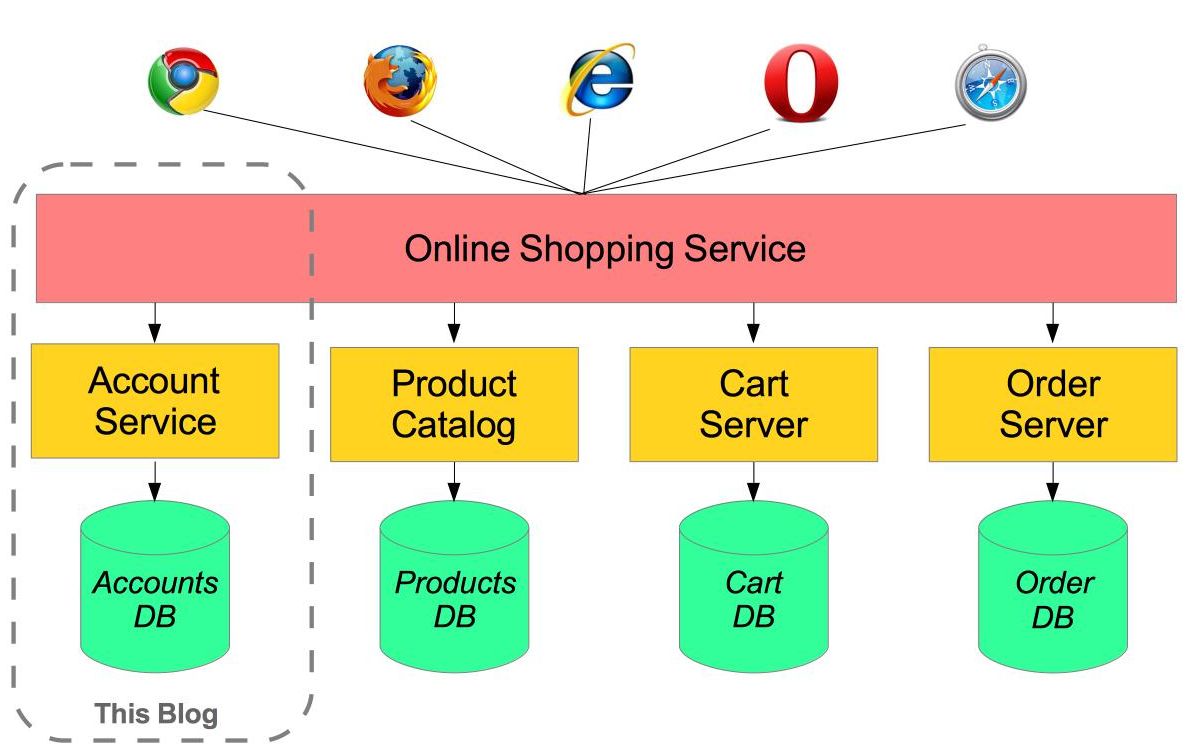
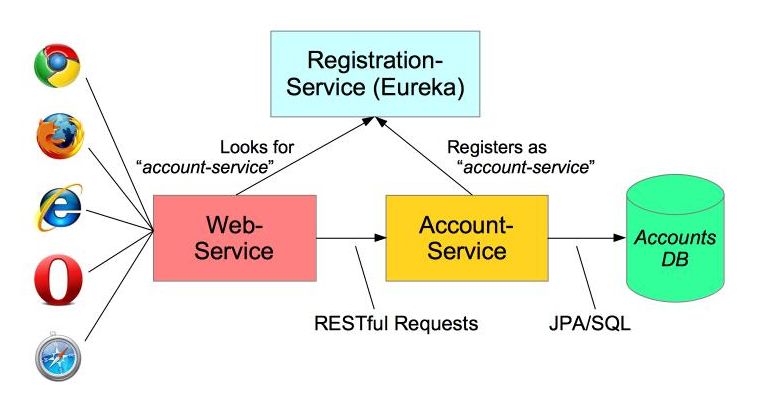
A simple example of setting up a microservices system using Spring, Spring Boot and Spring Cloud.

[Microservices](http://martinfowler.com/articles/microservices.html) allow large systems to be built up from a number of collaborating components. It does at the process level what Spring has always done at the component level: loosely coupled processes instead of loosely coupled components.

[](https://raw.githubusercontent.com/paulc4/microservices-demo/master/shopping-system.jpg)

For example imagine an online shop with separate microservices for user-accounts, product-catalog order-processing and shopping carts:

Inevitably there are a number of moving parts that you have to setup and configure to build such a system. How to get them working together is not obvious - you need to have good familiarity with Spring Boot since Spring Cloud leverages it heavily, several Netflix or other OSS projects are required and, of course, there is some Spring configuration “magic”!

[](https://raw.githubusercontent.com/paulc4/microservices-demo/master/mini-system.jpg)

In this article I aim to clarify how things work by building the simplest possible system step-by-step. Therefore, I will only implement a small part of the big system - the user account service.

The *Web-Application* will make requests to the *Account-Service*microservice using a RESTful API. We will also need to add a *discovery* service – so the other processes can find each other.

The code for this application is here: <https://github.com/paulc4/microservices-demo>.

**Follow-Up 1: Other Resources**

This article only discusses a minimal system. For more information, you might like to read Josh Long’s blog article [Microservice Registration and Discovery with Spring Cloud and Netflix’s Eureka](https://spring.io/blog/2015/01/20/microservice-registration-and-discovery-with-spring-cloud-and-netflix-s-eureka) which shows running a complete microservice system on Cloud Foundry.

The Spring Cloud projects are [here](https://projects.spring.io/spring-cloud/).

**Follow Up 2: SpringOne Platform 2016**

Book your place at [SpringOne2 Platform in Las Vegas, USA soon](https://springoneplatform.io/) - simply the best opportunity to find out first hand all that’s going on and to provide direct feedback. The name has changed, from *Spring One*, to reflect the growth of Spring in platform services (such as the Spring Cloud projects)

**Updates (May 2016)**

A number of changes since I wrote this blog last year:

1. A [discussion](https://spring.io/blog/2015/07/14/microservices-with-spring#configuration-options) of using multiple instances of the same service on the same host.. Demo application updated to match.
2. A [discussion](https://spring.io/blog/2015/07/14/microservices-with-spring#load-balanced-resttemplate) of @LoadBalanced - how this works *has changed* with the *Brixton* release-train ([Spring Cloud](https://projects.spring.io/spring-cloud) 1.1.0.RELEASE).
3. Refactored [configuration](https://spring.io/blog/2015/07/14/microservices-with-spring#accountsconfiguration-class) of Accounts microservice into its own class AccountsConfiguration.
4. Upgraded [demo application](https://spring.io/blog/2015/07/14/microservices-with-spring#running-the-system) to *Brixton* release-train (including various fixes from the comments at the end - thanks for the feedback).

*OK, let’s get started …*

Service Registration

When you have multiple processes working together they need to find each other. If you have ever used Java’s RMI mechanism you may recall that it relied on a central registry so that RMI processes could find each other. Microservices has the same requirement.

The developers at Netflix had this problem when building their systems and created a registration server called Eureka (“I have found it” in Greek). Fortunately for us, they made their discovery server open-source and Spring has incorporated into Spring Cloud, making it even easier to run up a Eureka server. Here is the *complete* discovery-server application:

@SpringBootApplication

@EnableEurekaServer

public class ServiceRegistrationServer {

public static void main(String[] args) {

// Tell Boot to look for registration-server.yml

System.setProperty("spring.config.name", "registration-server");

SpringApplication.run(ServiceRegistrationServer.class, args);

}

}

It really is that simple!

Spring Cloud is built on Spring Boot and utilizes parent and starter POMs. The important parts of the [POM](https://github.com/paulc4/microservices-demo/blob/master/pom.xml) are:

<parent>

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

<artifactId>spring-cloud-starter-parent</artifactId>

<version>\_Brixton\_.RELEASE</version> <!-- Name of release train -->

</parent>

<dependencies>

<dependency>

<!-- Setup Spring Boot -->

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter</artifactId>

</dependency>

<dependency>

<!-- Setup Spring MVC & REST, use Embedded Tomcat -->

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<!-- Spring Cloud starter -->

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

<artifactId>spring-cloud-starter</artifactId>

</dependency>

<dependency>

<!-- Eureka for service registration -->

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

<artifactId>spring-cloud-starter-eureka-server</artifactId>

</dependency>

</dependencies>

**Note:**\_Brixton\_.RELEASE is the current "release train" - a set of co-ordinated releases -- see note on Spring Cloud [home page](https://projects.spring.io/spring-cloud) (scroll down to "Release Trains" section).

By default Spring Boot applications look for an application.properties or application.ymlfile for configuration. By setting the spring.config.name property we can tell Spring Boot to look for a different file - useful if you have multiple Spring Boot applications in the same project - as I will do shortly.

This application looks for registration-server.properties or registration-server.yml. Here is the relevant configuration from registration-server.yml:

# Configure this Discovery Server

eureka:

instance:

hostname: localhost

client: # Not a client, don't register with yourself

registerWithEureka: false

fetchRegistry: false

server:

port: 1111 # HTTP (Tomcat) port

By default Eureka runs on port 8761, but here we will use port 1111 instead. Also by including the registration code in my process I might be a server or a client. The configuration specifies that I am not a client and stops the server process trying to register with itself.

**Using Consul**

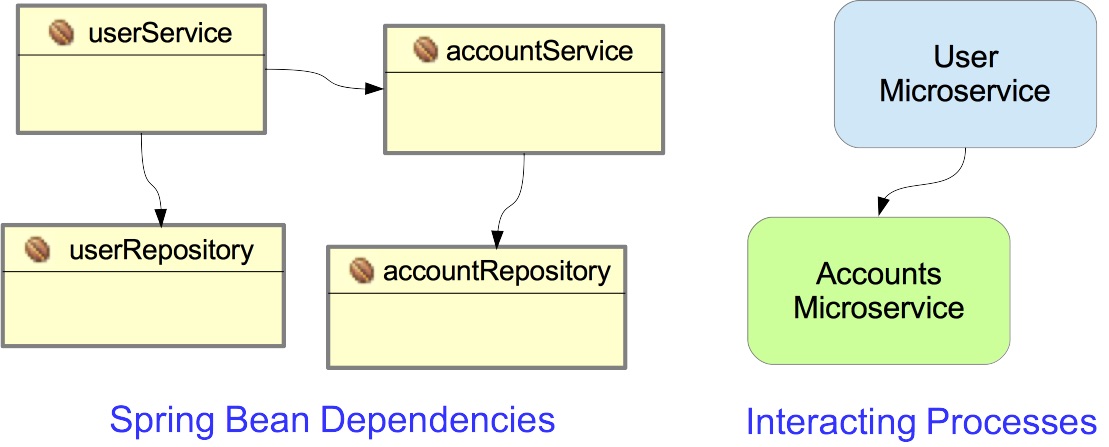
Spring Cloud also supports [Consul](https://www.consul.io/) as an alternative to Eureka. You start the Consul Agent (its registration server) using a script and then clients use it to find their microservices. For details, see this blog [article](https://spring.io/blog/2015/05/27/spring-cloud-consul-1-0-0-m1-available-now) or project [home page](https://cloud.spring.io/spring-cloud-consul).

Try running the *RegistrationServer* now (see [below](https://spring.io/blog/2015/07/14/microservices-with-spring#running-the-system) for help on running the application). You can open the Eureka dashboard here: [http://localhost:1111](http://localhost:1111/) and the section showing Applications will be empty.

From now on we will refer to the *discovery-server*since it could be Eureka or Consul (see side panel).

Creating a Microservice: *Account-Service*

A microservice is a stand-alone process that handles a well-defined requirement.

[](https://raw.githubusercontent.com/paulc4/microservices-demo/master/beans-vs-processes.jpg)

When configuring applications with Spring we emphasize Loose Coupling and Tight Cohesion, These are not new concepts (Larry Constantine is credited with first defining these in the late 1960s - [reference](https://en.wikipedia.org/wiki/Cohesion_%28computer_science%29)) but now we are applying them, not to interacting components (Spring Beans), but to interacting processes.

In this example, I have a simple Account management microservice that uses Spring Data to implement a JPA AccountRepository and Spring REST to provide a RESTful interface to account information. In most respects this is a straightforward Spring Boot application.

What makes it special is that it registers itself with the *discovery-server* at start-up. Here is the Spring Boot startup class:

@EnableAutoConfiguration

@EnableDiscoveryClient

@Import(AccountsWebApplication.class)

public class AccountsServer {

@Autowired

AccountRepository accountRepository;

public static void main(String[] args) {

// Will configure using accounts-server.yml

System.setProperty("spring.config.name", "accounts-server");

SpringApplication.run(AccountsServer.class, args);

}

}

The annotations do the work:

1. @EnableAutoConfiguration - defines this as a Spring Boot application.
2. @EnableDiscoveryClient - this enables service registration and discovery. In this case, this process registers itself with the *discovery-server* service using its application name (see below).
3. @Import(AccountsWebApplication.class) - this Java Configuration class sets up everything else (see [below](https://spring.io/blog/2015/07/14/microservices-with-spring#accountswebapplication-configuration) for more details).

What makes this a microservice is the registration with the *discovery-server* via @EnableDiscoveryClient and its YML configuration completes the setup:

# Spring properties

spring:

application:

name: accounts-service

# Discovery Server Access

eureka:

client:

serviceUrl:

defaultZone: http://localhost:1111/eureka/

# HTTP Server

server:

port: 2222 # HTTP (Tomcat) port

Note that this file

1. Sets the application name as accounts-service. This service registers under this name and can also be accessed by this name - see below.
2. Specifies a custom port to listen on (2222). All my processes are using Tomcat, they can’t all listen on port 8080.
3. The URL of the Eureka Service process - from the previous section.