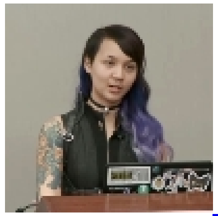
Featured in Development

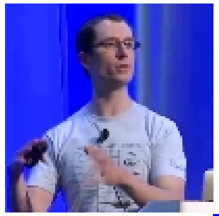
The Nihilist’s Guide to Wrecking Humans & Systems

 Christina Camilleri talks about how social engineering can be used in conjunction with technical attacks to create sophisticated and destructive attack chains, shares some real world war stories and highlights what can be done to protect against these threats.

- Security

Featured in Architecture & Design

The Seven (More) Deadly Sins of Microservices

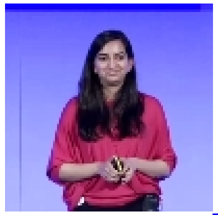

Daniel Bryant talks about the 2016 edition of the seven deadly sins that if left unchecked could easily ruin our next microservices project, and some of the nastiest anti-patterns in microservices, providing tools to not only avoid but also slay these demons before they tie up the project in their own special brand of hell.

All in **Architecture & Design**

- Data Science
  - Big Data
  - Machine Learning
  - NoSQL
  - Database
  - Data Analytics
  - Streaming

Featured in Data Science

Creating Customer-centric Products Using Big Data

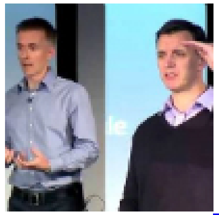

Kriti Sharma talks about how Barclays is solving some of the toughest big data challenges in financial services using scalable, open source technology. He shares case studies in the areas of: user experience centric predictive analytics and marketing, personalization of applications and targeted offers, social data analytics and sentiment analysis, open stack and open data collaboration etc.

All in **Data Science**

- Culture & Methods
  - Agile
  - Leadership
  - Team Collaboration
  - Testing
  - Project Management
  - UX
  - Scrum
  - Lean/Kanban
  - Personal Growth

Featured in Culture & Methods

Culture Eats Principles for Breakfast

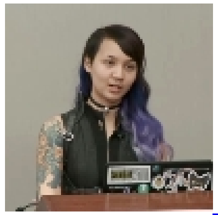

Ian Dugmore and Jonathan Smart talk about the story of Agile transformation across Barclays, what they have done and how they've done it, not only in IT but beyond IT. They look at the challenge of using Agile to bring about culture change from different perspectives, comparing and contrasting what needs to be done at an organizational level to make change happen.

All in **Culture & Methods**

- DevOps
  - Infrastructure
  - Continuous Delivery
  - Automation
  - Containers
  - Cloud

Featured in DevOps

The Nihilist’s Guide to Wrecking Humans & Systems


Christina Camilleri talks about how social engineering can be used in conjunction with technical attacks to create sophisticated and destructive attack chains, shares some real world war stories and highlights what can be done to protect against these threats.

All in **DevOps**

San Francisco	Nov 7-11
London	Mar 6-10, 2017
New York	Jun 26-30, 2017

- Mobile
- HTML5
- JavaScript
- APM
- Java
- Microservices
- DevOps
- API Design
- Database

All topics

You are here: InfoQ Homepage Articles Wiring Microservices with Spring Cloud

Wiring Microservices with Spring Cloud

Posted by Rob Harrop on Jun 15, 2016 | Discuss

- Share
- |
-
-
-
-
-
-
-
- "Read later"
- "My Reading List"

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.

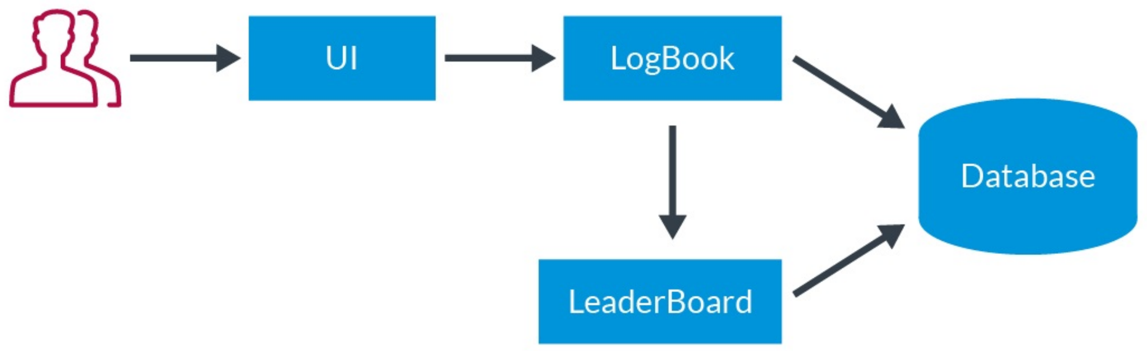
Introduction

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.

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, the `LeaderBoardApi`:

```
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 `POST` request to the `leaderboard` service, leaving subclasses to specify the exact address for the `leaderboard` 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 `StaticWiredLeaderBoardApi` class:

```
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 support in Spring Boot allows us to change the value of `leaderboard.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 `leaderboard` service without having to make code changes. If we are following 12 factor 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.

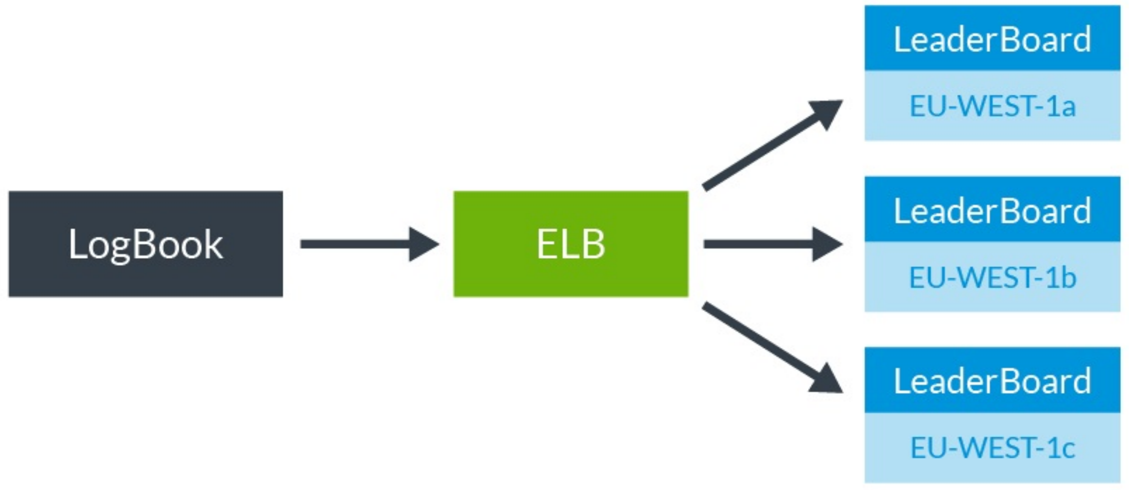
Platform-as-a-Service (PaaS) systems such as Cloud Foundry and Heroku expose wiring information for managed services such as databases and messaging systems through the environment, allowing us to wire in these dependencies in exactly the same way. Indeed, it makes little sense to differentiate between 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 `leaderboard` services. 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 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, Google Compute 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, say `leaderboard.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 with `leaderboard.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 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 `PropertySource` abstraction. 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 `repmax` application. 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:

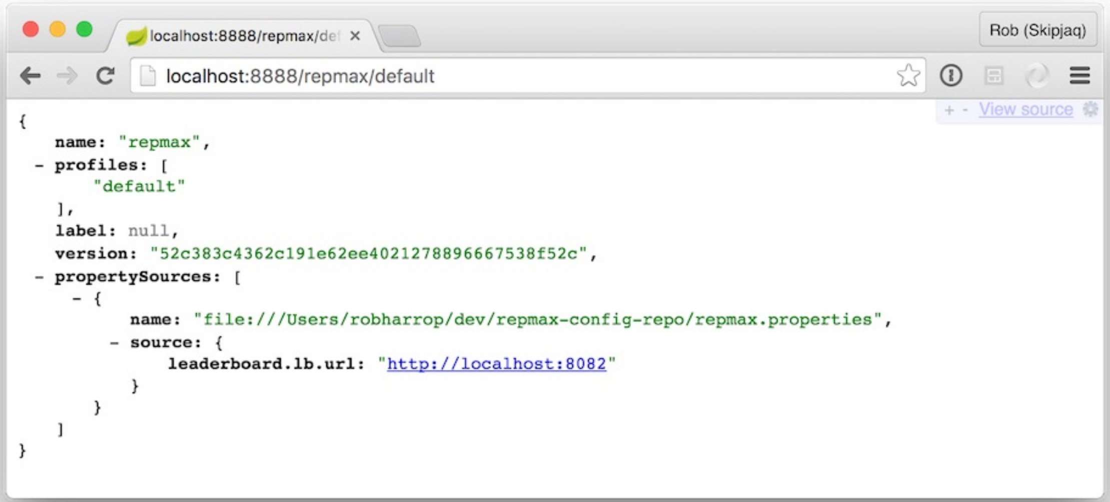
```
@SpringBootApplication
@EnableConfigServer
public class RepmaxConfigServerApplication {

    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.uri` property. 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:



```
{
    name: "repmax",
  - profiles: [
        "default"
    ],
    label: null,
    version: "52c383c4362c191e62ee4021278896667538f52c",
  - propertySources: [
      - {
            name: "file:///Users/robharrop/dev/repmax-config-repo/repmax.properties",
          - source: {
                leaderboard.lb.url: "http://localhost:8082"
            }
        }
    ]
}
```

Browsing configuration in Config Server

Here we can see that the `leaderboard.lb.url` property is exposed from the `repmax.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 the `build.gradle` file;

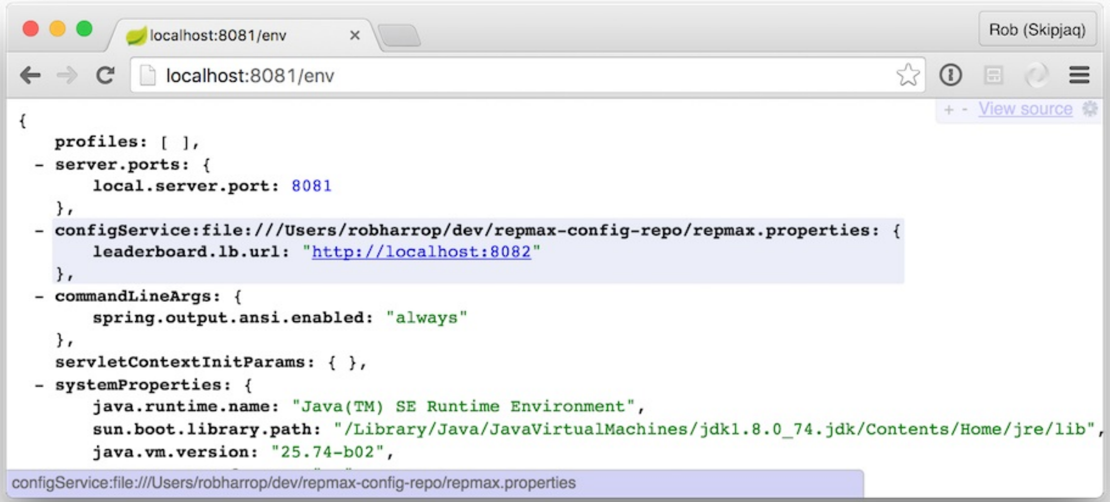
```
compile("org.springframework.cloud:spring-cloud-starter-config:1.1.1.BUILD-SNAPSHOT")
```

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 the `bootstrap.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`:



```
{
    profiles: [ ],
  - server.ports: {
        local.server.port: 8081
    },
  - configService:file:///Users/robharrop/dev/repmax-config-repo/repmax.properties: {
        leaderboard.lb.url: "http://localhost:8082"
    },
  - commandLineArgs: {
        spring.output.ansi.enabled: "always"
    },
    servletContextInitParams: { },
  - systemProperties: {
        java.runtime.name: "Java(TM) SE Runtime Environment",
        sun.boot.library.path: "/Library/Java/JavaVirtualMachines/jdk1.8.0_74.jdk/Contents/Home/jre/lib",
        java.vm.version: "25.74-b02",
```

configService:file:///Users/robharrop/dev/repmax-config-repo/repmax.properties

Checking that Config Client can see the Config Server

With the `logbook` service configured to use Config Client, we can modify the `ConfigurableLeaderBoardApi` to obtain the load balancer address from the `leaderboard.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 `repmax` configuration 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 `@ConfigurationProperties` annotation 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 a `LeaderboardConfig` 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 `LeaderboardConfig` rather 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 the `logbook` service:

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

Towards Service Discovery

With Spring Cloud Config and a load-balancing proxy between our `logbook` and `leaderboard` services, 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 our `leaderboard` proxy dies, then the `logbook` service can no longer talk to the `leaderboard` service 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 as `systemd` 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:

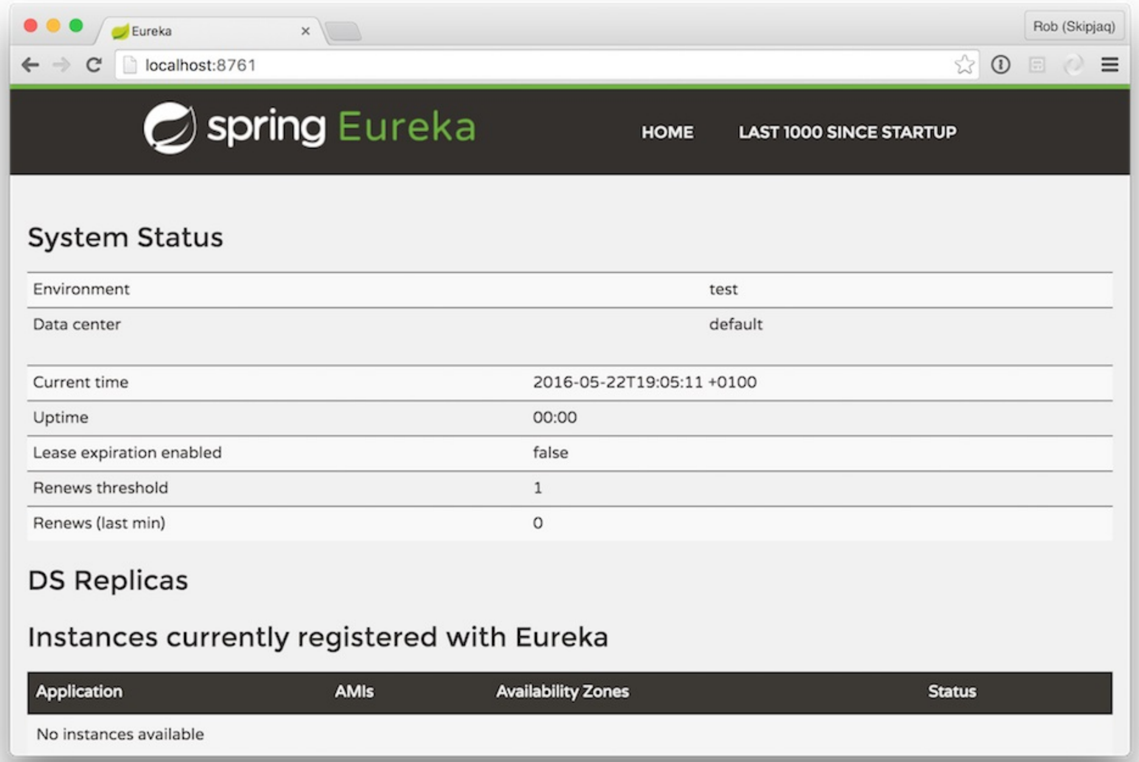
```
@SpringBootApplication
@EnableEurekaServer
public class RepmaxEurekaServerApplication {

    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`. Visiting `http://localhost:8761` shows the Eureka dashboard. Since we haven't registered any services yet, the list of available instance is empty:



System Status

Environment	test
Data center	default

Current time	2016-05-22T19:05:11 +0100
Uptime	00:00
Lease expiration enabled	false
Renews threshold	1
Renews (last min)	0

DS Replicas

Instances currently registered with Eureka

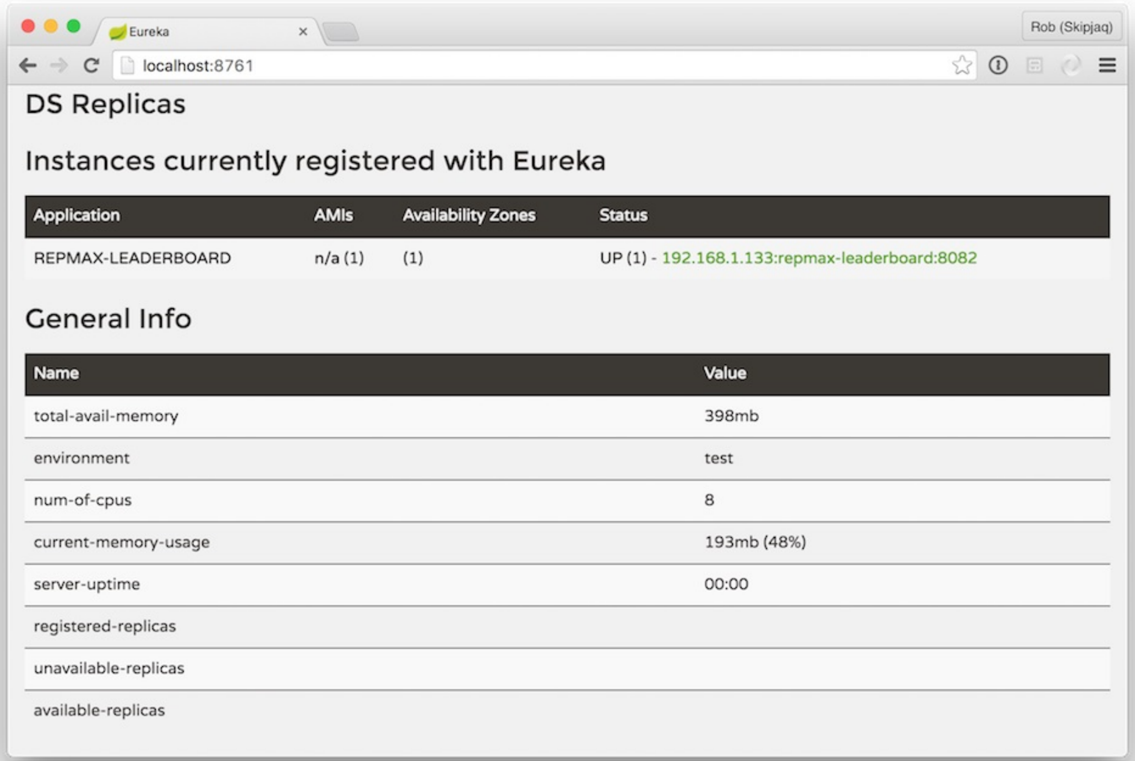
Application	AMIs	Availability Zones	Status
No instances available			

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. In `application.properties`:

```
spring.application.name=repmax-leaderboard
eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka
```

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



DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
REPMAX-LEADERBOARD	n/a (1)	(1)	UP (1) - 192.168.1.133:repmax-leaderboard:8082

General Info

Name	Value
total-avail-memory	398mb
environment	test
num-of-cpus	8
current-memory-usage	193mb (48%)
server-uptime	00:00
registered-replicas	
unavailable-replicas	
available-replicas	

Eureka dashboard after registration

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

With the Eureka Client enabled in the `logbook` service, Spring Cloud exposes a `DiscoveryClient` 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.

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".

- Personas
- **Architecture & Design**
- **Development**
- Topics
- Pivotal
- Netflix
- Spring Cloud
- Cloud
- Architecture
- Microservices

Related Editorial

Getting Started with Spring Cloud

Cloud-based Microservices powering BBC iPlayer

Automate Deployment & Management of Docker Cloud/Virtual Java Microservices with DCHQ

Securing Microservices with Spring Cloud Security

Spring Cloud Brixton.RELEASE Reaches General Availability

Hello stranger!

You need to Register an InfoQ account or Login or login to post comments. But there's so much more behind being registered.

Get the most out of the InfoQ experience.

Tell us what you think

Please enter a subject Message

Allowed html: a,b,br,blockquote,i,li,pre,u,ul,p

☐ Email me replies to any of my messages in this thread

Post Message

Community comments Watch Thread

Close

by

on

- View
- Reply
- Back to top

Close

Subject Your Reply Quote original message

Allowed html: a,b,br,blockquote,i,li,pre,u,ul,p

☐ Email me replies to any of my messages in this thread

Post Message Cancel

Close

Subject Your Reply

Allowed html: a,b,br,blockquote,i,li,pre,u,ul,p

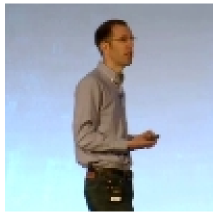
☐ Email me replies to any of my messages in this thread

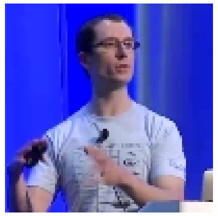
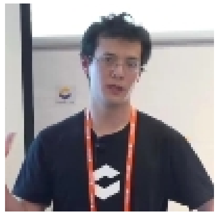
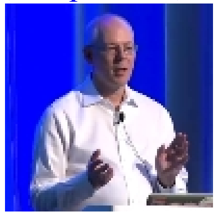
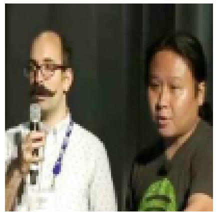
Cancel

Close

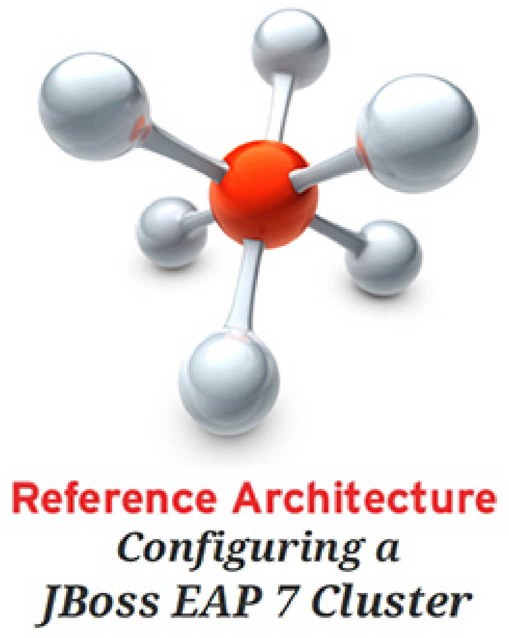
OK

RELATED CONTENT

- Cloud-based Microservices powering BBC iPlayer Jun 03, 2016 
- Automate Deployment & Management of Docker Cloud/Virtual Java Microservices with DCHQ Feb 23, 2016
- Peter Bourgon Discusses Coding in Idiomatic Go, Building Microservices with Go-kit, and Weave Net Jun 30, 2016
- AWS Adds Multi-Cloud Scripting to EC2 Run Command Feature Jul 12, 2016
- Samsung Acquires Cloud Provider Joyent Jul 06, 2016

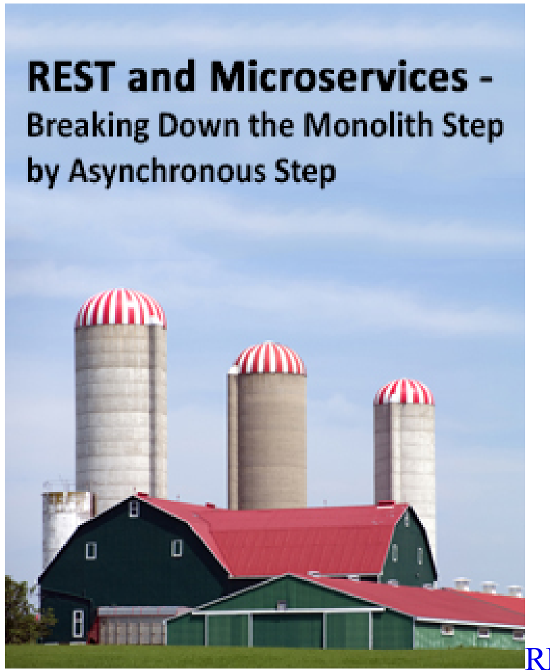
- The Seven (More) Deadly Sins of Microservices Jul 18, 2016 
- Lessons Learned on Uber's Journey into Microservices Jul 18, 2016
- Lessons Learned from the #api360 Microservices Summit 2016 Jun 24, 2016
- Pair Programming in the Cloud with Eclipse Che, Eclipse Flux, Orion, Eclipse IDE and Docker Jul 13, 2016 
- Microservices: State of the Union Jul 07, 2016 
- Handling Streaming Data in Spotify Using the Cloud Jul 06, 2016 

SPONSORED CONTENT

- 

[Reference Architecture: Configuring a JBoss EAP 7 Cluster]

This reference architecture stands up two JBoss EAP clusters as separate domains to eliminate downtime due to maintenance and upgrades. The goal is to provide a thorough description of the steps required for setup, while citing the rationale and explaining the alternatives at each decision juncture.

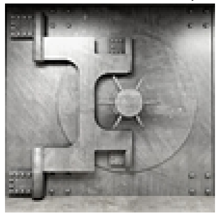
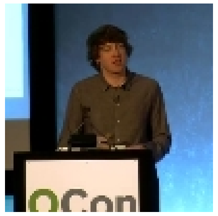
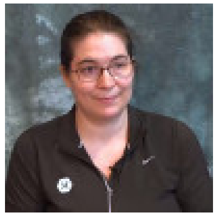
- 

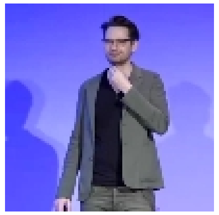
[REST and Microservices - Breaking Down the Monolith Step by Asynchronous Step]

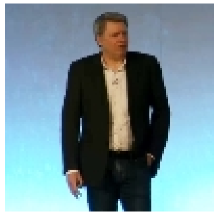
Mark Little explores the misuse and misunderstanding of REST (typically HTTP) for microservices.

Sponsored by RED HAT DEVELOPERS

RELATED CONTENT

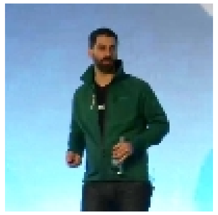
- Five Ways to Not Mess Up Microservices in Production Jun 19, 2016 
- Article Series: Cloud and "Lock-in" Jul 01, 2016
- Multi-Cloud is a Safety Belt for the Speed-Freaks Jun 24, 2016
- Virtual Panel on (Cloud) Lock-In Jun 17, 2016 
- Microsoft Cloud's Front Door: Building a Global API Jun 05, 2016 
- Rachel Reese on The Good and Bad of Microservices (with F#) Apr 28, 2016

- Microservices for a Streaming World May 02, 2016 
- Streaming Auto-scaling in Google Cloud Dataflow May 02, 2016

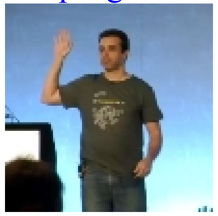
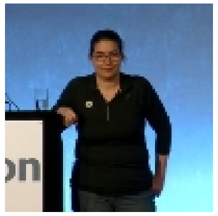
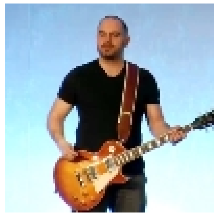
- Code in the Cloud with Eclipse Che and Docker Apr 28, 2016
- The InfoQ Podcast: Adrian Cockcroft on Microservices, Terraservices and Serverless Computing Apr 22, 2016
- DDD and Microservices: At Last, Some Boundaries! Apr 16, 2016 

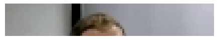
QCon
SAN FRANCISCO 2016
November 7 - 11
International Software Development Conference
SAVE $700 Before Jul 30th

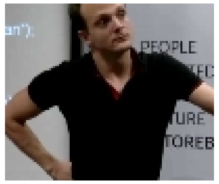
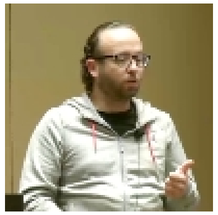
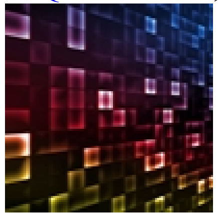
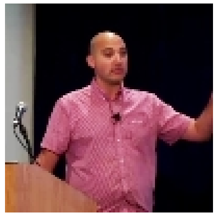
RELATED CONTENT

- Microservices Antipatterns Apr 16, 2016 

- Developing Cloud-native Applications with Eclipse and the Spring Tool Suite Apr 11, 2016

- The Microservices and DevOps Journey Apr 09, 2016 
- Microservices Chaos Testing at Jet Apr 01, 2016 
- Test-Driven Microservices: System Confidence Mar 24, 2016 
- Microservices Evolution at SoundCloud Mar 21, 2016
- Microservices to FastData in the Enterprise with Spring Mar 12, 2016



- The Simple Life of ReSTful Microservices Mar 11, 2016 
- Building Microservices with Event Sourcing and CQRS Mar 05, 2016 
- Microservices in the Real World Jan 27, 2016 
- Bind to the Cloud with Falcor Feb 01, 2016 

InfoQ Weekly Newsletter

Subscribe to our Weekly email newsletter to follow all new content on InfoQ

Your email here Subscribe

Development

The Nihilist's Guide to Wrecking Humans & Systems

Modern iOS Application Security

Facebook Announces Open-Source Swift SDK Beta for iOS

Architecture & Design

The Seven (More) Deadly Sins of Microservices

Lessons Learned on Uber's Journey into Microservices

Basho Open Sources Time Series Database Riak TS 1.3

Culture & Methods

Culture Eats Principles for Breakfast

Full Stack Kanban - Live

Making a Sandwich: Effective Feedback Techniques

Data Science

Creating Customer-centric Products Using Big Data

HTTP-RPC: A Lightweight Cross-Platform REST Framework

Key Takeaway Points and Lessons Learned from QCon New York 2016

DevOps

The Nihilist's Guide to Wrecking Humans & Systems

Fighting the #fintech Wave with DevOps

InfoQ eMag: Exploring Container Technology in the Real World

- Home
- All topics

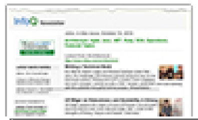
- QCon Conferences
- About InfoQ
- Our Audience
- Contribute
- About C4Media
- Create account
- Login

- **QCons Worldwide**
- Shanghai
  Oct 20-22, 2016
- San Francisco
  Nov 7-11, 2016
- Tokyo 2016
- London
  Mar 6-10, 2017
- Beijing
  Apr 16-18, 2017
- São Paulo
  Apr 24-26, 2017
- New York
  Jun 26-30, 2017

InfoQ Weekly Newsletter

Subscribe to our Weekly email newsletter to follow all new content on InfoQ

Click to view
an example



Your email here Subscribe

- Your personalized RSS
- For daily content and announcements
- For major community updates
- For weekly community updates

Personalize Your Main Interests

- ✓ Development
- ✓ Architecture & Design
- ✓ Data Science
- ✓ Culture & Methods
- ✓ DevOps

This affects what content you see on the homepage & your RSS feed. Click preferences to access more fine-grained personalization.

General Feedback	Bugs	Advertising	Editorial	Marketing
feedback@infoq.com	bugs@infoq.com	sales@infoq.com	editors@infoq.com	marketing@infoq.com

BT