Cloud Config Client

We will implement the driver-service, vehicle-service and trips-service are the clients to the config server. Lets start with the driver-service.

Step1: As we already have been taken the spring-boot-starter-web dependency in driver-service, lets add two more dependencies:

Config Client:

This dependency makes the micro services are as the client for the cloud server and supports fetching the properties from the config server.

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

Actuator:

For invoking the /actuator/refresh endpoint for refreshing and fetching the latest property file changes from the Config Server.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Step2:

Annotate the DriverController.java with @RefreshScope annotation.

Step3:

In the controller add an attribute 'todaysFuelRate' that will be read from the remote repository via Config Server. It will be populated using the, todaysFuelRate' attribute defined in **config-driver-service.properties** in the git repository.

Since all the property files related to different microservices and environments are placed in a single location, there should be a way to distinguish between them. Otherwise, spring cloud

config server will face problems when picking up the correct configuration file for the service. This can be achieved with the name of the property file.

The property file should be named based on the following rules:

config-<application_name>[-<profile>]

For Example: **config-driver-service-prod.properties**

If there are different profiles, the profile names should be come after the application name. Though profile is optional.

For Example: **config-driver-service-dev.properties**

The spring cloud configuraion will pick the correct property file based on the application name and the activated profile. If no profile is activated explicitely, it will pick the property file with no profile suffix.

```
 */
@EnableResourceServer
@RestController
@RefreshScope
@RequestMapping("/api")
public class DriversController {

    @Value("${todaysFuelRate}")
    private String todaysFuelRate;
```

Step4:

Add the config server details in application.properties as shown below:

```
spring.cloud.config.uri=http://localhost:8888/
spring.profiles.active=default
spring.cloud.config.label=fmp-config-server
```

Step-5:

Open the rest client and hit the url: http://localhost:7076/api/drivers

When we hit the above url, client microservice is able to get the todaysFuelRate property from the config-driver-service.properties file.

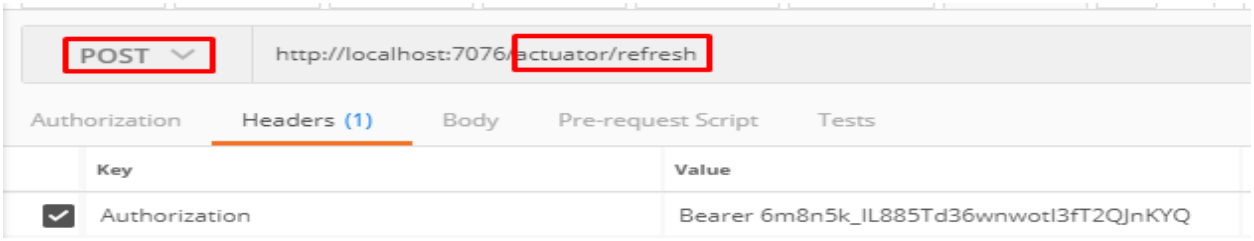
Update Configuration and Trigger Refresh Event

Change the value of **todaysFuelRate** property value in **config-driver-service.properties** and commit the changes to the repository.

Hit the Url http://localhost:7076/api/drivers and see the log file foe the todaysFuelRate value. Updated value is not reflected yet. Still the old value is showing.

This is because refresh event is not yet triggered for the application and the values that are loaded in client bean during server start up and reflecting.

Lets trigger the /actuator/refresh endpoint, for driver-service as shown below:



This will results in refreshing the beans annototated with @RefreshScope, since the DriversController is annototated with @Refreshscope, its properties also got refreshed.

Maintaining Common properties Across the Microservices

1. Any property added to the application.property file of the remote repository will be shared among all the microservices. There fore, if we want some properties (like database, security, mail server etc) to be shared among all the microservices, we can add them into this file in the git configuration repository. The Spring Cloud-Config server will enables these properties to available for all services as shared properties.

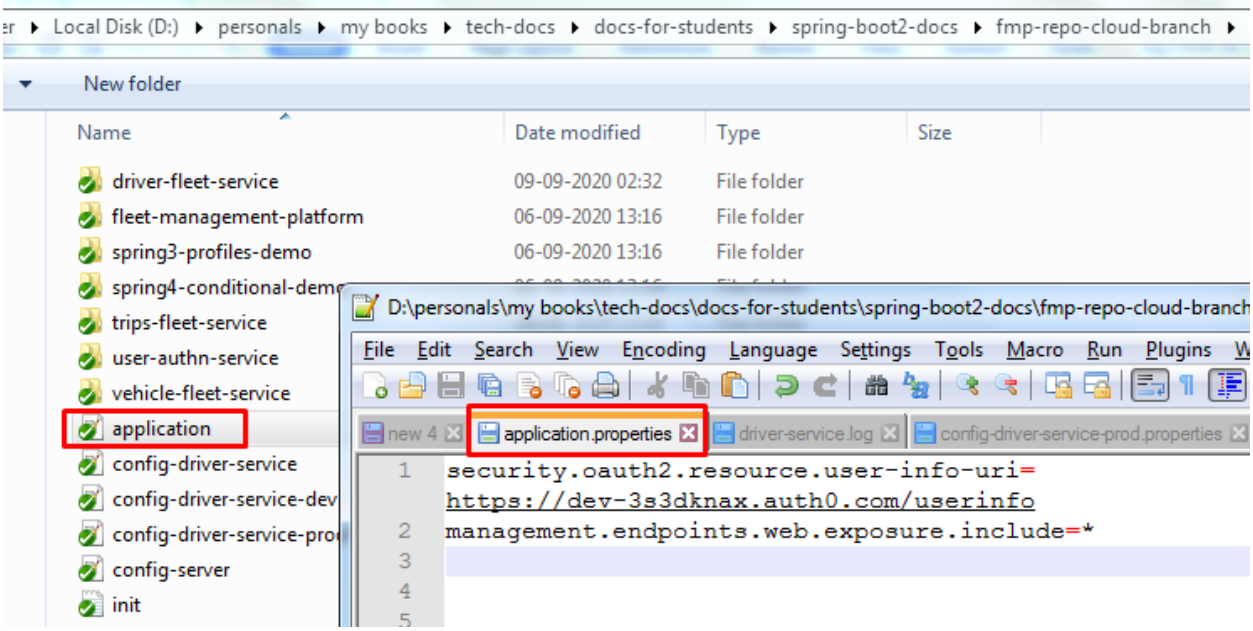
2. If we want to load properties before applicationContext created, we can add the properties into bootstrap.properties. It will be loaded before the application.properties are loaded. It will help to create a bootstrap context which is a parent context of the main application context.

Common properties Implementation:

1. Commit a file named application.properties under the git repo.

2. Lets add the property mail.server.port=5443

3. Update the microservices code to get this property value.



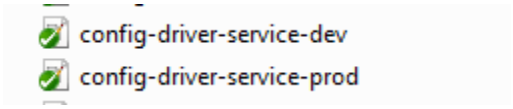
Maintaining Profiles

Lets add the below steps for the implementation:

1. Add spring.profiles.active=prod in clients (microservice) application.properties file to activate the production profile. This means only production related configuration is available to the microservice.

2. Restart the driver-service to reflect the prod profile.

3. Commit the different environment properties in the git repository as shown below:



4. HIt the driver-service to get all drivers

Config Clients- Search Paths