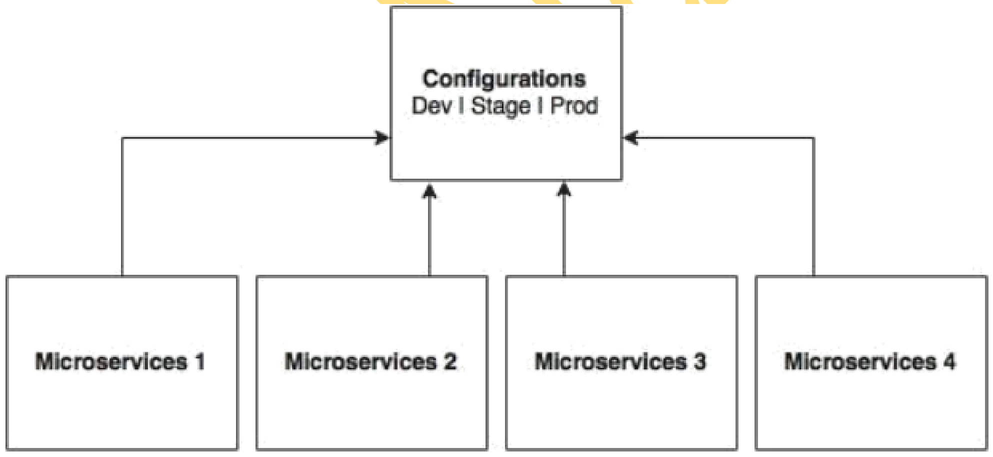
Spring Cloud Config

In Spring Boot applications, all configuration parameters were read from a property file packaged inside the project, either application.properties or application.yaml. This approach is good, since all properties are moved out of code to a property file. However, when Microservices are moved from one environment to another, these properties need to undergo changes, which require an application re-build. This is violation of one of the Twelve-Factor application principles, which suggests one-time build and moving of the binaries across environments.

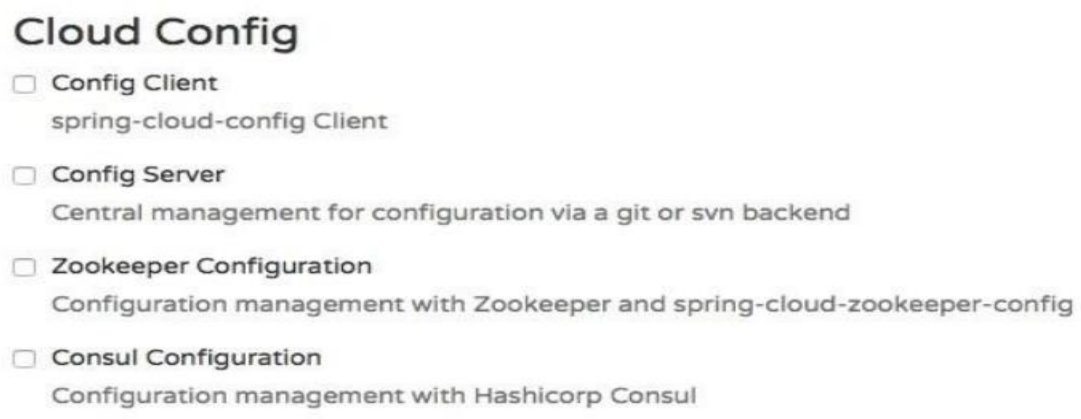
A better approach is to use the concept of profiles which is used for partitioning different properties for different environments. The profile-specific configuration will be named application-{profile}.properties. For example,application-development.properties represents a property file targeted for the development environment. However, the disadvantage of this approach is that the configurations are statically packaged along with the application. Any changes in the configuration properties require the application to be rebuilt.

Hence It is always recommended to externalize and centralize the configurations. **Spring Cloud Config is an externalized distributed configuration server** with Git or SVN as the backing repository which externalizes the application configurations to SVN/GIT repository.



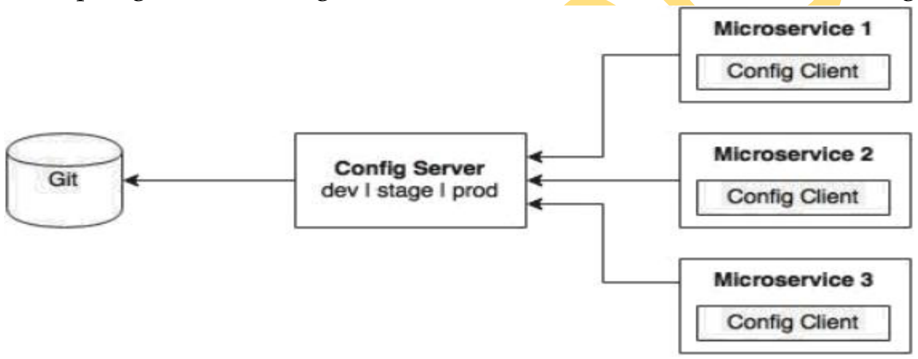
As shown in the preceding diagram, all microservices point to a central server to get the required configuration parameters. **The microservices then locally cache these parameters to improve performance. The Config server propagates the configuration state changes to all subscribed microservices so that the local cache's state can be updated with the latest changes**. The Config server also uses profiles to resolve values specific to an environment.

As shown in the following screenshot, there are multiple options available under the Spring Cloud project for building the configuration server. Config Server, Zookeeper Configuration, and Consul Configuration are available as options. However, we will use the Spring Config server implementation:


Cloud Config

- [ ] Config Client
  spring-cloud-config Client
- [ ] Config Server
  Central management for configuration via a git or svn backend
- [ ] Zookeeper Configuration
  Configuration management with Zookeeper and spring-cloud-zookeeper-config
- [ ] Consul Configuration
  Configuration management with Hashicorp Consul

The Spring Config server stores properties in a version-controlled repository such as Git or SVN. The Git repository can be local or remote. A highly available remote Git server is preferred for large scale distributed microservice deployments.
The Spring Cloud Config server architecture is shown in the following diagram:



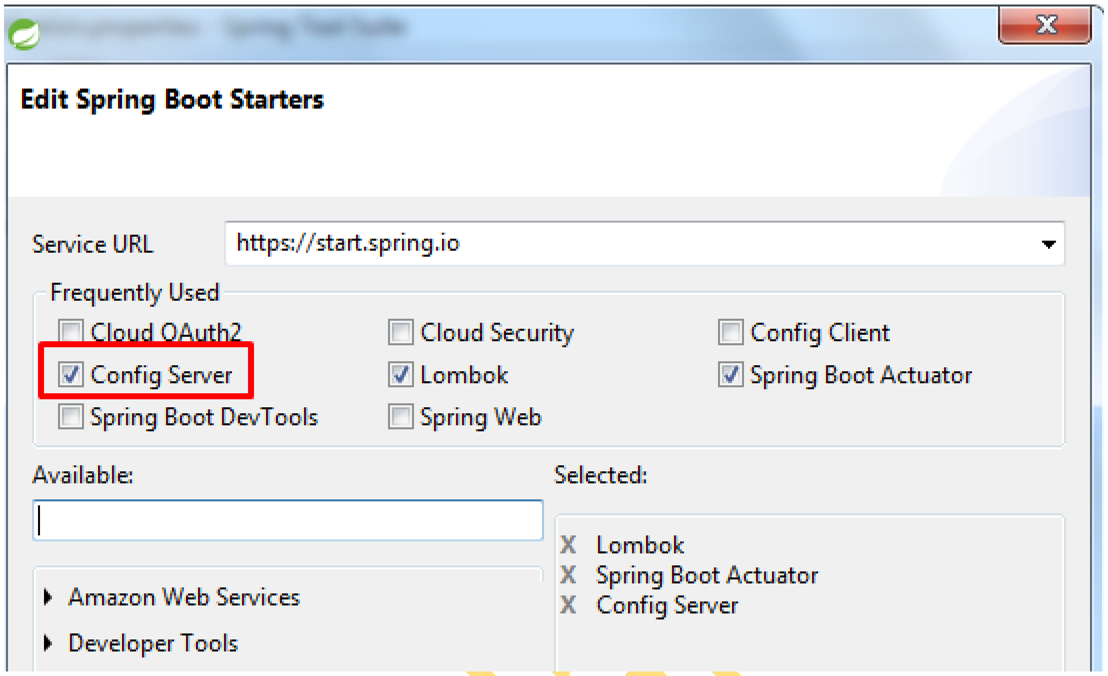
As shown in the preceding diagram, the Config client embedded in the Spring Boot MicroServices does a configuration lookup from a central configu   ration server using a simple declarative mechanism, and stores properties into the Spring environment. The configuration properties can be application-level configurations such as server URLs, credentials, and so on.

Unlike Spring Boot, Spring Cloud uses a bootstrap context, which is a parent context of the main application. Bootstrap context is responsible for loading configuration properties from the Config server. The bootstrap context looks for **bootstrap.yaml** or **bootstrap.properties** for loading initial configuration properties. To make this work in a Spring Boot application, rename the file **application.*** to **bootstrap.***.

Setting up the Config server

The following steps need to be followed to create a new Config server using STS:

Step-1: Create a new Spring Starter Project, and select config-server and Actuator as shown in the following diagram:



Edit Spring Boot Starters

Service URL: https://start.spring.io

Frequently Used

- [ ] Cloud OAuth2
- [ ] Cloud Security
- [ ] Config Client
- [✓] Config Server
- [✓] Lombok
- [✓] Spring Boot Actuator
- [ ] Spring Boot DevTools
- [ ] Spring Web

Available:

- Amazon Web Services
- Developer Tools

Selected:

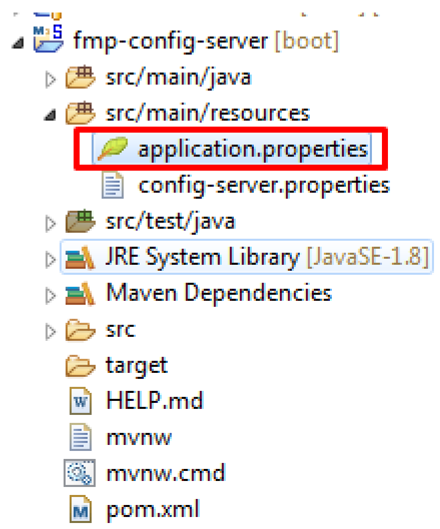
- X Lombok
- X Spring Boot Actuator
- X Config Server

Step-2: Set up a Git repository. This can be done by pointing to a remote Git configuration repository like the one at **https://github.com/spring-cloud-samples/config-repo**. This URL is an indicative one, a Git repository used by the Spring Cloud examples. We will have to use our own Git repository instead.

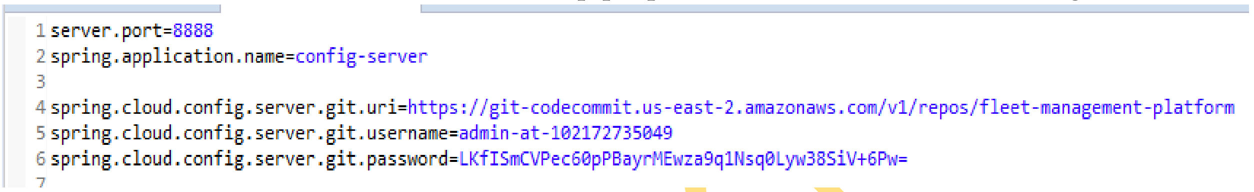
Note:

Alternately, a local filesystem-based Git repository can be used. In a real production scenario, an external Git is recommended.

Step-3: The next step is to change the configuration in the Config server to use the Git repository created in the previous step. In order to do this,include the git details in the file application.properties:


```
◢ fmp-config-server [boot]
  ▷ src/main/java
  ◢ src/main/resources
      application.properties
      config-server.properties
  ▷ src/test/java
  ▷ JRE System Library [JavaSE-1.8]
  ▷ Maven Dependencies
  ▷ src
    target
    HELP.md
    mvnw
    mvnw.cmd
    pom.xml
```

Step-4: Edit the contents of the new bootstrap.properties file to match the following:

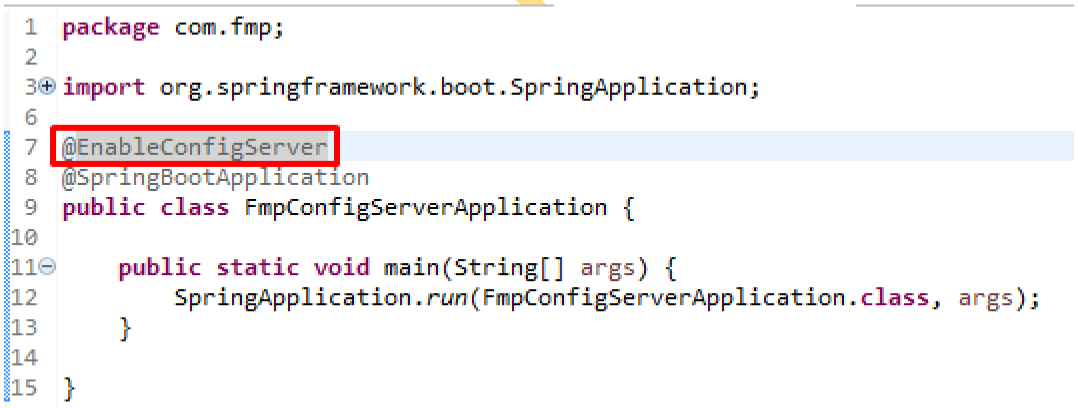

```
server.port=8888
spring.application.name=config-server

spring.cloud.config.server.git.uri=https://git-codecommit.us-east-2.amazonaws.com/v1/repos/fleet-management-platform
spring.cloud.config.server.git.username=admin-at-102172735049
spring.cloud.config.server.git.password=LKfISmCVPec60pPBayrMEwza9q1Nsq0Lyw38SiV+6Pw=

```

Port **8888** is the default port for the Config server. Even without configuring server.port, the Config server should bind to 8888.

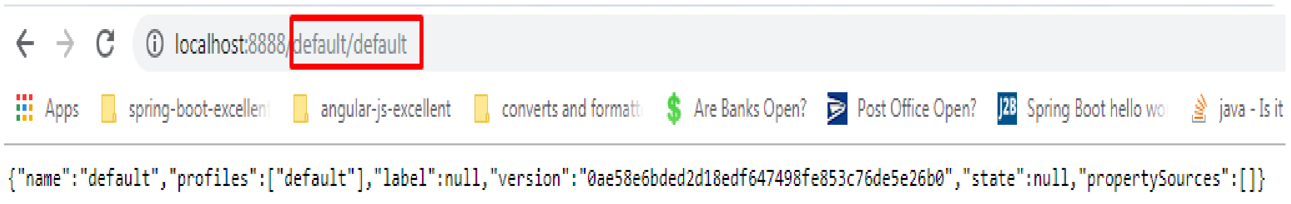
Note

Please don't add any context-path to any micro-service in spring cloud. Because the context will be appended by spring cloud components

Step-5: Add @EnableConfigServer in Application.java:


```java
package com.fmp;

import org.springframework.boot.SpringApplication;

@EnableConfigServer
@SpringBootApplication
public class FmpConfigServerApplication {

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

}
```

Step-6: Run the Config server by right-clicking on the project, and running it as a Spring Boot app. Visit http://localhost:8888/default/master to see whether the server is running. If everything is fine, this will list all environment configurations.



```
{"name":"default","profiles":["default"],"label":null,"version":"0ae58e6bded2d18edf647498fe853c76de5e26b0","state":null,"propertySources":[]}
```

The browser will display the properties configured in properties files which were there in git repo.

Spring-Cloud-Config: It provides Server-Client side support for externalized configurations in a distributed system.
Spring-Cloud-Config-Server: It provides a central place to manage the application's properties including all environments. By default, its storage backend uses git.

It exposes, the following REST GET endpoints, to get the application-specific configuration properties.

/{application}/{profile}/{label}
/{application}-{profile}.properties
/{label}/{application}-{profile}.properties
/{application}-{profile}.yml
/{label}/{application}-{profile}.yml

Here,{application} refers to the configuration property file name that is stored in the repository and the name that identifies the micro service ex:
spring.application.name=fmp-config-server property in the config server micro service.

In our case, it is fmp-config-server, from fmp-config-server.properties .
{profile} is an active profile at microservice indicating the environment like default, test, UAT, production,etc.
{label}: is an Optional git label or branch that defaults to 'master'.

Since passing label is an optional and it defaults to master, below two URLS are the same.
http://localhost:8888/fmp-config-server/default
http://localhost:8888/fmp-config-server/default/master

By default, the default profile is enabled, if we hit below URLS they will map to fmp-config-server.properties only.
http://localhost:8888/fmp-config-server-default.properties
http://localhost:8888/master/fmp-config-server-default.properties

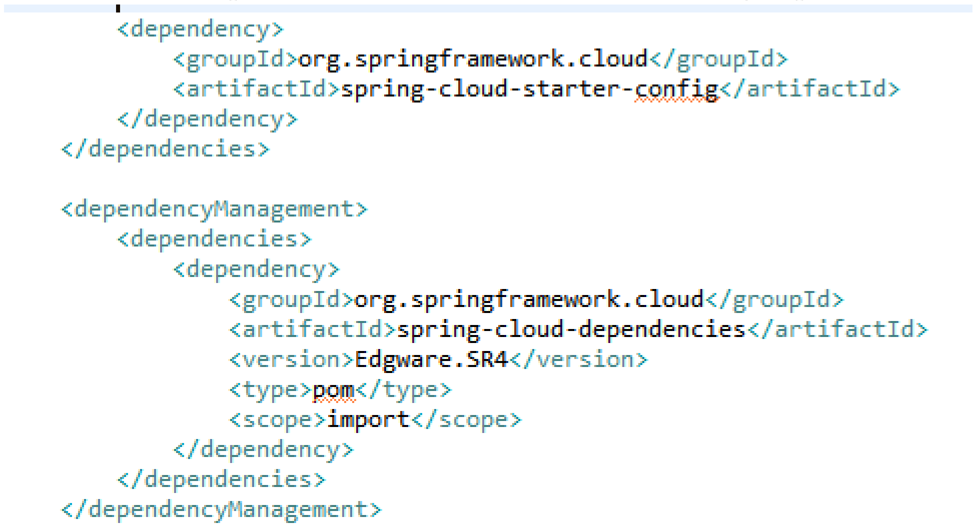
They both will return the properties and its values from the repository.

Accessing the Config Server from clients

Lets Consider we have the **patient-serice** micro service which will be now modified to use the Config Server. This service now should have spring config client configurations to communicate with the spring config server and hence forth we called it as spring config client. Follow these steps to use the Config server instead of reading properties from the application.properties file:

Step-1: Add the Spring Cloud Config dependency and the actuator (if the actuator is not already in place) to the pom.xml file. The actuator is mandatory for refreshing the configuration properties:

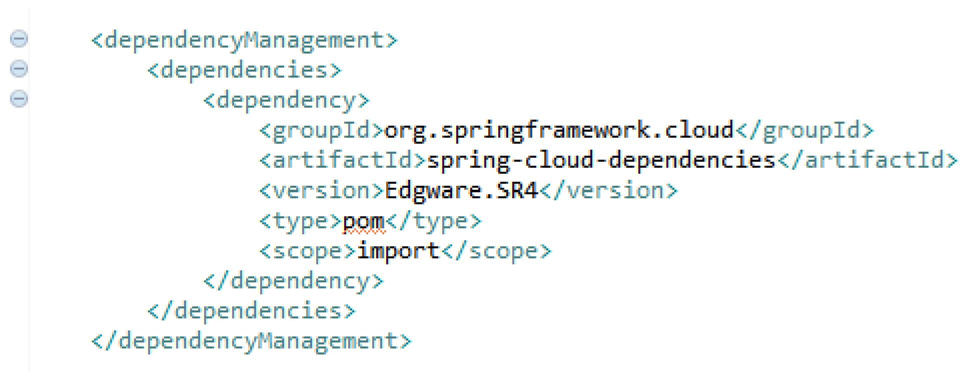
When we see the pom.xml file, we could see the following dependencies



```
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-config</artifactId>
        </dependency>
    </dependencies>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>Edgware.SR4</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
```

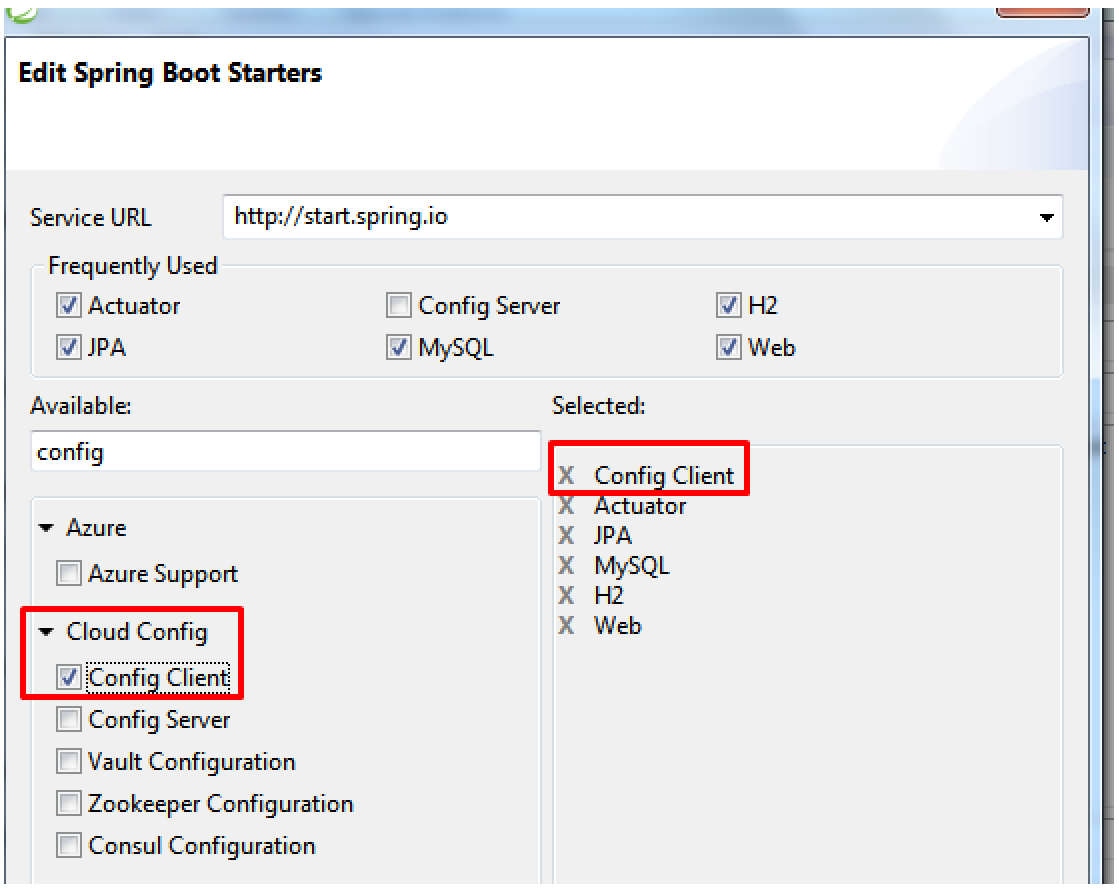
Note:

1) Since we are modifying the Spring Boot patient-service microservice that is already created, we have added the following to include the Spring Cloud dependencies. This is not required if the project is created from scratch



```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Edgware.SR4</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

2) If the application is built from the ground up, select the libraries as shown in the following screenshot:



Edit Spring Boot Starters

Service URL: http://start.spring.io

Frequently Used

[✓] Actuator [ ] Config Server [✓] H2
[✓] JPA [✓] MySQL [✓] Web

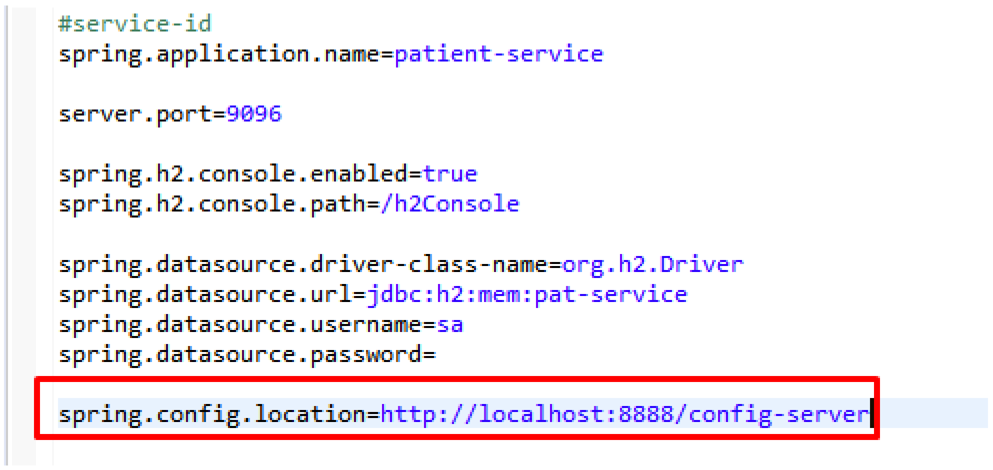
Available: config

Azure
[ ] Azure Support

Cloud Config
[✓] Config Client
[ ] Config Server
[ ] Vault Configuration
[ ] Zookeeper Configuration
[ ] Consul Configuration

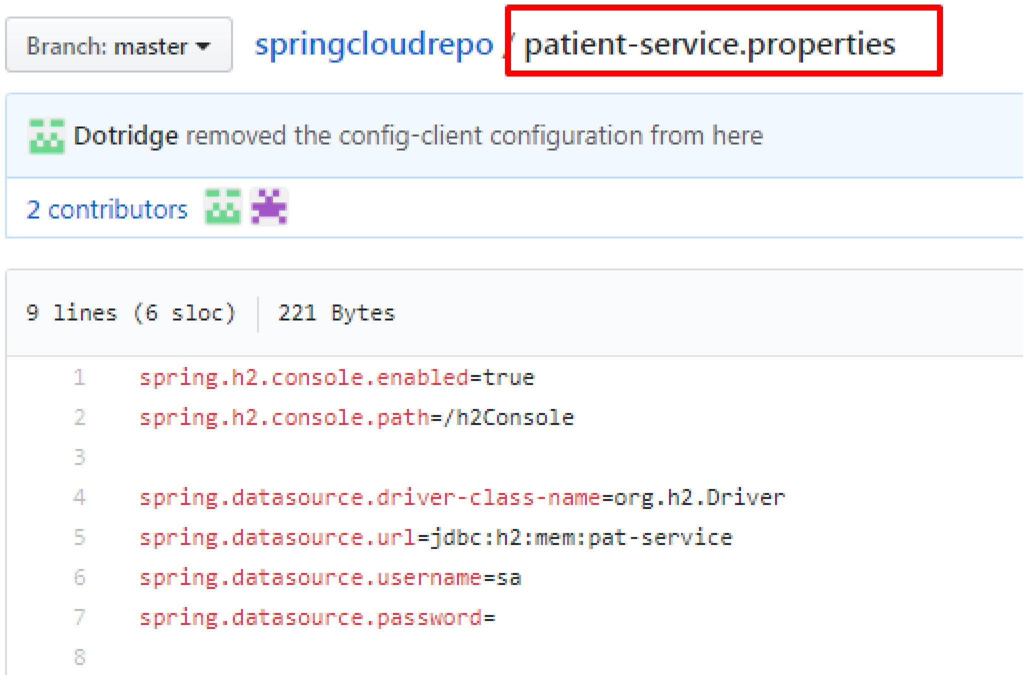
Selected:
X Config Client
X Actuator
X JPA
X MySQL
X H2
X Web

<u>Step-2:</u> In application.properties file add an application name and a configuration server URL. The configuration server URL is not mandatory if the Config server is running on the default port (8888) on the local host. The updated patient-service application.properties file will look as follows:



```
#service-id
spring.application.name=patient-service

server.port=9096

spring.h2.console.enabled=true
spring.h2.console.path=/h2Console

spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.url=jdbc:h2:mem:pat-service
spring.datasource.username=sa
spring.datasource.password=

spring.config.location=http://localhost:8888/config-server
```

Step-3: Create a new configuration file for **patient-service.properties** in the Git repository as shown below:



Branch: master ▾ springcloudrepo / patient-service.properties

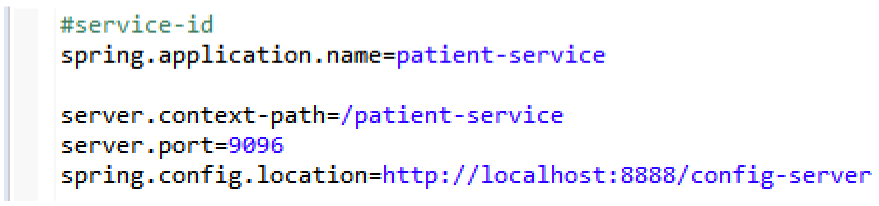
Dotridge removed the config-client configuration from here

2 contributors

9 lines (6 sloc) | 221 Bytes

```
spring.h2.console.enabled=true
spring.h2.console.path=/h2Console

spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.url=jdbc:h2:mem:pat-service
spring.datasource.username=sa
spring.datasource.password=

```

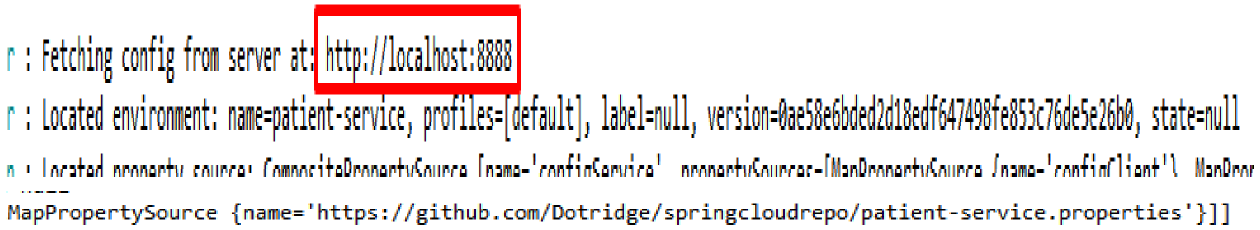
Note that the properties file name **patient-service** is the service ID given to the patient microservice in the application.properties file using the property **spring.application.name**. Move service-specific properties i.e. datasource etc properties from application.properties of patient-service microservice to the new patient-service.properties file created in git.
Now after moving the datasource properties from bootstrap/application.properties file, it should look like as below:



```
#service-id
spring.application.name=patient-service

server.context-path=/patient-service
server.port=9096
spring.config.location=http://localhost:8888/config-server
```

Step-4: Start the Config server. Then start the patient-service

Now when we run the patient-service, we should see the below log on console.



```
r : Fetching config from server at: http://localhost:8888
r : Located environment: name=patient-service, profiles=[default], label=null, version=0ae58e6bded2d18edf647498fe853c76de5e26b0, state=null
n : Located property source: CompositePropertySource [name='configService', propertySources=[MapPropertySource {name='configClient'}, MapProp
MapPropertySource {name='https://github.com/Dotridge/springcloudrepo/patient-service.properties'}]]
```

This is clear that,on bootstrapping of patient-service microservice, it is connecting to config server I.e. running on port 8888. Now Config Server connected to git, look-up for the properties file with the micro service name I.e patient-service.properties file, load the configurations from there and making available them to the Config Server client I.e. patient-service microservice. Hence patient-service micro service is connected to the database and will create the database tables according to the meta data descriptions in JPA Entities I.e. domains.

Note

Add all microservices as clients for config-server so that without restarting the microservices, we can pass runtime values from config-server