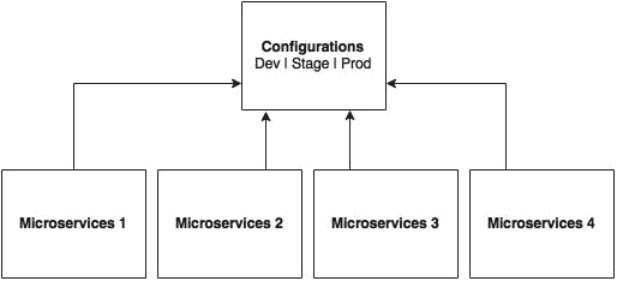
**Spring Cloud Config**

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 configuration server with Git or SVN as the backing repository which externalizes the application configurations to SVN/GIT repository.



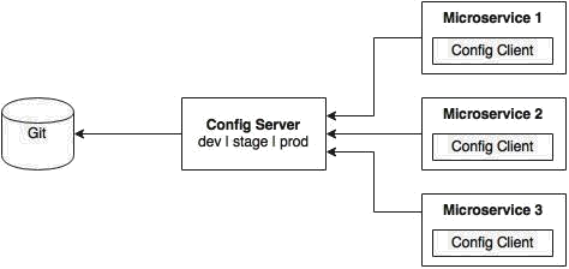
As shown in the preceding diagram, all microservices point to a central server to get the required configuration parameters. **The microservices then locally cache these parameters to improve performance.** **The Config server propagates the configuration state changes to all subscribed microservices 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:



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:



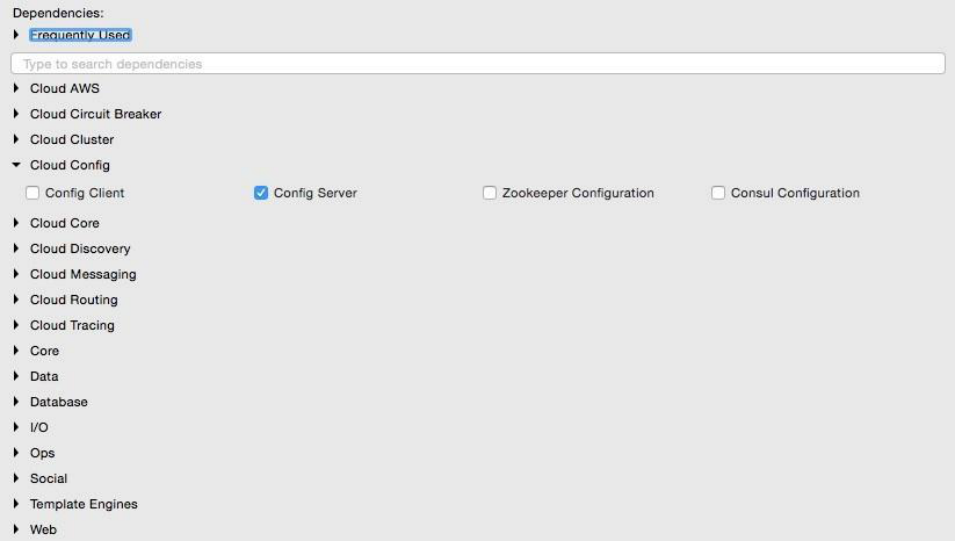
As shown in the preceding diagram, the Config client embedded in the Spring Boot MicroServices does a configuration lookup from a central configuration server using a simple declarative mechanism, and stores properties into the Spring environment. The configuration properties can be application-level 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:

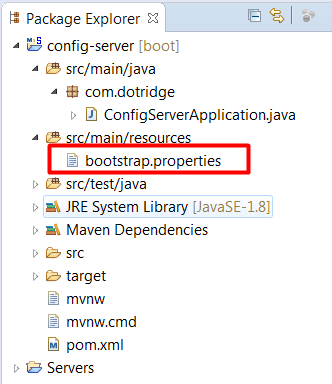


**Step-2:**Set up a Git repository. This can be done by pointing to a remote Git configuration repository like the one at **https://github.com/spring-cloud-samples/config-repo**. This URL is an indicative one, a Git repository used by the Spring Cloud examples. We will have to use our own Git repository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

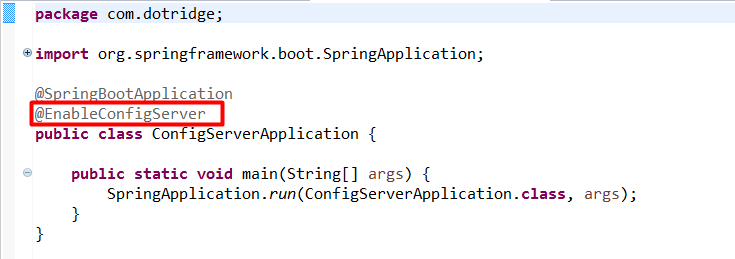


**Step-4:** Edit the contents of the new bootstrap.properties file to match the following:

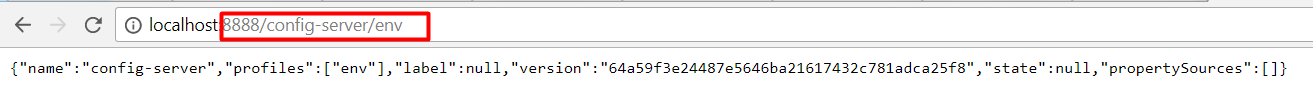


Port **8888** is the default port for the Config server. Even without configuring server.port, the Config server should bind to 8888.

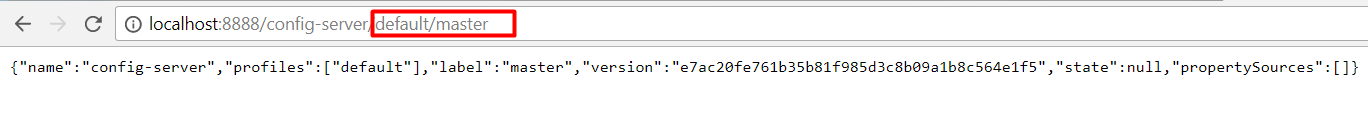
**Step-5:** Add @EnableConfigServer in Application.java:



**Step-6:** Run the Config server by right-clicking on the project, and running it as a Spring Boot app. Visit [**http://localhost:8888/config-server/env**](http://localhost:/config-server/8888/env) to see whether the server is running. If everything is fine, this will list all environment configurations. Note that /**env** is an actuator endpoint.



**Step-7:**.Check [**http://localhost:8888/config-server/default/master**](http://localhost:8888/config-server/default/master) to see the properties specific to properties file. The browser will display the properties configured in properties files which were there in git repo. The browser should display contents similar to the following:



**Understanding the Config server URL: http://localhost:8888/config-server/default/master**

The first element(after **http://localhost:8888/**) in the URL is the application name. In the given example, the application name should be **config-server**. The application name is a logical name given to the application, using the **spring.application.name** property in bootstrap.properties of the Spring Boot application. Each application must have a unique name. The Config server will use the name to resolve and pick up appropriate properties from the Config server repository. The application name is also sometimes referred to as service ID.

If there is an application with the name **patient-service**, then there should be a **patient-service.properties** in the configuration repository to store all the properties related to that application.

The second part (**default**)of the URL represents the profile. There can be more than one profile configured within the repository for an application. The profiles can be used in various scenarios. The two common scenarios are segregating different environments such as Dev, Test, Stage, Prod, and the like, or segregating server configurations such as Primary, Secondary, and so on. The first one represents different environments of an application, whereas the second one represents different servers where an application is deployed.

The profile names are logical names that will be used for matching the file name in the repository. The default profile is named default. To configure properties for different environments, we have to configure different files as given in the following example. In this example, the first file is for the development environment whereas the second is for the production environment:

application-development.properties

application-production.properties

These are accessible using the following URLs respectively:

http://localhost:8888/application/development,http://localhost:8888/application/production

The last part of the URL is the label, and is named master by default. The label is an optional Git label that can be used, if required. In short, the URL is based on the following pattern: **http://localhost:8888/{name}/{profile}/{label}.**

The configuration can also be accessed by ignoring the profile. In the preceding example, all the following three URLs point to the same configuration:

http://localhost:8888/application/default

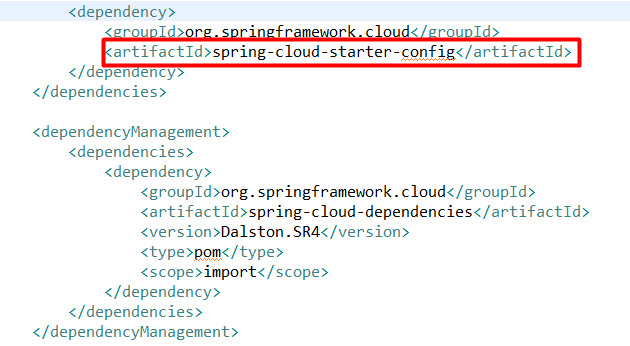
http://localhost:8888/application/master

<http://localhost:8888/application/default/master>

**Accessing the Config Server from clients**

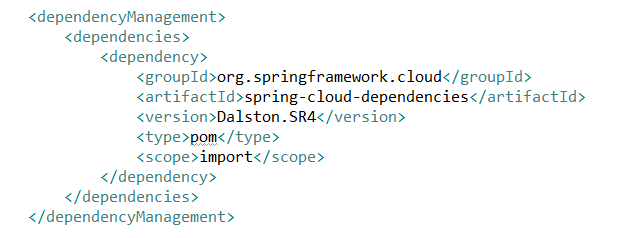
Lets Consider we have the **patient-serice** micro service which will be now modified to use the Config Server. This service now should have spring config client configurations to communicate with the spring config server and hence forth we called it as spring config client. Follow these steps to use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

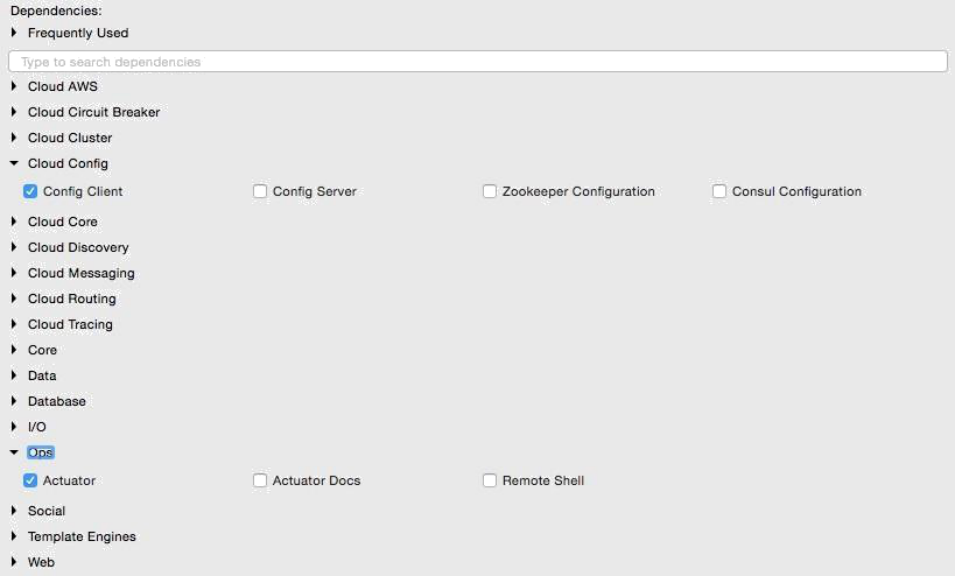


**Note:**

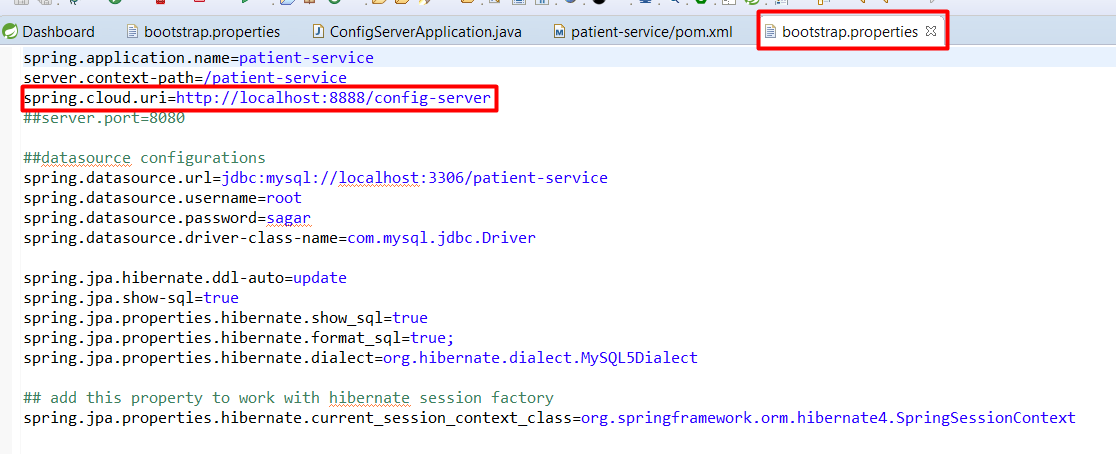
1) Since we are modifying the Spring Boot patient-service microservice that is already created, we have added the following to include the Spring Cloud dependencies. This is not required if the project is created from scratch



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

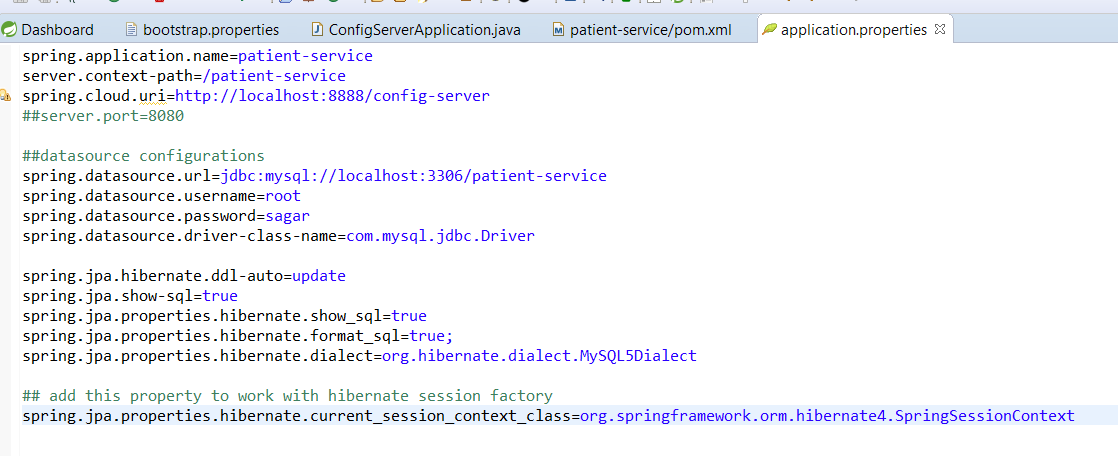


**Step-2:** Rename application.properties to bootstrap.properties, and add an application name and a configuration server URL. The configuration server URL is not mandatory if the Config server is running on the default port (8888) on the local host. The new bootstrap.properties file will look as follows:

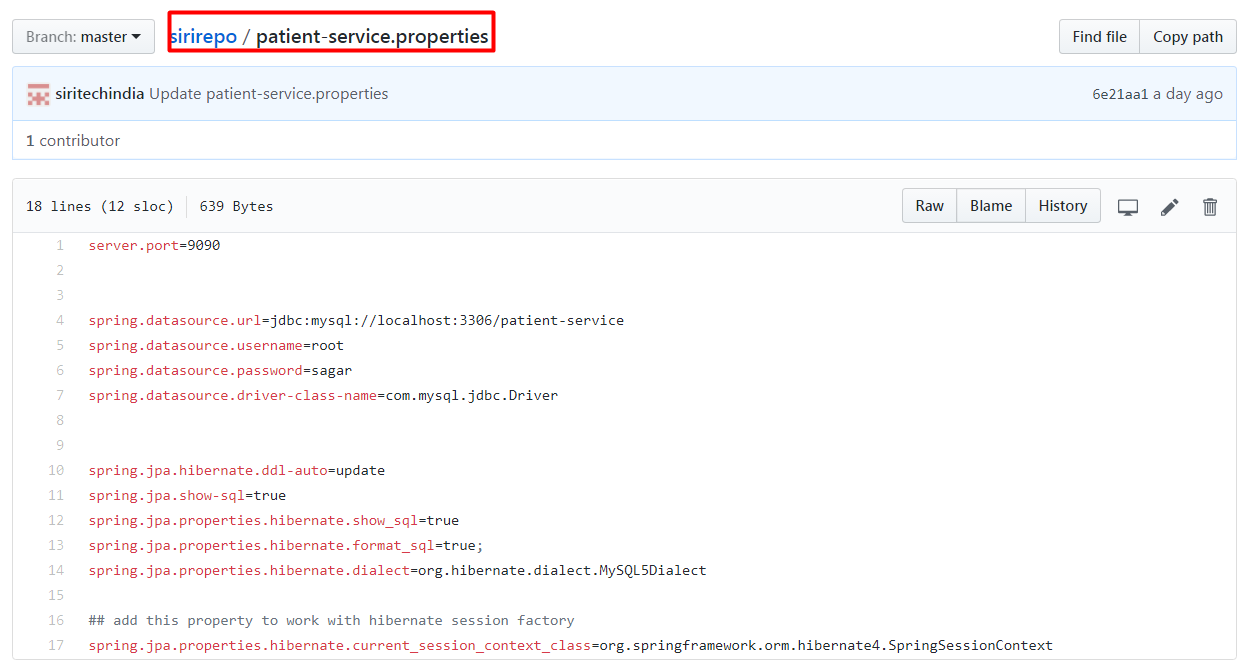


**Note:**

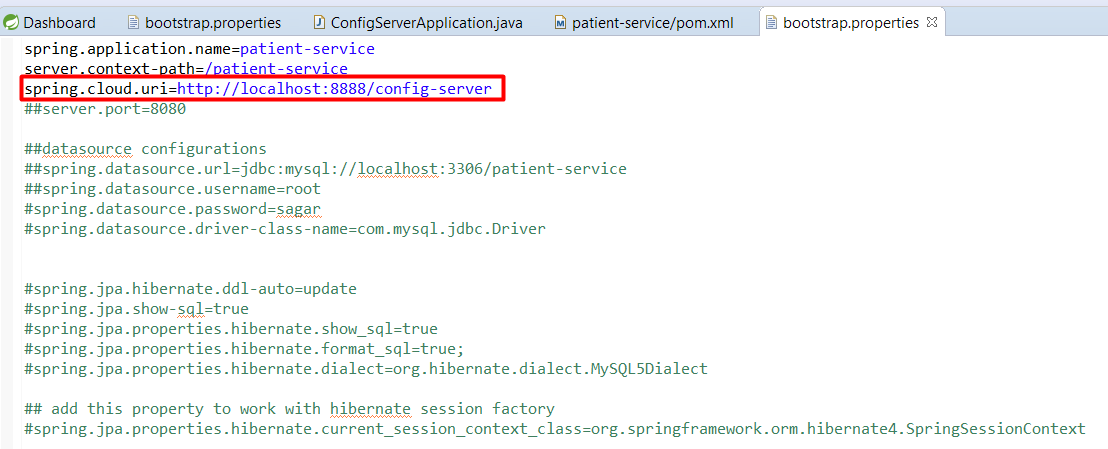
renaming application.properties file to bootstrap.properties file is not mandatory in config client micro services I.e. patient-service micro service. It will work even if we have application.properties file as shown below:



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

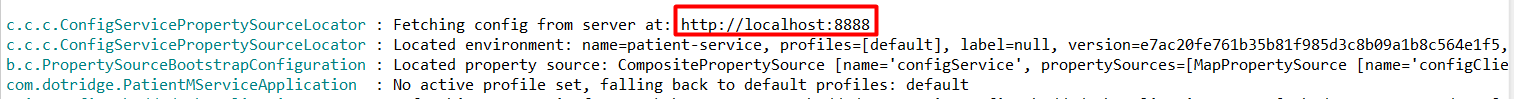


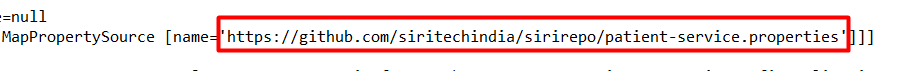
Note that **patient-service** is the service ID given to the patient microservice in the bootstrap.properties/application.properties file using the property **spring.application.name**. Move service-specific properties i.e. datasource etc properties from bootstrap.properties of patient-service microservice 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:



**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.





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.