Kenny Bastani (http://www.kennybastani.com/)

Kenny Bastani lives in Silicon Valley and builds really cool things with graphs and microservices. This is where he talks about it.

Blog (http://www.kennybastani.com/) | GitHub (http://www.github.com/kbastani)
Twitter (http://www.twitter.com/kennybastani)
LinkedIn (http://www.linkedin.com/in/kennybastani)

Building Microservices with Spring Cloud and Docker

Sunday, July 12, 2015

This blog series will introduce you to some of the foundational concepts of building a microservice-based platform using Spring Cloud and Docker.

What is Spring Cloud?

Spring Cloud (http://projects.spring.io/spring-cloud/) is a collection of tools from Pivotal (https://pivotal.io/) that provides solutions to some of the commonly encountered patterns when building distributed systems. If you're familiar with building applications with Spring Framework (https://en.wikipedia.org/wiki/Spring_Framework), Spring Cloud builds upon some of its common building blocks.

Among the solutions provided by Spring Cloud, you will find tools for the following problems:

- Configuration management (http://12factor.net/config)
- Service discovery (https://en.wikipedia.org/wiki/Service_discovery)
- Circuit breakers (http://martinfowler.com/bliki/CircuitBreaker.html)
- Distributed sessions (https://en.wikipedia.org/wiki/Distributed_cache)

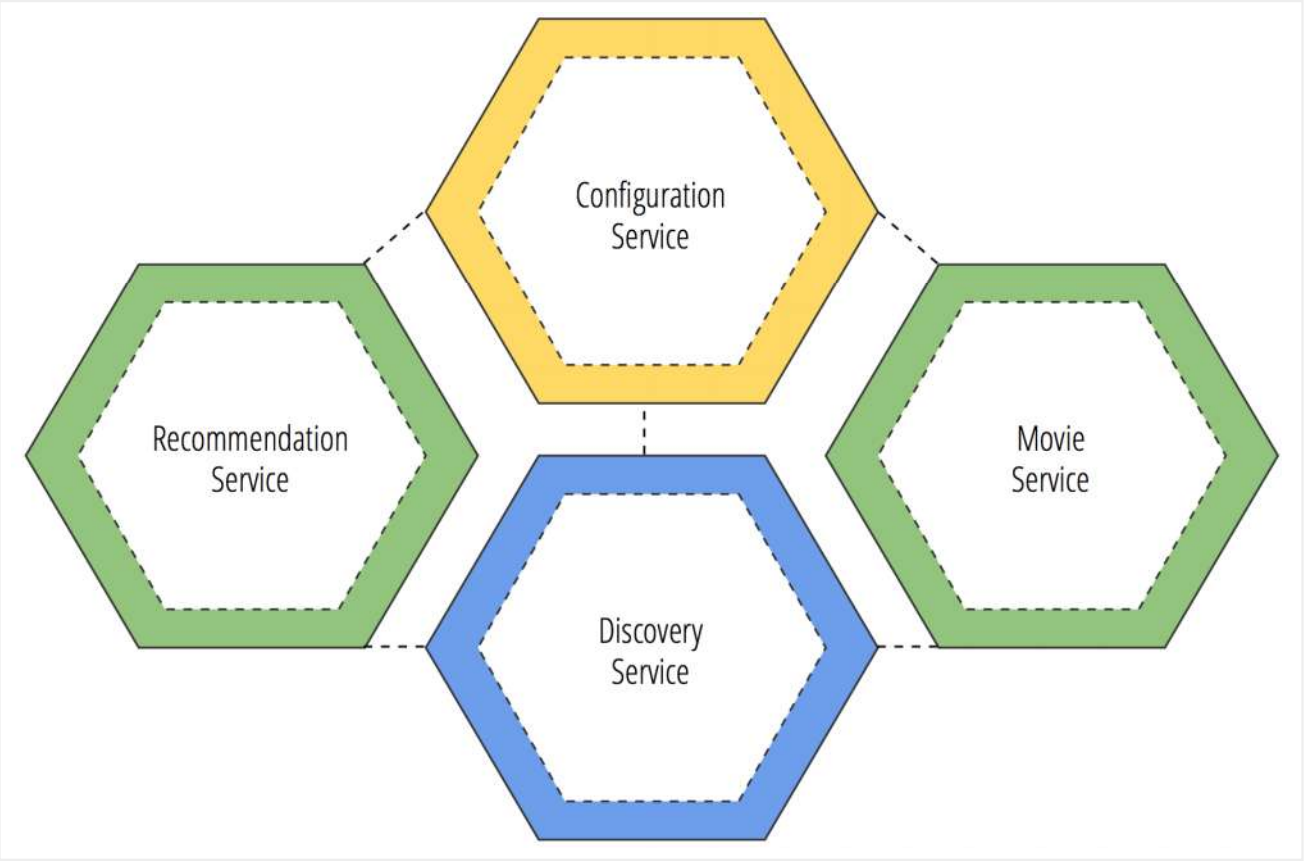
Spring Boot

The great part about working with Spring Cloud is that it builds on the concepts of Spring Boot (http://projects.spring.io/spring-boot/).

For those of you who are new to Spring Boot, the name of the project means exactly what it says. You get all of the best things of the Spring Framework and ecosystem of projects, tuned to perfection, with minimal configuration, all ready for production.

Service Discovery and Intelligent Routing

Each service has a dedicated purpose in a microservices architecture. When building a microservices architecture on Spring Cloud, there are a few primary concerns to deal with first. The first two microservices you will want to create are the **Configuration Service**, and the **Discovery Service**.



The graphic above illustrates a 4-microservice setup, with the connections between them indicating a dependency.

The configuration service sits at the top, in yellow, and is depended on by the other microservices. The discovery service sits at the bottom, in blue, and also is depended upon by the other microservices.

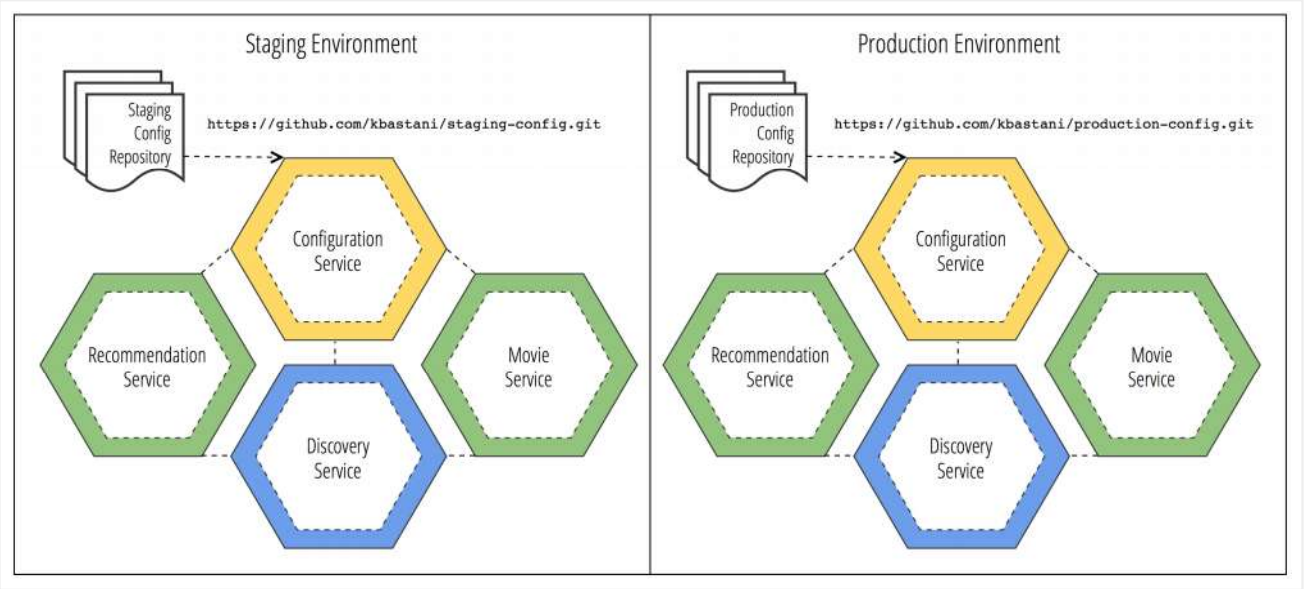
In green, we have two microservices that deal with a part of the domain of the example application I will use throughout this blog series: movies and recommendations.

Configuration Service

The configuration service is a vital component of any microservices architecture. Based on the **twelve-factor app** (http://12factor.net/config) methodology, configurations for your microservice applications should be stored in the environment and not in the project.

The configuration service is essential because it handles the configurations for all of the services through a simple point-to-point service call to retrieve those configurations. The advantages of this are multi-purpose.

Let's assume that we have multiple deployment environments. If we have a staging environment and a production environment, configurations for those environments will be different. A configuration service might have a dedicated Git repository for the configurations of that environment. None of the other environments will be able to access this configuration, it is available only to the configuration service running in that environment.



When the configuration service starts up, it will reference the path to those configuration files and begin to serve them up to the microservices that request those configurations. Each microservice can have their configuration file configured to the specifics of the environment that it is running in. In doing this, the configuration is both externalized and centralized in one place that can be version-controlled and revised without having to restart a service to change a configuration.

With management endpoints available from Spring Cloud, you can make a configuration change in the environment and signal a refresh to the discovery service that will force all consumers to fetch the new configurations.

Discovery Service

The discovery service is another vital component of our microservice architecture. The discovery service handles maintaining a list of service instances that are available for work within a cluster. Within applications, service-to-service calls are made using clients. For this example project, I used Spring Cloud Feign (https://github.com/spring-cloud-samples/feign-eureka), a client-based API for RESTful microservices that originated from the Netflix OSS project (https://github.com/spring-cloud-samples/feign-eureka).

```
@FeignClient("movie")
public interface MovieClient {
    @RequestMapping(method = RequestMethod.GET, value = "/movies")
    PagedResources findAll();

    @RequestMapping(method = RequestMethod.GET, value = "/movies/{id}")
    Movie findById(@RequestParam("id") String id);

    @RequestMapping(method = RequestMethod.POST, value = "/movies",
      produces = MediaType.APPLICATION_JSON_VALUE)
    void createMovie(@RequestBody Movie movie);
}
```

In the code example above, I am creating a Feign client that maps to the REST API methods that are exposed by the movie service. Using the `@FeignClient` annotation, I first specify that I want to create a client API for the `movie` microservice. Next I specify the mappings of the service that I want to consume. I do this by declaring a URL pattern over the methods that describes a route for a REST API.

The wonderfully easy part of creating Feign clients is that all I need to know is the ID of the service that I would like to create a client on. The URL of the service is automatically configured at runtime because each microservice in the cluster will register with the discovery service with its `serviceId` at startup.

The same is true for all other services of my microservice architecture. All I need to know is the `serviceId` of the service I want to communicate with, and everything else will be autowired by Spring.

API Gateway

The API gateway service is another vital component if we are going to create a cluster of services managing their own domain entities. The green hexagons below are our data-driven services that manage their own domain entities and even their own databases. By adding an API gateway service, we can create a proxy of each API route that are exposed by the green services.

Let's assume that both the recommendation service and the movie service expose their own REST API over the domain entities that they manage. The API gateway will discover these services through the discovery service and inject a proxy-based route of the API methods from the other services. In this way, both the recommendation and movie microservice will have a full definition of routes available locally from all the microservices that expose a REST API. The API Gateway will re-route the request to the service instances that own the route being requested through HTTP.

Example Project

I've put together an example project (https://github.com/kbastani/spring-cloud-microservice-example) that demonstrates an end-to-end cloud-native platform using Spring Cloud for building a practical microservices architecture.

Demonstrated concepts:

- Integration testing using Docker
- Polyglot persistence
- Microservice architecture
- Service discovery
- API gateway

Docker

Each service is built and deployed using Docker. End-to-end integration testing can be done on a developer's machine using Docker compose.

Polyglot Persistence

One of the core concepts of this example project is how polyglot persistence can be approached in practice. Microservices in the project use their own database while integrating with the data from other services through REST or a message bus. For example, you could have a microservice for each of the following databases.

- Neo4j (graph)
- MongoDB (document)
- MySQL (relational)

Microservice architecture

This example project demonstrates how to build a new application using microservices, as opposed to a monolith-first strategy. Since each microservice in the project is a module of a single parent project, developers have the advantage of being able to run and develop with each microservice

running on their local machine. Adding a new microservice is easy, as the discovery microservice will automatically discover new services running on the cluster.

Service discovery

This project contains two discovery services, one on Netflix Eureka (https://github.com/Netflix/eureka), and the other uses Consul from Hashicorp (https://consul.io/). Having multiple discovery services provides the opportunity to use one (Consul) as a DNS provider for the cluster, and the other (Eureka) as a proxy-based API gateway.

API gateway

Each microservice will coordinate with Eureka to retrieve API routes for the entire cluster. Using this strategy each microservice in a cluster can be load balanced and exposed through one API gateway. Each service will automatically discover and route API requests to the service that owns the route. This proxying technique is equally helpful when developing user interfaces, as the full API of the platform is available through its own host as a proxy.

Docker Demo

The example project uses Docker to build a container image of each of our microservices as a part of the Maven build process. We can easily orchestrate the full microservice cluster on our own machine using Docker compose.

Getting Started

To get started, visit the GitHub repository for this example project.

```
https://github.com/kbastani/spring-cloud-microservice-example (https://github.com/
kbastani/spring-cloud-microservice-example)
```

Clone or fork the project and download the repository to your machine. After downloading, you will need to use both Maven and Docker to compile and build the images locally.

Download Docker

First, download Docker if you haven't already. Follow the instructions found here (https://docs.docker.com/installation/), to get Docker up and running on your development machine.

You will also need to install Docker Compose (https://docs.docker.com/compose/), the installation guide can be found here (https://docs.docker.com/compose/install/). If you are using Mac OSX and boot2docker, make sure that you provision the boot2docker-vm on VirtualBox with at least 5GB of memory. The following command will allow you to do this.

```
$ boot2docker init --memory=5000
```

Requirements

The requirements for running this demo on your machine are found below.

- Maven 3
- Java 8
- Docker
- Docker Compose

Building the project

To build the project, from the terminal, run the following command at the root of the project.

```
$ mvn clean install
```

The project will then download all of the needed dependencies and compile each of the project artifacts. Each service will be built, and then a Maven Docker plugin will automatically build each of the images into your local Docker registry. Docker must be running and available from the command line where you run the `mvn clean install` command for the build to succeed.

After the project successfully builds, you'll see the following output:

```
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] spring-cloud-microservice-example-parent ..... SUCCESS [  0.268 s]
[INFO] users-microservice ............................ SUCCESS [ 11.929 s]
[INFO] discovery-microservice ........................ SUCCESS [  5.640 s]
[INFO] api-gateway-microservice ...................... SUCCESS [  5.156 s]
[INFO] recommendation-microservice ................... SUCCESS [  7.732 s]
[INFO] config-microservice ........................... SUCCESS [  4.711 s]
[INFO] hystrix-dashboard ............................. SUCCESS [  4.251 s]
[INFO] consul-microservice ........................... SUCCESS [  6.763 s]
[INFO] movie-microservice ............................ SUCCESS [  8.359 s]
[INFO] movies-ui ..................................... SUCCESS [ 15.833 s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

Start the Cluster with Docker Compose

Now that each of the images has been built successfully, we can using Docker Compose to spin up our cluster. I've included a pre-configured Docker Compose yaml file with the project.

From the project root, navigate to the `spring-cloud-microservice-example/docker` directory.

Now, to startup the microservice cluster, run the following command:

```
$ docker-compose up
```

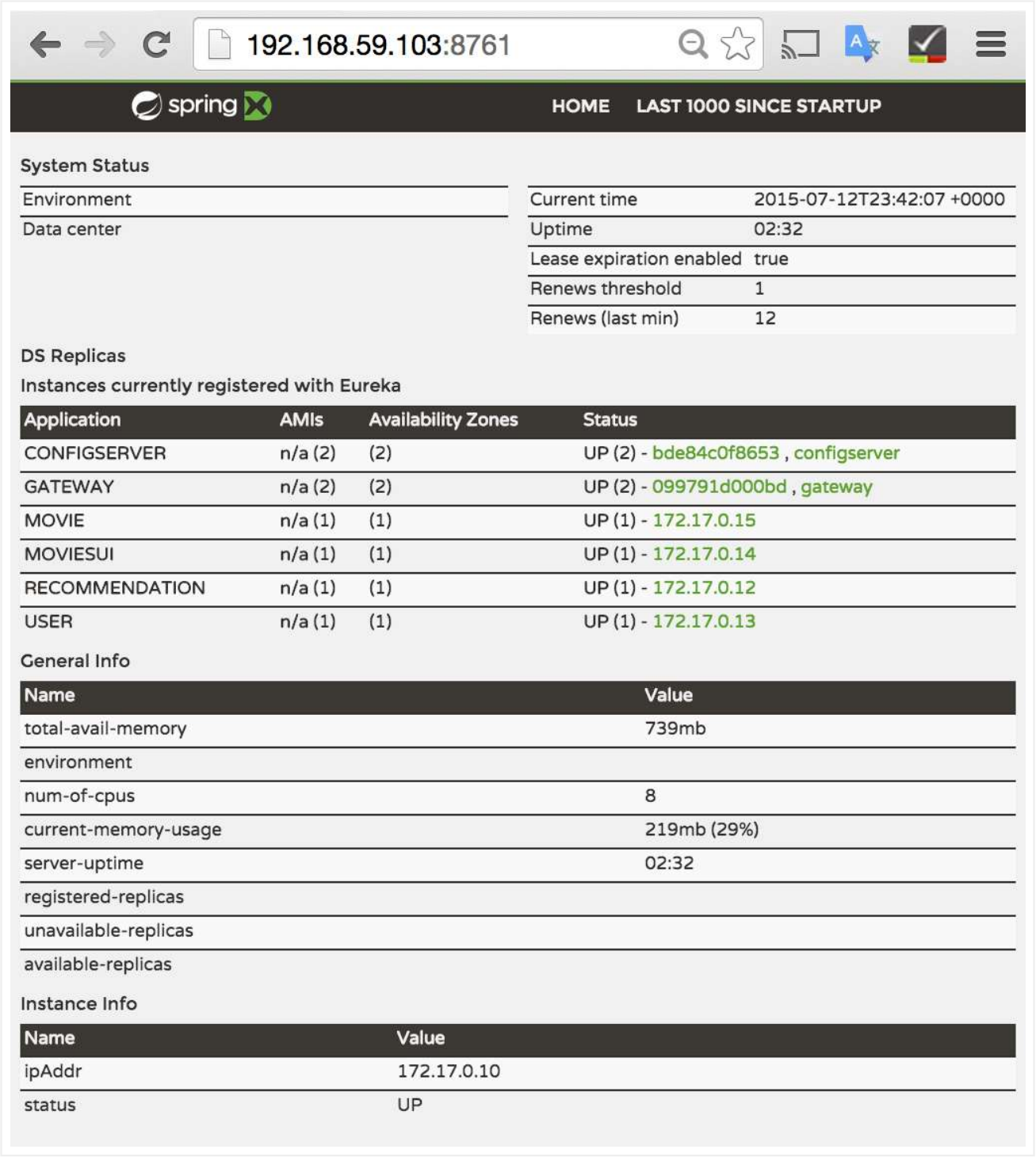
If everything is configured correctly, each of the container images we built earlier will be launched within their own VM container on Docker and networked for automatic service discovery. You will see a flurry of log output from each of the services as they begin their startup sequence. This might take a few minutes to complete, depending on the performance of the machine you're running this demo on.

Once the startup sequence is completed, you can navigate to the Eureka host and see which services have registered with the discovery service.

Copy and paste the following command into the terminal where Docker can be accessed using the `$DOCKER_HOST` environment variable.

```
$ open $(echo \"$(echo $DOCKER_HOST)\"|
            \sed 's/tcp:\/\//http:\/\//g'|
            \sed 's/[0-9]\{4,\}/8761/g'|
            \sed 's/\"//g')
```

If Eureka correctly started up, a browser window will open to the location of the Eureka service's dashboard, as shown below.



We can see each of the service instances that are running and their status. We can then access one of the data-driven services, for example the `movie` service.

```
$ open $(echo \"$(echo $DOCKER_HOST)/movie\"|
            \sed 's/tcp:\/\//http:\/\//g'|
            \sed 's/[0-9]\{4,\}/10000/g'|
            \sed 's/\"//g')
```

This command will navigate to the API gateway's endpoint and proxy to the movie service's REST API endpoints. These REST APIs have been configured to use HATEOAS (https://en.wikipedia.org/wiki/HATEOAS), which supports the auto-discovery of all of the service's functionality as embedded links.

```
{
  "_links" : {
    "self" : {
      "href" : "http://192.168.59.103:10000/movie"
    },
    "resume" : {
      "href" : "http://192.168.59.103:10000/movie/resume"
    },
    "pause" : {
      "href" : "http://192.168.59.103:10000/movie/pause"
    },
    "restart" : {
      "href" : "http://192.168.59.103:10000/movie/restart"
    },
    "metrics" : {
      "href" : "http://192.168.59.103:10000/movie/metrics"
    },
    "env" : [ {
      "href" : "http://192.168.59.103:10000/movie/env"
    }, {
      "href" : "http://192.168.59.103:10000/movie/env"
    } ],
    "archaius" : {
      "href" : "http://192.168.59.103:10000/movie/archaius"
    },
    "beans" : {
      "href" : "http://192.168.59.103:10000/movie/beans"
    },
    "configprops" : {
      "href" : "http://192.168.59.103:10000/movie/configprops"
    },
    "trace" : {
```

```
      "href" : "http://192.168.59.103:10000/movie/trace"
    },
    "info" : {
      "href" : "http://192.168.59.103:10000/movie/info"
    },
    "health" : {
      "href" : "http://192.168.59.103:10000/movie/health"
    },
    "hystrix.stream" : {
      "href" : "http://192.168.59.103:10000/movie/hystrix.stream"
    },
    "routes" : {
      "href" : "http://192.168.59.103:10000/movie/routes"
    },
    "dump" : {
      "href" : "http://192.168.59.103:10000/movie/dump"
    },
    "refresh" : {
      "href" : "http://192.168.59.103:10000/movie/refresh"
    },
    "mappings" : {
      "href" : "http://192.168.59.103:10000/movie/mappings"
    },
    "autoconfig" : {
      "href" : "http://192.168.59.103:10000/movie/autoconfig"
    }
  }
}
```

Conclusion

This has been the first part in a multi-part series about building microservice architectures with Spring Cloud and Docker. In this blog post, we went over the following concepts:

- Service Discovery
- Externalized Configuration
- API Gateway
- Service Orchestration with Docker Compose

In the next blog post, we will go over how to build application front-ends that integrate with our backend services. We will also take a look at a use case for polyglot persistence (http://martinfowler.com/bliki/PolyglotPersistence.html), using both MySQL (relational) and Neo4j (graph).

Special thanks

I want to give special thanks to Josh Long (http://www.twitter.com/starbuxman) and the rest of the Spring team for giving me the chance to learn first-hand about the many wonderful things that the Spring Framework has to offer. Without Josh's mentorship I would not be able to put into words all of the amazing things that the Spring ecosystem has to offer.

Many of the great open source tools, like Spring Cloud, wouldn't be possible without the thought leadership from people like Adrian Cockcroft (https://twitter.com/adrianco) (Netflix OSS), Martin Fowler (https://twitter.com/martinfowler) (everything), Sam Newman (https://twitter.com/samnewman) (O'Reilly's Building Microservices (http://shop.oreilly.com/product/0636920033158.do)), Ian Robinson (https://twitter.com/iansrobinson) (consumer driven contracts), Chris Richardson (https://twitter.com/crichardson) (Cloud Foundry) and the many others who have helped to make the world of open source software what it is today.

Top comments

Ronald Kurr shared this 1 month ago - Spring Framework (Tutorials)

+22 1

Kent Johnson via Google+ 1 month ago - Shared publicly

I am looking forward to getting this up and running. I want to see if I can swap out Maven for Gra

+1 1 · Reply

Claus Straube shared this via Google+ 1 month ago - Shared publicly

+1 1 · Reply

Daniel Bartl shared this via Google+ 1 month ago - Shared publicly

Ronald Kurr originally shared this

+1 1 · Reply

Shoaib Akhtar shared this via Google+ 1 month ago - Shared publicly

Ronald Kurr originally shared this

1 · Reply

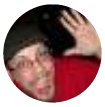
Oleg Soroka shared this via Google+ 2 days ago - Shared publicly

+2 1

VonVictor Valentino Rosenchild shared this via Google+ 2 days ago - Shared publicly

Oleg Soroka originally shared this

+1 1 · Reply



Andrew Bell 1 month ago - Shared publicly

I was really excited to find this article, as I'm trying to do the same sort of thing, and this is so cor

I do have one question though, how do services know to register with Euerka with their external I their docker container)? I couldn't tell from the blog, or the code, so a helpful nudge would be app

+1 1 · Reply

Kenny Bastani 1 month ago

Hi Andrew. The discovery service listens for pings from other services in the cluster. Wher loading procedures is to check in with the discovery service. It's kind of like Foursquare bu

Andrew Bell 1 month ago

Okay, so it's not the client sending it's IP, but rather the discovery service registering what does the trick, thanks for the reply, I'll keep an eye on your blog in the future!

Matt Reynolds 1 month ago - Shared publicly

One issue you may run into is the clients trying to contact the config server before it's available a (or stopping if you have failFast set). This becomes a real issue if you are setup for "Eureka first" and the clients get to it via Eureka. The solution is to use config client retry (http://cloud.spring.i config.html#config-client-retry)

+1 1 · Reply

Kenny Bastani 1 month ago

Thanks Matt

Michal Pavlasek shared this via Google+ 1 month ago - Shared publicly

+1 1 · Reply

Sultan Mahmood shared this via Google+ 1 month ago - Shared publicly

+3 1 · Reply

Sultan Mahmood 1 month ago *+1*

#docker

Marcel Körtgen via Google+ 1 month ago - Shared publicly

Kenny Bastani on Microservices with Docker. Reading tip.

+1 1

Marcel Körtgen 1 month ago *+1*

Have a look at neo4j-mazerunner as well: http://www.kennybastani.com/2014/11/using-a

George Kuznetsov shared this via Google+ 1 month ago - Shared publicly

+1 1 Reply

Kenny Bastani shared this via Google+ 1 month ago - Shared publicly

+2 1 · Reply

허태명 via Google+ 1 month ago - Shared publicly

Building Microservices with Spring Cloud and Docker

+1 1 · Reply

Paul Tian shared this via Google+ 1 month ago - Shared publicly

+1 1 · Reply

Manuel Peguero 1 month ago - Shared publicly

Very good...

+1 1 · Reply

Kenny Bastani via Google+ 1 month ago - Shared publicly

Building Microservices with Spring Cloud and Docker

This blog series will introduce you to some of the foundational concepts of building a microservi
and Docker. What is Spring Cloud? Spring Cloud is a collection of tools from Pivotal that provides

1 · Reply

Nazeel A K 1 month ago - Shared publicly

Spring-Cloud - introduction is simple and awesome

+1 1 · Reply

Shoaib Akhtar shared this via Google+ 1 month ago - Shared publicly

Sultan Mahmood originally shared this

+1 1 · Reply

Show more

Back (http://www.kennybastani.com/2015/07/spring-cloud-docker-microservices.html) Next (http://www.kennybastani.com/2015/07/hystrix-dashboard-spring-cloud.html)

Subscribe to: Post Comments (Atom)
(http://www.kennybastani.com/feeds/8368851254535781851/comments/default)

G+1 +22 Recommend this on Google

Author's Google+

Kenny Bastani

G+ Follow

110 followers

@kennybastani

Tweets Follow

Bridget Kromhout @bridgetkromhout 6h
Team @pivotal #cloudfoundry (yours truly & @rippmn) hanging out at the @DevOpsMSP meetup! pic.twitter.com/zy0FPgPCJi
Retweeted by Kenny Bastani

Expand

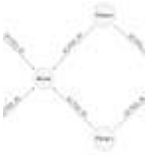
Neo Technology @NeoTechnology 7h
.@starbuxman of @pivotal + @kennybastani will present on #Microservices, #SpringCloud & #Neo4j buff.ly/1EAoB6e pic.twitter.com/3iJPFbz8UI

Tweet to @kennybastani

Follow by Email

Email address...

Popular Posts



(http://www.kennybastani.com/2014/11/using-apache-spark-and-neo4j-for-big.html)

Using Apache Spark and Neo4j for Big Data Graph Analytics (http://www.kennybastani.com/2014/11/using-apache-spark-and-neo4j-for-big.html)

As engineers, when we think about how to solve big data problems, evaluating technologies becomes a choice between scalable and not scalable...

Building Microservices with Spring Cloud and Docker (http://www.kennybastani.com/2015/07/spring-cloud-docker-microservices.html)

This blog series will introduce you to some of the foundational concepts of building a microservice-based platform using Spring Cloud a...

(http://www.kennybastani.com/2014/09/deep-learning-sentiment-analysis-for.html)

Deep Learning Sentiment Analysis for Movie Reviews using Neo4j (http://www.kennybastani.com/2014/09/deep-learning-sentiment-analysis-for.html)

While the title of this article references Deep Learning, it's important to note that the process described below is more of a deep lear...

Using a Graph Database for Deep Learning Text Classification (http://www.kennybastani.com/2014/08/using-graph-database-for-deep-learning-text-classification.html)

Graphify is a Neo4j unmanaged extension that provides plug and play natural language text classification . Graphify gives you a mechanis...

Building a Neo4j Reporting Service Part I (http://www.kennybastani.com/2014/04/building-graph-based-analytics-platform-part-1.html)

Data science is pretty hot right now. The obvious reason is that data is rapidly expanding in complexity and size. There is an opportunity t...

Blog Archive

▼ 2015 (http://www.kennybastani.com/search?updated-min=2015-01-01T00:00:00-