[](http://zoltanaltfatter.com/)

[Zoltan Altfatter](http://zoltanaltfatter.com/)

Software Engineer

[Blog](http://zoltanaltfatter.com/) [About](http://zoltanaltfatter.com/about)

**Centralized configuration with Spring Cloud Config**

*June 30, 2016*

In this blog post I am looking into [Spring Cloud Config](https://cloud.spring.io/spring-cloud-config/) project which is a great tool for centralized configuration management in a microservice environment. I have prepared a simple example project which you can find on my [github](https://github.com/altfatterz/spring-cloud-config-example) account. It contains a config-service leveraging *Spring Cloud Config Server* and three little services using *Spring Cloud Config Client*.

Clone the configuration and example repositories into your home directory and build the project.

$ git clone https://github.com/altfatterz/spring-cloud-config-example-repo

$ git clone https://github.com/altfatterz/spring-cloud-config-example

$ cd spring-cloud-config-example

$ mvn clean install

Before starting the config-service you need to start rabbitmq in another terminal. More on this later why it is needed for this example.

java -Dsecurity.user.name=config \

-Dsecurity.user.password=verysecure \

-Dspring.cloud.config.server.git.uri=${HOME}/spring-cloud-config-example-repo \

-Dencrypt.key=foobarbaz \

-jar config-service/target/config-service-0.0.1-SNAPSHOT.jar

The config-service is protected with HTTP Basic security and the credentials are set using the system properties. Using the spring.cloud.config.server.git.uri we tell where it can find the git repository with the externalized configurations. With the encrypt.key we set a symmetric key which is used for decrypting property values which were encrypted (values starting with *{cipher}*).

After the config-service started successfully request the following resource from the config-service:

$ curl http://config:verysecure@localhost:8888/foo.service/default

The resource follows the following format /{application}/{profile}[/{label}]. The label defaults to master. The request is handled by [ResourceController](https://github.com/spring-cloud/spring-cloud-config/blob/master/spring-cloud-config-server/src/main/java/org/springframework/cloud/config/server/resource/ResourceController.java) from *Spring Cloud Config Server*, which was added using the @EnableConfigServer annotation.

It will return the following response:

{

"name": "foo-service",

"profiles": [

"default"

],

"label": null,

"version": "aec3d698ed6d4d4d006a6ffe95a775340a05c0c7",

"propertySources": [

{

"name": "/Users/Zoltan/spring-cloud-config-example-repo/foo-service.properties",

"source": {

"info.app.name": "Foo Service",

"secret-message": "a702d2b5b0c6bc2db67e7d487c6142e7c23254108503d1856ff516d0a64bbd3663a2514a86647dcf8467d042abcb8a6e",

"server.port": "8081"

}

},

{

"name": "/Users/Zoltan/spring-cloud-config-example-repo/application.properties",

"source": {

"message": "Ria, Ria, Hungaria!"

}

}

]

}

As you can see from the response the foo-service should start on port 8081 and has a message property as well as a secret-message property. Next, start up the foo-service using the command:

$ java -Dspring.cloud.config.uri=http://config:verysecure@localhost:8888 \

-jar foo-service/target/foo-service-0.0.1-SNAPSHOT.jar

With the spring.cloud.config.uri we tell the foo-service where it can find the config-service. After the foo-service started successfully we can check the visible properties for the for-service via the /env Spring Boot Actuator endpoint.

$ http://localhost:8081/env

{

...

"configService":"/Users/Zoltan/projects/personal/spring-cloud-config-example-repo/foo-service.properties": {

"server.port": 8081,

"info.app.name": "Foo Service",

"secret-message": "Spring Cloud Rocks!"

}

"configService":"/Users/Zoltan/projects/personal/spring-cloud-config-example-repo/application.properties": {

"message": "Ria, Ria, Hungaria!"

},

...

}

As you can see in the response the secret-message was decrypted the other properties where also retrieved from the *config-service*.

**RefreshScope**

*Spring Cloud Config Client* has a nice feature that it can re-initialize Spring beans when configuration changes using the @RefreshScope annotation. In oder to try it let’s change the message property in the application.properties in your local spring-cloud-config-example-repo repository and commit the changes.

You will see that the config-service already sees the change when accessing the

curl http://localhost:8888/foo.service/default

however the foo-service does not.

In order to signal to foo-service to re-initialize the beans annotated with @RefreshScope you need to send an empty body request to /refresh endpoint.

$ curl -d{} http://localhost:8081/refresh

Note also that the re-initialization occurs lazily when they are used and not at the moment of handling the /refresh request.

**Spring Cloud Config Monitor**

What if you have many services which all need to re-initialize beans when external configuration changes? Instead of calling /refresh endpoint for each service there is a /monitor endpoint which is enabled when including the *spring-cloud-config-monitor* module as a dependency to the *config-service*. It is using [Spring Cloud Bus](http://cloud.spring.io/spring-cloud-bus/) to broadcast the change events. In order to work we need also a transport. In this example we chose RabbitMQ by including *spring-cloud-starter-bus-amqp* module dependency to the *config-service* and all other services. In order to try it let’s start the other two services as well in separate terminals:

$ java -Dspring.cloud.config.uri=http://config:verysecure@localhost:8888 \

-jar bar-service/target/bar-service-0.0.1-SNAPSHOT.jar

$ java -Dspring.cloud.config.uri=http://config:verysecure@localhost:8888 \

-jar baz-service/target/baz-service-0.0.1-SNAPSHOT.jar

After making sure that the other two services have started successfully, change again the message property value in the application.properties in your local spring-cloud-config-example-repo repository and commit the changes. And after that issue the following request:

$ curl http://config:verysecure@localhost:8888/monitor -d path="\*"

You will see in the logs of the services:

o.s.cloud.bus.event.RefreshListener : Received remote refresh request. Keys refreshed [message]

And indeed all the services see the updated value of the message property. You can also easily verify it by requesting the custom /message endpoint.

**Push notification**

Going further, wouldn’t it be nice if we do not need to call the /monitor endpoint at all when external configuration is changed? *Spring Cloud Config* provides support for this using the default git storage backend. Many git repository providers can notify you of changes in the repository through a webhook. Since the external repository is hosted on my github account, let’s configure a webhook with GitHub. For this we need that our config-sevice is accessible on the internet. We use ngrok for this, which a very cool lightweight tool that creates a secure tunnel on your local machine together with a public URL. When ngrok is running, it listens on the same port your config-service is running (port 8888 in this case) and proxies external requests to your local machine. Download ngrok from [here](https://ngrok.com/), unzip it to your Applications folder and create a symlink to it, this will allow you to run the ngrok command from any directory while in the terminal.

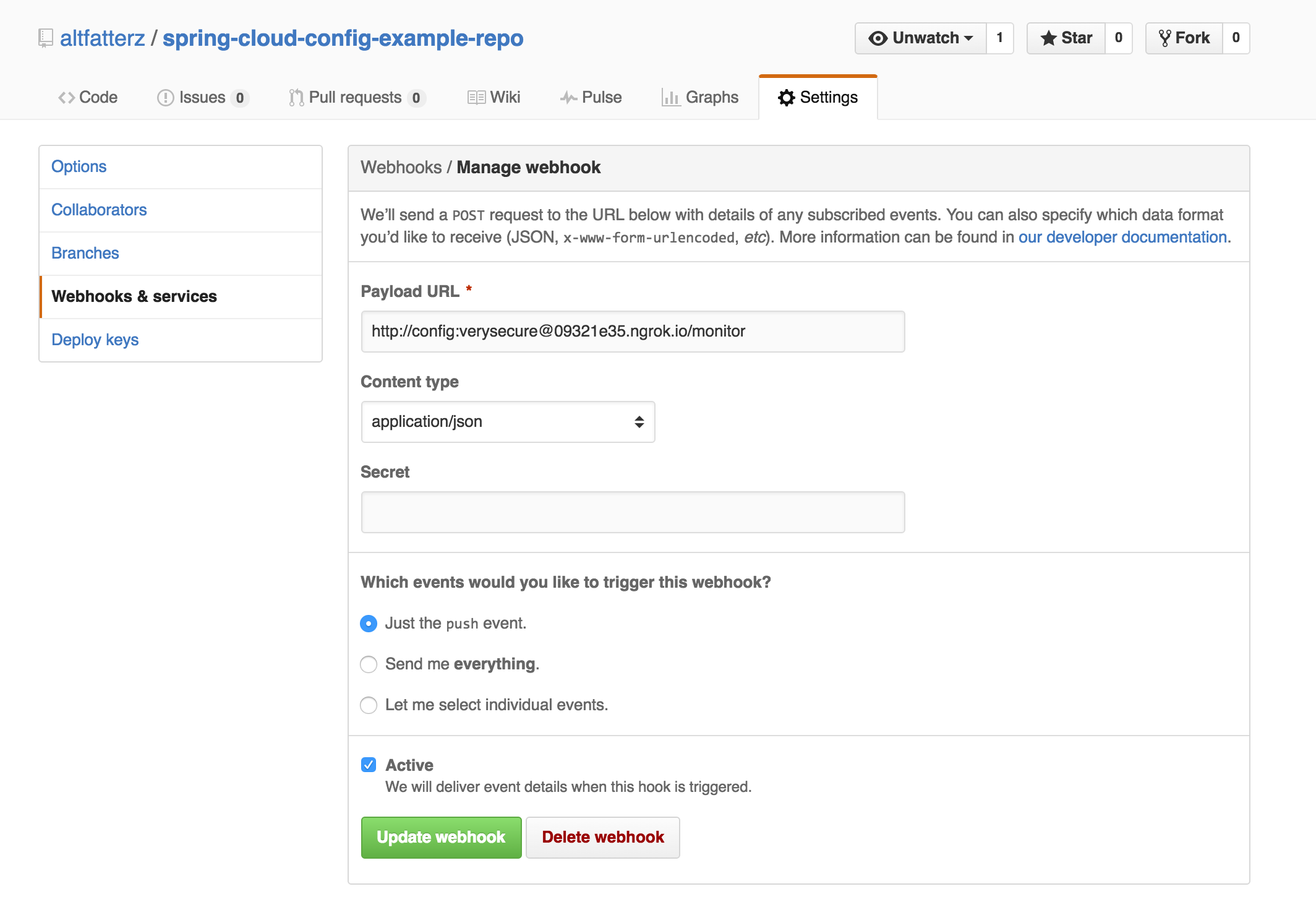
$ cd /usr/local/bin

$ ln -s /Applications/ngrok ngrok

Then start ngrok like this

$ ngrok http 8888

It will create a public URL which we can use to setup our webhook in Github like this:



We need to an activate the webhook functionality by setting the spring.cloud.config.server.monitor.github.enabled to true, and the spring.cloud.config.server.git.uri to the github url and restart the config-service

java -Dsecurity.user.name=config \

-Dsecurity.user.password=verysecure \

-Dspring.cloud.config.server.monitor.github.enabled=true \

-Dspring.cloud.config.server.git.uri=https://github.com/altfatterz/spring-cloud-config-example-repo \

-Dencrypt.key=foobarbaz \

-jar config-service/target/config-service-0.0.1-SNAPSHOT.jar

And again let’s change the message property in the application.properties in your local spring-cloud-config-example-repo commit the changes and push it up to the origin. Soon you will notice in the logs that the config-servicereceives the Github push event and sends RefreshRemoteApplicationEvents (delivered through Spring Cloud Bus with RabbitMQ transport) targeted at the applications it thinks might have changed (in this case to all targeted applications) As an exercise you can try if you just change a property in the foo-service.properties file then the config-service will send refresh event just to the foo-service.

**Summary**

Congratulations! You just setup centralized configuration management in a distributed system with automatic re-initialization using push notification, how cool is that? :)