Wiring Microservices with Spring Cloud

Posted by [**Rob Harrop**](https://www.infoq.com/author/Rob-Harrop) on Jun 15, 2016 *|* [Discuss](https://www.infoq.com/articles/spring-cloud-service-wiring#theCommentsSection)

* Share
* **[Share](https://www.addthis.com/bookmark.php?v=250&username=infoq)**
* |
* [**Read later**](https://www.infoq.com/articles/spring-cloud-service-wiring)
* [**My Reading List**](https://www.infoq.com/showbookmarks.action)

|  |
| --- |
| Key takeaways   * Spring Cloud provides a wealth of options for wiring service dependencies in microservice systems. * Spring Cloud Config provides Git-managed versioning for configuration data and enables dynamic refresh of this data without the need for restart. * Pairing Spring Cloud with the Netflix Eureka and Ribbon components allows application services to find each other dynamically, and pushes load-balancing decisions away from a dedicated proxying load-balancer and into the client services. * Load balancing solutions like AWS ELB still have a place at the edge of our system where we don't control the inbound traffic. * For communication between middle-tier microservices, Ribbon provides a more reliable and more performant solution that doesn't couple you to any particular cloud provider. |

Introduction

As we move towards microservice-based architectures, we're faced with an important decision: how do we wire our services together? Components in a monolithic system communicate through a simple method call, but components in a microservice system likely communicate over the network through REST, web services or some RPC-like mechanism.

**Related Vendor Content**

[**Modern Java EE Design Patterns (By O'Reilly) -- Download Now**](https://www.infoq.com/infoq/url.action?i=11471&t=f)

[**The Ultimate DevOps Toolkit**](https://www.infoq.com/vendorcontent/show.action?vcr=3934&utm_source=infoq&utm_medium=VCR&utm_campaign=vcr_articles_click)

[**Top 10 Java Performance Problems**](https://www.infoq.com/vendorcontent/show.action?vcr=3933&utm_source=infoq&utm_medium=VCR&utm_campaign=vcr_articles_click)

[**Docker Networking and Service Discovery - Download the FREE eBook (by O'Reilly)**](https://www.infoq.com/vendorcontent/show.action?vcr=3862&utm_source=infoq&utm_medium=VCR&utm_campaign=vcr_articles_click)

[**REST and Microservices - Breaking Down the Monolith Step by Asynchronous Step**](https://www.infoq.com/vendorcontent/show.action?vcr=3889&utm_source=infoq&utm_medium=VCR&utm_campaign=vcr_articles_click)

**Related Sponsor**

[Mark Little: Are there Different Types of Microservices?](https://www.infoq.com/infoq/url.action?i=12242&t=f)

In a monolith we can avoid issues of service wiring altogether and have each component create its own dependencies as it needs them. In reality, we rarely do this. Such close coupling between a component and its dependencies makes our system overly rigid, and hampers our testing efforts. Instead, we opt to externalise a component's dependencies and inject them when the component is created; dependency injection is service wiring for classes and objects.

Suppose that we decide to implement an application as a set of microservices, we have wiring options similar to those we have for a monolith. We can hard-code the addresses of our dependencies, tying our services together closely. Alternatively, we can externalise the addresses of the services we depend on, and supply them at deploy time or run time. In this article, we'll explore how each of these options manifests in a microservices application built with Spring Boot and Spring Cloud.

Consider the simple repmax microservice system shown below:

**The repmax system**

The repmax application tracks a user's weight-lifting history and tracks the top five users for each lift as the leaderboard. The logbook service receives workout data from the UI and stores the full history for each user. When a user logs a lift in a workout, the logbook sends the details for that lift to the leaderboard service.

From the diagram, we see that the logbook service depends on the leaderboard service. Following good practice, we abstract this dependency behind an interface, theLeaderBoardApi:

public interface LeaderBoardApi {

void recordLift(Lift lift);

}

Since this is a Spring application, we'll be using RestTemplate to handle the details of the communication between the logbook and leaderboard services:

abstract class AbstractLeaderBoardApi implements LeaderBoardApi {

private final RestTemplate restTemplate;

public AbstractLeaderBoardApi() {

RestTemplate restTemplate = new RestTemplate();

restTemplate.getMessageConverters().add(new FormHttpMessageConverter());

this.restTemplate = restTemplate;

}

@Override

public final void recordLift(Lifter lifter, Lift lift) {

URI url = URI.create(String.format("%s/lifts", getLeaderBoardAddress()));

MultiValueMap<String, String> params = new LinkedMultiValueMap<>();

params.set("exerciseName", lift.getDescription());

params.set("lifterName", lifter.getFullName());

params.set("reps", Integer.toString(lift.getReps()));

params.set("weight", Double.toString(lift.getWeight()));

this.restTemplate.postForLocation(url, params);

}

protected abstract String getLeaderBoardAddress();

}

The AbstractLeaderBoardApi class captures all the logic needed to package up a POSTrequest to the leaderboard service, leaving subclasses to specify the exact address for theleaderboard service.

The simplest possible model for wiring one microservice to another is to hard-code the address of each dependency that a service needs. This corresponds to hard-coding the instantiation of a dependency in the monolith world. This is easily captured in the StaticWiredLeaderBoardApiclass:

public class StaticWiredLeaderBoardApi extends AbstractLeaderBoardApi {

@Override

protected String getLeaderBoardAddress() {

return "http://localhost:8082";

}

}

The hard-coded address for the service allows us to get started quickly, but is not tenable in a real environment. Every different deployment of the services requires custom compilation, this quickly becomes painful and error-prone.

If this were a monolith and we were looking to refactor our application to remove the hard-coded address, we'd start by externalising the address into some configuration file. The same approach works for our microservice application: we push the address into a configuration file and have our API implementation read the address from the configuration.

Spring Boot makes defining and injecting configuration parameters trivial. We add the address parameter to the application.properties file:

leaderboard.url=http://localhost:8082

We then inject this parameter into our ConfigurableLeaderBoardApi implementation using the@Value annotation:

public class ConfigurableLeaderBoardApi extends AbstractLeaderBoardApi {

private final String leaderBoardAddress;

@Autowired

public ConfigurableLeaderBoardApi(@Value("${leaderboard.url}") String leaderBoardAddress) {

this.leaderBoardAddress = leaderBoardAddress;

}

@Override

protected String getLeaderBoardAddress() {

return this.leaderBoardAddress;

}

}

The [Externalized Configuration](http://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-external-config.html#boot-features-external-config) support in Spring Boot allows us to change the value ofleaderboard.url not just by editing the configuration file, but also by specifying an environment variable when starting our application:

LEADERBOARD\_URL=http://repmax.skipjaq.com/leaderboard java -jar repmax-logbook-1.0.0-RELEASE.jar

We can now point an instance of the logbook service to any instance of the leaderboardservice without having to make code changes. If we are following [12 factor](http://12factor.net/) principles in our system, then the wiring information will likely be available in the environment, so it can be mapped directly into the application without too much fuss.

Platform-as-a-Service (PaaS) systems such as [Cloud Foundry](https://www.cloudfoundry.org/) and [Heroku](https://www.heroku.com/) expose wiring information for managed services such as databases and messaging systems through the environment, allowing us to wire in these dependencies in exactly the same way. Indeed, it makes little sense to differentiate between wiring two services together and wiring a service to its datastore; in both cases we are simply connecting two distributed systems.

Beyond Point-to-Point Wiring

For simple applications, external configuration for dependency addresses may well be sufficient. For applications of any size though, it's likely that we'll want to move beyond simple point-to-point wiring and introduce some form of load-balancing.

If each of our services depends directly on a single instance of its downstream services, then any failure in the downstream chain is likely to be catastrophic for our end users. Likewise, if a downstream service becomes overloaded, then our users pay the price for this through increased response times. What we need is load balancing.

Instead of depending directly on a downstream instance, we want to share the load across a set of downstream service instances. If one of these instances fails or becomes overloaded then the other instances can pick up the slack. The simplest way to introduce load balancing into this architecture is using a load-balancing proxy. The diagram below shows how this might look in an Amazon Web Services deployment using Elastic Load Balancing:

**ELB for the Leader Board**

Rather than have the logbook service talk directly to the leaderboard service, each request is routed through an ELB. The ELB routes each request to one of the backend leaderboardservices. With the ELB acting as an intermediary, load is shared across multiple leaderboard instances, helping to reduce the load on individual instances.

Load balancing with ELB is dynamic; new instances can be added to set of backend instances at runtime so if we encounter a spike in incoming traffic, we can start more leaderboard instances to handle this spike.

Spring Boot applications using [the actuator](http://docs.spring.io/spring-boot/docs/current/reference/html/production-ready.html) expose a /health endpoint that the ELB monitors periodically. Instances that respond to these healthchecks remain in the ELB's active set, but after a number of failed checks, the instance is removed from service.

The leaderboard service isn't the only service in our system that can benefit from this load-balancing. The logbook service, and indeed the front-end UI, both benefit from the scalability and resilience afforded by load balancing.

Dynamic Re-Configuration

Whether we are using [AWS ELB](https://aws.amazon.com/elasticloadbalancing/), [Google Compute Load Balancing](https://cloud.google.com/compute/docs/load-balancing/) or even our own load-balancing proxy using something like HAProxy or NGINX, we still need to wire our services up to the load balancer.

One approach to this is to give each load balancer a well-known DNS name, sayleaderboard.repmax.local that can be hard-coded into the application using the static wiring approach shown earlier. This approach is reasonably flexible thanks in large part to how flexible DNS is. However, relying on a hard-coded name means we have to configure a DNS server in every environment we are running our services in. Requiring a customised DNS is particularly painful in development where we have to support many different OSes. A better solution is to inject whatever address the load balancer has naturally into our services as we did withleaderboard.url in the previous example.

In cloud environments such as AWS and GCP, load balancers - and their addresses - are ephemeral. When a load balancer gets destroyed and re-created it typically gets a new address. If we hard-code the address of the load balancer, we need to re-compile our code just to pick up the address change. With externalised configuration, we simply modify the configuration file and restart.

DNS is a handy way of hiding the ephemeral nature of load balancer addresses. Each load balancer is assigned a stable DNS name and it's this name that's injected into the calling service. When the load balancer is re-created, the DNS name is remapped to the new address for the load balancer. This DNS-based approach works well if you're prepared to run a DNS server in your environment. If you want to avoid running DNS but still allow dynamic re-configuration of your load balancers then [Spring Cloud Config](http://cloud.spring.io/spring-cloud-config/) is the answer.

Spring Cloud Config runs a small service, the Config Server, to provide centralised access to configuration data through a REST API. By default, configuration data is stored in a Git repository and is exposed to our Spring Boot services through the standard PropertySourceabstraction. Thanks to the use of PropertySource, we can seamlessly combine configuration in local properties files with configuration stored in the Config Server. For local development, we'll likely use configuration from a local properties file and only override this when deploying the application in a real environment.

To replace our ConfigurableLeaderBoardApi with an implementation that uses Spring Cloud Config we start by initialising a Git repo with the desired configuration:

mkdir -p ~/dev/repmax-config-repo

cd ~/dev/repmax-config-repo

git init

echo 'leaderboard.lb.url=http://some.lb.address' >> repmax.properties

git add repmax.properties

git commit -m 'LB config for the leaderboard service'

The repmax.properties file contains the configuration for the default profile of the repmaxapplication. If we want to add configuration for another profile, say development, then we simply commit another file called repmax-development.properties.

To run the Config Server, we can either run the default Config Server provided by the spring-cloud-config-server project or we can create our own simple Spring Boot project that hosts the Config Server:

@SpringBootApplication

@EnableConfigServer

public class RepmaxConfigServerApplication {

public static void main(String[] args) {

SpringApplication.run(RepmaxConfigServerApplication.class, args);

}

}

The @EnableConfigServer annotation starts Config Server in this tiny Spring Boot application. We point the Config Server at our Git repo using the spring.cloud.config.server.git.uriproperty. For local testing it makes sense to add this to the application.properties file for the Config Server application:

spring.cloud.config.server.git.uri=file://${user.home}/dev/repmax-config-repo

This way, every developer in our team can start up a Config Server on their machine and test it against a local Git repository. We can verify that the properties for the repmax application are exposed by Config Server by visiting http://localhost:8888/repmax/default in the browser when the Config Server is running:

**Browsing configuration in Config Server**

Here we can see that the leaderboard.lb.url property is exposed from therepmax.properties file and it has the value http://localhost:8083. The version property in the JSON payload shows which Git rev the config was loaded from.

At production time we take advantage of the PropertySource abstraction to supply the Git repository name as an environment variable:

SPRING\_CLOUD\_CONFIG\_SERVER\_GIT\_URI=https://gitlab.com/rdh/repmax-config-repo java -jar repmax-config-server-1.0.0-RELEASE.jar

A Spring Cloud Config Client

Modifying the logbook service to read its configuration from our new Config Server requires only a few steps. First we add a dependency on spring-cloud-starter-config in thebuild.gradle file;

compile("org.springframework.cloud:spring-cloud-starter-config:1.1.1.BUILD-SNAPSHOT")

Next, we supply some basic bootstrap configuration needed by the Config Client. Recall that our Config Server exposes configuration from a file called repmax.properties. We need to tell the Config Client the name of our application. Such bootstrap configuration goes in thebootstrap.properties file of the logbook service:

spring.application.name=repmax

By default, the Config Client looks for a Config Server at http://localhost:8888. To tweak this, specify the SPRING\_CLOUD\_CONFIG\_URI environment when starting the client application.

Once the client, in this case logbook is started, we can check that the configuration from the Config Server is correctly loaded by visiting http://localhost:8081/env:

**Checking that Config Client can see the Config Server**

With the logbook service configured to use Config Client, we can modify theConfigurableLeaderBoardApi to obtain the load balancer address from theleaderboard.lb.url property exposed in the Config Server.

Enabling Dynamic Refresh

With the configuration stored in a centralised place we have an easy way to change the repmaxconfiguration in a way that is visible to all services. However, picking up those configurations still requires a restart. We can do better. Spring Boot provides the @ConfigurationPropertiesannotation that allows us to map configuration directly on to JavaBeans. Spring Cloud Config goes a step further, and exposes a /refresh endpoint in every client service. Beans that are annotated with @ConfigurationProperties have their properties updated whenever a refresh is triggered through the /refresh endpoint.

Any bean can be annotated with @ConfigurationProperties, but it makes sense to restrict refresh support to just the beans that contain configuration data. To this end, we extract aLeaderboardConfig bean that serves as a holder for the leaderboard address:

@ConfigurationProperties("leaderboard.lb")

public class LeaderboardConfig {

private volatile String url;

public String getUrl() {

return this.url;

}

public void setUrl(String url) {

this.url = url;

}

}

The value of the @ConfigurationProperties annotation is the prefix for configuration values we want to map into our bean. Then, each value is mapped using standard JavaBean naming conventions. In this case, the url bean property is mapped to leaderboard.lb.url in the configuration.

We then modify ConfigurableLeaderBoardApi to accept an instance of LeaderboardConfigrather than the raw leaderboard address:

public class ConfigurableLeaderBoardApi extends AbstractLeaderBoardApi {

private final LeaderboardConfig config;

@Autowired

public ConfigurableLeaderBoardApi(LeaderboardConfig config) {

this.config = config;

}

@Override

protected String getLeaderBoardAddress() {

return this.config.getLeaderboardAddress();

}

}

To trigger a config refresh, send an HTTP POST request to the /refresh endpoint of thelogbook service:

curl -X POST http://localhost:8081/refresh

Towards Service Discovery

With Spring Cloud Config and a load-balancing proxy between our logbook and leaderboardservices, our application is in good shape. However, there are still some improvements to be made.

If we're deploying in AWS or GCP, we can take advantage of the elasticity of the load balancers in those environments, but if we're using an off-the-shelf load balancing proxy like HAProxy or NGINX, then we are forced to handle service discovery and registration ourselves. Every new instance of the leaderboard has to be configured with the proxy and every failed instance has to be removed from the proxy. What we really want is dynamic discovery where each service instance registers itself ready for discovery by its consumers.

There's another issue lurking with the use of load-balancing proxies: reliability. The need to route all traffic through the proxy puts our system at the mercy of that proxy's reliability. Downtime in the proxy is going to cause downtime in the system. We might also wonder at the overhead of having to talk from client to proxy and from proxy to server.

To overcome these problems, Netflix created Eureka. Eureka is a client-server system providing service registration and discovery. As service instances start they register themselves with the Eureka server. Client services, such as our logbook, contact the Eureka Server to obtain a list of available services. Communication between clients and servers is point-to-point.

Eureka removes the need for a proxy, improving the reliability of our systems. If ourleaderboard proxy dies, then the logbook service can no longer talk to the leaderboardservice at all. With Eureka in place, logbook knows about all of the leaderboard instances so, even if one fails, logbook can just move on to the next leaderboard instance and try again.

You might wonder whether the Eureka Server itself becomes a point of failure in our system architecture. Aside from the ability to configure a *cluster* of Eureka Servers, each Eureka Client caches the state of the running services locally. Provided we have a service monitor such assystemd running our Eureka Server, we can happily tolerate the occasional crash.

As with Config Server, we can run Eureka Server as a small Spring Boot application:

@SpringBootApplication

@EnableEurekaServer

public class RepmaxEurekaServerApplication {

public static void main(String[] args) {

SpringApplication.run(RepmaxEurekaServerApplication.class, args);

}

}

The @EnableEurekaServer annotation instructs Spring Boot to start the Eureka when the application starts. By default, the server will attempt to contact other servers for HA purposes. In a standalone installation it makes sense to turn this off. In application.yml:

server:

port: 8761

eureka:

instance:

hostname: localhost

client:

registerWithEureka: false

fetchRegistry: false

Note that we follow the common convention and run Eureka Server on port 8761. Visitinghttp://localhost:8761 shows the Eureka dashboard. Since we haven't registered any services yet, the list of available instance is empty:

**Blank Eureka dashboard**

To register the leaderboard service with Eureka we annotate the application class with@EnableEurekaClient. We need to tell the Eureka Client where to find the server and what name to use for the application when registering it with the server. Inapplication.properties:

spring.application.name=repmax-leaderboard

eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka

When the leaderboard service starts, Spring Boot detects the @EnableEurekaClientannotation and starts the Eureka Client that, in turn, registers the leaderboard service with the Eureka Server. The Eureka dashboard shows the newly registered service:

**Eureka dashboard after registration**

The logbook service is configured as a Eureka client in exactly the same way as theleaderboard service. We add the @EnableEurekaClient annotation and configure the Eureka Service URL.

With the Eureka Client enabled in the logbook service, Spring Cloud exposes aDiscoveryClient bean that allows us to lookup service instances:

@Component

public class DiscoveryLeaderBoardApi extends AbstractLeaderBoardApi {

public DiscoveryLeaderBoardApi(DiscoveryClient discoveryClient) {

this.discoveryClient = discoveryClient;

}

private final DiscoveryClient discoveryClient;

@Override

protected String getLeaderBoardAddress() {

List<ServiceInstance> instances = this.discoveryClient.getInstances("repmax-leaderboard");

if(instances != null && !instances.isEmpty()) {

ServiceInstance serviceInstance = instances.get(0);

return String.format("http://%s:%d", serviceInstance.getHost(), serviceInstance.getPort());

}

throw new IllegalStateException("Unable to locate a leaderboard service");

}

}

We call DiscoveryClient.getInstances to obtain a list of ServiceInstances, each one corresponding to an instance of the leaderboard service that is registered with the Eureka Server. For simplicity, we pick the first one in the list and use that for our remote call.

Load Balancing on the Client

With Eureka in place, services now discover each other dynamically and communicate directly with each other, avoiding the overhead and possible point-of-failure that a proxying load balancer introduces. The trade-off, of course, is that we've pushed the complexity of load balancing into our code.

You'll notice that our DiscoveryLeaderBoardApi.getLeaderBoardAddress method naively selects the first ServiceInstance it finds for every remote call. This is hardly balancing the load across the available instances. Thankfully, there's another Netflix Cloud component available that can handle this client-side load balancing for us: [Ribbon](https://github.com/Netflix/ribbon).

Using Ribbon with Spring Cloud and our existing Eureka setup is trivial. We simply add a dependency on spring-cloud-starter-ribbon in the logbook service and switch from using the DiscoveryClient to the LoadBalancerClient:

public class RibbonLeaderBoardApi extends AbstractLeaderBoardApi {

private final LoadBalancerClient loadBalancerClient;

@Autowired

public RibbonLeaderBoardApi(LoadBalancerClient loadBalancerClient) {

this.loadBalancerClient = loadBalancerClient;

}

@Override

protected String getLeaderBoardAddress() {

ServiceInstance serviceInstance = this.loadBalancerClient.choose("repmax-leaderboard");

if (serviceInstance != null) {

return String.format("http://%s:%d", serviceInstance.getHost(), serviceInstance.getPort());

} else {

throw new IllegalStateException("Unable to locate a leaderboard service");

}

}

}

Now, the job of a choosing a ServiceInstance is passed off to Ribbon which has all the smarts for monitoring endpoint health and load balancing built in.

Summary

During the course of this article we've looked at a variety of approaches for wiring microservices together. The simplest possible approach is to hard-code the address for each dependency that a service needs. This approach allows us to get started quickly, but is not tenable in a real environment.

For a basic real-world application external configuration using an application.properties file for dependency addresses may well be sufficient. Platform-as-a-Service (PaaS) systems such as Cloud Foundry and Heroku expose wiring information allowing us to wire in these dependencies in exactly the same way.

Larger applications however are likely to need to move beyond simple point-to-point wiring and introduce some form of load-balancing. Spring Cloud Config coupled with a load-balancing proxy is one solution, but if we're using an off-the-shelf load-balancing proxy like HAProxy or NGINX, then we are forced to handle service discovery and registration ourselves, and the proxy gives us a single point of failure for all traffic. Adding Netflix's Eureka and Ribbon components allows services in our applications to find each other dynamically and pushes load-balancing decisions away from a dedicated proxying load-balancer and in to the client services.

Load balancing solutions like AWS ELB still have a place at the edge of our system where we don't control the inbound traffic For communication between middle-tier microservices, Ribbon provides a more reliable and more performant solution that doesn't couple you to any particular cloud provider.

About the Author

**Rob Harrop** is CTO at Skipjaq, applying machine learning to the problem of performance management. Before Skipjaq, Rob was most well known as a co-founder of SpringSource, the software company behind the wildly-successful Spring Framework. At SpringSource he was a core contributor to the Spring Framework and led the team that built dm Server (now Eclipse Virgo). Prior to SpringSource, Rob was (at the age of 19) co-founder and CTO at Cake Solutions, a boutique consultancy in Manchester, UK. A respected author, speaker and teacher, Rob writes and talks frequently about large-scale systems, cloud architecture and functional programming. His published works include the highly-popular Spring Framework reference "Pro Spring".