Centralized Configuration

In this tutorial we will:

1. Create a Configuration Server connected to GitHub repository to read configurations for its responses.
2. Create a Configuration Client to consume configurations from the Configuration Server.
3. Test and explanation
4. See an example to use a Environment Variables to get configuration value

1. Create a Configuration Server

Create a new spring boot project using gradle.

Add dependency to the build.gradle file:

```
implementation 'org.springframework.cloud:spring-cloud-config-
server'
```

Create SpringBootApplication. Add the annotation ***@EnableConfigServer*** to allow Configuration Server:

```
@EnableConfigServer
@SpringBootApplication
public class ServerConfigDemoApplication {

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

Create a new repository in GitHub. Add 2 files to it:

1. application.properties
2. demo-config-client.properties

The content of the files:

application.properties

```
message=Hello World! application properties

errorMessage=ERROR in Hello World! application properties
```

demo-config-client.properties

```
message=Hello World! demo config client properties
```

** **IMPORTANT:** Make sure your files are committed.

Define the below configuration in the application.properties of the Configuration Server project:

/src/main/resources/application.properties

```
server.port=9099

spring.cloud.config.server.git.uri=https://github.com/naordavid2/twelve.factor.app.final.demo
```

** **NOTICE:** Use the URL of the GitHub repository which you created.

Run the **ServerConfigDemoApplication**

2. Create a Configuration Client

Create a new spring boot project using gradle.

Add dependency to the build.gradle file:

```
implementation 'org.springframework.boot:spring-boot-starter-actuator'
implementation 'org.springframework.boot:spring-boot-starter-web'
implementation 'org.springframework.cloud:spring-cloud-starter-config'
```

Create SpringBootApplication.

```
@SpringBootApplication
public class ClientConfigDemoApplication {

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

Create a RestController:

```
@RestController
public class MessageController {

	@Value("${message:Hello world! default value}")
	private String message;

	@Value("${errorMessage:ERROR in Hello world! default value}")
	private String errorMessage;

	@RequestMapping(path="/hello",
				method=RequestMethod.GET)
	public String sayHello() {
		return this.message;
	}

	@RequestMapping(path="/hello/error",
			method=RequestMethod.GET)
	public String sayErorHello() {
		return this.errorMessage;
	}
}
```

Define configurations in the application.properties:

/src/main/resources/application.properties

```
server.port=9091

management.endpoints.web.exposure.include=*
```

** **NOTICE:** Use the URL of the GitHub repository which you created.

Create and define configurations in the bootstrap.properties:

/src/main/resources/ bootstrap.properties

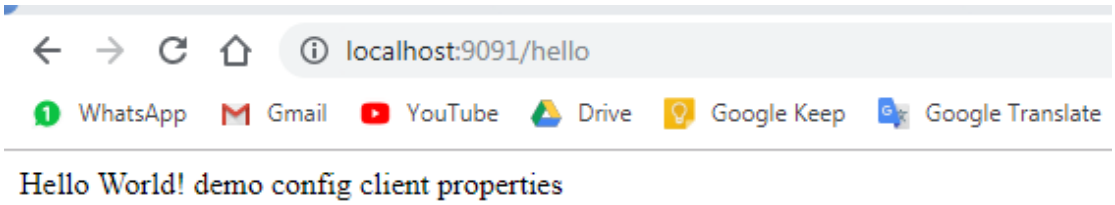
```
spring.application.name=demo-config-client

spring.cloud.config.uri=http://localhost:9099
```

** **NOTICE:** *spring.cloud.config.uri* is the URL of the Configuration Server.
** **INFO:** We define these configurations in the bootstrap.properties since we need them before the Application instanced.
** **ADDITIONAL:** See part 4 for example for injection of the Configuration Server URL using the Environment Variables.

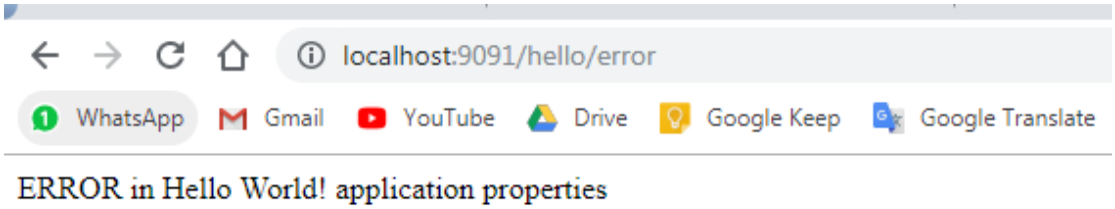
Run the **ClientConfigDemoApplication**

3. Test and explanation

Make a GET request: http://localhost:9091/hello
Then we get the response:



Make a GET request: http://localhost:9091/hello/error
Then we get the response:



Explanation:

The Configuration Server receives a request for configurations from a client which is recognized by the ***spring.application.name*** configuration, our client application name is ***demo-config-client***

The Configuration Server searches for a properties file with the application name and read the configuration values from it. In addition, it reads the *application.properties* file. Then, there is priority to configuration value that exists in the application name properties file.

Well, in our case, the "message" configuration exists in both files - *application.properties* and *demo-config-client.properties*. So the value was taken from the *demo-config-client.properties* file.

Same way, the "errorMessage" configuration exists only in *application.properties* file. So the value was taken from the *application.properties* file.

4. An example of using an Environment Variable to get configuration value

As we saw in part 2, we created and define configurations in the **bootstrap.properties** file**.**

One of the configurations was the Configuration Server URL - ***spring.cloud.config.uri***

Instead of defining this configuration value in the **bootstrap.properties** file, we can get it from the Environment Variables and manage its value there.

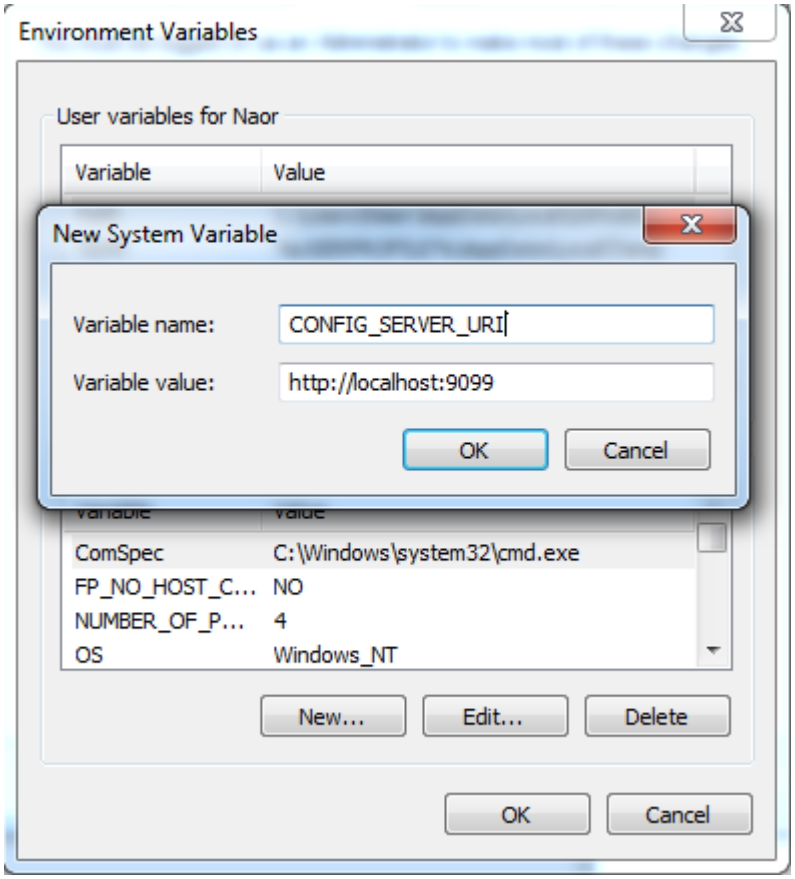
Let's change the **bootstrap.properties** file in the Client project:

/src/main/resources/ bootstrap.properties

```
spring.application.name=demo-config-client

spring.cloud.config.uri=${CONFIG_SERVER_URI:Undefined - please
define CONFIG_SERVER_URI env variable with Configuration Server URL}
```

Here we used the '' Environment Variable value. In Windows we define this variable as follow:
Control Panel\System and Security\System > Click "Advanced System Settings" > Go to "Advanced" tab > Click " Environment Variables"

Add new System Variable and approve all:



Now, the value will be taken from this variable.

** **IMPORTANT:** This change might take effect only after restarting Eclipse.

** **Credit:** https://spring.io/guides/gs/centralized-configuration/