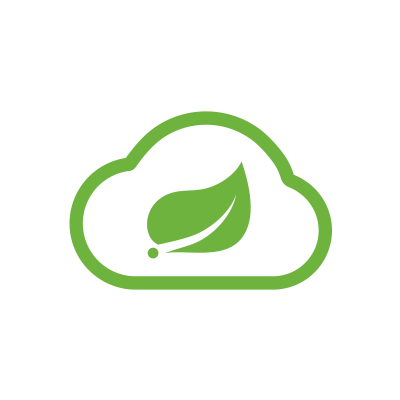
# Micro Services Spring Cloud

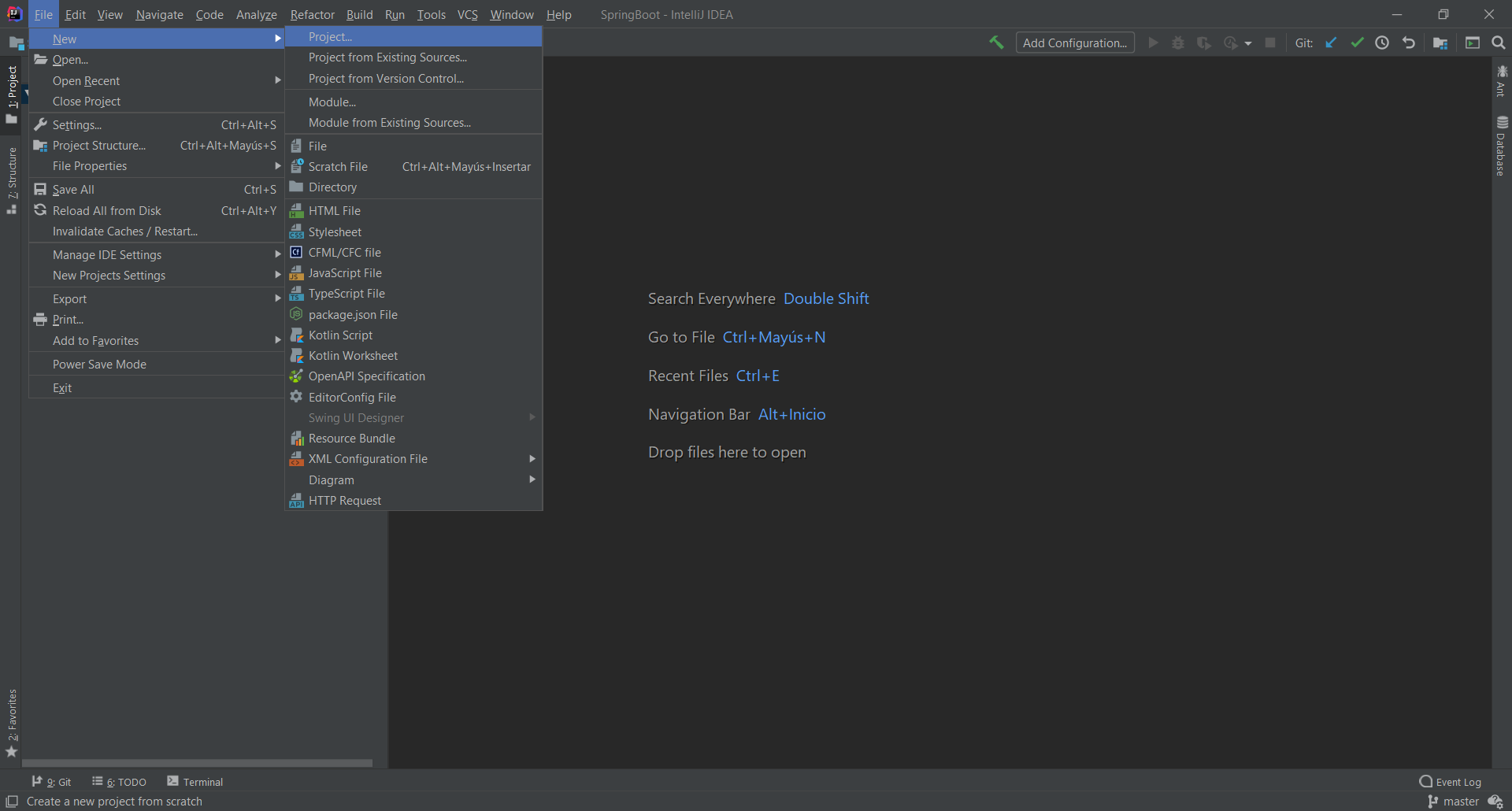


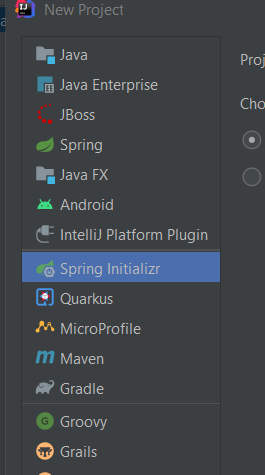
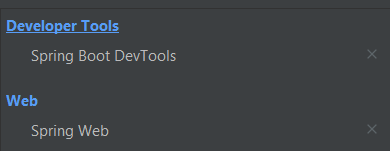
# Getting Started

There are many ways to start a project in spring, one is with initializr of <https://start.spring.io/>

Where you select your dependencies,

Another way its with the IDE, in this case Intellij IDE



 🡺Dependencies 

Then click on next 🡺 change name, click on next 🡺 select the dependencies. Click on next and finish🡺 Ready.

## @Rest Controller:

Documentation: <https://spring.io/guides/gs/rest-service/>

In Spring’s approach to building RESTful web services, HTTP requests are handled by a controller. These components are identified by the [@RestController](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RestController.html) annotation,

The Rest controller will be the one that handle your HTTP request and the class can support multiple annotations in order to create a CRUD.

The annotation [@RestController](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RestController.html) will allow the class to receive HTTP request but has to have a annotations on a function to do a specific function:

@RequestMapping will be the one that put the endpoint to the controller http://localhost:8080/**users**

@GetMapping is the one that will handle the request GET (to obtain objects or a response)

@PostMapping is the one that will handle the request POST (to create a new object)

@PutMapping is the one that will handle the request PUT (to edit a object already created)

@DeleteMapping is the one that will handle the request Delete (to delete a object already created)

Example:

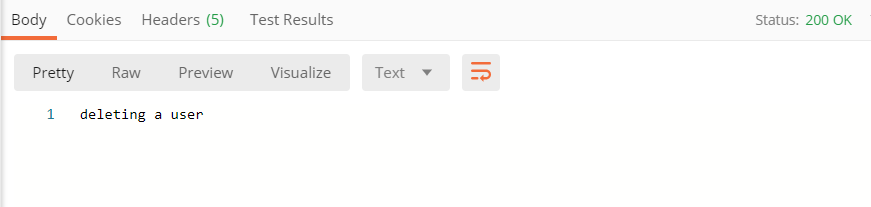
@RestController  
@RequestMapping("/users")  
public class Controller {  
  
 @GetMapping  
 public String getUser(){  
 return "getting user";  
 }  
  
 @PostMapping  
 public String postUser(){  
 return "user Created";  
 }  
  
 @PutMapping  
 public String editAUser(){  
 return "editing a user";  
 }  
  
 @DeleteMapping  
 public String deleteAUser(){  
 return "deleting a user";  
 }  
  
  
}

When we run the code, tomcat will deploy it to a local server by default the port is set in 8080

Using postman, we will send the request GET, PUT, POST, DELETE



Change the verb into , put, post ,delete



Status code:

<https://www.restapitutorial.com/httpstatuscodes.html>

Response of the http request

Rfttt

## @PathVariable

Path variable is an annotation that will allow take a value in the URL in order to user it in the function, for example when we want to obtain a specific user:

The annotation GetMapping will take the path (“/{**VARIABLE**}”) from the URL and will pass it to the function as an argument under the annotation @PathVariable dataType VARIABLE

@GetMapping(path = "/{VARIABLE}")  
public String getUser(@PathVariable String VARIABLE){  
 return "getting user: "+ VARIABLE;

@GetMapping(path = "/{userId}")  
public String getUser(@PathVariable String userId){  
 return "getting user: "+ userId;

PostMan:



## @RequestParam

Request params is an annotation that will take the parameters from the URL directly on the arguments of the function

Function(@RequestParams(value=”StringVariable”) dataType StringVariable, @RequestParams(value=”StringVariable2”) dataType StringVariable2)

@GetMapping()  
public String getUsers(@RequestParam(value = "page")int page,@RequestParam(value = "limit")int limit ){  
 return "getting users of page: "+page+" and limit of: "+ limit+" users";  
}



### Optional parameters:

To make optional parameters and void a error for Null Point Exception we will user **defaulValue=”value” AFTER value=”variable”**

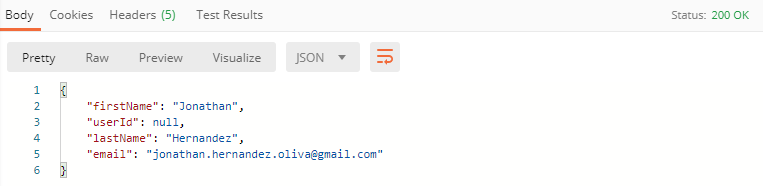
@GetMapping()  
public String getUsers(@RequestParam(value = "page", defaultValue = "1")int page,@RequestParam(value = "limit", defaultValue = "50")int limit ){  
 return "getting users of page: "+page+" and limit of: "+ limit+" users";  
}



## Returning a Java Object as Returning value.

In this example, the controller will return a object in JSON

@GetMapping(path = "/{userId}")  
public UserRest getUserModel(@PathVariable String userId){  
 UserRest user1 = new UserRest();  
 user1.setFirstName("Jonathan");  
 user1.setLastName("Hernandez");  
 user1.setEmail("jonathan.hernandez.oliva@gmail.com");  
 return user1;  
}



## Set Response Status Code:

As a best practice we need to return a reponse code status and a object, sometimes only status code its ok, for this reason we need to return a **new ResponseEntity<Class>(Object, HttpStatus.Status)**

Example:

@GetMapping(path = "/{userId}")  
public ResponseEntity<UserRest> getUserModel(@PathVariable String userId){  
 UserRest user1 = new UserRest();  
 user1.setFirstName("Jonathan");  
 user1.setLastName("Hernandez");  
 user1.setEmail("jonathan.hernandez.oliva@gmail.com");  
 return new ResponseEntity<UserRest>(user1,HttpStatus.*OK*);  
}

## Post Request:

For this example we will créate another Class called “UserPostRequest” and will be a copy of UserRest, it means the same parameters will be seted. In the function we will return the same as the provious but **in the parameters we will pass @RequestBody Class Object**

@PostMapping  
public ResponseEntity<UserRest> postUser(@RequestBody UserPostRequest userPost){  
 UserRest user1 = new UserRest();  
 user1.setFirstName(userPost.getFirstName());  
 user1.setLastName(userPost.getLastName());  
 user1.setEmail(userPost.getEmail());  
 return new ResponseEntity<UserRest>(user1,HttpStatus.*OK*);  
}

@RequestBody Class Object recieve the object and we will map the object with the UserRest in order to return UserRest as response

In Postman we will set it like this:

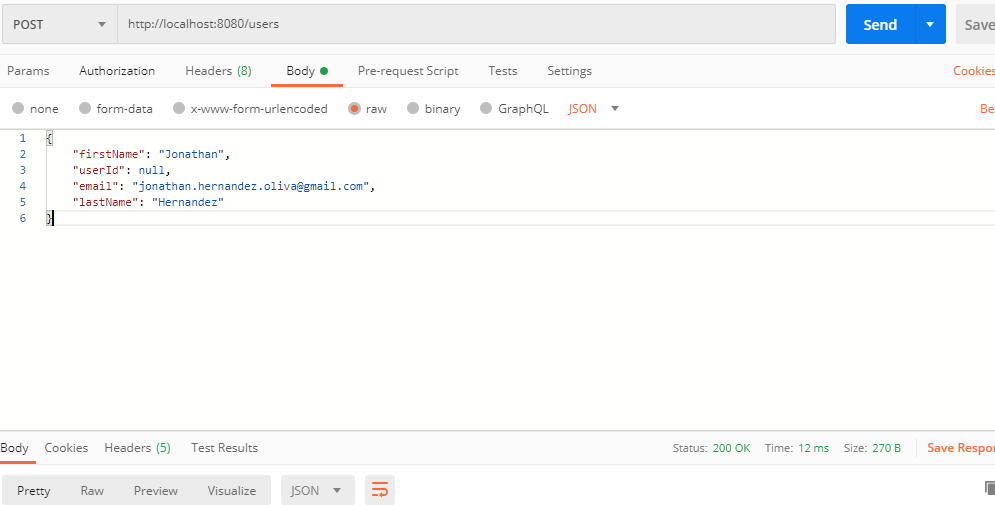
Post Verb

Body in JSON format

URL without params

Status code

Response



## Validating Http POST Request Body

In newer versions you may need this dependency to make validations with Javax

<dependencies>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-validation</artifactId>  
 </dependency>

We will use hibernate validater,

The annotation @valid from javax.valid will be the one that validate that all the requierements of the class are true, those validations comes from another annotations like @NotNull, @Email, @size etc.

Example:

@PostMapping  
public ResponseEntity<UserRest> postUser(@Valid @RequestBody UserPostRequest userPost){  
 UserRest user1 = new UserRest();  
 user1.setFirstName(userPost.getFirstName());  
 user1.setLastName(userPost.getLastName());  
 user1.setEmail(userPost.getEmail());  
 return new ResponseEntity<UserRest>(user1,HttpStatus.*OK*);  
}

The @Valid has to be placed in the beggining of the parameters of the function.

public class UserPostRequest {

@NotNull(message = "first name cannot be empty")  
 private String firstName;  
 @NotNull(message = "last name cannot be empty")  
 private String LastName;  
 @Email  
 @NotNull(message = "e-mail cannot be empty")  
 private String Email;  
 @NotNull(message = "password cannot be null")  
 @Size(max = 16, min = 6, message = "password must be greater than 6 characters and greater than 16 characters")  
 private String password;  
 //GETTERS AND SETTERS

## Store Users Temporary.

In order to user putMapping we need to persist the Object without using database,

For this we need to créate a HashMap object

Map<String,UserRest> users;

Map<String,UserRest> users;

Its a class that implements the class Map

This object accepts 2 objects, one will be the key(could be String) and the second will be the value, when we Access to the value, will be throught the key.

To créate a Map object we will use the method put()

**Example:**

Map<String ,UserRest> users;  
 //to post a new user  
  
 @PostMapping  
 public ResponseEntity<UserRest> postUser(@Valid @RequestBody UserPostRequest userPost){  
 UserRest user1 = new UserRest();  
 user1.setFirstName(userPost.getFirstName());  
 user1.setLastName(userPost.getLastName());  
 user1.setEmail(userPost.getEmail());  
 String userId = UUID.*randomUUID*().toString(); // create a UUID  
 user1.setUserId(userId);  
  
 if(users== null ){ //if users is null initialice the object  
 users= new HashMap<>();  
 users.put(userId,user1); // and create a new object  
 }  
 return new ResponseEntity<UserRest>(user1,HttpStatus.*OK*);  
 }  
//to get a specific user through the Id  
 @GetMapping(path = "/{userId}")  
 public ResponseEntity<UserRest> getUserModel( @PathVariable String userId){  
 if(users.containsKey(userId)){  
 return new ResponseEntity<>(users.get(userId),HttpStatus.*OK*);  
 }else{  
 return new ResponseEntity<>(HttpStatus.*NO\_CONTENT*);  
 }  
  
 }

## PUT:

For the put, we will look for the Id, and we will recieve a json so we will recieve the pathVariable and the RequestBody

When we recieve the pathVarible (userId) we will find it throught the Map id and create a new object

with this line: UserRest userEdit = users.get(userId);

after this, we will edit the object with the payload that we recieved :

@PutMapping(path = "/{userId}")  
public ResponseEntity<UserRest> editAUser(@PathVariable String userId,@RequestBody UserPostRequest userPost ){  
 UserRest userEdit = users.get(userId);  
 userEdit.setFirstName(userPost.getFirstName());  
 userEdit.setLastName(userPost.getLastName());  
   
 return new ResponseEntity<UserRest>(userEdit,HttpStatus.*OK*);  
}

DELETE:

To delete a object we will make a request call with the verb Delete, once we recieve it wit the pathvariable we will remove it

@DeleteMapping(path = "/{userId}")  
public ResponseEntity<Void> deleteAUser(@PathVariable String userId){  
 users.remove(userId);  
 return ResponseEntity.*noContent*().build();  
}

# Handling Errors:

For handling errors we must to create a new package with a new class that extends from **ResponseEntityExceptionHander** and annotated with **@ControllerAdvice**r is very important this annotation because once we have an error on our controller, this annotation will be on change to listen every error produced in the controller.

Also the method that will handle the exception, will have to be annotated with **@ExceptionHander(value={Exception.class})** but if i will handle a null pointer exception or a specific error i will have to putt he class of that expection.

@ControllerAdvice  
public class AppExceptionsHandler extends ResponseEntityExceptionHandler {  
  
 @ExceptionHandler(value = {Exception.class})  
public ResponseEntity<Object> handleAnyException(Exception ex, WebRequest request){  
return new ResponseEntity<>( ex, new HttpHeaders(), HttpStatus.*INTERNAL\_SERVER\_ERROR*);  
 }  
}

This method will provide all the **exception Stack trace** as body of the response and will provide a code status: 500 INTERNAL\_SERVER\_ERROR and as message will be on the body

## Custom Message Error:

We need to create a new class called ErrorMessage with only timestamp and mesage.

Two constructos, one empy and the other with the parameters

public class ErrorMessage {  
 private Date timestamp;  
 private String message;  
  
 //CONSTRUCTOR  
 public ErrorMessage(){}  
  
 public ErrorMessage(Date timestamp, String message){  
 this.timestamp= timestamp;  
 this.message = message;  
 }  
   
 //GETTES AND SETTES

and in the AppExceptionHandler we need to send it to the message

@ExceptionHandler(value = {Exception.class})  
public ResponseEntity<Object> handleAnyException(Exception ex, WebRequest request){  
 String errorMessageDescrition = ex.getLocalizedMessage();  
 if (errorMessageDescrition== null){  
 errorMessageDescrition = ex.toString();  
 }  
 ErrorMessage errorMessage = new ErrorMessage( new Date(), errorMessageDescrition);  
 return new ResponseEntity<>( errorMessage, new HttpHeaders(), HttpStatus.*INTERNAL\_SERVER\_ERROR*);  
}

## Another type of error:

Another kind of error could be Null pointer Exception that will,

For this we will change the class from Exception to NullpointerException and the argumento will be passed to the method.

//null pointer exception  
@ExceptionHandler(value = {**NullPointerException.class**})  
public ResponseEntity<Object> handleAnyException(**NullPointerException ex**, WebRequest request){  
 String errorMessageDescrition = ex.getLocalizedMessage();  
 if (errorMessageDescrition== null){  
 errorMessageDescrition = ex.toString();  
 }  
 ErrorMessage errorMessage = new ErrorMessage( new Date(), errorMessageDescrition, "null pointer exception");  
 return new ResponseEntity<>( errorMessage, new HttpHeaders(), HttpStatus.*INTERNAL\_SERVER\_ERROR*);  
}

## Custom Error Message

We have to create a new class and called UserServiceException(optional name) And this class has to extends from the type of error to be handled, example: RuntimeException. We have to create a constructor method and pass it a message(custom message) as parameter and call the super to mass it the message

public class UserServiceException extends RuntimeException {  
 public UserServiceException(String message){  
 super(message);  
 }  
}

In the class of the ControllerAdvicer we will create a new method using as value the class UserServiceException

@ExceptionHandler(value = {UserServiceException.class})  
public ResponseEntity<Object> handleAnyErrorCustom(UserServiceException ex, WebRequest request){  
 String errorMessageDescrition = ex.getLocalizedMessage();  
 if (errorMessageDescrition== null){  
 errorMessageDescrition = ex.toString();  
 }  
ErrorMessage errorMessage = new ErrorMessage( new Date(), errorMessageDescrition, "null pointer exception");  
 return new ResponseEntity<>( errorMessage, new HttpHeaders(), HttpStatus.*INTERNAL\_SERVER\_ERROR*);  
}

And finally when we have an error in the controller we will trigger it using the class UserServiceException with a custom message

if(true) throw new UserServiceException("a user servies exception was trown");

## Handle more tan one exception in a method.

Is basically the same method but we will recieve more tan one value, in this example we will use the custom message and the nullpointerexception in the same method

//null pointer exception // custom error message  
@ExceptionHandler(value = {UserServiceException.class, NullPointerException.class})  
public ResponseEntity<Object> handleAllExceptions(Exception ex, WebRequest request){  
 String errorMessageDescrition = ex.getLocalizedMessage();  
 if (errorMessageDescrition== null){  
 errorMessageDescrition = ex.toString();  
 }  
 ErrorMessage errorMessage = new ErrorMessage( new Date(), errorMessageDescrition);  
 return new ResponseEntity<>( errorMessage, new HttpHeaders(), HttpStatus.*INTERNAL\_SERVER\_ERROR*);  
}

# Dependency Injection:

First of all dependency injection works to avoid make a new instance of a object, and make the code less dependient of dependencies, it meas if we need to make a change on the code, this change can be done easily and faster causing less damages as possible.

<https://www.youtube.com/watch?v=sLY9umEahso>

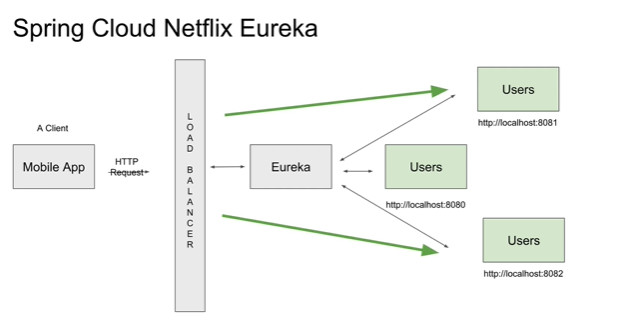
when we avoid to create a new instance of a object and we use only one instance (singleton) we make the application suitable due every instance created its memory ROM and ROM used.

For this, when we create a new object we use @Autowired and it will create only one instance of a object.

# Begining of the Project

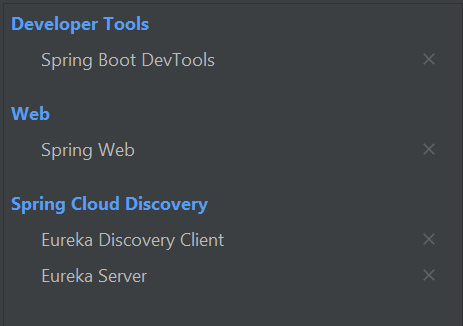
## Eureka Discover Service Netflix

Eureka discover service is a a service that will discover other services, these services send a (hearthbeat) to the adress that eureka has and eureka register this service and eureka ask to these sevices ping to eureka notifying that is still alive if any of these services fails will ask for it twice before turn it off untill it send thrice heatbeats notifying that is again alive



### Create a Eureka service

Set this dependencies



Once the created we have to enable it with this annotation **@EnableEurekaServer**

@EnableEurekaServer  
@SpringBootApplication  
public class EurekaApplication {

In the properties archive we need to set this configuration:

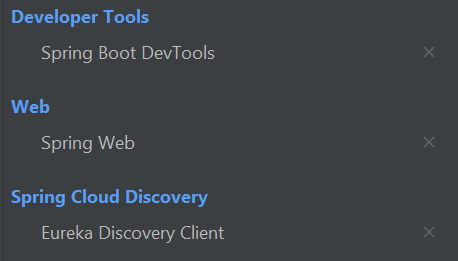
#name of the service  
spring.application.name= eureka-server #set server on default port  
server.port=8761  
#define eureka server as server and not as client  
eureka.client.register-with-eureka=false  
eureka.client.fetch-registry=false #adress of eureka server  
eureka.client.service-url.defaultZone=http://localhost:8761/eureka

We will be modifying this archive while the Project is more advenced

To check if this Works we need to open the following adress <http://localhost:8761/> if don’t, add /eureka

# Created a microservice hooked to Eureka:

Set the Project with this 3 dependencies:



After this, we have to hook the microservice to eureka, of the following configuration:

First of all we have to find eureka with this enabler, **@EnableDiscoveryClient**

@EnableDiscoveryClient  
@SpringBootApplication  
public class PhotoserviceApplication {

After that we need to set the properties archive like this:

#name of the servide  
spring.application.name=userPhoto  
#port of the service on 0 in order to set a random port  
server.port=${PORT:0}  
#eureka instance  
eureka.instance.instance-id=${spring.application.name}:${spring.application.instance\_id:${random.value}}

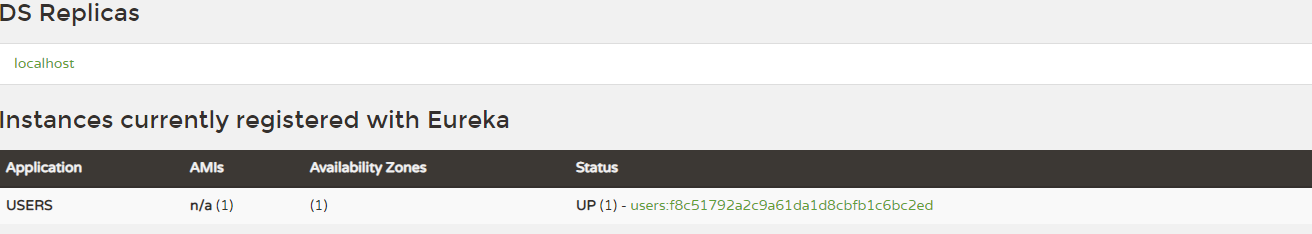
#adress of eureka server  
eureka.client.service-url.defaultZone=http://localhost:8761/eureka

#enable devtools restart in every change  
spring.devtools.restart.enabled=true

Create a controller following the best practices (a package and inside the controller ) in oder to check if everything is working well

@RestController  
@RequestMapping("/users")  
public class userController {  
 @GetMapping("/status")  
 public String status(){  
 return "its working!!";  
 }

Once done, we have to run eureka and after this microservice and opening the eureka adress ( <http://localhost:8761> ) we have to see this page



Let’s clik on the status of the microservice and write the path of the request of the controller in this case

/users/status and we have to see a “it’s working!!!”

# API GATEWAY (Zuul)

Zuul is a library generated by Netflix to have a single access point (gateway) to all the components that are part of our System, this means that in all our microservices they will have a single url instead of testing each one in the different ports or addresses in addition to functioning as an embedded load balancer (ribbon):

Without Zull:

<http://localhost:8090/ver/2/cantidad/5>

<http://localhost:53446/ver/1/>

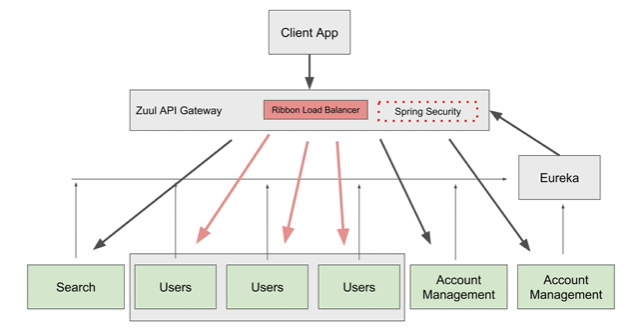
<http://localhost:53446/listar>

http://localhost:8090/listar

With Zuul:

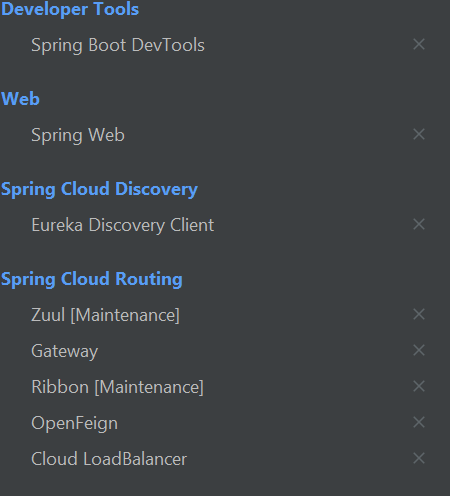
http://localhost:8080/api/items/listar

In addition to giving security to the microservices through the gateway, with spring security, if a request does not have the json web token (jwt) in the header (headers), it will reject the request.



## Create a Zuul service

Setting this dependencies



Configuration:

I n the main class we have to annotate it with **@EnableDiscoveryClient, @EnableZuulProxy**

@EnableDiscoveryClient //to register it to netflix  
@EnableZuulProxy //to enable it as gateway  
@SpringBootApplication  
public class ZuulgatewayApplication {

Y en los properties

#name of the servide  
spring.application.name=zuul-Gateway  
#port of the service 8080 to have a single port for all the services  
server.port=8080  
#register zuul as eureka client  
eureka.client.service-url.defaultZone=http://localhost:8761/eureka

To set the routes in zuul of how we are going to find the services:

zuul.routes.nameThatWeWillGiveToTheRoute.service-id=nameAsIsSetInTheMicroservice

zuul.routes.nameThatWeWillGiveToThePath.path=/nameOfThePath/\*\*

#zuul.routes.nameOfTheRoute.service-id=nameAsIsSetInTheMicrosevice   
zuul.routes.users.service-id=userphoto  
#path that we will give ejemplo apiusers/endpoints  
zuul.routes.users.path=/apiusers/\*\*  
#ACCOUNTMANAGER  
zuul.routes.account.service-id=accountmanagement  
zuul.routes.account.path=/apiaccount/\*\*

We will check if its working making a request in the browser of the following way,

http://localhost:8080/pathname/requestMapping/getMapping

http://localhost:8080/apiusers/users/status/

http://localhost:8080/apiaccount/account/status/check

# DATABASES Configuration

## SQL

Dependencias

<!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-data-jpa -->  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-jpa</artifactId>  
</dependency>

<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->  
<dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
</dependency>

spring.datasource.url=jdbc:mysql://localhost/DATABE\_NAME?serverTimezone=GMT-6

spring.datasource.url=jdbc:mysql://localhost/DATABASE\_NAME?serverTimezone=America/Mexico\_City  
spring.datasource.username=root  
spring.datasource.password=1234  
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver 🡸 pomxml lo autocomplet  
spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect  
spring.jpa.hibernate.ddl-auto=create-drop  
logging.level.org.hibernate.SQL=debug

Db that we will configure, can be any of both ways

Will create the tables automaticcally and will drop it when we the app is closed

ONLY FOR DEVELOPING ON PRODUCTION WE WILL HAVE TO CREATE BEFORE THE TABLES

Show the native SQL nativas queries

## H2 MEMORY DATABASE

To use a data base only for testing (H2)

Put this dependencies in POM.xml

Dependencies

<!-- https://mvnrepository.com/artifact/com.h2database/h2 -->  
<dependency>  
 <groupId>com.h2database</groupId>  
 <artifactId>h2</artifactId>  
 <scope>runtime</scope>  
</dependency>

<!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-data-jpa -->  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-jpa</artifactId>  
</dependency>

Properties

#H2 PROPERTIES  
#enable a console h2 to be displayed in a browser  
spring.h2.console.enabled=true  
#since we are going connect this service to zuul api gateway  
spring.h2.console.settings.web-allow-others=true

spring.datasource.url=jdbc:h2:mem:NOMBREDB  
spring.datasource.username=jonathan  
spring.datasource.password=sa  
spring.datasource.driver-class-name=org.h2.Driver

Corremos la aplicacion y abrimos la consola con la siguiente direccion

<http://localhost:8080/userphoto/h2-console>

<http://localhost:8080/nombre-del-microservicio/h2-console>

