Microservice Configuration: Spring Cloud Config Server Tutorial

by Brian Hannaway MVB · May. 17, 19 · Microservices Zone · Tutorial

Configuring Microservices: The Challenges

Managing application configuration in a traditional monolith is pretty straight forward. The configuration is usually externalized in a properties files on the same server as the application. If you need to update the configuration you simply amend the properties file and restart the application. Things can get a little tricker with microservices, but why is that?

Microservices are composed of many small, autonomous services, each with their own configuration. Rather than a centralised properties file (like the monolith), configuration is scattered across multiple services, running on multiple servers. In a production environment, where you likely have multiple instances of each service, configuration management can become a hefty task.

If you're running in the cloud things don't get any easier. Cloud environments tend to be quite fluid with instances regularly added and removed as a result of auto-scaling activity. This fluidity can make it difficult to apply updates and ensure that each service has the correct configuration.

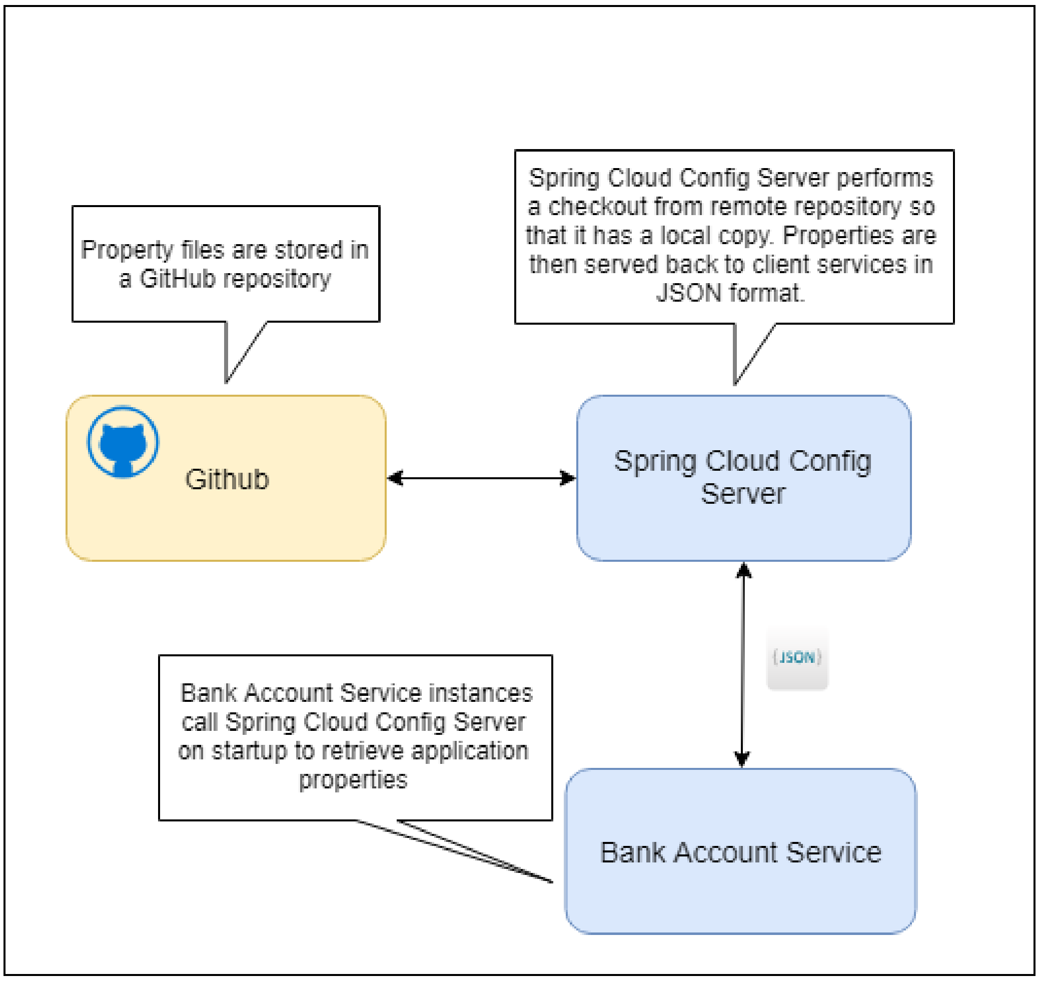
Introducing Centralized Configuration

Centralized configuration is a pattern where the configuration for all services is managed in a central repository rather than being scattered across individual services. Each service pulls its configuration from the central repository on startup.

Spring provides a basis for implementing this pattern via Spring Cloud Config, a subproject of Spring Cloud. With Spring Cloud Config you can create a Spring Boot application that exposes application properties via a REST API. Services can consume their application properties from the REST API rather than loading them locally from the file system or classpath. Configuration is not stored in the Cloud Config Server itself but pulled from a Git repository. This allows you to manage your application configuration with all the benefits of version control. Spring Cloud Config can be configured to use either a local git repository (useful during dev) or a remote repository. In a production environment, you'd want the Config Server to access configuration from a private Git repository.

Sample Application

In this post, I'm going to show you how to set up a Configuration Service that pulls configuration from GitHub. I'll also show you how the configuration is consumed by another service, in this case, a simple bank account service. The diagram below describes the main components involved.


Property files are stored in a GitHub repository
Spring Cloud Config Server performs a checkout from remote repository so that it has a local copy. Properties are then served back to client services in JSON format.
Github
Spring Cloud Config Server
JSON
Bank Account Service instances call Spring Cloud Config Server on startup to retrieve application properties
Bank Account Service

Centralized Configuration Architecture

The full source code for the sample app is available on GitHub.

Creating the Cloud Config Service

We'll begin by creating a simple Spring Boot app for the Config Service. Inside the main application class use `@EnableConfigServer` to enable the config server functionality.

```
@EnableConfigServer
@SpringBootApplication
```

```
public class ConfigServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(ConfigServerApplication.class, args);
    }
}
```

Configuring the POM

In order to make the required Spring Cloud dependencies available you'll need to add the following dependencies to the POM.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSche
   <modelVersion>4.0.0</modelVersion>
   <groupId>com.briansjavablog.microservices</groupId>
   <artifactId>config-server</artifactId>
   <version>0.0.1-SNAPSHOT</version>
   <packaging>jar</packaging>
   <name>config-server</name>
   <description>Demo config server provides centralised configuration for various micro serv
   <parent>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-parent</artifactId>
      <version>2.0.3.RELEASE</version>
   </parent>
   <properties>
      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
      <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
      <java.version>1.8</java.version>
      <spring-cloud.version>Finchley.RELEASE</spring-cloud.version>
   </properties>
   <dependencies>
      <dependency>
         <groupId>org.springframework.cloud</groupId>
         <artifactId>spring-cloud-config-server</artifactId>
      </dependency>
      <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-devtools</artifactId>
         <scope>runtime</scope>
      </dependency>
   </dependencies>
   <dependencyManagement>
      <dependencies>
         <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
         </dependency>
      </dependencies>
   </dependencyManagement>
   <build>
      <plugins>
```

```
        <plugin>
           <groupId>org.springframework.boot</groupId>
           <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
     </plugins>
  </build>
</project>-
```

Configuring the Config Server

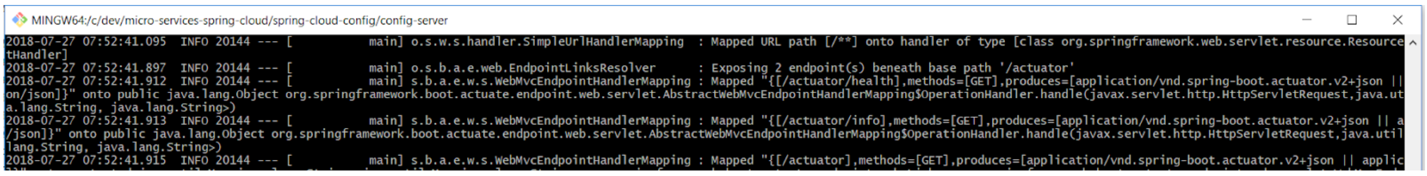
Next, you'll need to configure the Config Service via the `application.properties` file.

```
1 spring.application.name=config-server
2 server.port=8888
3
4 # URI of GIT repo containing properties
5 spring.cloud.config.server.git.uri=https://github.com/briansjavablog/micro-services-spring-c
6
7 # path to properties from root of repo
8 spring.cloud.config.server.git.searchPaths: configuration
9
0 logging.level.org.springframework.web=INFO
```

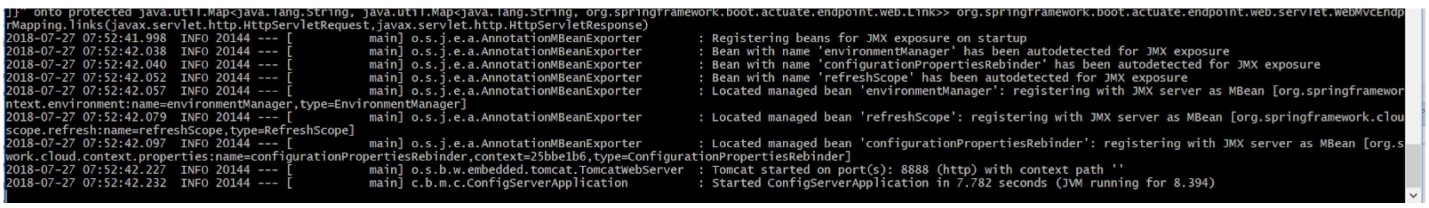
- Line 1 - `spring.application.name` specifies the name of the application. This isn't essential but it's good practice to name your Boot applications. This name will appear on the actuator/info endpoint and will be displayed in Spring Boot Admin if you use it.
- Line 2 - `server.port` specifies the port that the admin app will run on.
- Line 5 - `spring.cloud.config.server.git.uri` specifies the URL of the remote repository containing the property files that will be served up by the Config Service. In this example, I am pointing at my own repository on GitHub, but this could also point to a local Git repo. A local repository is useful during development and can be specified with the *file* prefix as follows *file:///c:/dev/micor-services/git-local-config-repo*.
- Line 8 - `spring.cloud.config.server.git.searchPaths` specifies the path to the properties files from the root of the repository. So, in the example above, I want to access property files in the `configuration` directory of the repository.

Running the Cloud Config Service

Now that we have the Config Server configured, it's time to fire it up and do a quick test. You can start the application in Eclipse or on the command line the same way you'd start any other Boot app. You should see the application start on port 8888 as follows.



```
MINGW64:/c/dev/micro-services-spring-cloud/spring-cloud-config/config-server
2018-07-27 07:52:41.095  INFO 20144 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/**] onto handler of type [class org.springframework.web.servlet.resource.Resource
tHandler]
2018-07-27 07:52:41.897  INFO 20144 --- [           main] o.s.b.a.e.web.EndpointLinksResolver      : Exposing 2 endpoint(s) beneath base path '/actuator'
2018-07-27 07:52:41.912  INFO 20144 --- [           main] s.b.a.e.w.s.WebMvcEndpointHandlerMapping : Mapped "{[/actuator/health],methods=[GET],produces=[application/vnd.spring-boot.actuator.v2+json ||
on/json]}" onto public java.lang.Object org.springframework.boot.actuate.endpoint.web.servlet.AbstractWebMvcEndpointHandlerMapping$OperationHandler.handle(javax.servlet.http.HttpServletRequest,java.ut
a.lang.String, java.lang.String>)
2018-07-27 07:52:41.913  INFO 20144 --- [           main] s.b.a.e.w.s.WebMvcEndpointHandlerMapping : Mapped "{[/actuator/info],methods=[GET],produces=[application/vnd.spring-boot.actuator.v2+json || a
/json]}" onto public java.lang.Object org.springframework.boot.actuate.endpoint.web.servlet.AbstractWebMvcEndpointHandlerMapping$OperationHandler.handle(javax.servlet.http.HttpServletRequest,java.util
lang.String, java.lang.String>)
2018-07-27 07:52:41.915  INFO 20144 --- [           main] s.b.a.e.w.s.WebMvcEndpointHandlerMapping : Mapped "{[/actuator],methods=[GET],produces=[application/vnd.spring-boot.actuator.v2+json || applic
```



```
]}" onto protected java.util.Map<java.lang.String, java.util.Map<java.lang.String, org.springframework.boot.actuate.endpoint.web.Link>> org.springframework.boot.actuate.endpoint.web.servlet.WebMvcEndp
rMapping.links(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
2018-07-27 07:52:41.998  INFO 20144 --- [           main] o.s.j.e.a.AnnotationMBeanExporter        : Registering beans for JMX exposure on startup
2018-07-27 07:52:42.038  INFO 20144 --- [           main] o.s.j.e.a.AnnotationMBeanExporter        : Bean with name 'environmentManager' has been autodetected for JMX exposure
2018-07-27 07:52:42.040  INFO 20144 --- [           main] o.s.j.e.a.AnnotationMBeanExporter        : Bean with name 'configurationPropertiesRebinder' has been autodetected for JMX exposure
2018-07-27 07:52:42.052  INFO 20144 --- [           main] o.s.j.e.a.AnnotationMBeanExporter        : Bean with name 'refreshScope' has been autodetected for JMX exposure
2018-07-27 07:52:42.057  INFO 20144 --- [           main] o.s.j.e.a.AnnotationMBeanExporter        : Located managed bean 'environmentManager': registering with JMX server as MBean [org.springframewor
ntext.environment:name=environmentManager,type=EnvironmentManager]
2018-07-27 07:52:42.079  INFO 20144 --- [           main] o.s.j.e.a.AnnotationMBeanExporter        : Located managed bean 'refreshScope': registering with JMX server as MBean [org.springframework.clou
scope.refresh:name=refreshScope,type=RefreshScope]
2018-07-27 07:52:42.097  INFO 20144 --- [           main] o.s.j.e.a.AnnotationMBeanExporter        : Located managed bean 'configurationPropertiesRebinder': registering with JMX server as MBean [org.s
work.cloud.context.properties:name=configurationPropertiesRebinder,context=25bbe1b6,type=ConfigurationPropertiesRebinder]
2018-07-27 07:52:42.227  INFO 20144 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8888 (http) with context path ''
2018-07-27 07:52:42.232  INFO 20144 --- [           main] c.b.m.c.ConfigServerApplication          : Started ConfigServerApplication in 7.782 seconds (JVM running for 8.394)
```

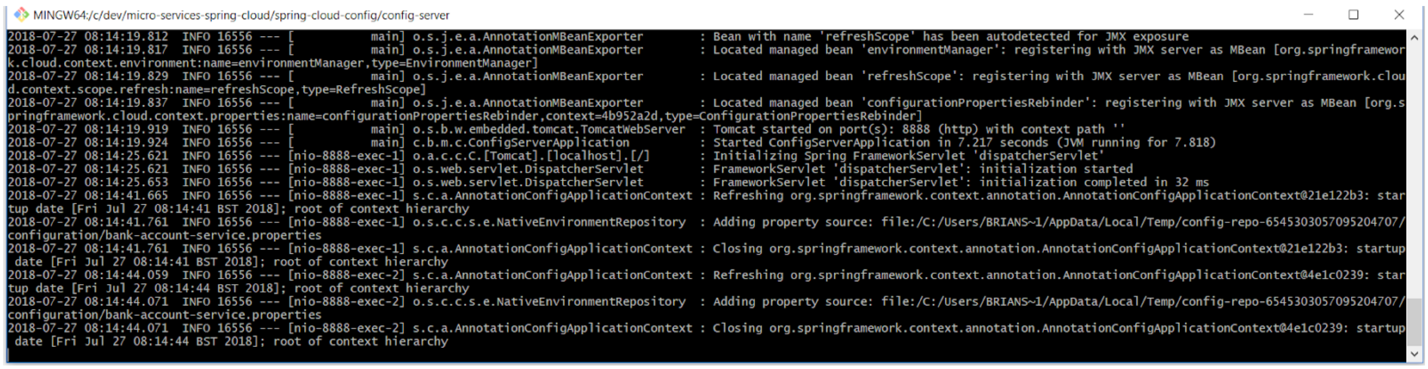
Cloud Config Service Startup Port 8888

Testing the Config Service

You can test the service by calling *http://localhost:8888/bank-account-service/default*. This GET request contains the name of the name and profile of the properties we want to load. In this instance, we are looking for the properties belonging to the `bank-account-service` and we want the properties associated with the `default` profile.

When a request is received, the Config Service uses the GIT URI from the `application.properties` file to perform a git clone of the remote repository. The screenshot below shows the repository being cloned to the local temp directory and includes the following line.

Adding property source: file:/C:/Users/BRIANS~1/AppData/Local/Temp/config-repo-6545303057095204707/configuration/bank-account-service.properties



MINGW64:/c/dev/micro-services-spring-cloud/spring-cloud-config/config-server

```
2018-07-27 08:14:19.812  INFO 16556 --- [           main] o.s.j.e.a.AnnotationMBeanExporter        : Bean with name 'refreshScope' has been autodetected for JMX exposure
2018-07-27 08:14:19.817  INFO 16556 --- [           main] o.s.j.e.a.AnnotationMBeanExporter        : Located managed bean 'environmentManager': registering with JMX server as MBean [org.springframewor
k.cloud.context.environment:name=environmentManager,type=EnvironmentManager]
2018-07-27 08:14:19.829  INFO 16556 --- [           main] o.s.j.e.a.AnnotationMBeanExporter        : Located managed bean 'refreshScope': registering with JMX server as MBean [org.springframework.clou
d.context.scope.refresh:name=refreshScope,type=RefreshScope]
2018-07-27 08:14:19.837  INFO 16556 --- [           main] o.s.j.e.a.AnnotationMBeanExporter        : Located managed bean 'configurationPropertiesRebinder': registering with JMX server as MBean [org.s
pringframework.cloud.context.properties:name=configurationPropertiesRebinder,context=4b952a2d,type=ConfigurationPropertiesRebinder]
2018-07-27 08:14:19.919  INFO 16556 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8888 (http) with context path ''
2018-07-27 08:14:19.924  INFO 16556 --- [           main] c.b.m.c.ConfigServerApplication          : Started ConfigServerApplication in 7.217 seconds (JVM running for 7.818)
2018-07-27 08:14:25.621  INFO 16556 --- [nio-8888-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring FrameworkServlet 'dispatcherServlet'
2018-07-27 08:14:25.621  INFO 16556 --- [nio-8888-exec-1] o.s.web.servlet.DispatcherServlet        : FrameworkServlet 'dispatcherServlet': initialization started
2018-07-27 08:14:25.653  INFO 16556 --- [nio-8888-exec-1] o.s.web.servlet.DispatcherServlet        : FrameworkServlet 'dispatcherServlet': initialization completed in 32 ms
2018-07-27 08:14:41.665  INFO 16556 --- [nio-8888-exec-1] s.c.a.AnnotationConfigApplicationContext : Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@21e122b3: star
tup date [Fri Jul 27 08:14:41 BST 2018]; root of context hierarchy
2018-07-27 08:14:41.761  INFO 16556 --- [nio-8888-exec-1] o.s.c.c.s.e.NativeEnvironmentRepository  : Adding property source: file:/C:/Users/BRIANS~1/AppData/Local/Temp/config-repo-6545303057095204707/
configuration/bank-account-service.properties
2018-07-27 08:14:41.761  INFO 16556 --- [nio-8888-exec-1] s.c.a.AnnotationConfigApplicationContext : Closing org.springframework.context.annotation.AnnotationConfigApplicationContext@21e122b3: startup
 date [Fri Jul 27 08:14:41 BST 2018]; root of context hierarchy
2018-07-27 08:14:44.059  INFO 16556 --- [nio-8888-exec-2] s.c.a.AnnotationConfigApplicationContext : Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@4e1c0239: star
tup date [Fri Jul 27 08:14:44 BST 2018]; root of context hierarchy
2018-07-27 08:14:44.071  INFO 16556 --- [nio-8888-exec-2] o.s.c.c.s.e.NativeEnvironmentRepository  : Adding property source: file:/C:/Users/BRIANS~1/AppData/Local/Temp/config-repo-6545303057095204707/
configuration/bank-account-service.properties
2018-07-27 08:14:44.071  INFO 16556 --- [nio-8888-exec-2] s.c.a.AnnotationConfigApplicationContext : Closing org.springframework.context.annotation.AnnotationConfigApplicationContext@4e1c0239: startup
 date [Fri Jul 27 08:14:44 BST 2018]; root of context hierarchy
```

Cloud Config Server - Cloning Repository

The properties in `bank-account-service.properties` are then read by the Config Service and returned to the client in JSON as shown below.

Config Service Response

```
{
   "name": "bank-account-service",
   "profiles": ["default"],
   "label": null,
   "version": "7b0732778b442726f8dd0bf7d1a36fc00f15c5b8",
   "state": null,
   "propertySources": [{
      "name": "https://github.com/briansjavablog/micro-services-spring-cloud-config/configu
      "source": {
         "bank-account-service.minBalance": "99"
```

```
          "bank-account-service.minBalance": "99",
          "bank-account-service.maxBalance": "200"
      }
   }]
}
```

The `name` attribute contains the name of the properties we requested. In this instance, we requested properties for the `bank-account-service`. Note that this matches the name of the default properties file in GitHub.

The `profiles` attribute is the Spring profile specified in the request. In this instance, we used the default profile, but we can specify any valid profile we want here. If you look at the property files on GitHub you'll see three files.

- bank-account-service.properties - this file contains the default properties and is used as the property source when the default profile is specified on the request.
- bank-account-service-dev.properties - this file has a '-dev' postfix and contains the properties returned when the dev profile is specified on the request.
- bank-account-service-uat.properties - this file has a '-uat' postfix and contains the properties returned when the uat profile is specified on the request.

If we specify `dev` or `uat` as the profile, the Config Service will return properties from the file matching that profile.

The `version` attribute is the current commit sha of the properties being returned. If you check GitHub this will match the commit sha of the latest commit. Finally, the `propertySources` attribute contains the GitHub source URI of the properties being returned along with the actual property values.

Creating a Bank Account Service

Now that the Config Service is up and running, its time to put it to work. We'll create a simple Bank Account Service that will call the Config Service on startup to retrieve its properties. The Bank Account Service has two REST endpoints, one to create a bank account and one to retrieve a bank account.

```
@RestController
@Slf4j
public class BankAccountController {

  @Autowired
  public BankAccountService bankAccountService;

  @PostMapping("/bank-account")
  public ResponseEntity << ? > createBankAccount(@RequestBody BankAccount bankAccount, HttpS

    bankAccountService.createBankAccount(bankAccount);
    log.info("created bank account {}", bankAccount);
    URI uri = new URI(request.getRequestURL() + "bank-account/" + bankAccount.getAccountId(
    return ResponseEntity.created(uri).build();
  }

  @GetMapping("/bank-account/{accountId}")
  public ResponseEntity<BankAccount> getBankAccount(@PathVariable("accountId") String accou

    BankAccount account = bankAccountService.retrieveBankAccount(accountId);
    log.info("retrieved bank account {}", account);
    return ResponseEntity.ok(account);
  }

}
```

The REST controller uses a simple `BankAccountService` to create and retrieve bank account details. When creating a new account, the service performs a check to see if the balance of the new account is between a set of minimum and maximum values.

```
/**
 * Add account to cache
 *
 * @param account
 */
public void createBankAccount(BankAccount account) {

  /* check balance is within allowed limits */
  if(account.getAccountBlance().doubleValue() >= config.getMinBalance() &&
      account.getAccountBlance().doubleValue() <= config.getMaxBalance()) {

    log.info("Account balance [{}] is is greater than lower bound [{}] and less than upper 
        account.getAccountBlance(), config.getMinBalance(), config.getMaxBalance());

    accountCache.put(account.getAccountId(), account);
  }
  else {

    log.info("Account balance [{}] is outside of lower bound [{}] and upper bound [{}]",
        account.getAccountBlance(), config.getMinBalance(), config.getMaxBalance());
    throw new InvalidAccountBalanceException("Bank Account Balance is outside of allowed th
  }
}
```

Bank Account Service Config

These minimum and maximum values are configurable and will be read from an injected `Configuration` object.

```
@Service
@Slf4j
public class BankAccountService {

  @Autowired
  private Configuration config;
```

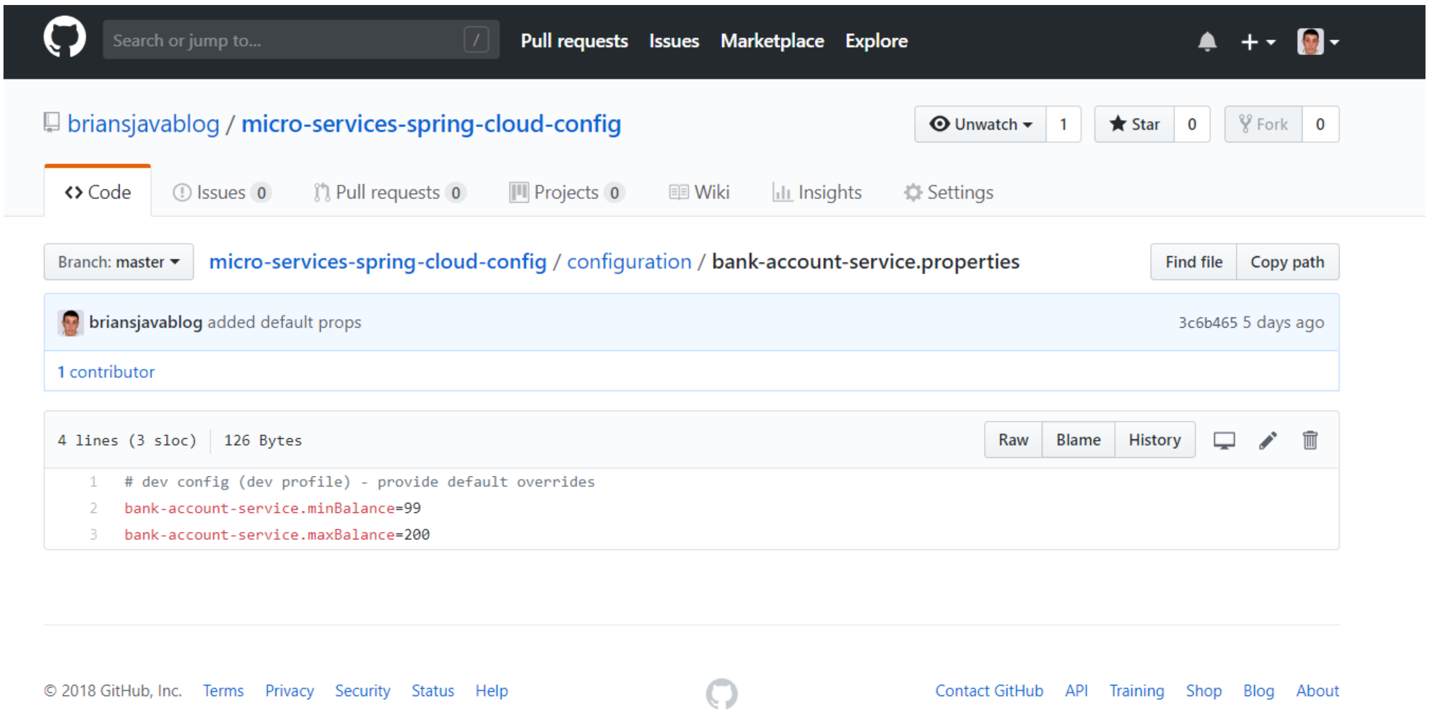
The `Configuration` object is defined as follows.

```
@Component
@ConfigurationProperties(prefix="bank-account-service")
public class Configuration {

  @Setter
  @Getter
  private Double minBalance;

  @Setter
  @Getter
  private Double maxBalance;

}
```

There are two important things to note.

- the `@ConfigurationProperties` prefix matches the name of the properties file in GitHub
- the key for each property in the property file is *<fileName>*.*<propertyName>* where the property name is the same as the instance variable name in the `@Configuration` class.

A screenshot from GitHub shows the file name and property names corresponding to the values in the `@Configuration` class.



micro-services-spring-cloud-config / configuration / bank-account-service.properties

briansjavablog added default props 3c6b465 5 days ago

1 contributor

4 lines (3 sloc) | 126 Bytes

```
# dev config (dev profile) - provide default overrides
bank-account-service.minBalance=99
bank-account-service.maxBalance=200
```

bank-account-service.properties in GitHub

Bank Account Service Local Configuration

Bank Account Service Local Configuration

We're going to use the Config Service to retrieve the Bank Account Service configuration. There are however a few properties which must be set locally.

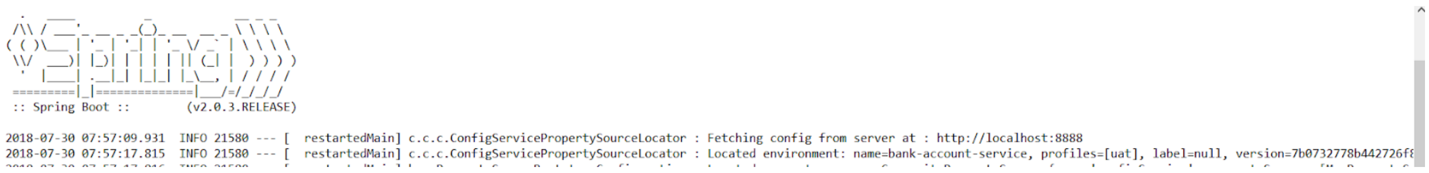
```
1 spring.application.name=bank-account-service
2 server.port=8080
3 spring.config.cloud.uri=htp://localhost:8888
4 spring.cloud.config.profile=uat
5 management.endpoints.web.exposure.include=*
```

Lines 1 and 2 are standard Boot configs and define the application name and port. Line 3 defines the URL of the Config Service. This is the URL that the Bank Account Service will call on startup to retrieve the `minBalance` and `maxBalance` properties. Line 4 defines the profile that will be used to call the Config Service. So given the configuration above, the Config Service will be called for the `bank-account-service` and `UAT` profile as follows.

```
http://localhost:8888/bank-account-service/uat.
```

Testing the Bank Account Service

Start the Bank Account Service on the command line or in Eclipse. In the log, you should see a call to the Config Service on http://localhost:8888 using the `uat` profile configured in `application.properties`.



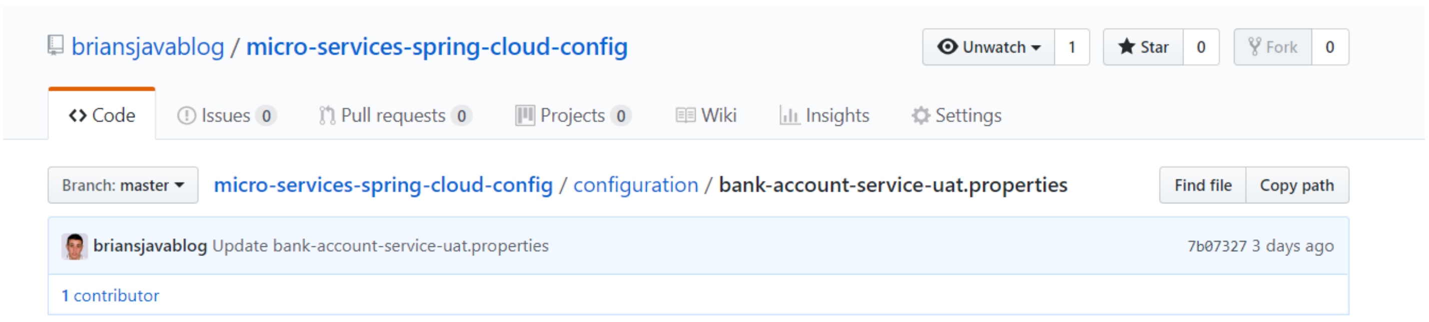
```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.0.3.RELEASE)

2018-07-30 07:57:09.931  INFO 21580 --- [  restartedMain] c.c.c.ConfigServicePropertySourceLocator : Fetching config from server at : http://localhost:8888
2018-07-30 07:57:17.815  INFO 21580 --- [  restartedMain] c.c.c.ConfigServicePropertySourceLocator : Located environment: name=bank-account-service, profiles=[uat], label=null, version=7b0732778b442726f8
```

Calling Config Service on startup

We can now test the application and confirm that the `uat` configuration was retrieved from the Config Service. Run the following cURL command to create a new bank account.

```
curl -i -H "Content-Type: application/json" -X POST -d '{"accountId":"B12345","accountName":
```

Note that we specify an account balance of £1250.38. This should be inside the allowed limits given that the `UAT` properties are defined as follows.

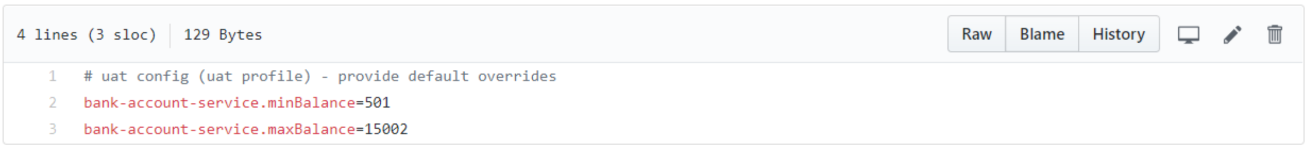


briansjavablog / micro-services-spring-cloud-config

micro-services-spring-cloud-config / configuration / bank-account-service-uat.properties

briansjavablog Update bank-account-service-uat.properties 7b07327 3 days ago

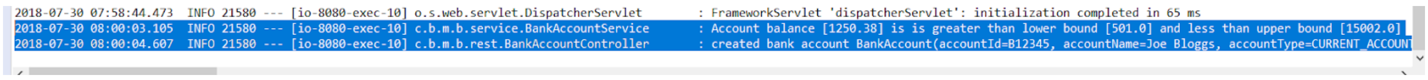
1 contributor



```
4 lines (3 sloc)  129 Bytes
# uat config (uat profile) - provide default overrides
bank-account-service.minBalance=501
bank-account-service.maxBalance=15002
```

Bank account service uat profile properties

The log extract below shows an account object is created successfully and the min and max values are logged as 501.0 and 15002.0 respectively. These values match those defined in the uat properties in GitHub.



```
2018-07-30 07:58:44.473  INFO 21580 --- [io-8080-exec-10] o.s.web.servlet.DispatcherServlet        : FrameworkServlet 'dispatcherServlet': initialization completed in 65 ms
2018-07-30 08:00:03.105  INFO 21580 --- [io-8080-exec-10] c.b.m.b.service.BankAccountService       : Account balance [1250.38] is is greater than lower bound [501.0] and less than upper bound [15002.0]
2018-07-30 08:00:04.607  INFO 21580 --- [io-8080-exec-10] c.b.m.b.rest.BankAccountController       : created bank account BankAccount(accountId=B12345, accountName=Joe Bloggs, accountType=CURRENT_ACCOUN
```

Log extract

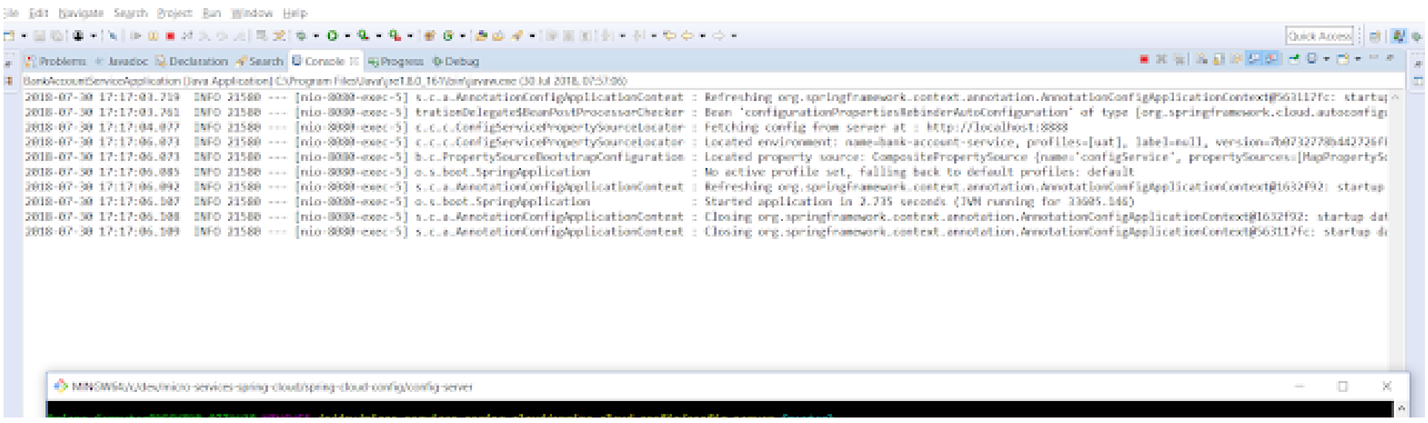
Updating Service Configuration

The main benefit of centralized configuration is that properties can be updated in one place (Git) and then those changes can be picked up by the Config Service right away. Every time an application requests properties from the Config Service, the Config Service will check if its locally cloned copy of the remote repository is up to date. If the local copy isn't up to date the Config Service will pull the latest properties from the remote repository and serve them back to the client. As a result, applications will always get the latest properties from the remote repository on startup. But what about updating properties for applications that are already running?

Thankfully, Spring Boot provides a way to reload application properties in a running application. The reload is triggered with a HTTP POST to the `refresh` actuator endpoint as follows.

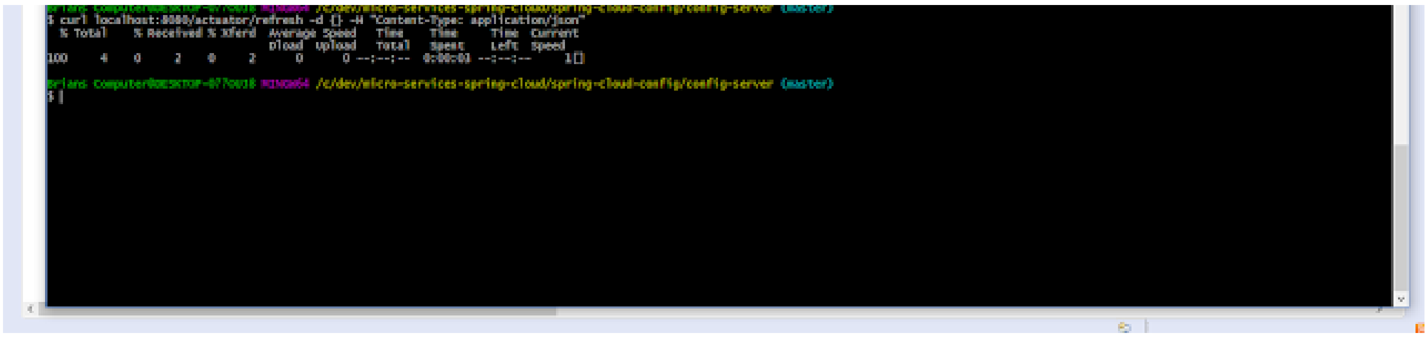
```
curl localhost:8080/actuator/refresh -d {} -H "Content-Type: application/json
```

The screenshot below shows the `/refresh` endpoint being called. In the application log, you can see a call to the Config Service to retrieve the latest properties for the UAT profile.


File Edit Navigate Search Project Run Window Help
Quick Access
Problems Javadoc Declaration Search Console Progress Debug

```
BankAccountServiceApplication [Java Application] C:\Program Files\Java\jre1.8.0_161\bin\javaw.exe (30 Jul 2018, 07:57:06)
2018-07-30 17:17:03.719  INFO 21580 --- [nio-8080-exec-5] s.c.a.AnnotationConfigApplicationContext : Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@563117fc: startu
2018-07-30 17:17:03.761  INFO 21580 --- [nio-8080-exec-5] trationDelegate$BeanPostProcessorChecker : Bean 'configurationPropertiesRebinderAutoConfiguration' of type [org.springframework.cloud.autoconfig
2018-07-30 17:17:04.077  INFO 21580 --- [nio-8080-exec-5] c.c.c.ConfigServicePropertySourceLocator : Fetching config from server at : http://localhost:8888
2018-07-30 17:17:06.073  INFO 21580 --- [nio-8080-exec-5] c.c.c.ConfigServicePropertySourceLocator : Located environment: name=bank-account-service, profiles=[uat], label=null, version=7b0732778bd42726f
2018-07-30 17:17:06.073  INFO 21580 --- [nio-8080-exec-5] b.c.PropertySourceBootstrapConfiguration : Located property source: CompositePropertySource {name='configService', propertySources=[MapPropertySo
2018-07-30 17:17:06.085  INFO 21580 --- [nio-8080-exec-5] o.s.boot.SpringApplication               : No active profile set, falling back to default profiles: default
2018-07-30 17:17:06.092  INFO 21580 --- [nio-8080-exec-5] s.c.a.AnnotationConfigApplicationContext : Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@1632f92: startup
2018-07-30 17:17:06.107  INFO 21580 --- [nio-8080-exec-5] o.s.boot.SpringApplication               : Started application in 2.735 seconds (JVM running for 33605.146)
2018-07-30 17:17:06.108  INFO 21580 --- [nio-8080-exec-5] s.c.a.AnnotationConfigApplicationContext : Closing org.springframework.context.annotation.AnnotationConfigApplicationContext@1632f92: startup dat
2018-07-30 17:17:06.109  INFO 21580 --- [nio-8080-exec-5] s.c.a.AnnotationConfigApplicationContext : Closing org.springframework.context.annotation.AnnotationConfigApplicationContext@563117fc: startup da
```

```
MINGW64:/c/dev/micro-services-spring-cloud/spring-cloud-config/config-server
```



```
brians ComputerDESKTOP-077OU18 MINGW64 /c/dev/micro-services-spring-cloud/spring-cloud-config/config-server (master)
$ curl localhost:8080/actuator/refresh -d {} -H "Content-Type: application/json"
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100     4    0     2    0     2      0      0 --:--:--  0:00:01 --:--:--     1[]

brians ComputerDESKTOP-077OU18 MINGW64 /c/dev/micro-services-spring-cloud/spring-cloud-config/config-server (master)
$ |
```

Property refresh via actuator endpoint

Automating Service Configuration Updates

After pushing config changes to GitHub it would be tedious if we had to grab the IP of each service and call its `/refresh` endpoint manually. We could easily create a script to poll the properties repository for changes and then use some kind of service discovery mechanism to get all registered service instances and call their `/refresh` endpoint. Some simple scripting used alongside a centralized Config Service should allow you to push configuration changes out across your microservices with minimal effort.

Wrapping Up

In this post, we looked at how Spring Cloud Config can be used to create a centralized configuration service that uses GitHub as its property repository. We also saw how properties can be managed for different environments using profiles, how these properties can be retrieved by a simple service and how updates can be applied to running services. The full source code for this post is available on GitHub so feel free to pull it down and experiment. If you have any questions or suggestions please leave a comment below.

Topics: MICROSERVICES, MICROSERVICES JAVA, MICROSERVICES TUTORIAL JAVA, SPRING CLOUD, SPRING CLOUD CONFIG SERVER