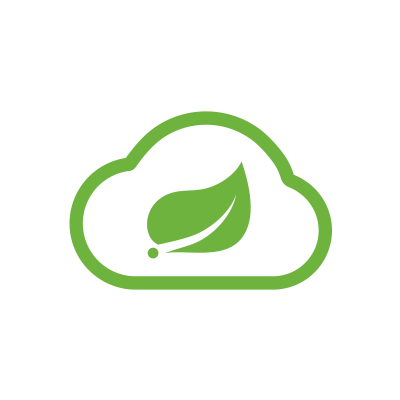
# Micro Services Spring Cloud

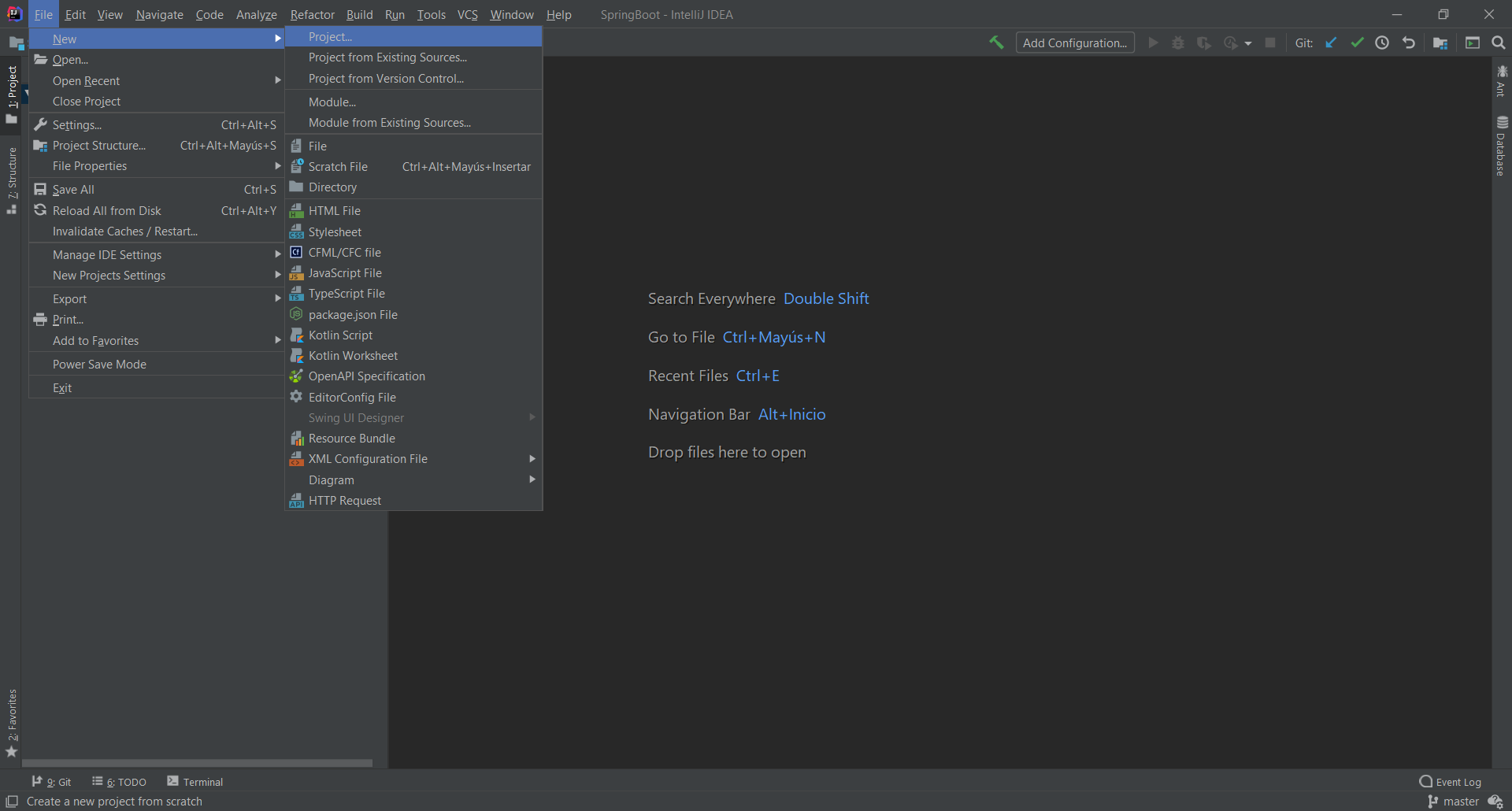


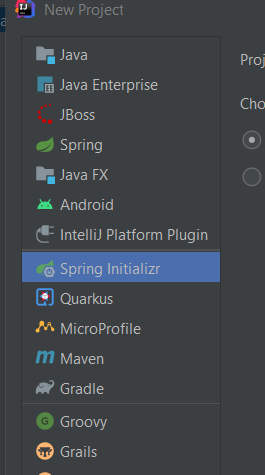
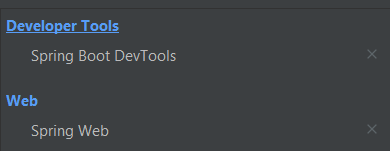
# Getting Started

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:

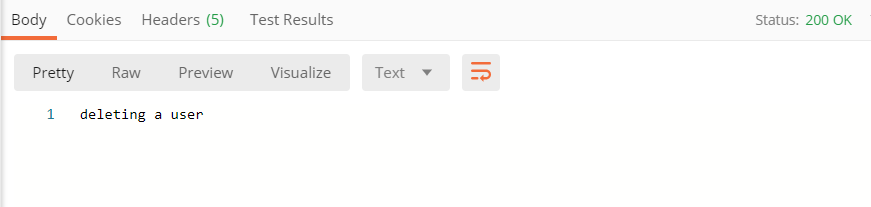
@RestController  
@RequestMapping("/users")  
public class Controller {  
  
 @GetMapping  
 public String getUser(){  
 return "getting user";  
 }  
  
 @PostMapping  
 public String postUser(){  
 return "user Created";  
 }  
  
 @PutMapping  
 public String editAUser(){  
 return "editing a user";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

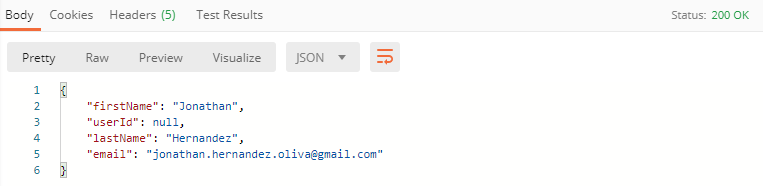
@GetMapping()  
public String getUsers(@RequestParam(value = "page", defaultValue = "1")int page,@RequestParam(value = "limit", defaultValue = "50")int limit ){  
 return "getting users of page: "+page+" and limit of: "+ limit+" users";  
}



## 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:

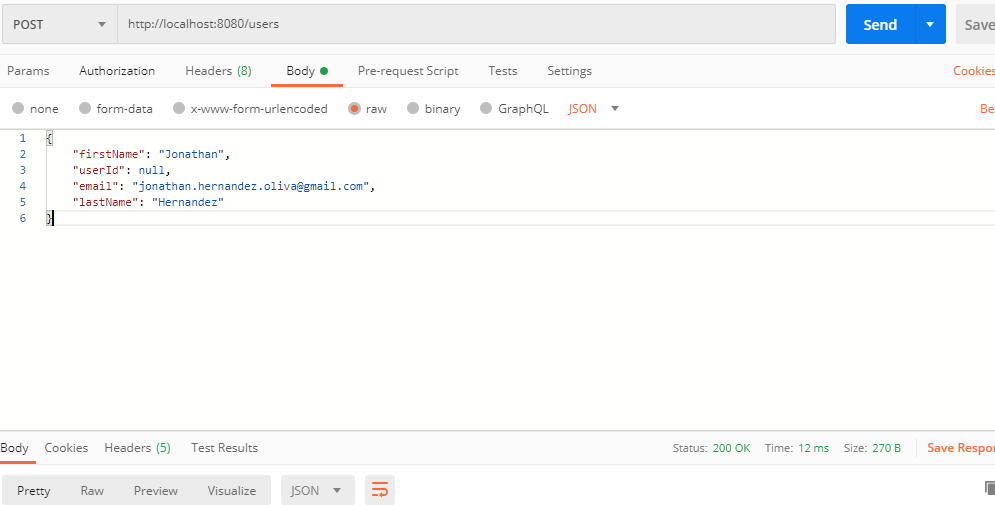
Post Verb

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>

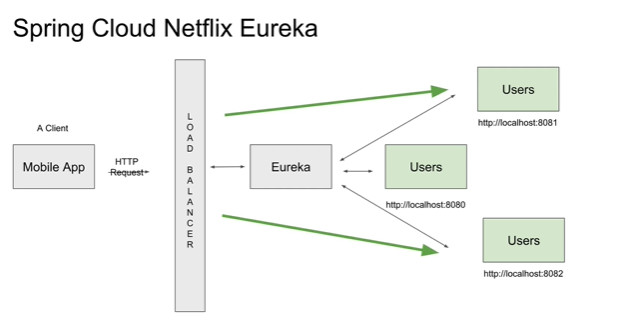
when we avoid to create a new instance of a object and we use only one instance (singleton) we make the application suitable due every instance created its memory ROM and ROM used.

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

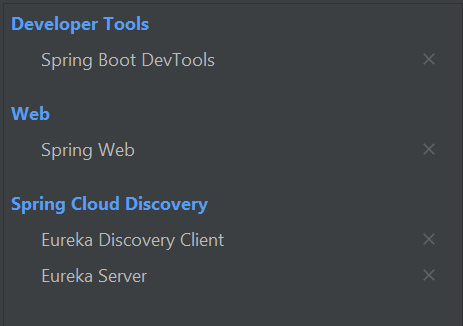
## Eureka Discover Service Netflix

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:

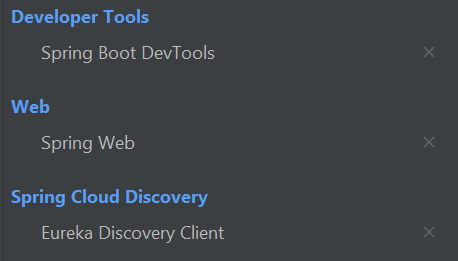
#name of the service  
spring.application.name= eureka-server #set server on default port  
server.port=8761  
#define eureka server as server and not as client  
eureka.client.register-with-eureka=false  
eureka.client.fetch-registry=false #adress of eureka server  
eureka.client.service-url.defaultZone=http://localhost:8761/eureka

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:

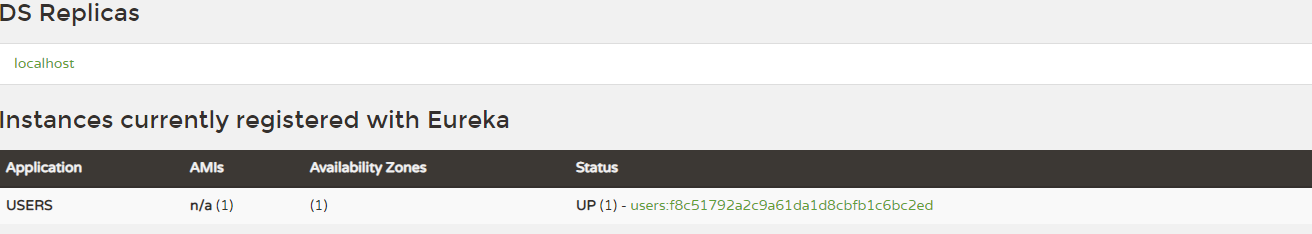
## properties

#name of the service  
spring.application.name=users  
#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}}  
#enable devtools restart in every change  
spring.devtools.restart.enabled=true  
#adress of eureka server  
eureka.client.service-url.defaultZone=http://localhost:8761/eureka

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/>

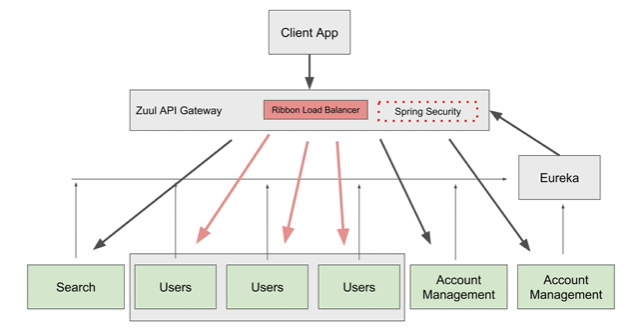
<http://localhost:53446/listar>

http://localhost:8090/listar

With Zuul:

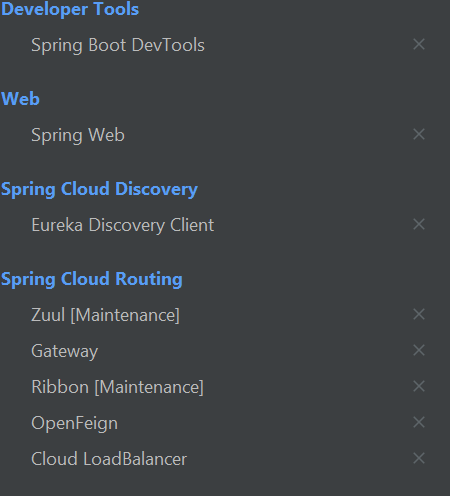
http://localhost:8080/api/items/listar

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 {

### 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/\*\*

**EUREKA DOES NOT NEED TO BE REGISTERED**

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

## Zuul as load balancer:

Once the endpoint is registered to zuul, we can test that RIBBON is embebed in zuul, just adding the dependency Environment form , then autorwire it and use the method getProperty(“local.server.port”)

@RestController  
@RequestMapping("/nameOfEnpoint")  
public class Controller {

@Autowired  
 private Environment env;  
   
 @getMapping("/test")  
 public String hello() {  
 return "working on port: "+env.getProperty("local.server.port");  
 }  
}

Once done, run, eureka 🡺 Zuul🡺 microservice

We open the browser with the following address: <http://localhost:8080/namegivenOfMicroservice/RequestMapping/getMapping>

We should have the following messageand if we refresh the port will change

# 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

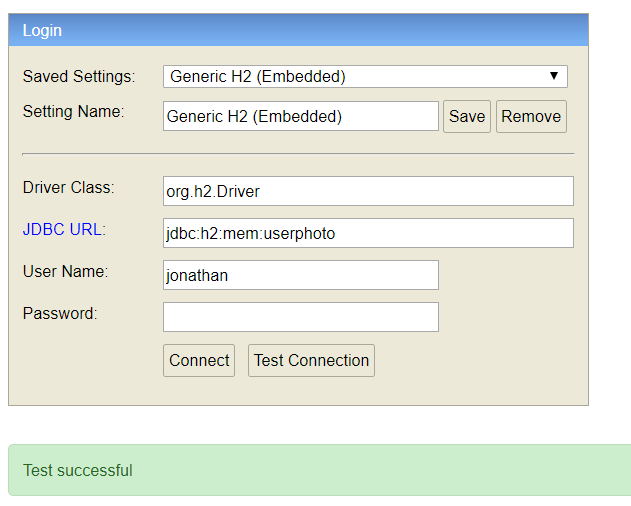
#H2 PROPERTIES  
#enable a console h2 to be displayed in a browser  
spring.h2.console.enabled=true  
#since we are going connect this service to zuul api gateway  
spring.h2.console.settings.web-allow-others=true

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

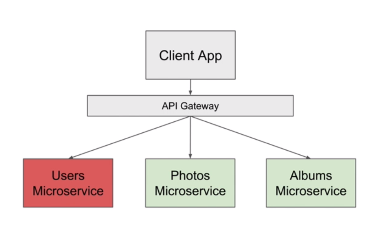
<http://localhost:8080/userphoto/h2-console>

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



# USER SIGN UP

The ms users will be in charge of register the users (create the users) – login – Ge update, delete.



### First trials

We have to create at first a method POST that will be in charge to create the user and just for testing will return a string “a user created”

@PostMapping  
public String createUser() {  
 return "user created ";  
}

Once created and working we can request for the body and validate it if all the inputs comply with the stipulations

### Create Model Filter

1.- create a class **CreateUserRequestModel,** this class will be in charge of **catching the inputs and validate it if all the inputs are ok**

**Note: the annotations must be from javax**

public class CreateUserRequestModel {  
 @NotNull(message = "first name cannot be null")  
 private String firstName;  
 @NotNull(message = "last name cannot be null")  
 private String lastName;  
 @NotNull(message = "password cannot be null")  
 @Size(min = 7, max = 16,message = "password must have at leat 7 Charactes and max 16 characters")  
 private String password;  
 @NotNull(message = "e-mail cannot be null")  
 private String email;  
  
 //GETTERS AND SETTERS

**@NotNull(message=”string”)** : indicates that this variale cannot be null

**@size(max=integer, min=integer, message=”string”):**  indicates the máximum and mínimum size of the variable and displays also a message

we add the annotation **@valid** that weill validate that all the annotations of the class acomply and **@RequestBody Class obj** that will transform the payload in a java object

@PostMapping  
public String createUser(@Valid @RequestBody CreateUserRequestModel user) {  
 return "working ";  
}

### Entity Class:

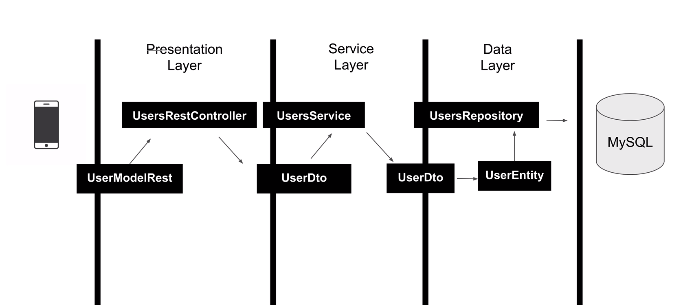
The Class Entity will be the one that will be “in touch” with the database extracting data or posting data so we need to be 100% sure that the data arrives to this class its correct.

@Entity  
@Table  
public class UserEntity implements Serializable {  
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private Long id;  
 @Column( nullable = false, length =50)  
 private String firstName;  
 @Column( nullable = false, length =50)  
 private String lastName;  
 @Column( nullable = false, length =50, unique = true)  
 private String email;  
 @Column( nullable = false, length =50, unique = true)  
 private String userId;  
 @Column( nullable = false, unique = true)  
 private String encryptedPassword;  
 @Column(nullable = false)  
 private Boolean enabled;

//GETTES AND SETTERS

### DTO Pattern:

This pattern pass a object to another class making tranformations in the object,

In this example we will use this patter passing the object in the following way:

For this patter we will user depenency model mapper

<!-- https://mvnrepository.com/artifact/org.modelmapper/modelmapper -->  
<dependency>  
 <groupId>org.modelmapper</groupId>  
 <artifactId>modelmapper</artifactId>  
 <version>2.3.8</version>  
</dependency>

You can delete the versión

**a class and a package Dto: explanation in the next page**

### UserDto

This class will be in charge to make the object bigger in order to transfer data from another object, example:

From CreateUserRequestModel data wich is only 5 fields to UserDto wich is 6 fields

public class UserDto implements Serializable {  
 private String firstName;  
 private String lastName;  
 private String email;  
 private String password;  
 private String userId;  
 private String encryptedPassword;  
  
 //GETTERS AND SETTERS

### UserRepository

Using this Repository will be very usefull and easier to persisto our data to the database

public interface UserRepository extends PagingAndSortingRepository<UserEntity, Long> {

### UserService

Create a interface in oder to implement it in a class that will work as intermediate between Entity(database) and controller, in this interface we will create a method to enforce the class that implement this interface override this method

public interface UserService {  
UserDto createUser(UserDto userDetails);

### UserServiceImpl:

In this class will implement the interface UserService and override the methods and will be decorated with the annotation **@Service**

@Service  
public class UserServiceImpl implements UserService{  
  
 @Override  
 public UserDto createUser(UserDto userDetails) {  
 return null;  
 }

firstly we will receive the payload directly from the client and **a filter will be applied**

@Valid @RequestBody CreateUserRequestModel user

If the object passes the filter is going to be **transferred** via the **modelmapper** object to the **UserDto class**

ModelMapper modelMapper = new ModelMapper();  
modelMapper.getConfiguration().setMatchingStrategy(MatchingStrategies.*STRICT*);  
UserDto userDto= modelMapper.map(user,UserDto.class);

in this class(userDto) the attributes of userId and encryptedPassword are added, making the object bigger and getting it ready to pass it to **userService interface** that will search throught **@Autowired** a class that implements the interface userService

userService.createUser(userDto);

once the class is found (UserServiceImpl) the method createUser will recieve the object userDto and will add the UUID to userId attribute and encryptedPassword will be added aswell once those attributes are added the **object will be transerred to UserEntity class** where will create the object to be stored in **userRepository interfac**

### Controller:

The controller will catch the payload **@RequestBody Class obejct** and will apply a filter **@Valid**

If the object passes the filter is going to be **transferred** via the **modelmapper** object to the **UserDto class**

@RestController  
@RequestMapping("/register")  
public class userController {  
  
 @Autowired  
 UserService userService;  
  
   
 @PostMapping  
 public String createUser(@Valid @RequestBody CreateUserRequestModel user) {  
 ModelMapper modelMapper = new ModelMapper(); modelMapper.getConfiguration().setMatchingStrategy(MatchingStrategies.*STRICT*);  
 UserDto userDto= modelMapper.map(user,UserDto.class);  
 userService.createUser(userDto);  
 return "working ";}  
}

**UserDto:**

in this class(userDto) the attributes of userId and encryptedPassword are added, making the object bigger and getting it ready to pass it to **userService interface** that will search throught **@Autowired** a class that implements the interface userService

once the class is found (**UserServiceImpl**) the method createUser will recieve the object userDto and will add the UUID to userId attribute and encryptedPassword will be added aswell once those attributes are added the **object will be transerred to UserEntity class** where will create the object to be stored in **userRepository interface**

### UserService Impl 2nd part

For this class will be needed to inject UserRepository via Contructor:

Once the controller sends the UserDto to the service, this serviceClass will catch the object UserDto and will add the UUID to the userId and will encript the password, and will send it to the UserRepository to persist ir throught UserEntity. But since userEntity does not admit null in encryptedPassword, we will set it as “test”

**note: encryptedPassword will be replaced in the following lecture because Spring secutiry is needed for this.**

@Service  
public class UserServiceImpl implements UserService {  
  
 UserRepository userRepository;  
  
 @Autowired  
 UserServiceImpl(UserRepository userRepository){  
 this.userRepository= userRepository;  
 }  
  
 @Override  
 public UserDto createUser(UserDto userDetails) {  
 userDetails.setUserId(UUID.*randomUUID*().toString());  
 ModelMapper modelMapper = new ModelMapper();  
 modelMapper.getConfiguration().setMatchingStrategy(MatchingStrategies.*STRICT*);  
 UserEntity userEntity=modelMapper.map(userDetails,UserEntity.class);  
 userEntity.setEncryptedPassword("test");  
 userRepository.save(userEntity);  
 return null;  
 }  
}

**At this moment the user is persisted in H2 or MySQL depending of the selected configuration of the developer:**

#######DATABASE######  
####activar H2 database#####  
#spring.h2.console.enabled=true  
#spring.h2.console.settings.web-allow-others=true  
#spring.datasource.url=jdbc:h2:mem:users  
#spring.datasource.username=jonathan  
#spring.datasource.password=sa  
#spring.datasource.driver-class-name=org.h2.Driver  
#####SQL#######  
spring.datasource.url=jdbc:mysql://localhost/users?serverTimezone=GMT-6  
spring.datasource.username=root  
spring.datasource.password=1234  
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver  
spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect  
spring.jpa.hibernate.ddl-auto=create-drop  
logging.level.org.hibernate.SQL=debug

### Send Response code:

**controller**

In the function where we created the new user we have to return ReponseEntity<T> :

public ResponseEntity createUser(@Valid @RequestBody CreateUserRequestModel user) {

…

return new ResponseEntity(HttpStatus.*CREATED*)

**the response that we are going to get is 200 Ok**

### Response Model

If you want to return a payload as response but its necessary do not show sensitive information as **Password** or **publicUserId,** its necessary create a new class(responseEntity), excluding password and userid.

public class ResponseModel {  
 private String firstName;  
 private String lastName;  
 private String email;  
 private String userId;  
//GETTERS AND SETTERS

**Controller:**

We specify that we are going to return ResponseModel in the function createUser

public ResponseEntity<ResponseModel> createUser(@Valid @RequestBody

and the return, will be

~~return new ResponseEntity(HttpStatus.~~*~~CREATED~~*~~)~~

return ResponseEntity.*status*(HttpStatus.*CREATED*).body(**body**);

the body will be the object that we are going to return (**responseModel**) but first, the method createUser from the object userService will return a object **userDto** (in the next step) and that will make a new object type UserDto.

UserDto createdUser=userService.createUser(userDto);

and then, that object will make throug modelMapper a new object (ResponseModel) and finally pass it as body

ResponseModel responseModel = modelMapper.map(createdUser, ResponseModel.class);  
return ResponseEntity.*status*(HttpStatus.*CREATED*).body(responseModel);

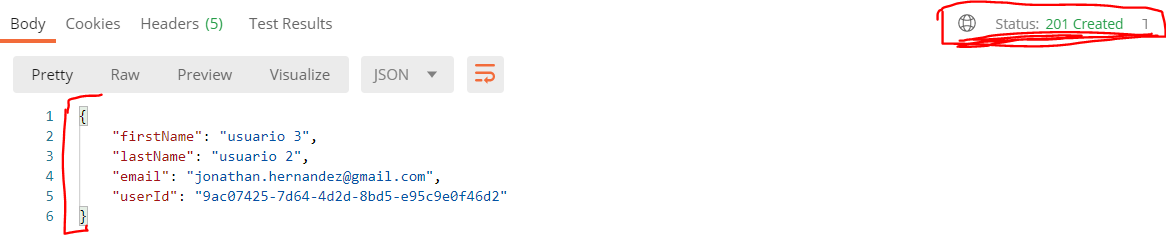
**Complete:**

@PostMapping  
public ResponseEntity<ResponseModel> createUser(@Valid @RequestBody CreateUserRequestModel user) {  
 ModelMapper modelMapper = new ModelMapper();  
 modelMapper.getConfiguration().setMatchingStrategy(MatchingStrategies.*STRICT*);  
 UserDto userDto= modelMapper.map(user,UserDto.class);  
 UserDto createdUser=userService.createUser(userDto);  
 ResponseModel responseModel = modelMapper.map(createdUser, ResponseModel.class);  
 return ResponseEntity.*status*(HttpStatus.*CREATED*).body(responseModel);  
}

**UserServiceImpl:**

we are going to to transer the user object **userEntity** (that has the password and private userid) to **userDto** againand make a object of **type UserDto** and return it

UserDto returnValue = modelMapper.map(userEntity, UserDto.class);  
return returnValue;



Cliente

Controller

Base de Datos

### Encrypt password:

#### WebSecurity

Is needed to add the security dependency, and after that all the calls will be **Unauthorized**

So its needed to unableit temporary

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

**Disable security**

Create a new package (**security**) and create a new class in named **WebSecurity,**

and we have to override a method from security named configure, this method recieve a object of type HttpSecurity and with this object we have to disable **crosssiteforgery** and we have to **permit all requests**

@Configuration  
@EnableWebSecurity  
public class webSecurity extends WebSecurityConfigurerAdapter {  
  
@Override  
 protected void configure(HttpSecurity http) throws Exception{  
 http.csrf().disable();  
 http.authorizeRequests().antMatchers("/users/\*\*").permitAll();  
}  
}

**Encrypt Password:**

We have to autowire the **BCryptPasswordEncoder.**

**userServiceImpl:**

**BCryptPasswordEncoder bCryptPasswordEncoder;**  
  
@Autowired  
UserServiceImpl(UserRepository userRepository, **BCryptPasswordEncoder bCryptPasswordEncoder**){  
 this.userRepository= userRepository;  
 **this.bCryptPasswordEncoder = bCryptPasswordEncoder;**  
}

**UsersApplication:**

We have to add it to the constructor but for BCrypritPasswordEncoder to be injected into our service implementation via constructor **its object needs to exist in application**

**Dont forget the annotations**

**@SpringBootApplication  
@EnableDiscoveryClient**  
public class UsersApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(UsersApplication.class, args);  
 }  
 **@Bean  
 public BCryptPasswordEncoder bCryptPasswordEncoder(){  
 return new BCryptPasswordEncoder();  
 }**  
}

**UserServiceImpl:**

**Detele the setEnctyptedPassword(“test”) and create the one that will be encrypted in the following way**

userDetails.setEncryptedPassword(bCryptPasswordEncoder.encode(userDetails.getPassword()));

### Allow only IP adress of zuul API adress:

properties

Add the ip adress in the properties file

#IP ADRESSS  
gateway.ip= 192.168.1.71

Then add the Environment dependency as contructor and Autowire it.

private Environment environment;  
@Autowired  
public webSecurity (Environment environment){  
 this.environment = environment;  
}

and modify the authorizeRequests

@Override  
 protected void configure(HttpSecurity http) throws Exception{  
 http.csrf().disable();  
 http.authorizeRequests().antMatchers("/users/\*\*").**hasIpAddress(environment.getProperty("gateway.ip"));  
}**  
}

## Login:

User

/login

AuthenticationFilter

attempAuthentication

successfulAuthentication

Token

userId

### Login model:

Create a model Class (LoginRequestModel) for email and password

public class LoginRequestModel {  
 private String email;  
 private String password;  
 //GETTES AND SETTERS

### AuthenticationFilter:

Create a new class for **AuthenticationFilter** that will extends **UsernamePasswordAuthenticationfilter** and this class has a method for authentication that we have to override

import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;  
import org.springframework.security.core.Authentication;  
import org.springframework.security.core.AuthenticationException;

When a user attempt to login Spring will call a endpoint (“/login”) where will find a method (attempAuthentication)

1. this method **attemptAuthentication** will use **httpRequest,** to read username and password from **httpRequest** (from client)and **objectMapper** will map username and password to **LoginRequestModel (return a LoginRequestModel object “ceeds”).**
2. And this function (**attemptAuthentication)** will return a object **getAuthenticationManager()** (from spring security) using the method **authenticate()** that recieves as parameter an object **usernamePasswordAuthenticationToken** wich recieves **email and password and a list of authorities**

public class AutenticationFilter extends UsernamePasswordAuthenticationFilter {  
 @Override  
 public Authentication attemptAuthentication(HttpServletRequest req,  
 HttpServletResponse res) throws AuthenticationException {  
 try{  
 LoginRequestModel creds = new ObjectMapper().readValue(req.getInputStream(),LoginRequestModel.class);  
 return getAuthenticationManager().authenticate(new UsernamePasswordAuthenticationToken(  
 creds.getEmail(),  
 creds.getPassword(),  
 new ArrayList<>()));  
 }catch (IOException e){  
 throw new RuntimeException(e);  
 }  
 }  
}

**if getAuthenticationManager is successfull Spring security will call successfulAuthentication method wich is also be override:**

@Override  
protected void successfulAuthentication(HttpServletRequest req,  
 HttpServletResponse res,  
 FilterChain chain,  
 Authentication auth) throws IOException, ServletException{  
   
}

**and now we need to register this AuthenticationFilter with httpSecurity on webSecurity class**

### register authenticationFilter method in websecurity

**Websecurity:**

In the configure method, modify it with and().addFiler(), in this action we are indicating to spring security that we are going to accept request from only one IP adress and we are going to add a filter

@Override  
 protected void configure(HttpSecurity http) throws Exception{  
 http.csrf().disable();  
 http.authorizeRequests().antMatchers("/users/\*\*").hasIpAddress(environment.getProperty("gateway.ip"))  
 **.and()  
 .addFilter( getAuthenticationFilter());  
}**

And we have to create the method getAuthenticationFilter

Creating a function that returns AuthenticationFilter object (the class that we already created) and create a object of AuthenticationFilter and set AuthenticationManager setAutenticationManager is not a custom method it comes from sping security the reason for setAutenticationManager is in **AuthenticationFilter** its called from the method getAuthenticationManager wich also comes from spring and this method will return **authenticationManager** object

protected AutenticationFilter getAuthenticationFilter() throws Exception{  
 AutenticationFilter autenticationFilter = new AutenticationFilter();  
 autenticationFilter.setAuthenticationManager(authenticationManager());  
 return autenticationFilter;  
}

for spring framework to be able to authenticate user with provided email and password it needs to know if we do have this user in the database in other words it need to call a method that will tells spring framework if this user is found in the database or not

when a user sends a hhtp request to perform login and spring will try to find the user and password and it need to know where and how to find userDetails and for that we will implement a couple of methods

@Override  
 protected void configure(AuthenticationManagerBuilder auth) throws Exception{  
 auth.userDetailsService(userService).passwordEncoder(bCryptPasswordEncoder);  
}

This method will use a service (userService) and wich passwordEncoder will use (BCryptPasswordEncoder)

But in this webSecurity Class we do not have the service and the password encoder so it need to be injected

So we need to injecte them

private Environment environment;  
**private UserService userService;  
private BCryptPasswordEncoder bCryptPasswordEncoder;**  
  
@Autowired  
public webSecurity(Environment environment, UserService userService, BCryptPasswordEncoder bCryptPasswordEncoder){  
 this.environment = environment;  
 **this.userService= userService;  
 this.bCryptPasswordEncoder= bCryptPasswordEncoder;**  
}

### Finduser by Emial

And after this, will show an error because userService need to implemento or extends a sprinf framework interface

**userService** interface

public interface UserService **extends UserDetailsService** {

**UserServiceImpl**

as UserService extends **UserDetailsService** will need to implement a method called **LoadUserByUsername** this for obtain the username(e-mail), so first will be created in **service** interface the implemented in **UserServiceImpl**

**UserService**

public interface UserService extends UserDetailsService {  
 UserDto createUser(UserDto userDetails);  
 **UserDto getUserDetailsByEmail(String email);**  
}

**UserServiceImpl**

@Override  
public UserDto getUserDetailsByEmail(String email) {  
  
}

**To obtain something from the database** it’s needed to use the interface **userRepository** like this:

userRepository.get---

if we type this, will appear some methods to obtain something like findBy… but there is not findByemail so its needed to create it.

**UserRepository**

There is a special wey to create queries, for example if we want to obtain something we will need to write it

findBy... example: findByEmail, findByLastName etc.

deleteBy…

in this way JPA will create the query

public interface UserRepository extends CrudRepository<UserEntity, Long> {  
 **UserEntity findByEmail(String email);**  
  
}

**userServiceImpl**

this method will return a User object that comes from **userDetails** interface with email, password encrypted and booleans arguments in usernamePasswordAuthenticationToken will the recieve principal(email) and credentials (password) the principal will send to this method(**LoadUserByUsername**)

@Override  
public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {  
 UserEntity userEntity= userRepository.findByEmail(username);  
 if(userEntity== null){  
 throw new UsernameNotFoundException(username);  
 }  
 return new User(userEntity.getEmail(),userEntity.getEncryptedPassword(),true,true,true,true,new ArrayList<>());  
}

**At this moment the code should work, if we register a new user and we login user the /login path intead of users sould work, this path will be change later**

<http://localhost:8080/users/login>

<http://localhost:8080/microserviceName/login>}

the servlet will respond with a status 200 OK

next step its get a Token and return it in the header

### Returning Token

The method **successfulAuthentication** in **UserServiceImpl** will be atctived **if the username and password are correct** so we have to find the user by the email given to encrypt it with the token, so we have to **create a new method in the service** **interface** to get the email **getUserDetailsByEmail** and implement it in **userServiceImpl**

**UserService**

public interface UserService extends UserDetailsService {  
 UserDto createUser(UserDto userDetails);  
 **UserDto getUserDetailsByEmail(String email);**  
}

**UserServiceImpl**

In the implementation we will find the object by the email using the query already done **(findByEmail)** once fund it makes this object into a object Entity (UserEntity) **if its found we will return a Dto object** to take out the encripted password and userid and **if its not found we will return a exception** **UsernameNotFoundException**

@Override  
public UserDto getUserDetailsByEmail(String email) {  
 UserEntity userEntity= userRepository.findByEmail(email);  
 if(userEntity== null){  
 throw new UsernameNotFoundException(email);  
 }  
 return new ModelMapper().map(userEntity,UserDto.class);  
}

**AuthenticationFilter**

in successfulAuthentication method will get the principal(email) and save it as username

and will call the function getUserDetailByEmail throught userService and make at object type of userDetails

@Override  
protected void successfulAuthentication(HttpServletRequest req,  
 HttpServletResponse res,  
 FilterChain chain,  
 Authentication auth) throws IOException, ServletException{  
 **String username =((User) auth.getPrincipal()).getUsername();  
 UserDto userDetails= userService.getUserDetailsByEmail(username);**

### Add JWT Dependency

Add the JWT dependency in POM in order to create tokens

<!-- https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt -->  
<dependency>  
 <groupId>io.jsonwebtoken</groupId>  
 <artifactId>jjwt</artifactId>  
 <version>0.9.1</version>  
</dependency>

**AuthenticationFilter:**

We will need the environment dependency to inject the token.expiration\_time, and token.secret

//GENERATE JWT  
String token = Jwts.*builder*()  
 .setSubject(userDetails.getUserId())  
 .setExpiration( new Date(System.*currentTimeMillis*()+ Long.*parseLong*(environment.getProperty("token.expiration\_time"))))  
 .signWith(SignatureAlgorithm.*HS512*, environment.getProperty("token.secret"))  
 .compact();

//RETURN HEADERS   
 res.addHeader("token", token);  
 res.addHeader("id",userDetails.getUserId());

**properties**

###TOKEN EXPIRATION  
#10 days  
token.expiration\_time= 864000000  
###TOKEN SECRET  
token.secret=secrettoken05082020M

### Change Login URL path

**webSercurity**

To change the url path default will add the following snipet in

**autenticationFilter.setFilterProcessesUrl(environment.getProperty("path.login"));**

protected AutenticationFilter getAuthenticationFilter() throws Exception{  
 AutenticationFilter autenticationFilter = new AutenticationFilter(userService, environment,authenticationManager());  
 autenticationFilter.setAuthenticationManager(authenticationManager());  
 **autenticationFilter.setFilterProcessesUrl(environment.getProperty("path.login"));**  
 return autenticationFilter;  
}

**properties**

##PATH TO LOGIN  
path.login=/users/login

**Try it**

## Security in Gateway

The Gateway will provide security to the privates Microservices, asking for the JWT, if the JWT provided by the client, the Gateway will response with the request if the JWT its not provided, the Gateway will reject the request

### Add dependencies

The Gateway will need 2 dependencies, Spring security and JWT

**POM.xml**

<!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-security -->  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-security</artifactId>  
</dependency>  
<!-- https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt -->  
<dependency>  
 <groupId>io.jsonwebtoken</groupId>  
 <artifactId>jjwt</artifactId>  
 <version>0.9.1</version>  
</dependency>

### Enable web security in Zuul

Create a new package (Security) and create in a new class, WebSecurity

#### Websecurity

The class will be annotated with **Configutation** and **EnableWebSecurity** and the class will extends **WebSecurityConfigurerAdapter**

Also will be needed some variables form the properties files so we need to **inject the Enviroment object**

@Configuration  
@EnableWebSecurity  
public class WebSecurity extends WebSecurityConfigurerAdapter {  
  
 @Autowired  
 Environment environment;  
 WebSecurity(Environment environment){  
 this.environment= environment;  
 }  
   
 @Override  
 protected void configure(HttpSecurity http) throws Exception{  
 http.csrf().disable();  
 http.headers().frameOptions().disable();  
 http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.*STATELESS*);  
 }  
}

And override the method consigure with the same configurations as before buut adding **sessionManagement**

**sessionManagement().sessionCreationPolicy()SessionCreationPolicy.STATELESS**

this configuration will make our API Stateless. So it means will never create a HttpSession so its more safe because if the customer creates a sesión can get in the cookies some information about the request.

#### Allow Access to Registration and Login Urls

In this configuration will allow certains endpoints make post like users because it doesnt need to request for the JWT because, the client doesnt have an account yet,

@Override  
protected void configure(HttpSecurity http) throws Exception{  
http.csrf().disable();  
http.headers().frameOptions().disable();  
http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.*STATELESS*);  
//for H2 console **http.authorizeRequests().antMatchers(environment.getProperty("h2.console\_allowed")).permitAll();**  
//for register and login  
**http.authorizeRequests().antMatchers(HttpMethod.*POST*,environment.getProperty("user.registration")).permitAll();**  
**http.authorizeRequests().antMatchers(HttpMethod.*POST*,environment.getProperty("user.endpoint\_allowed")).permitAll()**  
**.anyRequest().authenticated();**

.anyRequest().Authenticated():

Means that any Request needs to be authenticated with JWT

Properties

##ENDOINT ALLOWED TO MAKE POST LOGIN  
user.endpoint\_allowed=/users/users/login  
##REGISTRATION USERS ALLOWED  
user.registration=/users/register/  
##h2 console allowed check  
h2.console\_allowed=/users/h2-console/\*\*

### Implementing an Authorizaiton Filter

This filter will validate the JWT provided in the request in the headers

Create in the package security a class named **AuthenticationFilter** end extends from **BasicAuthenticationFilter** and will demand a **constructor** calling to the **super** and **Environment** will de added aswell.

public class AuthorizationFilter extends BasicAuthenticationFilter {  
 Environment environment;  
  
 public AuthorizationFilter(AuthenticationManager authenticationManager, Environment environment) {  
 super(authenticationManager);  
 this.environment= environment;  
 }

AuthorizationFilter:

First in the Http call, this function will catch the request and get if in the header exist a header with name **Authorization**. If this header does not exist or does not starts with the prefix **Bearer**, then call the next filter in chain, but if does exist we need to perform authentication and return a object of type **UsernamePasswordAuthenticationToken.**

@Override  
protected void doFilterInternal(HttpServletRequest req,  
 HttpServletResponse res,  
 FilterChain chain) throws IOException, ServletException{  
 String authorizationHeader = req.getHeader(environment.getProperty("authorization.token.header.name"));  
 if(authorizationHeader== null || !authorizationHeader.startsWith(environment.getProperty("authorization.token.header.prefix"))){  
 chain.doFilter(req,res);  
 return;  
 }  
 UsernamePasswordAuthenticationToken authentication = getAuthentication(req);  
 SecurityContextHolder.*getContext*().setAuthentication(authentication);  
 chain.doFilter(req,res);  
  
}

Since we are using environment to get the properties, in the properties file will add the constants

Properties:

###NAME OF THE AUTHORIZATION HEADER  
authorization.token.header.name= Authorization  
##prefix of the authorization header  
authorization.token.header.prefix=Bearer

AuthorizationFilter:

private UsernamePasswordAuthenticationToken getAuthentication(HttpServletRequest req){  
 String authorizationHeader = req.getHeader(environment.getProperty("authorization.token.header.name"));  
 if(authorizationHeader==null){  
 return null;  
 }  
 String token = authorizationHeader.replace(environment.getProperty("authorization.token.header.prefix"),"");  
 String userId= Jwts.*parser*()  
 .setSigningKey(environment.getProperty("token.secret"))  
 .parseClaimsJws(token)  
 .getBody()  
 .getSubject();  
 if(userId==null){  
 return null;  
 }  
 return new UsernamePasswordAuthenticationToken(userId,null,new ArrayList<>());  
  
}

# Oauth 2.0

## What is Oauth 2.0

In this exercise we will create the same funcitonality but in different way

### Entities

**Users**

As the previous example we will add the same fields but **enabled**, instead of deleting a user, will be unabled it so we keep the information, and for every user we will add **Roles**

And since the user can have many roles we will add a list of roles for every user.

@ManyToMany(fetch = FetchType.*LAZY*)

private List<RoleEntity> role;

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

Since we are using a List of another Entity spring knows that we are going to join two tables UserEntity and RoleEntity so it needs to know the cardinality (@ManyToMany), once compiling this JPA will create the tables automatically, but if we want to custom the name of the jointable we will nee to add

**Fetch=FetchType.LAZY**, indicates when we request for the users, ONLY the users will be returned and not the roles aswell, by default this annotation is EAGER

**@JoinTable(name=”user\_to\_role”)** this command will renama the table

**JoinColumns=@JoinColumn(name=”user\_id”)** will change the name userEntity\_id to user\_Id

**inverseJoinColumns=@JoinColumn(name=”role\_id”)** will change the name of the Foreing key

**uniqueConstraints={@UniqueContraints(columnNames={“user\_id”,”role\_id”})}** indicates that those two files together must be uniques, **it means the user cannot have the same role id**

UserEntity

@Entity  
@Table  
public class UserEntity implements Serializable {  
 @Id  
 @GeneratedValue  
 private Long id;  
 @Column( nullable = false, length =50)  
 private String firstName;  
 @Column( nullable = false, length =50)  
 private String lastName;  
 @Column( nullable = false, length =50, unique = true)  
 private String email;  
 @Column( nullable = false, length =50, unique = true)  
 private String userId;  
 @Column( nullable = false, unique = true)  
 private String encryptedPassword;  
 @Column(nullable = false)  
 private Boolean enabled;  
  
 @ManyToMany(fetch = FetchType.*LAZY*)  
 @JoinTable(name = "users\_to\_roles" , joinColumns = @JoinColumn(name = "user\_id"),inverseJoinColumns = @JoinColumn(name="role\_id"),uniqueConstraints = {@UniqueConstraint(columnNames = {"user\_id","role\_id"})})  
 private List<RoleEntity> role;

Role Entity

@Entity  
@Table  
public class RoleEntity implements Serializable {  
   
 @Id  
 @GeneratedValue(strategy = GenerationType.*IDENTITY*)  
 private String id;  
   
 @Column(unique = true, nullable = false, length = 20)  
 private String name;  
   
 //GETTERS AND SETTERS

### Repository

Intead of using **CrudRepository** as the previous example, we will **use PagingAndSortingRepository<Entity,Long>** interface, this interface extendas from CrudReposiory but has more functionalities such as sorting and paging

Will create a method findByEmail, this method will create a query to find a user by email

public interface UserRepository extends PagingAndSortingRepository<UserEntity, Long> {

UserEntity findByEmail(String email);  
}

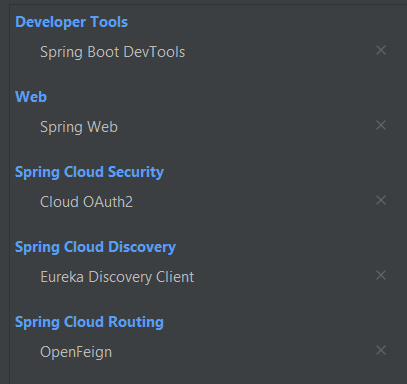
But if we want to make a custom query or using a native query, will be the following example.

@Query(value = "select u from UserEntity u where u.email=?1 and u.firstName=?2", nativeQuery = false)  
 UserEntity obtenerPorEmailYUsername(String email, String username);  
}

And we can create the same way the users

# Create Authorization server

Create a new Project with the following starter dependencies



The new dependency that we find here is OpenFeign

OpenFeign works to comunícate between Microservices