Recomendaciones:

* Dedicarle al menos 28 minutos diarios al curso.
* Después de cada lección Udemy preguntará si estamos listos para hacer una revisión o no.

NOTAS: todos los pasos de las lecciones se encuentran disponibles en un repositorio de GIT: <https://github.com/in28minutes/spring-microservices>

De lo destacado del curso:

* Aprenderá la implementación de micro servicios con spring boot y spring cloud.

Enfocándose con los servicios web RESTful, y los micro servicios de la nube. (Crux)

* Documentación de los micro servicios, control de versiones de los micro servicios, monitoreo de los micro servicios, negociación e internacionalización.
* Herramientas que se usaran en el curso
* Spring
* Spring boot
* JPA
* Maven
* Eclipse IDE
* Balanceo de carga implementado: Ribbon, Feing e implementando un servidor configurado con Eureka.
* Para el monitoreo de las aplicaciones se utilizara: API Zuul, Zipkin
* Para el manejo de la tolerancia a fallas, utilizaremos: Hysterix

**Temas importantes del curso:**

* Tipos de servicios web
  + SOAP
  + RestFul
* Definiciones importantes de los servicios web
  + Formas de definir los servicios web

@RequestMapping(method = RequestMethod.***GET***, path = "/hello-world")

@GetMapping(path = path = "/hello-world")

* + Convertir objetos json a bean de java y viceversa
  + Paso de información a web services a través de la URI, body, header.
  + Manejo de exceptions en los servicios web.
  + Incluir las url de otros servicios como parte de la respuesta de un web service.
  + Validación de la información que se envía a los servicios web.
    - URI
    - Hateoas
  + Configuración de la internacionalización de los servicios web. (i18n)
  + Configuración para que los servicios web permitan tanto el manejo de objetos JSON como XML.
  + Configuración de SWAGGER para la documentación de los web services.
* Monitoreo de web services.
* Manejo de versiones.
* JPA. (Utilizando una base de datos H2 – Base de datos en memoria)
  + Dependencias requeridas.
  + Configuración de la base de datos H2.
  + Cómo insertar registros en la base de datos.
  + Cómo ingresar a la consola de la base de datos H2.
  + Creación y consumo de servicios web implementado JPA
  + Relaciones ManyToOne , OneToMany mediante JPA.

NOTA: tener cuidado cuando se usen las relaciones ManyToOne , OneToMany.

En el atributo de clase donde se especifica la relación @ManyToOne se debe considerar la anotación @JsonIgnore para evitar que la información de la respuesta se cicle (Ver ejemplos, donde se detalla este tema).

* Micro servicios.
  + Definiciones importantes.
  + Creación se micro servicio básico.
    - Dar de alta el nombre de la aplicación
    - Asignar el puerto de la aplicación.
    - Mapear los valores del archivo application.properties a una clase java
  + **Creación de micro servicio con Configuration Spring Cloud.**
    - Dar de alta el nombre de la aplicación
    - Asignar el puerto de la aplicación.
    - Mapear los valores del archivo application.properties a una clase java
    - Creación y sincronización con un repositorio de git
    - Agregar el repositorio git a un proyecto de spring boot
    - Centralizar la configuración en un servidor de configuraciones.
    - Conectar el servidor de configuraciones con el repositorio de git
    - Generar archivos de configuración para ambientes Dev, QA, Default
  + **Creación de micro servicios gestionando las configuraciones en un servidor de configuraciones. (Centralización de las configuraciones)**
  + **Interacción de micro servicios** 
    - Creación de micro servicios implementando JPA y una base de datos H2. (Base de datos en memoria)
      * Configuración de JPA
      * Configuración de base de datos H2.
      * Habilitación de la consola de la base de datos H2
  + **Interacción de micro servicios implementando Fein para facilitar la interacción entre los micro servicios.**
  + **Balanceo de carga de los micro servicios mediante Ribbon**

----

Especificación del servicio.

* Definición del formato de la solicitud y de la respuesta. (XML, Json, etc…)
* Definir el formato de la solicitud. (**Request** 🡪 es el input del servicio web)
* Definir el formato de la respuesta. (Response 🡪 es el output del servicio web)
* Definir cómo se va a invocar el servicio. (Endpoint)

**Terminologías clave de los servicios web.**

* Request, Response
  + El request es el input del servicio web.
  + El response es el output del servicio web.
* Formato del intercambio de mensajes.

Definir si se va a manejar un formato XML, Json, etc…

* Proveedor del servicio o servidor.

Servidor donde se encuentra alojado el servicio web.

* Consumidor del servicio o cliente.

Por ejemplo, las aplicaciones son consumidores de los servicios web.

* Definición del servicio.

Es el contrato entre el proveedor del servicio y el consumidor del servicio.

En el contrato de define:

* + El formato del request y del response. (Si se va a usar XML, Json, etc…)
  + La estructura del request.
  + La estructura del response.
  + La definición del **EndPoint.** (
    - Definición de la **URL** donde se encontrara disponible el servicio web.
    - Se define la forma en como los clientes pueden consumir el servicio web.
* Transporte

Donde se define cómo se llama a un servicio web.

* + HTTP (Hyper Text Transfer Protocol) 🡪internete

La invocación del servicio web se hace a través de internet, mediante una URL.

* + MQ 🡪 web

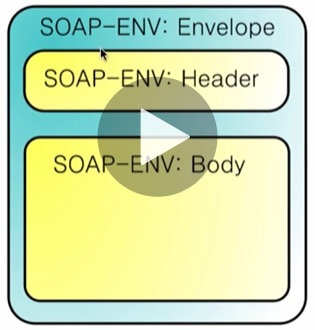
La invocación del servicio web se hace a través de una cola (MQ), por ejemplo, Websphere MQ, Oracle, etc..

**Grupos de servicios web.**

* Basados en **SOAP**. (Protocol Access Object Simple)

Plantea restricciones en el formato XML que se intercambian entre su proveedor de servicios y los consumidores del servicio.

Los servicios basados en soap deben tener el siguiente formato:



El **header** contiene meta información como autenticación, autorización, firmas, y cosas de este estilo.

El **body** contiene información del request y del response. (Estructura y demás cosas del request y del response)

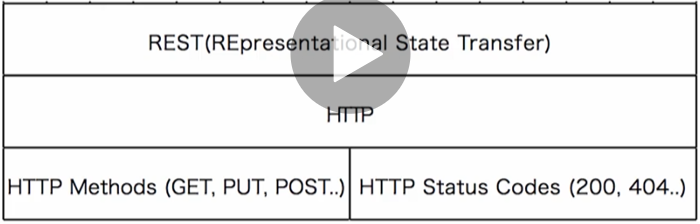
Por lo tanto los servicios basados deben cumplir lo siguiente:

* + Formato
    - SOAP XML Request
    - SOAP XML Response
  + Transporte
    - SOAP sobre MQ
    - SOAP sobre HTTP
  + Definición del servicio.
    - WSDL (Web Service Deifinition Language) aquí es donde se define:
      * EndPoint.
      * Todas las operaciones del servicio web.
      * Se define la estructura del Request.
      * Se define la estructura del Response.
* Tipo **Rest**. (REpresentational State Transfer)

Define un enfoque arquitectónico. Por lo que siempre que hablemos de servicios Rest, debemos pensar en **recursos**. (Los recursos que nos ofrece este tipo de servicios)

Los servicios tipo Rest los crea Roy Fielding, él mismo que desarrollo el protocolo HTTP.

Los servicios Rest se generan con la intensión de hacer un mejor uso del protocolo HTTP.



Abstracciones clave de los recursos.

* + Una de las cualidades de este tipo de servicios web es que se le asigna una **URI** (Uniform Resource Identifier) a cada uno de los **recursos** del servicio, por ejemplo:
    - obternerUsuario🡪GET 🡪 /usuario/obtenerUsuario/1
    - obtenerUsuarios 🡪 GET 🡪 /usuario/obtenerUsuarios/
    - crearUsuario 🡪 POST 🡪 /usuario/crearUsuario/
    - actualizarUsuario 🡪PUT 🡪 /usuario/actualizarUsuario/
    - eliminrUsuario 🡪 DELETE 🡪/ usuario/eliminarUsuario/
  + Un recurso puede tener diferentes representaciones
    - XML
    - HTML
    - JSON
  + Siempre que hablemos de servicios de tipo **rest**,
    - Debemos de pensar en **recursos**. (los recursos que nos ofrece este tipo de servicios)
    - Hacer uso del HTTP.
    - Para realizar alguna operación sobre los recursos, considerando los verbos (GET, POST, UPDATE, DELETE)
  + Formato de intercambio de datos.
    - No tiene restricciones, JSON es muy popular.
  + Transporte
    - Únicamente HTTP.
  + Definición del servicio.
    - NO existe un estándar. Por lo que puede usar WALD (Web Application Description Language), Swagger, etc…

**Web services RestFul vs. Web services SOAP.**

RestFul

* Basado en una definición arquitectónica.
* Formato de intercambio de datos:

Puede ser en XML, Json, etc..

* Definición de lenguaje:

No tiene definido un lenguaje. (Aunque se puede usar WADL)

* Transporte

Únicamente HTTP.

* Son más fáciles de implementar.

SOAP

* Formato de intercambio de datos:

El intercambio de datos está basado en XML. (request y response)

* Definición de lenguaje:

WSDL

* Transporte

HTTP, MQ

* Más difíciles de implementar, ya que se tiene que definir su WSDL. (Entre otros)

========================================================================================================================================================================================================

Sitio para generar bases de proyectos java con spring boot. <http://start.spring.io>

Referencias para resolver issues de los proyectos que se generen en las prácticas: <https://github.com/in28minutes/in28minutes-initiatives/blob/master/The-in28Minutes-TroubleshootingGuide-And-FAQ/quick-start.md>

**Creación de primer proyecto. (Servicios RestFul)**

* Se genera el proyecto **restful-web-services**
* Considere los siguientes escenarios:
  + Identificación de servicios de **User**.
    - Crear un user 🡪 POST 🡪 /createUsers/
    - Leer todos los user 🡪 GET 🡪 /getUsers/
    - Leer un solo user 🡪 GET 🡪/getUsers/{id}
    - Actualizar un user 🡪 UPDATE 🡪/updateUsers/
    - Eliminar un user 🡪 DELETE 🡪 /deleteUsers/
  + Identificación de **permisos** de un **user**. (con cardinalidad 1..\* 🡪 Un usuario tiene 1..\* permisos asignados.)
    - Obtener todos los permisos de un user 🡪 GET 🡪 /users/{id\_user}/permisos
    - Crear un permiso a un user 🡪 POST 🡪 /users/{id\_user}/permisos
    - Obtener un permiso de un user 🡪 GET 🡪 /users/{id\_user}/permisos/{id\_permiso}
* Creación de servicio **hello-world**
  + La clase se debe etiquetar con **@RestController** para indicar que la clase atiende peticiones http.
  + El método se debe etiquetar con **@RequestMapping** para indicar el **tipo de servicio** (GET, POST, PUT, DELETE) y la **URI del servicio**.

@RequestMapping(method = RequestMethod.***GET***, path = "/hello-world")

@RequestMapping(method = RequestMethod.POST, path = "/hello-world")

@RequestMapping(method = RequestMethod.PUT, path = "/hello-world")

@RequestMapping(method = RequestMethod.DELETE, path = "/hello-world")

Otra forma de especificarlo es mediante las anotaciones específicas:

* + - @GetMapping(path = path = "/hello-world")
    - @PostMapping(path = path = "/hello-world")
    - @PutMapping(path = path = "/hello-world")
    - @DeleteMapping(path = path = "/hello-world")
  + Ejemplo: **conversión de bean a json** (La conversión de bean a json y de json a bean se hace mediante **Jackson**)

Los servicios tipo GET se pueden invocar desde una aplicación, navegador o postman.

/\*\*

\* Si en la clase HelloWorldBean no se define el getter del atributo de clase donde se almacena el valor

\* que se pasa como parámetro en el constructor, de genera un error.

\* La conversión del bean HelloWorldBean al Json se hace automáticamente.

\* **@return**

\*/

@GetMapping(path = "/hello-world-bean")

**public** HelloWorldBean getHelloWorldBean() {

**return** **new** HelloWorldBean("Hello World.");

}

* + **Ejemplo (pasando parámetros a la URI)**

Los servicios tipo GET se pueden invocar desde cualquier navegador o postman.

@GetMapping(path = "/hello-world/path-variable/{name}")

**public** HelloWorldBean getHelloWorldBean(@PathVariable String name) {

**return** **new** HelloWorldBean(String.*format*("Hello World %s", name) );

}

**@PathVariable** para indicar al servicio web que se le está pasando un parámetro. (En el path del servicio se pone el nombre de la variable entre parentesis)

Y la forma de invocar el servicio es: <http://localhost:8081/hello-world/path-variable/Curso> de micro servicios

Donde el parámetro es: [Curso](http://localhost:8081/hello-world/path-variable/Curso) de micro servicios

* + **Ejemplo Get personalizando exception.**

Se recomienda que en todos los casos se haga un manejo de las exceptions a nivel empresarial. (Con el objetivo de manejarlas de la misma forma)

@GetMapping("/users/{id}")

**public** User retrieveUser(@PathVariable **int** id) {

User user = userDaoService.findOne(id);

**if**( user == **null** )

**throw** **new** UserNotFoundException("id-"+id);

**return** user;

}

**La Exception personalizada es:**

@ResponseStatus(HttpStatus.***NOT\_FOUND***)

**public** **class** UserNotFoundException **extends** RuntimeException {

/\*\*

\*

\*/

**private** **static** **final** **long** ***serialVersionUID*** = 7660278402019240607L;

**public** UserNotFoundException(String arg0) {

**super**(arg0);

}

}

* + **Ejemplo Get definiendo un controlador (controller) de exceptions.**

@GetMapping("/users/{id}")

**public** User retrieveUser(@PathVariable **int** id) {

User user = userDaoService.findOne(id);

**if**( user == **null** )

**throw** **new** UserNotFoundException("id-"+id);

**return** user;

}

**Las clases requeridas para la definición del manejo de las exception, son: ExceptionResponse, CustomizedResponseEntityExceptionHandler.**

**Mediante la anotación @ControllerAdvice se indica que el recurso se debe compartir/heredar a todos los controller, por lo que no es necesario realizar ninguna configuración adicional.**

1. Definición del Bean.

**public** **class** ExceptionResponse {

**private** Date timestamp;

**private** String message;

**private** String details;

**public** ExceptionResponse(Date timestamp, String message, String details) {

**super**();

**this**.timestamp = timestamp;

**this**.message = message;

**this**.details = details;

}

…

//Definición de getter/setter

}

1. Creación de la clase para el manejo de las exception

@ControllerAdvice

@RestController

**public** **class** CustomizedResponseEntityExceptionHandler **extends** ResponseEntityExceptionHandler {

//Para el uso exclusivo de exceptions del tipo Exception

@ExceptionHandler(Exception.**class**)

**public** **final** ResponseEntity<Object> HandleAllExceptions(Exception ex, WebRequest request){

ExceptionResponse exceptionResponse = **new** ExceptionResponse(**new** Date(), ex.getMessage(), request.getDescription(**false**));

//Se define manejo de exceptions del tipo HttpStatus.INTERNAL\_SERVER\_ERROR

**return** **new** ResponseEntity(exceptionResponse, HttpStatus.***INTERNAL\_SERVER\_ERROR***);

}

//Para el uso exclusivo de exceptions de tipo UserNotFoundException

@ExceptionHandler(UserNotFoundException.**class**)

**public** **final** ResponseEntity<Object> HandleUserNotFoundExceptions(UserNotFoundException ex, WebRequest request){

ExceptionResponse exceptionResponse = **new** ExceptionResponse(**new** Date(), ex.getMessage(), request.getDescription(**false**));

//Se define manejo de exceptions del tipo HttpStatus.NOT\_FOUND

**return** **new** ResponseEntity(exceptionResponse, HttpStatus.***NOT\_FOUND***);

}

}

**Donde:**

**ResponseEntityExceptionHandler** 🡪 Para extender del manejo de Exceptions que nos proporciona spring.

**@RestController** 🡪 Para indicar que es un controller.

**@ControllerAdvice** 🡪 Para indicar que este recurso se debe compartir entre todos los controller.

@ExceptionHandler(Exception.class) -- Para indicar que se manejan todas las exceptions

* + **Ejemplo Post (mapea el contenido del body a un objeto java (entity, bean, etc…))**

Los servicios tipo Post solo se pueden invocar desde una aplicación o postma. (No se pueden invocar desde un navegador)

@PostMapping("/users")

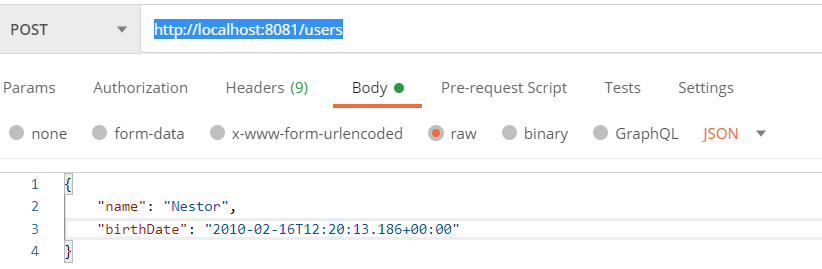
**public** User createUser(@RequestBody User user) {

User usr = userDaoService.save(user);

**return** usr;

}

**@RequestBody** para hacer el mapeo del contenido del body del request con el objeto java, en este caso el entity User. (Los nombres del json del body deben ser iguales a los nombres del entity y este último debe tener los getter/setter correspondientes, de no tener los getter/setter se genera un error)



* + **Ejemplo Post (Devolviendo http status y location del objeto creado)**

@PostMapping("/users")

**public** ResponseEntity<?> createUser(@RequestBody User user) {

User saveUser = userDaoService.save(user);

//Obtiene la URI del user persistido.

URI location = ServletUriComponentsBuilder.*fromCurrentRequest*().path("/{id}").buildAndExpand(saveUser.getId()).toUri();

**return** ResponseEntity.*created*(location).build();

}

**ServletUriComponentsBuilder** Se encuentra disponible en la dependencia spring-boot-starter-web

* + **Ejemplo Post (Agregando validaciones a la información que se pretende persistir)**

Para ello se necesita de la dependencia.

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-validation</artifactId>

</dependency>

@PostMapping("/users")

**public** ResponseEntity<?> createUser(@Valid @RequestBody User user) {

User saveUser = userDaoService.save(user);

//Obtiene la URI del user persistido.

URI location = ServletUriComponentsBuilder.*fromCurrentRequest*().path("/{id}").buildAndExpand(saveUser.getId()).toUri();

**return** ResponseEntity.*created*(location).build();

}

Con la instrucción **@Valid** se indica que se tiene que validar que la información cumpla con las especificaciones indicadas en el **Bean User**, que en este caso y a manera de ejemplo se agregan validaciones a los campos nombre, birthDate:

* + - * @Size(min = 2, message = "El nombre debe contener al menos dos caracteres.")

**private** String name;

* + - * @Past(message = "La fecha de nacimiento no puede ser una fecha futura.")

**private** Date birthDate;

NOTA: Si no se cumplen la especificaciones anteriores, de genera un error, mismo que se controla el método **handleMethodArgumentNotValid**

@ControllerAdvice

@RestController

**public** **class** CustomizedResponseEntityExceptionHandler **extends** ResponseEntityExceptionHandler {

...

/\*\*

\* Exception para el manejo de validaciones en la información que se pasa en los request y se mapea a los bean de Java.

\*/

@Override

**protected** ResponseEntity<Object> handleMethodArgumentNotValid(

MethodArgumentNotValidException ex, HttpHeaders headers, HttpStatus status, WebRequest request) {

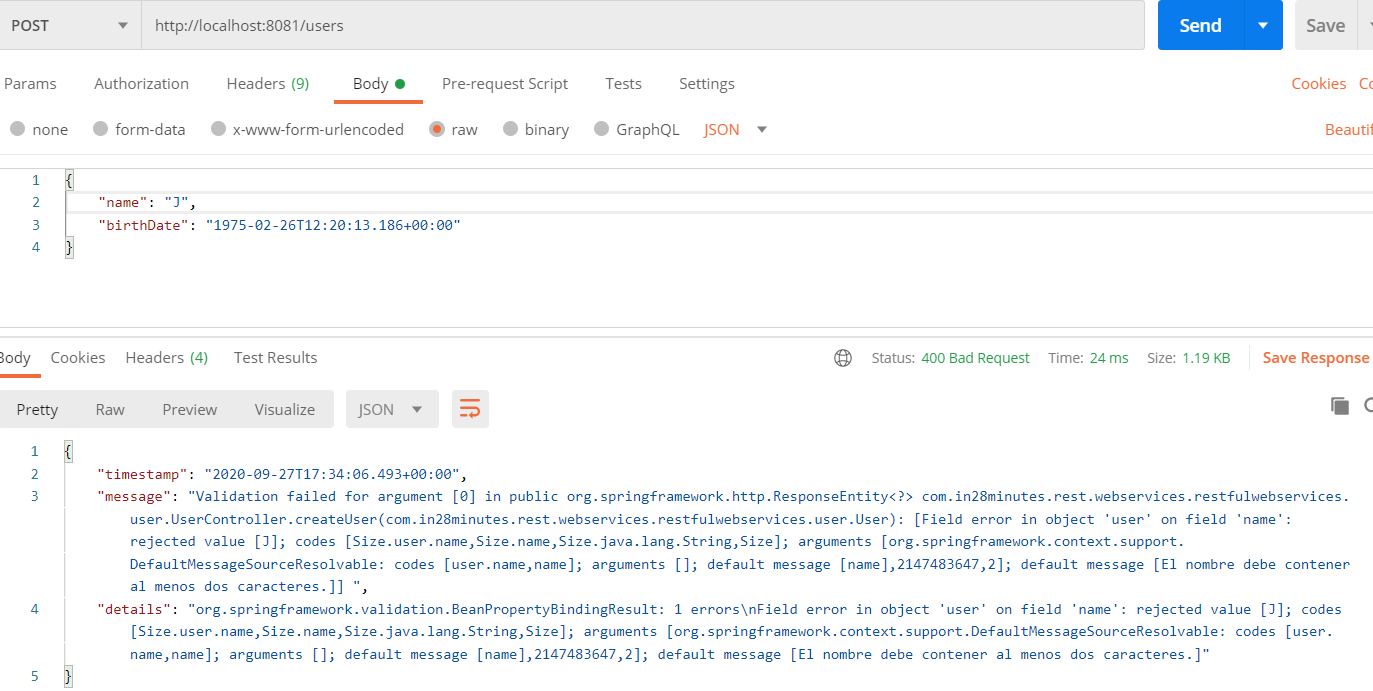
ExceptionResponse exceptionResponse = **new** ExceptionResponse(**new** Date(), ex.getMessage(), ex.getBindingResult().toString());

**return** **new** ResponseEntity(exceptionResponse, HttpStatus.***BAD\_REQUEST***);

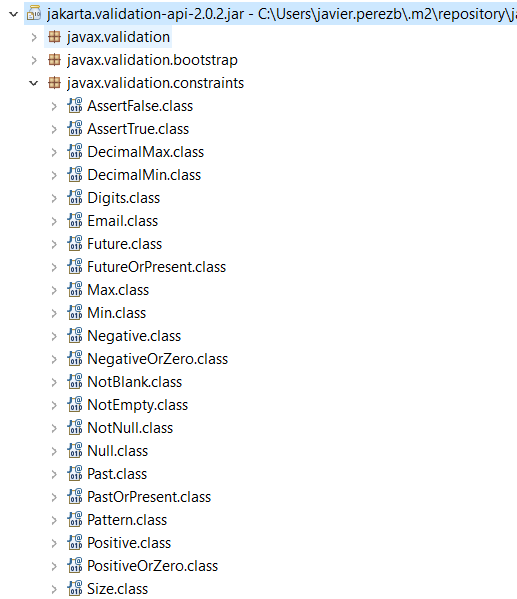
}

}

Test



Para más referencias sobre otros tipos de validaciones, consultar:



* **Agregar dependencia HATEOS, esta dependencia nos permite agregar enlaces a nuestros recursos muy fácilmente. (nos permite agregar las URL de otros servicios, útil para el concepto de aplicaciones WEB - Microservicios)**

**HATEOS** funciona como motor del estado de la aplicación. (Hipermedia As The Engine Of Application State - HATEOS)

* + Para ello debemos agregar la dependencia:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-hateoas</artifactId>

</dependency>

**NOTA**: Si la primera vez que agreguemos la dependencia no la encuentra, hay que hacer un refactor de la aplicación.

* + Para poder utilizar HATEOS, se debe importar la librería como **static**
    - Para versiones de spring-boot menores o igual a 2.2.0

**import** **static** org.springframework.hateoas.server.mvc.~~ControllerLinkBuilder~~.\*;

* + - Para versiones de spring-boot mayores a 2.2.0

**import** **static** org.springframework.hateoas.server.mvc.WebMvcLinkBuilder.*linkTo*;

**import** **static** org.springframework.hateoas.server.mvc.WebMvcLinkBuilder.*methodOn*;

**import** org.springframework.hateoas.EntityModel;

**import** org.springframework.hateoas.server.mvc.WebMvcLinkBuilder;

* + Ejemplo:

En el siguiente ejemplo, además de devolver la información de un user, también devolvemos los enlaces para consultar a todos los user.

Código importante:

//Para versiones de spring-boot menores o igual a 2.2.0

//import static org.springframework.hateoas.mvc.ControllerLinkBuilder.linkTo;

//import static org.springframework.hateoas.mvc.ControllerLinkBuilder.methodOn;

//import org.springframework.hateoas.Resource;

//import org.springframework.hateoas.mvc.ControllerLinkBuilder;

//Para versiones de spring-boot mayores a 2.2.0

**import** **static** org.springframework.hateoas.server.mvc.WebMvcLinkBuilder.*linkTo*;

**import** **static** org.springframework.hateoas.server.mvc.WebMvcLinkBuilder.*methodOn*;

**import** org.springframework.hateoas.EntityModel;

**import** org.springframework.hateoas.server.mvc.WebMvcLinkBuilder;

//Con esta versión se obtiene además de la información del user, los enlaces (URL's) para consultar a todos los user.

@GetMapping("/users/{id}")

**public** EntityModel<User> retrieveUser(@PathVariable **int** id) {

User user = userDaoService.findOne(id);

//Para versiones de spring-boot menores o igual a 2.2.0

//Resource<User> resource = new Resource<User>(user);

//ControllerLinkBuilder linkTo = linkTo(methodOn(this.getClass()).retrieveAllUsers());

//resource.add(linkTo.withRel("all-users"));

//return resource;

//Para versiones de spring-boot mayores a 2.2.0

EntityModel<User> resource = EntityModel.*of*(user);

WebMvcLinkBuilder linkTo = *linkTo*(*methodOn*(**this**.getClass()).retrieveUsers());//Donde retrieveUsers() es el servicio con el que se obtiene la información de todos los user

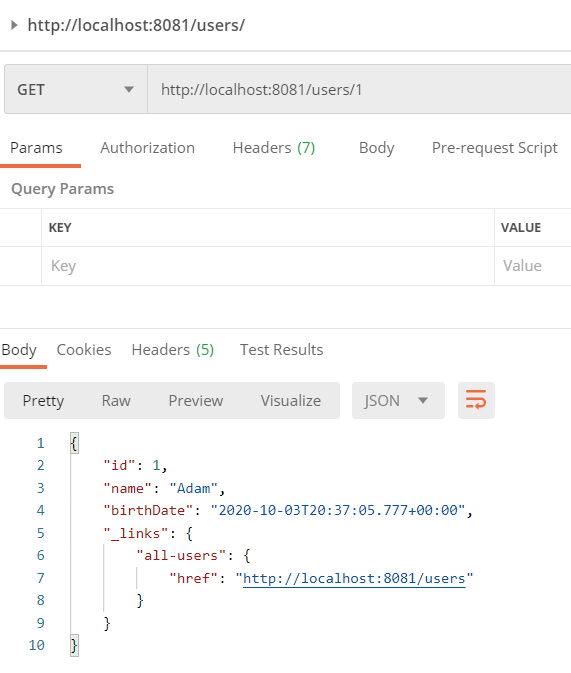
resource.add(linkTo.withRel("all-users"));

**return** resource;

}

El resultado es:

Además de la información del user, proporciona la URL para consultar la información de todos los user.



* **Configuración de internacionalización de mensajes**. (i18n)

Mediante la configuración de la internacionalización se configura el lenguaje a usar.

* + Para la configuración de i18n se requiere de:
    - Configuración del **LocalResolver**. (Esto se configura en la misma clase donde se encuentra el método main(…) en este caso y a manera de ejemplo se encuentra en la clase RestfulWebServicesApplication)

@Bean

**public** LocaleResolver localeResolver() {

SessionLocaleResolver localeResolver = **new** SessionLocaleResolver();

localeResolver.setDefaultLocale(Locale.***US***);

**return** localeResolver;

}

* + - Creación de .properties para definir el contenido del mensaje dependiendo del lenguaje que se utilice.
      * messages.properties 🡪 con el contenido good.morning.message=Good morning
      * messages\_fr.properties 🡪 con el contenido good.morning.message=Bonjour
      * messages\_mx.properties 🡪 con el contenido good.morning.message=Buenos días
    - Configuración de un **ResourceBoundleMessageSource**. (Esto se configura en la misma clase donde se encuentra el método main(…))

@Bean

**public** ResourceBundleMessageSource messageSource() {

ResourceBundleMessageSource messageSource = **new** ResourceBundleMessageSource();

messageSource.setBasename("messages");

**return** messageSource;

}

* + - Consumir el messageSource (ResourceBundleMessageSource) en nuestro controller. (En este caso y a manera de ejemplo utilizamos el controller HelloWorldController)

@Autowired

**private** MessageSource messageSource;

/\*\*

\* Para el manejo de la internacionalización. (i18n)

\* **@RequestHeader** para indicar que el parámetro se encuentra en el header del request.

\* y por consiguiente, el lenguaje se pasa como parámetro.

\* **@return**

\*/

@GetMapping(path = "/hello-world-internationalized")

**public** String getHelloWorldInternationalized(@RequestHeader(name="Accept-Language" , required = **false**) Locale locale) {

**return** messageSource.getMessage("good.morning.message", **null**, locale);

}

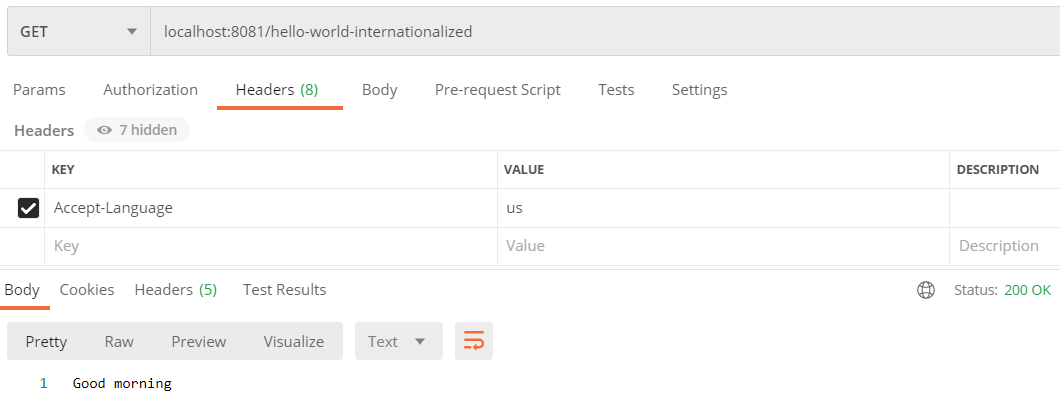
NOTA:

Con la anotación **@RequestHeader** le indicamos al servicio web que el parámetro se encuentra en el **header** del **request** y lo puede identificar con el nombre “Accept-Language” y que la información del parámetro lo mapee a la variable “**Local”**.

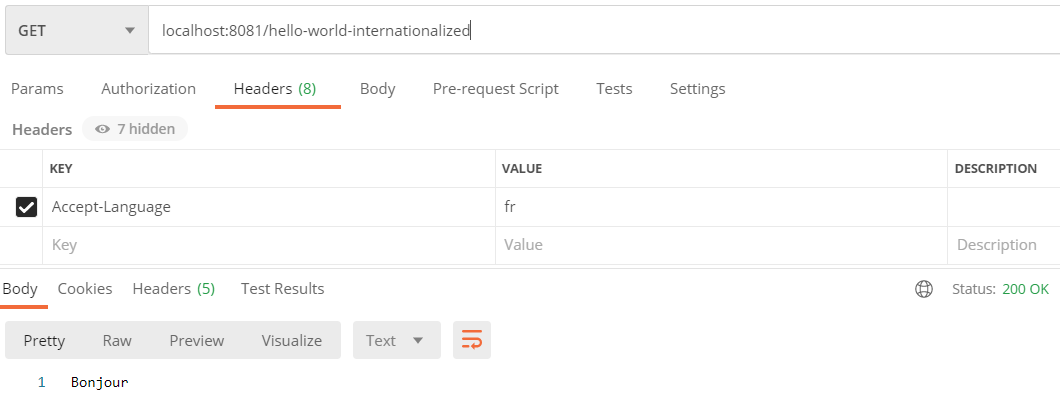
Con la bandera **riquire = false** se indica que no es requerido/obligatorio dicho valor. (En caso de no indicar un valor se toma uno por default)

* + - Resultado al consumir el servicio web

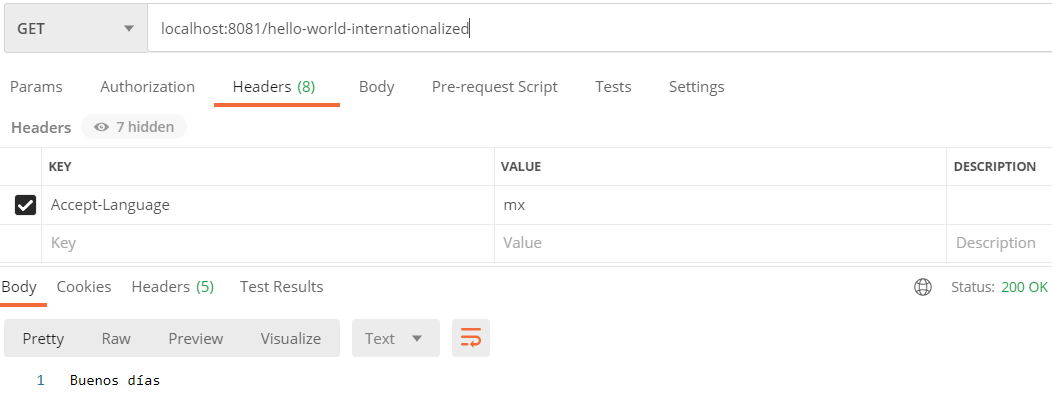
Pasando el valor Accept-Language=us



Pasando el valor Accept-Language=fr



Pasando el valor Accept-Language=mx



* + **Optimización de la internacionalización. (i18n)**

Realizar los siguientes ajustes:

* + - Obtener la internacionalización de **LocaleContextHolder**. (De esta forma se evita tener que pasar como parámetro en el header la internacionalización a aplicar)

//Optimizando la configuración se la internacionalización.

@GetMapping(path = "/hello-world-internationalized")

**public** String getHelloWorldInternationalized() {

**return** messageSource.getMessage("good.morning.message", **null**, LocaleContextHolder.*getLocale*());

}

* + - Modificar el método localResolver()

Cambiar el **SessionLocaleResolver** por **AcceptHeaderLocalResolve**

@Bean

**public** LocaleResolver localeResolver() {

AcceptHeaderLocaleResolver localeResolver = **new** AcceptHeaderLocaleResolver();

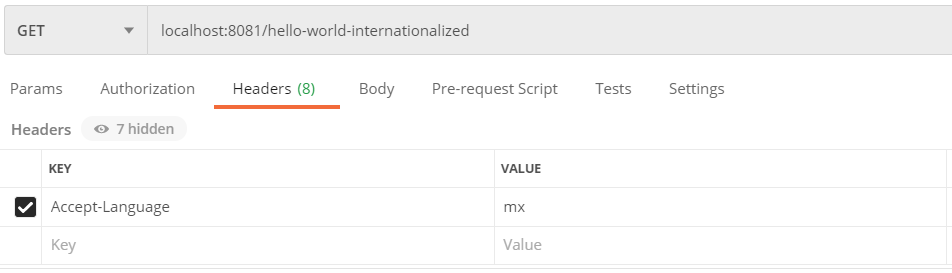
localeResolver.setDefaultLocale(Locale.***US***);

**return** localeResolver;

}

NOTA: la variable **AcceptHeaderLocaleResolver** está directamente relacionado con los parámetros que se pasan en el header del request. (Los valores seleccionados se mapean en automático al objeto AcceptHeaderLocaleResolver)

Para este ejemplo, si no se pasara ningún valor para Accept-Language, en automático se carga el valor por default indicado en el método **localResolver()**. (Locale.US)



* + - Agregar la siguiente línea al archivo **application.properties**

spring.massages.basename=messages

Esta línea está relacionada con los archivos .properties

* + - * messages.properties
      * messages\_fr.properties
      * messages\_mx.properties

Al agregar la línea anterior al **application.properties**, y comentar el siguiente Bean, la aplicación sigue funcionando. (Haga la prueba)

@Bean

**public** ResourceBundleMessageSource messageSource() {

ResourceBundleMessageSource messageSource = **new** ResourceBundleMessageSource();

messageSource.setBasename("messages");

**return** messageSource;

}

* **Servicios Rest con input/output en formato XML.**

Para lograr que un servicio RestFul maneje inputs/outputs en formato XML se requiere de lo siguiente:

* + Agregar la siguiente dependencia al pom.xml

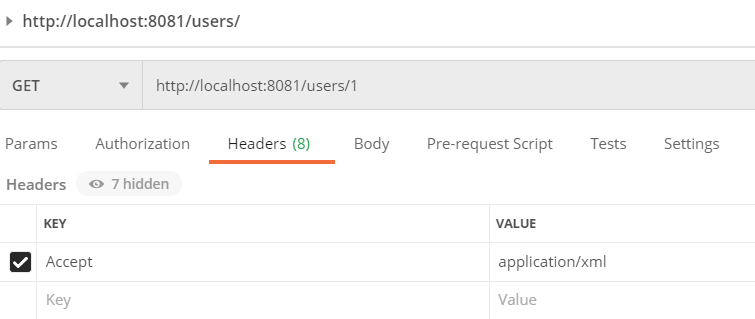
<dependency>

<groupId>com.fasterxml.jackson.dataformat</groupId>

<artifactId>jackson-dataformat-xml</artifactId>

</dependency>

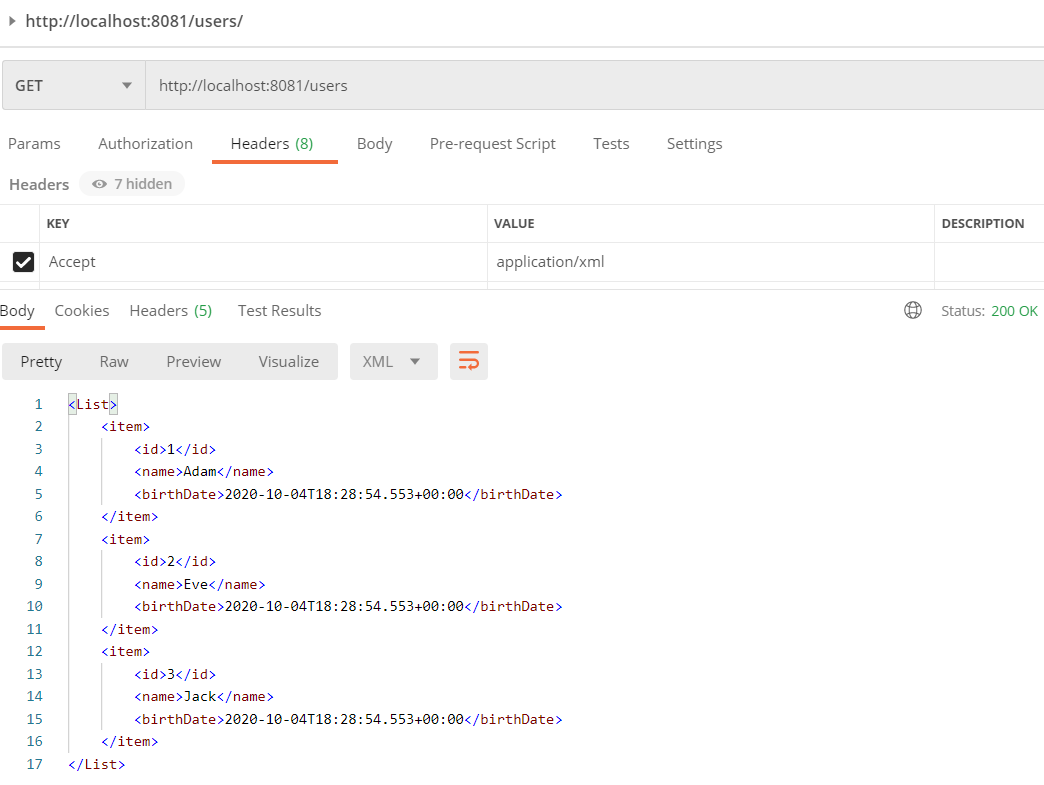
* + Configurar en el header del request el tipo de dato XML



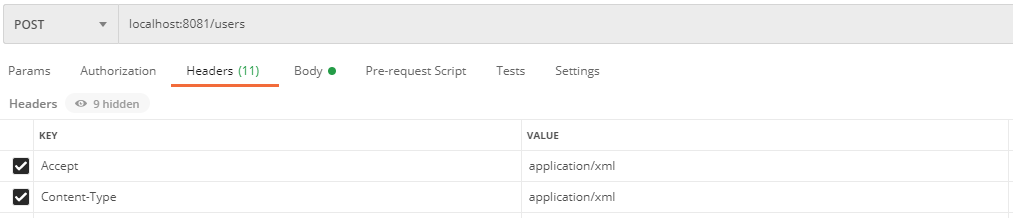
* + Ejemplo de servicio restful con formato XML

NOTAS: Se requiere hacer un refactor a la aplicación para que se contemple la nueva dependencia.

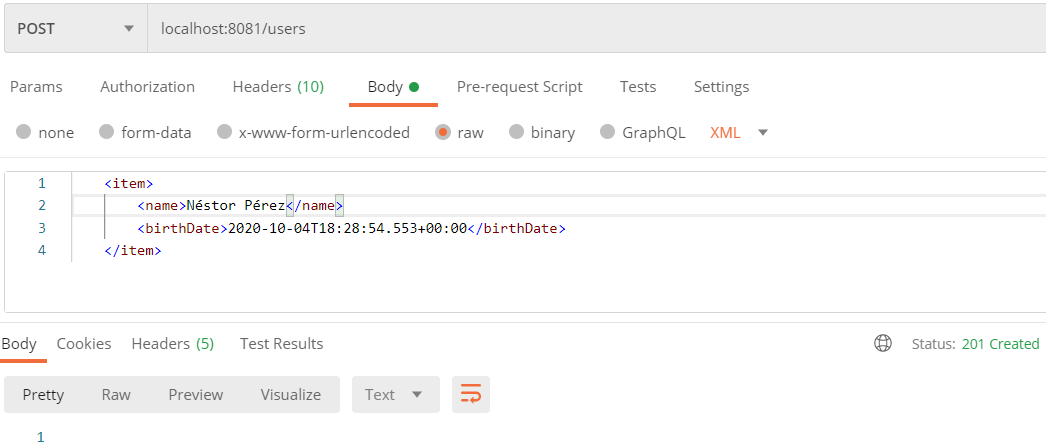
Importante configurar en el header el manejo de datos de tipo XML.



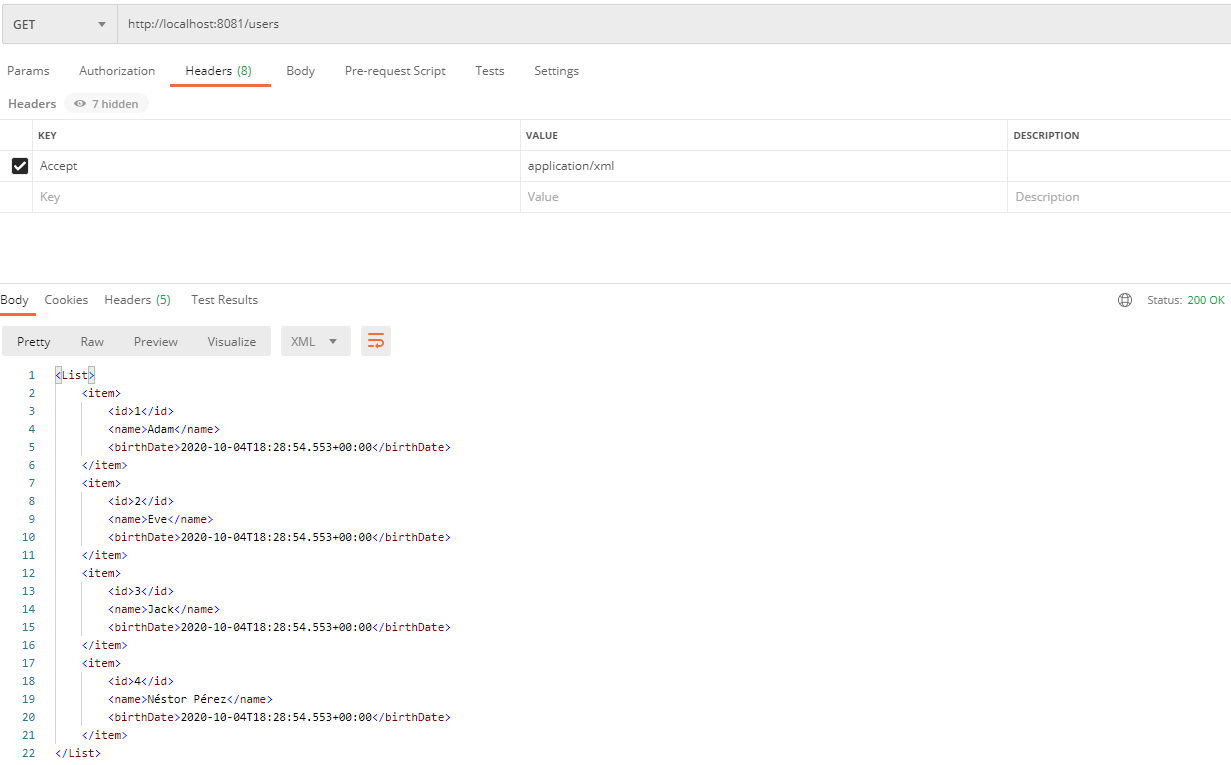
* + Ejemplo 2. Persistiendo la información de un user, se usa formato XML.



Note que aquí también se indica el formato. (XML)



Y al consultar la información, en efecto, se persistió la información del user.



* Configuración de Swagger. (Para obtener el contrato de los servicios y el formato de cada una de las solicitudes)

Swagger es uno de los formatos de documentación más populares para los servcios restful.

Para configurar Swagger en nuestro proyecto, necesitamos realizar lo siguiente (Tener cuidado con las versiones de las dependencias, ya que de no usar las versiones indicadas se pueden tener problemas para lograr los resultados deseados):

* + Agregar las siguientes dependencias al pom.xml

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger2</artifactId>

<version>3.0.0</version>

</dependency>

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger-ui</artifactId>

<version>2.9.2</version>

</dependency>

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-boot-starter</artifactId>

<version>3.0.0</version>

</dependency>

* + Crear la clase donde se configuran las propiedades requeridas de swagger.

La clase se crea en el package principal de nuestro proyecto. (Donde se encuentra la clase RestfulWebServicesApplication)

@Configuration

@~~EnableSwagger2WebMvc~~

**public** **class** SwaggerConfig {

@Bean

**public** Docket api() {

**return** **new** Docket(DocumentationType.***SWAGGER\_2***)

.select()

.apis(RequestHandlerSelectors.*any*())

.paths(PathSelectors.*any*())

.build();//En esta línea puede dejar solo el constructor o agregar las extensiones. (.select().apis(...).paths(...).build())

}

}

**NOTAS:**

Con la anotación **@Configuration** le indicamos a spring que es una clase donde se definen configuraciónes.

Con la anotación **@EnableSwagger2WebMvc** habilitamos swagger.

* + Los servicios los podemos consumir en las siguientes URL’s

Tal vez sea necesario hacer un refactor a la aplicación para que detecte las dependencias que se agregaron al pom.xml

**NOTA:**

Para que se tenga un formato adecuado de json de los servicios, tal vez sea necesario agregar la extensión al navegador. (JSON Formatter o algún otro)

* + - La documentación de los servicios la encontramos disponible en:

http://<IP>:<Port>/<contextPath>/v2/api-docs

Por ejemplo:

<http://localhost:8081/v2/api-docs>

* + - Los servicios de la aplicación los encontramos disponibles en:

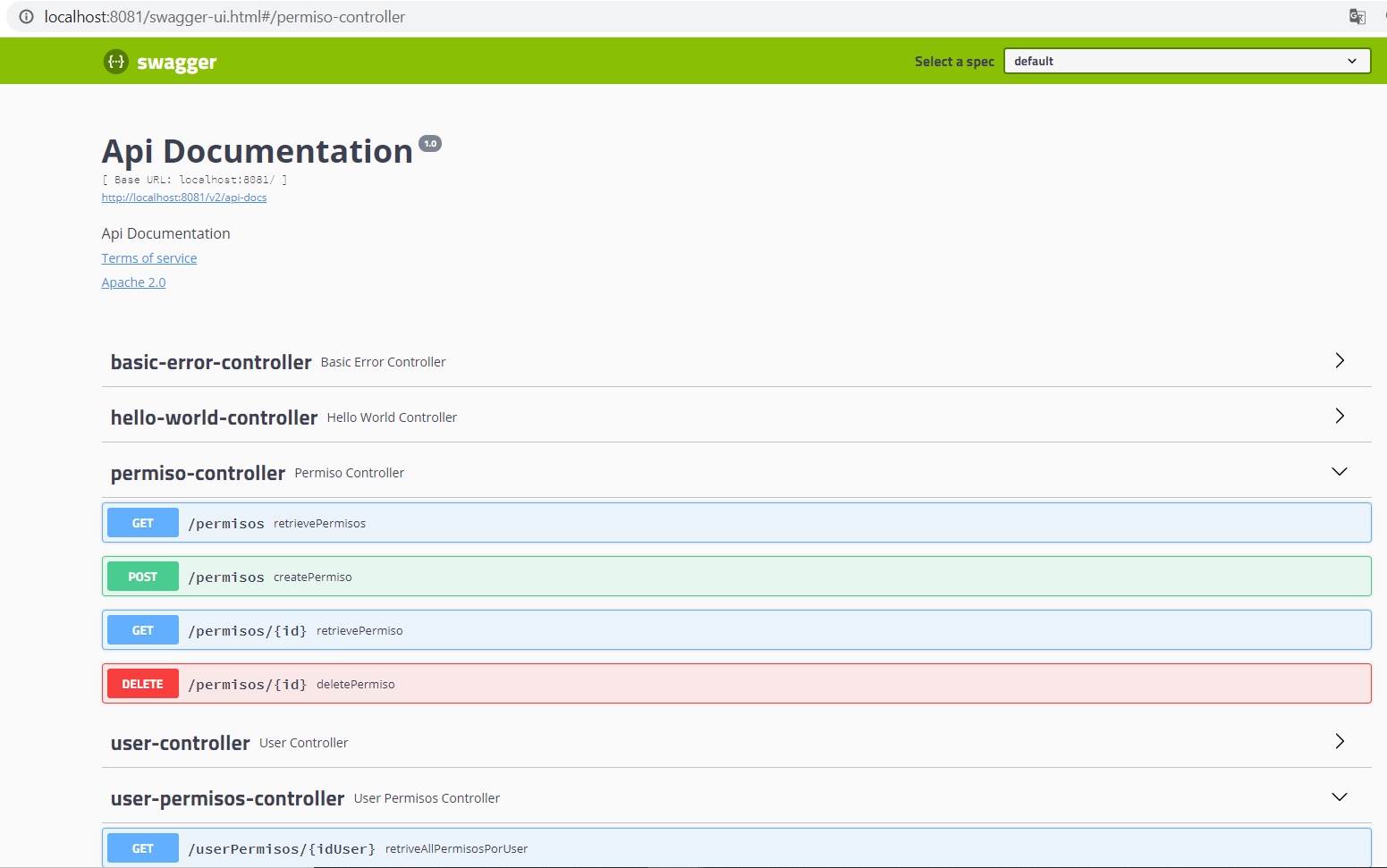
http://<IP>:<Port>/<contextPath>/swagger-ui.html

Por ejemplo:

<http://localhost:8081/swagger-ui.html>

* + - Ejemplos de los resultados obtenidos:





* + **Optimizando la configuración de swagger.**

El objetivo es personalizar la documentación de nuestros servicios. (Con el objetivo de proporcionar a los consumidores de los servicios información más puntual)

Por ejemplo, definir si los atributos tienen alguna restricción.

Por ejemplo, el nombre debe tener al menos 2 caracteres, la fecha de nacimiento no puede ser una fecha futura, etc…

* + - En la clase SwaggerConfig.java se deben hacer los siguientes ajustes:

**public** **static** **final** Contact ***DEFAULT\_CONTACT***

= **new** Contact(

"Javier P. B.",

"http://www.misitiooficial.com",

"pbjavouam@gmail.com");

**public** **static** **final** ApiInfo ***DEFAULT\_API\_INFO***

= **new** ApiInfo(

"Api Documentation - Curso Udemy micro servicios con spring boot",

"Api Documentation - Curso Udemy micro servicios con spring boot",

"1.0",

"urn:tos",

***DEFAULT\_CONTACT***,

"Apache 2.0",

"http://www.apache.org/licenses/LICENSE-2.0",

**new** ArrayList<>());

//Personalizando algunas secciones de la documentación de los servicios.

@Bean

**public** Docket api() {

**return** **new** Docket(DocumentationType.***SWAGGER\_2***).apiInfo(***DEFAULT\_API\_INFO***);

}

* + - El resultado se debe ver algo similar a:



* + Una optimización más, indiquemos que los servicios están preparados para manejar **json** como **xml**.

Para ello se deben hacer los siguientes ajustes:

* + - En la clase iii se deben hacer los siguientes ajustes:

Agregar el siguiente set:

**private** **static** **final** Set<String> ***DEFAULT\_PRODUCES\_AND\_CONSUMES*** = **new** HashSet<String>(Arrays.*asList*("application/json", "application/xml"));

Agregar el siguiente código al Bean:

//Personalizando algunas secciones de la documentación de los servicios e indicando que los servicios aceptan tanto JSON como XML.

@Bean

**public** Docket api() {

**return** **new** Docket(DocumentationType.***SWAGGER\_2***)

.apiInfo(***DEFAULT\_API\_INFO***)

.produces(***DEFAULT\_PRODUCES\_AND\_CONSUMES***)

.consumes(***DEFAULT\_PRODUCES\_AND\_CONSUMES***);

}

* + - El resultado se debe ver de la siguiente manera (Tal vez sea necesario reiniciar la aplicación para que vea los cambios):



* + **INDICAR EN LA DOCUMENTACION DE SWAGGER LAS ESPECIFICACIONES QUE DEBE TENER LA INFORMACIÓN DE LOS SERVICIOS.**

Se deben leer las propiedades que se especifican en los atributos de los entitys.

* + - Realizar los siguientes ajustes en los entitys.

@ApiModel(description = "Todos los detalles acerca de los user. ")

**public** **class** User {

**private** Integer id;

@Size(min = 2, message = "El nombre debe contener al menos dos caracteres.")

@ApiModelProperty(notes = "El nombre debe contener al menos dos caracteres.")

**private** String name;

@Past(message="La fecha de nacimiento no puede ser una fecha futura.")

@ApiModelProperty(notes = "La fecha de nacimiento debe ser una fecha en el pasado.")

**private** Date birthDate;

**…**

}

NOTAS:

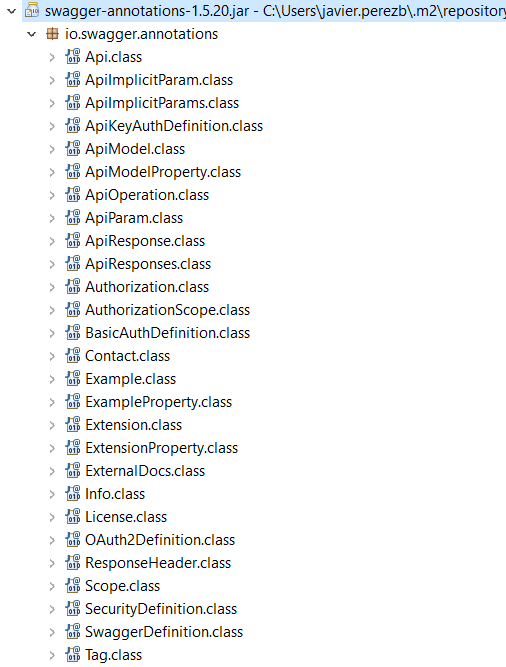
Con la anotación **@ApiModel** le indicamos a swagger que debe contemplar las propiedades de los atributos de la clase.

Con la anotación **@ApiModelProperty** indicamos el mensaje que se debe visualizar en la documentación de los servicios.

* + - El resultado se debe ver de la siguiente forma (Tal vez sea necesario que reinicie la aplicación para que vea los cambios):



* + - Referencia de otras anotaciones, consultar e contenido del siguiente .jar



* Agregar log a la aplicación.

En el archivo **application.properties** agregar la siguiente línea: logging.level.org.springframework = debug

* Para agregar un determinado formato de fecha.

En el archivo **application.properties** agregar la siguiente instrucción: **application.properties**

* Puntos importantes a considerar
  + dispatcherServlet to [\]

Sabe a partir de que ruta debe mapear los servicios solicitados. (URI)

* + Mediante las URI se mapean los servicios solicitados. (Se mapea la URI con el método asociado)
  + Mediante Jackson se hace la conversión automática de un bean a un json y viceversa.
  + Como pasar parámetros a las URI.

**Monitoreo de web services.**

* Para el monitoreo de servicios web se requiere de la siguiente configuración.
  + Agregar las siguientes dependencias:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.data</groupId>

<artifactId>spring-data-rest-hal-browser</artifactId>

</dependency>

Con la dependencia **actuator** se gestiona la parte del monitoreo de los servicios. (Funciona como un servidor en el que se exponen servicios en formato hal)

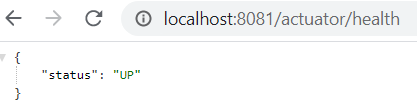
Con la dependencia **hal-browser** nos permite consultar los servicios que nos brinda el servidor **actuator**.

hal – Hypertext Application Language.

* + Para consumir los servicios ir a las siguientes URLs
    - <http://localhost:8081/actuator>



* + - <http://localhost:8081/actuator/health>



* + - Para poder consultar información de todos los endPoints se necesita hacer la siguiente configuración:

Agregar la siguiente línea en el archivo **application.properties**

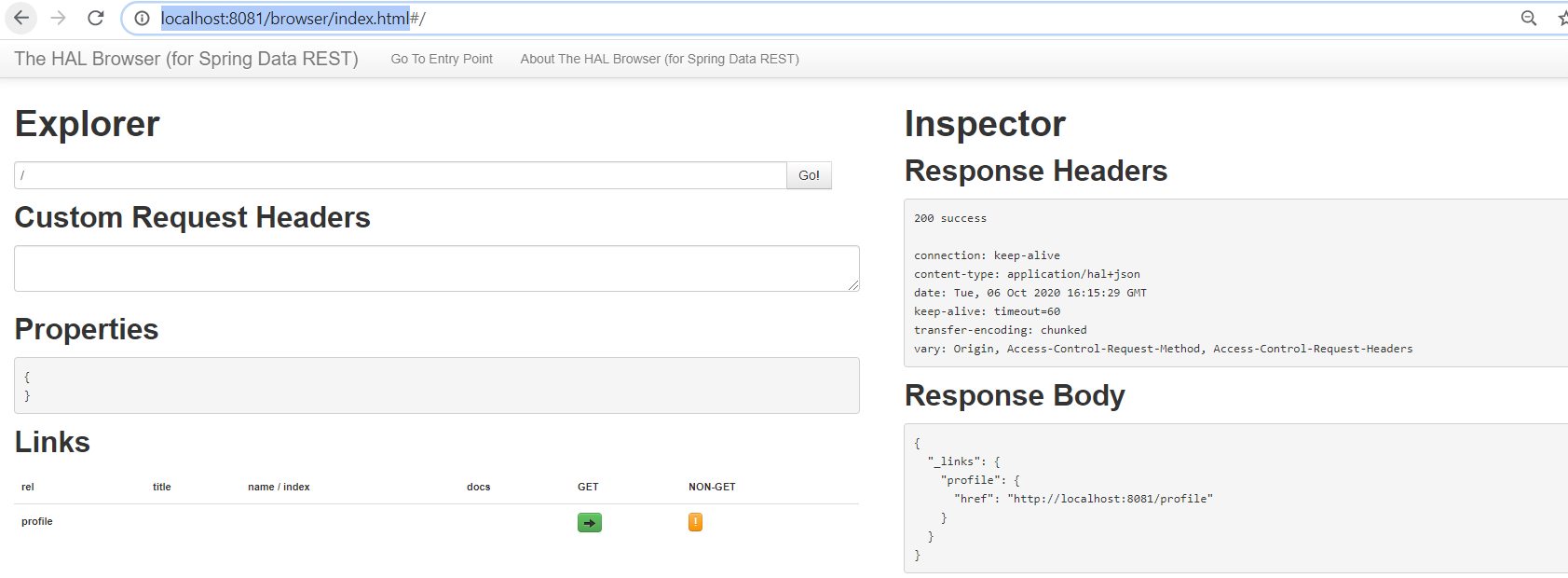
# Requerido en la parte del monitoreo de los web services, con la instrucción de abajo se indica que se deben contemplar todos los endPoints

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

NOTA:

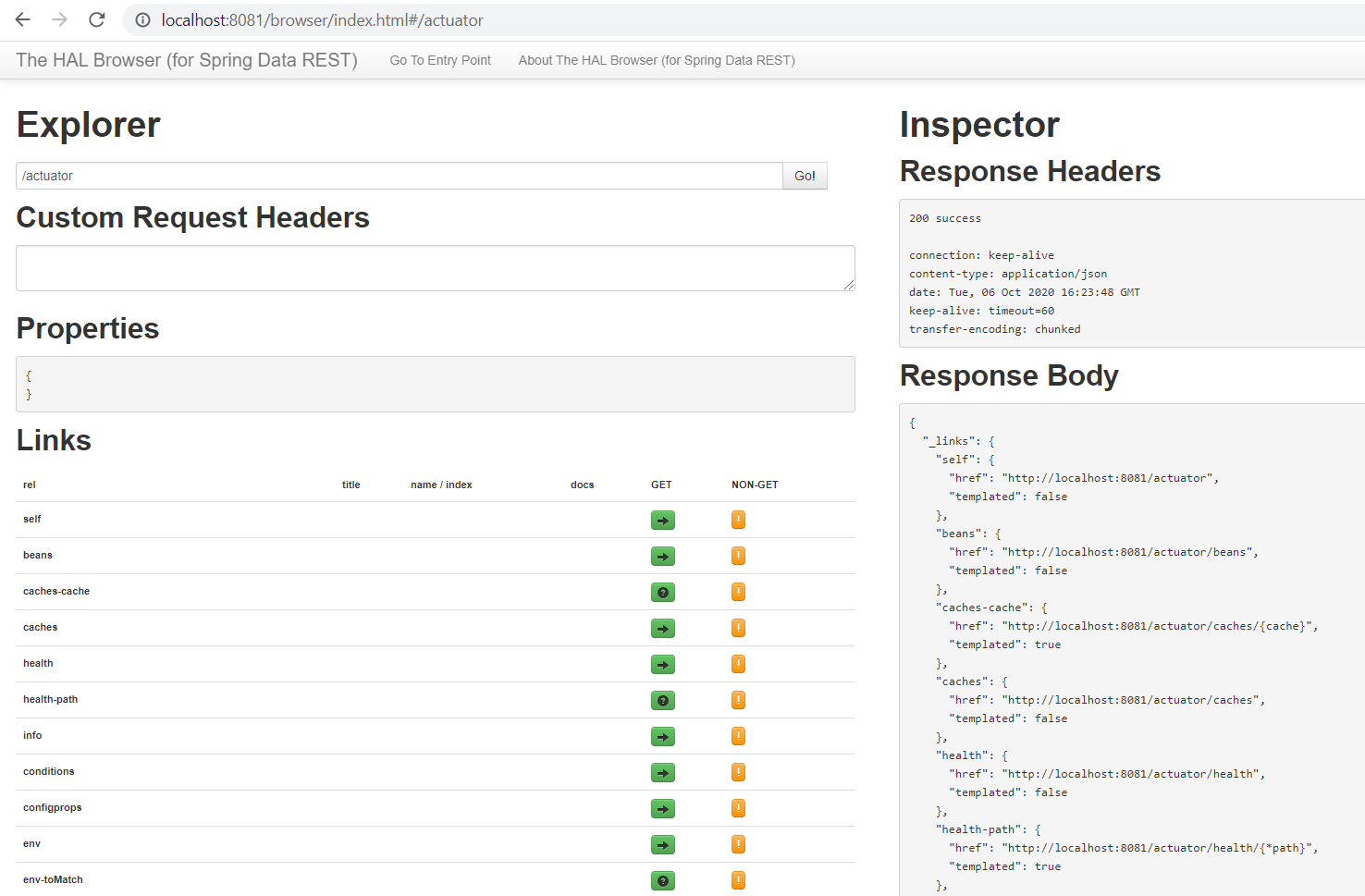
Tal vez sea necesario inicializar la aplicación para ver reflejados los cambios.

* + - Para consumir los servicios, ir a la siguiente URL: <http://localhost:8081/browser/index.html> (Dirección del navegador HAL)

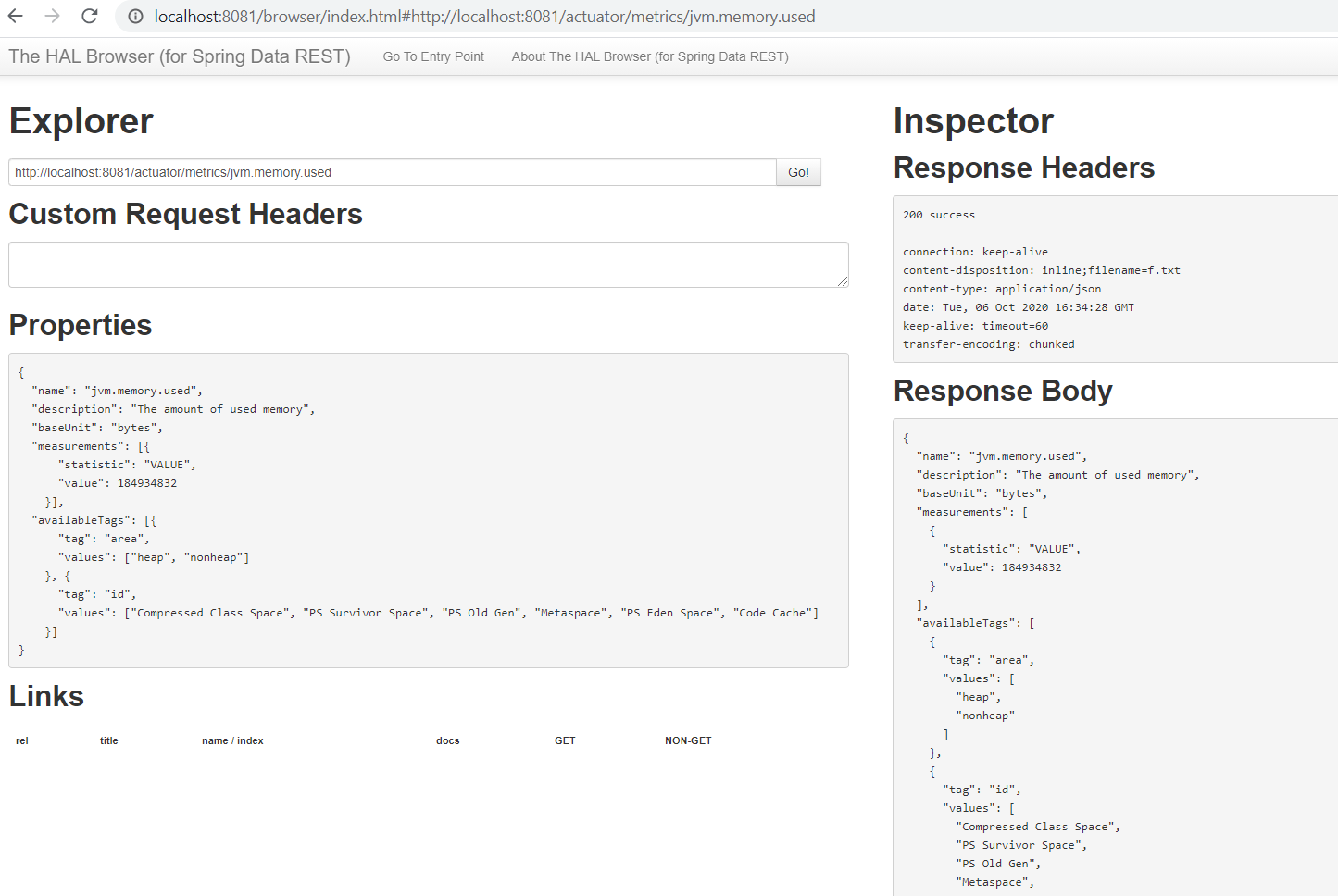


Consultemos algunos de los servicios (escribir en la sección Explorer **actuator** y dar clic en **Go!**):

Una vez que nos encontremos en la pantalla de abajo, podemos consultar los servicios de los **actuator** disponibles.



Por ejemplo, para consultar información de la memoria, seleccionar **metrics** y concatenar a la URL **jvm.memory.used.** (<http://localhost:8081/actuator/metrics/jvm.memory.used>)



**Filtrado ESTATICO de información que deben considerar los web services.**

* **Filtrado ESTATICO de información. (@JsonIgnore)**
* Ejemplo
  + Creación del bean **SomeBean.java**

**public** **class** SomeBean {

**private** String valor1;

**private** String valor2;

**private** String valor3;

**…**

}

* + Creación del controller **FilteringController.java**

@RestController

**public** **class** FilteringController {

@GetMapping("/filtering")

**public** SomeBean retrieveSomeBean() {

**return** **new** SomeBean("valor1", "valor2", "valor3");

}

}

* + El consumo del servicio nos proporcionaría la siguiente información:



* + **Hasta aquí, se obtiene la información de forma norma, para hacer el FILTRADO de la información, se deben poner las siguientes anotaciones en el Bean SomeBean.java**

Existen dos formas de implementar las anotaciones

1. Indicando el filtro a nivel atributo.

**public** **class** SomeBean {

**private** String valor1;

**private** String valor2;

@JsonIgnore

**private** String valor3;

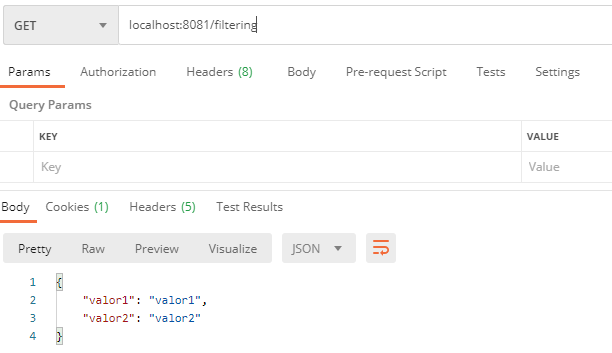
...

}

Con la anotación **@JsonIgnore** se indica que ese atributo de clase no se debe contemplar en los web services.

Es probable que para que pueda ver los resultados, tenga que reiniciar la aplicación.

Como se puede comprobar en la imagen de abajo, ya no se contempla el atributo con la anotación @JsonIgnore.



1. Indicando el filtro a nivel clase.

@JsonIgnoreProperties(value = {"valor2"})

**public** **class** SomeBean {

**private** String valor1;

**private** String valor2;

**private** String valor3;

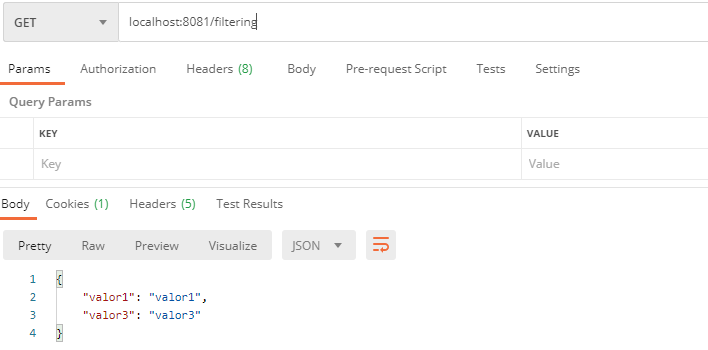
...

}

Con la anotación **@JsonIgnoreProperties** se indica los atributos de clase que se quieren filtrar.

Es probable que para que pueda ver los resultados, tenga que reiniciar la aplicación.

Como se puede comprobar en la imagen de abajo, ya no se contempla el atributo especificado en la anotación @JsonIgnoreProperties.



* **Filtrado dinámico de información. (MappingJacksonValue)**

Para el filtrado dinámico requerimos de lo siguiente:

* + Se necesita la definición de un servicio web como el siguiente:
    - Para el caso de obtener un solo resultado.

@GetMapping("/dinamic-filtering")

**public** MappingJacksonValue retriveSomeDinamicBean() {

SomeDinamicBean someBean = **new** SomeDinamicBean("valor1", "valor2", "valor3");

//Se definen los campos que no se quieren filtrar.

SimpleBeanPropertyFilter filter = SimpleBeanPropertyFilter.*filterOutAllExcept*("valor1", "valor2");

//Se hace el mapeo del filter con la clase del bean.

FilterProvider filters = **new** SimpleFilterProvider().addFilter("SomeDinamicBeanFiltering", filter);

//Asocia el filtro que se debe aplicar a la información.

MappingJacksonValue mapping = **new** MappingJacksonValue(someBean);

mapping.setFilters(filters);

**return** mapping;

}

* + - * Con el objeto **filter** se indica los valores que no deben ser filtrados, es decir que se deben mostrar en los resultados del servicio web.
      * Con el objeto **filters** se hace el mapeo del **entity/bean** al que se le debe aplicar el **filter**. (El nombre del primer parámetro debe ser igual al que se defina en el entity/bean – "SomeDinamicBeanFiltering")
      * Con el objeto **mapping** se asocia el **bean** que contiene la información con los filtros definidos.
    - Para el caso de obtener una colección de resultados.

@GetMapping("/dinamic-filtering-list")

**public** MappingJacksonValue retriveListOfSomeDinamicBean() {

List<SomeDinamicBean> lista = **new** ArrayList<SomeDinamicBean>();

lista.add(**new** SomeDinamicBean("valor1", "valor2", "valor3"));

lista.add(**new** SomeDinamicBean("valor11", "valor12", "valor13"));

//Se definen los campos que no se quieren filtrar.

SimpleBeanPropertyFilter filter = SimpleBeanPropertyFilter.*filterOutAllExcept*("valor2", "valor3");

//Se hace el mapeo del filter con el la clase del bean.

FilterProvider filters = **new** SimpleFilterProvider().addFilter("SomeDinamicBeanFiltering", filter);

//Asocia el filtro que se debe aplicar a la información. (Es decir, se hace el filtrado de la información)

MappingJacksonValue mapping = **new** MappingJacksonValue(lista);

mapping.setFilters(filters);

**return** mapping;

}

* + - * Con el objeto **filter** se indica los valores que no deben ser filtrados, es decir que se deben mostrar en los resultados del servicio web.
      * Con el objeto **filters** se hace el mapeo del **entity/bean** al que se le debe aplicar el **filter**. (El nombre del primer parámetro del método **.addFilter(…)** debe ser el mismo con el que se califique el entity/bean. En este caso y a manera de ejemplo, lo nombramos "SomeDinamicBeanFiltering")
      * Con el objeto **mapping** se asocia el **bean** que contiene la información con los filtros definidos.

* + En el **entity/bean** poner la siguiente anotación.

@JsonFilter("SomeDinamicBeanFiltering")

**public** **class** SomeDinamicBean {

**private** String valor1;

**private** String valor2;

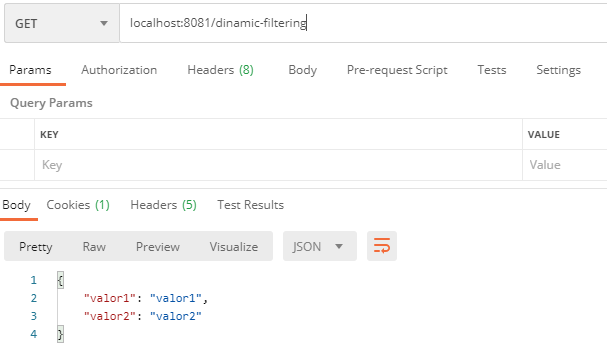
**private** String valor3;

**...**

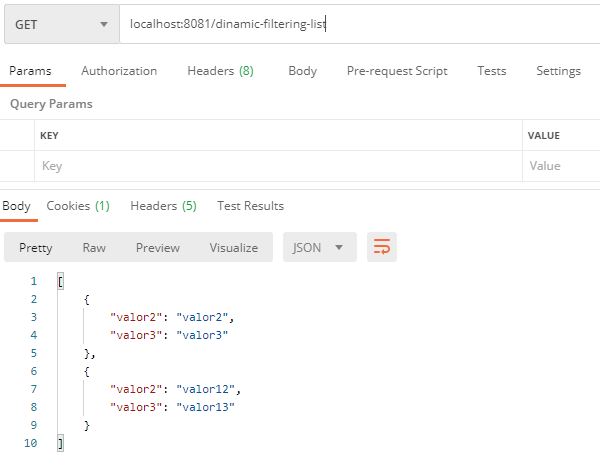
}

* + - Con la anotación **@JsonFilter** se indica el nombre del identificador del entity/bean para hacer el filtrado dinámico de información. Este nombre ("SomeDinamicBeanFiltering") debe ser el que se indique al momento de hacer la configuración del filtrado dinámico.
  + Al consumir el servicio, debe obtener algo como lo siguiente:

Consulta de un solo registro (Observe que solo se muestran los campos que se indicaron en el filtro):



Consulta de más de un registro (Observe que solo se muestran los campos que se indicaron en el filtro):



**Manejo de versiones. (Existen varias formas de controlar el manejo de versiones)**

* **Forma básica del manejo del control de versiones. (Conocida como la versión URI, ya que se genera una URI para cada una de las versiones)**

Para ello, imagine el siguiente escenario: En un inicio tiene la clase **Person**, pero con el tiempo de da cuenta de la necesidad de agregar más información a la clase, por lo que tiene que ir versionándola, así surge la necesidad de crear las clases **PersonV1**, **PersonV2**.

**public** **class** PersonV1 {

**private** String name;

**public** PersonV1() {

}

**public** PersonV1(String name) {

**super**();

**this**.name = name;

}

...

}

**public** **class** PersonV2 {

**private** Name name;

**public** PersonV2() {

}

**public** PersonV2(Name name) {

**super**();

**this**.name = name;

}

...

}

**public** **class** Name {

**private** String firstName;

**private** String lastName;

**public** Name() {}

**public** Name(String firstName, String lastName) {

**super**();

**this**.firstName = firstName;

**this**.lastName = lastName;

}

}

**Y también se genera el controller para poder consumir los servicios.**

@RestController

**public** **class** PersonVersioningController {

@GetMapping("/v1/person")

**public** PersonV1 personV1() {

**return** **new** PersonV1("Néstor Pérez");

}

@GetMapping("/v2/person")

**public** PersonV2 personV2() {

**return** **new** PersonV2(**new** Name("Néstor", "Pérez"));

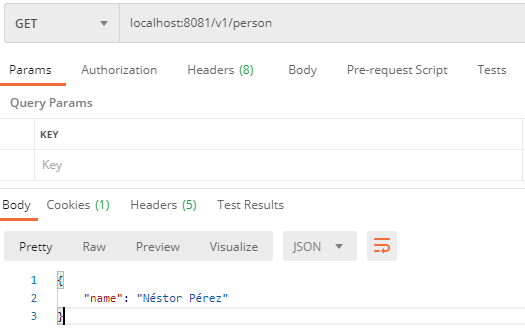
}

}

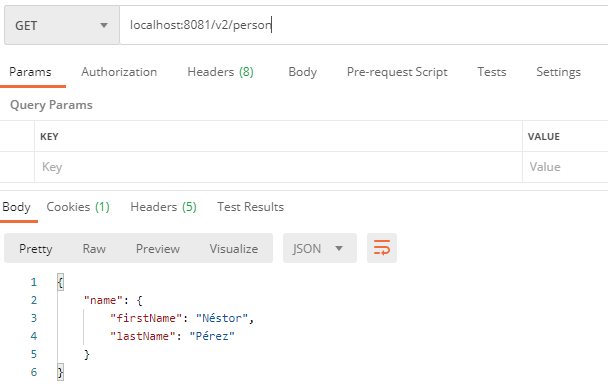
**El resultado de consumir los servicios, debe ser algo similar a lo siguiente:**

Para este ejemplo básico de control de versiones, se manda a llamar uno u otro método. (Así de simple, V1 o V2)

Para la invocación de la versión 1



Para la invocación de la versión 2



* **Otra formas del control de versiones.**

NOTA: Recuerde que para que visualice los servicios, tal vez sea necesario reiniciar la aplicación.

* + **Mediante el uso de parámetros en el request.**

Para ello, necesitamos de los siguientes servicios en el controller.

@GetMapping(value = "/person/param", params = "version=1")

**public** PersonV1 paramV1() {

**return** **new** PersonV1("Néstor Pérez");

}

@GetMapping(value = "/person/param", params = "version=2")

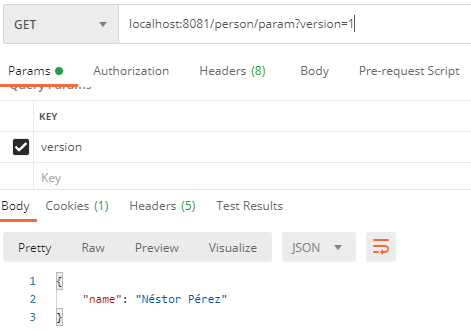
**public** PersonV2 paramV2() {

**return** **new** PersonV2(**new** Name("Néstor", "Pérez"));

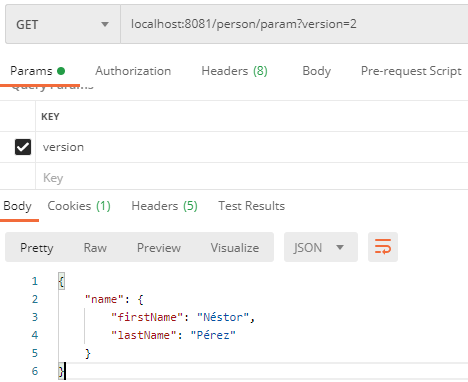
}

Para consumir los servicios, debemos agregar el parámetro al request.

Para la versión V1 (con el parámetro versión = 1)



Para la versión V2 (con el parámetro versión = 2)



* + **Mediante el uso de parámetros en el header.**

Para ellos, necesitamos de los siguientes servicios en el controller.

@GetMapping(value = "/person/header", headers = "X-API-VERSION=1")

**public** PersonV1 headerV1() {

**return** **new** PersonV1("Néstor Pérez");

}

@GetMapping(value = "/person/header", headers = "X-API-VERSION=2")

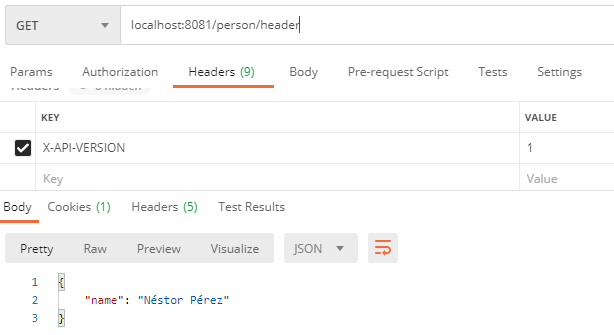
**public** PersonV2 headerV2() {

**return** **new** PersonV2(**new** Name("Néstor", "Pérez"));

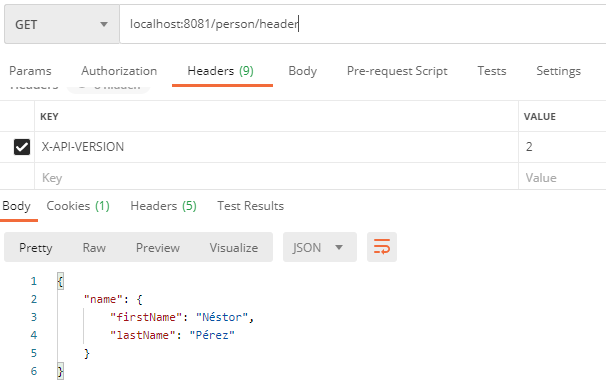
}

Para consumir los servicios, debemos agregar el parámetro al header.

Para la versión V1 (con el parámetro X-API-VERSION=1 en el header)



Para la versión V2 (con el parámetro X-API-VERSION=2 en el header)



NOTA: el nombre del parámetro en el header se puede llamar como mejor le convenga. Es este caso y a manera de ejemplo se decidió llamarle X-API-VERSION. Lo que si es importante, es que se llame igual en el header y en el servicio.

* + **Mediante el uso de parámetros en el header, pero ahora usando PRODUCES. (También conocido como MIME TYPE debido a que se indica el tipo de dato a usar)**

Para ello, necesitamos de los siguientes servicios en el controller.

/\*\*

\* En el parámetro produce se proporciona la siguiente información:

\* - Informaición relevante de la empresa.

\* - La versión que se esta generando.

\* - Tipo de dato a usar. (en este caso json)

\* **@return**

\*/

@GetMapping(value = "/person/produces", produces = "application/vnd.company.app-v1+json")

**public** PersonV1 producesV1() {

**return** **new** PersonV1("Néstor Pérez");

}

/\*\*

\* En el parámetro produce se proporciona la siguiente información:

\* - Informaición relevante de la empresa.

\* - La versión que se esta generando.

\* - Tipo de dato a usar. (en este caso json)

\* **@return**

\*/

@GetMapping(value = "/person/produces", produces = "application/vnd.company.app-v2+json")

**public** PersonV2 producesV2() {

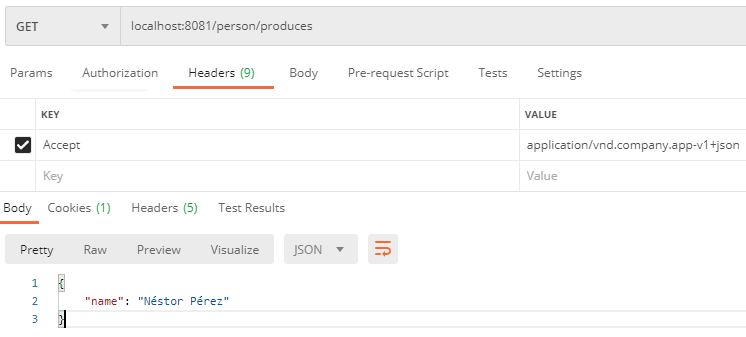
**return** **new** PersonV2(**new** Name("Néstor", "Pérez"));

}

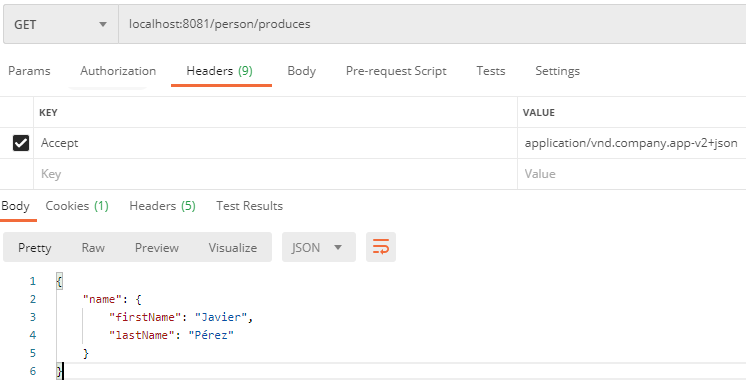
NOTA: En el parámetro produce se proporciona la siguiente información, “información relevante de la empresa”, “la versión que se está generando”, “el tipo de dato a usar”

Para consumir los servicios, debemos agregar el parámetro en el header.

Para el caso de la versión V1. (Con el parámetro Accept="application/vnd.company.app-v1+json" en el header)



Para el caso de la versión V2. (Con el parámetro Accept="application/vnd.company.app-v2+json" en el header)



**Seguridad.**

* **Implementación básica de seguridad. (Autenticación a la aplicación)**

Para ello necesitamos agregar al pom.xml la siguiente dependencia:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

Después de agregar la dependencia, reiniciar la aplicación, spring maneja por default los siguientes datos de autenticación:

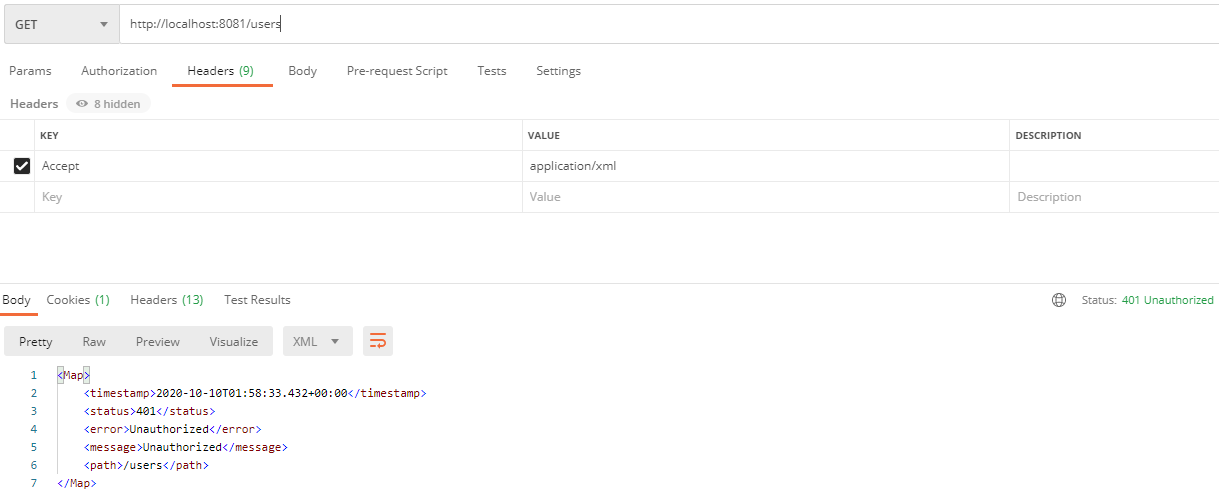
* + User: user
  + Password: En la consola se desplegara un nuevo password cada vez que se inicialice la aplicación. (copiar este password)



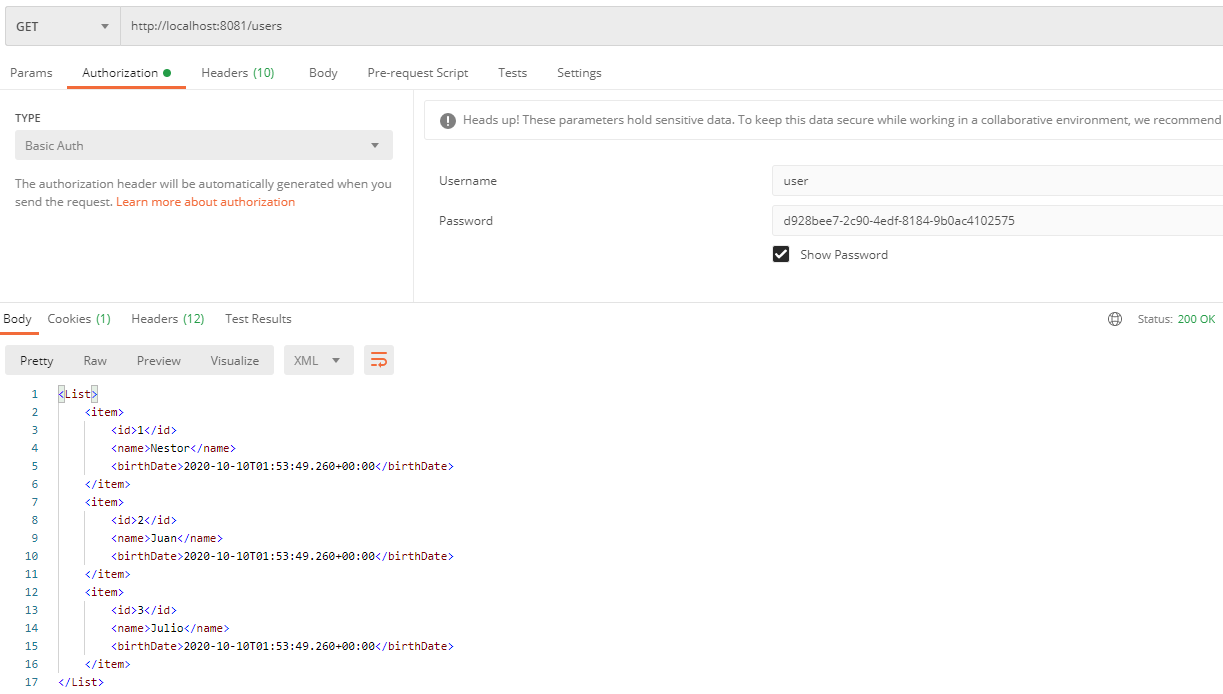
Para consumir los servicios, tenemos que autenticarnos con los datos antes mencionados:

* + User: user
  + Password: el que proporcione spring boot al momento de levantar la aplicación.

Si intentamos consumir algún servicio sin autenticarnos, se genera el siguiente error:



Autenticarnos con el user y password antes mencionados seleccionando la opción **Basic Auth**



Si queremos definir nuestro propio user y password, debemos agregar las siguientes líneas en el archivo **application.properties**

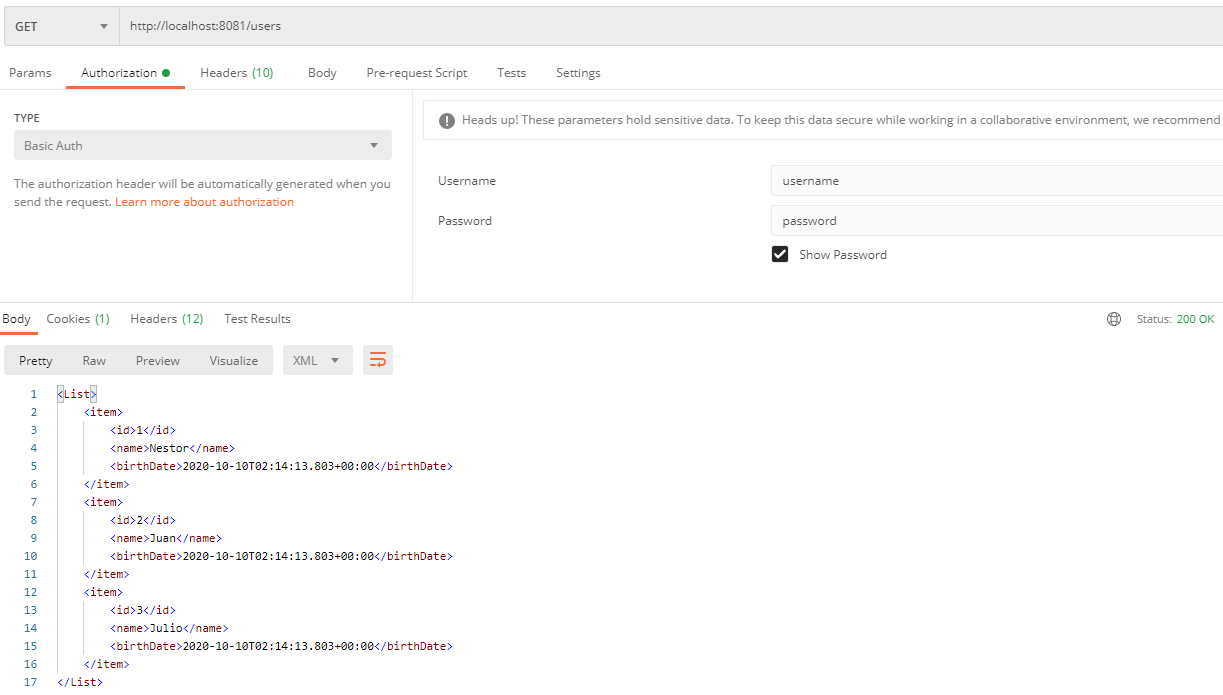
spring.security.user.name=username

spring.security.user.password=password

Para poder consumir los servicios, tenemos que autenticarnos con los datos indicados en el archivo **application.properties**

**User: username**

**Password: password**



**JPA (Implementando H2 como base de datos en memoria)**

* **Para el manejo de JPA, se requiere lo siguiente:**
  + Agregar al pom.xml las siguientes dependencias:
    - Dependencias requeridas:

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

* + - Configuración de la base de datos H2.

En el archivo **application.properties** agregar las siguientes líneas:

#Para la configuración de la base de datos H2

spring.datasource.url=jdbc:h2:mem:testdb

spring.data.jpa.repositories.bootstrap-mode=default

* + - Configuración de las clases java que queremos mapear con la base de datos. (entity)

Poner las siguientes anotaciones en el entity/bean. (Para temas de nuestro ejemplo tomaremos el bean User)

@Entity

**public** **class** User {

@Id

@GeneratedValue

**private** Integer id;

**private** String name;

**private** Date birthDate;

...

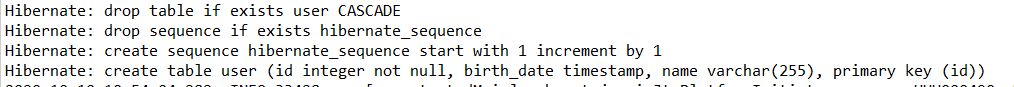
}

Con la anotación **@Entity** le indicamos a JPA que lo contemple como una entidad.

Con la anotación **@Id** le indicamos a JPA que el atributo marcado con esta etiqueta es el identificador único.

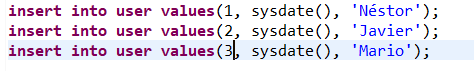
Con la anotación **@GeneratedValue** se le indica a JPA que el valor lo debe generar la base de datos.

Con estas modificaciones, la próxima vez que inicie la aplicación, la base de datos H2 generará la tabla User en memoria. (Consultar el log de la aplicación)



* Insertar información en la base de datos. (En este caso, agregaremos información a la tabla User que es con la que hemos estado trabajando la configuración de JPA)

Crear el archivo **data.sql** en el directorio **src/main/resources** de la aplicación con algo similar al siguiente contenido.



Para verificar que en efecto, los registros fueron persistidos, ingrese a la consola de H2 y ejecute los query requeridos.

* Habilitar los logs de SQL, H2.

En el archivo **application.properties** agregar las siguientes líneas:

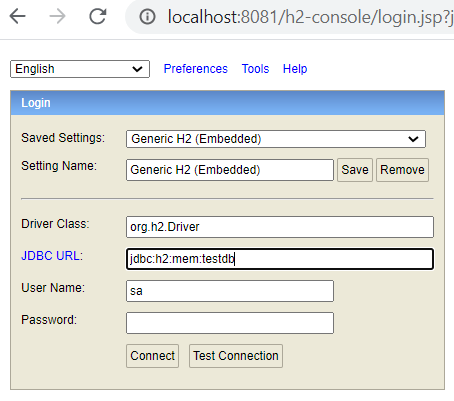
spring.jpa.show-sql=true

spring.h2.console.enabled=true

* Ingresar a la consola de H2.

Si tiene habilitada la seguridad de spring boot, antes de poder ingresar a la consola de H2 se debe autenticar.

Ingresar a la URL <http://localhost:8081/h2-console>

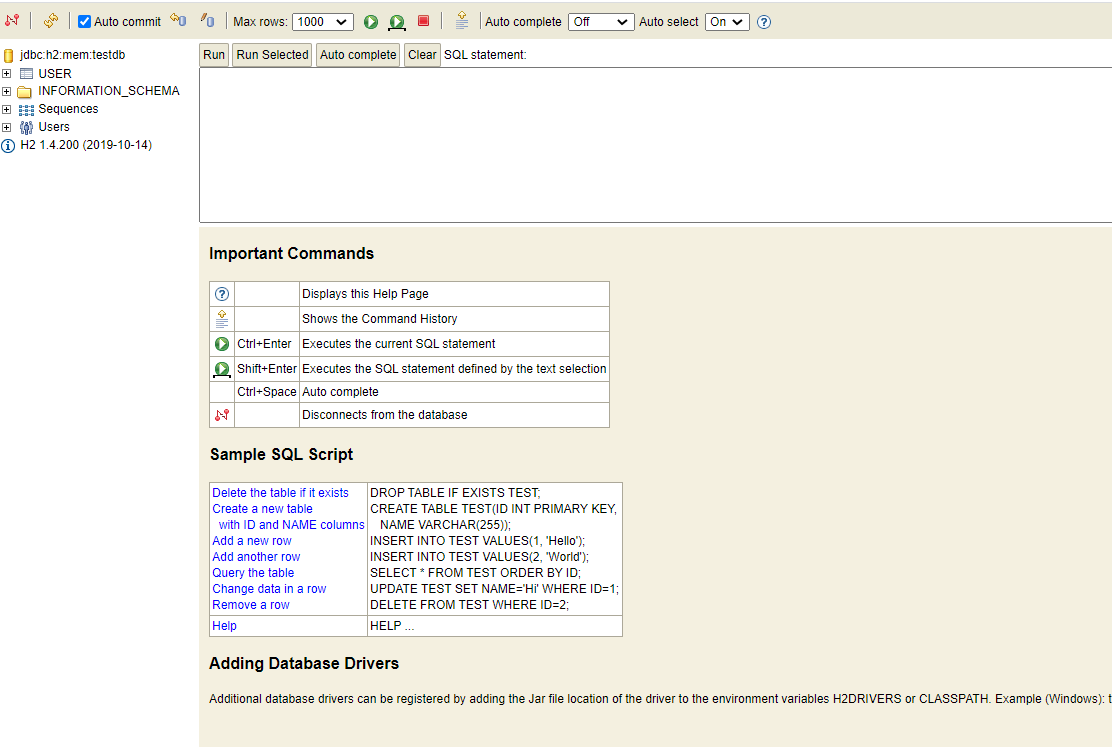


NOTAS:

Asegúrese de que en el campo **JDBC URL** se tenga el mismo valor que dio de alta en el archivo **application.properties. (**jdbc:h2:mem:testdb**)**

En el campo **Saved Setting** tenga seleccionada la opción **Generic H2 (Embedded)**

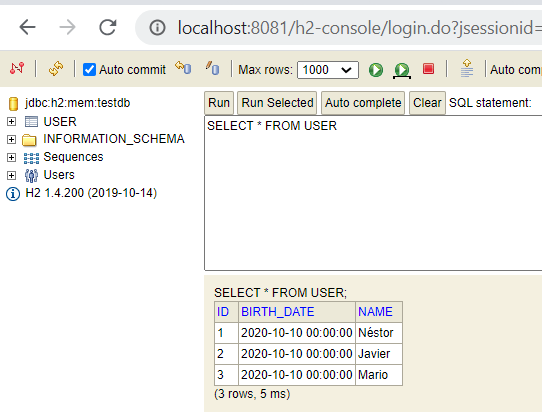
Si todo se realizó de forme exitosa, debe poder ver una pantalla como la siguiente:



Para consultar la información de **User** persistida en pasos anteriores, ejecute el siguiente query:

NOTA: Recuerde que tiene que reiniciar la aplicación para poder ver tomar la información del archivo **data.sql** generado en los pasos anteriores.

SELECT \* FROM USER



* **Creación y consumo de servicios con JPA usando una base de datos H2. (Recordemos que la base de datos H2 es a nivel memoria)**
  + Crear la clase del controller con cada uno de los servicios que se requieran.

@RestController

**public** **class** UserJpaController {

@Autowired

**private** UserDaoService userDaoService;

@GetMapping("/jpa/users")

**public** List<User> retrieveUsers(){

List<User> users = userDaoService.findAll();

**return** users;

}

...

}

* + Crear la interface del Repository con la cual se interactúa con la base de datos.

@Repository

**public** **interface** UserRepository **extends** JpaRepository<User, Integer> {

}

Con la anotación **@Repository** para indicar que es un repositorio de JPA con la cual se debe interactuar con la base de datos.

La interface extiende de JpaRepository<User, Integer> y se indica la entidad que se necesita administrar, en este caso User, además se indica el tipo de dato del Id de la clase User.

* + Creación de la clase controller con cada uno de los servicios que se requieran.

@RestController

**public** **class** UserJpaController {

@Autowired

**private** UserRepository userRepository;

@GetMapping("/jpa/users")

**public** List<User> retrieveUsers(){

List<User> users = userRepository.findAll();

**return** users;

}

@GetMapping("/jpa/users/{id}")

**public** User retrieveUser(@PathVariable **int** id) {

Optional<User> user = userRepository.findById(id);

**if**( !user.isPresent() )

**throw** **new** UserNotFoundException("User not found id-"+id);

**return** user.get();

}

...

}

NOTAS:

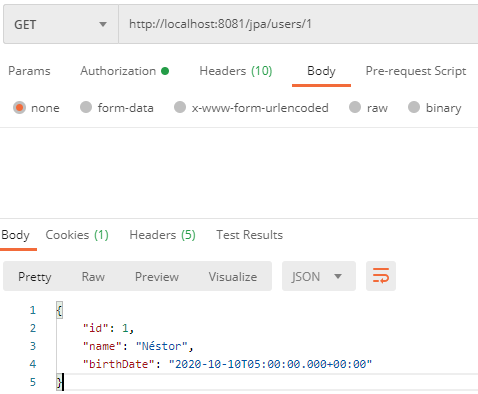
La interacción con la base de datos se hace mediante el objeto userRepository.

En el caso del servicio web para consultar un solo user, se maneja el tipo de dato **Optional<User>**.

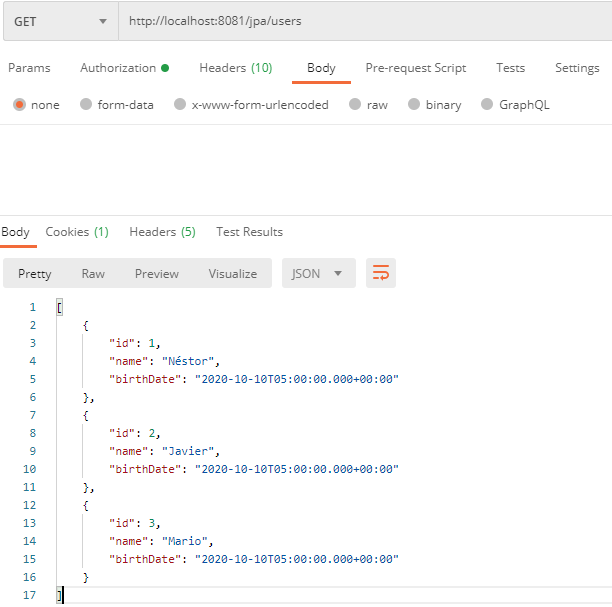
Con la instrucción **user.isPresent()** se valida si la consulta obtuvo algún resultado.

* + Al consumir los servicios debemos obtener algo similar a lo siguiente.

Para el servicio web para obtener un solo user

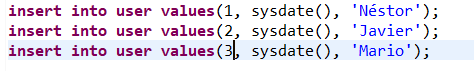


Para el servicio web para obtener todos los user

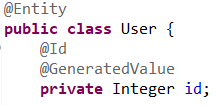


Para el caso del servicio web para crear un user.

Para este caso tener cuidado con el siguiente detalle, en el archivo **data.sql** se insertaron los regitros



Por otro lado en el entity User se indica que a base de datos se encargue de generar los valores para los Id’s:



Y por otro lado, cuando se levanta la aplicación, nos indica hibernate que la secuencia se inicializa en 1:



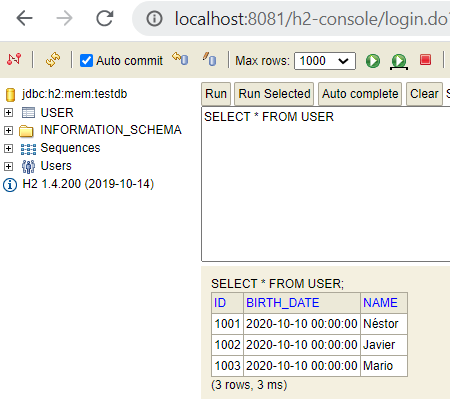
ES DECIR, CUANDO DE INTENTE GENERAR UN NUEVO REGISTRO EN LA BASE DE DATOS, HIBERNATE ASIGNA EL ID=1, esto genera el error



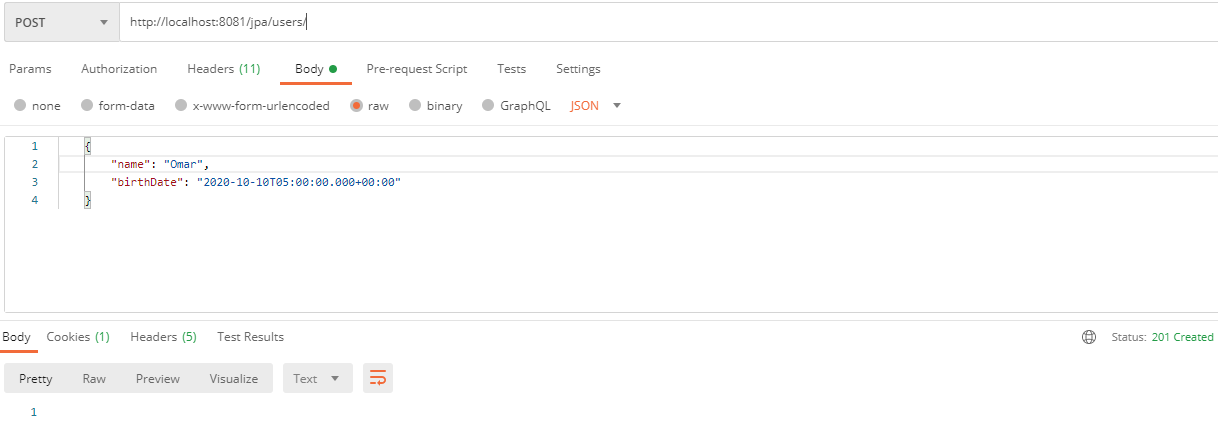
Una solución es modificar la numeración de los Id del archivo **data.sql**



Para consultar que las modificaciones ya fueron tomadas en cuenta, consulte en la consola de H2

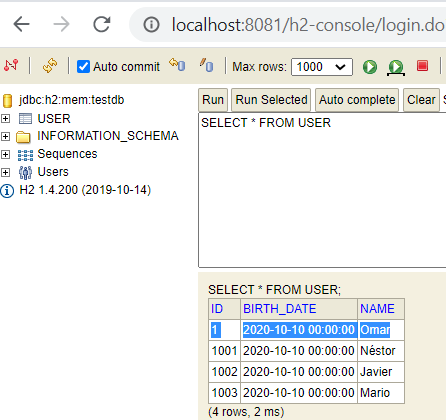


Con las modificaciones realizadas ahora si podremos crear el nuevo user. (Recuerde reiniciar la aplicación para que se contemplen los cambios)

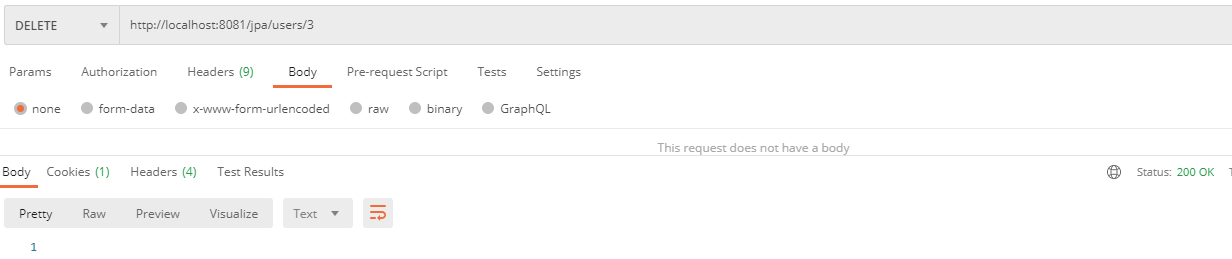


Listo, el registro ya fue persistido en la base de datos H2. (Recuerde que al ser una base de datos de memoria, si reinicia la aplicación únicamente se conservan los registros del archivo **data.sql**)

Como se puede apreciar en la imagen de abajo, el user Omar fue persistido en la base de datos.



Para el caso del servicio web para eliminar un user.



* **Relaciones OneToMany, ManyToOne mediante JPA.**

Para ello requeriremos de lo siguiente:

* + Creación de un nuevo entity, para este caso y a manera de ejemplo, lo llamaremos Post.java (Se genera en el mismo package donde se encuentra la clase User.java)

@Entity

**public** **class** Post {

@Id

@GeneratedValue

**private** Integer id;

**private** String descripcion;

/\*\*

\* Con esta anotación se indica que la relación entre user y post es la siguiente:

\* Muchos Post están relacionados con un User

\*

\* Esto implica que en el entity User se tenga que poner la anotación OneToMeny

\* Un User esta relacionado con muchos Post. (List<Post> posts)

\* \*/

@ManyToOne(fetch = FetchType.***LAZY***)

**private** User user;

…

}

NOTA: Si dejamos el atributo **user** así, se presenta un problema al momento de consumir la información. Este detalle se tratara más adelante.

La anotación @ManyToOne en el atributo **user** nos obliga a realizar la siguiente modificación en el entity User.java:

Agregar el atributo de clase List<Post> posts con sus respectivos métodos get/set

/\*\*

\* Con esta anotación se indica que un User está relacionado con muchos Post

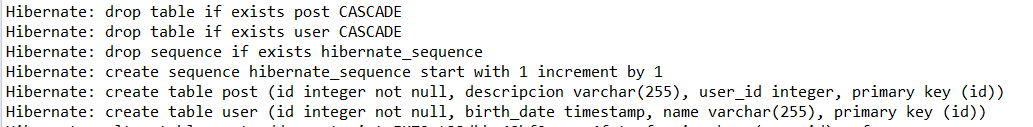
\* y que la columna por la que tiene que hacer el mapeo es "user" misma que se encuentra en el entity Post.java

\*/

@OneToMany(mappedBy = "user")

**private** List<Post> posts;

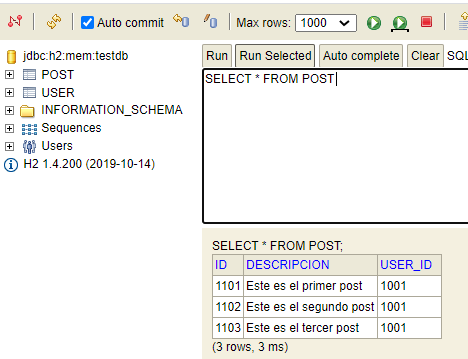
* + Reiniciar la aplicación para que se tomen en cuenta los cambios realizados. En la consola debe aparecer la creación del entity Post.



* + Puede verificar en la consola de H2 la creación de la tabla Post. Para agregar registros en la tabla post, agregue los registro en el archivo **data.sql**



Verifique en la consola de H2 la información de la tabla post.



* **Creando y consumiendo servicios web de tipo GET. (/jpa/user/{id}/post)**

Para ello, necesitamos lo siguiente:

* + En el controller del User, agregar los siguientes servicios:

Con el siguiente servicio se obtienen todos los post de un user. (Recuerde que para ver los cambios tal vez sea necesario que reinicie la aplicación)

@GetMapping("/jpa/users/{id}/post")

**public** List<Post> retrieveAllPostByUser(@PathVariable **int** id) {

Optional<User> user = userRepository.findById(id);

**if**( !user.isPresent() )

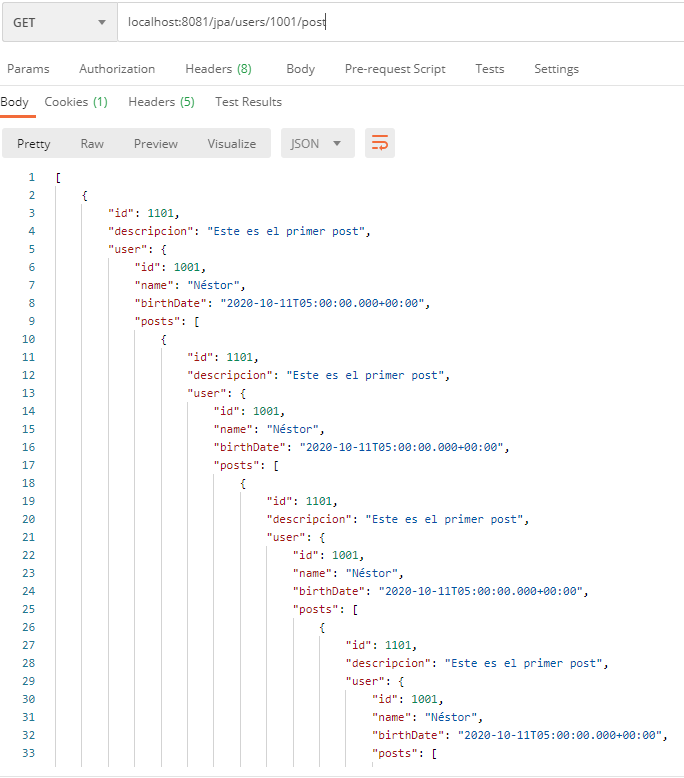
**throw** **new** UserNotFoundException("User not found id-"+id);

**return** user.get().getPosts();

}

* + Al consumir el servicio se presente el siguiente problema:

La consulta se vuelve cíclica ya que en la clase **User** se tiene una lista de Post, pero en la clase **Post** se tiene un objeto de tipo User.



* + Solucionar el problema anterior.

Para solucionar este problema, debemos agregar la siguiente anotación al atributo **user** de la clase **Post**

@Entity

**public** **class** Post {

@Id

@GeneratedValue

**private** Integer id;

**private** String descripcion;

/\*\*

\* Con esta anotación se indica que la relación entre user y post es la siguiente:

\* Muchos Post están relacionados con un User

\*

\* Esto implica que en el entity User se tenga que poner la anotación OneToMeny

\* Un User esta relacionado con muchos Post. (List<Post> posts)

\*

\* Con la anotación @JsonIgnore se indica que no se debe contemplar como parte de la respuesta.

\* \*/

@ManyToOne(fetch = FetchType.***LAZY***)

@JsonIgnore

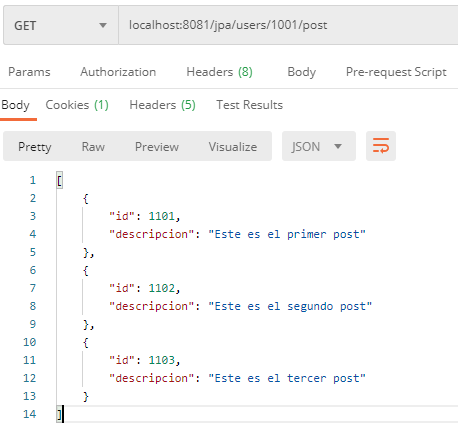
**private** User user;

**...**

}

Con la anotación **@JsonIgnore** se indica que este atributo no se debe considerar como parte de la respuesta y de esta forma se evita que se cicle la información cuando se consuma el servicio.

La respuesta del servicio ahora debe verse de la siguiente manera:



* **Creando y consumiendo servicios web del tipo POST. (/jpa/user/{id}/post)**

Para ello, necesitamos de lo siguiente:

* + Crear la interface del repository para la los objetos de la clase Post.

@Repository

**public** **interface** PostRepository **extends** JpaRepository<Post, Integer> {

}

* + Agregar los siguientes servicios en el controller del User.
    - Inyectar el repository para los objetos post.

@Autowired

**private** PostRepository postRepository;

* + - Agregar el servicio de tipo Post para crear registros en la tabla post para un determinado User.

@PostMapping("/jpa/users/{id}/post")

**public** ResponseEntity<?> createPostByUser(@PathVariable **int** id, @Valid @RequestBody Post post) {

Optional<User> userFind = userRepository.findById(id);

**if**( !userFind.isPresent() )

**throw** **new** UserNotFoundException("User not found id-"+id);

User user = userFind.get();

post.setUser(user);

Post savePost = postRepository.save(post);

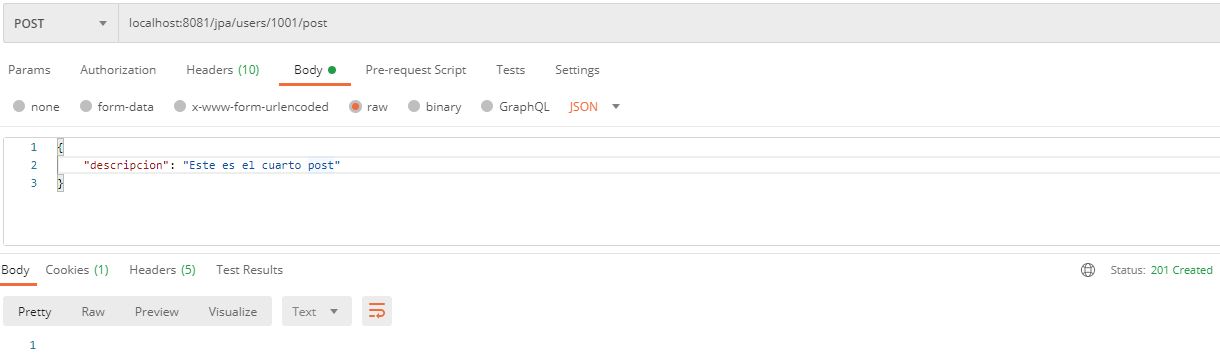
//Obtiene la URI del user persistido.

URI location = ServletUriComponentsBuilder.*fromCurrentRequest*().path("/{id}").buildAndExpand(savePost.getId()).toUri();

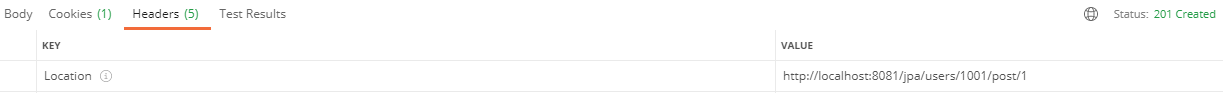
**return** ResponseEntity.*created*(location).build();

}

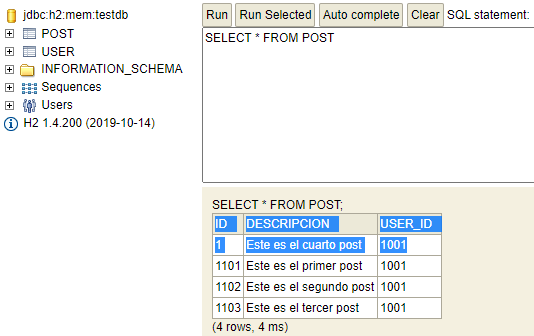
* + Al consumir el servicio se debe obtener algo similar a lo siguiente:



En los header también se indica la url donde se puede consultar el nuevo post.



Consultar el nuevo post en la consola de H2



* **Mejores prácticas de los servicios RestFul**
  + Richardson Maturity Model.

Considera 3 niveles dentro de los servicios RestFul:

* + - Nivel 0

Se exponen las acciones no los recursos. (Se exponen las URL de los servicios web)

Se exponen servicios web SOAP al estilo Rest. (En la cual se contemplan a los usuarios, cuentas, etc…)

* + - Nivel 1

Expone los recursos con una URI adecuada.

Se inicia con la migración de los servicios web SOAP a RestFul. (Es decir, ya se habla del manejo de recursos)

* + - Nivel 2

Es como el nivel 1 más el uso adecuado de métodos HTTP.

* + - Nivel 3

Es como el nivel 2 más el uso de HATEOAS como el motor del estado de la aplicación.

En este nivel no solo nos preocupamos por devolver los resultados de las consultas, sino que además se proporciona información de otras posibles acciones a realizar.

* + El diseño y nombrado de los servicios web siempre deben definir contemplando a los clientes.
  + Documentación de los servicios web
    - Mediante el uso de Swagger.
  + Los servicios web siempre deben aprovechar al máximo lo que HTTP proporciona, por ejemplo:
    - Nuestros servicios deben estar basados en los métodos:
      * Get
      * Post
      * Delete
      * Update
      * etc…
    - Nuestros servicios web siempre deben responder con un estatus adecuado:
      * 200 – Succes
      * 404 – Resource not found
      * 400 – Bad request
      * 201 – Created
      * 401 – Unauthorized
      * 500 – Server error
    - No proporcionar información clasificada como confidencial en las URI de los servicios web. (Como por ejemplo, SSN, )
    - En el nombrado de los servicios siempre usar plurales: **/jpa/users**
    - Cuando defina el nombre de los servicios web, piense en sustantivos.

**Micro servicios.**

* **Definición.**

Definición corta:

Pequeño servicio autónomo que funciona en conjunto.

Definición larga:

Es como desarrollar una sola aplicación como un conjunto de pequeños servicios, cada uno de los cuales ejecuta su propio proceso de comunicación con mecanismos muy livianos. Apoyándose en las bondades de HTTP basada en las capacidades empresariales (implementación automatizada y una gestión centralizada para los micro servicios)

Los micro servicios pueden incluso der desarrollados con diferentes lenguajes de programación. Y tal vez para ideas de almacenamiento de datos.

* **Características importantes de los micro servicios.**
  + Son servicios Rest
  + Son pequeñas unidades desplegables.
  + Se pueden habilitar en la nube.
  + Imagine una **aplicación monolito** que la divide en micro servicios con limites bien definidos que interactúan entre si y están habilitados para la nube.
* **Desafíos a considerar en el desarrollo de micro servicios.**
  + Contexto acotado.

¿Cómo identificar el límite de cada uno de estos micro servicios?

¿Cómo identificar que hacer en cada uno de estos micro servicios?

¿Cómo decide que debe hacer y que no debe hacer cada uno de los micro servicios?

A través del tiempo se van haciendo ajustes a los micro servicios hasta encontrar el contexto óptimo.

* + Gestión de la configuración.

Para administrar las n configuraciones que se requieran.

* + Escalamiento dinámico. (hacia arriba y hacia abajo)

Lo recomendable es hacer una **escala dinámica hacia arriba y hacia abajo** para establecer la tecnología requerida. (Pero siempre manejando un esquema dinámico, ya que no siempre se requieren las mismas cargas de trabajo)

Se recomienda un balanceo de carga

Para evitar que los servidores tengan sobre carga de trabajo o que se queden desaprovechados.

Para la asignación o des asignación de servicios de forma dinámica.

* + Visibilidad.

Monitoreo del estatus de los servidores y de los micro servicios. (Quien tiene recursos disponibles, configuraciones, etc…)

* + Tolerancia a fallas.

Definición de una arquitectura adecuada para los micro servicios.

Entre otras cosas se define la comunicación entre los micro servicios.

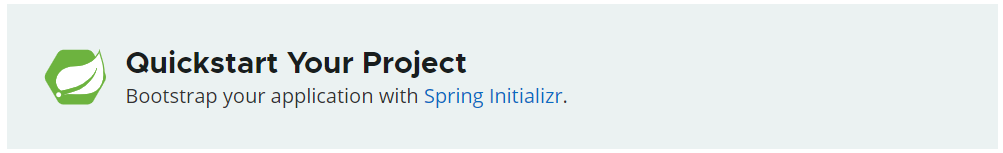
Consideración de tolerancia a fallas. (Para garantizar que no se caiga todo el sistema cuando se presente algún problema)

NOTA: los micro servicios pueden estar escritos en diferentes lenguajes.

* Página para construir cascarones de proyectos:

<https://spring.io/projects/spring-cloud>

Ir al apartado:



Notas:

Netflix

Son de las primeras organizaciones en trabajar con la arquitectura de los micro servicios.

Eureka Hystrix

Es otra de las formas de trabajar con arquitectura de los micro servicios.

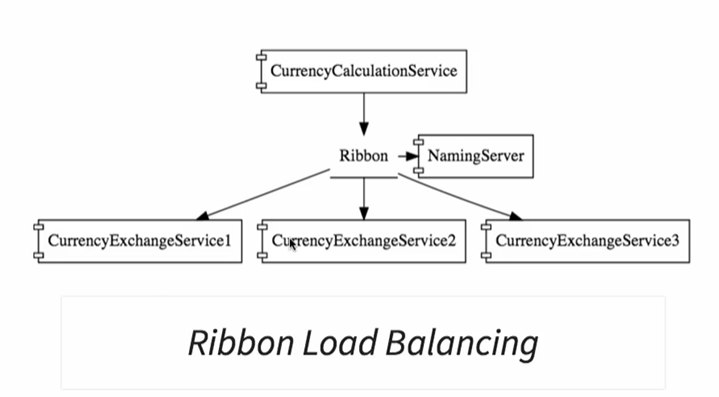
Finchley M2

Librería para interactuar con la nube de spring.

SpringCloudConfigServer

Para centralizar la administración de la configuración de todos los micro servicios.

Se va a trabajar con la siguiente arquitectura de micro servicios.



Como puede notar se tienen varias instancias del micro servicio **CurrencyExchangeService** con el cual se va a manejar la parte del escalamiento y el balanceo de carga.

El micro servicio **CurrencyCalculationService** interactúa con la n instancias del micro servicio **CurrencyExchangeService**

**Para lo cual se van a utilizar las siguientes herramientas:**

* + **Escalamiento dinámico:**
    - Eureka

Naming server. (Servidor de nombres – Para el registro de nombres de los micro servicios)

* + - Ribbon

Para el manejo del escalamiento hacia arriba y hacia abajo. (Asignar o desasignar recursos)

Para el manejo del balanceo de carga de lado del servidor

* + - Feing

Facilitador de servicios Rest para el cliente.

Para escribir clientes RestFul simples. (Facilita el manejo de los servicios RestFul)

* + **Visibilidad y monitoreo**
    - Zipking

Para el monitoreo de los micro servicios.

Spring Cloud Sloat para el manejo de asignación de Id´s a los procesos para facilitar su monitoreo mediante el Id asignado.

* + - Netflix API Gateway

Para el manejo de temas de seguridad entre otras muchas cosas.

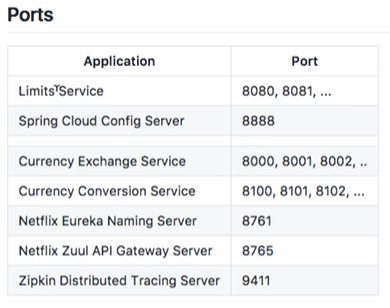
API Netflix Zuel para el manejo de la puerta de enlace. (Interceptor de solicitudes)

* + **Tolerancia a fallas**
    - Hystrix para el manejo de la tolerancia a fallas.

**Características de los micro servicios.**

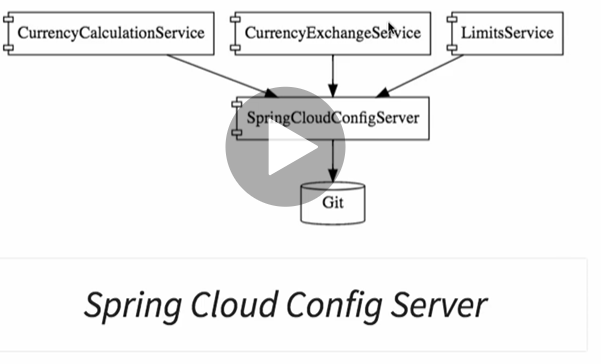
* + Los micro servicios pueden estar escritos en diferentes lenguajes.
  + El escalamiento de los micro servicios es dinámico.
  + Los ciclos de lanzamiento son mucho más rápido.

**Para el ejemplo que desarrollaremos, se utilizaran los siguientes puertos:**





Para el servidor de configuraciones, usaremos el siguiente esquema. (Usaremos un repositorio de git, para el almacenamiento de las configuraciones)



* **Creación de micro servicio básico.** (El micro servicio lo llamamos **limits-service**)

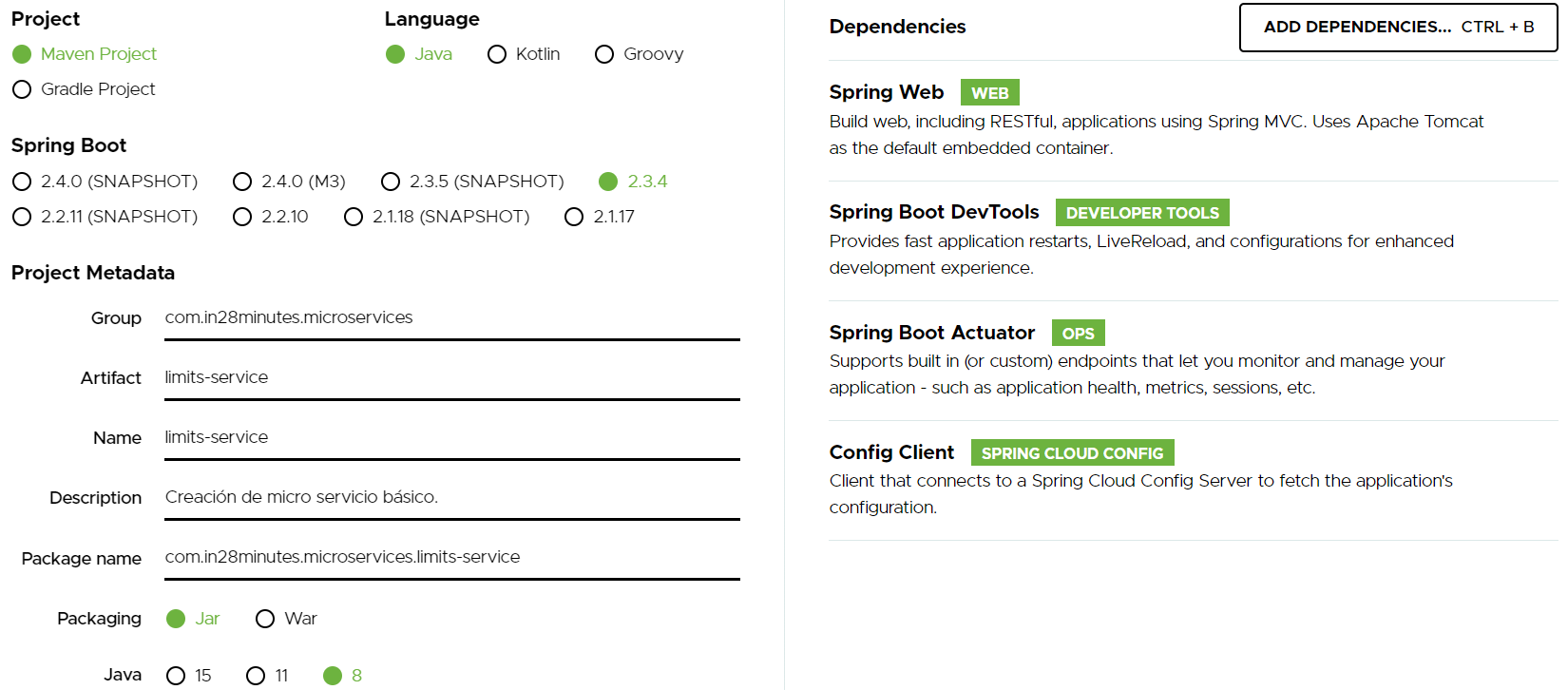
Con el servicio **limit-service** se obtiene un par de valores (mínimo, máximo), estos valores los obtendremos del application.properties

* + Generar el proyecto

Ir al sitio <https://start.spring.io/>

Con la siguiente configuración:

Importante respetar el nombre del group-id y el artifact, así como las dependencias seleccionadas.



**Spring Boot Actuator** es el servidor que usaremos.

* + Importar el proyecto a un IDE de java. (Eclipse, etc…)

Si al levantar el proyecto le manda un error por el puerto 8080, cambiar el puerto al 8081 en el archivo application.properties

#Define el puerto que usa nuestra app

server.port=8081

Con los ajustes realizados, el micro servicio **limits-service** debería levantar sin ningún problema.

* + Agregar configuraciones y funcionalidades al micro servicio **limits-service**
    - Dar de alta el nombre de la aplicación.

Para ello, agregar las siguientes líneas en el archivo **application.properties**

spring.application.name=limits-service

* + - Crear el controller siguiente.

**import** com.in28minutes.microservices.limitsservice.bean.LimitConfiguration;

@RestController

**public** **class** LimitsConfigurationController {

@GetMapping("/limits")

**public** LimitConfiguration retrieveLimitsFromConfigurations() {

**return** **new** LimitConfiguration(1000, 1);

}

}

**Crear el bean**

**package** com.in28minutes.microservices.limitsservice.bean;

**public** **class** LimitConfiguration {

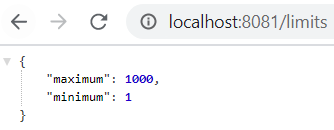
**private** **int** maximum;

**private** **int** minimum;

**...**

}

* + - Consumir el nuevo servicio



* + Leer valores (maximum, minimum) desde el archivo **application.properties**
    - En el archivo **application.properties** dar de alta las siguientes constantes. (NOTE QUE SE PONE EL NOMBRE DE LA APLICACIÓN MÁS EL NOMBRE DE LA CONSTANTE – limits-service.<constantName>)

limits-service.minimum=99

limits-service.maximun=9999

* + - Mapear las constantes del archivo **application.properties** a una clase **java**. (generar propiedades de configuración)

**package** com.in28minutes.microservices.limitsservice;

**import** org.springframework.boot.context.properties.ConfigurationProperties;

**import** org.springframework.stereotype.Component;

@Component

@ConfigurationProperties("limits-service")

**public** **class** Configuration {

**private** **int** minimum;

**private** **int** maximum;

//Con sus respectivos getter/setter

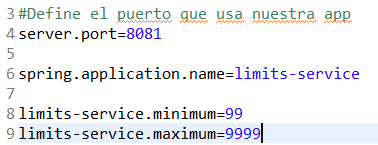
}

Con la anotación **@Component** se le indica a spring que debe contemplar la clase como un bean.

Con la anotación **@ConfigurationProperties** indicamos el prefijo de las constantes que queremos leer del archivo **application.properties**

**Nota:** Las variables de la clase java se deben llamar igual que como fueron nombradas en el archivo **application.properties**

Por consiguiente, de la siguiente imagen, los únicos valores que se leerán, son: minimum, maximum.



* + - Inyectar en el controller **LimitsConfigurationController.java** una instancia de la clase **Configuration.java**

@RestController

**public** **class** LimitsConfigurationController {

@Autowired

**private** Configuration configuration;

@GetMapping("/limits")

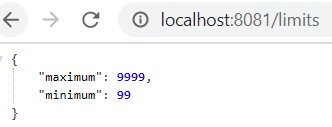
**public** LimitConfiguration retrieveLimitsFromConfigurations() {

**return** **new** LimitConfiguration(configuration.getMaximum(), configuration.getMinimum());

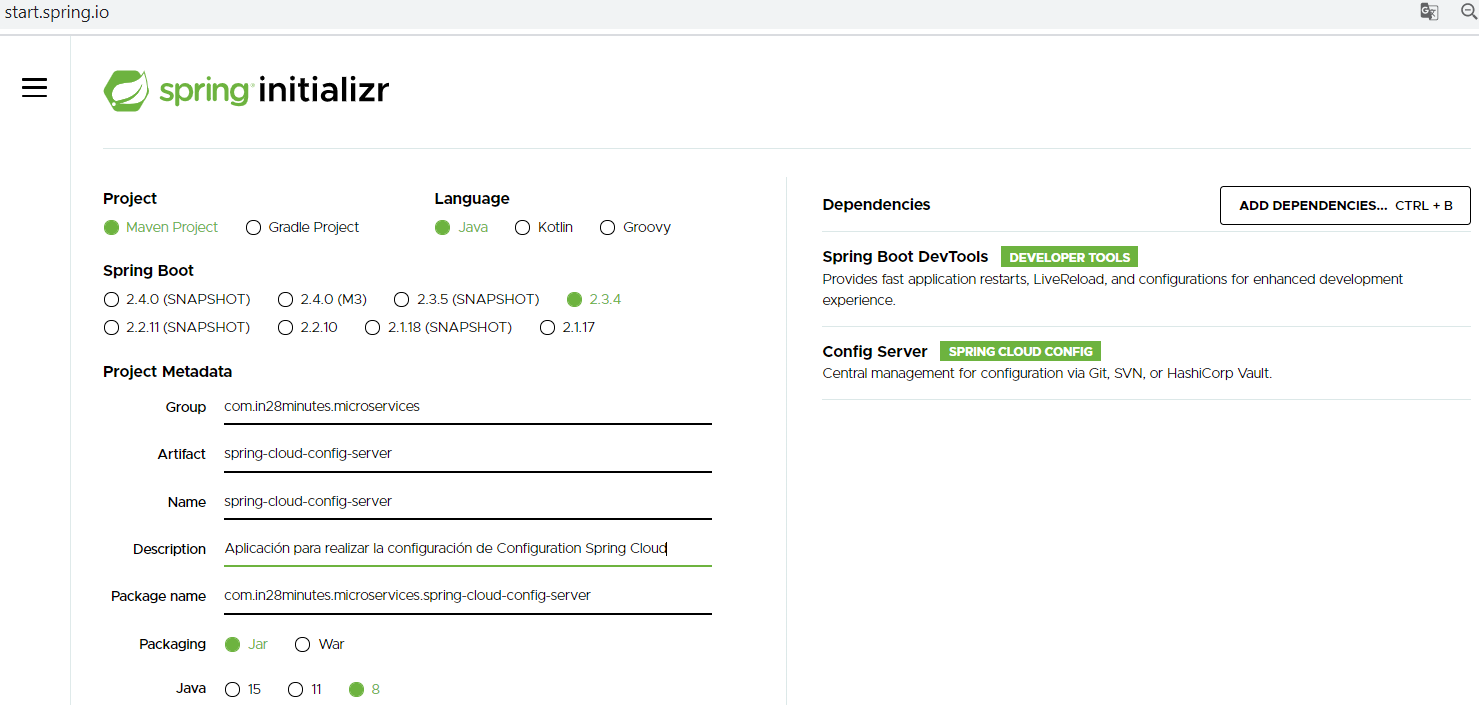
}

}

* + - Listo, al consumir el servicio, debemos obtener algo similar a lo siguiente:



* **Creación de micro servicio con Configuration Spring Cloud.**
  + Crear un proyecto con las siguientes propiedades.



**Note que el nombre del Group es el mismo que el del micro servicio anterior. (limits-service)**

* + Dar de alta el nombre de la aplicación.

En el archivo **application.properties** agregar a siguiente línea:

#Este atributo lo puede obtener del pom.xml

spring.application.name=spring-cloud-config-server

* + Asignar el puerto en el que correrá la aplicación

En el archivo **application.properties** agregar la siguiente línea:

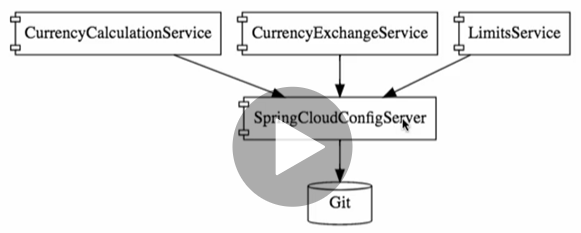
#Asignación del puerto

server.port=8888

* + Creación del repositorio git

El servidor de configuraciones de Spring Cloud realiza las configuraciones desde un repositorio git.

Conectaremos nuestro micro servicio **limits-service** al servidor de configuraciones de Spring Cloud **spring-cloud-config-server.**



Para generar el repositorio local de git, ejecutar el siguiente:

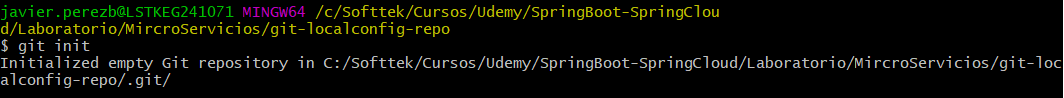
* + - Ubicarse en el directorio donde se va a generar el repositorio de git.

En este caso y a manera de ejemplo, crearemos el repositorio de git en el siguiente directorio:

**C:\Softtek\Cursos\Udemy\SpringBoot-SpringCloud\Laboratorio\MircroServicios\git-localconfig-repo**

* + - Ejecutar el comando:

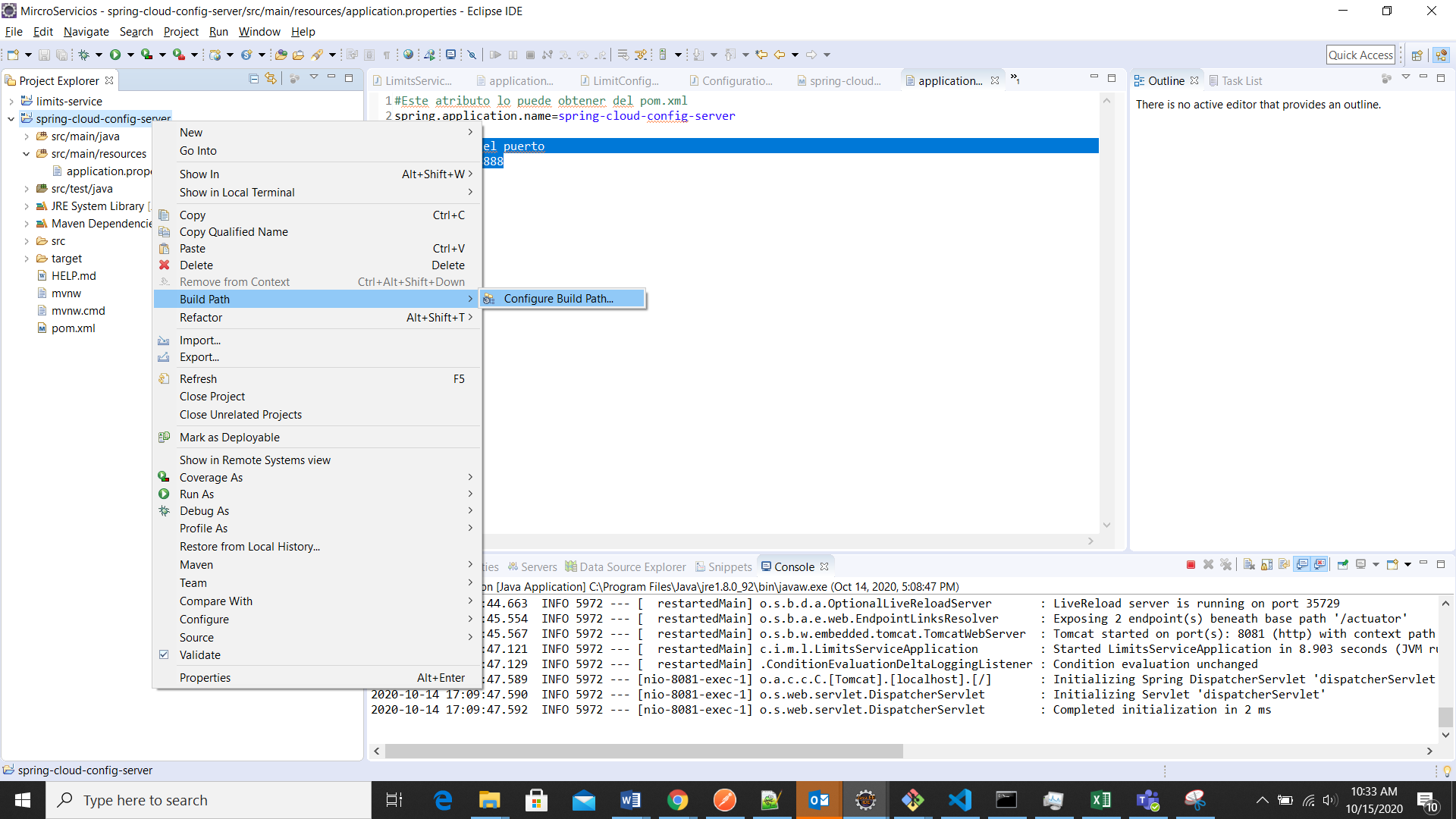
**git init**



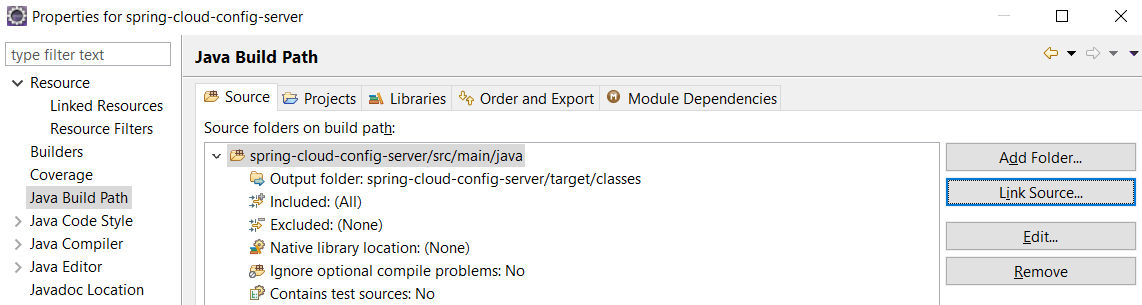
* + Agregar el directorio de git **git-localconfig-repo** (generado en el paso anterior) en nuestro aplicativo **spring-cloud-config-server.**

Para ello se debe realizar lo siguiente:

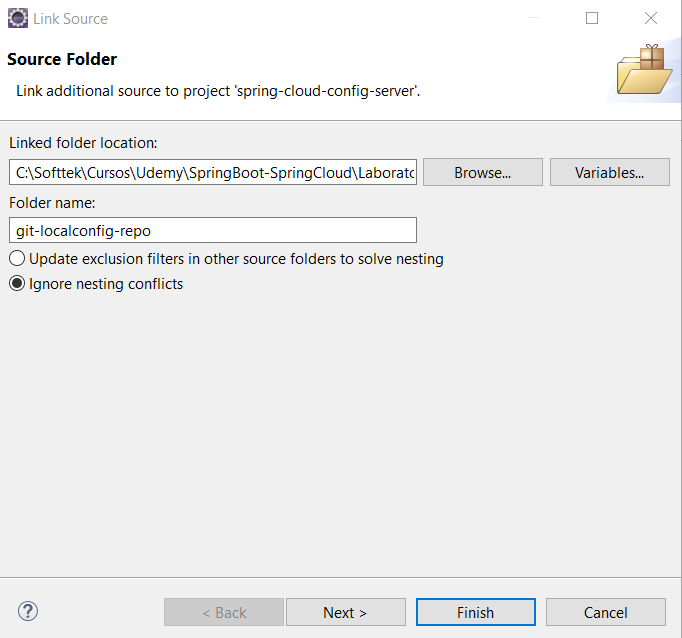
* + - Ir al aplicativo **spring-cloud-config-server** y dar clic derecho



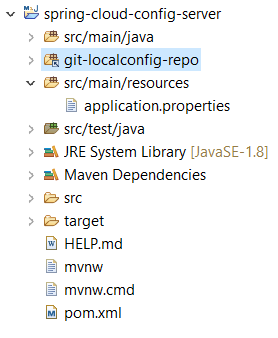
* + - Seleccionar la opción **Java Build Path** y dar clic en el botón **Linke Source**



* + - Seleccionar el directorio de nuestro repositorio de git y dar clic en el botón Finish.



Listo, con esto ya agregamos el directorio de git a nuestra aplicación **spring-cloud-config-server**



* + **Centralizar las configuraciones** de la aplicación **limits-service** en el servidor de configuraciones **spring-cloud-config-server.**

Para ello, necesitamos copiar la información requerida del archivo **application.properties** de la aplicación **limits-service** en un nuevo archivo dentro del repositorio git que acabamos de agregar a la aplicación **spring-cloud-config-server**.

* + - Crear un archivo en el repositorio de git que acabamos de agregar a la aplicación **spring-cloud-config-server**

El archivo lo llamaremos **limits-service.properties**  con el siguiente contenido:

#Estas propiedades son para el aplicativo limits-service

limits-service.minimum=8

limits-service.maximum=888

NOTA:

El contenido del archivo es una copia de la información que requerimos del archivo **application.properties** de la aplicación **limits-service**. (Se le cambian los valores para poder comprobar que en efecto se están consumiendo los parámetros del nuevo archivo de configuraciones)

* + - Guardar la información del archivo en el repositorio de git.

git add . –A

git commit -m “<descripción de lo que se está guradando>”

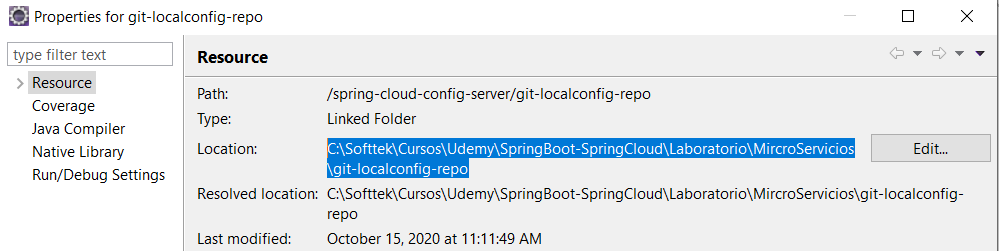
* + Conectar el servidor de configuraciones **spring-cloud-config-server** con el repositorio de git que ya hemos configurado.

El objetivo de este paso es ver cómo podemos centralizar la configuración de n servicios.

Esto es muy práctico, por ejemplo, cuando tiene n ambientes QA, Dev, etc… puede configurar en el servidor de configuraciones los n ambientes.

* + - Obtener la ubicación del reposito de git.

En la aplicación **spring-cloud-config-server** ubicar el repositorio de git y dar clic derecho, copiar la información de Location.



* + - Dar de alta la ubicación del repositorio de git en el archivo **application.properties** de la aplicación **spring-cloud-config-server**

#Ubicación del repositorio de git

spring.cloud.config.server.git.uri=file://C:\\Softtek\\Cursos\\Udemy\\SpringBoot-SpringCloud\\Laboratorio\\MircroServicios\\git-localconfig-repo

**NOTA:**

Como puede observar el formato de la ruta tiene un formato especial, de inicio, se agrega a la ruta el prefijo file//: y se sustituye la \ por \\

A partir de este momento, en el archivo **application.properties** de la aplicación **spring-cloud-config-server** se tienen configuraciones de las aplicaciones **spring-cloud-config-server** y de la aplicación **limits-service**

* + - Listo, con las configuraciones anteriores, debería poder levantar la aplicación **spring-cloud-config-server**. Ir a la siguiente url. (Recuerde que la aplicación corre en el puerto 8888)

NOTA:

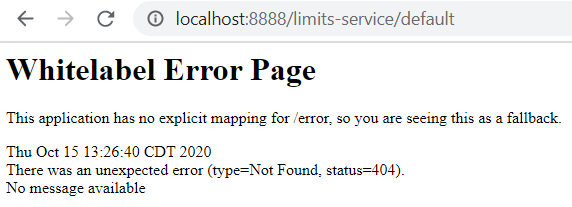
Como se conforma la URI:

/limits-service/default 🡪 la primera parte (limits-service) indica el nombre del archivo de configuraciones que definimos en el repositorio de git, la segunda parte indica que consultaremos el archivo default. (Más adelante se hacen algunos ejercicios al respecto)

Con este servicio vamos a consumir la versión **default** del contenido del archivo **limits-service** a través del servidor de configuraciones **spring-cloud-config-server.**



Se genera un error, por qué?



Solución:

Importante recordar que siempre que tengamos un servidor de configuraciones, éste lo debemos habilitar, para ello, debemos agregar la siguiente anotación en la clase inicial de nuestro aplicativo.

@EnableConfigServer

@SpringBootApplication

**public** **class** SpringCloudConfigServerApplication {

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

SpringApplication.*run*(SpringCloudConfigServerApplication.**class**, args);

}

}

Con la anotación **@EnableConfigServer** indicamos que la aplicación **spring-cloud-config-server** se está habilitando como un servidor de configuraciones.

Como podemos ver en la imagen de abajo, se está consumiendo la información del archivo **limits-service.properties** que generamos en el repositorio de git.

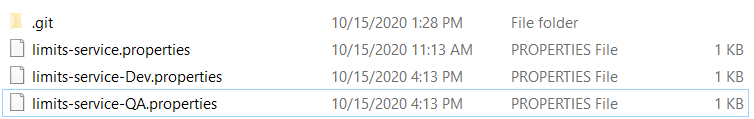
Con esto se comprueba que el servidor de configuraciones **spring-cloud-config-server** ya está conectado con el repositorio de git.



* + Generando archivos de configuración para diferentes ambientes. (Dev, QA, etc…)

Para ello, debemos crear archivos similares a **limits-service.properties** en el repositorio de git.

Para nuestro ejemplo, generaremos archivos para el ambiente Dev, QA:



Para el ambiente por default:



Para el ambiente de desarrollo (Dev):



Para el ambiente de pruebas QA:



Imaginemos el siguiente escenario:

Comentemos el valor **maximum** del archivo del ambiente de **desarrollo**, esto genera que el valor **maximum** se tome del ambiente por **default**, es decir, si no se encuentra alguno de los valores de los ambientes Dev, QA, entonces toma los valores del archivo del ambiente default.

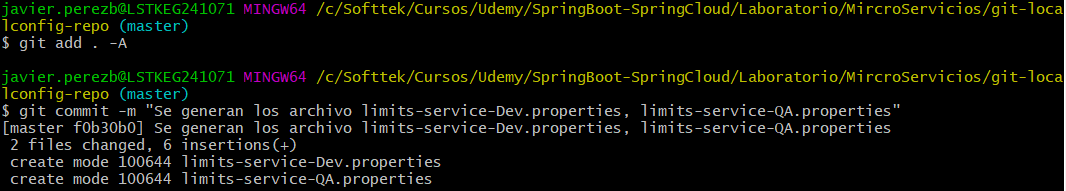


NOTA:

Los archivos que acabamos de generar en el repositorio de git (limits-service-Dev.properties, limits-service-QA.properties) no serán visibles para el servidor de configuraciones **spring-cloud-config-server** mientras no se guarden en el repositorio de git. Para guardar los archivos en el repositorio de git, ejecutar los siguientes comandos:

git add . –A

git commit -m “<descripcionDeLoQueSeAlmacena>”

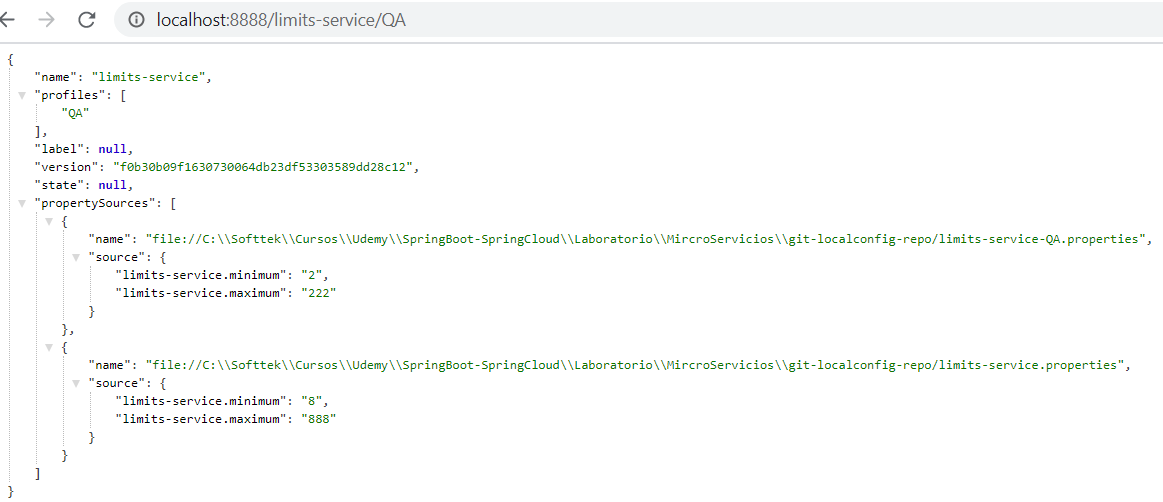


Consumiendo información de los archivos **limits-service-Dev.properties y limits-service-QA.properties**

Consumiendo información del archivo **limits-service-Dev.properties** importante notar que aparte de la información de Dev también nos proporciona la información del archivo default.



Consumiendo información del archivo **limits-service-QA.properties** importante notar que aparte de la información de Dev también nos proporciona la información del archivo default.



Y si queremos consumir información del archivo default **limits-service.properties**



* **Creación de micro servicio conectando la aplicación limits-service con el servidor de configuraciones spring-cloud-config-server.**

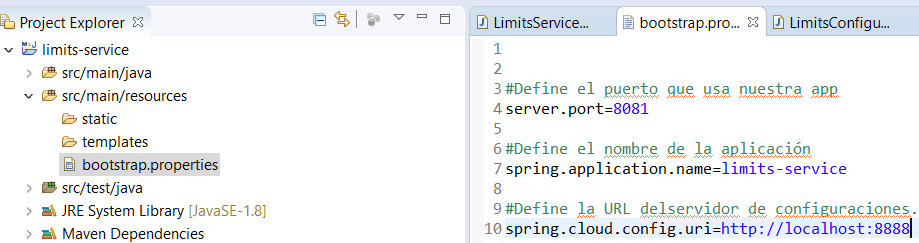
El objetivo es consumir la información de los archivos **limits-service.properties**, **limits-service-Dev.properties**, **limits-service-QA.properties** que se generaron en el repositorio de git a través del aplicativo **limits-service** y gestionando las configuraciones en el servidor de configuraciones **spring-cloud-config-server**

**Ventajas de separar las configuraciones de las aplicaciones:**

En los despliegues de las aplicaciones ya no se tiene que preocupar pos las configuraciones.

**Importante:**

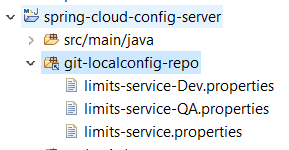
Para poder consumir las configuraciones del servidor de configuraciones **spring-cloud-config-server** debemos sustituir el archivo **application.properties** por **bootstrap.properties** en la aplicación **limits-service** y en este archivo indicar la URL del servidor de configuraciones.



**La otra cosa importante** es que el nombre de los archivos de configuraciones (.properties) generados en el repositorio de git que cargamos en el servidor de configuraciones **spring-cloud-config-server** deben iniciar con el prefijo igual al nombre de la aplicación que se quiere gestionar su configuración (En este caso y a manera de ejemplo: **limits-service**):

Nombre de la aplicación: **limits-service**

Nombre de los archivos de configuración generados en el repositorio de git contenido en el servidor de configuraciones: **limits-service-Dev.properties, limits-service-QA.properties, limits-service.properties**



Al reiniciar las aplicaciones se puede apreciar como ya se ha contemplado la administración de la aplicación **limits-service** en el servidor de configuraciones **spring-cloud-config-server**.

El orden en el que tiene que levantar las aplicaciones es:

* + **spring-cloud-config-server**
  + **limits-service**

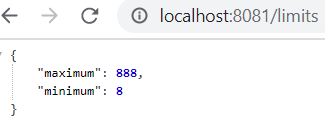
En la consola de la aplicación **limits-service** podremos ver algo similar a la siguiente salida:



En el log se observa que estamos usando el **profile** **defaul**, esto porque no se ha indicado un profile en particular (Dev, QA, etc…), esto implica que se consuma la información del archivo **limits-service.properties** generado en el repositorio de git

Consumir el servico **/limit** de la aplicación **limits-service** – pero utilizando los archivos de configuración (limits-service.properties, limits-service-Dev.properties, limits-service-QA.properties) generados en el repositorio de git.

Como se puede apreciar en la imagen de abajo, ya se está consumiendo la información del archivo **limits-service.properties** del repositorio de git



**Resumen para conectar la aplicación limits-service con el servidor de configuraciones spring-cloud-config-server**

* + Cambiar el nombre del archivo de propiedades de la aplicación **limits-service** de **application.properties** a **bootstrap.properties**
  + En el archivo **bootstrap.properties** definir la url del servidor de configuraciones.

#Define la URL delservidor de configuraciones.

spring.cloud.config.uri=http://localhost:8888

* + Utilizar los archivos de configuración definidos en el repositorio git y estos deben iniciar con un prefijo igual al nombre de la aplicación.
    - limits-service.properties 🡪 para el ambiente default
    - limits-service-Dev.properties 🡪 para el ambiente de desarrollo
    - limits-service-QA.properties 🡪 para el ambiente QA

**Configuración de profiles Dev y QA**

La configuración de los profile se puede hacer de varias formas, en este caso trabajaremos con los archivos de configuración. (.properties)

Para ello, necesitamos de lo siguiente:

* + Configuración de los profile en la aplicación **limits-service**

En el archivo **bootstrap.properties** agregar las siguientes líneas:

#Define el profile de desarrollo (Dev)

spring.profiles.active=Dev

**NOTA:**

El valor de la constante debe ser igual al sufijo del archivo de configuraciones creado en el repositorio git que se importó en el servidor de configuraciones **spring-cloud-config-server**, de no ser así se genera un error porque no se encuentra al archivo del profile.

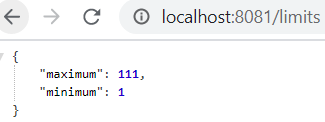
Por ejemplo:

Hemos definido el profile **Dev** por lo que el archivo de configuración se debe llamar **limits-service-Dev.properties**. (De no ser así, se genera un error)

* + Consumir el servicio utilizando el archivo de configuración de desarrollo. (Dev)

Ir a la siguiente URL: <http://localhost:8081/limits>

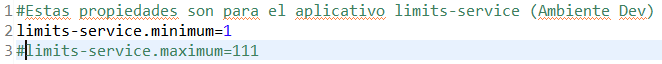
Como se puede apreciar en el imagen de abajo, se está consumiendo el servicio **/limits** y se está utilizando el archivo de configuración de desarrollo 🡪 **limits-service-Dev.properties**



**¿Qué pasa si comentamos alguno de los valores del archivo limits-service-Dev.properties?** por ejemplo, comentemos el valor **maximum**

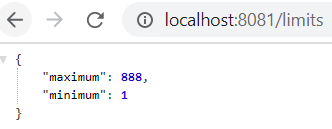
Lo que pasa es que al no encontrar el valor en el archivo de desarrollo, toma el valor del archivo del ambiente default. (**limits-service.properties**)

**Recuerde que para que el servidor de configuraciones tome en cuenta los cambios realizados, se deben guardar en el repositorio de git.**

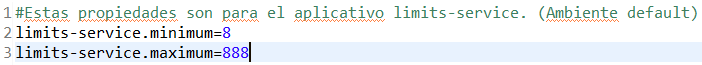


También se debe reiniciar la aplicación **limits-service** para que tome los cambios realizados a los archivos de configuración del repositorio git.

Listo, al consumir el servicio **/limits** debemos obtener lo siguiente:



Como no encontró la información de **maximum** en el ambiente de desarrollo, toma la información del ambiente default.



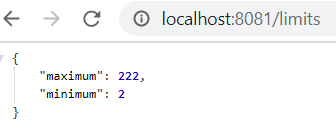
* + Consumir el servicio utilizando el archivo de configuración de pruebas. (QA)

Para ello, debemos habilitar en el archivo **bootstrap.properties** del aplicativo **limits-service** el ambiente QA

#Define el profile de pruebas (QA)

spring.profiles.active=QA

Al consumir el servicio /limits, debemos obtener los siguientes resultados:



Como se puede apreciar en la imagen de arriba, los valores coinciden con la información del archivo de configuraciones del ambiente QA.



* **Interacción entre micro servicios.**

Para este apartado, se realizara el siguiente ejercicio:

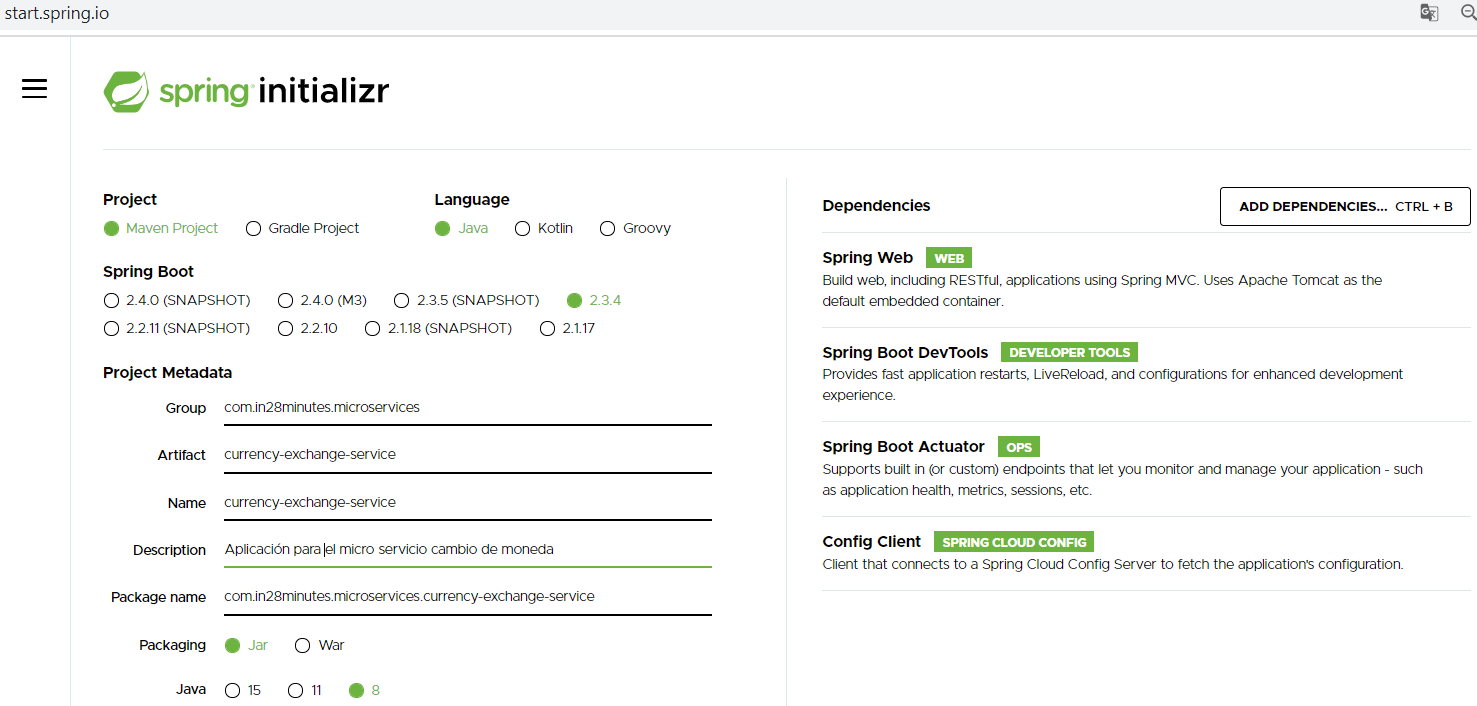
Crearemos dos micro servicios “servicio de cambio de moneda” y el “servicio de cálculo de moneda” los cuales interactuaran para obtener información que requieren para poder llevar a cabo su trabajo. (Cada uno de los servicios tendrá n instancias disponibles, cada una de las instancias se levantaran en puertos diferentes)

Ver la siguiente imagen.



Para ello, requerimos de lo siguiente:

* + Creación del micro serviciocambio de divisas que lo llamaremos **currency-exchange-service** con la siguiente configuración:



**Definir el nombre de la aplicación:**

En el archivo **application.properties** agregar las siguientes líneas:

#Define el nombre de la aplicación

spring.application.name=currency-exchange-service

**Definir el puerto que ocupara la aplicación:**

# Define el puerto que ocupara la aplicación

server.port=8000

Crear el siguiente servicio rest en la aplicación **currency-exchange-service**.

Para ello, requerimos de lo siguiente:

* + - Creación del siguiente bean.

**public** **class** ExchangeValue {

**private** Long id;

**private** String from;

**private** String to;

**private** BigDecimal multiploConversion;

...

}

* + - Creación de la siguiente clase controller con el siguiente servicio:

@RestController

**public** **class** CurrencyExchangeController {

@GetMapping("/currency-exchange/from/{from}/to/{to}")

**public** ExchangeValue retrieveValue(@PathVariable String from, @PathVariable String to) {

**return** **new** ExchangeValue(1000L, from, to, BigDecimal.*valueOf*(65));

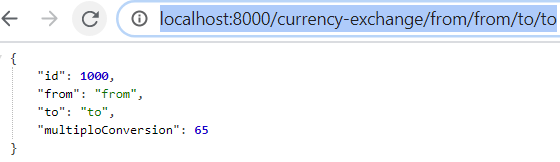
}

}

Verificar que el servicio esté disponible.

Ir a la siguiente url: <http://localhost:8000/currency-exchange/from/from/to/to>

Recordemos que el servicio espera dos parámetros **from** y **to**



Listo, el servicio ya se encuentra disponible.

Recordemos que los servicios tendrán n instancias disponibles y nos interesa saber que instancia es la que nos está ateniendo, para ello, vamos a requerir que en la respuesta venga el puerto de a instancia que nos está atendiendo, por consiguiente, necesitamos agregar al bean **ExchangeValue** el atributo para el puerto.

**public** **class** ExchangeValue {

**private** Long id;

**private** String from;

**private** String to;

**private** BigDecimal multiploConversion;

**private** **int** puerto;

**...**

}

También necesitamos modificar el servicio **retrieveValue** de la siguiente manera:

@RestController

**public** **class** CurrencyExchangeController {

//Con esta varible obtenemos el puerto que está usando la aplicación.

@Autowired

Environment environment;

@GetMapping("/currency-exchange/from/{from}/to/{to}")

**public** ExchangeValue retrieveValue(@PathVariable String from, @PathVariable String to) {

ExchangeValue exchangeValue = **new** ExchangeValue(1000L, from, to, BigDecimal.*valueOf*(65));

exchangeValue.setPuerto(Integer.*parseInt*(environment.getProperty("local.server.port")));

**return** exchangeValue;

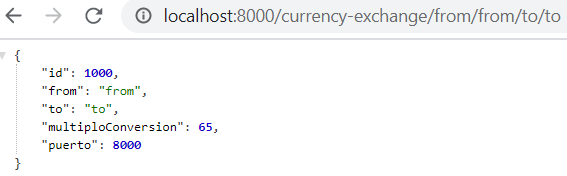
}

}

**NOTA:**

Con el objeto environment obtenemos el puerto que está usando la aplicación. Para obtener el puerto, necesitamos la siguiente instrucción: environment.getProperty("local.server.port")

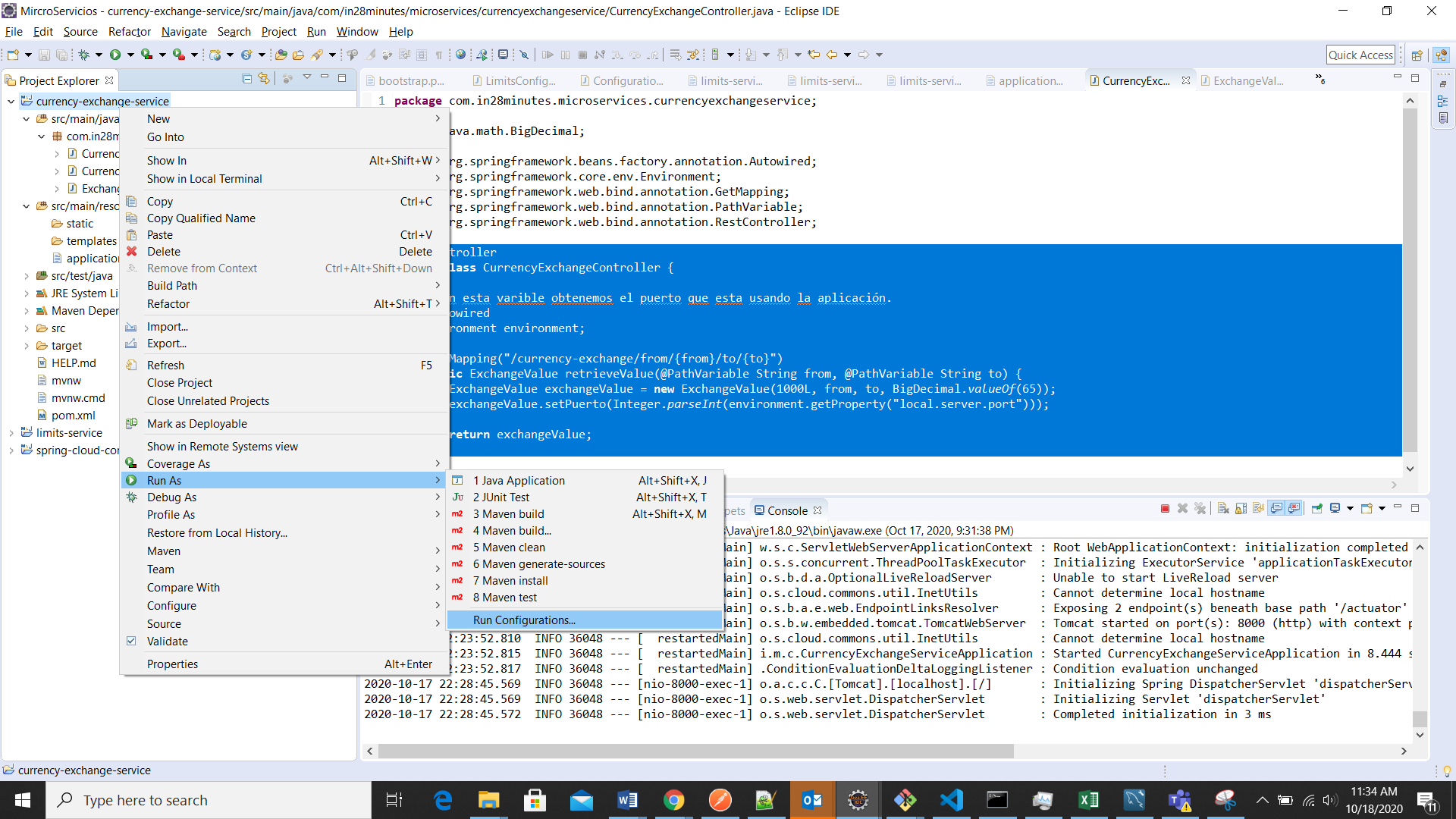
Verificar que la aplicación siga respondiendo después de realizar los cambios anteriores:



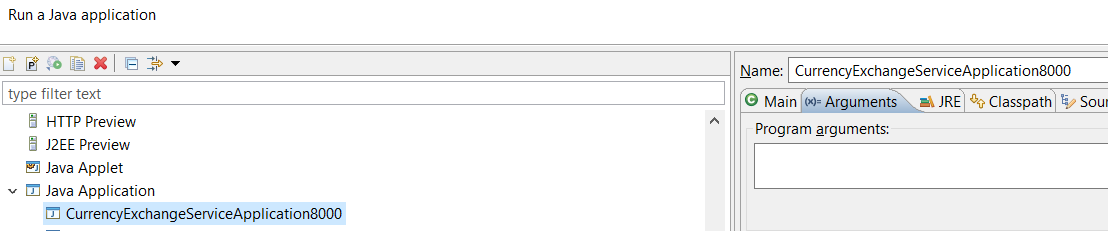
**Configuración externa del puerto de la aplicación (Se generan dos instancias del micro servicio CurrencyExchangeServiceApplication corriendo en los puertos 8000 y 8001) :**

Para ello, necesitamos realizar lo siguiente:

Clic derecho sobre el proyecto 🡪 Run As 🡪 Run Configurations…



Y renombrar el Java application de nuestro proyecto con el nombre **CurrencyExchangeServiceApplication8000**. (Para indicar que esta instancia ocupa el puerto 8000)



Copiamos el archivo, para ello, damos clic derecho sobre el archivo y seleccionamos **Duplicate,** al archivo duplicado lo renombramos, en este caso y a manera de ejemplo, lo llamaremos **CurrencyExchangeServiceApplication8001** (Para indicar que esta instancia ocupa el puerto 8001)

En la instancia **CurrencyExchangeServiceApplication8001** seleccionar la pestaña **Arguments** e introducir el siguiente parámetro: **-Dserver.port=8001**

Para confirmar los cambios dar clic en el botón **Apply** y finalmente dar clic en el botón **Run** para indicarle a la aplicación que la instancia con el puerto 8001 está habilitado.

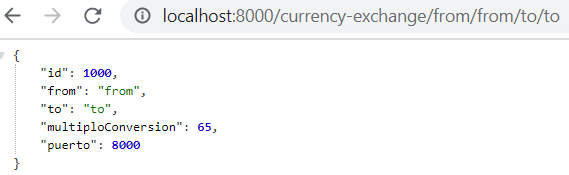
NOTA: Las configuraciones que acabamos de hacer (configuraciones externas) sobre escriben las configuraciones definidas en el archivo **application.properties**

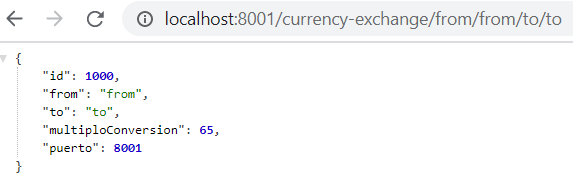


**Listo, ahora tenemos dos instancias de micro servicio CurrencyExchangeServiceApplication corriendo en los servicios 8000 y 8001**

Verificar que la aplicación **CurrencyExchangeServiceApplication** se encuentra disponible en ambas isntancias (puerto 8000 y 8001):

Note que aparte de respondernos en ambos puertos, en la repuesta se devuelve el puerto de la instancia que no está atendiendo.





**Agreguemo las dependencias necesarias para poder usar una base de datos en memoria como H2 y JPA**

Para ello, requerimos lo siguiente:

* + - Agregr al **pom.xml** las siguientes dependencias:

<!-- Para el manejo de JPA -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-jpa</artifactId>

</dependency>

<!-- Para la base de datos en memoria (H2) -->

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

</dependency>

* + - Agregar las siguientes anotaciones al bean **ExchangeValue.java**

@Entity

**public** **class** ExchangeValue {

@Id

@GeneratedValue

**private** Long id;

@Column(name = "currency\_from")

**private** String from;

@Column(name = "currency\_to")

**private** String to;

**private** BigDecimal multiploConversion;

**private** **int** puerto;

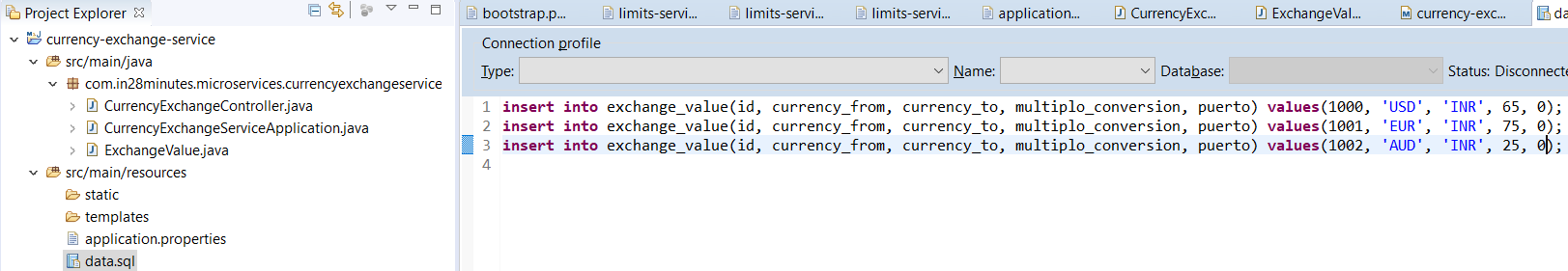
…

}

Las modificaciones al bean son para indicarle a spring boot que el bean se debe de mapear con una tabla de la base de datos.

En los atributos de clase from, to se les agrega la anotación **@Column** para cambiar el nombre de la columna en la base de datos, ya que la palabra from es una palabra reservada. (Y en el caso de la columna to para ser un poco consistentes con el nombrado de los atributos)

* + - Crear el archivo **data.sql** en src/main/resourcespara nuestra base de datos en memoria.



**Nota:**

La notación camellada en las clases java se sustituye con guiones bajos en base de datos, por ejemplo:

multiploConversion 🡪 multiplo\_conversion

* + - Habilitar el log y la consola de la base de datos de H2.

Para ello, agregar las siguientes líneas en el archvio **application.properties**

#Para imprimir las consultas SQL que se ejecutan

spring.jpa.show-sql=true

#Para habilitar la consola de la base de datos H2

spring.h2.console.enabled=true

#Para la configuración de la base de datos H2

spring.datasource.url=jdbc:h2:mem:testdb

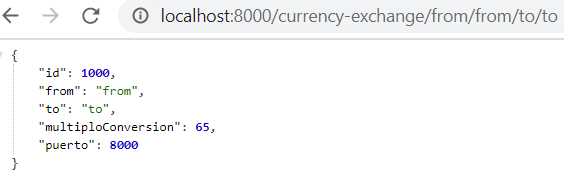
spring.data.jpa.repositories.bootstrap-mode=default

**NOTA:**

**La última configuración es muy importante, ya que de no ponerla, no podrá ingrsar a la consola de la base de datos H2.**

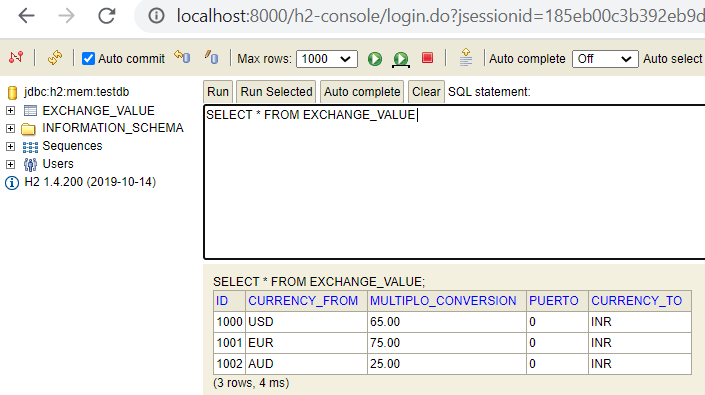
* + - Verificar que el micro servicio esta disponible.

Para ello, ir a la URL: <http://localhost:8000/currency-exchange/from/from/to/to>



* + - Verificar que la consola de H2 ya esta habilitada.

Para ello, ir a la URL: <http://localhost:8000/h2-console>



**Consumir la información de la base de datos H2 mediante JPA**

* + - Crear el repository para interactuar con la base de datos H2.

@Repository

**public** **interface** ExchangeValueRepository **extends** JpaRepository<ExchangeValue, Long> {

}

Pero en este caso, requerimos hacer consultas un poco más especializadas, necesitamos hacer búsquedas por las columnas from, to. Por coniguiente, debemos definir nuestra consulta en el repository **ExchangeValueRepository.**

/\*\*

\* Para que JpaRepository pueda interpretar nuestra consulta, los parámetros de la consulta

\* se deben llamar igual que como fueron nombrados en el bean ExchangeValue.java

\* **@param** from

\* **@param** to

\* **@return**

\*/

**public** ExchangeValue findByFromAndTo(String from, String to);

NOTA:

Para que JPA pueda interpretar la consulta, los parámetros de la consulta se deben llamar exactamente igual que como fueron nombrados en el entity de java. (ExchangeValue.java)

* + - Consumir en el controller la consulta del repository que acabamos de generar en el paso anterior.
      * Inyectar la dependencia del repository en el controller.

@Autowired

ExchangeValueRepository exchangeValueRepository;

* + - * Utilizar el servicio que generamos en el repository para obtener la información que requerimos. (Modificar la versión original del servicio)

@GetMapping("/currency-exchange/from/{from}/to/{to}")

**public** ExchangeValue retrieveValue(@PathVariable String from, @PathVariable String to) {

ExchangeValue exchangeValue = exchangeValueRepository.findByFromAndTo(from, to);

**if**( exchangeValue != **null** )

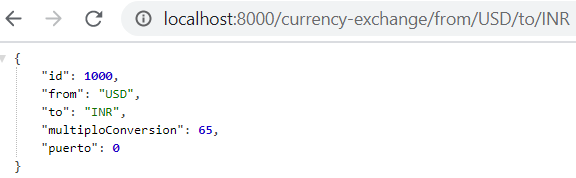
exchangeValue.setPuerto(Integer.*parseInt*(environment.getProperty("local.server.port")));

**return** exchangeValue;

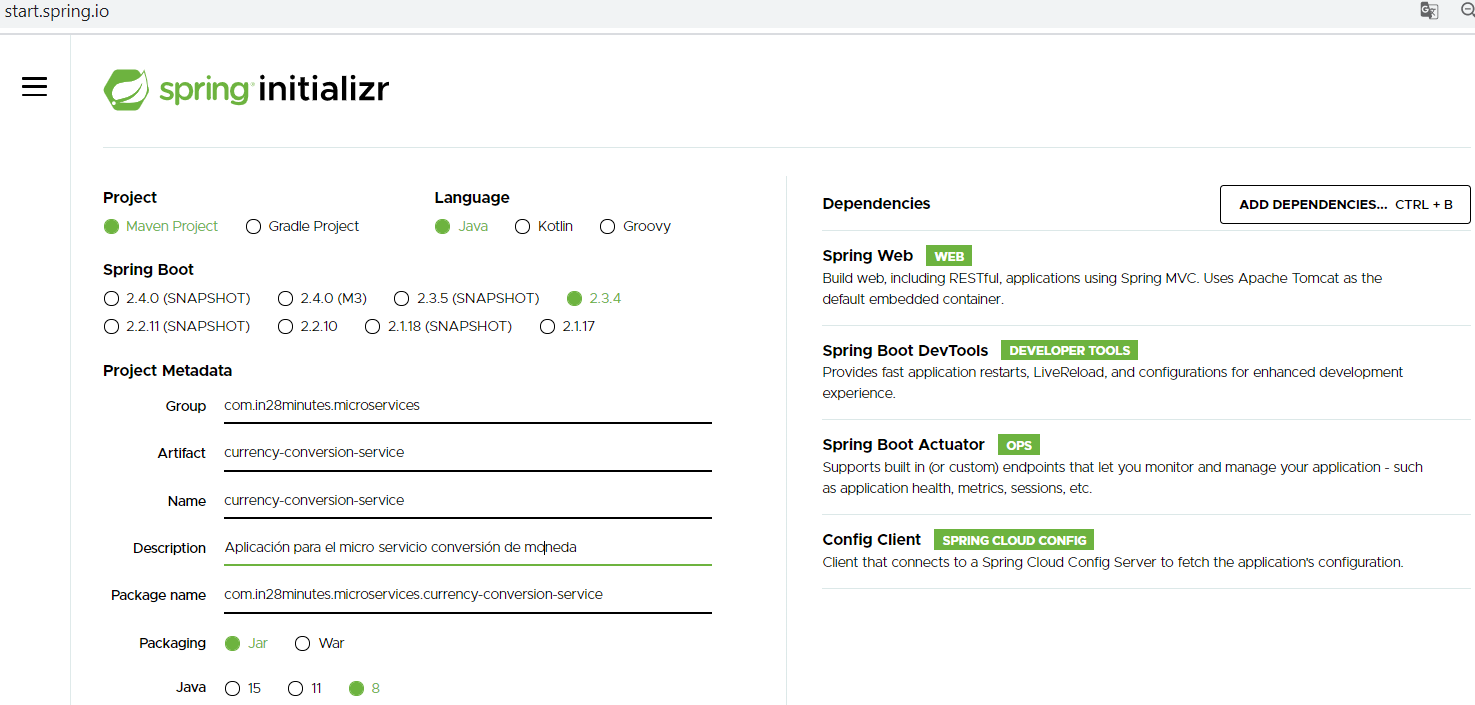
}

* + - Verificar que el servicio está respondiendo correctamente.

Importante notar que se le están pasando los parámetros de la consulta: **USD, INR.** (En efecto, la información que nos devuelve la consulta es la que se almaceno en la base de datos H2)



* + Creación del micro servicioconversión de divisas que lo llamaremos **currency-conversion-service** con la siguiente configuración:



* + - Configurar el nombre y puerto de la aplicación.

Para ello, debemos ingresar los siguientes valores en al archivo **application.properties**

#Define el nombre de la aplicación

apring.application.name=currency-conversion-service

#Define el puerto que debe usar la aplicación

server.port=8100

* + - Crearemos un servicio para hacer los cálculos de conversión de un tipo de moneada a otro.

Para ello, requerimos de lo siguiente:

* + - * Creación del siguiente bean.

**package** com.in28minutes.microservices.currencyconversionservice;

**public** **class** CurrencyConversionBean {

**private** Long id;

**private** String from;

**private** String to;

**private** BigDecimal multiploConversion;

**private** BigDecimal quantity;

**private** BigDecimal totalMontoCalculado;

**private** **int** puerto;

**...**

}

* + - * Creación de la siguiente clase controller:

**package** com.in28minutes.microservices.currencyconversionservice;

@RestController

**public** **class** CurrencyConversionController {

@GetMapping(name = "/currency-converter/from/{from}/to/{to}/quantity/{quantity}")

**public** CurrencyConversionBean convertCurrency(@PathVariable String from, @PathVariable String to, @PathVariable BigDecimal quantity) {

**return** **new** CurrencyConversionBean(1L, from, to, BigDecimal.***ONE***, quantity, quantity, 0);

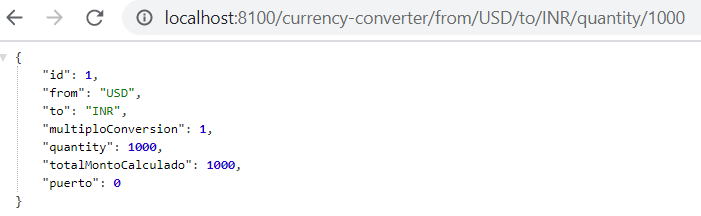
}

}

* + - Verificar que el servicio que acabamos de crear está disponible.

Ir a la URL: <http://localhost:8100/currency-converter/from/USD/to/INR/quantity/1000>

Los valores de los parámetros que pasamos en la URL son: **USD, INR, 1000**



* + Invicación de servicios entre las aplicaciones **currency-exchange-service**, **currency-conversion-service**

Para poder realizar la interacción de los servicios de las aplicaciones **currency-exchange-service**, **currency-conversion-service** debemos realizar lo siguiente:

* + - En el servicio "/currency-converter/from/{from}/to/{to}/quantity/{quantity}" de la aplicación **currency-conversion-service** se mandallamar el servicio "/currency-exchange/from/{from}/to/{to}" de la aplicación **currency-exchange-service**

@RestController

**public** **class** CurrencyConversionController {

@GetMapping("/currency-converter/from/{from}/to/{to}/quantity/{quantity}")

**public** CurrencyConversionBean convertCurrency(@PathVariable String from, @PathVariable String to, @PathVariable BigDecimal quantity) {

Map<String, String> uriVariables = **new** HashMap<String, String>();

uriVariables.put("from", from);

uriVariables.put("to", to);

/\*\*

\* Los parámetros que se pasan son:

\* Primer parámetro: URI del servicio de la aplicacion currency-exchange-service necesitamos consumir.

\* Segundo parámetro: La respuesta se debe devolver en un objeto de tipo CurrencyConversionBean.class.

\* Tercer parámetro: Se pasan los parámetros que requiere el servicio que estamos invocando.

\*

\* IMPORTANTE:

\* Invocamos el servicio http://localhost:8000/currency-exchange/from/{from}/to/{to} que devuelve un objeto de tipo ExchangeValue.java

\* pero le indicamos al RestTemplate que la respuesta la mepee a un objeto de tipo CurrencyConversionBean.java

\* por lo que es imortante que los campos en comun se llemen igual en ambos beans (id, from, to, multiploConversion) de no ser así no se mapearan correctamente los

\* valores.

\*/

ResponseEntity<CurrencyConversionBean> responseEntity = **new** RestTemplate().getForEntity("http://localhost:8000/currency-exchange/from/{from}/to/{to}", CurrencyConversionBean.**class**, uriVariables);

//Obtenemos la respuesta del servicio.

CurrencyConversionBean response = responseEntity.getBody();

//Obtenemos la información que necesitamos.

**return** **new** CurrencyConversionBean(response.getId(), from, to, response.getMultiploConversion(), quantity, quantity.multiply(response.getMultiploConversion()), response.getPuerto());

}

}

NOTA:

Como posemos observar, el servcio "/currency-converter/from/{from}/to/{to}/quantity/{quantity}" de la aplicación **currency-conversion-service** invoca el servicio "/currency-exchange/from/{from}/to/{to}" por el puerto 8000 de la aplicación **currency-exchange-service.**

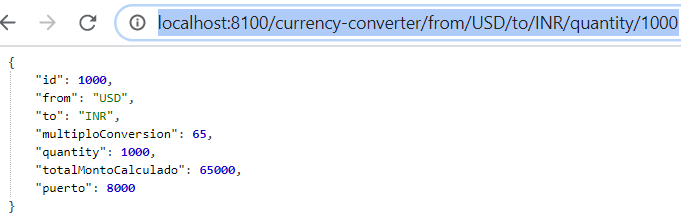
Por consiguiente, debemos tener corriendo las aplicaciones **currency-conversion-service, currency-exchange-service** (Debemos tener activa la instancia del puerto 8000)**.**

**En la url del servicio que invocamos en el RestTemplate es importante poner el http:// de lo contrario se genera un error. (**"http://localhost:8000/currency-exchange/from/{from}/to/{to}"**)**

* + - Validar que el servicio "/currency-converter/from/{from}/to/{to}/quantity/{quantity}" de la aplicación **currency-conversion-service** esta respondiendo correctamente.

Ir a la URL: <http://localhost:8100/currency-converter/from/USD/to/INR/quantity/1000>

Como podemos apreciar en la imagen se abajo, la información de los campos **id, from, to, multiploConversion** se obtienen de la base de datos H2 de la aplicación **currency-exchange-service** a través del servicio "/currency-exchange/from/{from}/to/{to}" y los valores de los campos **totalMontoCalculado, puerto** se generan en la aplicación **currency-conversion-service** a través del servicio "/currency-converter/from/{from}/to/{to}/quantity/{quantity}"



Conclusión:

La comunicación entre los servicios converter/from/{from}/to/{to}/quantity/{quantity}", "/currency-exchange/from/{from}/to/{to}" de las aplicaciones **currency-conversion-service, currency-exchange-service** (respectivamente) fue exitosa.

* **Interacción de micro servicios implementando Feign.**

**Feign:** Para simplificar la invocación de micro servicios dentro de otros micro servicios. (Feign es un cliente de servicios Rest)

Para la configuración de Feing, requerimos de lo siguiente:

* + Agregar las siguientes dependencias al pom.xml

Esta dependencia la hereda spring boot de Netflix

<!-- Facilita la interaccion entre micro servicios -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-openfeign</artifactId>

</dependency>

* + Habilitar Feign para la búsqueda de clientes. (micro servicios)

En la clase donde se genera el arranque de la aplicación debemos poner la siguiene anotación:

@SpringBootApplication

@EnableFeignClients("com.in28minutes.microservices.currencyconversionservice")

**public** **class** CurrencyConversionServiceApplication {

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

SpringApplication.*run*(CurrencyConversionServiceApplication.**class**, args);

}

}

**Nota:**

Con la anotación **@EnableFeignClients** se habilita **Feign** y se le pasa como parámetro el paquete donde debe escanear la búsqueda de servicios. (En este caso y amanera de ejemplo, se le indica que la búsqueda de servicios la debe realizar en el paquete "com.in28minutes.microservices.currencyconversionservice)

* + Creación de proxy Feign.

Lo llamaremos **CurrencyExchangeServiceProxy** ya que estaremos invocando servicios del micro servicio **currency-exchange-service**

@FeignClient(name="currency-exchange-service", url="localhost:8000")

**public** **interface** CurrencyExchangeServiceProxy {

@GetMapping("/currency-exchange/from/{from}/to/{to}")

**public** CurrencyConversionBean retrieveValue(@PathVariable("from") String from, @PathVariable("to") String to);

}

**Notas:**

Con la anotación **@FeignClient** se importan los servicios indicados en la nube.

Con la anotación **@FeignClient**(name="currency-exchange-service", url="localhost:8000") se indica el nombre y URL de la aplicación del micro servicio que se va a consumir. (En la URL se indica la isntancia del servicio que se desea ocupar, en este caso se esta indicando que se va a usar la instancia del puerto 8000)

El nombrado y los parámetros del método son los mismo tanto en el proxy que acabamos de crear como en el controller de la apliación **currency-exchange-service**, la única diferencia es el tipo de valor de retorno, en el método del controller de la aplicación **currency-exchange-service** devuelve un objeto de tipo **ExchangeValue.java** y en el proxy que acabamos de crear se devuelve un objeto de tipo **CurrencyConversionBean.java** por lo que si hay atributos de instancia de las clases en común, estos de deben llamar exactamente igual para que se pueda hacer el mapeo correcto de la información.

Ademas, se debe especificar puntualmente el nombre del request que se debe pasar como parámetro @PathVariable("from") de no hacerlo, segenera un erro, ya que spring boot no sabe que parámetros del request debe mapear en lo parámetros del método.

Conclusión:

Con la creación de esta interface, estamo indicando el nombre y la URL de la aplicación del micro servicio que necesitamos invocar, también indicamos el nombre del(los) servicio(s) que estaremos invocando. Poner especial atención en el tipo de dato de retorno, ver la nota anterior.

* + Consumir servicios a través del proxy recién creado. (**CurrencyExchangeServiceProxy**)

Para ello, requerimos hacer los siguientes ajustes en el controller CurrencyConversionController.java

* + - Agregar el siguiente proxy Feign en la clase CurrencyConversionController.java.

@Autowired

**private** CurrencyExchangeServiceProxy proxy;

* + - Consumir los servicios del micro servicio **currency-exchange-service** a través del proxy.

@GetMapping("/currency-converter-feign/from/{from}/to/{to}/quantity/{quantity}")

**public** CurrencyConversionBean convertCurrencyFeign(@PathVariable String from, @PathVariable String to, @PathVariable BigDecimal quantity) {

//A través del proxy consumimos el servicio de la aplicación currency-exchange-service.

CurrencyConversionBean response = proxy.retrieveValue(from, to);

//Obtenemos la información que necesitamos.

**return** **new** CurrencyConversionBean(response.getId(), from, to, response.getMultiploConversion(), quantity, quantity.multiply(response.getMultiploConversion()), response.getPuerto());

}

NOTA:

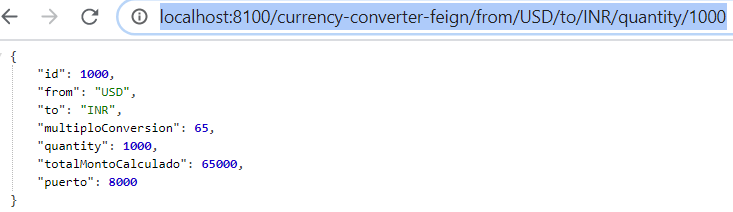
Como se puede observar, el manejo de la interacción entre micro servicios es más fácil.

* + - Verificar que el servicio esta disponible y está respondiendo bien.

Ir a la URL: <http://localhost:8100/currency-converter-feign/from/USD/to/INR/quantity/1000>

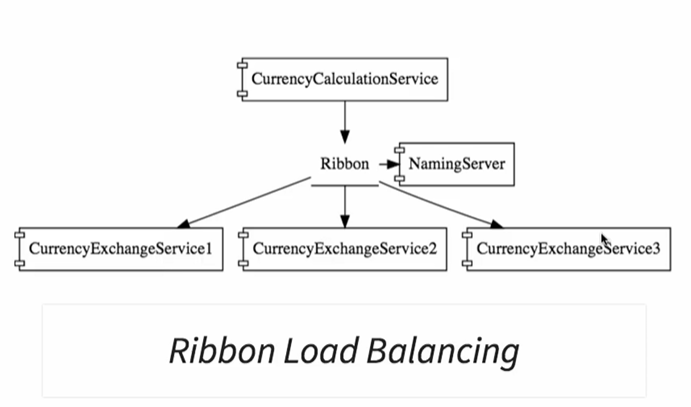
Como podemos apreciar en la imagen de abajo, la infromacion de los campos

* + - * id, from, to, puerto son los que se almacenaron en la base de datos de la aplicación **currency-exchange-service** (A excepción del puerto, este dato es el puerto a través del cual nos atendio el micro servicio)
      * totalMontoCalculado se calcula en el micro servicio **currency-conversion-service**



Listo, la interacción de los micro servicios mediante Feign se genero de manera exitosa.

* **Balanceo de carga de micro servicios medinate Ribbon.**
  + **Ribbon:** Para se utiliza para el balanceo de carga en los micro servicios.



Para la configuración de Ribbon, necesitamos lo siguiente:

* + Agregar la siguiente dependencia en el archivo pom.xml de la aplicación

<!-- Para el balanceo de carga en los micro servicios -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-ribbon</artifactId>

</dependency>

* + Habilitar cliente de Ribbon en el proxy que generamos en pasos anteriores.

Para ello, debemos agregar la siguiente a notación en la interface CurrencyExchangeServiceProxy.java

//@FeignClient(name="currency-exchange-service", url="localhost:8000")

@FeignClient(name="currency-exchange-service")

@RibbonClient(name="currency-exchange-service")

**public** **interface** CurrencyExchangeServiceProxy {

@GetMapping("/currency-exchange/from/{from}/to/{to}")

**public** CurrencyConversionBean retrieveValue(@PathVariable("from") String from, @PathVariable("to") String to);

}

Con la anotación **@RibbonClient** habilitamos el cliente de Ribbon y le indicamos el nombre del micro servicio en el que debe realizar el balanceo de carga.

De la anotación **@FeignClient** se elimina el parámetro de la URL, éstoporque ahora ya no queremos invocar una instancia en particular del micro servicio **currency-exchange-service,** ahora lo que nos interesa es interactuar con todas la instancias disponibles del micro servicio **currency-exchange-service** y hacer un balanceo de carga con todas estás instancias.

* + Configurar las instancias del micro servicio **currency-exchange-service** sobre los que se va a generar el balanceo de carga.

Agregar en el archivo **application.properties** las siguientes constantes:

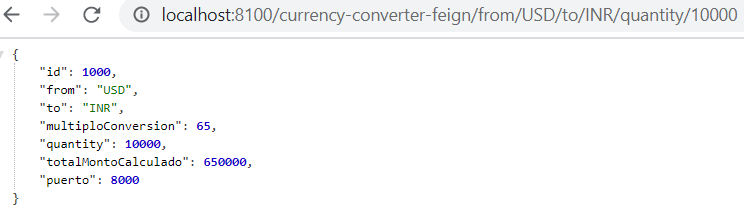
#Define las instancias del aplicativo currency-exchange-service sobre los que se genera el balanceo de carga. (En este caso se definen dos instancias)

currency-exchange-service.ribbon.listOfServers=http://localhost:8000,<http://localhost:8001>

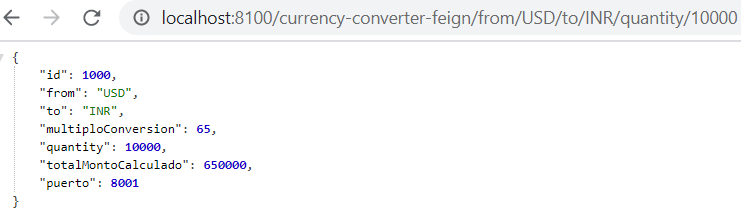
Para corroborar que los cambios ya se tomaron en cuneta, reiniciar la aplicación y consuma los servicios:

Ir a la URL: <http://localhost:8100/currency-converter-feign/from/USD/to/INR/quantity/10000>

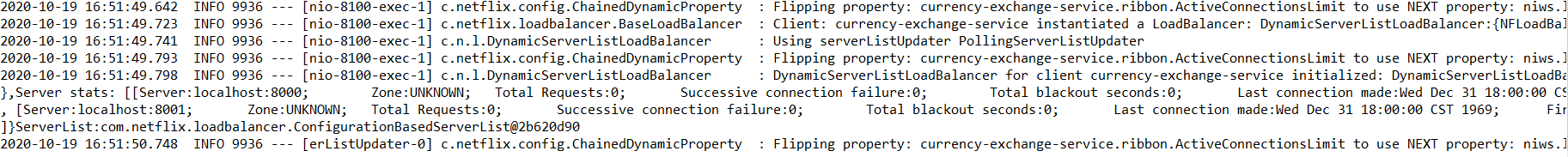
Como puede observar, la primera vez nos atiende la instancia del micro servicio **currency-exchange-service** desplegado en el puerto 8080. (Ver en la respuesta el campo **puerto**)



La segunda vez nos atiende la instancia del micro servicio **currency-exchange-service** desplegado en el puerto 8081. (Ver en la respuesta el campo **puerto**)



Y en la consola del micro servicio **currency-conversion-service** debe visualizar algo similar a lo siguiente:



Conclusión:

Con las configuraciones implementadas se pudo implementar exitosamente el balanceo de carga en las n instancias del micro servicio **currency-exchange-service.** (Para efectos de nuestro ejemplo, se levantaron dos instancias en los puertos 8000 y 8001)

* **Aumentar o disminuir instancias de micro servicios de forma dinámica implementando Eureka que es un naming server. (También conocido como servicio de descubrimiento)**

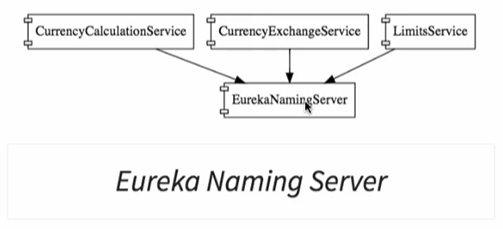
Las dos principales características de **Eureka** **naming server** son:

* + Alta el registro de los servicios.
  + Descubrimiento de los servicios. (Proporciona información de cada uno de los servicios que tiene dados de alta cuando alguien se lo solicita)
  + Facilitar la gestión de la comunicación entre micro servicios. (Ya no tenemos que configurar en cada una de las aplicaciones y a mano las instancias disponibles de cada una de las aplicaciones, ahora, basta con dar de alta las instancias de las aplicaciones en el servidor de nombrado eureka)

Ejemplo de las configuraciones a mano que teníamos que realizar en cada una de las aplicaciones:

#Define las instancias del aplicativo currency-exchange-service sobre los que se genera el balanceo de carga.

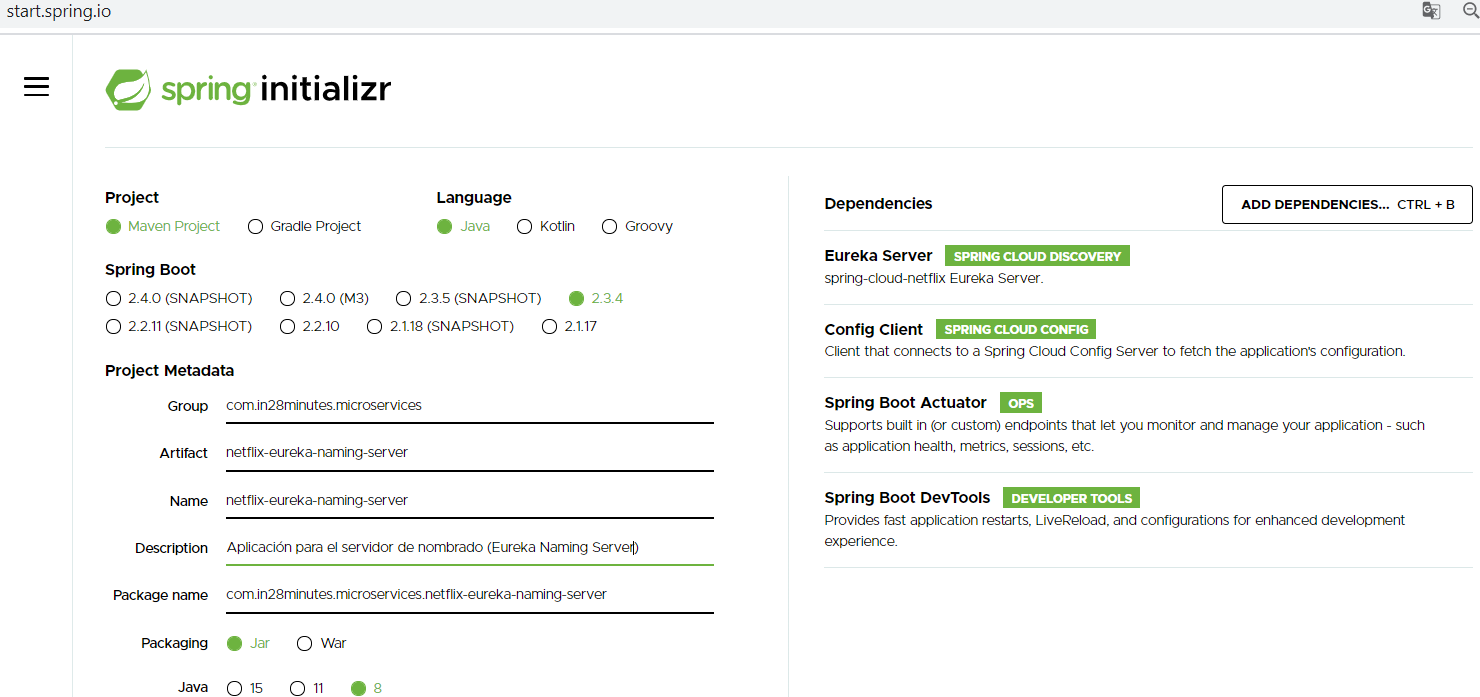
currency-exchange-service.ribbon.listOfServers=http://localhost:8000,http://localhost:8001



Para poder implementar el funcionamiento de micro servicios mediante **Eureka Naming Server** requerimos de lo siguiente:

* + Creación del servidor de nombrado Eureka.

Generar un proyecto de spring boot con la siguiente configuración:



* + Ralizar las siguientes configuraciones en el servidor de nombrado de Eureka.
    - Asignar el nombre y puerto de la aplicación.

Para ello, agregar las siguientes constantes en el archivo **application.properties**

#Define el nombre de la aplicación.

spring.application.name=netflix-eureka-naming-server

#Define el puerto que debe usar la aplicación.

server.port=8761

* + - Definición de algunas configuraciones del servidor de nombrado eureka.

Agregar en el archivo **application.properties** las siguientes líneas.

#Para evitar que el servidor de eureka contemple a esta aplicacion (server) como parte de sus registros.

eureka.client.register-with-eureka=false

#

eureka.client.fetch-registry=false

* + Habilitar el servidor de nombrado eureka.

Para ello, debemos agregar la siguiente anotación en la clase principal de la aplicación.

@SpringBootApplication

@EnableEurekaServer

**public** **class** NetflixEurekaNamingServerApplication {

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

SpringApplication.*run*(NetflixEurekaNamingServerApplication.**class**, args);

}

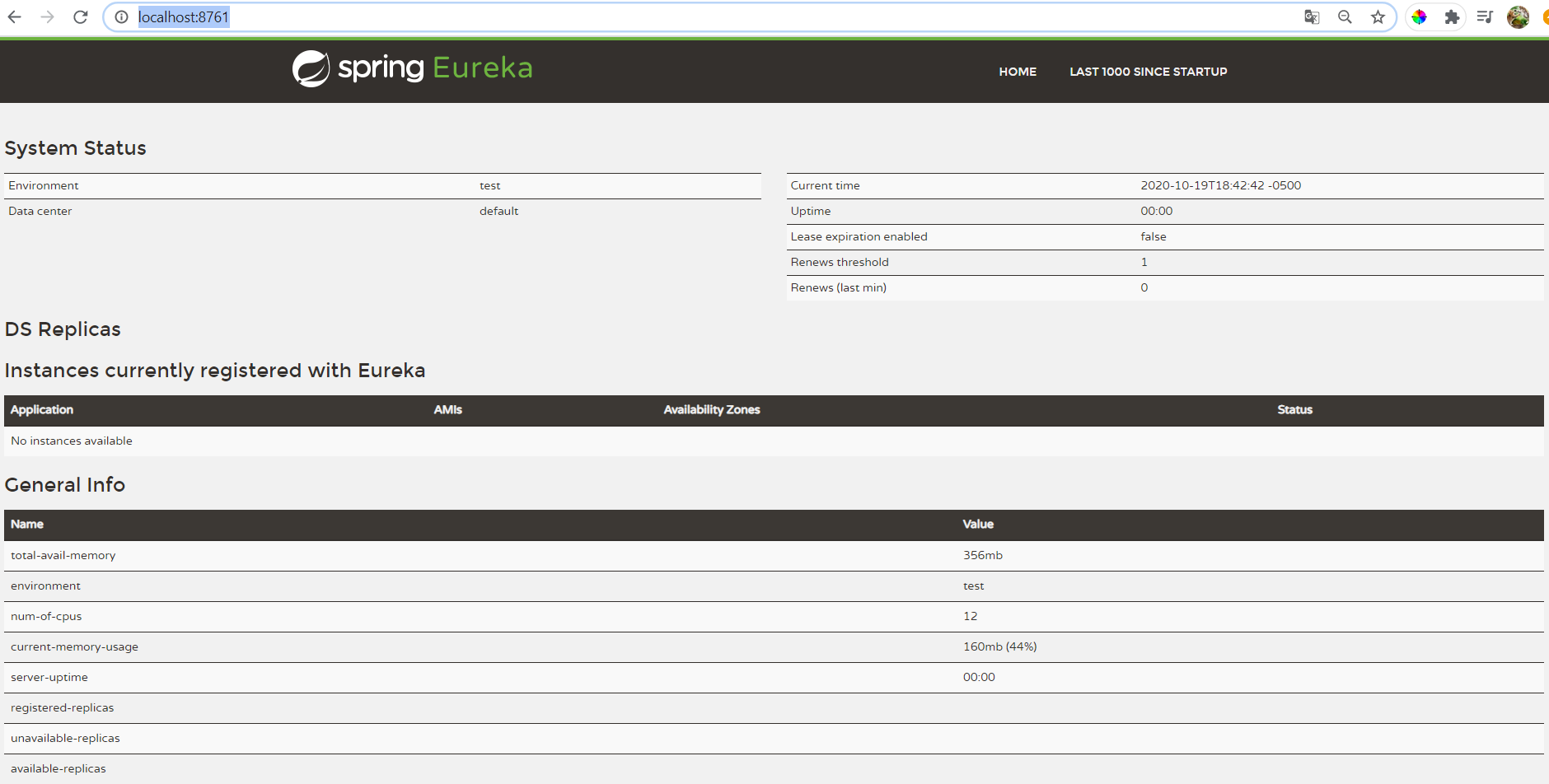
}

Con la anotación **@EnableEurekaServer** se habilita el servidor de nombrado.

* + Verificar que el servidor de nombrado eureka **netflix-eureka-naming-server** se ha generado correctamente y esta respondiendo bien.

Ir a la siguiente URL: <http://localhost:8761/>

Como se puede observar en la imagen se abajo, ya se tiene habilitado el servidor de nombrado eureka. (Por el momento no tenemos nada registrado en el servidor de nombrado eureka)



* **Conexión de las aplicaciones currency-exchange-service, currency-conversion-service al servidor de nombrado eureka netflix-eureka-naming-server.**



Para ello, equerimos de lo siguiente:

* + Conexión de la aplicación **currency-conversion-service** alservidor de nombrado eureka **netflix-eureka-naming-server**
    - Agregar las siguientes líneas en el pom.xml de la aplicación **currency-conversion-service**

<!-- Para la conexión con el servidor de nombrado eureka -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

* + - Para que nuestra aplicación sea reconocida por el servidor de nombrado eureka, debemos poner la siguiente anotación en la clase principal de nuestra aplicación.

@SpringBootApplication

@EnableFeignClients("com.in28minutes.microservices.currencyconversionservice")

@EnableDiscoveryClient

**public** **class** CurrencyConversionServiceApplication {

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

SpringApplication.*run*(CurrencyConversionServiceApplication.**class**, args);

}

}

Con la anotación **@EnableDiscoveryClient** habilitamos la aplicación **currency-conversion-service** para que el servidor de nombrado eureka lo detecte.

* + - Dar de alta la URL del servidor de nombrado eureka en el archivo **application.properties** de la aplicación **currency-conversion-service**

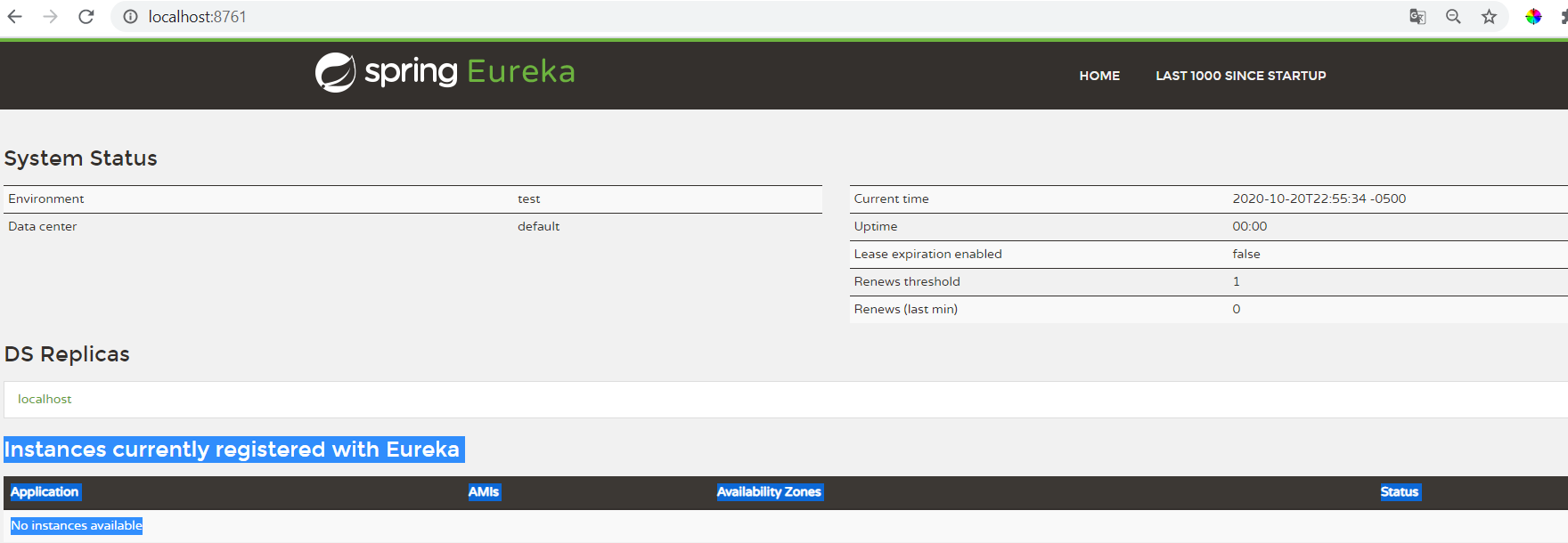
#Define la URL del servidor de nombrado eureka

eureka.client.service-url.default-zone=http://localhost:8761/eureka

* + - Verificar que la aplicación **currency-conversion-service** este dado de alta en el servidor de nombrado eureka **netflix-eureka-naming-server.**

Demos de baja la aplicación **currency-conversion-service** y consulte el servidor de nombrado eureka, como puede ver, no se enuentra registrada ninguna aplicación.

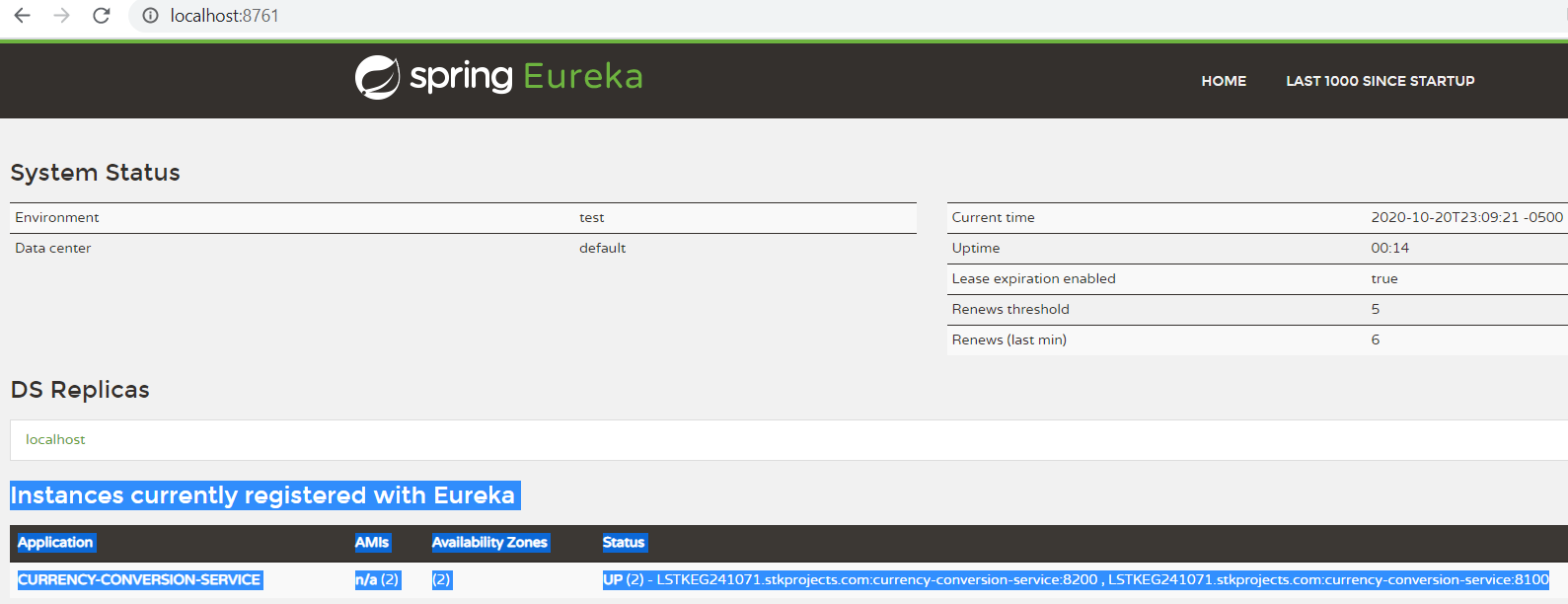
Ir a la URL: <http://localhost:8761/>



Ahora levante la aplicación **currency-conversion-service** y vuelva a consultar el servidor de nombrado eureka, como puede ver, ya se enuentra registrada la aplicación **currency-conversion-service**.



Y que pasa si tenemos más de una instancia disponible de la aplicación **currency-conversion-service** (en los puertos 8100 y 8200)



* + Conexión de la aplicación **currency-exchange-service** alservidor de nombrado eureka **netflix-eureka-naming-server**
    - Agregar las siguientes líneas en el pom.xml de la aplicación **currency- exchange -service**

<!-- Para la conexión con el servidor de nombrado eureka -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

</dependency>

* + - Para que nuestra aplicación sea reconocida por el servidor de nombrado eureka, debemos poner la siguiente anotación en la clase principal de nuestra aplicación.

@SpringBootApplication

@EnableDiscoveryClient

**public** **class** CurrencyExchangeServiceApplication {

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

SpringApplication.*run*(CurrencyExchangeServiceApplication.**class**, args);

}

}

Con la anotación **@EnableDiscoveryClient** habilitamos la aplicación **currency- exchange -service** para que el servidor de nombrado eureka lo detecte.

* + - Dar de alta la URL del servidor de nombrado eureka en el archivo **application.properties** de la aplicación **currency- exchange-service**

#Define la URL del servidor de nombrado eureka

eureka.client.service-url.default-zone=http://localhost:8761/eureka

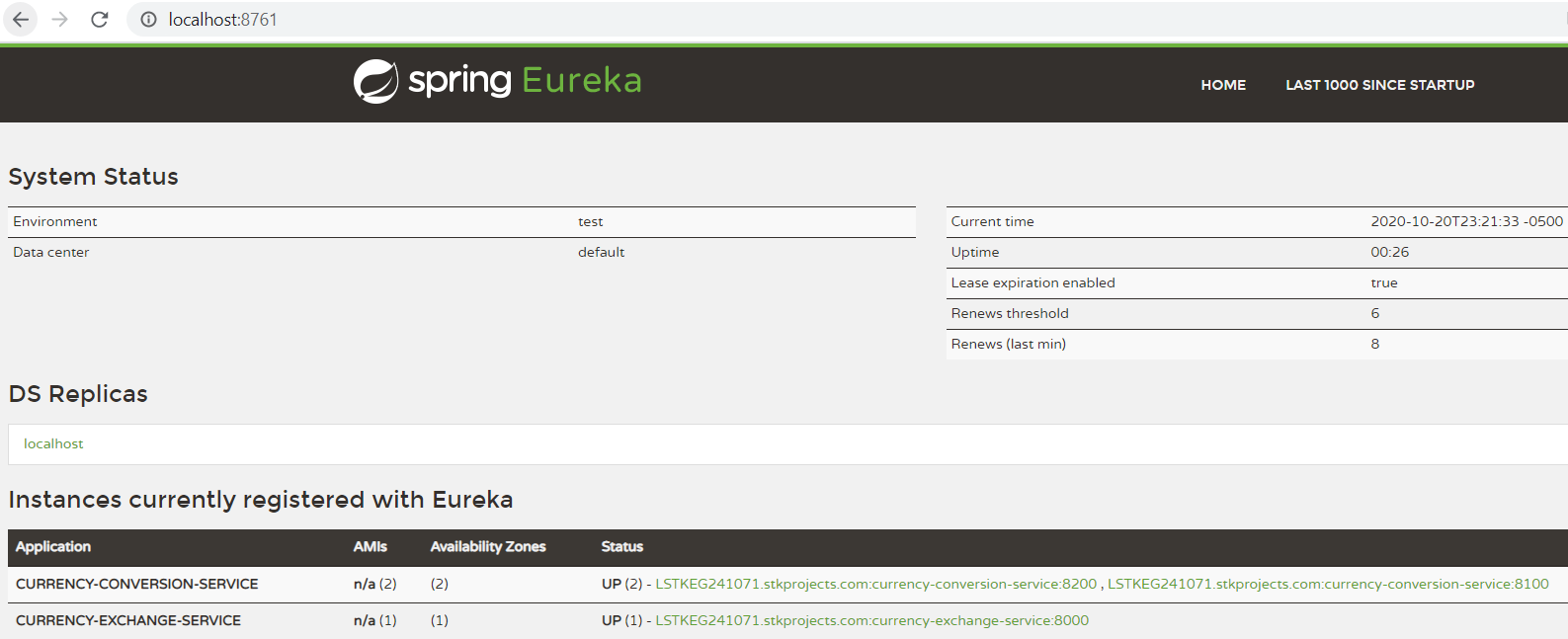
* + - Verificar que la aplicación **currency- exchange-service** este dado de alta en el servidor de nombrado eureka **netflix-eureka-naming-server.**

Demos de baja la aplicación **currency- exchange-service** y consulte el servidor de nombrado eureka, como puede ver, no se enuentra registrada la aplicación **currency- exchange-service**.

Ir a la URL: <http://localhost:8761/>



Ahora levante la aplicación **currency- exchange-service** y vuelva a consultar el servidor de nombrado eureka, como puede ver, ya se enuentra registrada la aplicación **currency- exchange-service**.



* + Si queremos ver el response en formato Json, agregar las siguientes restricciones en el **pom.xml** de las aplicaciones **currency-conversion-service** y **currency-exchange-service**

Modificar la dependencia para realizar la conexión con el servidor de nombrado eureka:

<!-- Para la conexión con el servidor de nombrado eureka -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

<!-- Para poder obtener response en formato Json -->

<exclusions>

<exclusion>

<groupId>com.fasterxml.jackson.dataformat</groupId>

<artifactId>jackson-dataformat-xml</artifactId>

</exclusion>

</exclusions>

</dependency>

* **Interacción de los micro servicios a través del servidor de nombrado eureka.**

El objetivo de este paso es facilitar la configuración requerida para hacer que los micro servicios interactúen entre si. Con esta opción ya no se hacen a mano varias de las configuraciones que realizamos en pasos anteriores. Configuraciones como por ejemplo:

//@FeignClient(name="currency-exchange-service", url="localhost:8000")

@FeignClient(name="currency-exchange-service")

@RibbonClient(name="currency-exchange-service")

**public** **interface** CurrencyExchangeServiceProxy {

@GetMapping("/currency-exchange/from/{from}/to/{to}")

**public** CurrencyConversionBean retrieveValue(@PathVariable("from") String from, @PathVariable("to") String to);

}

**Para poder utilizar la información del servidor de nombrado eureka, realizar lo siguiente:**

* + En los archivos **application.properties** de las aplicaciones **currency-conversion-service** y **currency-exchange-service** comentar o aliminar la constante en la que se indico la lista de isntancias de cada uno de los servicios

#Define las instancias del aplicativo currency-exchange-service sobre los que se genera el balanceo de carga.

#currency-exchange-service.ribbon.listOfServers=http://localhost:8000,http://localhost:8001

* + Este paso solo aplica si se generaron proxy’s para realizar conexiones con los micro servicios.

Verificar que en los proxy no se tenga habilitada la siguiente anotación:

//@FeignClient(name="currency-exchange-service", url="localhost:8000")

@FeignClient(name="currency-exchange-service")

@RibbonClient(name="currency-exchange-service")

**public** **interface** CurrencyExchangeServiceProxy {

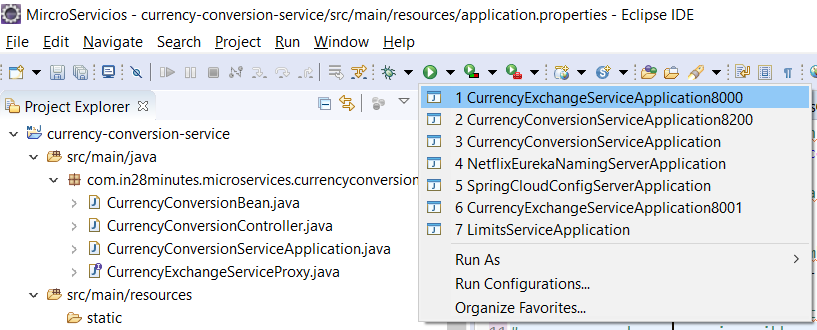
@GetMapping("/currency-exchange/from/{from}/to/{to}")

**public** CurrencyConversionBean retrieveValue(@PathVariable("from") String from, @PathVariable("to") String to);

}

La anotación **@FeignClient** es la que debe estar comentada.

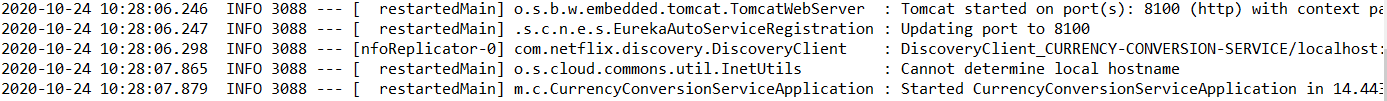
* + Dar de baja todas la instancias de las aplicaciones **currency-conversion-service** y **currency-exchange-service** y después solo levantar las instancias disponibles en los puertos 8100, 8000 respectivamente.



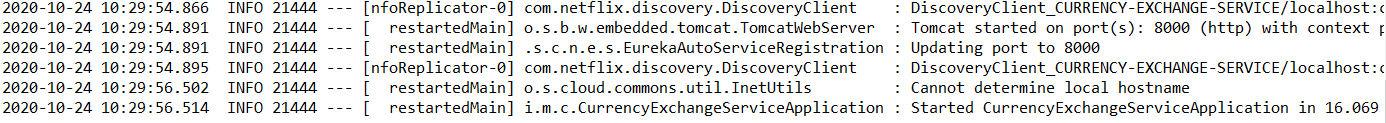
Levantando las aplicaciones **currency-conversion-service** y **currency-exchange-service** para las instancias de los puertos 8100 y 8000 respectivamente.

**Para ello, primero debemos dar de alta el servidor de nombrado eureka. (Importante recordar que el servidor de nombrado eureka, necesito un poco de tiempo para poder levantar en su totalidad, si se utilizan antes de que cargue completamente, puede mandar algunos mendajes de advertencia en pantalla)**

Levantando la aplicación **currency-conversion-service:**



Levantando la aplicación **currency-exchange-service:**



* + Verificar que en el servidor de nombrado eureka **netflix-eureka-naming-server** se han dado de alta las aplicaciones **currency-conversion-service** y **currency-exchange-service** que están disponibles en los puertos 8100, 8000 respectivamente.

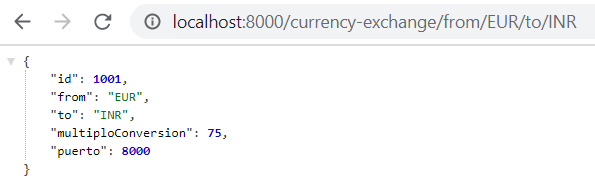
Ir a la URL: <http://localhost:8761/>

Como se puede apreciar en la imagen de abajo, ya se han dado de alta las instancias de las aplicaciones **currency-conversion-service** y **currency-exchange-service.** (Si se llegara a mandar algún mensaje de error al momento de levantar el servidor de nombrado eureka, basta con volver a reiniciarlo para solucionar el problema)



* + Verificar que las aplicaciones **currency-conversion-service**, **currency-exchange-service** están funcionando correctamente.

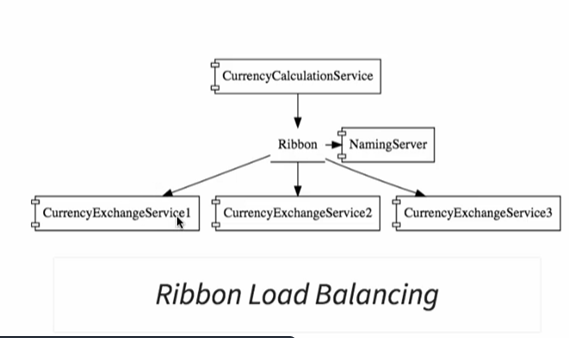
Como podemos ver en la imagen de abajo, el micro servicio **currency-exchange-service** está disponible en el puerto 8000.



Como podemos ver en la imagen de abajo, el micro servicio **currency-conversion-service** está disponible en el puerto 8100.



* + **Entonces, hasta el momento, la arquitecura que hemos adoptado, es la siguiente:**



El micro servicio **currency-conversion-service** se conecta a Ribbon (Balanceador de carga), luego, Ribbon interactua con el servidor de nombrado eureka **netflix-eureka-naming-server** (De donde se obtienen las URL de los servicios)para verificar de las instancias disponibles de la aplicación **currency-exchange-service** que pueden atender los request que se generan desde la aplicación **currency-conversion-service.** (Importate recordar que Ribbon deberá hacer el balanceo de carga)

Enntonces, implementamos exitosamente el balanceador de carga (Ribbon), el escalamiento dinámico hacia arriba y hacia abajo, es decir se agregaron y quitaron instancias de los micro servicios de forma dinámica, esto se logra dando de alta las instancias de los micro servicion en el servidor de nombrado eureka.

* **Implementación de micro servicios mediante la implementación de API Gateway. (Puertas de enlace)**

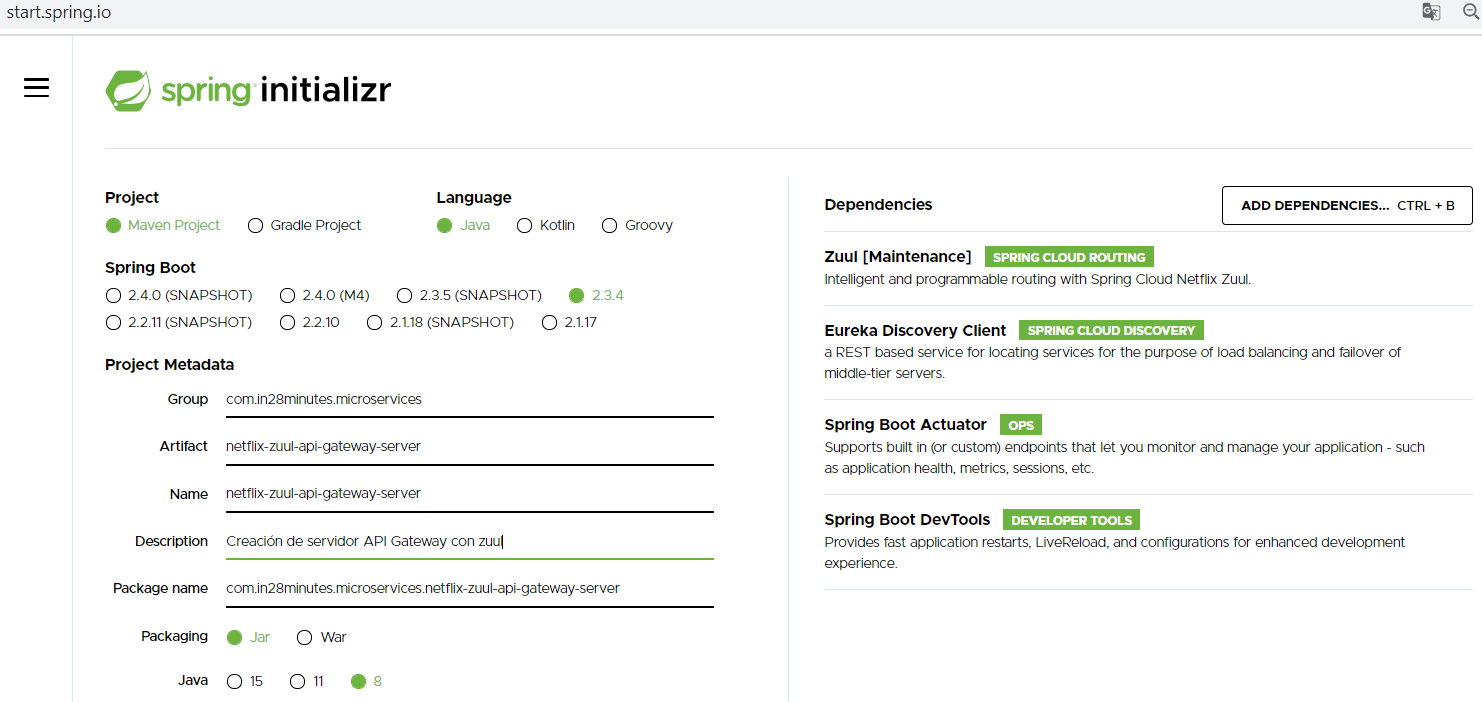
El objetivo es garntizar que se cumpla lo siguinet:

* + - Autenticación, autorización, y seguridad.
    - Definir límites.
    - Tolerancia a fallas.
    - Agregado del servicio.

Para ello, requerimos de lo siguiente:

* + Configuración de un servidor **API Gateway** con **Zuul**.

Generar un proyecto de spring boot con la siguiente configuración:



* + Habilitar el **servidor zuul** y el **servidor de nombrado eureka**.

Para ello, debemos poner las siguientes anotaciones a la clase de arranque de nuestra aplicación.

@EnableZuulProxy

@EnableDiscoveryClient

@SpringBootApplication

**public** **class** NetflixZuulApiGatewayServerApplication {

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

SpringApplication.*run*(NetflixZuulApiGatewayServerApplication.**class**, args);

}

}

Con la anotación **@EnableZuulProxy** se habilita el servidor API Gateway. (Puertas de enlace)

Con la anotación **@EnableDiscoveryClient** se habilita el servidor de nombrado eureka.

* + Configuració del nombre de la aplicación, puerto que debe usar la aplicación, y configuración del servidor de nombrado eureka.

En el archivo **application.properties** agregar las siguientes constantes:

#Define el nombre de la aplicación

spring.application.name=netflix-zuul-api-gateway-server

#Define el puerto que debe usar la aplicación

server.port=8765

#Define la URL del servidor de nombrado eureka

eureka.client.service-url.default-zone=http://localhost:8761/eureka

* + Implementación de funcionalidades en el servidor API Gateway. En este caso, vamos a registrar cualquier solicitud que llegue al servidor API Gateway.

Para ello, debemos generar lo siguiente:

* + - Creación de la clase **zuulLoginFilter.java** con la que se manejan varios filtros.

@Component

**public** **class** ZuulLogginFilter **extends** ZuulFilter {

@Override

**public** **boolean** shouldFilter() {

**return** **true**;

}

@Override

**public** Object run() **throws** ZuulException {

**return** **null**;

}

@Override

**public** String filterType() {

**return** "pre";

}

@Override

**public** **int** filterOrder() {

**return** 1;

}

}

Con la anotación **@Component** le indicamos a spring que esta clase la debe contemplar para su gestión.

Como podemos observar, de inicio tenemos 4 filtros (shouldFilter, run, filterType, filterOrder).

* + - * **shouldFilter** se indica que se activa el filtrado. (**return** **true**)
      * **run** es un método donde se definen las acciones a realizar dependiendo de las configuraciones de los filtros.
      * **filterType** se definen los tipos de filtro “pre”, “post”, “error”, etc... (“Pre-routing filtering”, “Post-routing filtering”).

Por ejemplo:

Se hace el filtrado de todos los request con “error”

Se hace el filtrado de todos los request con “pre” 🡪 Se interceptan todos los request antes de ejecutarlas.

Se hace el filtrado de todos los request con “post” 🡪 Se interceptan todos los request después de ejecutarlas.

Para efectos de nuestro ejemplo, estamos habilitando los filtros "pre" (es decir intercepta el request antes de ejecutarla)

* + - * **filterOrder** se define el orden de los filtros.

Para efectos de nuestro ejemplo, estamos definiendo un orden 1

**Por el momento nos enfocaremos en el método Run,** Es decir, queremos definir lo que de debe realizar en cada uno de los request solicitados. Por el momento, únicamente mandaremos a imprimir la información de los request. Por lo que el método **run** queda de la siguiente manera:

@Override

**public** Object run() **throws** ZuulException {

HttpServletRequest request = RequestContext.*getCurrentContext*().getRequest();

logger.debug("request --> {} request uri --> {}", request, request.getRequestURI());

**return** **null**;

}

Con la implementación del método **run** hemos definido un filtro básico.

* + Verificar que el servidor API Gateway esta en funcionamiento.

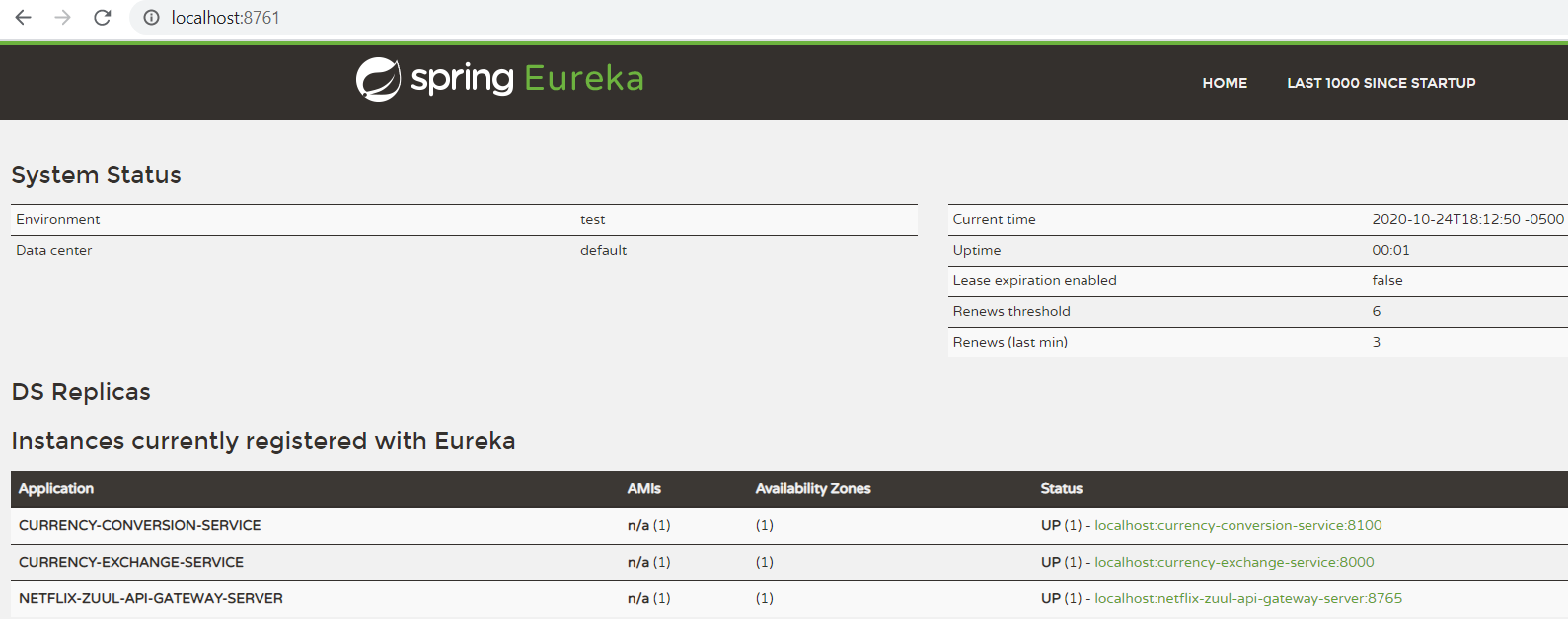
Para ello, el servidor de nombrado eureka **netflix-eureka-naming-server** y al menos una instancia de las aplicaciones **currency-conversion-service**, **currency-exchange-service** debe estar arriba.

Por consiguiente debemos levantar las siguientes aplicaciones en el orden en que se listan:

* + - * **netflix-eureka-naming-server**
      * **currency-exchange-service, currency-conversion-service**
      * **netflix-zuul-api-gateway-server**

Consultar si el servidor API Gateway ya se ha dado de alta en el servidor de nombrado eureka: ir a la URL <http://localhost:8761/>

Como se puede apreciar en la imagen de abajo, ya se ha dado de alta el servidor API Gateway **netflix-zuul-api-gateway-server** en el servidor de nombrado eureka **netflix-eureka-naming-server**



* + Consumir los servicios de la aplicación **currency-exchange-service** a través del servidor API Gateway **netflix-zuul-api-gateway-server**

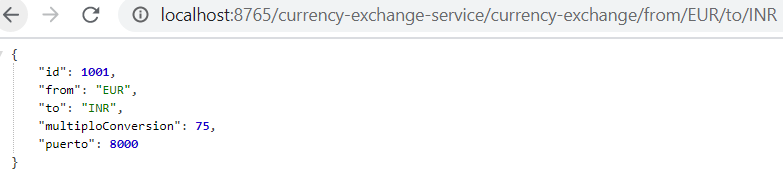
Para poder lograr esto, debemos mapear la url del servicio de **currency-exchange-service** a la IP y puerto del servidor API Gateway **netflix-zuul-api-gateway-server.**

URL del servicio se la aplicación **currency-exchange-service:** [**http://localhost:8000/currency-exchange/from/EUR/to/INR**](http://localhost:8000/currency-exchange/from/EUR/to/INR)

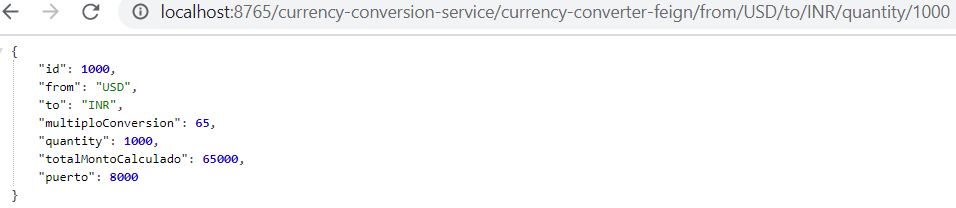
Para poder consumir el servicio a través del servidor API Gateway debemos armar la siguiente URL:

[**http://localhost:8765/{application-name}/{uri}**](http://localhost:8765/%7bapplication-name%7d/%7buri%7d)

[**http://localhost:8765/currency-exchange-service/currency-exchange/from/EUR/to/INR**](http://localhost:8765/currency-exchange-service/currency-exchange/from/EUR/to/INR%20)

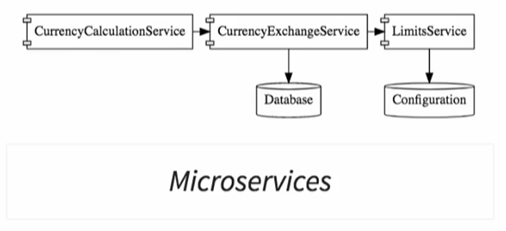


**http://localhost:8765/currency-conversion-service/currency-converter-feign/from/USD/to/INR/quantity/1000**



Conclusión: Se han podido consumir los servicios de los micro servicios **currency-conversion-service**, **currency-exchange-service** a través del servidor API Gateway **netflix-zuul-api-gateway-server** mediante los filtros definidos en la clase **ZuulLogginFilter.java** del servidor API Gateway.

* + Consumir servicios de las aplicaciones **currency-conversion-service**, **currency-exchange-service** a través del servidor API Gateway **netflix-zuul-api-gateway-server** sin tener que armar a mano las URL.



En la imagen e muestra la secuencia de invocacines de los dervicios, hasta ahora el servicio **CurrencyCalculationService** manda llamar al servicio **CurrencyExchangeService** de forma directa, lo que ahora se va a implementar es que el servicio **CurrencyCalculationService** mande llamar al aservicio **CurrencyExchangeService** a través del servidor API Gateway **netflix-zuul-api-gateway-server**

Para lograr el objetivo, debemos realizar las siguientes configuraciones:

* + - Dar de alta en el servidor de nombrado eureka **netflix-eureka-naming-server** las aplicaciones **currency-exchange-service, currency-conversion-service.**

**NOTA:**

En nuestro caso ya lo tenemos configurado, pero si no fuera así, se tendría que realizar dicha configuración.

* + - Configurar la interface CurrencyExchangeServiceProxy.java (Ubicado en la aplicación **currency-conversion-service**) para que se conecte al servidor API Gateway **netflix-zuul-api-gateway-server** y para que sepa interpretar la URL del servicio solicitado.

//@FeignClient(name="currency-exchange-service", url="localhost:8000")

//@FeignClient(name="currency-exchange-service")

@FeignClient(name="netflix-zuul-api-gateway-server")

@RibbonClient(name="currency-exchange-service")

**public** **interface** CurrencyExchangeServiceProxy {

//@GetMapping("/currency-exchange/from/{from}/to/{to}")

@GetMapping("/currency-exchange-service/currency-exchange/from/{from}/to/{to}")

**public** CurrencyConversionBean retrieveValue(@PathVariable("from") String from, @PathVariable("to") String to);

}

Con la anotación **@FeignClient**(name="netflix-zuul-api-gateway-server") estamos indicando que todos los servicios definidos en la interface deben pasar por el servidor API Gateway **netflix-zuul-api-gateway-server**

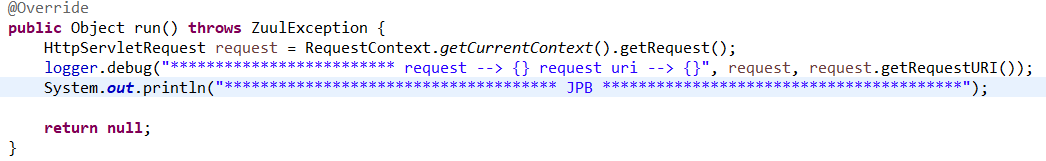
La anotación del GetMapping ahora debe ser @GetMapping("/**currency-exchange-service**/currency-exchange/from/{from}/to/{to}") a la ruta original, se le agrega al inicio e nombre de la aplicación, recordemos que cuando armamos a mano las URL [**/{application-name}/{uri}**](http://localhost:8765/%7bapplication-name%7d/%7buri%7d)

Para verificar que el micro servicio **CurrencyCalculationService** manda llamar al micro servicio **CurrencyExchangeService** a través del servidor API Gateway **netflix-zuul-api-gateway-server.**

Ir a la URL: <http://localhost:8100/currency-converter-feign/from/USD/to/INR/quantity/1000>

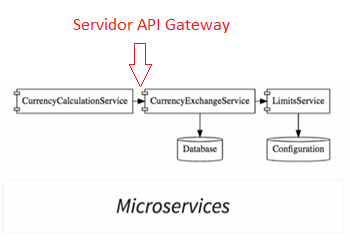


Y en la consola de **netflix-zuul-api-gateway-server** se debe desplegar el mensaje que pusimos en el metodo **run()**

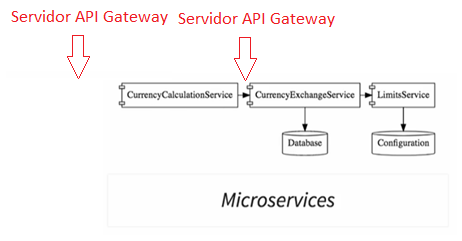




**Conclusión: Los servicios ya los esta interceptando el servidor API Gateway**

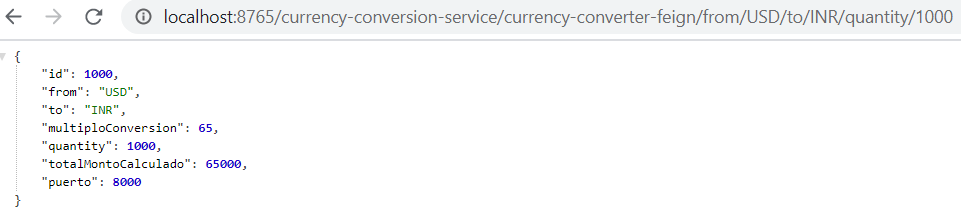


Ahora queremos que desde la primera invocación se genere a través del servidor API Gateway, es decir:

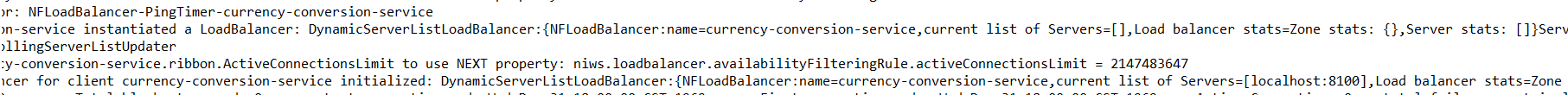


Para ello, tendríamos que invocar el servicio con la siguiente URL:

[http://localhost:8765/{service-name}/{URI}](http://localhost:8765/%7bservice-name%7d/%7bURI%7d) 🡪 <http://localhost:8765/currency-conversion-service/currency-converter-feign/from/USD/to/INR/quantity/1000>



Y en la consola del servidor API Gateway se debe ver un mensaje similar al asiguiente:



Listo, nuestro servicio es interceptado por el servior API Gateway

* **LOGS - Rastreo distribuido mediante la implementación de Sleuth.**

**NOTA:** Una desventaja del método que veremos, es que tenemos que andar rastreando los log de las n aplicaciones involucradas. Más adelante veremos como optimizar este inconveniente.

* + Asignación de identificadores únicos a los procesos mediante **sleuth.**

Agregar las siguientes configuraciones en los siguientes proyectos:

* + - Servidor API Gateway **netflix-zuul-api-gateway-server**
      * En el archivo pom.xmlagregar la siguiente dependencia

<!-- Para la asignación de identificadores únicos a los procesos (ID's) -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-sleuth</artifactId>

</dependency>

* + - * En la clase principal del servidor API Gateway agregar el siguiente bean.

@EnableZuulProxy

@EnableDiscoveryClient

@SpringBootApplication

**public** **class** NetflixZuulApiGatewayServerApplication {

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

SpringApplication.*run*(NetflixZuulApiGatewayServerApplication.**class**, args);

}

@Bean

**public** Sampler defaultSampler() {

**return** Sampler.***ALWAYS\_SAMPLE***;

}

}

Con el bean **defaultSampler()** indicamos que se van a rastrear todas las solicitudes/request.

* + - * En la clase ZuulLogginFilter.java en el método **run()** poner algunos logs para poder ver el ID asignado a los procesos y lo más importante, poder dar seguimiento a los request con el ID asignado y con la ayuda de los logs.
    - En la aplicación del micro servicio **currency-conversion-service.**
      * En el archivo pom.xml agregar las siguientes dependencias.

<!-- Para la asignación de identificadores únicos a los procesos (ID's) -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-sleuth</artifactId>

</dependency>

* + - * En la clase principal del micro servicio **currency-conversion-service** agregar el siguiente bean.

@SpringBootApplication

@EnableFeignClients("com.in28minutes.microservices.currencyconversionservice")

@EnableDiscoveryClient

**public** **class** CurrencyConversionServiceApplication {

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

SpringApplication.*run*(CurrencyConversionServiceApplication.**class**, args);

}

@Bean

**public** Sampler defaultSampler() {

**return** Sampler.***ALWAYS\_SAMPLE***;

}

}

Con el bean **defaultSampler()** indicamos que se van a rastrear todas las solicitudes/request.

* + - * En el controller de la aplicación poner algunos logs para poder ver el ID asignado a los procesos y lo más importante, poder dar seguimiento a los request con el ID asignado y con la ayuda de los logs.

@GetMapping("/currency-converter-feign/from/{from}/to/{to}/quantity/{quantity}")

**public** CurrencyConversionBean convertCurrencyFeign(@PathVariable String from, @PathVariable String to, @PathVariable BigDecimal quantity) {

//A través del proxy consumimos el servicio de la aplicación currency-exchange-service.

CurrencyConversionBean response = proxy.retrieveValue(from, to);

logger.info("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Message From CurrencyConversionController ---->>> {}", response);

//Obtenemos la información que necesitamos.

**return** **new** CurrencyConversionBean(response.getId(), from, to, response.getMultiploConversion(), quantity, quantity.multiply(response.getMultiploConversion()), response.getPuerto());

}

**Estos logs los estaremos viendo en la consola del servidor API Gateway**

* + - En la aplicación del micro servicio **currency-exchange-service.**
      * En el archivo pom.xml agregar las siguientes dependencias.

<!-- Para la asignación de identificadores únicos a los procesos (ID's) -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-sleuth</artifactId>

</dependency>

* + - * En la clase principal del micro servicio **currency-conversion-service** agregar el siguiente bean.

@SpringBootApplication

@EnableDiscoveryClient

**public** **class** CurrencyExchangeServiceApplication {

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

SpringApplication.*run*(CurrencyExchangeServiceApplication.**class**, args);

}

@Bean

**public** Sampler defaultSampler() {

**return** Sampler.***ALWAYS\_SAMPLE***;

}

}

Con el bean **defaultSampler()** indicamos que se van a rastrear todas las solicitudes/request.

* + - * En el controller de la aplicación poner algunos logs para poder ver el ID asignado a los procesos y lo más importante, poder dar seguimiento a los request con el ID asignado y con la ayuda de los logs.

@GetMapping("/currency-exchange/from/{from}/to/{to}")

**public** ExchangeValue retrieveValue(@PathVariable String from, @PathVariable String to) {

ExchangeValue exchangeValue = exchangeValueRepository.findByFromAndTo(from, to);

**if**( exchangeValue != **null** )

exchangeValue.setPuerto(Integer.*parseInt*(environment.getProperty("local.server.port")));

logger.info("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Message From CurrencyExchangeController ---->>> {}", exchangeValue);

**return** exchangeValue;

}

**Estos logs los estaremos viendo en la consola del servidor API Gateway**

* + Verificar que las aplicaciones siguen respondiendo.

Levantar las siguientes aplicacines en el orden que se listan a continuación (Recordemos que el server API Gateway es un poco lento para levantar, por lo que si le manda un erro, vuelva a intentar):

* + - * **netflix-eureka-naming-server**
      * **currency-exchange-service, currency-conversion-service**
      * **netflix-zuul-api-gateway-server**

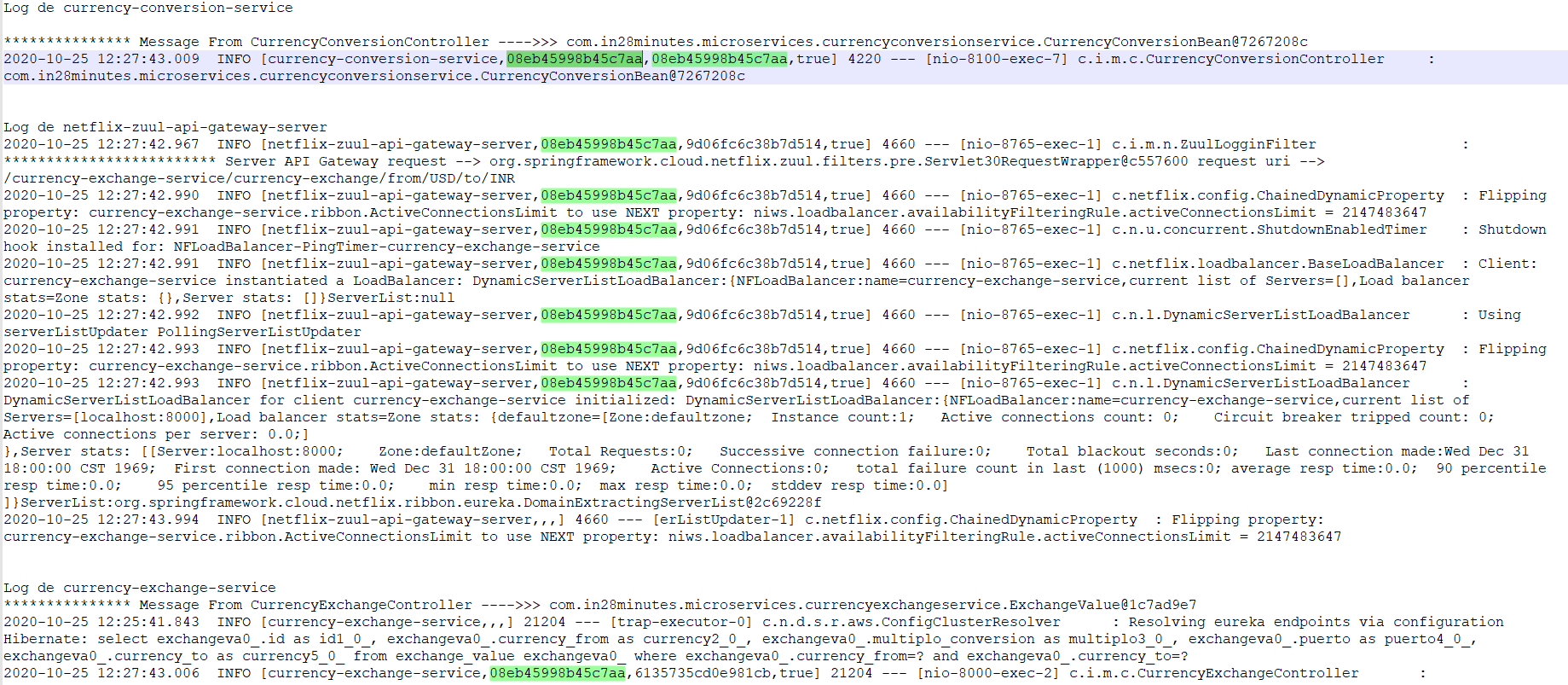
Consultar el siguiente servicio, ir a la URL: <http://localhost:8100/currency-converter-feign/from/USD/to/INR/quantity/1000>



**Importante:**

Ver los logs de las aplicaciones involucradas en nuestro request:

En el log de la aplicación **currency-conversion-service** vemos como se inicia nuestro request con el ID **08eb45998b45c7aa**, mismo que podemos ver en los logs de las aplicaciones **currency-exchange-service, netflix-zuul-api-gateway-server.** (Recordemos que la comunicación entre los micro servicios **currency-conversion-service, currency-exchange-service,** esta interceptada por el servidor API Gateway **netflix-zuul-api-gateway-server**)



Listo, se concluye que la asignación de ID a los reques se ha generado de forma exitosa y que con los ID’s asignados se monitorea el estatus de los request a través de las aplicaciones involucradas.

Recordemos que lo importante de esta implementación fue la asignación de ID’s para monitorear el estatus de nuestros request.

* **LOGS - Rastreo distribuido mediante la implementación de Sleuth, RabbitMQ, y zipkin (Centralizar la información de los micro servicios para cuando se tenga la necesidad de consultarla)**

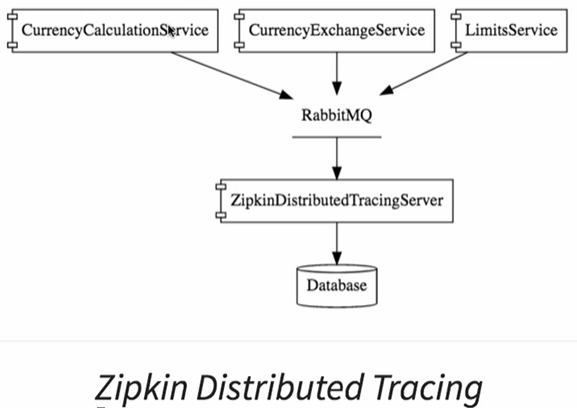
**NOTA:** con esta implementación se resuelve el tema de andar monitoreando los logs en cada una de las consolas de las aplicaciones involucradas en el request.

Ahora lo que se hace:

* + Asignar ID’s a los request con la ayuda de **Sleuth**.
  + Centralizar los logs de los request en **RabbitMQ**.
  + Monitoreo de estatus de los request solicitados mediante **zipkin**. (Monitorear el estatus de un request específico en RabbitMQ donde se centraliza la información)

**zipkin** por lo general requiere conectarse a una base de datos, por lo que la conectaremos a la base de datos H2. (Base de datos en memoria)

**Es decir, los mensajes de los logs se encolan en RabbitMQ y mediante zipkin los consumimos utilizando los ID’s asignados con Sleuth.**



* + Para la instalación de RabbitMQ necesitamos de los siguiente:
    - Referencias bibliográficas:

Al momento de tomar el curso se nos habilito la siguiente liga: <https://github.com/in28minutes/spring-microservices>

En la sección **Installing Tools** buscar la sección **Installing** **RabbitMQ** en la que se indican referencias para la instalación de RabbitMQ

* <https://www.rabbitmq.com/install-windows.html>
* <https://www.rabbitmq.com/which-erlang.html>
* <http://www.erlang.org/downloads>
* Video - <https://www.youtube.com/watch?v=gKzKUmtOwR4>
  + - Instalación de **RabbitMQ**.

**NOTA:**

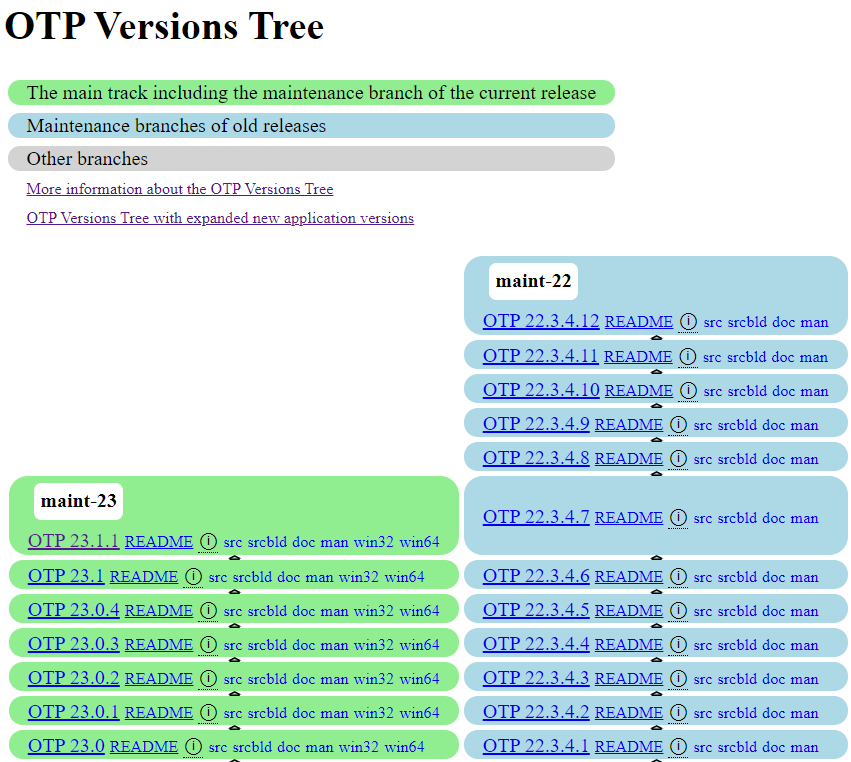
Tenere cuidado de verificar que las versiones de **RabbitMQ** y **erlang** que se van a instalar sean compatibles.

Las instrucciones de instalación se encuentran en <https://www.rabbitmq.com/install-windows.html> buscar la sección **Install: Windows (manual)**

* + - * Instalación de **erlang**

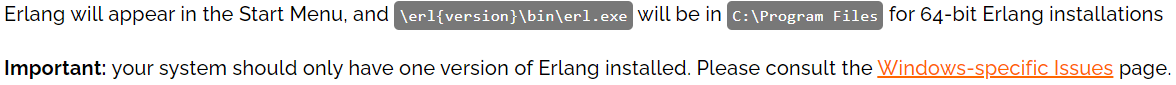
El software se puede descargar en el siguiente sitio: <https://erlang.org/download/otp_versions_tree.html>

Vamos a descargar la versión **OPT 23.1.1** Dar clic en el link con la leyenda **win64**

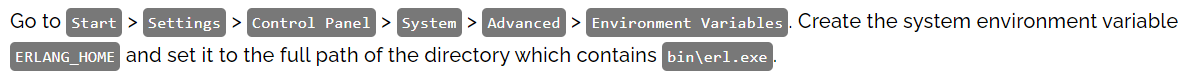


**El programa de debe instalar en:**

El programa **erlang** se debe instalar en



Dar de alta la varible de entorno ERLANG\_HOMEy asignarle la ruta absoluta del directorio que contiene el bin\erl.exe



* + - * Instalación de **RabbitMQ**

La versión que instalaremos será la **3.8.9**

Descargar la versión 3.8.9 y descomprimirla en



* + - * Sincronización de las **cookies de erlang**



(Ir a la siguiente liga para descargar el software <http://www.erlang.org/downloads>)

* **Tolerancia a fallas, seguridad, implementación de API Gateway Zuul (Que funciona como un interceptor)**
* **Asignación de ID a cada uno de los procesos.**
* **Zipkin – es un sistema de rastreo distribuido.**
* **RabbitMQ – Para almacenar el registro de todos los servicios.**

Diapositiva 106 – Apuntes

Diapositiva 105 -- Escuchadas