Difference between **@PathVariable and @RequestParam** :

***@RequestParam*s extract values from the query string, *@PathVariable*s extract values from the URI path**:

From <[*https://www.baeldung.com/spring-requestparam-vs-pathvariable*](https://www.baeldung.com/spring-requestparam-vs-pathvariable)>

**@PathVariable** : It is used where the parameter is mandatory to pass in the requested URL, It can't be optional. In URL it is like : localhost:8080/product/1 , so here 1 is the Path Variable, similarly we ca pass multiple path variables like localhost:8080/product/1/2 , so here 1 and 2 are path variables. Now let us see how can we map them in our request mapping i.e. in our controller so we need to add the these param in the requested URL like @GetMapping("/product/{id1}/{id2}") so here by putting them in {}, we are telling these are the path Variables, now we to pass them into the function we have to use @PathVariable there like (@PathVariable Long id1, @PathVariable Long id2) now we will get these id1 and id2 in our method, if we are using some other name here in the method like (@PathVariable("id1") Long employeeId), then we can use() to specify for which Path Variable we are using this, so here as I specified that map the id1 variable to employeeId.

From **Spring 4.3.3 onwards we can use @PathVariable also as optional,** we can do **(@PathVariable(required = false) Long id), so now it is the optional field, but even for this we need to specify two URLs one with path variable and one without. Ex : @GetMapping("/product","/product/{id}"**), now the path variable is optional.

**@RequestParam** : We can use request param also in both the ways as optional and required by specifying required field as true or false. In the URL it seems like "**/product?id=2&categroy=laptop**" so here we have used two request parameters, we pass the params using ? Then param name = param value like id=2, and we can separate the request params using &, we can pass any number of params from here. In mapping it seems like "/product" and in method it will be like (**@RequestParam(name ="id") Long employeeId , @RequestParam(required = false, name = "category") String cat**), So here we made id as mandatory and mapped it to employeeId, and for category we made it as optional and mapped it to cat, same thing we follows here if we are using the same name for variable then no need to externally mention the name, but if we are using different names then we to specify with name, which to map with which one.

**Logging** :

We can just mark our class with **@Log4j2 and now we don't need to create a new object of logger and then use we can direct use log.debug() like this**.

If we want to throw some custom exception on some issue then we can create a class with the name what we want and extend Exception class like we do in our simple java.

**Handling Custom Exception's Response.**

A screenshot of a computer program

Description automatically generated

Like here we have created a class ProductServiceCustomException, we will throw this error, when there will be any issue with product in our project. We kept one extra field here as errorCode so to tell what is the issue with functionality. To throw this error we can simply do :

A computer screen shot of text

Description automatically generated

Here we are throwing this error and giving error code and message accordingly. Now whenever the issue will occur this error will be thrown, but now it will throw this error, we also have to handle this error and need to send a good response that can simply tell what the issue occurred.

So for that we need to make a class with whatever name we want and we need to extend **ResponseEntityExceptionHandler** and need to mark the class with **@ControllerAdvice annotation** that will tell our application that this class is made to handle the exception. Then we need to create the function that will handle the error for particular class we need to mark that function with **@ExceptionHandler** and need to pass the Exception name in that like here in our case it would be like **@ExceptionHandler(ProductServiceCustomException.classs)** and need to pass the argument of that exception class to that function.

A screen shot of a computer code

Description automatically generated

Here the implementation will be like this. We have used **builder** class here from **lombok**, It helps us in building the objects with those fields which we want to pass. Like if this **ErrorResponse** has five fields then we can make the project by passing these two values only **error code and error message**.

How to register our microservice as eureka server : **the microservice is known as service registry which will be registered as eureka server and our all other microservices that are running will be registered there as a client and we can see status of all microservices from that service registry**.

To register our microservice as eureka server we need to add **eureka server** dependency and cloud bootstrap dependency. And after adding that, we need to add the **@EnableEurekaServer** annotation on our main Application class.

A computer code with black text

Description automatically generated

Now we need to add some properties in application.properties file :

A screen shot of a computer code

Description automatically generated

So in the very first line we are giving this application a name, then we have given it a port as 8761, then we gave it a hostname, at which host will it run. **After that we gave false for two values, as this is the eureka server and we don't itself to register it with itself that's why we gave it as false, in the case of clients we will give it as true**.

How to register our microservice as client in the above eureka server : **So for that we need to add one dependency Eureka Discovery Client it will allows us to register the microservice on this eureka server**. After that we need to add some properties listed below.

A computer screen with blue text

Description automatically generated

These properties we need to add to register our application as client on eureka server.

To add our application with **MYSQL using JPA we need to add dependencies of MYSQL and JPA**, and then need to add these properties.

A screen shot of a computer

Description automatically generated

Here we are telling JPA to update the tables, and in URL we provided localhost and orderdb is the database/schema name with which we want to connect, then we need to give username and password for our DB and we have provided the driver name as MYSQL driver.

**Config Server** :

Now we are having many microservices that will register themselves as a client on eureka server, and we have the same configuration for all of them, so this are just redundant and we need to write all this again and again, so now there are only few configurations we can write them again and again as well, but what will happen if we have 100 lines of configuration or even more that will remain same, so in this case we can put the configuration in a single file on our server and directly use this configurations directly in our microservices using the config server. So let us understand how can we do that.

Now to resolve this issue we will create a repository on **GitHub and create a file there application.properties or application.yaml and will put our configuration there for eureka client. And we will import the configuration from that file in a Config Server (we will make a microservice which will behave as Config Server) and then we will use that config server in our all other microservice**s to get those configurations.

So let us first create a GitHub repo and properties file there for our configuration.

A screenshot of a computer

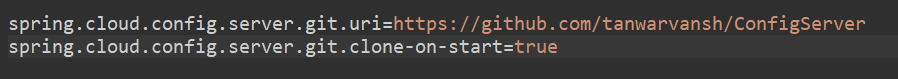
Description automatically generated

Ok so as we can see here I have created a repo with name ConfigServer and a file application.properties, where we will put our configurations. So we will create a microservice named as Config Server and need to add 'Config Server' and 'Cloud Bootstrap' dependencies and obviously it should have eureka client dependency as well as it also need to register itself on Eureka Server.

A black background with white text

Description automatically generated

So here, we have added Config Server dependency and now we need to configure the properties, let's do it.



So there are the properties that we should add when we are fetching the properties from the repo. So we gave URL of our GitHub repo. NOTE : We just need to provide repo URL only it will automatically fetch the properties file from there. Now it will fetch the **properties file from there and will implement those configuration here in the Config Server**. Another thing we did is **clone-on-start = true**, so this will automatically clone the changes from the repo at every restart to detect any changes.

Now as these all configurations are available in Config Server, we need to add these configurations in our microservice from this config server. So to do that, we will add a dependency in all microservices '**Config Client**' which will allow us to pull these configurations from this **Config Server into our microservices**.

A screen shot of a computer

Description automatically generated

So we have added the dependency for Config Client, now we need to add property to fetch the configurations from the Config Server.

A grey and orange text

Description automatically generated

So we just need to add this configuration, we need to provide the URL of the Config Server, and the Config Client will fetch those configuration and implement here in our microservice. Here if our Config Server is down or there are some issues in mapping/URL then it will throw the error and application won't start in this case, but if we want it to be optional that is, if server is not available then application should start without fetching these configuration then we can optional in the start of the value .e.



Now even though if server is not available it will treat it as optional and will start the application without fetching those properties.

Now our microservices are running and all are registered in Eureka Server and we can see their status in Server Registry. Now let's say if we need to call any API in any other microservice from a microservice then how can we do that, let's say we need to **call some API of Product microservice from Order microservice**, then how should we do that, so there are basically two ways to do that, one is using **Feign Client and other is using Rest Template**, so let us first understand and use the Feign Client and later we will see the Rest Template also.

**Feign Client** : Feign Client allows us to call APIs from other microservices, this will allow us to create the interface having the methods of those APIs which we want to call and then we can directly call those functions and we will get required output. Let us understand it with an example. So we are taking the same example that we used above, so we will call an API of Product Service from Order Service. So first of all we need to add **Feign Client(OpenFeign) dependency in Order Service**.

A screen shot of a computer

Description automatically generated

So here we have added the Feign Client dependency, now we need to **Enable feign Client** in Order Service.

A screen shot of a computer program

Description automatically generated

So now we have enabled the Feign Client in Order Service, now we will create a separate package as external that will tell us that this is an external package means it isn't part of Order Service but we are using it for some reason. Then we will create a sub package as 'client' so this will tell us that we are having client related things here.

A screenshot of a computer

Description automatically generated

Now we have created the package, so now we will create an interface here as ProductService that will tell us that we need to call product APIs from this interface. We need to mark this interface with **@FeignClient** annotation that will tell us that this will work as FeignClient. And inside that annotation we need to provide a name, so in the name we will provide the name of the P**roduct Service that we used in application.properties in Product Service and then need to give mapping if we had put any on Product Service Controller. Then we need to just copy the function which we want to call from Product Controller to this interface without body**. We need to copy signature only not body.

A screen shot of a computer code

Description automatically generated

So here what we did is we have enabled @FeignClient and inside the name I gave **ProductService/product** because in Product Service properties I gave name as **ProductService and for its controller I gave request mapping of '/product'** that why we put name like this. And then we just copy paste the signature of this method, this method we want to call from Order Service.

Now we can direct Autowired this interface in our service and we can call this function direct over it's object and we will be able to call the API.

A screen shot of a computer program

Description automatically generated

**Adding CustomErrorDecoder** :

So now this is working fine, We are able to call the API and get the expected results. Now the issue is what will happen if there will be any error thrown by Product API, so in our backend we will get the handled error result only but it will send back the Internal Server Error in response back. So to resolve this issue we need to implement **CustomErrorDecoder that will implement the Error Decoder**. And we will implement its abstract method decode(String methodKey, Response response). We will now throw our custom exception from this function.

A screen shot of a computer program

Description automatically generated

Here we return the new **CustomException this is our custom exception class**.

Now we need to handle this error and need to send back a response so we will use a **custom class with @ControlAdvice as we used earlier also to handle this exception and instead of this exception sends back a understandable message.** Let's do that.

A screen shot of a computer code

Description automatically generated

Now it will send back ErrorResponse instead of the error. We need to tell **Spring also that use CustomErrorDecoder instead of ErrorDecoder because by default spring will use ErrorDecoder to handle and decode the feign errors**. Let's create a config file for that.

A computer screen shot of text

Description automatically generated

By this config file Spring will use **CustomErrorDecoder instead of ErrorDecoder**.

**Rest Template** : Now let us see an example of calling the API using Rest Template, so first of all to use Rest Template we need to define its been so let's first do that.

A screen shot of a computer program

Description automatically generated

So here, we have define the bean for rest template in our Main Spring boot application. Now we need to autowired it and use it in our service class to call the APIs, let's do that.

A screen shot of a computer code

Description automatically generated

So here we called an API on Product Service from Order Service using Rest Template, let's understand what we did, so we have already created an object of Rest Template so there is a method **getForObject() here we need to give two values one is URL for the API and the second is class for Response( means the result it will send back) so here we passed the URL + id as path variable and then we pass the ProductResponse.class** in which we are expecting the its output. If we want to **use post request then we can use postForObject() it took three values, first URL, second Request Body, third Response Class** . There are many other methods that Rest Template supports, we can explore over the internet.

**API GATEWAY** : Whenever a client send a API request that request will always go to API gateway and not directly to out microservices, API gateway will then further contact with microservice according to the requests and will return the response. It's simple that client or open request can directly access our microservices, it need to connect with API gateway and API gateway will contact with microservice. It may be possible that it need to contact with multiple microservices so API gateway will connect with all required microservices, will gathered the response from all and then will send back the responses accordingly.

**Adding Zipkin for Logs Tracing** : Zipkin helps us in tracing the logs, as in a real prod application there can be 1000 or more microservices running so how can we trace the logs or how can we debug any issue. So for that purpose we will use Zipkin, to install Zipkin there are many ways, we can directly install it using docker by giving the command : 'docker run -d-p9411:9411 openzipkin/zipkin'

Now after giving this command it will download all required things and will run its container on localhost:9411.