# Microservices with Spring

Introduction

**NOTE:** [*Revised 21 May 2016*](https://spring.io/blog/2015/07/14/microservices-with-spring#updates-may-2016)

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.

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”!

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](http://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](http://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](http://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](http://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.

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.

Run the *AccountsService*application now and let it finish initializing. Refresh the dashboard [http://localhost:1111](http://localhost:1111/) and you should see the ACCOUNTS-SERVICE listed under Applications. Registration takes up to 30 seconds (by default) so be patient - check the log output from *RegistrationService*

**Warning:**Do not try to display XML output using the internal web-viewer of Eclipse/STS because it cannot do so. Use your favorite web browser instead.

For more detail, go here: <http://localhost:1111/eureka/apps/> and you should see something like this:

<applications>

<versions\_\_delta>1</versions\_\_delta>

<apps\_\_hashcode>UP\_1\_</apps\_\_hashcode>

<application>

<name>ACCOUNTS-SERVICE</name>

<instance>

<hostName>autgchapmp1m1.corp.emc.com</hostName>

<app>ACCOUNTS-SERVICE</app>

<ipAddr>172.16.84.1</ipAddr><status>UP</status>

<overriddenstatus>UNKNOWN</overriddenstatus>

<port enabled="true">3344</port>

<securePort enabled="false">443</securePort>

...

</instance>

</application>

</applications>

Alternatively go to <http://localhost:1111/eureka/apps/ACCOUNTS-SERVICE> and see just the details for *AccountsService* - if it’s not registered you will get a 404.

**Configuration Options**

**Registration Time:** Registration takes up to 30s because that is the default client refresh time. You can change this by setting the eureka.instance.leaseRenewalIntervalInSeconds property to a smaller number (in the demo application I have set it to 5). This *is not recommended* in [production](http://cloud.spring.io/spring-cloud-static/docs/1.0.x/spring-cloud.html#_why_is_it_so_slow_to_register_a_service). See [also](https://github.com/spring-cloud/spring-cloud-netflix/issues/373).

eureka:

instance:

leaseRenewalIntervalInSeconds: 5 # DO NOT DO THIS IN PRODUCTION

**Registration Id:** A process (microservice) registers with the discovery-service using a unique id. If another process registers with the *same* id, it is treated as a restart (for example some sort of failover or recovery) and the first process registration is discarded. This gives us the fault-tolerant system we desire.

To run multiple instances of the *same* process (for load-balancing and resilience) they need to register with a unique id. When I first wrote this blog, that was automatic and since the *Brixton*release-train, it is again.

Under the *Angel* release train, the instance-id, used by a client to register with a discovery server, was derived from the client’s service name (the same as the Spring application name) and also the client’s host name. The same processes running on the same host would therefore have the same id, so only one could ever register.

Fortunately you could set the id property manually via the client’s Eureka metadata map, like this:

eureka:

instance:

metadataMap:

instanceId: ${spring.application.name}:${spring.application.instance\_id:${server.port}}

Since the *Brixton* release train, *this is now the default*. So what does it do?

We are setting the instanceId to application-name:instance\_id, but if instance\_id is not defined, we will use application-name::server-port instead. Note that the spring.application.instance\_id is *only* set when using Cloud Foundry but it conveniently provides a unique id number for each instance of the same application. We can do something similar when running elsewhere by using the server-port (since different instances on the same machine *must* listen on different ports. Another example you will often see is ${spring.application.name}:${spring.application.instance\_id:${random.value}} but I personally find using the port number makes each instance easy to identify - the random values are just long strings that don’t mean anything.

**Note:** The syntax ${x:${y}} is Spring property shorthand for ${x} != null ? ${x} : ${y}.

Since the *Brixton* release there is also a dedicated property for this:

eureka:

instance:

instanceId: ${spring.application.name}:${spring.application.instance\_id:${random.value}}

Accessing the Microservice: *Web-Service*

To consume a RESTful service, Spring provides the RestTemplate class. This allows you to send HTTP requests to a RESTful server and fetch data in a number of formats - such as JSON and XML.

**Note:**The Accounts microservice provides a RESTful interface over HTTP, but any suitable protocol could be used. Messaging using [AMQP](http://rabbitmq.docs.pivotal.io/) or JMS is an obvious alternative.

Which formats can be used depends on the presence of marshaling classes on the classpath - for example JAXB is always detected since it is a standard part of Java. JSON is supported if Jackson jars are present in the classpath.

A microservice (discovery) client can use a RestTemplate and Spring will automatically configure it to be microservice aware (more of this in a moment).

**Encapsulating Microservice Access**

Here is part of the WebAccountService for my *client* application:

@Service

public class WebAccountsService {

@Autowired // NO LONGER auto-created by Spring Cloud (see below)

@LoadBalanced // Explicitly request the load-balanced template

// with Ribbon built-in

protected RestTemplate restTemplate;

protected String serviceUrl;

public WebAccountsService(String serviceUrl) {

this.serviceUrl = serviceUrl.startsWith("http") ?

serviceUrl : "http://" + serviceUrl;

}

public Account getByNumber(String accountNumber) {

Account account = restTemplate.getForObject(serviceUrl

+ "/accounts/{number}", Account.class, accountNumber);

if (account == null)

throw new AccountNotFoundException(accountNumber);

else

return account;

}

...

}

Note that my WebAccountService is just a wrapper for the RestTemplate fetching data from the microservice. The interesting parts are the serviceUrl and the RestTemplate.

**Accessing the Microservice**

As shown below, the serviceUrl is provided by the main program to the WebAccountController (which in turn passes it to the WebAccountService):

@SpringBootApplication

@EnableDiscoveryClient

@ComponentScan(useDefaultFilters=false) // Disable component scanner

public class WebServer {

// Case insensitive: could also use: http://accounts-service

public static final String ACCOUNTS\_SERVICE\_URL

= "http://ACCOUNTS-SERVICE";

public static void main(String[] args) {

// Will configure using web-server.yml

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

SpringApplication.run(WebServer.class, args);

}

@LoadBalanced // Make sure to create the load-balanced template

@Bean

RestTemplate restTemplate() {

return new RestTemplate();

}

/\*\*

\* Account service calls microservice internally using provided URL.

\*/

@Bean

public WebAccountsService accountsService() {

return new WebAccountsService(ACCOUNTS\_SERVICE\_URL);

}

@Bean

public WebAccountsController accountsController() {

return new WebAccountsController

(accountsService()); // plug in account-service

}

}

A few points to note:

1. The WebController is a typical Spring MVC view-based controller returning HTML. The application uses Thymeleaf as the view-technology (for generating dynamic HTML)
2. WebServer is also a @EnableDiscoveryClient but in this case as well as registering itself with the *discovery-server* (which is not necessary since it offers no services of its own) it uses Eureka to locate the account service.
3. The default component-scanner setup inherited from Spring Boot looks for @Componentclasses and, in this case, finds my WebAccountController and tries to create it. However, I want to create it myself, so I disable the scanner like this @ComponentScan(useDefaultFilters=false).
4. The *service-url* I am passing to the WebAccountController is the name the service used to register itself with the *discovery-server* - by default this is the same as the spring.application.name for the process which is account-service - see account-service.yml above. The use of upper-case is not required but it does help emphasize that *ACCOUNTS-SERVICE* is a logical host (that will be obtained via discovery) not an actual host.

**Load Balanced RestTemplate**

The RestTemplate bean will be intercepted and auto-configured by Spring Cloud (due to the @LoadBalanced annotation) to use a custom HttpRequestClient that uses Netflix [Ribbon](http://techblog.netflix.com/2013/01/announcing-ribbon-tying-netflix-mid.html) to do the microservice lookup. Ribbon is also a load-balancer so if you have multiple instances of a service available, it picks one for you. (Neither Eureka nor Consul on their own perform load-balancing so we use Ribbon to do it instead).

**Note:** From the *Brixton* Release Train (Spring Cloud 1.1.0.RELEASE), the RestTemplate is no longer created automatically. Previously it was created for you, which caused confusion and potential conflicts (sometimes Spring can be *too* helpful!).

Note that this instance is qualified using @LoadBalanced. (The [annotation](https://github.com/spring-cloud/spring-cloud-commons/blob/master/spring-cloud-commons/src/main/java/org/springframework/cloud/client/loadbalancer/LoadBalanced.java) is itself annotated with @Qualifier - see [here](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html#beans-autowired-annotation-qualifiers) for details). Thus if you have more than one RestTemplate bean, you can make sure to inject the right one, like this:

@Autowired

@LoadBalanced // Make sure to inject the load-balanced template

protected RestTemplate restTemplate;

If you look in the [RibbonClientHttpRequestFactory](https://github.com/spring-cloud/spring-cloud-netflix/blob/master/spring-cloud-netflix-core/src/main/java/org/springframework/cloud/netflix/ribbon/RibbonClientHttpRequestFactory.java) you will see this code:

String serviceId = originalUri.getHost();

ServiceInstance instance =

loadBalancer.choose(serviceId); // loadBalancer uses Ribbon

... if instance non-null (service exists) ...

URI uri = loadBalancer.reconstructURI(instance, originalUri);

The loadBalancer takes the logical service-name (as registered with the *discovery-server*) and converts it to the actual hostname of the chosen microservice.

A RestTemplate instance is thread-safe and can be used to access any number of services in different parts of your application (for example, I might have a CustomerService wrapping the same RestTemplate instance accessing a customer data microservice).

**Configuration**

Below the relevant configuration from web-server.yml. It is used to:

1. Set the application name
2. Define the URL for accessing the discovery server
3. Set the Tomcat port to 3333

# Spring Properties

spring:

application:

name: web-service

# Discovery Server Access

eureka:

client:

serviceUrl:

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

# HTTP Server

server:

port: 3333 # HTTP (Tomcat) port

How to Run the Demo

A small demo of this system is at [http://github.com/paulc4/microservices-demo](https://github.com/paulc4/microservices-demo). Clone it and either load into your favorite IDE or use maven directly. Suggestions on how to run the demo are included in the [README](https://github.com/paulc4/microservices-demo/blob/master/README.md) on the project homepage.

Extra Notes

Some notes about Spring Boot usage by these applications. If you are not familiar with Spring Boot, this explains some of the “magic”!

**View Templating Engines**

The Eureka dashboard (inside RegistrationServer) is implemented using FreeMarker templates but the other two applications use Thymeleaf. To make sure each uses the right view engine, there is extra configuration in each YML file.

This is at the end of registration-server.yml to disable Thymeleaf.

...

# Discovery Server Dashboard uses FreeMarker. Don't want Thymeleaf templates

spring:

thymeleaf:

enabled: false # Disable Thymeleaf spring:

Since both AccountService and WebService use thymeleaf, we also need to point each at their own templates. Here is part of account-server.yml:

# Spring properties

spring:

application:

name: accounts-service # Service registers under this name

freemarker:

enabled: false # Ignore Eureka dashboard FreeMarker templates

thymeleaf:

cache: false # Allow Thymeleaf templates to be reloaded at runtime

prefix: classpath:/accounts-server/templates/

# Template location for this application only

...

web-server.yml is similar but its templates are defined by

prefix: classpath:/web-server/templates/

Note the / on the end of each spring.thymeleaf.prefix classpath - this is *crucial*.

**Command-Line Execution**

The jar is compiled to automatically run io.pivotal.microservices.services.Main when invoked from the command-line - see [Main.java](https://github.com/paulc4/microservices-demo/blob/master/src/main/java/io/pivotal/microservices/services/Main.java).

The Spring Boot option to set the start-class can be seen in the [POM](https://github.com/paulc4/microservices-demo/blob/master/pom.xml):

<properties>

<!-- Stand-alone RESTFul application for testing only -->

<start-class>io.pivotal.microservices.services.Main</start-class>

</properties>

**AccountsConfiguration class**

@SpringBootApplication

@EntityScan("io.pivotal.microservices.accounts")

@EnableJpaRepositories("io.pivotal.microservices.accounts")

@PropertySource("classpath:db-config.properties")

public class AccountsWebApplication {

...

}

This is the main configuration class for AccountService which is a classic Spring Boot application using Spring Data. The annotations do most of the work:

1. @SpringBootApplication - defines this as a Spring Boot application. This convenient annotation combines @EnableAutoConfiguration, @Configuration and @ComponentScan(which, by default, causes Spring to search the package containing this class, and its sub-packages, for components - potential Spring Beans: AccountController and AccountRepository) .
2. @EntityScan("io.pivotal.microservices.accounts") - because I am using JPA, I need to specify where the @Entity classes are. Normally this is an option you specify in JPA’s persistence.xml or when creating a LocalContainerEntityManagerFactoryBean. Spring Boot will create this factory-bean for me because the spring-boot-starter-data-jpadependency is on the class path. So an alternative way of specifying where to find the @Entity classes is by using@EntityScan. This will find Account.
3. @EnableJpaRepositories("io.pivotal.microservices.accounts")- look for classes extending Spring Data’s Repository marker interface and automatically implement them using JPA - see [Spring Data JPA](http://projects.spring.io/spring-data-jpa).
4. @PropertySource("classpath:db-config.properties") - properties to configure my DataSource – see [db-config.properties](https://github.com/paulc4/microservices-demo/blob/master/src/main/resources/db-config.properties).

**Configuring Properties**

As mentioned above, Spring Boot applications look for either application.properties or application.yml to configure themselves. Since all three servers used in this application are in the same project, they would automatically use the same configuration.

To avoid that, each specifies an alternative file by setting the spring.config.name property.

For example here is part of WebServer.java.

public static void main(String[] args) {

// Tell server to look for web-server.properties or web-server.yml

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

SpringApplication.run(WebServer.class, args);

}

At runtime, the application will find and use web-server.yml in src/main/resources.

**Logging**

Spring Boot sets up INFO level logging for Spring by default. Since we need to examine the logs for evidence of our microservices working, I have raised the level to WARN to reduce the amount of logging.

To do this, the logging level would need to be specified in each of the xxxx-server.ymlconfiguration files. This is usually the best place to define them as logging properties *cannot*be specified in property files (logging has already been initialized before @PropertySource directives are processed). There is a note on this in the Spring Boot manual, but it’s easy to miss.

Rather than duplicate the logging configuration in each YAML file, I instead opted to put it in the logback configuration file, since Spring Boot uses logback - see [src/main/resources/logback.xml](https://github.com/paulc4/microservices-demo/blob/master/src/main/resources/logback.xml). All three services will share the same logback.xml.