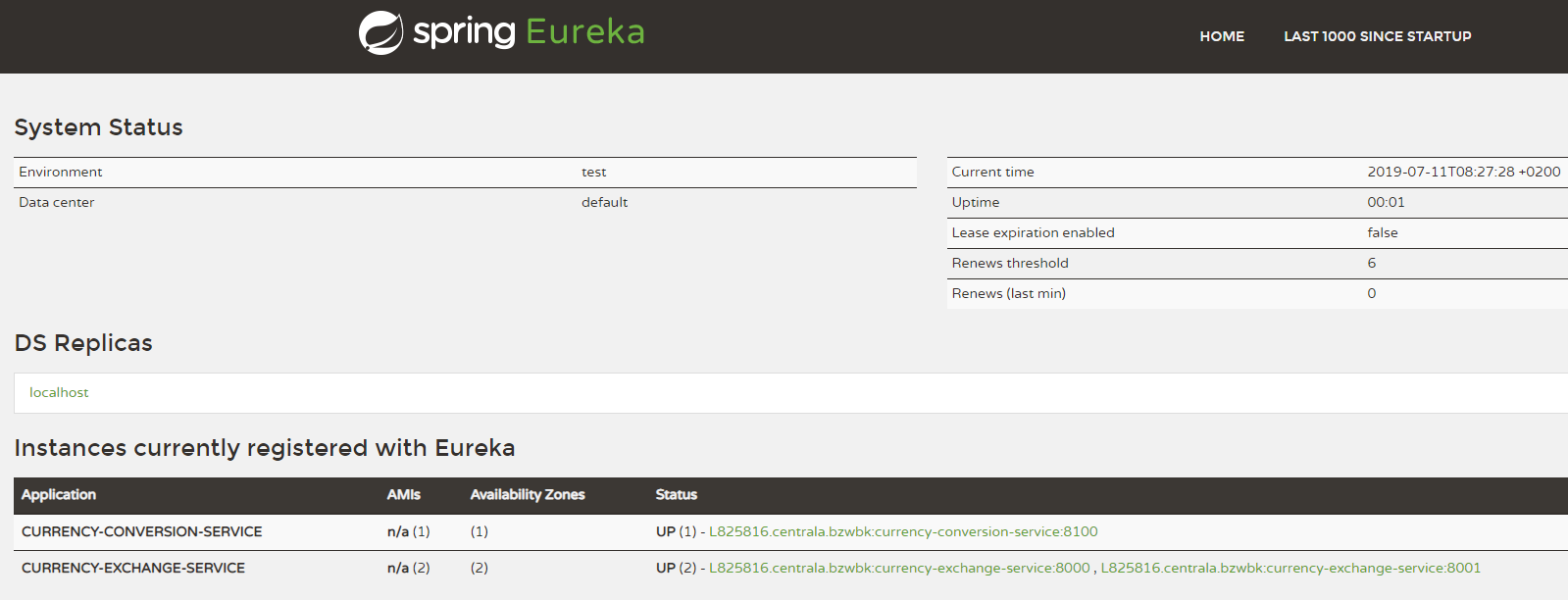
Podłączmy teraz nasze mikroserwisy do Naming Servera. Zacznijmy od dodania zależności w każdym z mikroserwisów, które chcemy podłączyć.



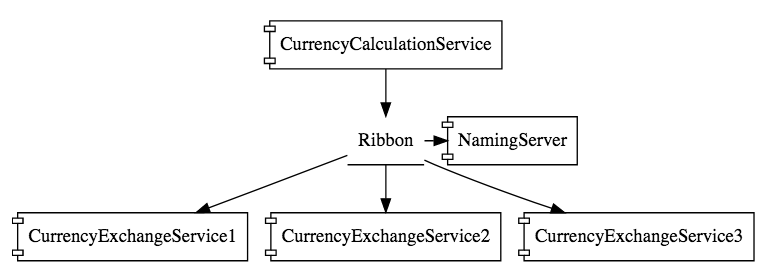
Oraz zarejestrujmy je w naszym nowym Eureka Naming Server. Uzyskamy to dodając @EnableDiscoveryClient w każdej z klas głównych w naszych mikroseriwsach. Można stosować również @EnableEurekaClient to już opcjonalne.

„There are multiple implementations of "Discovery Service" (eureka, consul, zookeeper). @EnableDiscoveryClient lives in spring-cloud-commons and picks the implementation on the classpath. @EnableEurekaClient lives in spring-cloud-netflix and only works for eureka. If eureka is on your classpath, they are effectively the same.” – StackOverflow

Ponadto każdy z naszych klientów powinien wiedzieć gdzie znajduje się nasz Naming Server więc należy dodać  do każdego z mikroserwisów, który ma zostać podłączony. Po odpaleniu CurrencyConversionService oraz dwóch instancji ExchangeService możemy zobaczyć w jaki sposób Eureka je zarejestrowała:



Widzimy odpalone 2x ExchangeService oraz CurrencyConversionService, na portach 8000,8001,8100.

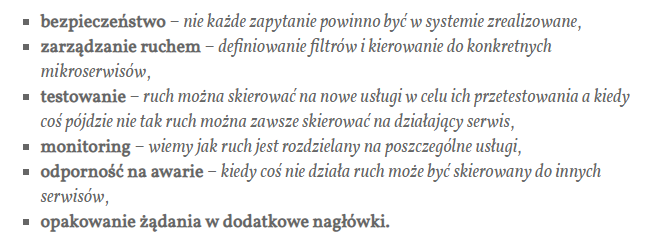


Wrzucę jeszcze raz bo ważną rzecz do zrozumienia musi być mechanizm współpracy między Ribbon, a jakimś NamingServer w naszym przypadku Eureka. Jeśli CurrencyCalculationService chce skomunikować się z dowolną instancją CurrencyExchangeService to musi z pomocą Ribbon-a spytać Naming Server o szczegóły zarejestrowanych na nim serwisów i dopiero po otrzymaniu informacji zwrotnej uzyska możliwość komunikowania się z CurrencyExchangeService, które obecnie są podłączone.

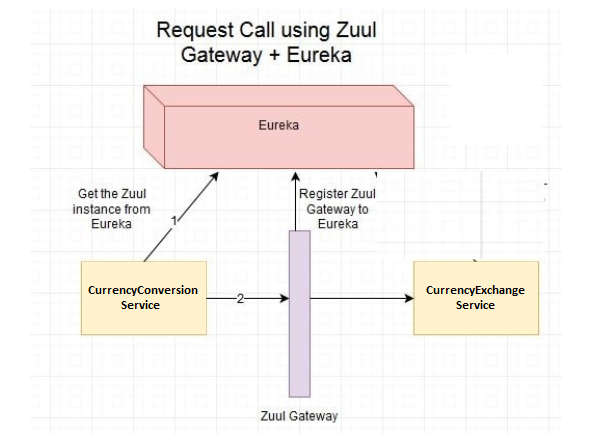
Zróbmy małe podsumowanie tego co do tej pory zrobiliśmy. Chodzi o małe powtórzenie drogi implementacji Eureka, Ribbon i Feign. Więc: zaczęliśmy od prostej implementacji dwóch mikroserwisów tj. CurrencyConversionService i CurrencyExchangeService, z których ten pierwszy jest konsumentem zasobów tego drugiego. Wskazaliśmy na sztywno port do komunikacji między nimi, ale wyszedł problem co w przypadku gdy będziemy chcieli mieć dwie instancje ExchangeService. Dodaliśmy Feign aby w łatwy sposób umożliwić komunikację CurrencyConversionService z CurrencyExchangeService. Po tym dodaliśmy Ribbon i uzyskaliśmy dostęp do obu instancji ExchangeService jednocześnie. Wszystkie połączenia między CurrencyConversionService, a CurrencyExchangeService mogły wówczas być dystrybuowane między obie instancje CurrencyExchangeService. Uzyskaliśmy to poprzez hardkoding dwóch url’s w pliku konfiguracyjnym naszego konsumenta usług tj. CurrencyConversionService. Ale nam to nie wystarczyło ☺ i dodaliśmy Eureka Naming Server i uzyskaliśmy całkiem sporo. Zamiast hardkodować url’s do serwisów z którymi nasz konsument może się komunikować, po dodaniu Naming Servera możemy wprost jako CurrencyConversionService spytać się go o dostępne mikroserwisy i uzyskać w odpowiedzi szczegóły interesującego nas aktywnego mikroserwisu.

# API Gateway – Zuul

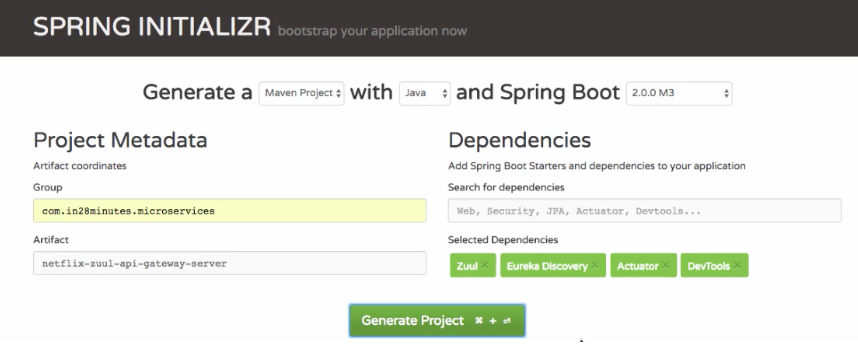
Wprowadzając w temat: Zull jest serwerem, który pracuje w charakterze API Gateway czyli na polski bramy do systemu. Pod tą bramą czy też jak gdzieś widziałem – fasadą – kryją się wywołania danych mikroserwisów, a ruch do nich jest kierowany za pomocą wcześniej zdefiniowanych filtrów. Wynalazłem zalety. Źródło – gdzieś w internecie:



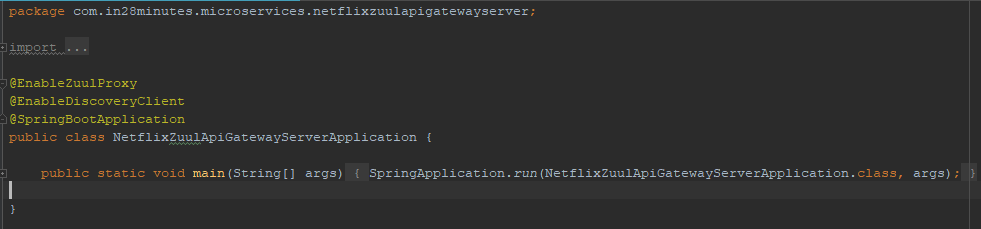
Baa… znalazłem obrazek prezentujący mniej więcej idee naszego tutka. Oczywiście go obrobiłem ☺



Postawmy więc nasz Zuul Gateway w 3 krokach. Po pierwsze stwórzmy go w ogóle. Po drugie określmy co on ma robić między naszymi mikroserwisami. Po trzecie upewnijmy się, że wszystkie ważne żądania przechodzą przez Gateway. Więc numero uno:



Dodajmy mu możliwość pracy jako Zuul API Gateway oraz zarejestrujmy w naszym Naming Serwerze.



I wyspecyfikujmy konfigurację:



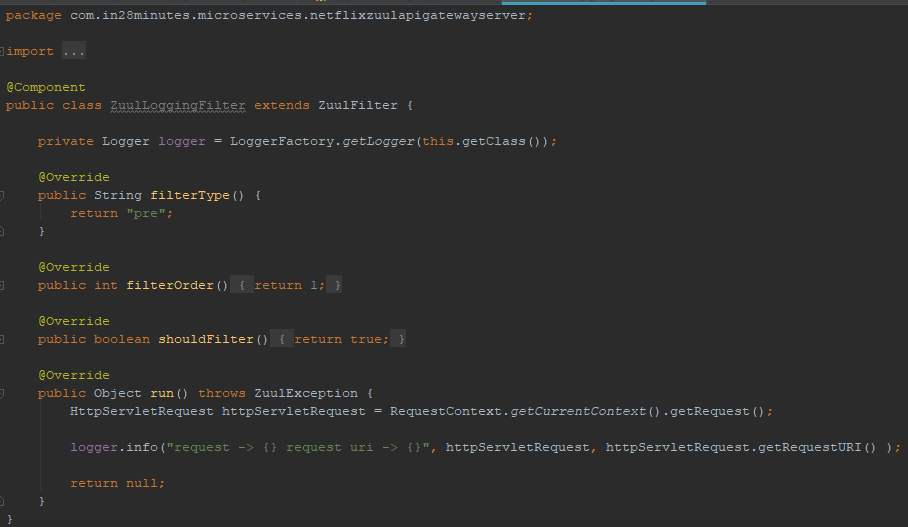
Stwórzmy teraz prosty mechanizm logowania, tak żeby każdy request przechodzący przez Gateway logował swoją obecność. Na tą okoliczność stwórzmy klasę, która nam w tym pomoże z zastrzeżeniem, że taka klasa powinna rozszerzać klasę **ZuulFilter.** Dzięki temu kompilator zadba abyśmy nadpisali kilka metod potrzebnych do implementacji filtra. Metoda filterOrder() zapewnia kolejność filtrowania co oznacza, że gdybyśmy mieli kilka filtrów np. ten nasz – ZuulLoggingFilter, ale również jakiś SecurityFilter czy jeszcze inny to właśnie dzięki niej możemy nadać naszym filtrom priorytety. Metoda shouldFilter() w momencie gdy jej wartość zwracana zostanie ustawiona na true to uruchamia metodę run() i nasz filter jest włączony. Inaczej… metoda shouldFilter() decyduje czy chcemy mieć włączony nasz filter czy nie. Natomiast metoda run() jest core-ową metodą naszego filtra i to właśnie ona odpowiada za filtrowanie wg zaimplementowanego schematu. Ostatnia metoda to filterType(), dzięki której możemy określić czy żądania będą filtrowane przed (”pre”) czy po (”post”) ich wykonaniu, albo możemy filtrować tylko błędne żądania (”error”).

Przejdźmy do implementacji.

Co chcemy logować?

Chcemy logować szczegóły aktualnego żądania, które przechodzi w danym momencie przez Gateway. Metoda getRequest() zwróci nam szczegóły żądania HTTP, które zostało złapane w tym momencie. Teraz fajnie by było żebyśmy zalogowali ten request w jakiś sposób. Stwórzmy więc sobie zmienną typu Logger i zaciągnijmy odpowiednią bibliotekę. Dodajmy logowanie do naszej metody run().

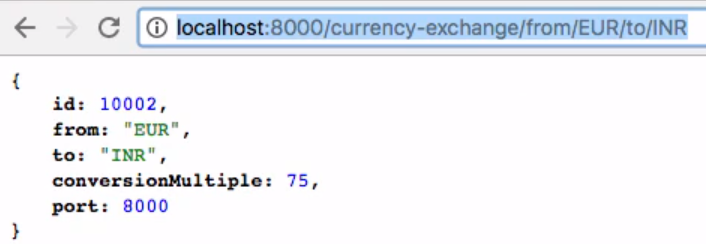
W sumie tyle, kod poniżej:



Skrótowo – utworzyliśmy filter (albo filtr może ☺ ), który filtruje żądania przed ich wykonaniem (”pre”), który jest włączony (true), który ma najwyższy priorytet (1) oraz taki który w przypadku przechodzenia żądania przez Gateway loguje tą informację za pomocą konkretnego komunikatu.

Potestujmy teraz trochę nasze rozwiązanie:

Odpalmy sobie nasz Eureka Naming Server, CurrencyExchangeService, CurrencyConversionService no i oczywiście nasz ZuulAPIGateway. Po wejściu w Eurekę – localhost:8761/ widzimy odpalone instancje mikroserwisów. Wszystkie uruchomione mikroserwisy są poprawnie zarejestrowane w Eureka i to jest cool ☺

Widzimy efekt wywołania usługi exchangeService-u: 

Ale chcemy uzyskać ten efekt dopiero po przejściu żądania przez nasz Gateway. Obecnie bezpośrednio uzyskaliśmy dostęp do danych z serwisu i pominęliśmy Gateway. Naprawmy to ☺

Ale jak?

Port na jakim uruchomiony jest nasz Zuul Gateway to localhost:8765 więc po zasób powinniśmy uderzyć właśnie pod ten port. Następną rzeczą jaką musimy zrobić jest powiedzenie „Gateway-owi”, od jakiego, podłączonego do niego mikroserwisu chcemy uzyskać odpowiedź podajac nazwę z jaką mikroserwis zarejestrował się w naszym Naming Server. Kończąc dodajemy uri naszego mikroserwisu i żądanie jest gotowe.

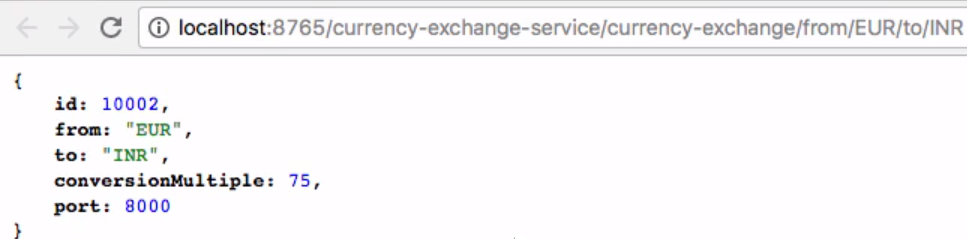
**Bezpośrednio: http://localhost:8000/currency-exchange/from/EUR/to/INR**

**Schemat: http://localhost:8765/{nazwa mikroserwisu w eureka}/{uri mikroserwisu}**

**U nas:** [**http://localhost:8765/currency-exchange-service/currency-exchange/from/EUR/to/INR**](http://localhost:8765/currency-exchange-service/currency-exchange/from/EUR/to/INR)

BTW -> INR – rupia indyjska ☺

I co w związku z tym powinien zrobić nasz Zuul API Gateway? A no powinien zalogować (bo zaimplementowaliśmy filter), że podane wyżej żądanie faktycznie przeszło przez Gateway i zwrócić nam zasób jakiego oczekujemy.



Et voila :D mamy wynik. Sprawdźmy teraz poprawność logowania.

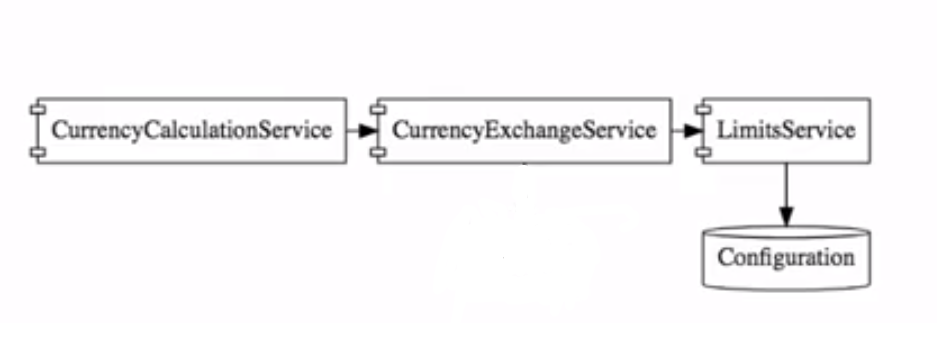


Wkleję przecięte na pół bo tu w sumie nic nie widać :/



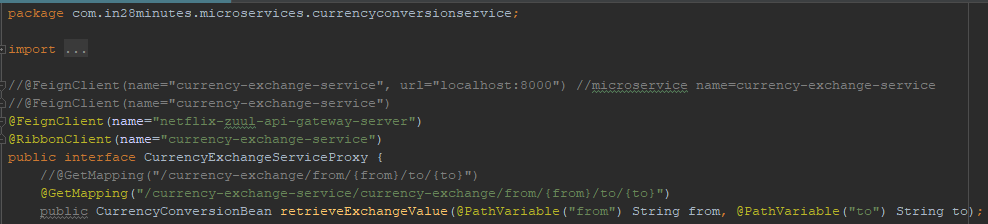
* 

Co teraz? Teraz pokombinujemy i całą odpowiedzialność za ruch w naszym „systemie” zwalimy na Gateway. Luz, w sensie każdy request faktycznie będzie dawał znać, że przeszedł przez Gateway.



Zrobimy coś takiego, że w momencie gdy CurrencyCalculationService faktycznie chce pogadać z CurrencyExchangeService to nie uderza do niego bezpośrednio, ale ruch kierowany jest najpierw do naszego Zuul API Gateway a dopiero potem do CurrencyExchangeService i w ten sposób żądanie zostawi po sobie ślad w postaci logów w Gateway-u.

Dobra to teraz wypada w jakiś sposób wskazać CurrencyCalculationService, że ruch z niego powinien być kierowany na Zuul Gateway.

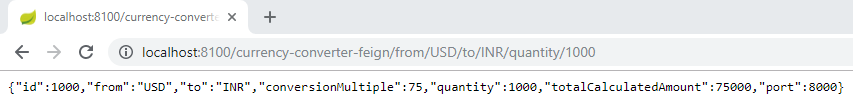


Pierwsze co trzeba powiedzieć, to żeby nasz Feign nie łączył się z CurrencyExchangService bo wtedy uzyskujemy bezpośrednie połączenie z mikroserwisem CurrencyExchange. Zależy nam aby logować żądanie **w Gateway-u** więc jako FeignClient musimy wskazać nasz Zuul API Gateway. Drugą rzeczą do zmiany jest ścieżka do zasobu, który chcemy uzyskać, a jak już wiemy bo przeczytaliśmy cosik kilka linijek wyżej Zuul do zasobu dobiera się u nas poprzez:

**Schemat: http://localhost:8765/{nazwa mikroserwisu w eureka}/{uri mikroserwisu}**

Należy wobec tego zmienić ścieżkę docelową uwzględniając powyższy schemat.

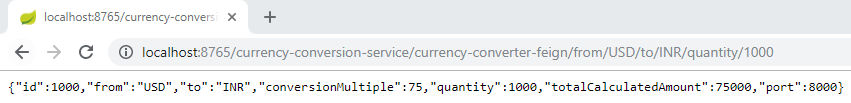
Potestujmy. Odwołajmy się bezpośrednio do CurrencyConversionService.



Oczywiście dostajemy ładną zwrotkę. Teraz sprawdzimy co zostało zalogowane.

*2019-07-13 16:56:39.252 INFO [netflix-zuul-api-gateway-server,b4b08839a687f79b,7546f4c1de9a24a5,true] 27888 --- [nio-8765-exec-3] c.i.m.n.ZuulLoggingFilter : request -> org.springframework.cloud.netflix.zuul.filters.pre.Servlet30RequestWrapper@36f80a3d request uri -> /currency-exchange-service/currency-exchange/from/USD/to/INR*

Jak widzimy przy odpowiedzi nasz Zuul API Gateway (logi oczywiście sprawdzamy na Gateway-u bo przecież o to nam chodzi) zalogował poprawnie to co faktycznie zostało zażądane. Teraz pora na test numer 2. Sprawdzimy w jaki sposób zachowa się nasz Gateway gdy uderzymy po zasób nie od strony mikroserwisu (CurrencyConversionService), ale właśnie od strony Gateway-a zażądamy odpowiedzi.



Ti ru ri ru, co dostaliśmy?

*2019-07-13 17:08:22.735 INFO [netflix-zuul-api-gateway-server,977361767b8e9697,977361767b8e9697,true] 27888 --- [nio-8765-exec-3] c.i.m.n.ZuulLoggingFilter : request -> org.springframework.cloud.netflix.zuul.filters.pre.Servlet30RequestWrapper@7902e8a9 request uri -> /currency-conversion-service/currency-converter-feign/from/USD/to/INR/quantity/1000*

*2019-07-13 17:08:22.751 INFO [netflix-zuul-api-gateway-server,977361767b8e9697,4750d4772cfeca5d,true] 27888 --- [nio-8765-exec-5] c.i.m.n.ZuulLoggingFilter : request -> org.springframework.cloud.netflix.zuul.filters.pre.Servlet30RequestWrapper@2803a4ca request uri -> /currency-exchange-service/currency-exchange/from/USD/to/INR*

W logach konsoli widzimy dwa momenty, kiedy został uruchomiony nasz filter logujący i to jest cool! Za pierwszym razem zalogowaliśmy ruch do serwisu CurrencyConversion, a jak wiemy ma on zaimplementowany mechanizm, który odwołuje się do CurrencyExchange więc musi „strzelić” po zasoby właśnie tam. I dzięki naszej magii nie komunikuje się z nim bezpośrednio, a właśnie przez nasz Gateway ☺ i dopiero po uzyskaniu zgody (u nas to zwykłe logowanie obecności) ze strony bramy może się do ExchangeService dobrać.

Czemu uważam to za mega dobre rozwiązanie? Ponieważ wszystkie żądania przechodzą przez jedno miejsce. Dopracowując nasz Gateway mamy jedno miejsce gdzie możemy kontrolować w pełni przepływ ruchu we wszystkich mikroserwisach tworzących system. Możemy dzięki temu logować wszystko co chcemy o wszystkim co chcemy (xD), możemy zaimplementować filtry bezpieczeństwa, autentykacji, autoryzacji i masę innych – I TO WSZYSTKO W JEDNYM MIEJSCU ☺

# Rozproszone śledzenie

Uaa… a na co to komu?

Przez śledzenie jeśli mam odnosić się do wytwarzania oprogramowania, rozumiem, że punkt w którym jesteśmy w programie jest nam znany na każdym etapie pracy z tym programem. Śledzimy nasz program i w odpowiednim momencie, odpowiednio reagujemy. Ale… jak śledzimy? A no logujemy sobie dane z różnych, najczęściej dużej ilości miejsc w których coś może pójść nie tak co ułatwia nam potem naprawę błędów. Noi to jest spoko bo wchodzimy potem w logi i widzimy – oho tu coś nie zadziałało i mamy dobry początek w kwestii rozwiązania problemu. Takie rozwiązanie śmiało można stosować w aplikacjach monolitycznych, ale rozważając nasz system mikroserwisowy problem monitorowania aplikacji jest bardziej złożony bo sprawdzanie logów jest żmudne i czasochłonne. Bo wyobraźmy sobie sklep internetowy. Jeden zakup w takich rozproszonym sklepie to tak naprawdę odwiedzenie kilku serwisów. Niestety podczas zakupu wystąpił błąd i musimy to naprawić. Analiza logów jest tak strasznie czasochłonna w tym wypadku, że ludzie z natury leniwi wymyślili sobie takiego Zipkina i to był strzał w dziesiątkę.

## Sleuth

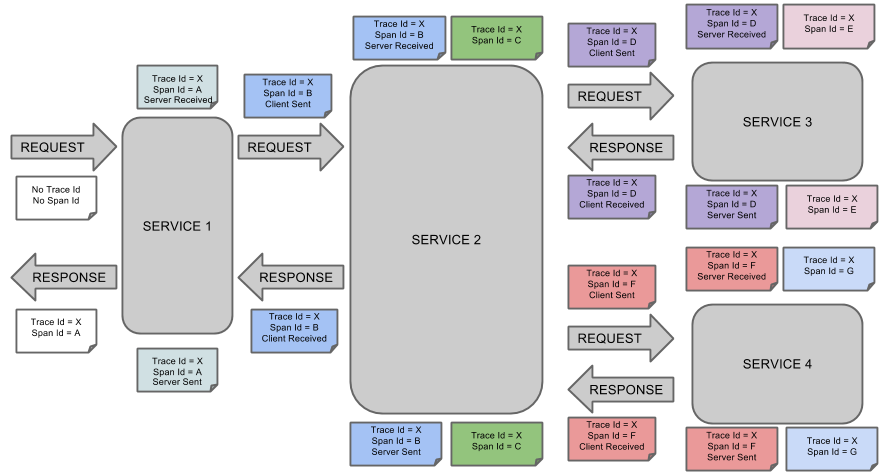
A to co za twór?

**Sleuth** jest bardzo przydatny i chyba nieodłączny jeśli chodzi o Zipkin. Zapewnia on tak zwane correlation ID czyli unikalne ID który jest przekazywany pomiędzy requestami w jednym logicznym ciągu. Logiczny ciąg operacji w nomenklaturze Dappera oznaczany jest jako **trace**. Natomiast każda pośrednia operacja (jedna jednostka pracy, na przykład wysłanie requestu) w obrębie jednego trace nazywana jest **spanem**.

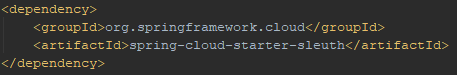
Każda operacja składa się z czterech eventów:

* **Client Send –** klient stworzył i wysłał request
* **Server Received –** server odebrał request i zaczyna go przetwarzać
* **Server Send –** server przetworzył request i wysyła go z powrotem do klient
* **Client Received –** klient otrzymał przetworzony request

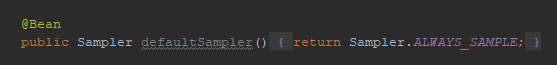
Fajny obrazek – źródło gdzie w Internecie – prezentujący przepływ przez cztery serwisy:



Dodajmy sobie teraz Spring Cloud Sleuth do naszego projektu. Dodajmy go sobie do CurrencyConversionService, CurrencyExchangeService i do naszego Zuul API Gateway. Żeby to zrobić musimy przejść przez dwa proste kroki.

Numer 1 to konieczność dodania zależności Spring Cloud Sleuth do mavenowskiego pom.xml:  


Następny krok to określenie co chcemy aby nasz Sleuth w każdym z mikroserwisów przechwytywał i w sposób przemyślany zaimplementowanie tego. Możemy to osiągnąć poprzez stworzenie w klasie bootstrapującej mikroserwis beana, który będzie za to odpowiadał:



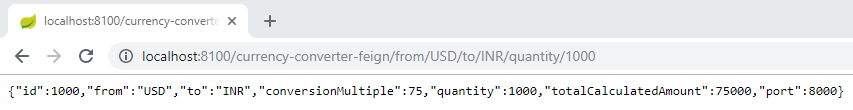
Oczywiście modyfikacje wprowadzamy we wszystkich mikroserwisach w których chcemy używać Sleuth dokładnie takie same.

Dodajmy trochę prostych logów do naszych CurrencyConversionService i CurrencyExchangeService.





Sprawdźmy działanie i wykonajmy trochę testów:



Po wywołaniu żądania sprawdźmy logi:

**CurrencyConversionService:**

2019-07-13 18:47:28.494 INFO [currency-conversion-service,**9e5a223c8fa411ef**,9e5a223c8fa411ef,true] 27480 --- [nio-8100-exec-6] c.i.m.c.CurrencyConverterController : [com.in28minutes.microservices.currencyconversionservice.CurrencyConversionBean@70c003a7](mailto:com.in28minutes.microservices.currencyconversionservice.CurrencyConversionBean@70c003a7)

**CurrencyExchangeService:**

2019-07-13 18:47:28.475 INFO [currency-exchange-service,**9e5a223c8fa411ef**,95b8054bfc633d0f,true] 78720 --- [nio-8000-exec-9] c.i.m.c.CurrencyExchangeController : com.in28minutes.microservices.currencyexchangeservice.ExchangeValue@1657b5fa

**Zuul API Gateway:**

2019-07-13 18:47:28.462 INFO [netflix-zuul-api-gateway-server,**9e5a223c8fa411ef**,14bff744dc88eae4,true] 77516 --- [nio-8765-exec-8] c.i.m.n.ZuulLoggingFilter : request -> org.springframework.cloud.netflix.zuul.filters.pre.Servlet30RequestWrapper@48c064bc request uri -> /currency-exchange-service/currency-exchange/from/USD/to/INR

WSPANIALE!!!

Czemu xd?

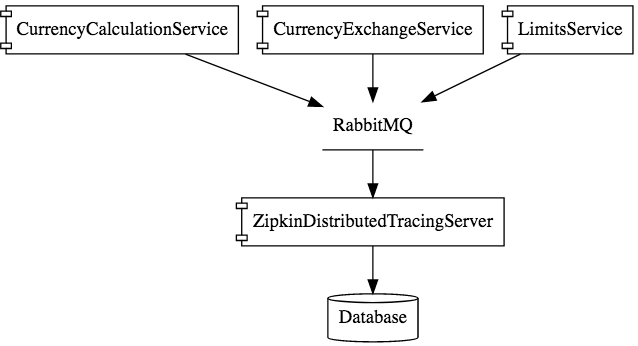
Ponieważ z pomocą Spring Cloud Sleuth jednoznacznie jesteśmy w stanie zidentyfikować, że ten request w trzech miejscach to ten sam request? Dlaczego?

Widzimy (dodałem pogrubienie) jak Sleuth nadał naszemu żądaniu unikalne id, które powtórzyło się w każdym miejscu po drodze przez jaką przechodził request. Tzn z CurrencyConversionService żądanie przeszło przez Zuul API Gateway do CurrencyExchangeService. BANG! 9e5a223c8fa411ef jest właśnie naszym unikatowym id requestu ☺

Ale pojawia się mały problem ponieważ tak jak widać te logi pojawiają się w wielu miejscach. Potrzebujemy wobec tego jakiegoś rozproszonego serwera, który te logi scentralizuje i będzie przechowywał w jednym miejscu. I tu wkracza Zipkin.

## Zipkin

Tak jak wspomniałem wyżej – inaczej – Zipkin jest swojego rodzaju bazą danych do której wysyłane są logi ze Sleutha. Używa się go aby w razie potrzeby nie musieć przeglądać logów każdej z aplikacji (mikroserwisu) oddzielnie. Z pomocą specjalnego systemu kolejkowania np. RabbitMQ możemy w jednym miejscu zgromadzić wszystkie interesujące nas informacje, które w razie potrzeby mogą usprawnić nam debuggowanie programu.



Jak można się domyślić po strzałkach – chodzi o to by wszystkie logi z utworzonych mikoroserwisów scentralizować oraz przesłać do serwera, który nimi ma zarządzać.

Pojawia się pytanie. W jaki sposób mogę sobie przesłać logi z CurrencyConversionService do naszego ZipkinDistributedTracingServer? Posłuży nam do tego RabbitMQ, do którego logi „wsadzą” wszystkie mikroserwisy i utworzą specjalną kolejkę, a Zipkin będzie nasłuchiwał co ciekawego ma mu do powiedzenia system kolejkowania ☺

Zainstalujmy sobie teraz nasz system kolejkowania:

<https://www.rabbitmq.com/install-windows.html>

filmik instruktażowy: <https://www.youtube.com/watch?v=gKzKUmtOwR4>

warto dodać, że do prawidłowego funkcjonowania RabbitMQ potrzeba zainstalować również Erlang:

<https://www.rabbitmq.com/which-erlang.html>

<https://www.erlang.org/downloads>

Po pobraniu i poprawnym zainstalowaniu RabbitMQ startuje automatycznie z domyślnymi ustawieniami. W każdej chwili można zatrzymać, wystartować ponownie czy zmienić ustawienia. Wszystkie ważne informacje znajdują się w „manualu” w linkach wyżej.

W następnym kroku uruchomimy rozproszone śledzenie z naszym Zipkin. Przy okazji taka informacja:

Żeby pobrać Zipkin należy wejść pod link: <https://zipkin.io/pages/quickstart> i wybrać interesujący nas sposób pobierania. Pobrać…

Proponuję aby przenieść pobranego jar-a do naszego katalogu projektowego tak aby potem w razie potrzeby go nie szukać po całym systemie.

Uruchamiamy jak każdy inny plik .jar --> cmd: **java –jar {nazwa jar-a do uruchomienia}**

Czekamy aż wszystko ładnie się uruchomi.

Sprawdźmy czy wszystko działa pod adresem **localhost:9411/zipkin/**

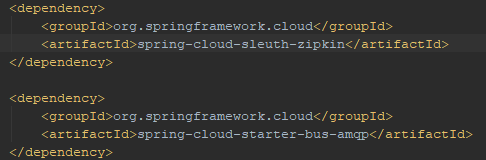
Teraz należy powiedzieć naszemu Zipkin Distributed Tracing Server że korzystamy z RabbitMQ i żeby zwrócił na to uwagę. W momencie pisania jestem na Windowsie, tak jak czytający to w tej chwili ☺ więc należy to zrobić używając dwóch rozdzielnych komend:

Pierwsza 🡪 **SET RABBIT\_URI=amqp://localhost**

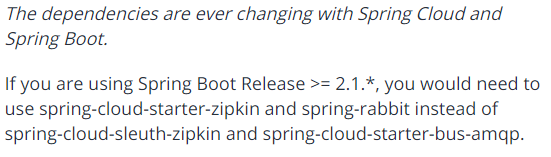
Druga 🡪 **java –jar {nazwa jar-a do uruchomienia}**

W tym momencie mamy uruchomiony RabbitMQ oraz ZipkinDistributedTracingServer, który nasłuchuje na logi z RabbitMQ. Brakuje nam połączenia naszych mikroserwisów do systemu kolejkowania i to właśnie zrobimy.

Żeby to osiągnąć musimy dodać odpowiednie zależności do naszych mikroserwisów – oczywiście tych, które chcemy aby były podłączone do systemu kolejkowania.



**Ważny Update!!!**





(Update należy wprowadzić do każdego zaktualizowanego wcześniej mikroserwisu)

Na potrzeby przetestowania systemu należy uruchomić wszystkie instancje mikroserwisów, których logi chcemy zebrać, nasz Gateway, Naming Server i mieć uruchomione Zipkin i RabbitMQ.

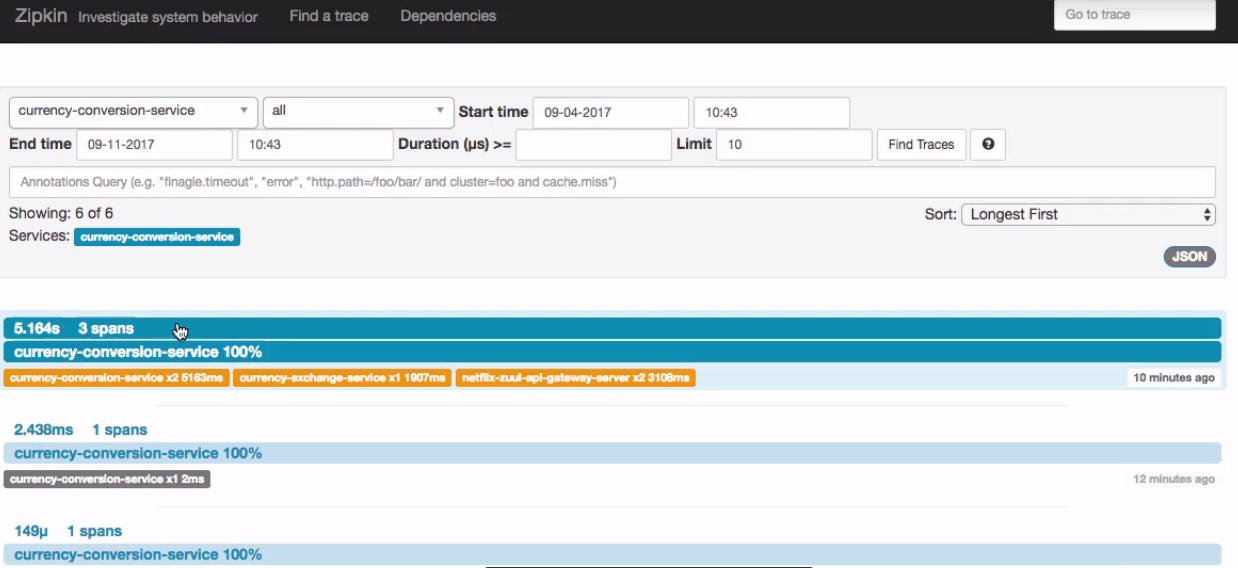
Wykonajmy sobie test i strzelmy pod:

**localhost:8100/currency-converter-feign/from/USD/to/INR/quantity/1000**

request zwrócił wynik, ale tym razem interesuje nas „background” naszego żądania. I w tym właśnie pomocny ma być Zipkin. Wejdźmy pod localhost:9411/zipkin/ i skonfigurujmy wyniki wyszukiwania wybierając spośród dostępnych mikroserwisów nasz tj. currency-conversion-service.

Kliknijmy „Find Traces”.

Wow… Zipkin listuje nam graficznie nasze żądania.

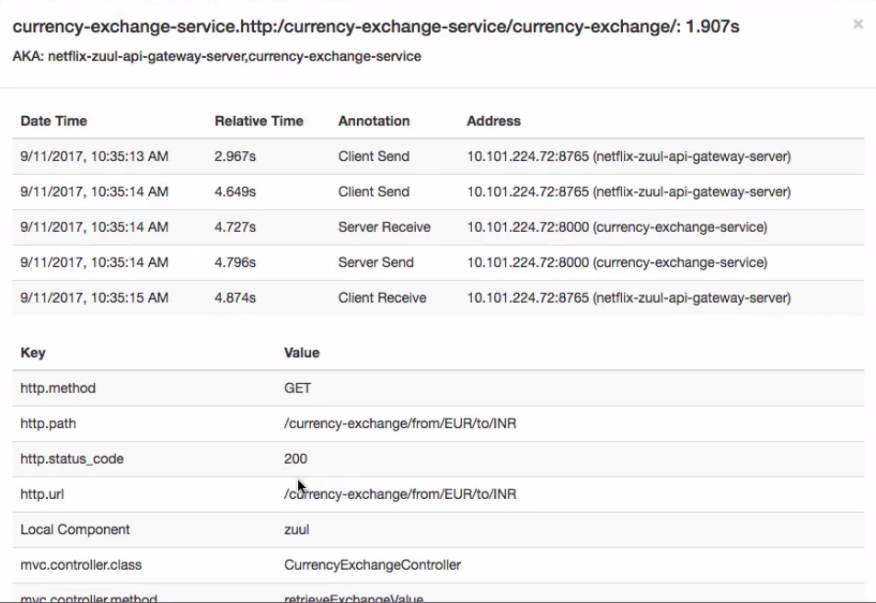


Wejdźmy w jeden:



Widzimy tu jaką dokładnie drogę przebył nasz request. Widzimy, że zanim żądanie przeszło z CurrencyConversionService do CurrencyExchangeService to po drodze zahaczyło o nasz Gateway. Jak powiedział prowadzący – AWESOME!

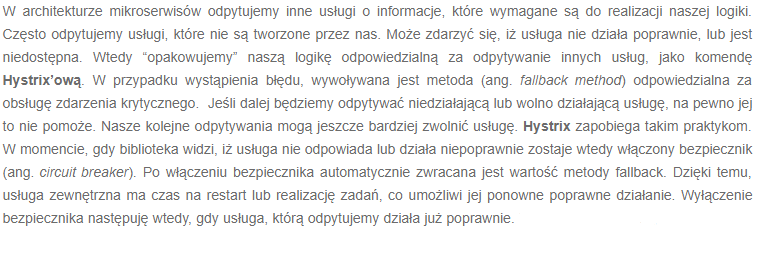
Mało? Gdy klikniemy na dowolny z kroków widzimy szczegóły:



# Odporność na uszkodzenia – Hystrix

Żeby zacząć mówić o Hystrix, warto wspomnieć o tym, że realizuje on podejście FailFast czyli podejście w którym w jak najszybszym czasie dostajemy informację o tym, że coś działa niepoprawnie. I to tyle wstępu, choć można tu wspomnieć o tym, że Hystrix realizuje swoje rozwiązanie jako CircuitBreaker czyli bezpiecznik, który jest uruchamiany gdy dzieje się coś niepożądanego.

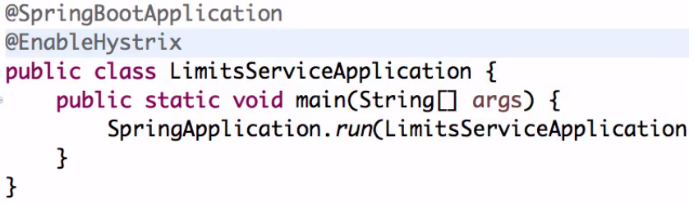
Przedstawmy zasadę działania. Źródło już klasycznie – gdzieś w Internecie:



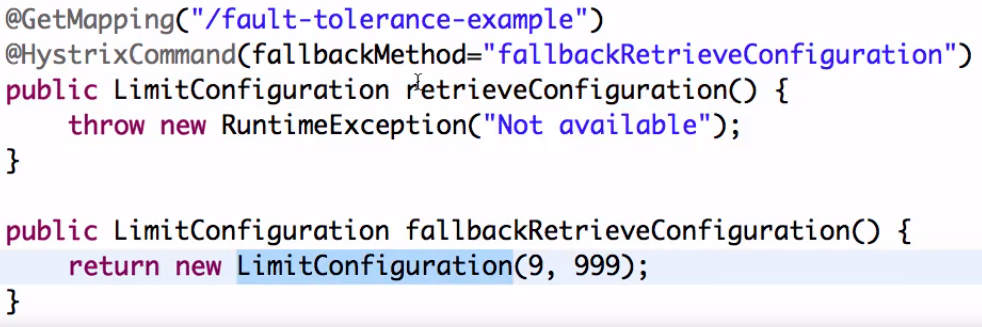
Dodajmy Hystrix do naszego LimitService. W związku z tym należy zaciągnąć odpowiednią zależność Hystrix-ową do naszego pom.xml:



I włączmy go dodając do naszej klasy bootstrapującej LimitService adnotację @EnableHystrix:



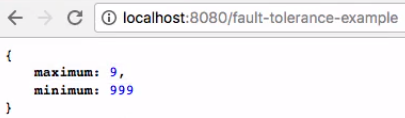
I zmodyfikujmy nasz LimitService-owy RestController:



Możemy dzięki @HystrixCommand(nazwaMetodyDoFallbacku) zdefiniować naszą metodę powrotną. Po co? Tak jak wyżej opisywałem – w momencie gdy metoda w naszym wypadku retrieveConfiguration() nie zadziała prawidłowo, to wywoła się nasza metoda powrotna to znaczy przekażemy sterowanie z metody niedziałającej do działającej – jakoś tak – takie zabezpieczenie ☺

Fakt, u nas metoda retrieveConfiguration() rzuca wyjątek :/ ale na potrzeby testów to wystarczy. Jako przekierowanie ustawiamy jej metodę fallbackRetrieveConfiguration() i implementujemy przykładowe ciało tej metody.

Przetestujmy:



Widzimy jak po wywołaniu metody retrieveConfiguration() przekazaliśmy sterowanie do metody fallbackRetreiveConfiguration() i otrzymaliśmy domyślne wartości w odpowiedzi.

# Podsumowanie

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 ;) Starałem się pisać jak najbardziej kolokwialnie, niemniej jednak jeśli pojawiły się jakieś pytania podczas czytania proszę śmiało o zadawanie ich. Baaa, ja na bank tu popełniłem nie jeden błąd w opracowywaniu tego więc wszelkie niedociągnięcia chętnie wyjaśnię i poprawię. Serio, serio bardzo chętnie odpowiem na wszystkie pytania bo w wielu momentach mogłem faktycznie napisać coś mega niezrozumiale, albo nie do końca szczegółowo tylko pobieżnie i że kilka rzeczy pojawiło się nagle na raz. W sumie tyle.

Kacper - nikt ważny ☺