SOL Software d.o.o.

www.sol.rs

Public

SOLoist Data-Model-Based Authorization

Project: SOLoist V4
Document Type: Project Documentation (PD)
Document Version: 1.0
Date: 01.06.2015

Introduction

This document describes SOLoist V4 *authorization framework* (introduced in SOLoist 1.0.8). It allows SOLoist developers to define access rights on the level of UML model elements.

Authorization

One part of the job of specifying authorization is done in soloist-config.xml file and one part in SOLoist Java classes (generated from the model), like *Person*, *Document*, etc.

The first part is configuring SOLoist authorization framework in soloist-config.xml file. There are three Boolean elements that define behavior of authorization and should be placed in soloist-config.xml. These three elements are explained in the table below.

Element Name	Element Type	Default Value	Description
authorization	Boolean	False	Is authorization turned on (true) or off (false)?
default-authorization-result	Boolean	True	Enables two authorization strategies. The first (true) assumes every access is allowed unless otherwise explicitly specified. The second (false) assumes every access is forbidden unless otherwise explicitly specified. This element matters only when authorization is turned on. See below for more details.
use-authorization-cache	Boolean	False	Whether to use (cache-based) performance optimization for authorization? See Authorization Cache section of this document. This element matters only when authorization is turned on. See below for more details.

For example, in order to turn the authorization on, put this into the soloist-config.xml:

<authorization>true</authorization>

To use allow-access-unless-otherwise-specified strategy:

<default-authorization-result>true</default-authorization-result>

And finally, to use cache-based optimization:

<use-authorization-cache>true</use-authorization-cache>

Before authorization cache is understood and mastered (section *Authorization Cache*), it is recommended to set the parameter *use-authorization-cache* to false.

After the step of configuration, authorization rules should be specified. These rules tell SOLoist what type of access to objects and their properties is allowed for which user or user role. Obviously, setting these rules makes sense only if authorization is turned on in the soloist-config.xml.

Authorization rules are specified on the basis of a SOLoist domain class. Namely, in order to define access rights for objects of a certain class, one should implement (override) specific methods in that particular class. More precisely, one should override one or more methods out of the four methods declared in *IClassifierInstance* interface (every SOLoist class already implements this interface). Overriding these methods should be done for each SOLoist class that has specific authorization rules. These methods tell SOLoist whether to allow or disallow access to SOLoist objects of this class (or their parts, i.e., slots). By default, they return the value of the *default-authorization-result* parameter (see above). In other words, if not overridden, these methods return the default value specified in soloist-config.xml. According to that, only those methods that should behave differently than just returning the default value should be overridden.

For example, if *default-authorization-result* is set to true, and no methods are overridden, then all access is allowed, for each user and for every SOLoist object. When there is a need to forbid access for some users to some objects / properties, the corresponding methods should be overridden. This is the so called "allow-access-unless-otherwise-specified" strategy. On the other hand, if *default-authorization-result* is set to false, and no methods are overridden, then all access is forbidden, for each user and for every SOLoist object. In this case, authorization methods should allow access for some users to some objects / properties, and the corresponding methods should be overridden to allow that. This is the "disallow-access-unless-otherwise-specified" strategy. Sometimes it is easier to define authorization rules in one way, and sometimes in the other. This depends on the application domain. SOLoist allows for both approaches.

Before explaining authorization methods there is one precondition that must be met in order to have the authorization working. The developer should make sure that HTTP session (HttpSession object) has the following attributes set correctly:

HttpSession Attribute Name	HttpSession Attribute Type	HttpSession Attribute Value
username	String	The username of the currently logged-in user.
role	String	The role of the currently logged-in user.

Username and role values are important for authorization because they are provided to the above-mentioned authorization methods as parameters. This way, when implementing the authorization methods, the developer is able to decide whether a certain user with a certain role is allowed to perform a certain action or not.

For example, this code should be put in the domain class *Person* to define access rights for objects of that class:

```
@Override
public boolean isAllowedToCreateInstance(String user, String role){
```

```
      …
}
@Override
public boolean isAllowedToDestroyInstance(String user, String role){
      …
}
@Override
public boolean isAllowedToReadSlot(String user, String role, IProperty property){
      …
}
@Override
public boolean isAllowedToEditSlot(String user, String role, IProperty property){
      …
}
```

If authorization is turned on, SOLoist runtime will call these methods on every single user's attempt to create a new instance of class *Person*, destroy any existing object of *Person*, read any existing Person's slot, or write (in any way) any existing Person's slot, respectively. For example, if a user tries to read the slot *firstName* of a *Person* object, the method *isAllowedToReadSlot* will be called on this very object of class *Person* just before the reading of the slot occurs. The method will be called with the parameters *username* nd *role* of the user trying to read the slot, and with the parameter *property* having value *Person.PROPERTIES.firstName* indicating the target slot for the read action. (The user and role are determined by the corresponding attributes of *HTTPSession* in whose context the action is being performed.) If the method returns true, the read access to the slot *firstName* will be allowed. Otherwise, a n *ActionAuthorizationFailedException* would be thrown indicating a violation of access rights. The same stands for other three methods for creating and destroying instances and for writing (modifying) a slot.

As already said, every one of these four methods may be left out and not overridden. In that case there is a default implementation. The default implementation returns the value of the *default-authorization-result* parameter from the soloist-config.xml file.

Overriding these methods should be easy. Implementation of these methods should tell SOLoist whether to let the user of the given role (defined by parameters *user* and *role*) do what the method name tells. For example, "Is the user allowed to create an instance of class Person?", or "Is the user allowed to destroy this particular Person?", or "Is the user allowed to read the slot of this particular Person?", or "Is the user allowed to edit the slot of this particular Person?"

Important note. Since these methods are obviously NOT static, it is perfectly fine to check the host (this) object in these methods while deciding what is allowed and what is not. In other words, it is fine to check the host object, to read its slots, to read slots of the connected objects, and finally to make a decision based on all that information.

Reading the host object or its connected objects makes sense when implementing *isAllowedToDestroyInstance*, *isAllowedToReadSlot*, o r *isAllowedToEditSlot* methods because the authorization result may depend on the state of the host object. On the other hand, reading the host object does not really make sense when implementing *isAllowedToCreateInstance* method, because the host object in that case does not really have any state yet (it is in the process of creation, because the

authorization method is called during the execution of the create object action). So, without any state initialized yet, the host object cannot really contribute to making the decision of whether the user is allowed to create it or not. In short, when implementing *isAllowedToDestroyInstance*, *isAllowedToReadSlot*, or *isAllowedToEditSlot* it is fine to use the *this* reference; when implementing *isAllowedToCreateInstance* method, it most often does not make sense to use it.

There are two constraints that must be satisfied when implementing the authorization methods.

a) These methods *must not* modify any SOLoist object.
b) When reading the host object's slots or its connected objects' slots, the authorization methods must only use slot._read() or slot._val() methods. These two methods are replacement for usual slot.read() and slot.val(), since the latter include calls of authorization methods, and would thus imply endless recursions.

Regarding point a) it is not really an issue, because it typically does not make much sense to modify any object in the authorization methods. They are meant only to determine if certain access attempt should be authorized or not. Regarding b), the new _read() and _val() methods are introduced for technical reasons in order to avoid endless recursion.

Example of Use

The following example defines access rights for objects of class *ApplicationForMembership*. After that, an explanation for every method is provided.

```
class ApplicationForMembership {
      …
@Override
public boolean isAllowedToCreateInstance(String user, String role){
      return “enrollment”.equals(role);
}
@Override
public boolean isAllowedToDestroyInstance(String user, String role){
      return “admin-x”.equals(user)
            && Text.fromString(“Closed.Rejected”).equals(this.state._val());
}
@Override
public boolean isAllowedToReadSlot(String user, String role, IProperty property){
      if
      (ApplicationForMembership.PROPERTIES.paymentTransactionNumber.equals(property
      )){
            return “payment-officer”.equals(role);
      }
      return true;
}
@Override
public boolean isAllowedToEditSlot(String user, String role, IProperty property){
      Office officeOfReg = this.applicant._val().officeOfRegistration._val();
      User u = QueryUtils.findSingleInstanceBy(User.CLASSIFIER,
            User.PROPERTIES.username, Text.fromString(user));
      return u.myOffice._val().equals(officeOfReg);
```

```
}
}
```

The first method authorizes every user with “enrollment” role to create a new instance of class *ApplicationForMembership*. Other users are not allowed. The second method authorizes the user with username “admin-x” to destroy only *ApplicationForMembership* objects that have been rejected. The third method authorizes all users to read any slot of any *ApplicationForMembership* object, except from reading the slot *paymentTransactionNumber* for which only payment officers are authorized. The final, fourth method authorizes editing of any slot of a certain *ApplicationForMembership* object for all users from the office where the applicant of this *ApplicationForMembership* object is registered.

It should be noted that checking authorization rules has its effect on performance. The effect on performance depends on complexity of authorization rules and frequency of access to objects / slots that have authorization rules attached. Therefore, authorization framework should be exploited with the greatest care.

Authorization Methods’ Summary (excerpt from JavaDoc)

```
boolean isAllowedToCreateInstance(String user, String role)
```

Is the given user having the given role allowed to create an object of this class?

```
boolean isAllowedToDestroyInstance(String user, String role)
```

Is the given user having the given role allowed to destroy this object?

```
boolean isAllowedToEditSlot(String user, String role, IProperty property)
```

Is the given user having the given role allowed to change the slot (that corresponds to the given property) of this object.

```
boolean isAllowedToReadSlot(String user, String role, IProperty property)
```

Is the given user having the given role allowed to read the slot (that corresponds to the given property) of this object.

Authorization Cache

As explained in the previous section of this document, every access attempt to SOLoist objects / slots is being checked by the authorization framework. If one SOLoist transaction reads *n* times the same or different slots of the same object *x*, then the method *isAllowedToReadSlot* of object *x* will be called *n* times as well. This is particularly typical when the object space is accessed from the GUI, e.g., from a form with many fields reading or writing slots, when *n* can be exceptionally large. This could lead to unacceptable degradation of performance. Sometimes it is necessary to compute the authorization result every time someone tries to access some object / slot. However, if the computation result should be the same for a certain type of access and for a certain period of time, then it makes sense to cache this result and skip its computation many times.

For that purpose, the authorization mechanism incorporates the *authorization cache*. Principally, the authorization cache cashes the information about the authorization, in a form of mappings of tuples *<user, role, class, object, property, operation_kind>* to the result that can be either true (operation allowed), or false (operation forbidden).

In order to use the authorization cache, set the *use-authorization-cache* parameter in soloist-config.xml to true.

The authorization cache allows SOLoist developers to store pre-calculated authorization results. This way, they can save authorization computation time: if the authorization system finds a result for an access request in the cache, the authorization method of the domain class will not be called.

There are three options regarding how long a certain authorization result will hold (i.e. how long it will be stored in the cache) and also for which user(s) the result will hold.

- Option REQUEST: The result will hold for the lifetime of the HTTP request which stored the authorization result into the cache. The result will hold only for the user who put the authorization result into the cache.
- Option SESSION: The result will hold for the lifetime of the HTTP session of the HTTP request which stored the authorization result into the cache. The result will hold only for the user who owns the session, i.e. who put the authorization result into the cache.
- Option FOREVER: The result will hold for the lifetime of the whole application (forever). The result will hold for the user or for the role which was specified at the moment of putting the authorization result into the cache.

Implementation note 1. All pre-calculated authorization results with REQUEST option will be stored in requests' attributes. Similarly, all pre-calculated authorization results with SESSION option will be stored in sessions' attributes. Finally, all pre-calculated authorization results with FOREVER option will be stored in the *ServletContext's* attributes. This implementation detail determines the lifetime of pre-calculated authorization results as explained above. In other words, if an authorization result is put into the cache with the REQUEST option, it means that it goes into the request's attribute and lives as long as the request lives. It also means that it holds only for actions performed by the corresponding user who initiated that request. If the authorization result is put into the cache with the SESSION option, it means that it goes into the session's attribute and lives as long as the session lives. It holds only for actions performed by the corresponding user who owns the session. Finally, if the authorization result is put into the cache with the FOREVER option, this means that it goes into the *ServletContext's* attribute and lives as long as the *ServletContext* lives. It holds for actions performed by users or roles specified when putting the authorization result into the cache.

Implementation note 2. The API for putting authorization results into the cache does not take parameters such as *HttpServletRequest*, *HttpSession*, nor *ServletContext*, although it does work with these objects. These objects are obtained internally and transparently to the user of the API. They are obtained by calling methods from *RequestResponse* class from SOLoist. SOLoist ensures that the *RequestResponse* class has access to these objects. The only case when the user of the API should consider this is when trying to put something in the authorization cache from code which is not in the context of SOLoist. For example, if implementing some custom-made servlet, the user of the API should first set the request and

response into the *RequestResponse* class by using the *set* method and then call the authorization cache API. This will rarely be the case.

The authorization cache API is really simple. There is only one class to know about: rs.sol.soloist.server.uml.actions.Authorization. This class has many static methods that capture pre-calculated authorization results in a simple to use way. For example, in order to say that creating an instance of class *ApplicationForMembership* is allowed to all users having the role “enrollment” during the application lifetime, the following code should be executed:

```
Authorization.allowCreateNewInstanceOfClassForever(
     ApplicationForMembership.CLASSIFIER, “enrollment”, null);
```

For example, in order to allow creating new instances of class *ApplicationForMembership* for the current user (the user who executes the following code) and for the lifetime of its session, this should be executed:

```
Authorization.allowCreateNewInstanceOfClassForSession(ApplicationForMembership.CLAS
SIFIER);
```

To do the same thing, but for shorter time (the request lifetime) execute this:

```
Authorization.allowCreateNewInstanceOfClassForRequest(ApplicationForMembership.CLAS
SIFIER);
```

In a similar way many other authorization rules can be set using other methods of the API. In order to disallow something, instead of allowing it, just call the not() method after calling any of available static methods. For example, to disallow creating new instances of class *ApplicationFormembership* for the current user and for the lifetime of its session execute this:

```
Authorization.allowCreateNewInstanceOfClassForRequest(
     ApplicationForMembership.CLASSIFIER).not();
```

The following section provides more information on the whole authorization cache API.

Authorization Cache Methods Summary (excerpt from JavaDoc)

Authorization class fully qualified name: rs.sol.soloist.server.uml.actions.Authorization.

```
static AuthorizationRecord allowCreateNewInstanceOfClassForRequest(IClassifier classifier)
```

Allows the current user to create a new instance of the classifier many times during the current request's lifetime.

```
static AuthorizationRecord allowCreateNewInstanceOfClassForSession(IClassifier classifier)
```

Allows the current user to create a new instance of the classifier many times during the current session's lifetime.

```
static AuthorizationRecord allowCreateNewInstanceOfClassForever(IClassifier classifier,
String role, String user)
```

Allows the user or the role to create a new instance of the classifier many times during the application's lifetime.

```
static AuthorizationRecord allowDestroyAnyInstanceOfClassForRequest(IClassifier classifier)
```

Allows the current user to destroy any instance of the classifier during the current request's lifetime.

```
static AuthorizationRecord allowDestroyAnyInstanceOfClassForSession(IClassifier classifier)
```

Allows the current user to destroy any instance of the classifier during the current session's lifetime.

```
static AuthorizationRecord allowDestroyAnyInstanceOfClassForever(IClassifier classifier, String role, String user)
```

Allows the user or the role to destroy any instance of the classifier during the application's lifetime.

```
static AuthorizationRecord allowDestroyInstanceForRequest(IClassifierInstance instance)
```

Allows the current user to destroy the given instance during the current request's lifetime.

```
static AuthorizationRecord allowDestroyInstanceForSession(IClassifierInstance instance)
```

Allows the current user to destroy the given instance during the current session's lifetime.

```
static AuthorizationRecord allowDestroyInstanceForever(IClassifierInstance instance, String role, String user)
```

Allows the user or the role to destroy the given instance during the application's lifetime.

```
static AuthorizationRecord allowEditAnySlotOfAnyInstanceOfClassForRequest(IClassifier classifier)
```

Allows the current user to edit any slot of any instance of the classifier many times during the current request's lifetime.

```
static AuthorizationRecord allowEditAnySlotOfAnyInstanceOfClassForSession(IClassifier classifier)
```

Allows the current user to edit any slot of any instance of the classifier many times during the current session's lifetime.

```
static AuthorizationRecord allowEditAnySlotOfAnyInstanceOfClassForever(IClassifier classifier, String role, String user)
```

Allows the user or the role to edit any slot of any instance of the classifier many times during the application's lifetime.

```
static AuthorizationRecord allowEditAnySlotOfInstanceForRequest(IClassifierInstance instance)
```

Allows the current user to edit any slot that belongs to the given instance many times during the current request's lifetime.

```
static AuthorizationRecord allowEditAnySlotOfInstanceForSession(IClassifierInstance instance)
```

Allows the current user to edit any slot that belongs to the given instance many times during the current session's lifetime.

```
static AuthorizationRecord allowEditAnySlotOfInstanceForever(IClassifierInstance instance, String role, String user)
```

Allows the user or the role to edit any slot that belongs to the given instance many times during the application's lifetime.

```
static AuthorizationRecord allowEditSlotForRequest(ISlot<?> slot)
```

Allows the current user to edit the given slot many times during the current request's lifetime.

```
static AuthorizationRecord allowEditSlotForSession(ISlot<?> slot)
```

Allows the current user to edit the given slot many times during the current session's lifetime.

```
static AuthorizationRecord allowEditSlotForever(ISlot<?> slot, String role, String user)
```

Allows the user or the role to edit the given slot many times during the application's lifetime.

```
static AuthorizationRecord allowEditSlotOfAnyInstanceForRequest(IProperty property)
```

Allows the current user to edit any slot that corresponds to the given property many times during the current request's lifetime.

```
static AuthorizationRecord allowEditSlotOfAnyInstanceForSession(IProperty property)
```

Allows the current user to edit any slot that corresponds to the given property many times during the current session's lifetime.

```
static AuthorizationRecord allowEditSlotOfAnyInstanceForever(IProperty property, String role, String user)
```

Allows the user or the role to edit any slot that corresponds to the given property many times during the application's lifetime.

```
static AuthorizationRecord allowReadAnySlotOfAnyInstanceOfClassForRequest(IClassifier classifier)
```

Allows the current user to read any slot of any instance of the given classifier many times during the current request's lifetime.

```
static AuthorizationRecord allowReadAnySlotOfAnyInstanceOfClassForSession(IClassifier classifier)
```

Allows the current user to read any slot of any instance of the given classifier many times during the current session's lifetime.

```
static AuthorizationRecord allowReadAnySlotOfAnyInstanceOfClassForever(IClassifier classifier, String role, String user)
```

Allows the user or the role to read any slot of any instance of the given classifier many times during the application's lifetime.

```
static AuthorizationRecord allowReadAnySlotOfInstanceForRequest(IClassifierInstance instance)
```

Allows the current user to read any slot that belongs to the given instance many times during the current request's lifetime.

```
static AuthorizationRecord allowReadAnySlotOfInstanceForSession(IClassifierInstance instance)
```

Allows the current user to read any slot that belongs to the given instance many times during the current session's lifetime.

```
static AuthorizationRecord allowReadAnySlotOfInstanceForever(IClassifierInstance instance, String role, String user)
```

Allows the user or the role to read any slot that belongs to the given instance many times during the application's lifetime.

```
static AuthorizationRecord allowReadSlotForRequest(ISlot<?> slot)
```

Allows the current user to read the given slot many times during the current request's lifetime.

```
static AuthorizationRecord allowReadSlotForSession(ISlot<?> slot)
```

Allows the current user to read the given slot many times during the current session's lifetime.

```
static AuthorizationRecord allowReadSlotForever(ISlot<?> slot, String role, String user)
```

Allows the user or the role to read the given slot many times during the application's lifetime.

```
static AuthorizationRecord allowReadSlotOfAnyInstanceForRequest(IProperty property)
```

Allows the current user to read any slot that corresponds to the given property many times during the current request's lifetime.

```
static AuthorizationRecord allowReadSlotOfAnyInstanceForSession(IProperty property)
```

Allows the current user to read any slot that corresponds to the given property many times during the current session's lifetime.

```
static AuthorizationRecord allowReadSlotOfAnyInstanceForever(IProperty property, String role, String user)
```

Allows the user or the role to read any slot that corresponds to the given property many times during the application's lifetime.