Row and column security with ACL filtering in Java Spring

(filtering and securing Hibernate entities)

1.1 General overview

Row security with ACL filtering is already implemented in Java spring with the @PostFilter and @PreFilter annotation and the SpEL expression hasPermissoion. However, a need may arise to filter out the columns too as an additional security measure using the available ACL information. For example, we might have an entity, containing sensitive information about a person, like social security number, and telephone number that can only be accessed by certain users, or certain roles. Moreover, we may need to specify the permission the users or roles need to have, such as BasePermission.READ, BasePermission.ADMINISTRATION or any type of permission (including custom permissions) and the security identities (user or role) in order to be able to have access to the column. In order to do this, custom advice has to be created, because we need access to the permissions of each row, firstly to determine that access to the row is granted, and then to compare the defined requirements for column security with the same granted permissions to determine if access to a certain column is granted. This means that row filtering and column filtering both have to be implemented together. The implication of this is that the PostFilter or the PreFilter cannot be used i.e custom implementation of the same features is needed. I have developed a few solutions for filtering the return result (equivalent to PostFilter hasPermission plus column filtering). However the same can be done for filtering the arguments of a method (equivalent to PreFilter hasPermission plus column filtering) very easily just by changing the code relating to acquiring the collection source.

The first problem that has to be considered in order to determine the way we approach the implementation is the way Hibernate handles projection. When using Hibernate's criteria and Hibernate' projections to select one or multiple columns, the return result can only be cast as a list of arrays of objects. Each array of objects (each element of the list) is either an attribute or an aggregation function result. Therefore, there are two options available regarding this issue:

- The projection is done using Hibernate's criteria and Hibernate' projections in the DAO layer, which means that the service layer is aware of the projection, and it returns a list of arrays of objects. This means that before the method is called, the criteria (required attribute) has to be initialized so that it filters the columns by the SIDs (principal or role) the user grants access to in a custom annotation to each column. The permissions for each row are obtained after the method returns, so additional filtering has to be done after the method returns in order to compare the permissions for each column.
- The DAO method returns a list of the entities and is not aware of the column filtering, the advice handles the filtering removing the proxy from the returned entities and in case access if granted for the object (row), by setting the forbidden fields values (if any) to null if they are Object subclasses

and if they are primitive values, to some minimum value or false in case of boolean meaning that setter methods have to be present.

The second conceptual problem is if we should assume that each instance of the same class (entity) has the same permissions assigned to the same SIDs. So far we have assumed the opposite. If we reexamine the previous two approaches based on the new assumption, then it wouldn't be necessary to obtain the permissions for each object. Instead we can obtain the permissions for one row, which will result in a much better performance. However, it will also reduce the flexibility and the applicability of the solution.

All four approaches will be analyzed in detail in the following sections along with their advantages and disadvantages.

1.2 Custom annotations and proxying

I have created two custom annotations for this purpose:

- @ColumnSecurity- field annotation
  This annotation has two arguments usersOrRoles and typeOfPermission.
- @RowColSecurity- method annotation
  This annotation has one argument typeOfPermission (used for row filtering).

Note: If the types do not match, access to column can still be granted, because the type of permission attribute in column security is compared with all the granted permissions for a certain row.

A proxy with the dao as target class and advice class that implements MethodInterceptor is created. In the invoke method, lies all the filtering and securing logic. The intercepted method has to meet one prerequisite: The DAO method has to be annotated with @RowColSecurity annotation.

Improvement that should be done: We usually need to get a list of entities for a number of purposes including reading, editing or removing them. Therefore if we annotate the DAO class with @RowColSecurity(typeOfPermission=BasePermission.READ), we cannot distinguish between different purposes of the service layer methods, that utilize this particular DAO method. Consequently, the RowColSecurity 's typeOfPermission ideally should be dynamically adjusted to match the permission of an additional custom annotation used to annotate methods in the service layer.

1.3 Removing proxy from objects and setting field values approach

1.3.1 Different permissions assigned to different SIDs

Firstly, the user and its roles are extracted from the Authentication object. Then the target class is scanned for the method that is intercepted and if the RowColSecurity annotation is present (by checking the interfaces of the annotation objects), the filtering is done. If the RowColSecurity annotation is not present, the result is returned as it is (a list of entities, no filtering).

The filtering is done by iterating the collection (using java reflections to invoke the interator() method) and repeating the following steps for each object:

- Invoking the object's getId() method using Java Reflection (OID recovered)
- Try to get the AclImpl object for the OID from the `org.springframework.security.acl.domain.EhCacheBasedAclCache.` If it is not there then get the AclImpl from the MutableAclService class.
- Get the list of AccessControlEntryImpls from the AclImpl
- Check if both the entry' SID and the entry's permission match the user or the role and the RowColSecurity's permission
- If they do then we proceed to filter out the columns too, else the entity is removed from the list
- The class of the collection's object is obtained, as are all its declaredFields. Then similarly the interfaces of each field's annotations are checked if they match the ColumnSecurity. If the annotation is not found the object is added to the new collection of objects of the same type. If the annotation is found, the usersOrRoles' s elements are compared with the logged in user and its roles.
- If a match is not found, the object is unproxied (removing the HibernateProxy is necessary because any changes to the object will become permanent)
- Then the setter method of the appropriate field is invoked (null for no primitive types, min for primitive, false for boolean) and the object is added in the new collection

1.3.1.1 Advantages

The main advantage of this method is that what we get in return is a list of entities that we can easily work with, pass them up to the presentation layer, and render them. This is not a projection by the book, but it facilitates a lot of things afterwards.

Also it is easy to recover the OID from the object, whereas in the other case, we have to establish an additional convention so that we can extract the oid, such as the first object of the array of objects is always the OID or other alternatives that will be discussed later, and a few other rules.

This implementation is more intuitive, and fewer details for the implementation have to be known which is very important, since we strive to produce reusable code that can easily be understood and be more effective that way.

Another advantage is that it works with all classes that implement the collection interface.

1.3.1.2 Disadvantages

The main disadvantage of this implementation is that it doesn't filter the collection with a query, so it takes more time to execute, and along with the additional logic to filter out the columns and the creation of a new collection with unproxied elements contributes to a very bad performance. Therefore this solution is very ineffective and costly.

1.3.1.2 Performance

Num	RowColSecurity	Field	Column Security	Rows	Execution time(seconds)
1	*typeOfPermission="BasePermission.READ"*	Sms_id	UserOrRoleNames={"tamara","ROLE_PRIVATE"} *typeOfPermission="BasePermission.READ"*	3765	*4.63547324*

1.3.2 Same permissions assigned to same SIDs

The difference in this case is that the permissions and the sids extracted from acls for each element are extracted only from the first element.

1.3.2.1 Advantages

The advantage of this implementation is that has better performance than the previous one.

1.3.2.2 Disadvantages

The disadvantage is that it is less applicable, as discussed above. The most problematic part probably is maintenance. What if we want to change the logic, assign different permissions to users? If we need to keep the original acls as they are without changing them, a non-resolvable problem arises. In that case this implementation is not suitable. However if want to change the original acls , then all we have to do is remove the entries for them in the acl tables, and iterating over every object in interest , get their oid, and create a new set of acl entries for each oid.

1.3.2.3 Performance

1.4 DAO layer filtering with Criteria approach

1.4.1 Different permissions assigned to different SIDs

Firstly, the user and its roles are extracted from the Authentication object. Then the target class is scanned for the method that is intercepted and if the RowColSecurity annotation is present (by checking the interfaces of the annotation objects), the filtering is done. If the RowColSecurity annotation is not present, the methodInvocation.procees() result is returned .

- The intercepted method has to have parameters types Class and Criteria
- The Class object has to be recovered (methodInvocation.getArguments()) so that the criteria can be initialized accordingly with the entity class that is passed as argument. The criteria object has to be passed as argument because before invoking the method,

the projection list has to be set. Since there is no way to find out which entity/table is queried before we invoke the method (the method returns a list of arrays of objects therefore giving no indication as to which entity the attributes belong to) the class objection also has to be passed as argument, or alternatively a new attribute entity can be added to the RowColSecurity. In this implementation I have chosen the first one, but regarding the performance this shouldn't make a difference since it executes only once and both approaches have similar complexity trivial to the overall performance.

- Next, the ProjectionList of Projections has to initialized, according to the ColumnSecurity's fields(userOrRoleNames and typeOfpermission). The filtering can only be done considering the usersOrGroups property, since we cannot compare the permission with the granted permissions (the granted permissions can be recovered only after the method returns). All the fields in the Entity class that are not annotated are added to the list of projections (the opposite can also be done, however this approach is more intuitive).
- Then the criteria's list of projections is set
- The method is invoked with its original arguments (with one exception, the criteria we just created is given instead of the original, that can be either null or not). The method returns a list of arrays of objects.
- In order to extract the oid for each row in this case we can either establish a rule that the first object of the list (the first column, projection) is the oid, or suppose that the first column is the id, then get the object by id, and then invoke the getId() method or use another custom annotation (@OID). I have chosen to last approach because it is the least expensive from the three and it is the most flexible. That way we can have a variable indexOID that will be appropriately updated. The aceImpl are obtained and the same procedure as before is executed.
- If the user is granted access to the row, additional column security has to be done by comparing the permission (ColumnSecurity' permission) with all the user's granted permissions. For this purpose I have created an array of all the column names (while filling the projectionList) and a hashmap <methodname, permission (ColumnSecurity's permission)>.
- If the column is annotated with ColumnSecurity the permission is matched against all the permissions for the oid .

  Improvement: filter out some of the permissions, match the permission only with the entries that have both a matching SID with the ColumnSecurity's usersOrRoles
- If access to one or more columns is not granted, then a new array of objects is created without those objects, and added to a new list of arrays of objects. Otherwise the same array is added to the new list. This has to be done this way since, there is no way to add and remove an element while iterating.
- As a result the new list is returned.

1.4.1.1 Advantages

The best thing about this implementation is that the filtering is done in the DAO, therefore dramatically increasing the performance (much faster even than the @PostFilter and @PreFilter implementation). This is a great achievement because it makes the effort of enforcing both row and column security worthwhile and much more effective than row security only. However it does have a cost.

1.4.1.2 Disadvantages

The main problem with this approach is the manipulation with the list of arrays of objects which is far more tedious and far more complicated than having the entity available. The presentation is far more difficult since we do not know which columns are returned.

Improvement: Return an array of columns names along with the list of objects, or better the first row of objects is an array of Strings of the name of the columns

Another disadvantage is that the presentation layer, the service layer and the dao layer all acutely aware of the filtering, so transparency isn't this solution's best suit. This solution might be less elegant it is by far a very effective one.

1.4.1.1 Performance

Num	RowColSecurity	Field	Column Security	Rows	Execution time(msec)
1	*typeOfPermission="BasePermission.READ"*	Sms_id	UserOrRoleNames={"tamara","ROLE_PRIVATE"} *typeOfPermission="BasePermission.READ"*	3765	*434.397359*

1.4.2 Same permissions assigned to same SIDs

If we suppose that the same set of acl entries is persisted for every oid, then we only have to extract the granted permissions once. The rows will all be filtered, and accordingly if access is granted to the row, the columns are filtered too, and a new set of arrays of objects is created.

1.4.2.1 Advantages

The main advantage is that the execution time is even better than the previous one which yielded excellent results as it is. This solution is an extremely efficient one.

1.4.2.2 Disadvantages

Additional disadvantages regarding the previous solution are the same disadvantages when we assume same permissions (decreased applicability etc..)

1.4.2.3 Performance

1.5 Comparison with @PostFIlter

1.5.1 Code and Configuration

Java Spring PostFilter relies on a couple of beans that are initialized when they are needed, which is reflected in the execution time. In the four solutions above I create the MethodInterceptor in my xml file which is parsed during start up, meaning that this bean is created and ready to be used before the need arises. This is also one of the reasons why the performance results for the last two solutions are so good.

1.5.2 Testing and maintenance

1.5.3 Security

1.5.4 @PostFilter performance (securing Hibernate entity)

Num	PostFilter expression	Rows	Execution time(seconds)
1	*hasPermission('ROLE_ADMINISTRATION')*	3765	2.825732308

1.6 Improvement options

The column security I introduced is based on the ColumnSecurity field annotation, which means that we cannot deny access to a certain column in one instance, and grant access to the same column in another instance of the same class. A certain level of granularity already is achieved with the previous four implementations. However an even more granular approach can be achieved by making the previously discussed option available. The main issue is all that trouble is worth it. We should be aware that it will affect the performance for sure because in order to achieve this we have to extend the acl design by adding new tables, and make a custom implementation of the already existing spring classes that deal with retrieving the access control list information. It all comes down to one thing: do we need such level of granularity and are we willing to pay the price for it?

The purpose of these solutions is to take the security to the dao level…

Also PostFilter SpEL offers expressions that are much more powerful and offer more features in comparison with these custom implementations. The purpose of these implementations was primarily to introduce column security (with UserOrRoleNames OR is assumed). However once perfected, another possible improvement is make these implementations be able to evaluate expressions, and implement that logic with in a custom voter with additional functionalities and methods that will be called for each object.