RabbitMQ:

struggled to know how it works, then Eng. Alaa. told us not to work on their free online service, but to do it locally, because maybe the network will prevent its connection.

after the installing, I found that it requires another program to work with, and finally i got to the point that the RabbitMQ is started but i couldn't access it! So, after searching I found:

there is a command to run that will make the service configured and up for use.

then i started by configuring it out of the official website documentations. all we need is

@EnableRabbit: annotation for the whole application.

Sender Configurations: to determine the queue to use.

Queue Sender: to use the queue and send messages with it.

Queue Consumer: to receive the messages and consume it.

I started by writing logic straightly inside of the receiver, but faced many errors, and apparently the queue doesn't stop processing the message even it throws an error, so it's an infinity loop of errors.

Then I started by checking the input in the Resource.

MySQL DB:

I already studied DB before, so i didn't struggle there, but started with studying the given Scheme, and converting it to SQL queries.

I faced a naming issue when I used spring MySQL connector, at the camelCase convention, but later i change it to kebab_case and that solved the issue.

Entities:

@Data: The Lombok annotation to give my class auto generated getters and setters and constructors without having to write them manually or editing them when editing the class attributes names.

@Entity: Used to let the system know this class is an entity.

@Table (name = "table_name"): Used to let the system knows that this entity named in data base with name table_name.

@Column: Used to specify the name of the column for this class attribute, needed if the attribute name is different from the real field name of the table.

@**OneToMany(mappedBy="class_attribute_name"):** Used to let the entity know that this List is another entity, and they relate to each other, and mappedBy, was little tricky for me because i didn't recognize that it's the class attribute name, i thought it could be the column name which use in the fk.

@ManyToOne: Used to identify that this attribute relates to another entity.

@JoinColumn(name="fk_class_attribute"): Used to identify which column to use it as fk.

@JsonIgnore: Used to let the entity know to ignore this attribute in case it's sent to the user as json response from the Resources.

Repositories:

@Repository: Used to identify that this interface is repository.

We extend the JpaRepository<ParameterType, Long> which make it easier for us because it contains punch of methods like find, findAll, findById and many more, also there is an ORM which we can type queries with English words according to some convention like “findByName" and it will generate its code!

Services:

I found that we need to use a service layer, it looks like the encapsulation of the OOP, to control what the user gets and how to change the return of any method to be more suitable for the user. also, it'll make it easier to make some methods ready to use instead of implementing them whenever and wherever i want to use them.

@Service: on the implementation classes of the services layer, we use this annotation to identify that this class is a service.

Resources:

ActionResource: there is 2 main end points

findActions: Made to receive a GET request with many parameters to be used in searching all over the actions, after taking the request I start processing it and building the SearchDto to be used later in the queryDsl repository.

addAction: Made to receive POST request with a json body, then validating it, if it passed all the way, i publish it in the queue else, it'll throw the Exceptions.

SelectorsResource: just one end point

findSelectors: Made to receive a GET request and return all the selectors needed for the frontend.

Mapstruct:

@Mapper: used to identify that this class is a mapper of the mapstruct.

@Mapping: used with many properties, like target: which used to specify the target attribute, and source to specify the origin attribute, expression which is used to give a programming expression and to be evaluated when the mapstruct generate the mapper implementation.

@Context: used to identify the needed variables in the mapping expressions.

Lombok:

@Data: The Lombok annotation to give my class auto generated getters and setters and constructors without having to write them manually or editing them when editing the class attributes names.

I found that there are many problems between mapstruct and the Lombok and after searches on google found that there are specific versions run the best with specific Lombok versions.

Instead of letting it for the luck i made the getters and setters manually in the classes.

QueryDsl:

I used it to make the searching of actions more easier to be generic, by checking on the already built searchDto from the actionResource and its fields, I add accordingly the where queries on the main query, and in the case of searching with the parameter type, and value, I faced a logical error, which is my searching query search in actions 2 times separately, 1 for any action that have any parameter with needed parameter type, and 1 to search for the returned actions about which one got any parameter with that parameter value, each one was separate, instead it should searched with the action that have a parameter with that parameter type and that parameter value. I figured out we need to join the parameters table to our main action table and start searching in that joined parameter table about the needed parameter with the parameter type and parameter value needed.

Adding New Actions:

it's a process starts by validating the json body at the actionResource, then validating it once more at the consumer of the queue if it pass, then we start converting the coming actionRequestDto back to the action entity to be saved, then generating the description of this action with the action type template en/ar filled with the parameters sent in the request, after generating those, we start saving the parameters came in the request by converting them back to parameters and assign their parameter types until a parameter type entity is ready to save.

Template Description Converter:

the template contains the {{name.type}} name could be one of the parameter types name, user, application or BE. and type could be value, name or id, so i add the user and application and business entity info with the parameters came in the request and then looping through all of them and checking if they are found in the template or not and replacing them.

I used the stream().forEach() but i got an error of the must use static variables inside of the forEach lambda expression body, so i turned to use the simple for to loop over the parameters.

Validations:

started to validate the input, with many test cases by providing invalid inputs, not found inputs, invalid json body and absence of attributes from the json body, then started to identify all those errors and i made custom exceptions to be more meaningful.

TODO IN FUTURE: provide error coding for the exceptions as i have seen in the big projects

Object Mapper:

I used it to convert the string to json object then to actionRequestDto at the actionResource, and at the queue consumer.

why i didn't receive it as the DTO from the beginning, as far as i learned about the RabbitMQ that you must send a string, you can't send a class object or something, so to use that string in the consumer! we need that object mapper.

DTOs:

It's more powerful to use DTOs and it's like making a view in the database systems, i could change the fields names, and combine more than one class, and it's made easier with the usage of Lombok and mapstruct.

Testing:

I made testing for the service layer, where i used Mockito to mock the repositories and control their behavior, also used AssertJ to produce more readable and flexible good assertions.

Also i tried to test the search about actions with querydsl at the repository layer, i faced tons of errors but at the end i found that i can't use the h2 for testing as a db because my tables might be named with some reserved words in h2, and even when i tried to test over the real db by disabling the database replace in the autoconfigurations, i couldn’t right good tests, all what came to my mind that I’ll make a test for each combination of the searchDto attributes, which is a very bad solution. so, i couldn't do it.