Sommario

[Title: Full lifecycle of a microservice: how to realize a fault-tolerant and reliablearchitecture and deliver it as a Docker container or in a Cloud environment 2](#_Toc456695310)

[1.1 Microservices: implementing the “database per service” pattern 4](#_Toc456695311)

[1.1.1 Pattern: Database per service 4](#_Toc456695312)

[1.1.2 Backing Service integration and configuration 8](#_Toc456695313)

[1.1.3 Implementing the spring profile pattern 8](#_Toc456695314)

[1.1.4 How to create a backing services in Pivotal Cloud Foundry 9](#_Toc456695315)

[1.1.5 How to bind a backing services (PAAS CONNECTOR/JAVA CONFIGURATION) 10](#_Toc456695316)

[1.1.6 Define an automated build (Jenkins@Openshift) and promote the docker image in container registry (DockerHub) 10](#_Toc456695317)

[1.1.7 NEW transaction behaviuor 10](#_Toc456695318)

[1.2 Interactions between Microservices: service discovery and service registration 15](#_Toc456695319)

[1.2.1 Eureka service registry facilities 15](#_Toc456695320)

[1.2.2 How to implement a microservice and register inside Eureka with Spring Cloud 15](#_Toc456695321)

[1.2.3 Implementing a client application that consume an Eureka server application with Ribbon client side load balancing features 15](#_Toc456695322)

[1.3 Solution delivery 15](#_Toc456695323)

[1.3.1 In a Cloud Environment: Pivotal Cloud Foundry 15](#_Toc456695324)

[1.3.2 As a docker container (Nota: deprecate boot2docker with docker machine) 15](#_Toc456695325)

OREILLY

# Full lifecycle of a microservice: how to realize a fault-tolerant and reliablearchitecture and deliver it as a Docker container or in a Cloud environment

**Abstract**: How to realize, and containerize in a Docker image, a "database per service" REST microservice, using Spring Cloud to simplify the complexity of Eureka service registry and Ribbon client side load balancing. Then its lifecycle will be orchestrated till delivery with Paas Cloud environments: Github, Jenkins@Openshift, DockerHub and Pivotal Cloud Foundry.

**Fast Presentation of myself**

Hello, thank you for coming, my name is luigi bennardis and I come from Rome –Italy. First of all I would like to thanks Oreily for having invited me in such an important event.

Just something about me about

MASTER DEGREE in statistcs and economics at the university of rome then twenty year of experience in the field of Information Technology ( seniority ) as a developer than as architect and finally as a system integrator and technical project leader.

Actually for about 6 years Software Configuration Manager - responsible of the design implementation diffusion and management of the ALm platform of an important Italian Company. Platform based both on market and open source tools like GIT Jenkins/ Microsoft TFS and IBM Jazz.

What about this talk: I will try to describe the end to end process of delivery of a microservice from design pattern till delivery

**Fast Presentation of this work**

It will be shown, the most relevant steps , the full lifecycle development process of a microservice. From architectural or a design pattern (database per service) and technological (Spring Boot) aspects to delivery related scenarios (development, Cloud or dockerized environments), in an ecosystem context where microservices are each other reliable and fault tolerant (Eureka service registry, Ribbon load balancing, Spring Cloud).

Making a focus on the design/framework/technologies/infrastructure

Domain-driven design. Continuous Delivery. On-demand virtualization. Infrastructure automation. Small autonomous teams. DevOps. Microservices have emerged from this world: they weren't invented or described before the fact. They emerged as a trend, or a pattern, from real-world use, starting from all that stuff. With Microservices architecture, the use of small developer teams becomes a reality as it is much more feasible to assign smaller independent team of developers compared to building a large monolithic application. Similarly, the responsibility in the IT operations team can be assigned to a smaller group, building a foundation for stronger and deeper collaboration between the developers and IT Ops. Such a close collaboration is at the heart of DevOps and Microservices can go a long way to seamlessly enable such a collaboration. Containers (see docker) offer the right abstraction to encapsulate Microservices. A PaaS offering that takes advantage of containers offers the right standardization to streamline deployment pipelines, maximizing DevOps benefits. When this is combined with the collaboration advantage enjoyed by the small teams building and deploying Microservices, you are in a position to achieve DevOps nirvana. Throughout this talk, we will try to paint a picture of how to design, build, manage, and deploy microservices. And, remember: do one thing and do it well.

focusing which will be the technology choises that will fullfill each phase in terms of development framework and infrastructure

|  |  |  |  |
| --- | --- | --- | --- |
| Phase | Arch. pattern | Technology FRAMEWORK | INFRASTRUCTURE |
| Design | DATABASE PER SERVICE | ? Event sourcing |  |
| Development | Microservices | Spring Boot |  |
| Configuration | Spring Configuration |  |
| Cloud based architecture | Spring cloud | DBAS A SERVICE & JENKINS@OPENSHIFT |
| SERVICE REGISTRY | EUREKA | NA |
| LOAD BALANCING | RIBBON | NA |
| Deliver |  | LOCAL |  |
|  | Integration/test | Docker |
|  | DOCKER | DOCKER HUB |
|  | Cloud paas | Pivotal web service |

Details of the framework / infrastructure involved and relative phase

|  |  |
| --- | --- |
|  |  |
|  |  |
| Spring Boot | JPA - |
| Spring cloud |  |
| EUREKA |  |
| RIBBON |  |
| LOCAL |  |
| Integration/test |  |
| DOCKER |  |
| Cloud paas |  |

|  |
| --- |
| DBAS A SERVICE & JENKINS@OPENSHIFT |
|  |
|  |
|  |
| Docker |
| DOCKER HUB |
| Pivotal web service |

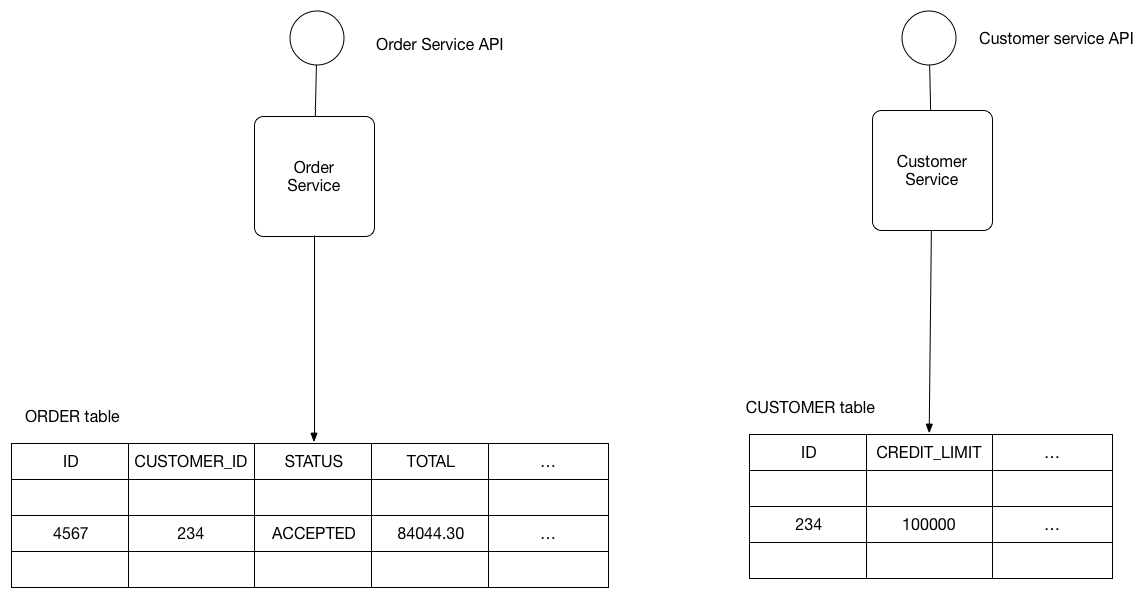
## Microservices: implementing the “database per service” pattern

### Pattern: Database per service

#### Context

<http://www.scribd.com/doc/2569355/Geo-Distance-Search-with-MySQL>

Let’s imagine you are developing an online store application using the [Microservices pattern](http://microservices.io/patterns/microservices.html). Most services need to persist data in some kind of database. For example, theOrder Service stores information about orders and the Customer Service stores information about customers.



#### Problem

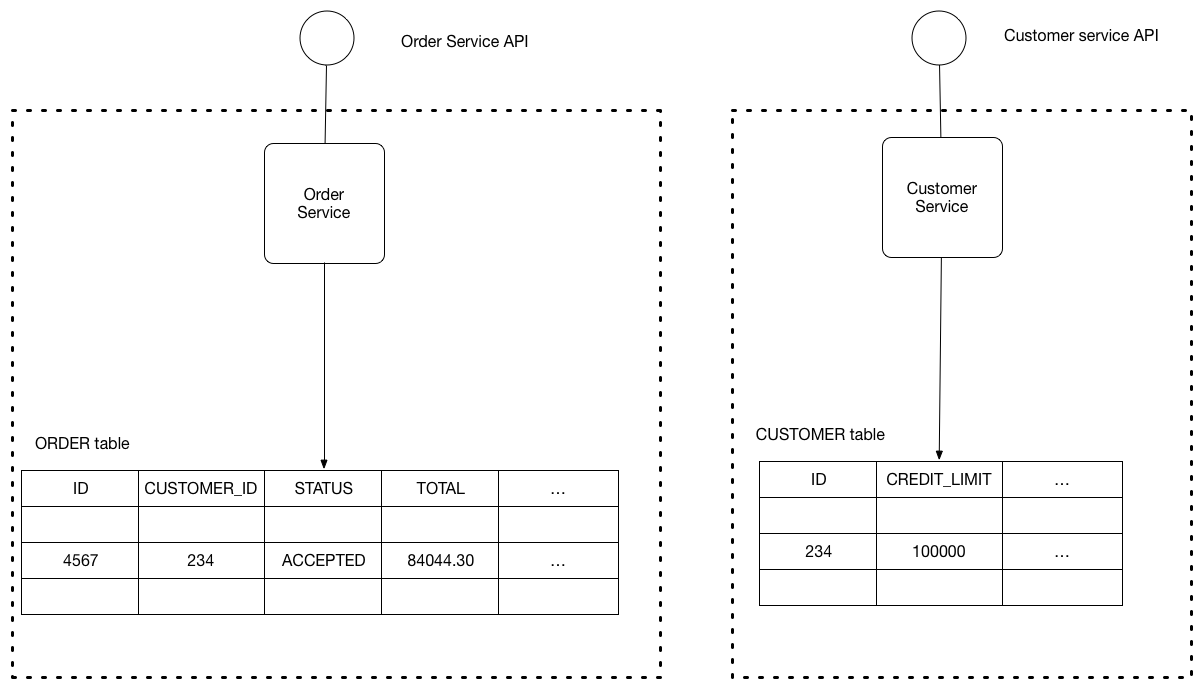
What’s the database architecture in a microservices application?

#### Forces

* Services must be loosely coupled so that they can be developed, deployed and scaled independently
* Some business transactions need to update data that is owned by multiple services. For example, the Place Order use case updates the Customer to reserve credit for the Order and creates an Order.
* Some queries must join data that is owned by multiple services. For example, finding customers in a particular region and their recent orders requires a join between customers and orders.
* Databases must sometimes be replicated and sharded in order to scale. See the [Scale Cube](http://microservices.io/articles/scalecube.html).
* Different services have different data storage requirements. For some services, a relational database is the best choice. Other services might need a NoSQL database such as MongoDB, which is good at storing complex, unstructured data, or Neo4J, which is designed to efficiently store and query graph data.

#### Solution

Keep each microservice’s persistent data private to that service and accessible only via its API. The following diagram shows the structure of this pattern.



The service’s database is effectively part of the implementation of that service. It cannot be accessed directly by other services.

There are a few different ways to keep a service’s persistent data private. You do not need to provision a database server for each service. For example, if you are using a relational database then the options are:

* Private-tables-per-service – each service owns a set of tables that must only be accessed by that service
* Schema-per-service – each service has a database schema that’s private to that service
* Database-server-per-service – each service has it’s own database server.

Private-tables-per-service and schema-per-service have the lowest overhead. Using a schema per service is appealing since it makes ownership clearer. Some high throughput services might need their own database server.

It is a good idea to create barriers that enforce this modularity. You could, for example, assign a different database user id to each service and use a database access control mechanism such as grants. Without some kind of barrier to enforce encapsulation, developers will always be tempted to bypass a service’s API and access it’s data directly.

#### Resulting context

Using a database per service has the following benefits:

* Helps ensure that the services are loosely coupled. Changes to one service’s database does not impact any other services.
* Each service can use the type of database that is best suited to its needs. For example, a service that does text searches could use ElasticSearch. A service that manipulates a social graph could use Neo4j.

Using a database per service has the following drawbacks:

* Implementing business transactions that span multiple services is not straightforward. Distributed transactions are best avoided because of the CAP theorem. Moreover, many modern (NoSQL) databases don’t support them. The best solution is to use an [eventually consistent, event-driven architecture](http://microservices.io/patterns/data/event-driven-architecture.html). Services publish events when they update data. Other services subscribe to events and update their data in response.
* Implementing queries that join data that is now in multiple databases is challenging. There are various solutions:
* Application-side joins - the application performs the join rather than the database. For example, a service (or the API gateway) could retrieve a customer and their orders by first retrieving the customer from the customer service and then querying the order service to return the customer’s most recent orders.
* Command Query Responsibility Segregation (CQRS) - maintain one or more materialized views that contain data from multiple services. The views are kept by services that subscribe to events that each services publishes when it updates its data. For example, the online store could implement a query that finds customers in a particular region and their recent orders by maintaining a view that joins customers and orders. The view is updated by a service that subscribes to customer and order events.
* Complexity of managing multiple SQL and NoSQL databases

**Describe base approach vs event sourcing**

**Event sourcing and command query segregation pattern**

[**https://github.com/cer/event-sourcing-examples/wiki/DeveloperGuide**](https://github.com/cer/event-sourcing-examples/wiki/DeveloperGuide)

[**https://github.com/cer/event-sourcing-examples/wiki/WhyEventSourcing**](https://github.com/cer/event-sourcing-examples/wiki/WhyEventSourcing)

[**https://webcache.googleusercontent.com/search?q=cache:4O9SFBtXxowJ:https://plainoldobjects.com/2015/09/02/does-each-microservice-really-need-its-own-database-2/+&cd=2&hl=it&ct=clnk&gl=it**](https://webcache.googleusercontent.com/search?q=cache:4O9SFBtXxowJ:https://plainoldobjects.com/2015/09/02/does-each-microservice-really-need-its-own-database-2/+&cd=2&hl=it&ct=clnk&gl=it)

[**http://www.slideshare.net/chris.e.richardson/microservices-in-java-and-scala-sfscala**](http://www.slideshare.net/chris.e.richardson/microservices-in-java-and-scala-sfscala)

**Event sourcing**

[**https://github.com/cer/event-sourcing-examples**](https://github.com/cer/event-sourcing-examples)

[**https://ookami86.github.io/event-sourcing-in-practice**](https://ookami86.github.io/event-sourcing-in-practice)

**1.2 Implementing a microservice with Spring Boot**

### Backing Service integration and configuration

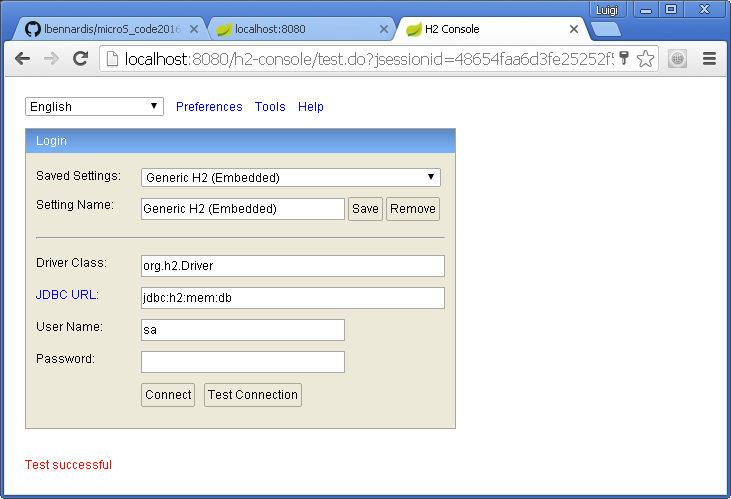
### Implementing the spring profile pattern

application.properties

1. localH2
2. mySqlLocal
3. mySqlDocker
4. dockerContainer (specializzare la connessione jdbc secondo lo standard docker run)
5. cloudFoundry

**DEMO**

1. **Esecuzione localH2 (in memory - local Console)**



1. **Esecuzione mySqlLocal -/Env**
2. **Esecuzione mySqlDocker -/Env**

**PANORAMICA SU DOCKER**

**BOOT2DOCKER**

**DOCKER IMAGES GIA’ SCARICATE**

1. **BUILD maven docker local published -/Env**
2. **Esecuzione dockerContainer**
3. **Promozione sul branch /INTEGRAZIONE del sorgente**

**JENKINS ON OPENSHIFT PANORAMICA**

**ACCESSO A JENKINS CON FILEZILLA SULLA MACCHINA JENKINS**

**BUILD**

**PUBBLICAZIONE E BUILD DEL CONTANER SU DOCKER HUB PER CONDIVIDERE**

### How to create a backing services in Pivotal Cloud Foundry

manifest.yml

cf create-service ClearDb <plan> <name>

env:

SPRING\_PROFILE\_ACTIVE: <profile\_name>

**DEMO**

1. **Esecuzione cloudFoundry /Env**
2. **Accesso al database sul cloud**

### How to bind a backing services (PAAS CONNECTOR/JAVA CONFIGURATION)

**PAAS CONNECTOR**

Definizione datasource

@Configuration

@Profile(“cloudFoundry”)

Mettere log print sulla classe che definisce la configuration

**DEMO**

1. **Esecuzione cloudFoundry /Env**

**JAVA CONFIGURATION**

Definizione datasource

@Bean

@Profile(“cloudFoundry”)

Definizione di un custom connetion pool

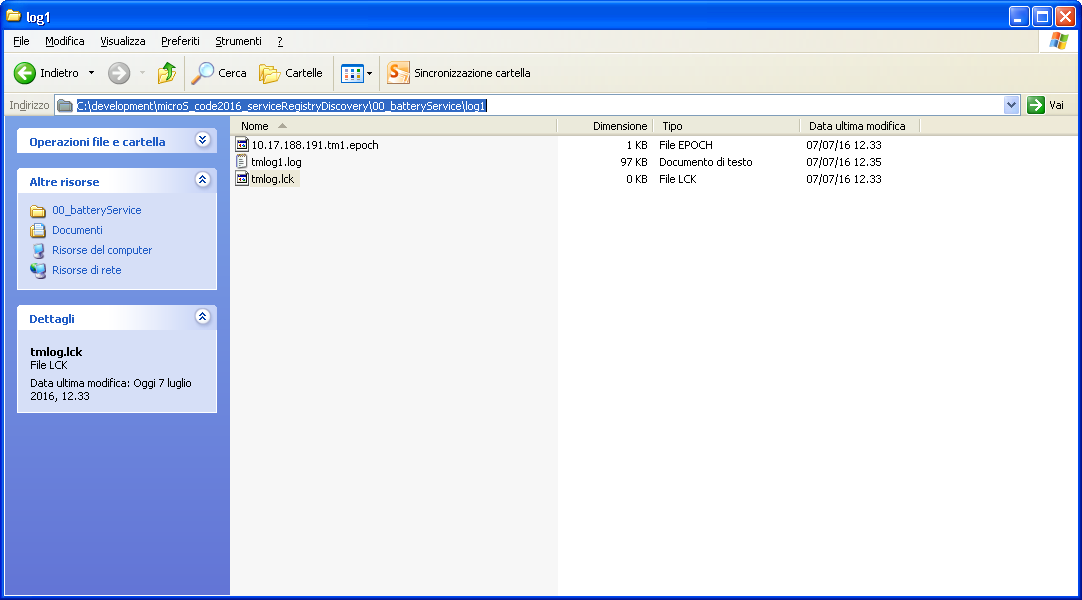
### Define an automated build (Jenkins@Openshift) and promote the docker image in container registry (DockerHub)

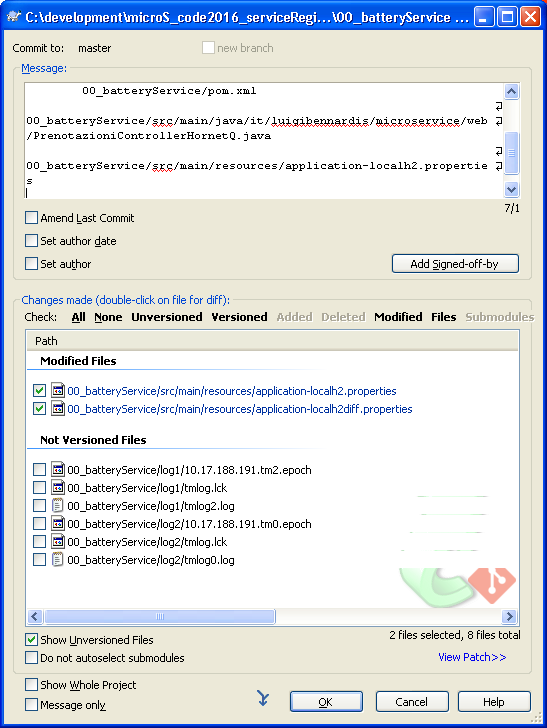
### NEW transaction behaviuor

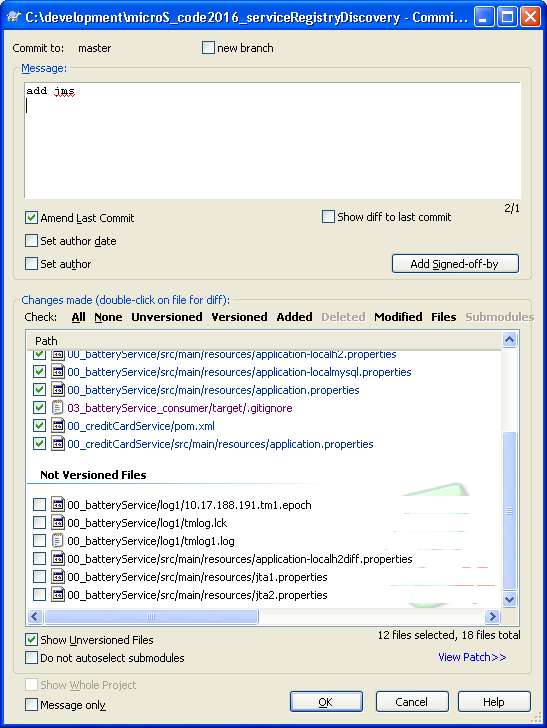
Transactional behavior with atomiko and HornetQ

Atomikos lock file

C:\development\microS\_code2016\_serviceRegistryDiscovery\00\_batteryService\log1







#### Pattern: Event-driven architecture

##### Summary

Maintain data consistency across microservices by exchanging events

##### Problem

You have applied the [Database per Service](http://microservices.io/patterns/data/database-per-service.html) pattern. Each service has its own database. Some business transactions, however, span multiple service so you need a mechanism to ensure data consistency across services.

For example, lets imagine that you are building an e-commerce store where customers have a credit limit. The application must ensure that a new order will not exceed the customer’s credit limit. Since Orders and Customers are in different databases the application cannot simply use a local ACID transaction. In theory, it could use a distributed transaction that spans theCustomer Service and the Order Service. However, for a variety of reasons distributed transactions are not a viable choice for most modern applications.

##### Solution

Use an event-driven, eventually consistent approach. Each service publishes an event whenever it update it’s data. Other service subscribe to events. When an event is received, a service updates it’s data.

##### Resulting context

This pattern has the following benefits:

* It enables an application to maintain data consistency across multiple services without using distributed transactions

This solution has the following drawbacks:

* The programming model is more complex

There are also the following issues to address:

* In order to be reliable, an application must atomically update its database and publish an event. It cannot use the traditional mechanism of a distributed transaction that spans the database and the message broker. Instead, it must use one the patterns listed below.

##### Example

An e-commerce application that uses this approach would work as follows:

1. The Order Service creates an Order in a pending state and publishes an OrderCreated event.
2. The Customer Service receives the event and attempts to reserve credit for that Order. It then publishes either a Credit Reserved event or a CreditLimitExceeded event.
3. The Order Service receives the event from the Customer Service and changes the state of the order to either approved or cancelled

##### Related patterns

* The [Database per Service pattern](http://microservices.io/patterns/data/database-per-service.html) creates the need for this pattern
* The following patterns are ways to atomically update state and publish events:
  + [Event sourcing](http://microservices.io/patterns/data/event-sourcing.html)
  + [Application events](http://microservices.io/patterns/data/application-events.html)
  + [Database triggers](http://microservices.io/patterns/data/database-triggers.html)
  + [Transaction log tailing](http://microservices.io/patterns/data/transaction-log-tailing.html)

See also

The article [Event-Driven Data Management for Microservices](https://www.nginx.com/blog/event-driven-data-management-microservices/) by @crichardson describes this pattern

<https://www.nginx.com/blog/event-driven-data-management-microservices/>

## Interactions between Microservices: service discovery and service registration

### Eureka service registry facilities

### How to implement a microservice and register inside Eureka with Spring Cloud

### Implementing a client application that consume an Eureka server application with Ribbon client side load balancing features

## Solution delivery

### In a Cloud Environment: Pivotal Cloud Foundry

### As a docker container (Nota: deprecate boot2docker with docker machine)

<https://docs.docker.com/engine/installation/windows/>