https://dzone.com/articles/microservices-with-spring-boot-part-1-getting-star

We will create a couple of microservices and get them to talk to each other using Eureka Naming Server and Ribbon for client-side load balancing.

## Resources Overview

In this guide, we will create a Student Resource exposing three services using the proper URIs and HTTP methods:

* Retrieve all Students - @GetMapping("/students")
* Get details of specific student - @GetMapping("/students/{id}")
* Delete a student - @DeleteMapping("/students/{id}")
* Create a new student - @PostMapping("/students")
* Update student details - @PutMapping("/students/{id}")

## Microservices Overview - A Big Picture

In this series of articles, we would create two microservices:

* **Forex Service - Abbreviated as FS**
* **Currency Conversion Service - Abbreviated as CCS**

### Forex Service

Forex Service (FS) is the Service Provider. It provides currency exchange values for various currency. Let's assume that it talks to a Forex Exchange and provides the current conversion value between currencies.

An example request and response is shown below:

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

The request above is the currency exchange value for EUR to INR. In the response, conversionMultiple is 75. We will talk about port in the response a little later.

### Currency Conversion Service

Currency Conversion **Service (CCS) can convert a bucket of currencies into another currency**. It uses **the Forex Service to get current currency exchange values. CCS is the Service Consumer.**

An example request and response is shown below:

GET to <http://localhost:8100/currency-converter/from/EUR/to/INR/quantity/10000>

{

id: 10002,

from: "EUR",

to: "INR",

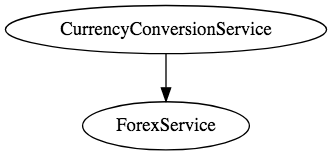
conversionMultiple: 75,

port: 8000,

}

The request above is to find the value of 10000 EUR in INR. The totalCalculatedAmount is 750000 INR.

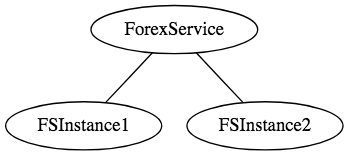
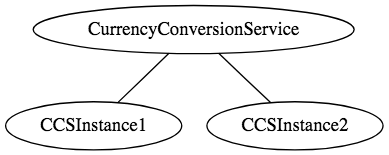
The diagram below shows the communication between CCS and FS.

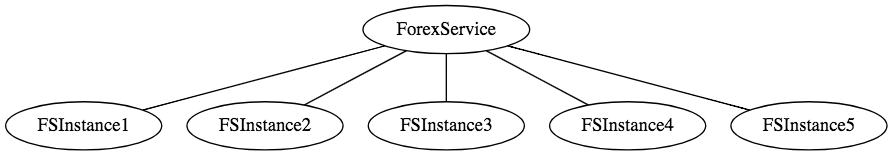


### \

### Eureka Naming Server and Ribbon

Based on the load, we can have multiple instances of the Currency Conversion Service and the Forex Service running.

.

The number of instances for each service might vary with time. Below picture shows a specific instance where there are five instances of the Forex Service.

What needs to happen in the **above situation is load should be uniformly distributed among these five instances.**

will use **Ribbon for Load Balancing and Eureka Naming server for registering all microservices.**

**What Is a Monolith Application?**

Have you ever worked on a project

* Which is **released (taken to production) once every few months**?
* Which has a wide range of features and functionality?
* Which has a team of more than 50 working on it?
* Where **debugging problems is a big challenge**?
* Where bringing in new technology and processes is almost impossible?

These are typical characteristics of a monolithic application. **Monolith applications are typically huge - more 100,000 line of code.** In some instances even more than million lines of code. Monoliths are characterized by

* Large ***Application Size***
* **Long Release Cycles**
* **Large Teams**

Typical challenges include

* **Scalability challenges**
* **New technology adoption**
* New processes - Agile?
* **Difficult to perform automation tests**
* **Difficult to adapt to modern development practices**
* Adapting to device explosion

## Microservices

Microservice architectures evolved as a **solution to the scalability and innovation challenges with monolithic architectures.**

There are a number of definitions proposed for microservices:

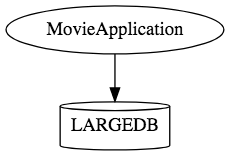
**Small autonomous services that work together. - Sam Newman**

**Developing a single application as a suite of small services each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies. - James Lewis and Martin Fowler**

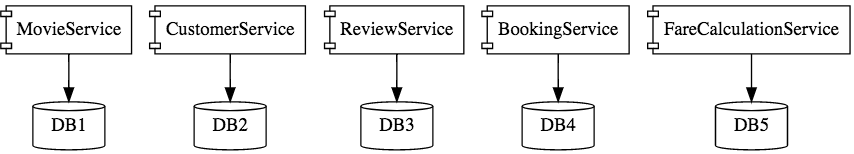
While there is no single accepted definition for microservices, for me, there are a few important characteristics:

* REST - Built around RESTful Resources**. Communication can be HTTP or event based.**
* ***Small*** Well Chosen **Deployable Units - Bounded Contexts**
* Cloud Enabled - **Dynamic Scaling**

## What Does Microservice Architecture Look Like?

This is how a monolith would look. One application for everything:

This is how the **same application would look like when developed using microservices architecture**.

This is how the same application would look like when developed using microservices architecture.

Microservice architectures involve a number of small, well-designed components interacting with **messages**.http://www.springboottutorial.com/images/Microservices-Chain-Example.png

**Advantages of Microservices**

Advantages of microservices include

* New **technology and process adaption becomes easier**. You can try new technologies with the newer microservices that we create.
* Faster release cycles.
* **Scaling with the cloud.**

## Challenges With Microservice Architectures

While developing a number of smaller components might look easy, there are a number of inherent complexities that are associated with microservices architectures. Let's look at some of the challenges:

* **Quick setup needed:** You cannot spend a month setting up each microservice..
* Automation: Because there are a number of smaller components instead of a monolith, you need to **automate everything - Builds, Deployment, Monitoring, etc.**
* Visibility: You now have a number of smaller components to deploy and maintain, maybe 100 or maybe 1000 components. **You should be able to monitor and identify problems automatically. You need great visibility around all the components.**
* Bounded Context**: Deciding the boundaries of a microservice is not an easy task.** Bounded contexts from Domain Driven Design are a good starting point. Your understanding of the domain evolves over a period of time. You need to ensure that the microservice boundaries evolve.
* Configuration Management: You need to **maintain configurations for hundreds of components across environments. You would need a Configuration Management solution**
* Dynamic scale-up and scale-down: The advantages of microservices will only **be realized if your applications can be scaled up and down easily in the cloud**.
* Pack of Cards: If a microservice at the bottom of the call chain fails, it can have **knock-on effects on all other microservices. Microservices should be fault tolerant by Design**.
* Debugging: When there is a problem that needs investigation**, you might need to look into multiple services across different** components. **Centralized Logging and Dashboards are essential to make it easy to debug problems.**
* Consistency:. While it is important to foster innovation**, it is also important to have some decentralized governance around the languages, platforms, technology and tools used for implementing/deploying/monitoring microservices.**

## Solutions to **Challenges with Microservice Architectures**

### Spring Boot

Spring Boot enables building production-ready applications quickly and provides non-functional features:

* **Embedded servers (easy deployment with containers**)
* Metrics (monitoring)
* Health checks (monitoring)
* **Externalized configuration**

### Spring Cloud

Spring Cloud provides solutions to **cloud-enable your microservices. It leverages and builds on top of some of the Cloud solutions opensourced by Netflix (Netflix OSS).**

#### Important Spring Cloud Modules

* Dynamically scale up and down. using a combination of
  + Naming Server (Eureka)
  + Ribbon (Client Side Load Balancing)
  + Feign (Easier REST Clients)
* Visibility and monitoring with
  + Zipkin Distributed Tracing
  + Netflix API Gateway
* Configuration Management with Spring Cloud Config Server.
* Fault Tolerance with Hystrix.

Implementing a solution for dynamic scale up and down needs to answer two questions

* How **does the Currency Conversion Service (CCS) know how many instances of Forex Service (FS) are active?(Naming server)**
* How does the **Currency Conversion Service (CCS) distribute the load between the active instances?(Load Balancing)**

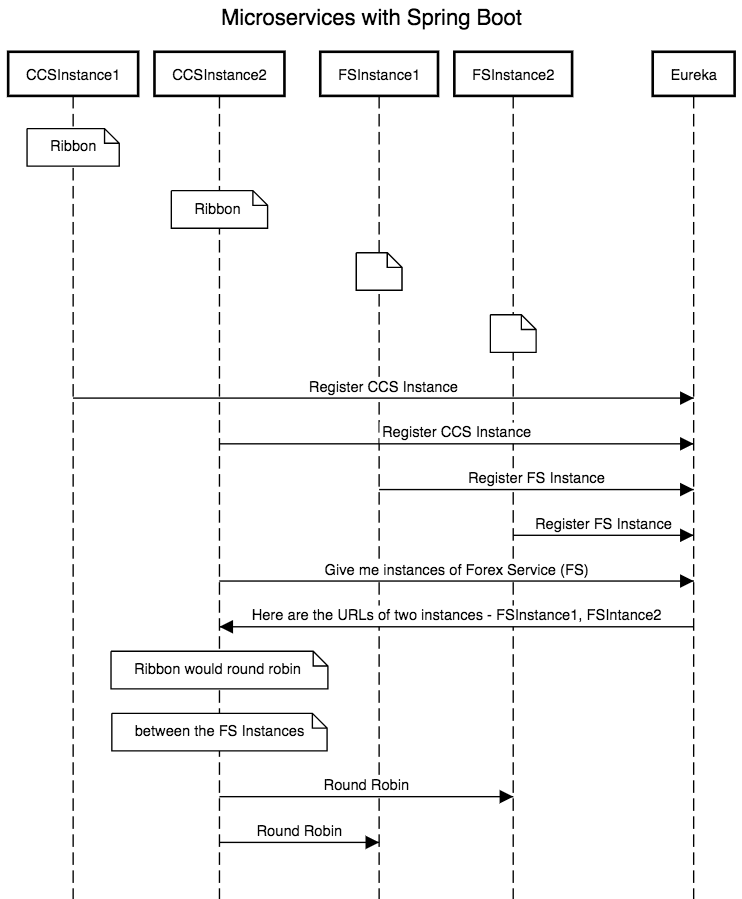
Because we **want this to be dynamic, we cannot hardcode the URLs of FS in CCS. That's why we bring in a Naming Server.**

**All instances of the components (CCS and FS) register with the Eureka Naming Server**. **When FS needs to call the CCS, it will ask Eureka Naming Server for the active instances.**\

**We will use Ribbon to do Client Side Load Balancing between the different instances of FS.**

A high-level sequence diagram of what would happen **when there is a request from CCS to FS is shown below:**

<http://www.springboottutorial.com/images/Spring-Boot-Microservice-7-Eureka-Sequence-Diagram.png>



Next in this series of articles:

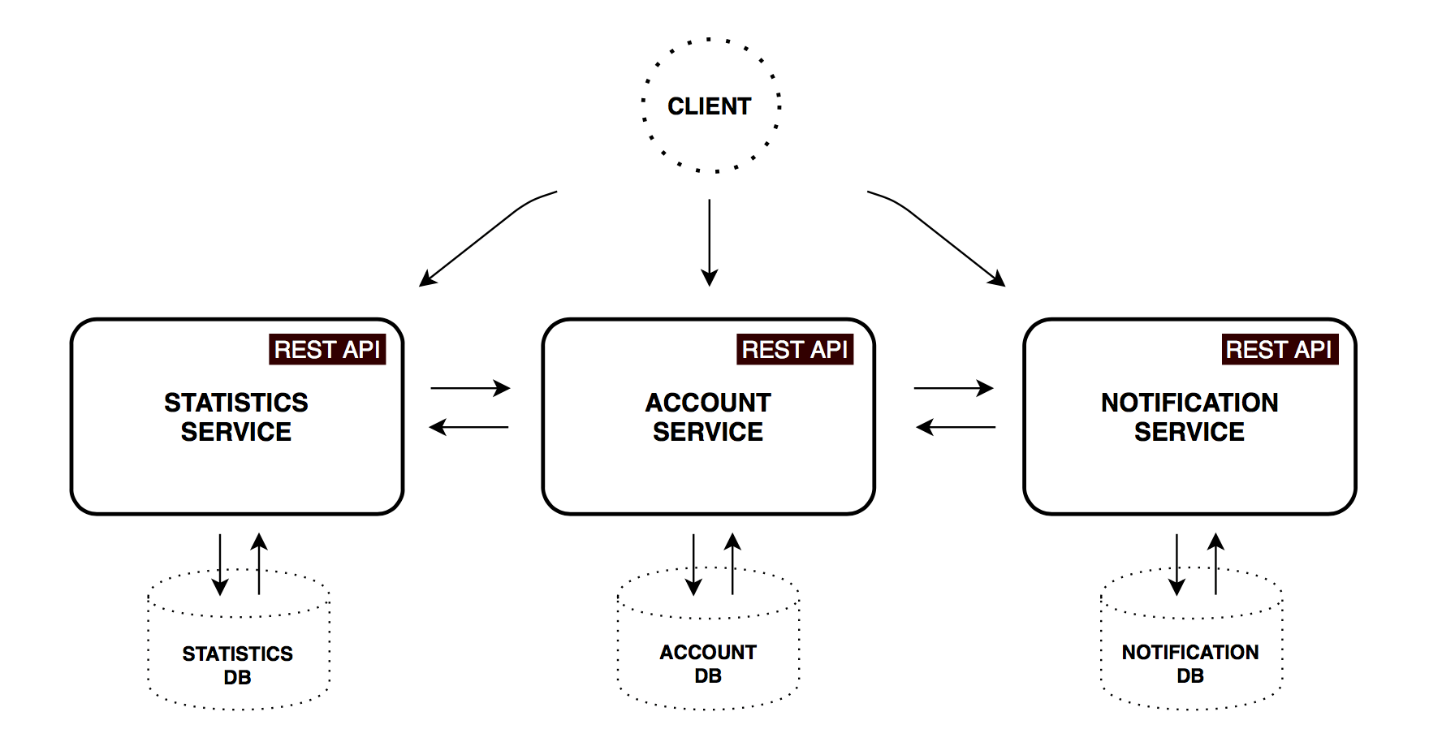
* Creating a Forex Microservice - We will create a **simple rest service based on Spring Boot Starter Web and Spring Boot Started JPA.** We will use Hibernate as JPA implementation and connect to H2 database.
* Create the CCS - Currency Conversion Service - We will **create a simple rest service using feign to invoke the Forex Microservice**
* Use Ribbon for Load Balancing.
* Implement Eureka Naming Service and connect FS and CCS through Eureka.

<https://dzone.com/articles/microservice-architecture-with-spring-cloud-and-do>

The article  provides a starting point for understanding common **Microservice architecture patterns by example of a proof-of-concept application built with Spring Boot, Spring Cloud, and Docker.**

The basis for this system I chose an old project, whose backend used to be a monolith. The application provides a way to deal with personal finances, organize incomes and expenses, manage savings, analyze statistics, and create simple forecasts.

**Functional Services**

The monolith application was **decomposed into three core microservices. All of them are independently deployable applications, organized around certain business capabilities.**

#### Account Service

Contains general user input logic and validation: incomes/expenses items, savings, and account settings.

| **METHOD** | **PATH** | **DESCRIPTION** | **USER AUTHENTICATED** | **AVAILABLE FROM UI** |
| --- | --- | --- | --- | --- |
| GET | /accounts/{account} | Get specified account data |  |  |
| GET | /accounts/current | Get current account data | × | × |
| GET | /accounts/demo | Get demo account data (pre-filled incomes/expenses items, etc) |  | × |
| PUT | /accounts/current | Save current account data | × | × |
| POST | /accounts/ | Register new account |  | × |

#### Statistics Service

Performs calculations on **major statistics parameters and captures time series for each account.** A datapoint contains values normalized to base currency and time period. This data might be used to track cash flow dynamics in an account's lifetime.

| **METHOD** | **PATH** | **DESCRIPTION** | **USER AUTHENTICATED** | **AVAILABLE FROM UI** |
| --- | --- | --- | --- | --- |
| GET | /statistics/{account} | Get specified account statistics |  |  |
| GET | /statistics/current | Get current account statistics | × | × |
| GET | /statistics/demo | Get demo account statistics |  | × |
| PUT | /statistics/{account} | Create or update time series datapoint for specified account |  |  |

#### Notification Service

Stores a user's **contact information and notification settings (like remind and backup frequency).** Scheduled worker collects required information from other services and sends e-mail messages to subscribed customers.

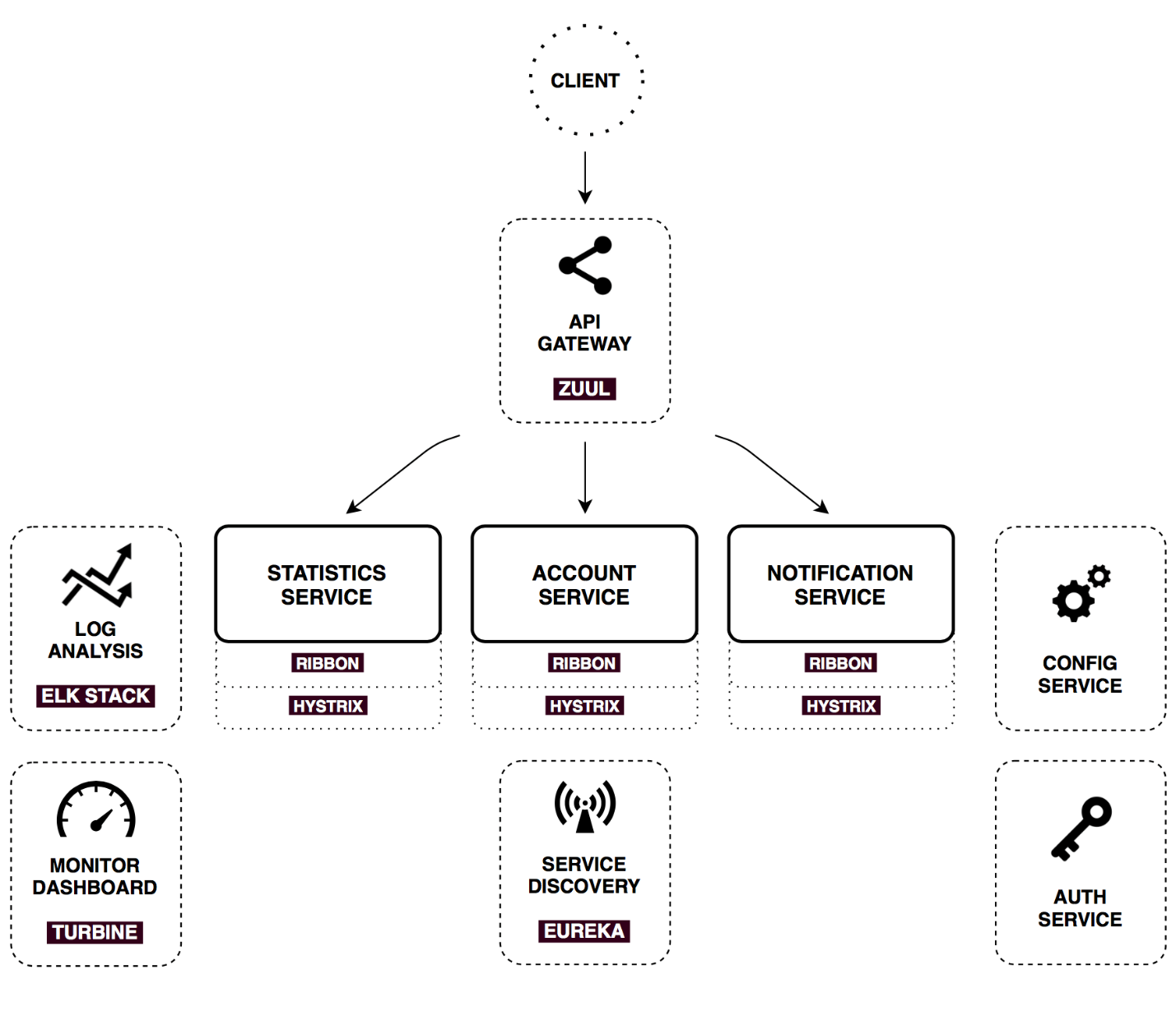
| **METHOD** | **PATH** | **DESCRIPTION** | **USER AUTHENTICATED** | **AVAILABLE FROM UI** |
| --- | --- | --- | --- | --- |
| GET | /notifications/settings/current | Get current account notification settings | × | × |
| PUT | /notifications/settings/current | Save current account notification settings | × | × |

#### Notes

* Each **microservice has it's own database, so there is no way to bypass the API and access persistance data directly.**
* For this project, I used MongoDB as the primary database for each service. It also might make sense to have a polyglot persistence architecture (to сhoose the type of database that is best suited to the service requirements).
* Service-to-service communication is quite simplified: **microservices talking using only synchronous REST API**. Common practice in a real-world systems is to use combination of interaction styles. For example, **perform synchronous GET request to retrieve data and use asynchronous approach via Message broker for create/update operations in order to decouple services and buffer messages.** However, this brings us in [eventual consistency](http://martinfowler.com/articles/microservice-trade-offs.html#consistency) world.

## Infrastructure Services

There's a bunch of common patterns in distributed systems, which could help us to make described core services work. [**Spring cloud**](http://projects.spring.io/spring-cloud/)**provides powerful tools that enhance Spring Boot applications behaviour to implement those patterns.** I'll cover them briefly.



**Config Service**

[Spring Cloud Config](http://cloud.spring.io/spring-cloud-config/spring-cloud-config.html) is **horizontally scalable centralized configuration service for distributed systems**. It uses a **pluggable repository layer that currently supports local storage, Git, and Subversion.**

In this project, I **use native profile, which simply loads config files from the local classpath.** You can see shared directory in [Config service resources](https://github.com/sqshq/PiggyMetrics/tree/master/config/src/main/resources). Now, **when Notification-service requests it's configuration, Config service responses with shared/notification-service.yml and shared/application.yml (which is shared between all client applications).**

**Client-side Usage**

Just build Spring Boot application with spring-cloud-starter-config dependency, autoconfiguration will do the rest.

Now you don't need any embedded properties in your application. Just provide bootstrap.yml with the **application name** and Config service url:

spring:

application:

name: notification-service

cloud:

config:

uri: http://config:8888

fail-fast: true

**With Spring Cloud Config, You Can Change App Configuration Dynamically**

For example, the [EmailService bean](https://github.com/sqshq/PiggyMetrics/blob/master/notification-service/src/main/java/com/piggymetrics/notification/service/EmailServiceImpl.java) was annotated with @RefreshScope. That means you can change e-mail text and subject lines without rebuilding and restarting the Notification service application.

First, change the required properties in the Config server. Then, perform the refresh request to the Notification service: curl -H "Authorization: Bearer #token#" -XPOST http://127.0.0.1:8000/notifications/refresh

You could also use [webhooks to automate this process](http://cloud.spring.io/spring-cloud-config/spring-cloud-config.html" \l "_push_notifications_and_spring_cloud_bus).

**Notes**

* There are some limitations for dynamic refreshes though. @RefreshScope doesn't work with @Configuration classes and can't affect @Scheduled methods.
* fail-fast property means that the Spring Boot application will fail startup immediately if it cannot connect to the Config Service. That's very useful when you're starting [all applications together](https://github.com/sqshq/PiggyMetrics#how-to-run-all-the-things).
* There are significant security notes below.

**Auth Service**

Authorization responsibilities are **completely extracted to separate server, which grants**[**OAuth2 tokens**](https://tools.ietf.org/html/rfc6749)**for backend resource services**. **Auth Server is used for user authorization as well as for secure machine-to-machine communication inside a perimeter.**

In this project, I use [Password credentials](https://tools.ietf.org/html/rfc6749#section-4.3) as a grant type for user authorization (since it's only used by the native application UI) and [Client Credentials](https://tools.ietf.org/html/rfc6749#section-4.4) as a grant type for microservices authorization.

Spring Cloud **Security provides convenient annotations and autoconfigurations to make this really easy to implement from both the server and client side**. You can learn more about it in the [documentation](http://cloud.spring.io/spring-cloud-security/spring-cloud-security.html) and check configuration details in [Auth Server code](https://github.com/sqshq/PiggyMetrics/tree/master/auth-service/src/main/java/com/piggymetrics/auth).

From the client side, everything works exactly the same as with traditional session-based authorization. You can retrieve Principal objects from request, check user roles and other stuff with expression-based access control and @PreAuthorize annotation.

Each client in PiggyMetrics (account-service, statistics-service, notification-service and browser) has a scope: server for backend services, and ui - for the browser. So we can also protect controllers from external access, for example:

@PreAuthorize("#oauth2.hasScope('server')")

@RequestMapping(value = "accounts/{name}", method = RequestMethod.GET)

public List<DataPoint> getStatisticsByAccountName(@PathVariable String name) {

return statisticsService.findByAccountName(name);

}

**API Gateway**

As you can see, there are three core services, which expose external APIs to the client. **In a real-world system, this number can grow very quickly as well as whole system complexity. Actuallyy,**[**hundreds of services**](http://highscalability.com/amazon-architecture)**might be involved in rendering one complex webpage.**

**In theory, a client could make requests to each of the microservices directly.** But obviously there are challenges and limitations with this option, **like necessity to know all endpoints addresses, perform http request for each peace of information separately, merge the result on a client side**. Another problem is **non-web-friendly protocols, which might be used on the backend.**

Usually a much better approach is to use an **API Gateway. It is a single entry point into the system, used to handle requests by routing them to the appropriate backend service or by invoking multiple backend services and**[**aggregating the results**](http://techblog.netflix.com/2013/01/optimizing-netflix-api.html)**.** Also, it can be used for **authentication, insights, stress and canary testing, service migration, static response handling, active traffic management.**

Netflix **open sourced**[**such an edge service**](http://techblog.netflix.com/2013/06/announcing-zuul-edge-service-in-cloud.html)**, and now with Spring Cloud we can enable it with one @EnableZuulProxy annotation**. In this project, **I use Zuul to store static content (UI application) and to route requests to the appropriate microservices.** Here's a simple prefix-based routing configuration for the Notification service:

zuul:

routes:

notification-service:

path: /notifications/\*\*

serviceId: notification-service

stripPrefix: false

That **means all requests starting with /notifications will be routed to Notification service**. There is no hardcoded address, as you can see. **Zuul uses**[**Service discovery**](https://github.com/sqshq/PiggyMetrics/blob/master/README.md#service-discovery)**mechanism to locate Notification service instances and also**[**Circuit Breaker and Load Balancer**](https://github.com/sqshq/PiggyMetrics/blob/master/README.md#http-client-load-balancer-and-circuit-breaker)**, described below.**

**Service Discovery**

Another commonly known **architecture pattern is service discovery.** It allows automatic detection of network locations for service instances, which **could have dynamically assigned addresses because of auto-scaling, failures, and upgrades.**

The key part of service discovery is the registry. I used Netflix Eureka for this project. Eureka is a good example of the client-side discovery pattern, **when the client is responsible for determining the locations of available service instances (using a registry server) and load balancing requests across them.**

With Spring Boot, you can easily build Eureka Registry with a spring-cloud-starter-eureka-server dependency, @EnableEurekaServer annotation, and simple configuration properties.

Client support is enabled with @EnableDiscoveryClient annotation and bootstrap.yml with application name:

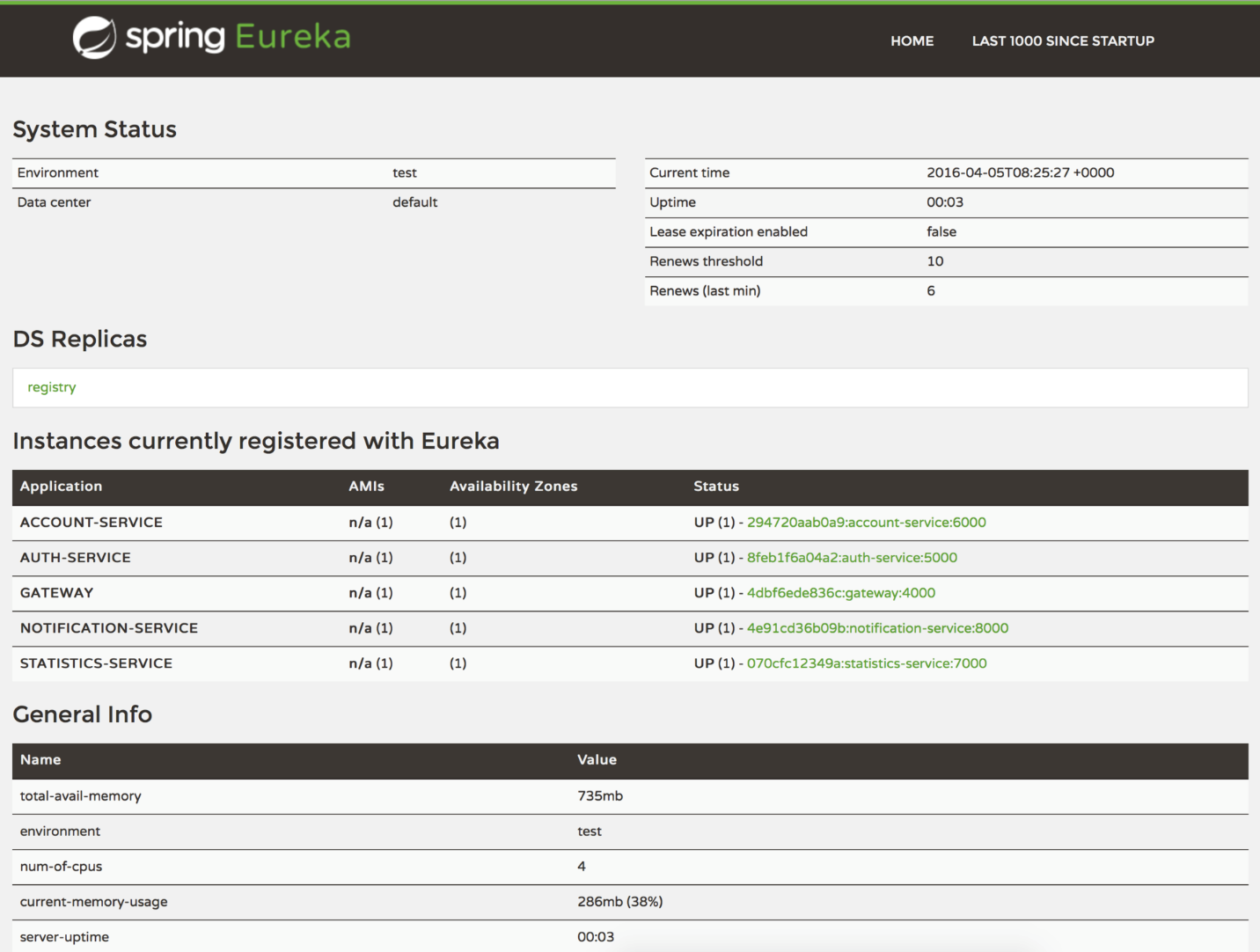
spring:

application:

name: notification-service

Now, on **application startup, it will register with Eureka Server and provide meta-data, such as host and port, health indicator URL, home page, etc**. Eureka receives heartbeat messages from each instance belonging to a service. If the heartbeat **fails over a configurable timetable, the instance will be removed from the registry.**

Also, Eureka provides a simple interface, where **you can track running services and the number of available instances: http://localhost:8761**



**Load Balancer, Circuit Breaker, and Http Client**

Netflix OSS provides another great set of tools.

**Ribbon**

Ribbon is **a client side load balancer which gives you a lot of control over the behavior of HTTP and TCP clients**. Compared to a traditional load balancer**, there is no need of an additional hop for every over-the-wire invocation — you can contact the desired service directly.**

Out of the box**, it natively integrates with Spring Cloud and Service Discovery.**[**Eureka Client**](https://github.com/sqshq/PiggyMetrics#service-discovery)**provides a dynamic list of available servers so Ribbon could balance between them**.

**Hystrix**

Hystrix is the implementation of a [Circuit Breaker pattern](http://martinfowler.com/bliki/CircuitBreaker.html), which gives a control over latency and failure from dependencies accessed over the network. The main idea is to **stop cascading failures in a distributed environment with a large number of microservices**. That **helps to fail fast and recover as soon as possible** — important aspects of fault-tolerant systems that self-heal.

Besides circuit breaker control, with **Hystrix you can add a fallback method that will be called to obtain a default value in case the main command fails.**

Moreover, **Hystrix generates metrics on execution outcomes and latency for each command, that we can use to**[**monitor system behavior**](https://github.com/sqshq/PiggyMetrics#monitor-dashboard).

**Feign**

Feign is **a declarative HTTP client, which seamlessly integrates with Ribbon and Hystrix**. Actually, with **one spring-cloud-starter-feign dependency** and **@EnableFeignClients annotation** you have a full suite of a **load balancer, circuit breaker, and HTTP client with a sensible ready-to-go default configuration**.

Here is an example from Account Service:

@FeignClient(name = "statistics-service")

public interface StatisticsServiceClient {

@RequestMapping(method = RequestMethod.PUT, value = "/statistics/{accountName}", consumes = MediaType.APPLICATION\_JSON\_UTF8\_VALUE)

void updateStatistics(@PathVariable("accountName") String accountName, Account account);

}

* Everything you need is just an ***interface***
* You can share **@RequestMapping part between Spring MVC controller and Feign methods**
* Above example specifies just desired **service id - statistics-service, thanks to autodiscovery through Eureka (but obviously you can access any resource with a specific url)**

**Monitor Dashboard**

In this project configuration, **each microservice with Hystrix on board pushes metrics to Turbine via Spring Cloud Bus** (with AMQP broker). The Monitoring project is just a small Spring boot application with [Turbine](https://github.com/Netflix/Turbine) and [Hystrix Dashboard](https://github.com/Netflix/Hystrix/tree/master/hystrix-dashboard).

Let's see our system behavior under load: **Account service calls Statistics service and it responses with a vary imitation delay**. **Response timeout threshold is set to 1 second.**

**Log Analysis**

**Centralized logging can be very useful when attempting to identify problems in a distributed environment. Elasticsearch, Logstash, and Kibana stack lets you search and analyze your logs**, utilization and network activity data with ease. Ready-to-go Docker configuration is described [in my other project](http://github.com/sqshq/ELK-docker).

**Security**

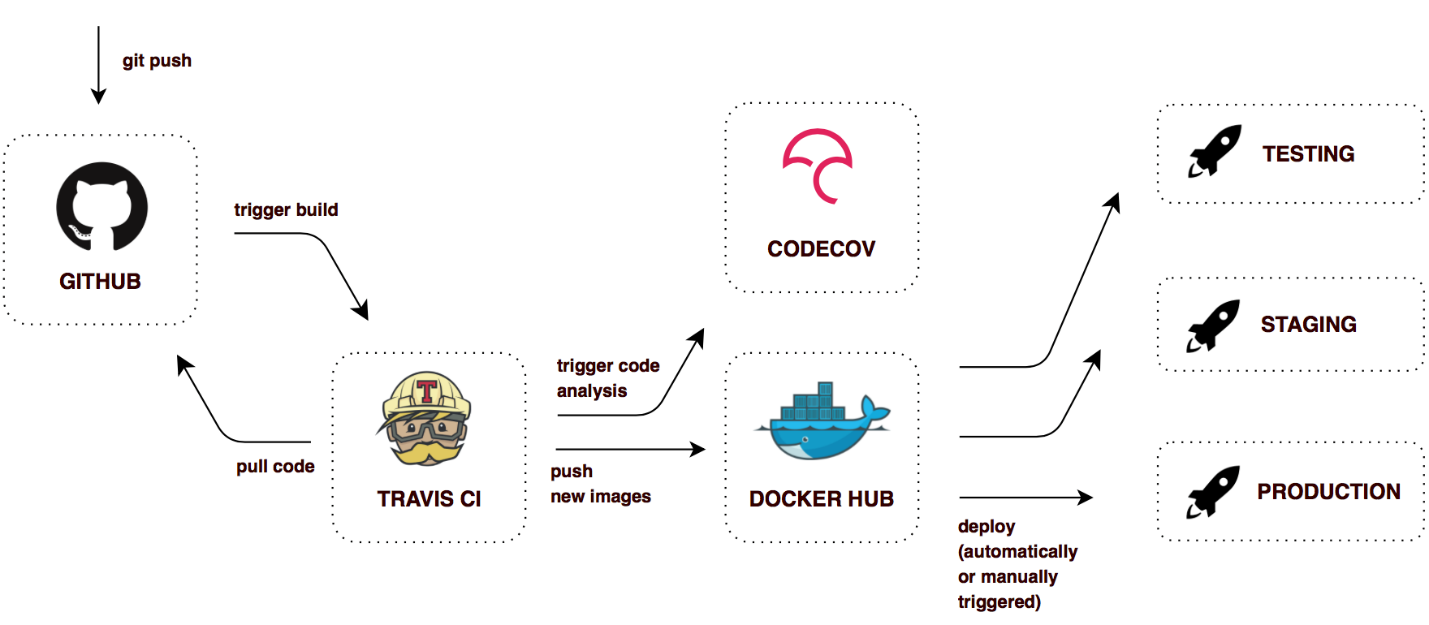
An advanced security configuration is beyond the scope of this proof-of-concept project. For a more realistic simulation of a real system, **consider using https and JCE keystore to encrypt microservices passwords and Config server properties content** (see [documentation](http://cloud.spring.io/spring-cloud-config/spring-cloud-config.html#_security) for details).

**Infrastructure Automation**

Deploying microservices, with their interdependence, is **a much more complex process than deploying a monolith application**. It is important to have a fully automated infrastructure. We can achieve following benefits with a Continuous Delivery approach:

* The ability to **release software anytime.**
* Any build **could end up being a release.**
* Build **artifacts once, deploy as needed.**

Here is a simple Continuous Delivery workflow, implemented in this project:



In this configuration, **Travis CI builds tagged images for each successful Git push.** So there is  always a **latest image for each microservice on**[**Docker Hub**](https://hub.docker.com/r/sqshq/)**and older images are tagged with Git commit hash**. It's easy to deploy any of them and quickly roll back, if needed.

**How to Run All the Things?**

It's really easy and I suggest you to try. Keep in mind, that you are going to start 8 Spring Boot applications, 4 MongoDB instances, and RabbitMq. Make sure you have 4 Gb RAM available on your machine. You can always run just vital services though Gateway, Registry, Config, Auth Service, and Account Service.

**Before You Start**

* Install Docker and Docker Compose.
* Export environment variables: CONFIG\_SERVICE\_PASSWORD, NOTIFICATION\_SERVICE\_PASSWORD, STATISTICS\_SERVICE\_PASSWORD, ACCOUNT\_SERVICE\_PASSWORD, MONGODB\_PASSWORD

**Production Mode**

In this mode, all of the latest images will be pulled from Docker Hub. Just copy docker-compose.yml and hit docker-compose up -d.

**Development Mode**

If you'd like to build images yourself (with some changes in the code, for example), you have to clone all repository and build artifacts with Maven. Then, run docker-compose -f docker-compose.yml -f docker-compose.dev.yml up -d

docker-compose.dev.yml inherits docker-compose.yml with additional possibility to build images locally and expose all containers ports for convenient development.

**Important Endpoints**

* localhost:80 - Gateway
* localhost:8761 - Eureka Dashboard
* localhost:9000 - Hystrix Dashboard
* localhost:8989 - Turbine stream (source for Hystrix Dashboard)
* localhost:15672 - RabbitMq management

**Notes**

**All Spring Boot applications require already running [Config Server](https://github.com/sqshq/PiggyMetrics" \l "config-service) for startup.** But we can start **all containers simultaneously because of fail-fast Spring Boot property and restart: always docker-compose option.**

**That means all *dependent containers* will try to restart until Config Server will be up and running.**

Also, **Service Discovery mechanism needs some time after all applications startup**. Any service is not available for discovery by clients ***until the instance, the Eureka server and the client all have the same metadata in their local cache, so it could take 3 hearbeats***. Default **hearbeat period is 30 seconds**.

<https://dzone.com/articles/microservices-with-spring-boot-part-2-creating-a-f>

<https://www.optisolbusiness.com/insight/micro-services-architecture-spring-boot-and-netflix-infrastructure>

https://medium.com/@marcus.eisele/microservices-with-mo-part-two-the-architecture-3845b5228ddb