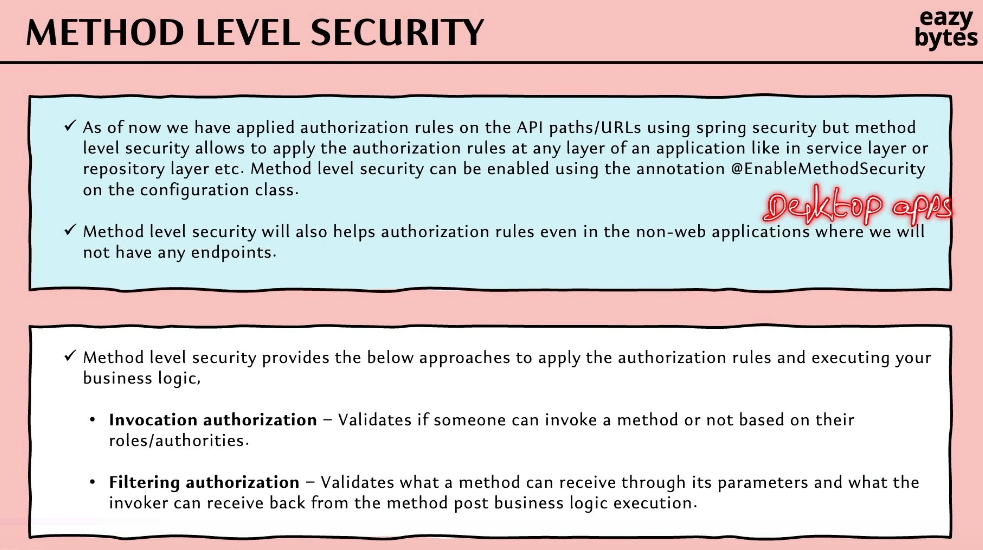
**11.method level security section12**



it supports two different approaches on how we are going to apply the authorization rules on top of a Java method or on top of a business logic.

The very first approach is Invocation authorization.

As part of this Invocation authorization, spring security is going to help us to decide

whether to invoke a method or not based upon the authorization rules that we are going to define. That's why inside the name of the approach,

you can also see invocation. So here, invocation refers to the invoking a method.

The next approach is Filtering authorization.

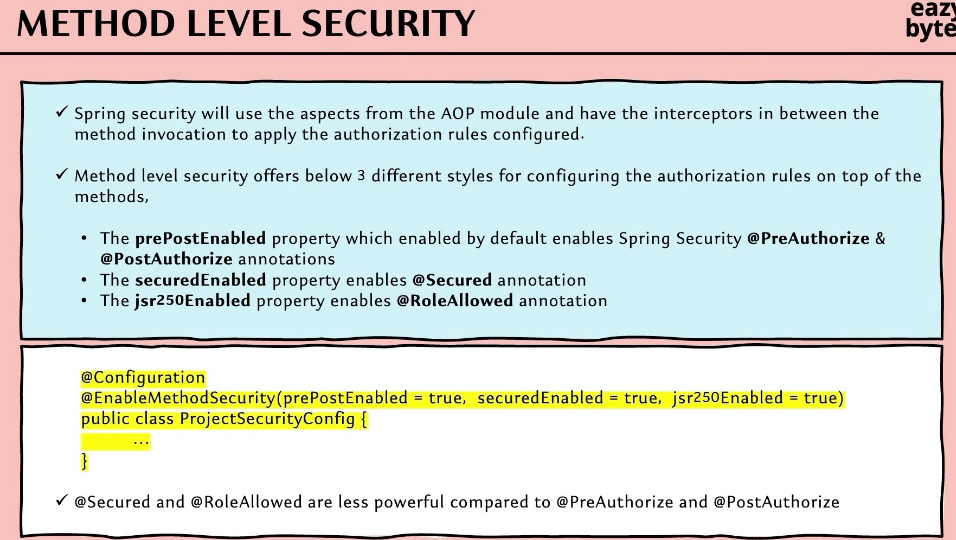
We know all our Java methods, they're going to accept lot of input parameters

and they also going to return lot of data as part of the response.

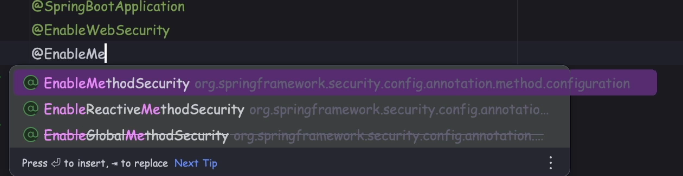
With the help of these Filtering authorization approach, we should be able to perform validations and accept the data only if the data satisfies to our filtering criteria

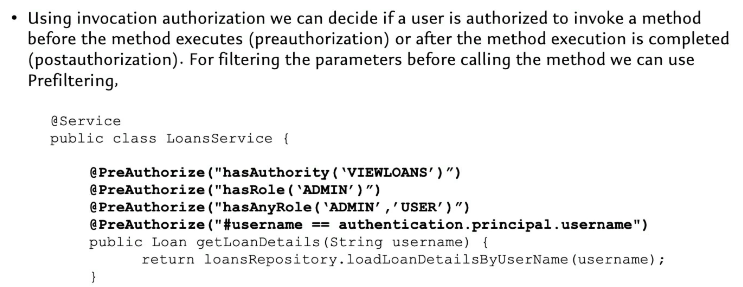
or to the authorization criteria that we have mentioned.

Similarly, the return type of the method also can be controlled based upon the filtering condition or based upon the authorization condition that we are going to define.



 Details about method invocation authorization in method level security





But in reality, you are not allowed to use this annotation multiple times on top of a method.

It is going to perform the validations, whether a given end user has enough authority or roles to invoke this method. If the end user does not have given role or authorities,

then Spring Security, it is going to throw the 403 error.

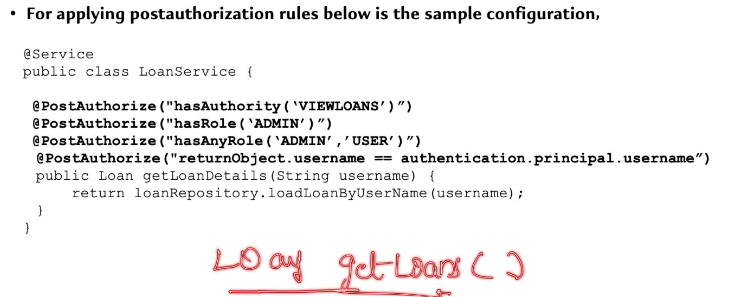
Invoke this method if the username is equal to the logged in user username;

that means only the logged-in user should be able to fetch their own loan records

due to these authorization check that I'm trying to enforce.

If someone is trying to manage to invoke this method with a different username,

they will not be able to invoke it due to the method-level security that we are trying to enforce here. So this is one style of enforcing the invocation authorization.



The other style, like I was telling, it is with the help of @PostAuthorize annotation.

This @PostAuthorize annotation, it is going to work exactly opposite of @PreAuthorize annotation.

When we enforce an authorization rule with the help of @PreAuthorize annotation,

the Java method is never going to be invoked if the authorization check fails;

whereas when we're using the @PostAuthorize annotation, the Spring Security framework is not going to perform any authorization checks during the method invocation.

Once the method invocation is completed , while returning the output from the method, at that point of time, the Spring Security framework, it is going to enforce the authorization checks that we have mentioned here.

Why anyone want to invoke a method without authorization but enforce authorization while returning the output?

Sometimes during the method invocation, you may not have enough information

to enforce the authorization. But once the method execution is completed

and once the method is trying to return the object, using the return object, you want to enforce authorization.

In such scenarios, we can use the @PostAuthorize annotation.

There is no meaning to use @PostAuthorize annotation with the hasAuthority, hasRole, and hasAnyRole approaches, because these approaches we can happily use with the @PreAuthorize annotation. But technically, it is possible to use these styles with the help of @PostAuthorize annotation.

But if you try to look at the last syntax or last two configurations that I have,

this is the scenario where it is going to make more sense to use @PostAuthorize annotation. So here I'm trying to enforce the authorization based upon the return object that I'm trying to return here.

So whatever return object that my method is going to return, it is going to be referred as return object inside my authorization configurations. So if the logged-in user username is same as with the username present inside the return object, then only we want to return the object. Otherwise, we want to stop returning that object

by throwing the 403 error.

Think, like, I have a method which is getLoans().

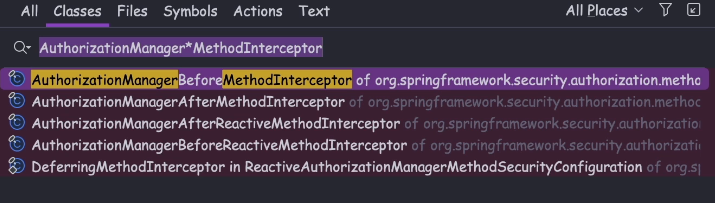
And this method is not going to accept anything, but it is going to return list of loans as an output.

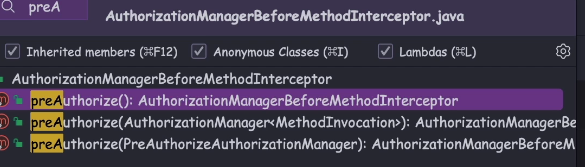
Here I can't enforce authorization based upon the username using the @PreAuthorize annotation, because this method, it is not accepting any input parameter;

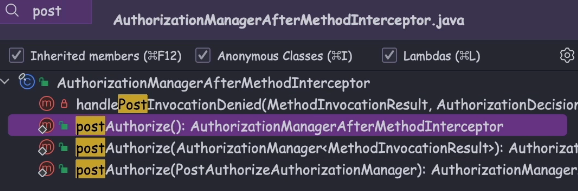
Whereas by inspecting the return object, I will be able to apply the authorization checks with the help of @PostAuthorize annotation.

The AOP concept is being leveraged by the Spring Security framework here

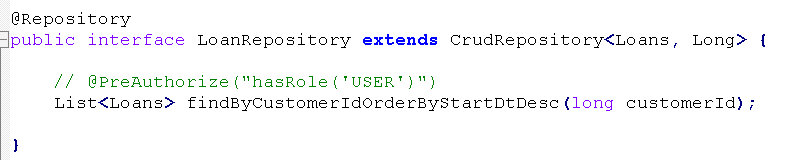
to decide whether a method should be invoked or whether a output from a method should be returned or not.

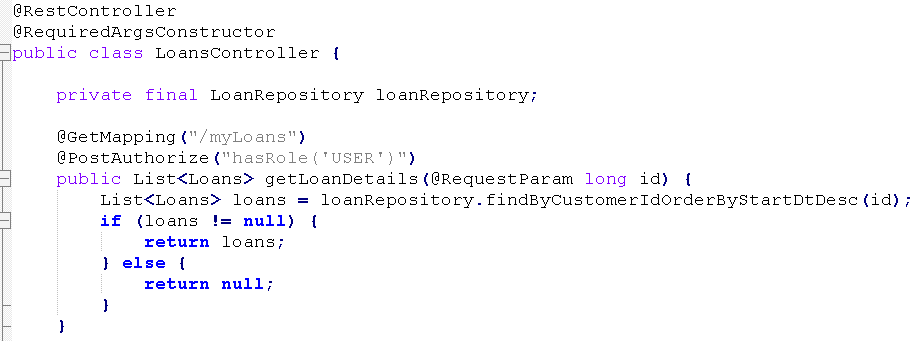




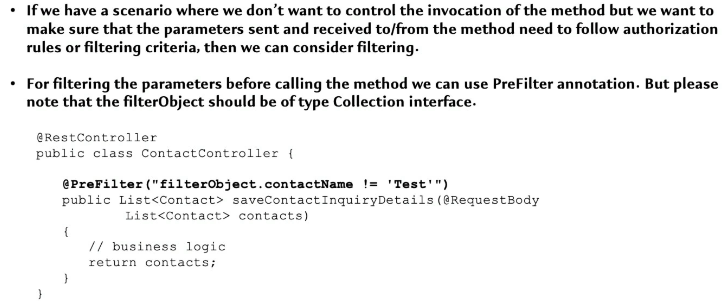


Demo of method level security using @PreAuthorize and @PostAuthorize





 Details about filtering authorization in method level security



With the help of this Filtering authorization, we should be able to control what kind of input that a method can accept and what kind of output that a method can return

to the client application.

So this method is going to accept list of contact objects unless my method is not accepting a list or a collection object, I cannot enforce the Filtering authorization.

That's why as a very first check, please make sure that your method is accepting

a collection object as an input parameter. So inside this collection object,

we are going to store the contact related objects.

So whatever expression that I'm going to write, with the help of @PreFilter annotation, it's going to be applied for each object that I have inside this list input that I have received. Every object that we are going to receive as part of the collection object, we're going to refer it with a standard name, which is filterObject.

So here the filterObject is going to be of data type Contact because we're trying to accept list of Contact as an input. So inside this Contact to Pojo class, there might be a field name with the name, contactName. So if this contactName is not equal to test, then only I want to accept that Contact object as an input inside my List object.

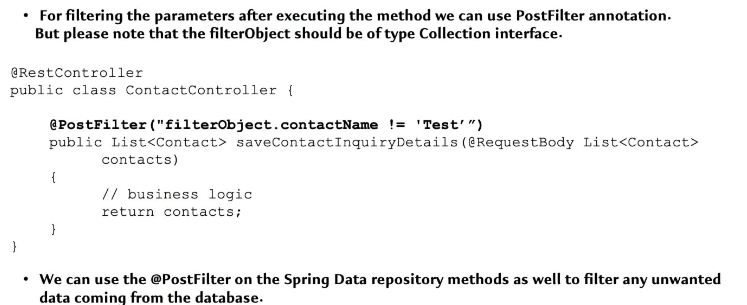
Otherwise, if someone is trying to send a contact details with a contactName as test,

I'm going to simply ignore such records and I'm not going to send such record

as an input to this method.

So this is one style of Filtering authorization.

If you want, instead of this test, you can write the logic to check the current authenticated username.



With this post filter, what is going to happen is inside these list of contact response

that I'm going to get from this method, if there is a Contact object with a contactName as test, then search record will never be received by my client application.

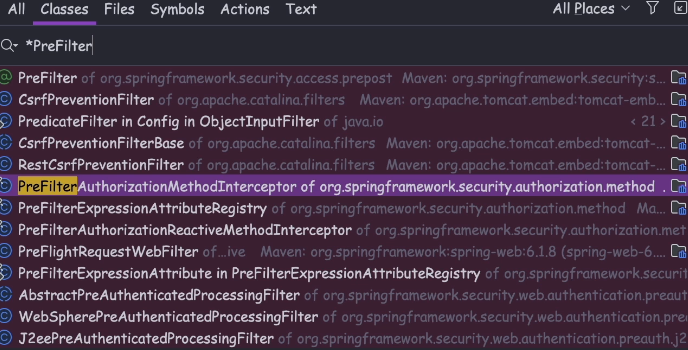
If required, we can also use @PreFilter and @PostFilter annotations, both of them on a single Java method.

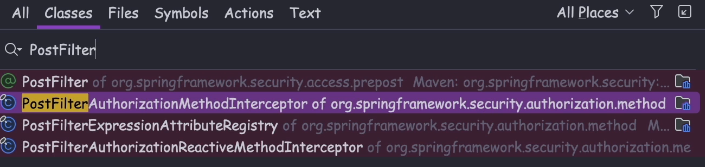
The same applies for @PreAuthorized and @PostAuthorize as well.

If you have a business requirement, we can club both @PreAuthorized

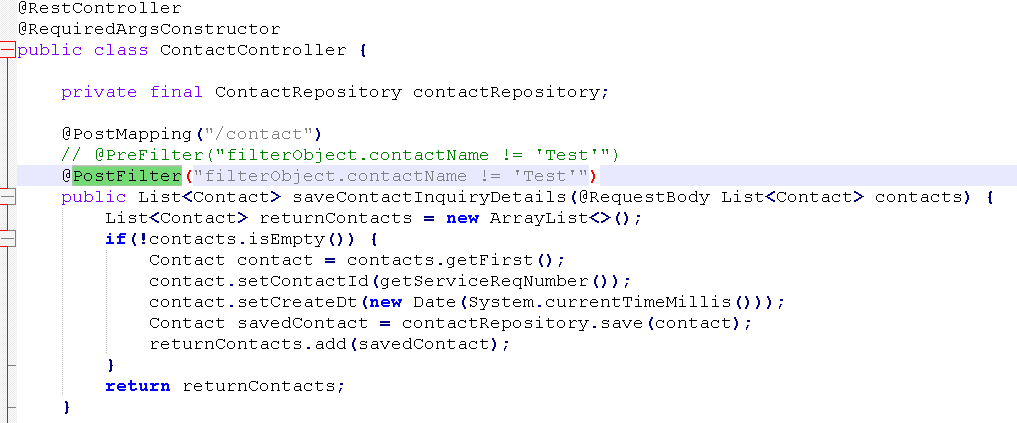
and @PostAuthorize on top of a single method.

But the same annotation is not allowed to use multiple times on top of a Java method.





Demo of @PreFilter / @PostFilter annotation



Top of Form

Which of the following annotation should be used on top of the configuration class to enable method level security using Spring Security framework?

**@EnableMethodSecurity(prePostEnabled = true, securedEnabled = true, jsr250Enabled = true)**

Given the following method-level security configuration in Spring Security, which annotation ensures that the method can only be accessed by users with the role "ADMIN"?

1. public class AdminService {
3. @\_\_\_("hasRole('ADMIN')")
4. public void performAdminTask() {
5. // method implementation
6. }
7. }

@PostAuthorize

Top of Form

Which annotation in Spring Security is used to filter the returned collection or stream based on a security expression?

**@PostFilter**

Bottom of Form