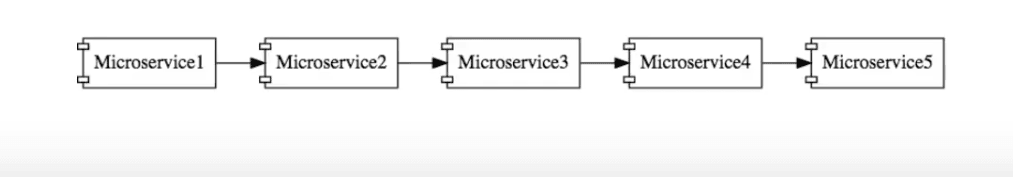
## Why Centralized Configuration?

When we talk about a microservices architecture, we visualize a large number of small microservices talking to each other. The number of microservices depends on the size of the enterprise.

Basic Microservices Architecture

The interesting part is that each of these microservices can have their own configuration.

Such configurations include details like:

* Application configuration.
* Database configuration.
* Communication Channel Configuration - queues and other infrastructure.
* URLs of other microservices to talk to.

In addition, each microservice will have a separate configuration for different environments, such as development, QA, and production.

**If maintaining a single configuration for a large application is difficult, imagine maintaining configurations for hundreds of microservices in different environments.**

# Centralized Config Server to the Rescue

That's where a centralized configuration server steps in.

Configuration for all microservices (for all environments) is stored at one place — a centralized configuration store.

When a microservice needs its configuration, it provides an ID at launch — a combination of the name of the microservice and the environment.

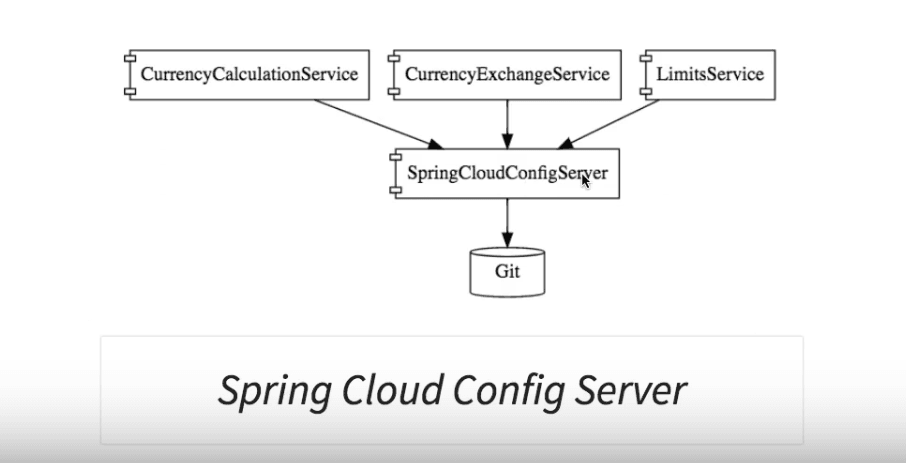
The centralized config server looks up the configuration and provides the configuration to the microservice.

**Ensure that the configuration in a centralized config server is secured and has role-based access.**

## Introducing Spring Cloud Config Server

Spring Cloud Config Server is one of the popular implementations of a cloud config server.

Spring Cloud Config Server enables you to store all the configurations for multiple microservices for different environments in a git or SVN Repository. A set of folder structures and conventions needs to be followed for the setup to work.



A microservice can connect to the config server and identify itself, and also specify the instance it represents. This enables it to get the required configuration.

The setup ensures that the operations team does not need to take time out to configure the individual microservices on a case-by-case basis. All that they need to worry about is configuring the centralized config server, and starting to put relevant configurations into the git repository.

### Automatically Picking Up Configuration Changes

An interesting feature present with the Spring Cloud Config Server is auto refresh. Whenever a change is committed to the git repository, configuration in the application is auto-refreshed.

.

# Managing Configuration for Microservices with Spring Cloud Config

**Last Updated :**20 May, 2024

Spring Cloud Config provides the centralized external configuration management system and it is designed to work well with modern microservices architectures. It is the part of larger Spring Config suite of the tools that aim to help the developers built the cloud-native applications.

Spring Cloud Config Server is the central place where all the configuration parameters of the applications are stored and maintained. It typically backend by the source control system like Git. It can serve the configuration across the multiple environments and applications.

Client microservices connect to the Config Server to fetch their configurations. This setup ensures all the configurations are centralized, version-controlled and maintained in the secure and efficient manner.

### Key Terminologies:

* **Config Server**: A server in the Spring Cloud Config and its setup that can centralizes and manages all the configuration properties for the client applications.
* **Config Client**: The application that can fetches its the configuration from the Config Server using the Spring Cloud Config.
* **@EnableConfigServer**: This annotation can be used in Spring applications to designate them as the Config Server.
* **Environment Repository**: The storage system can be used by the Config Server to the hold and serve configuration properties and it is commonly backed by the Git repository.
* **spring.config.import**: The property in Spring Boot that can specifies the sources from which the application should import the additional configuration data of the Spring application.

## Implementation of Managing Configuration for Microservices with Spring Cloud Config

Below are the implementations to manage configuration for Microservices with Spring Cloud Config.

# Components of Microservices

There are the following components of microservices:

* Spring Cloud Config Server
* Netflix Eureka Naming Server
* Hystrix Server
* Netflix ZuulAPI Gateway Server
* Netflix Ribbon
* Zipkin Distributed Tracing Server

### **Spring Cloud Config Server**

Spring Cloud Config Server provides the HTTP resource-based API for external configuration in the distributed system. We can enable the Spring Cloud Config Server by using the annotation **@EnableConfigServer**.

### **Netflix Eureka Naming Server**

Netflix Eureka Server is a discovery server. It provides the REST interface to the outside for communicating with it. A microservice after coming up, register itself as a discovery client. The Eureka server also has another software module called **Eureka Client**. Eureka client interacts with the Eureka server for service discovery. The Eureka client also balances the client requests.

### **Hystrix Server**

Hystrix server acts as a fault-tolerance robust system. It is used to avoid complete failure of an application. It does this by using the **Circuit Breaker mechanism**. If the application is running without any issue, the circuit remains closed. If there is an error encountered in the application, the Hystrix Server opens the circuit. The Hystrix server stops the further request to calling service. It provides a highly robust system.

Backward Skip 10sPlay VideoForward Skip 10s

### **Netflix Zuul API Gateway Server**

Netflix Zuul Server is a gateway server from where all the client request has passed through. It acts as a unified interface to a client. It also has an inbuilt load balancer to load the balance of all incoming request from the client.

### **Netflix Ribbon**

Netflix Ribbon is the client-side Inter-Process Communication (IPC) library. It provides the client-side balancing algorithm. It uses a Round Robin Load Balancing:

* Load balancing
* Fault tolerance
* Multiple protocols(HTTP, TCP, UDP)
* Caching and Batching

### **Zipkin Distributed Server**

Zipkin is an open-source project m project. That provides a mechanism for sending, receiving, and visualization traces.

One thing you need to be focused on that is port number.

|  |  |
| --- | --- |
| **Application** | **Port** |
| Spring Cloud Config Server | 8888 |
| Netflix Eureka Naming Server | 8761 |
| Netflix Zuul API gateway Server | 8765 |
| Zipkin distributed Tracing Server | 9411 |

# Creating a Simple Microservice

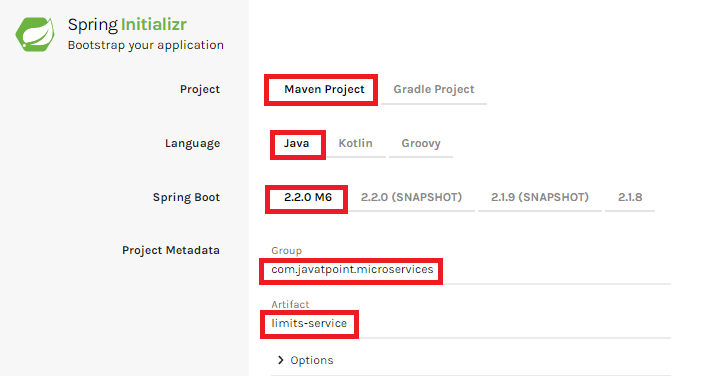
**Step 1**: Create a Maven project using Spring Initializr <https://start.spring.io/>

**Step 2**: Choose the Spring Boot version **2.2.0 M6** or higher version. Do not choose the snapshot version.

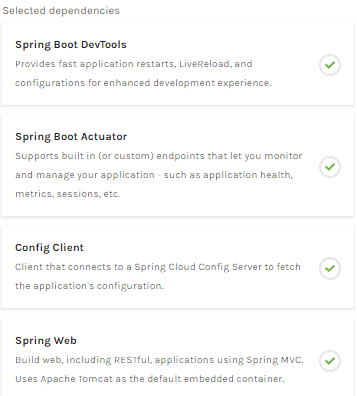
**Step 3**: Provide the **Group** name. In our case **om.javatpoint**

**Step 4**: Provide the **Artifact id**. We have provided **limits-service**.

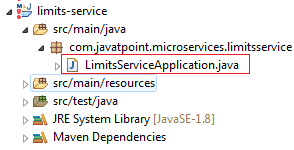
Backward Skip 10sPlay VideoForward Skip 10s



**Step 5**: Add the following dependencies: **Spring Web, Spring Boot DevTools, Spring Boot Actuator, Config Client**.



**Step 8**: Once the project is downloaded, go to **src/main/java**. Open the **LimitsServiceApplication**.



**Step 9**: Now run the **LimitsServiceApplication.java** as Java Application.

**It started the Tomcat on port(s) 8080 (http).**

Advertisement

Now we will add couple of services in the above project. For this we will have to follow the following steps:

**Step 1**: Open **application.properties** file and write the following code:

1. spring.application.name=limits-service      //name of application

**Step 2**: Create a class file with name **LimitsController.java** in the folder src/main/java under the package **com.javatpoint.microservices.limitsservice** and write the following code:



**Step 3**: Create a class file with name **Limits.java** in the folder **src/main/java** under the package **com.javatpoint.microservices.limitservice.bean** and write the following code:

*package* com.example.limits\_service.bean;  
  
*public class* Limits {  
  
 *private int* maximum;  
 *private int* minimum;  
  
 *public* Limits() {  
 }  
  
 *public* Limits(*int* maximum, *int* minimum) {  
 *this*.maximum = maximum;  
 *this*.minimum = minimum;  
 }  
  
 *public int* getMaximum() {  
 *return* maximum;  
 }  
  
 *public void* setMaximum(*int* maximum) {  
 *this*.maximum = maximum;  
 }  
  
 *public int* getMinimum() {  
 *return* minimum;  
 }  
  
 *public void* setMinimum(*int* minimum) {  
 *this*.minimum = minimum;  
 }  
}

Type the **localhost:8080/limits** in the browser and press enter, we get the JSON response as output.

**Output**

*{*

*maximum: 1000,*

*minimum: 1*

*}*

# Enhance limits service - Get configuration from application props - V3

**Adding services to the application.properties**

In the previous program, we will modify the code according to the requirement.

Now we call the **limits-service** from the **application.properties** file. In this file, we are configuring a couple of values.

1. limits-service.minimum=99
2. limits-service.maximum=9999

There is a better approach in Spring Boot to read values from the configuration using the annotation **@ConfigurationProperties**.

**Step 1**: Create a class with name **Configuration.java** in the folder **src/main/java** under the package **com.javatpoint.microservices.limitservice**.

**Step 2**: Add the annotations **@Component** and **@ConfigurationProperties**.

**Step 3**: Declare two variables **minimum** and **maximum**.

**Step 4**: If we are using the Configuration file, we need to generate getters and setters.

**Step 4**: If we are using the Configuration file, we need to generate getters and setters.

The Configuration.java file look like this.

1. **package** com.javatpoint.microservices.limitsservice;
2. **import** org.springframework.boot.context.properties.ConfigurationProperties;
3. **import** org.springframework.stereotype.Component;
4. @Component
5. @ConfigurationProperties("limits-service")
6. **public** **class** Configuration
7. {
8. **private** **int** maximum;
9. **private** **int** minimum;
10. **public** **void** setMaximum(**int** maximum)
11. {
12. **this**.maximum = maximum;
13. }
14. **public** **void** setMinimum(**int** minimum)
15. {
16. **this**.minimum = minimum;
17. }
18. **public** **int** getMaximum()
19. {
20. **return** maximum;
21. }
22. **public** **int** getMinimum()
23. {
24. **return** minimum;
25. }
26. }

**Step 5**: Now move to **LimitsConfigurationController.java** file and modify the code. In this we will use Configuration.

1. **package** com.javatpoint.microservices.limitsservice;
2. **import** org.springframework.beans.factory.annotation.Autowired;
3. **import** org.springframework.web.bind.annotation.GetMapping;
4. **import** org.springframework.web.bind.annotation.RestController;
5. **import** com.javatpoint.microservices.limitsservice.bean.LimitConfiguration;
6. @RestController
7. **public** **class** LimitsConfigurationController
8. {
9. @Autowired
10. **private** Configuration configuration;
11. @GetMapping("/limits")
12. **public** LimitConfiguration retriveLimitsFromConfigurations()
13. {
14. //getting values from the properties file
15. **return** **new** LimitConfiguration(configuration.getMaximum(), configuration.getMinimum());
16. }
17. }

Now refresh the browser page. It shows the JSON format of the updated values which are configured in **application .properties** file.

**Output**

*{*

*maximum: 999,*

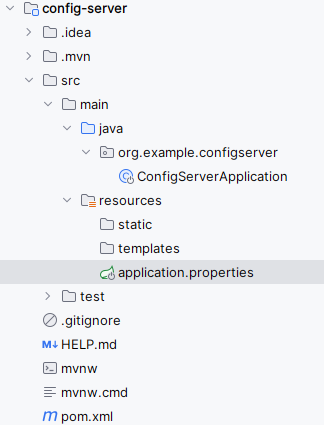
# Setup the Config Server

**Step 1**: Create the spring project using spring initializer and add the below required dependencies.

**Dependencies:**

* Spring Web
* Spring Dev Tools
* Lombok
* Spring Cloud Config

After creating the Spring project, the file structure will be like below image.



**Step 2**: Open the application.properties file and add the configuring the server port and git repository uri configuration of the application.

spring.application.name=config-server  
  
server.port=8888  
spring.cloud.config.server.git.uri=https://github.com/iammahesh123/spring-cloud-config-server.git

**Git repository application.yml code:**

user:  
 role: Admin  
  
welcome:  
 message: Welcome to the Spring Cloud Config managed application!

**Step 2**: Open the main class, add the ***@EnableConfigServer*** to activate the Spring cloud config functionality of the application.

Go to **src > main >java > org.example.configserver > ConfigServerApplication** and put the below code.

**package** **org.example.configserver**;

**import** **org.springframework.boot.SpringApplication**;

**import** **org.springframework.boot.autoconfigure.SpringBootApplication**;

**import** **org.springframework.cloud.config.server.EnableConfigServer**;

@SpringBootApplication

@EnableConfigServer

**public** **class** **ConfigServerApplication** {

**public** **static** void main(String[] args) {

SpringApplication.run(ConfigServerApplication.class, args);

}

}

**Step 3: Run the application**

Once the Spring project is completed and successfully runs as a Spring application, it will start at port 8888.

# Application Runs

## **Installing Git and creating a local repository**

**Step 1:**Download Git from https://git-scm.com/ and install it.

**Step 2:**Create a Git repository and store the files that we want to be able to configure a limits-service. We will try to access them from the spring-cloud-config-server. Open the Git bash and type the following commands:

Advertisement

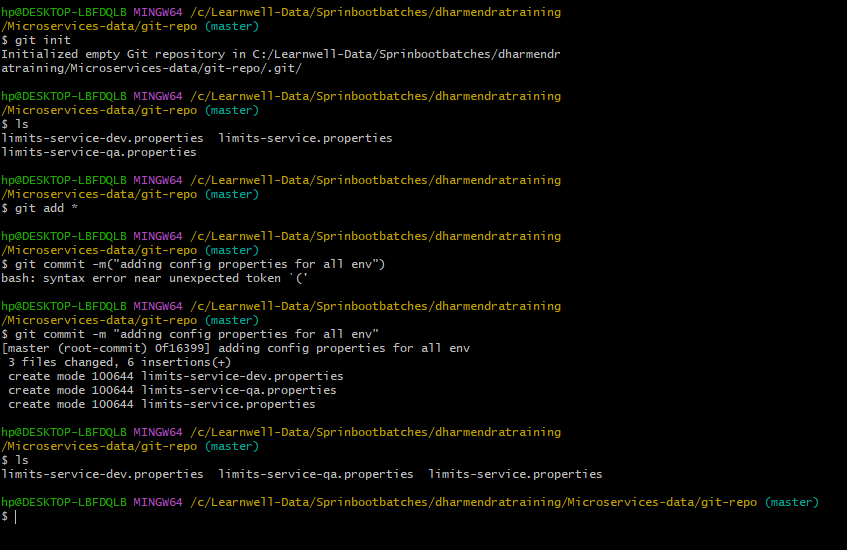
Creating a new directory:

1. mkdir git-localconfig-repo
2. cd git-localconfig-repo/

Initializing a new Git repository:

1. git init

It initializes an **empty**git repository.



# Connect Spring Cloud Config Server to Local Git Repository

In this section, we are going to learn how to connect spring-cloud-config-server to the local git repository. First, we will find the folder path.

Right-click on **git-localconfig-repo** -> **Properties** -> copy the **Location** label address and paste it into the **application.properties** file.

Add the annotation **@EnableConfigServer**in the SpringCloudConfigServerApplication.java file.

Type the following URL in the browser:

**localhost:8888/limits-service/default**

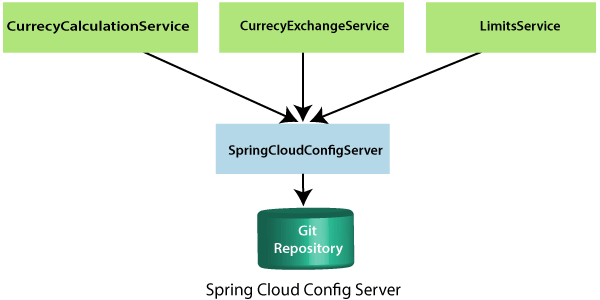
**Output**

1. {
2. name: "limits-service",
3. -profiles: [
4. "default"
5. ],
6. label: **null**,
7. version:"0898c54ae1deb62733728e37e4c7962f529ee9ad",
8. state: **null**,
9. -propertySources: [
10. - {
11. name: C:\Users\Anubhav\git-localconfig-repo\limits-service.properties",
12. -source: {
13. limits-service-minimum: "8",
14. limits-service-maximum: "88"
15. }
16. }
17. ]
18. }

In this we have establish the connection between **SprinCloudConfigServer** and the **Git repository**.

We can see that it displays a set of property and values. It also retrieves the file name of the property file from where these values (minimum and maximum) are retrieved.

The important thing about SpringCloudConfigServer is that **it stores configuration for multiple services.**It can also store configuration for each of the services for different environments.



In the above figure, there are three services **CurrencyCalculationService**, **CurrencyExchangeService**, and **LimitsService**. The LimitsService has four environment services **Dev, QA, Stage,** and **Production**. We can configure these three services in SpringCloudConfigServer.

## **Configuration for Multiple Environment in Git Repository**

services **Dev, QA, Stage,** and **Production**. We can configure these three services in SpringCloudConfigServer.

**Configuration for Multiple Environment in Git Repository**

In the spring-cloud-config-server project, we have added a link to git-localconfig-repo, which contains the limits-service.properties file. It becomes the default configuration for the limits-service.

However, we can overwrite them for a specific environment. To overwrite these values, copy the **limits-service.properties** and paste in the folder **git-localconfig-repo**rename it with **limits-service-dev.properties**. Now update the minimum and maximum values.

1. limits-service.minimum=1
2. limits-service.maximum=111

Again copy the same file and paste it in the same folder. Rename it with **limits-service-qa.properties**. Now update the minimum and maximum values.

1. limits-service.minimum=2
2. limits-service.maximum=222

If we want to pick the default value of the maximum instead of modified value, put a **introduction-to-currency-conversion-and-currency-exchange-service** symbol at the starting of the statement. Now the second statement becomes a comment.

1. limits-service.minimum=1
2. introduction-to-currency-conversion-and-currency-exchange-servicelimits-service.maximum=111

When we execute it, it picks up the maximum value 888 from the default properties file instead of maximum value 111. Whenever we make the changes in the file, commit the changes in the local repository.

Advertisement

Advertisement

Now open the Git Bash and execute the following commands:

Create the directory in which we want to add files.

1. cd git-localconfig-repo

Add the files into the Git repository.

Advertisement

Advertisement

1. git add -A

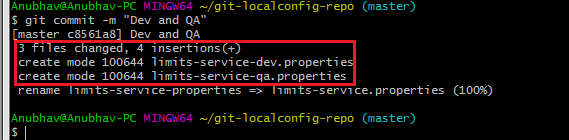
Now check the status of the files that have to be committed.

1. git status



Now commit the changes

1. git commit -m "Dev and QA"



Advertisement

Advertisement

Now we can access the properties Dev and QA.

Type the following in the address bar of the browser.

1. localhost:8888/limits-service/qa

**Output**

1. {
2. name: "limits-service",
3. -profiles: [
4. "qa"
5. ],
6. label: **null**,
7. version:"0898c54ae1deb62733728e37e4c7962f529ee9ad",
8. state: **null**,
9. -propertySources: [
10. - {
11. name: C:\Users\Anubhav\git-localconfig-repo\limits-service-qa.properties",
12. -source: {
13. limits-service-minimum: "2",
14. limits-service-maximum: "222"
15. }
16. },
17. -{
18. name: C:\Users\Anubhav\git-localconfig-repo\limits-service.properties?,
19. -source: {
20. limits-service-minimum: "8",
21. limits-service-maximum: "888"
22. }
23. }
24. ]
25. }

We can observe that it is retrieving the property sources. These list of property are in the list of priority. The heights priority is whatever values are configured in the QA file.

If there is a value that is not present in the QA file, then the value from the default file will be picked up. So whatever is in the QA file gets the highest property.

## **Connect limits-service to Spring Cloud Config Server**

In this section, we will connect limits-service to pick up the configuration from the spring-cloud-config-server. We do not need to configure values in the application.properties file. Move to the **limits-service** project and rename the **application.properties** file to **bootstrap.properties**. We do not need to configure values in the bootstrap.properties. All the configuration values picked from the spring-cloud-config-server. Specify the URI in the bootstrap.properties.

1. spring.application.name=limits-service
2. spring.cloud.config.uri=http://localhost:8888