



[New Contest] Contributors, Enter the "At Your (Micro) Service" Contest

[Read Rules](#) ▶

Microservices With Spring Boot - Part 5 - Using Eureka Naming Server

by Ranga Karanam 🐙 MVB · Jan. 29, 18 · Microservices Zone

Learn the Benefits and Principles of Microservices Architecture for the Enterprise

In this series, let's learn the basics of microservices and microservices architectures. We will also start looking at a basic implementation of a microservice with Spring Boot. 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.

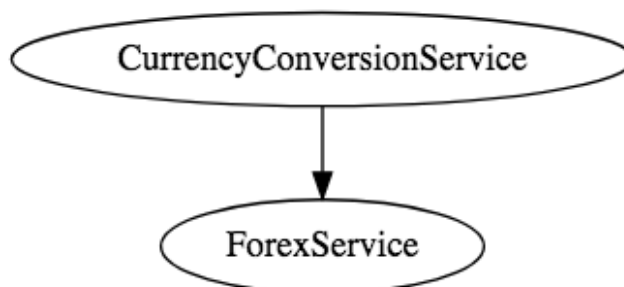
This is part 5 of this series. In this part, we will focus on enabling Eureka Naming Server and have the microservices communicate with it.

You will learn:

- What is the need for a Naming Server?
- What is Eureka?
- How does a Naming Server enable location transparency between microservices?

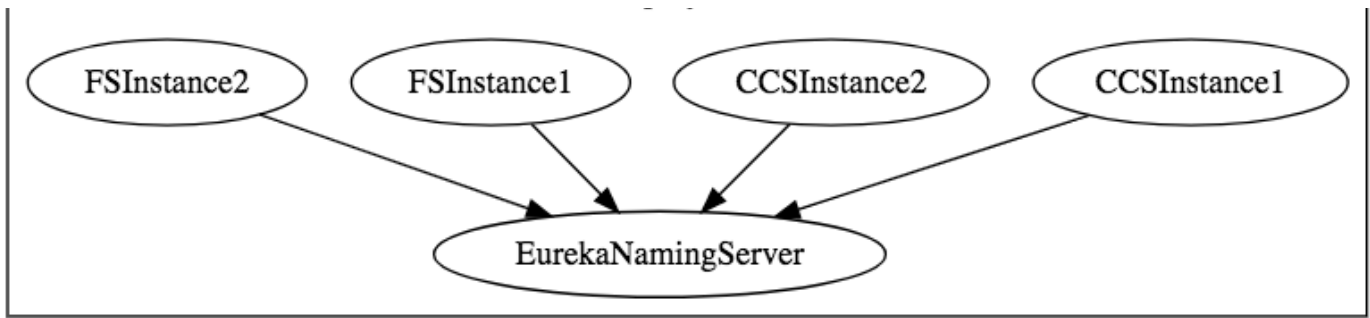
Microservices Overview

In Parts 2 and 3, we created two microservices and established communication between them.



In Part 4, we used Ribbon to distribute load between the two instances of Forex Service. However, we are hardcoding the URLs of both instances of the Forex Service in CCS. That means that every time there is a new instance of FS, we would need to change the configuration of CCS. That's not cool.

In this part, we will use Eureka Naming Server to fix this problem.



Tools you will need:

- Maven 3.0+ is your build tool
- Your favorite IDE. We use Eclipse.
- JDK 1.8+

Complete Maven Project With Code Examples

Our GitHub repository has all the code examples.

Bootstrapping Eureka Naming Server With Spring Initializr

Creating a Eureka Naming Server with Spring Initializr is a cake walk. Spring Initializr is a great tool for bootstrapping your Spring Boot projects. You can create a wide variety of projects using Spring Initializr.

The following steps have to be done for a web services project:

- Launch Spring Initializr and choose the following:
 - Choose `com.in28minutes.springboot.microservice.eureka.naming.server` as Group
 - Choose `spring-boot-microservice-eureka-naming-server` as Artifact
 - Choose following dependencies
- Click Generate Project.
- Import the project into Eclipse. File -> Import -> Existing Maven Project.
- **Do not forget** to choose Eureka in the dependencies.

Enabling Eureka

EnableEurekaServer in `SpringBootMicroserviceEurekaNamingServerApplication`.

```
1 @SpringBootApplication
2 @EnableEurekaServer
3 public class SpringBootMicroserviceEurekaNamingServerApplication {
```

Configure the application name and port for the Eureka Server:

/spring-boot-microservice-eureka-naming-

server/src/main/resources/application.properties

```
1 spring.application.name=netflix-eureka-naming-server
2 server.port=8761
3
4 eureka.client.register-with-eureka=false
5 eureka.client.fetch-registry=false
```

Launching Eureka Naming Server

Launch `SpringBootMicroserviceEurekaNamingServerApplication` as a Java application.

You can launch Eureka at `http://localhost:8761`

You will see that there are no instances connected to Eureka yet:

The screenshot shows the Eureka Naming Server web interface. The browser tabs include 'localhost:8100/currency-convi...' and 'Eureka'. The address bar shows 'localhost:8761'. The main content area is divided into three sections:

- System Status**: A table with two columns. The left column contains 'Environment' (test) and 'Data center' (default). The right column contains 'Current time' (2017-12-12T12:48:04 +0530), 'Uptime' (00:13), 'Lease expiration enabled' (true), 'Renews threshold' (5), and 'Renews (last min)' (8).
- DS Replicas**: A single entry 'localhost'.
- Instances currently registered with Eureka**: A table with columns 'Application', 'AMIs', 'Availability Zones', and 'Status'. The table is currently empty.

Connect FS and CCS Microservices With Eureka

Make these changes on both the microservices

Add to pom.xml:

```
1 <dependency>
2   <groupId>org.springframework.cloud</groupId>
3   <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
4 </dependency>
```

Configure Eureka URL in application.properties:

```
1 eureka.client.service-url.default-zone=http://localhost:8761/eureka
```

Restart all the instances of CCS and FS. You will see that the CCS and FS microservices are registered with Eureka Naming Server. That's cool!

The screenshot shows the Eureka Naming Server web interface. The browser tabs include 'localhost:8100/currency-convi...' and 'Eureka'. The address bar shows 'localhost:8761'. The main content area is divided into three sections:

- System Status**: A table with two columns. The left column contains 'Environment' (test) and 'Data center' (default). The right column contains 'Current time' (2017-12-12T12:48:04 +0530), 'Uptime' (00:13), 'Lease expiration enabled' (true), 'Renews threshold' (5), and 'Renews (last min)' (8).
- DS Replicas**: A single entry 'localhost'.
- Instances currently registered with Eureka**: A table with columns 'Application', 'AMIs', 'Availability Zones', and 'Status'. The table is currently empty.

System Status

Environment	test	Current time	2017-12-12T12:48:04 +0530
Data center	default	Uptime	00:13
		Lease expiration enabled	true
		Renews threshold	5
		Renews (last min)	8

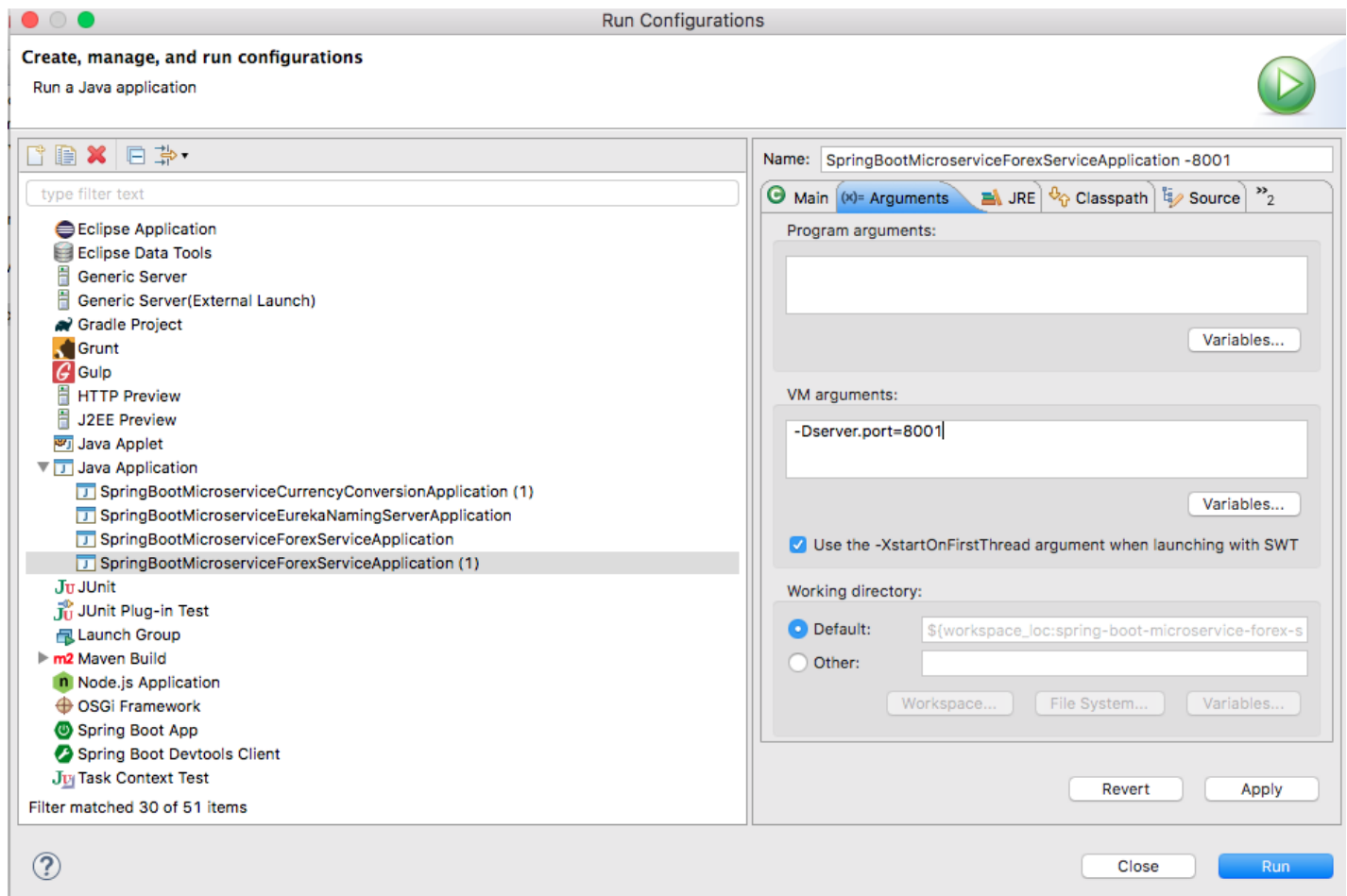
DS Replicas

localhost

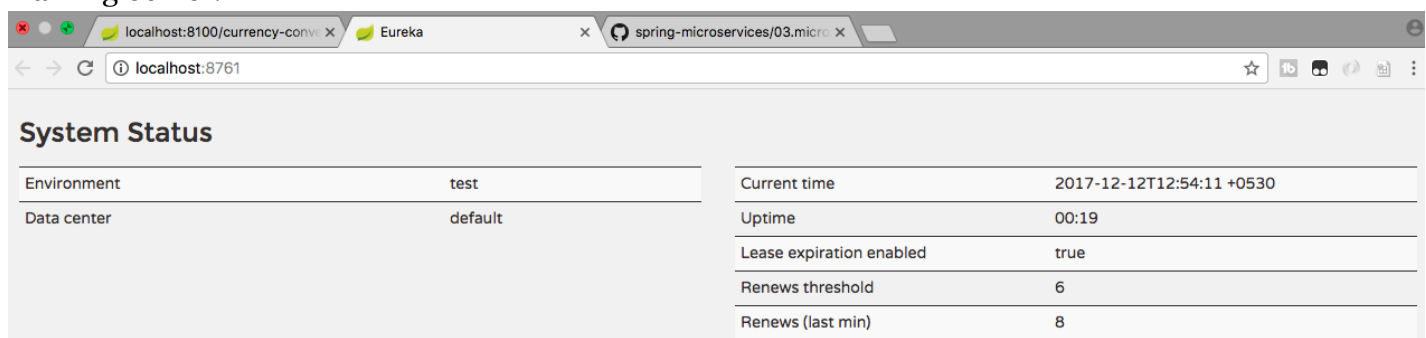
Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CURRENCY-CONVERSION-SERVICE	n/a (1)	(1)	UP (1) - 192.168.12.133:currency-conversion-service:8100
FOREX-SERVICE	n/a (1)	(1)	UP (1) - 192.168.12.133:forex-service:8000

This screenshot shows how to launch an additional instance of Forex Service on 8081.



You will see that one instance of CCS and two instances of FS microservices are registered with Eureka Naming Server.



DS Replicas

localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CURRENCY-CONVERSION-SERVICE	n/a (1)	(1)	UP (1) - 192.168.12.133:currency-conversion-service:8100
FOREX-SERVICE	n/a (2)	(2)	UP (2) - 192.168.12.133:forex-service:8001 , 192.168.12.133:forex-service:8000

Routing Ribbon Requests Through Eureka

All that you would need to do is to remove this configuration. Remove this configuration from application.properties:

```
1 forex-service.ribbon.listOfServers=localhost:8000,localhost:8001
```

Restart the CCS instance.

Eureka in Action

Currently, we have the following services up and running:

- Currency Conversion Microservice (CCS) on 8100
- Two instances of Forex Microservice on 8000 and 8001
- Eureka Server launched

Now you will see that the requests to CCS will be distributed between the two instances of the Forex Microservice by Ribbon through Eureka.

Request 1

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

```
1 {
2   id: 10002,
3   from: "EUR",
4   to: "INR",
5   conversionMultiple: 75,
6   quantity: 10000,
7   totalCalculatedAmount: 750000,
8   port: 8000,
9 }
```

Request 2

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

```
1 {
2   id: 10002,
3   from: "EUR",
```

```
4    to: "INR",
5    conversionMultiple: 75,
6    quantity: 10000,
7    totalCalculatedAmount: 750000,
8    port: 8001,
9  }
```

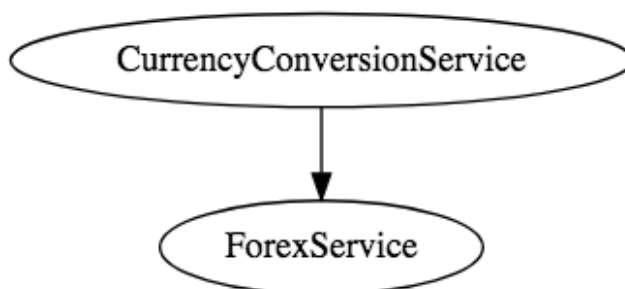
You can see that the port numbers in the two responses are different.

Exercise: Launch another instance of Forex Service on 8002. You will see that load gets automatically routed to it as well.

Cool! That's awesome, isn't it?

Summary

We have now created two microservices and established communication between them.



We are using Ribbon to distribute load between the two instances of Forex Service and Eureka as the naming server. When we launch new instances of Forex Service, you will see that load is automatically distributed to them.

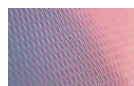
The idea behind this series of five articles was to give the flavor of Spring Boot and Spring Cloud with microservices.

There is a lot more ground to cover with microservices. Until next time, cheers!

The complete code example for this project can be found in the GitHub repository.

Microservices for the Enterprise eBook: Get Your Copy Here

Like This Article? Read More From DZone



Using Netflix Eureka With Spring Cloud/Spring Boot Microservices (Part 1)



Microservices With Spring Boot - Part 3 - Creating Currency Conversion Microservice




Microservices With Spring Boot - Part 2 - Creating a Forex



**Free DZone Refcard
Microservices in Java**

Microservice

Topics: MICROSERVICES , EUREKA , SPRING BOOT , TUTORIAL

Published at DZone with permission of Ranga Karanam, DZone MVB. [See the original article here.](#) 
Opinions expressed by DZone contributors are their own.

Microservices Partner Resources

Learn the Benefits and Principles of Microservices Architecture

CA Technologies



Microservices Architecture eBook: Download Your Copy Here

CA Technologies



The Microservices Paradox

by Dirk Louwers  MVB · Feb 02, 18 · Microservices Zone

Gianna has joined Avidoo Inc., a productivity platform, as a senior software engineer. In a kick-off meeting with the rest of her team, she brings up the subject of microservices and whether the team has adopted them in any way. She immediately gets a strong reaction.

"We have tried adopting microservices, but they don't work," Byron offers.

"It became a terrible mess!" Kary adds.

Gianna blinked her eyes three times expecting some kind of elaboration, but none followed. After an uncomfortable silence, Gianna asks: "So what happened?"

"At first it was great. Every time we were asked to create something new, we had the opportunity to add a service and use whatever languages and frameworks we wanted to experiment with. We exposed REST APIs on systems it needed to collaborate with or worked on their databases directly. But after a while, things started to break more and more often, and development slowed to a crawl."

Gianna sighs. It sounds to her like her team had been building a distributed monolith, while what they had meant to build were microservices.

Distributed Monoliths and Other Monstrosities

What Gianna has run into at Avidoo Inc., which is, of course, a fictional company, is unfortunately all too common. Drawn by the idea that microservices are a panacea, IT managers and engineers tend to skip identifying what advantages are a good fit with their organization.

People forget that there is no such thing as a free lunch. As well as advantages, well-built microservices architectures have tradeoffs. There are no "wrong" microservices, only microservices that do not deliver the advantages they were built for or pose unacceptable risks through their disadvantages.

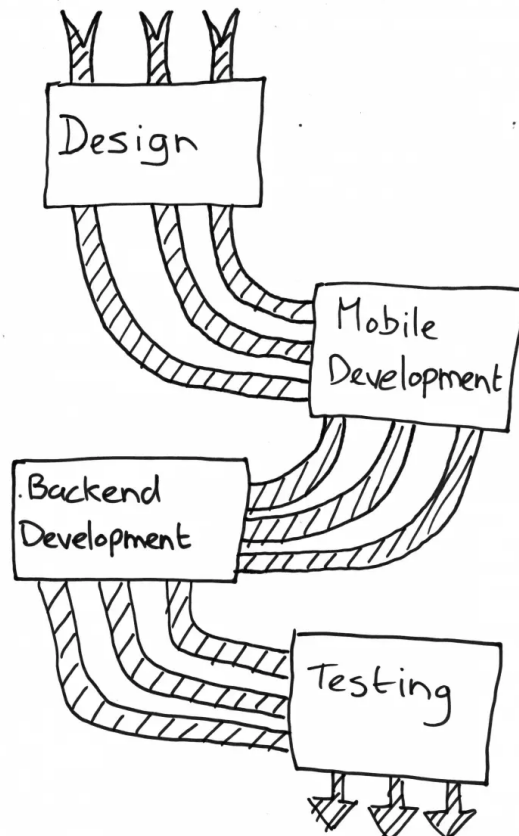
Advantages of Using Microservices

Advantages of Using Microservices

Choosing to adopt microservices should start with deciding which of its advantages are a good fit for your organization. Below are some of those advantages.

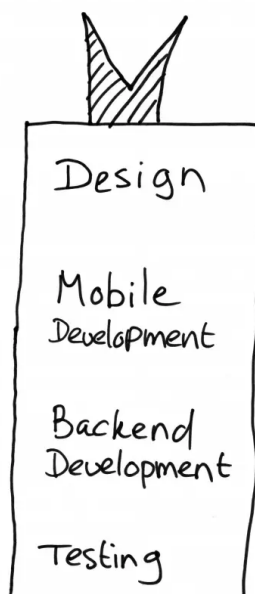
Increased Team Autonomy

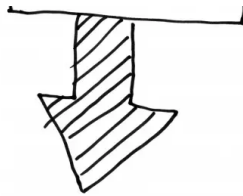
Many companies organize teams around their member's discipline or components. When creating real customer value, this asks for a lot of coordination between teams and makes it next to impossible to work on one feature in parallel.



Creating value with single-disciplinary teams.

Microservices facilitate autonomy by covering one feature. Therefore one team can fully own it instead of multiple teams. This helps reduce cross-team coordination.

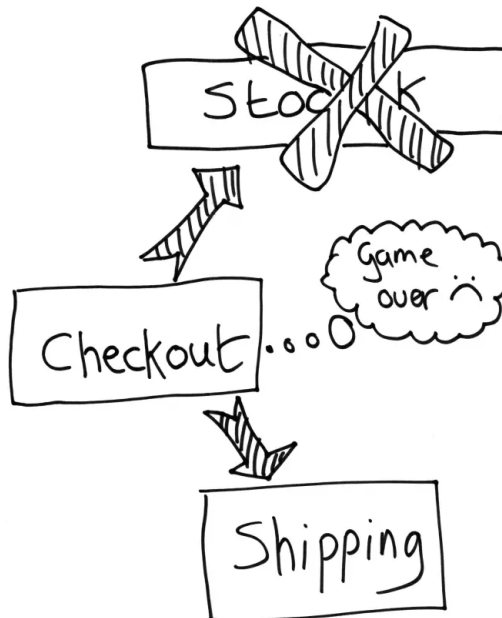




Creating value with a multi-disciplinary team.

Greater Fault Tolerance

Where there is autonomy of a team there should also be the autonomy of a feature. Features often depend on one another. In most environments, communication is on-demand and pull-based, often over a REST interface. When this interaction is mission critical, the service depending on this communication either has to have a sensible fall-back or it will, in turn, fail. This unhealthy pattern is often illustrated by system health checks that fail when one of its dependencies is unhealthy. Aside from causing deployment order to be difficult to manage, it illustrates hard system dependency.

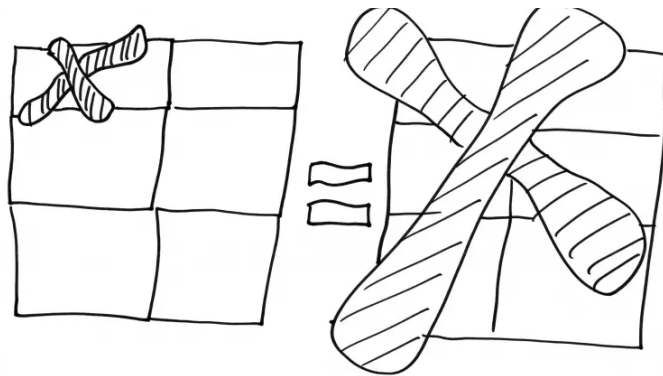


Run-time dependencies.

Using the right software architectures like event sourcing, possibly complemented by CQRS, it is possible to erase run-time dependency between most features completely. This is mainly due to the transition from a pull-based system to a push-based one.

Granular Software Lifecycle Management

A common wish is to replace a certain feature within an application with a shiny new one. This is because, either the requirements have diverged so far as to warrant a rewrite or development speed has been run into the ground by technical debt contracted by strenuous time-to-market demands. It might be naively thought that these should be replaceable as fast as it took to write them in the first place, but this mostly proves to be untrue. All too often replacing one feature results in having to make changes to many systems it depends on or vice versa.

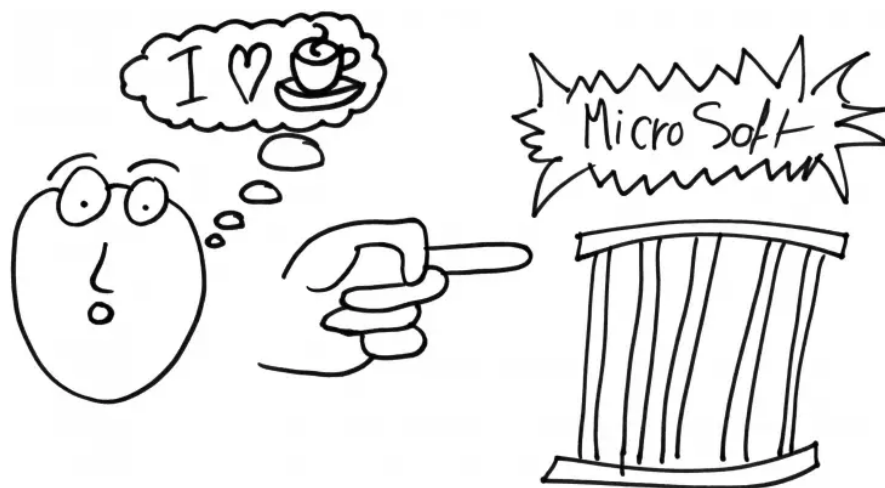


Lack of granular software lifecycle management.

By highly regulating inter-system communications, it is possible to switch out one or more features completely without having to touch any of its dependent systems.

Flexible Technology Choices

Admittedly this is a tricky one. Onboarding and training people to switch to a common technology helps with inter-team mobility, driven by demand or personal interest. But reshuffling staff from several departments using different technology stacks that, frankly, they are quite religious about, can result in mass walk-outs.



Forcing technology choices.

As long as the technologies can be integrated into your automated testing and deployment workflow, a team's technology choices can remain their own. Why change a winning team that's united around their love for everything C# as long as they produce an artifact that adheres to your platform's monitoring, logging and communication rules?

So Why Do They Fail?

There isn't one way to do microservices. Adopting microservices doesn't fail because people don't know how to do them but rather because they don't remember what problems they were solving in the first place. As with any other decision, adopting certain aspects of microservices comes with a cost. Software architects tend to forget that they shouldn't be helping their employer adopt microservices but should help them solve real business problems. Properly weighing the costs and benefits of these aspects against an organization's need is of vital importance. None the less, there is a default set of choices that

can form a sensible inception from which to start this bold adventure.

Like This Article? Read More From DZone



Low-Risk Monolith to Microservice Evolution: Part II



Separating Microservices Hype and Reality for Pragmatic Java Developers




Are You Building Microservices or Microliths?



**Free DZone Refcard
Microservices in Java**

Topics: MICROSERVICES, MONOLITH, DISTRIBUTED SYSTEMS, SOFTWARE ARCHITECTURE

Published at DZone with permission of Dirk Louwers, DZone MVB. [See the original article here.](#) 
Opinions expressed by DZone contributors are their own.
