Version 0.3

Access Control Lists

Proposal for Access Control Lists in CMIS

12/18/2008

CONTENTS

Contents 2

Introduction 5

Status 5

Relation to the CMIS 0.5 Specification 5

Design objectives 5

How Does This Proposal Relate to Other Standards 7

Content Repository for Java – JSR 283 7

HTTP Extensions for Distributed Authoring -- WebDAV 8

XACML 8

General Concepts 9

Overview of ACLs 9

Access Control Lists 9

Access Control Entries 9

Principal 10

Permissions 10

Data Model 15

Option 1: Applying ACLs with policies 15

Option 2: Applying ACLs using properties 16

Conclusion 16

Discovering ACL Capabilities OF A REPOSITORY 17

Checking Privileges 18

Discovering the ACLs of an Object 18

Applying an ACL to an Object 18

Implications when creating objects 19

Operations and Required Permissions 20

Required permissions per operation 23
Object Service 23
CreateDocument Operation 23
CreateFolder Operation 23
CreateRelationship Operation 24
CreatePolicy Operation 24
GetAllowableActions Operation 24
GetProperties Operation 25
GetContentStream Operation 25
UpdateProperties Operation 26
MoveObject Operation 26
DeleteObject Operation 27
DeleteTree Operation 27
SetContentStream Operation 28
DeleteContentStream Operation 28
Repository Service 28
GetRepositories Operation 28
getRepositoryInfo Operation 29
getTypes Operation 29
getTypeDefinition Operation 30
Navigation Service 30
GetDescendants Operation 30
GetChildren Operation 31
GetFolderParent Operation 31
GetObjectParents Operation 32
GetCheckedoutDocuments Operation 32
Multifiling Service 32

AddObjectToFolder Operation 32
RemoveObjectFromFolder Operation 33
Discovery Service 33
Query Operation 33
Versioning Service 34
checkOut Operation 34
cancelCheckOut Operation 34
checkIn Operation 35
getAllVersions Operation 35
RelationShip Service 36
getRelationships Operation 36
Policy Service 36
ApplyPolicy Operation 36
RemovePolicy Operation 37
GetAppliedPolicies Operation 37

INTRODUCTION

The CMIS specification defines a generic policy model. This proposal is about Access Control Lists (ACLs) as a specific subset of the policy model. Other options to support ACLs with CMIS are briefly discussed as well (see *DATA MODEL* below for more details).

STATUS

This document is just a first proposal. It's sole purpose is to clarify the wording and outlines the assumptions which are relevant when dealing with ACLs in CMIS.

RELATION TO THE CMIS 0.5 SPECIFICATION

The 0.5 version of the specification draft introduces a concept of policy objects as part of the specification.

The purpose of policies is to restrict access to certain methods of an object to a subset of principals. Policies like other primary entities of the CMIS specification are typed, have an id and have properties. A policy is created using the createPolicy method of the Object Service. Input of this method is a description of the policy (name, type, properties, etc.), output is an ID of the created policy instance. Providing this ID, a policy can be applied to a controllable object (applyPolicy), removed (removePolicy), or retrieved from an object (getAppliedPolicies) via the Policy Service. An controllable object can have zero or more policies applied. Not having a policy applied means that there exist no restrictions in accessing the object.

This draft proposes a specification for policies affecting the object, navigation, versioning, multifiling and discovery services. The repository, relationship and policy services are out-of-scope for this proposal.

DESIGN OBJECTIVES

This proposal is based on the CMIS policy concepts. For a specific set of applications it might be easier to use a predefined set of policies – just to make it easier to consume for the application.

We tried to classify applications and their security requirements in three kinds of szenarios:

1. Collaborative applications, like Collaborative Content Creation, Portals, Mashups, where an end user decides about the permissions to be applied to the documents at runtime (e.g. "My working drafts for the documentation should only be editable by my co-workers and be visible to my team, but the must not to be seen by someone else outside the team").

2. Background tasks, like an archiving application, where a developer has to specify the permissions to be applied at designtime (e.g. "Documents moved by the archiving service become read-only, except for the special group 'archive admin'").

3. Business applications, like attaching the scanned images of an invoice to the ERP data, will require application specific security (e.g. "Invoices with a total of more than 1M$ should not be visible by anyone who's not a member of the controlling team and has doesn't have at least a clearance level 2").

➔ As 1. and 2. are the "Core ECM" use cases for CMIS, we will focus on these kind of use cases. 3. is considered as currently out of scope for CMIS, thus we will not try to provide a solution for this kind of scenarios, as we assume that policies are a better choice to handle this kind of specific security constraints anyway.

Applications of type 2 would need to know about the semantics of the policies to be applied already during designtime, thus some semantics have to be defined (either already as part of the CMIS specification or as a mapping at latest during runtime). And allthough applications of type 1 do not necessarily need to know about the semantics (since the end user would have to deal with the permissions), it would be helpful at least for documentation purposes.

Thus, the requirement is as follows: A developer should be able to work with *permissions* for CMIS objects in an interoperable manner at designtime – without needing to know what the concrete repository will be at runtime. ACLs imply at least a basic semantic for a policy in terms of "*who* is allowed to do *what*" – in the scenarios above the *who* is known by the application, so this proposal will focus on the *what* (the permissions).

There are basically two different options how permissions can be made "interoperable": Either there is one permission model already predefined by CMIS (the applications and the repositories would have to map their permissions to that "standard", requiring potentially 1:n mappings for the applications and 1:m mappings for the repositories). Or the mapping of the permissions is done for potentially every combination of application and repository (resulting in n:m-mappings, either to be handled by the application or the repository).

The second options allows for more flexibility – but requires more mappings and therefore more administrative effort than the first option.

As we don't have a clear picture on the applications requirements yet, we potentially have to support both options. Therefore we would suggest to divide the specification for handling policies and permissions in three different levels (related to the three application categories above):

- Level 0: Applications working with generic policies.
  → security defined via generic policy (for business applications, category 3.)
  → for interoperability this will require a notation for specifying generic policies in an interchangeable format (e.g. using XACML)
  → out of scope for this proposal

- Level 1: Applications with a flexible mapping of permissions per repository.
  → security defined via ACL, where permissions are unknown to the application (for end user applications,

category 1.)
→ will require mappings either on the application or the repository side, thus requires at least the capability to discover the available permissions and their relationship
➔ will be refered to as Level-1-ACLs or Generic ACLs.

- Level 2: Applications with a fixed mapping.
  → like Level 1, but permissions are known to the application (for background jobs or application specific permissions, category 2.)
  → requires mappings on the repository side, assuming that the known permissions are predefined by CMIS
  ➔ will be refered to as Level-2-ACLs or CMIS-Defined ACLs.

As we focus on ACLs within the scope of this proposal, and as the ACEs of an ACL define *who* is allowd to do *what*, two additional assumptions:

1. Regarding the *who*: We assume that all the systems share a common understanding of the principals to be checked. In an enterprise or intranet scenario, this is more likely to be the case, as a central LDAP or other kind of directory service will most probably be available. For extranet/internet scenarios, we assume that more generic authentication standards will be relevant (in the worst case, the CMIS consumer would have to do the user mapping by means beyond the scope of CMIS).
   ➔ We assume that principals are known to both, consumer and provider – thus user/group discovery is not within the scope of this document.

2. Regarding the *what*: We assume that ACLs are applied to folder- and document-like objects only, and that checks against ACLs are performed for operations on those objects only.
   ➔ We assume that ACLs are appropriate for the basic object types folder and document (not for relationship, policy) as this concept is known from existing file system implementations – other CMIS objects would have to be secured by policies then.

HOW DOES THIS PROPOSAL RELATE TO OTHER STANDARDS

Content Repository for Java – JSR 283

As we expect that JCR could serve as a local Java API for the CMIS protocol (either for consumers or for providers), the ACL concepts proposed by CMIS should be mappeable to JCR.

Level 0 (Policies) can be mapped to the JCR's AccessControlPolicy objects and handling of policies can be mapped to the AccessControlManager's `get`…/`set`…/`delete`… methods (while CMIS' addPolicy could be mapped to a getApplicablePolicies on a specific system path, or createing a node with a specific structure (→ XACML?)).

Level 1 is not covered by JCR (to be mapped to Level 0 then as well, if not restricted to Level 2).

Level 2 should be mappable (by taking care that the semantics defined in this proposal are compliant with the JCR's standard privileges jcr:read, jcr:setProperties, jcr:addChildNodes, jcr:removeChildNodes, jcr:write, jcr:getAccessControlPolicy, jcr:setAccessControlPolicy, and jcr:all).

HTTP Extensions for Distributed Authoring -- WebDAV

Reference: http://www.ietf.org/rfc/rfc2518.txt and http://www.webdav.org/acl/

As we expect that WebDAV plus some specific extensions (like the system view) are used by JCR as a potential protocol for remote access, and as we assume that the CMIS providers will expose themselves via WebDAV as well (this would make some sense if CMIS is considered as being a more ECM focused protocol for enterprise system-to-system interoperability (SOAP) and for enterprise clients (REST), while WebDAV was developed as a more document centric protocol for the internet (lacking features like relations or document types, but providing advanced versioning)), the proposed ACL concept should be mappeable to WebDAV as well.

XACML

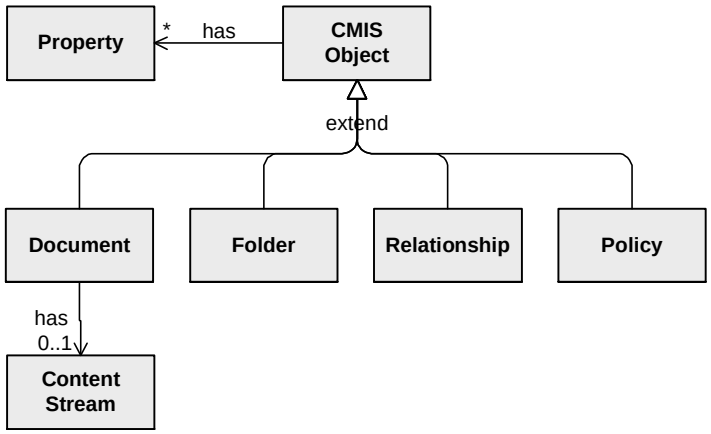
Reference: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml#XACML20

XACML is much more generic and flexible then the proposed ACLs. As outlined above, we consider more generic security handling and policies as being out of scope for this proposal. Thus, although XACML might become relevant when getting into more details for the policies, we won't take XACML into account for this proposal when looking at ACLs.

GENERAL CONCEPTS

To provide a more formal naming, the following sections use a Java style pseudo code to illustrate the proposal.

The CMIS specification currently defines the following object hierarchy:



OVERVIEW OF ACLS

Access Control Lists

An Access Control List (ACL) is a list of Access Control Entires (ACE):

```
public List<AccessControlEntry> AccessControlList;
```

TBD: Identification of an ACL as Policy vs. ACEs as Policies?
Documentum has a concept of ACL Templates, which could be mapped to policies straight forward. But a concrete instance of an ACL is usually something bound to a document or folder – mapping these specific instances to policies would result in a large number of policies.

Proposal: ACLs itself should not be mapped policies – only the ACEs are mapped to policies (see below), an ACL would be defined by a set of policies.

Access Control Entries

An Access Control Entry (ACE) specifies a *permission* and a *principal*:

```
public interface AccessControlEntry {
  public String getPrincipalID();
  public String getPrivilege();
}
```

Alternatively, specific object types *PrivilegeType* and *PrincipalType* could be defined as well.

TBD: ACEs to be mapped either to policies or to properties? How to map ACEs to Properties then: either as multivalued string properties (permission as name, principals as values of the multivalued string), or as XML property (what schema to use then)? Or should a new ACE type be defined?

Proposal: ACEs are mapped to a set of policies, where each policy represents a permission, holding a a multivalue String property "principals", which in turn lists all the principal IDs having the permission represented by the property (see *PERMISSIONS* below for more details on the mapping).

Principal

A principal represents either a single user, or a set of users – this can be a group, or some other notion like a "role". We assume that every system has some kind of user management, which is able to retrieve the relevant information from the central LDAP or directory service:

```
public interface SecurityPrincipal {
  public String getPrincipalID();
  public boolean equals(SecurityPrincipal user);
}
```

And for a set of users:

```
public interface SecurityPrincipalSet extends SecurityPrincipal {
  public boolean contains(SecurityPrincipal user);
}
```

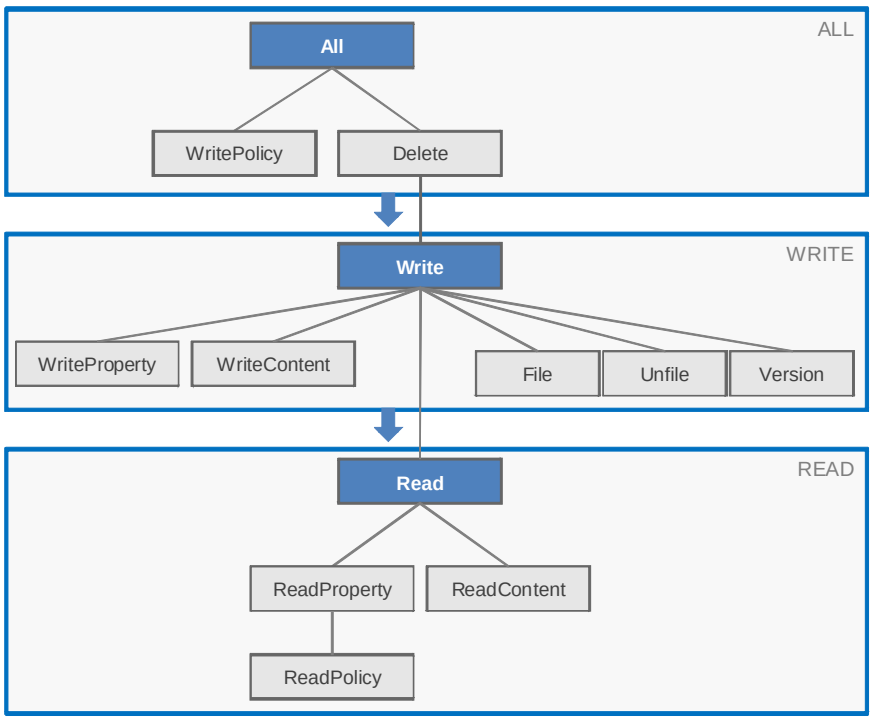
In order to simplify serialization/deserialization, we assume that some kind of user management is available (see the assumptions in the *DESIGN OBJECTIVES* above):

```
public class UserManagement {
  public SecurityPrincipal getPrincipalByID(String id);
}
```

Permissions

As outlined in the *DESIGN OBJECTIVES*, the goal for this proposal is to specifiy an interoperable permission model – at least providing mechanisms enableing applications to discover the "meaning" of the permissions provided by the repositories; or if possible to specify a predefined set of permissions.

The following figure shows the proposed set of predefined permissions:



The blue colored boxes show the basic/mandantory permissions. At least this three permissions (ALL, WRITE, READ) should to be exposed by a repository as a minimal set.

The light grey colored boxes show extended/optional permissions. These are permissions that might be exposed by a repository in addition to the basic/mandatory permissions. This can be done for any individual optional permission (e.g. a repository could choose to decide to provide READ and READPOLICY only, it is not required to provide READPROPERTY and READCONTENT as well).

The blue boxes should also illustrate that the basic permissions include their "related" set of extended permissions (e.g. READ includes READPROPERTY, READCONTENT and READPOLICY when applied to an object).

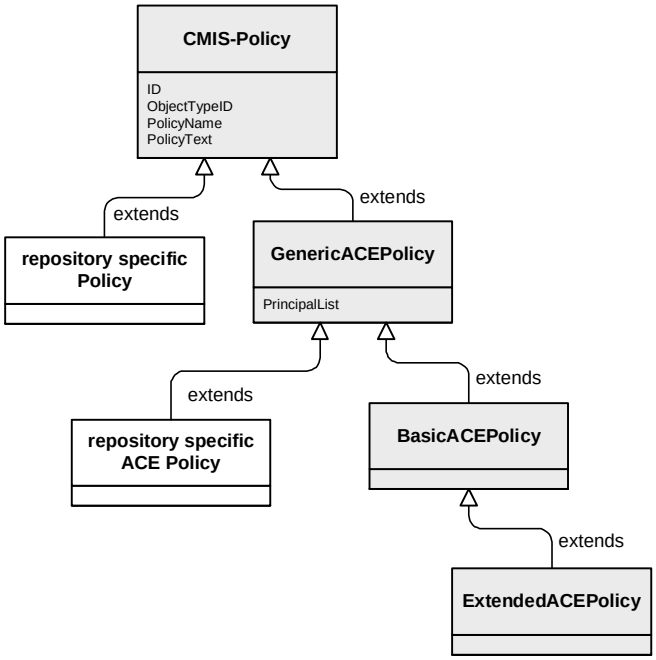
The lines show the "include"-relations for the permissions. The permission ALL includes all other permissions, thus if ALL is assigned to a user, this includes all other permissions. DELETE includes WRITE, which in turn includes READ. The blue arrows should illustrate that this "include" relation is currently specified for the basic permissions and their "related" set of extended permissions – the "include" relations between extended permissions "related" to different basic permissions (e.g. WRITEPROPERTY (related to WRITE) and READPROPERTY (related to READ)) is not specified.

- ALL: (jcr:all) includes all other permissions.
- WRITEPOLICY: (jcr:setAccessControlPolicy) permits to change the permission of the given object.
- DELETE: (jcr:removeChildNodes on the properties) permits to delete the object (in that sense that the object is removed and all depending objects like filings are deleted as well). This also includes WRITE.
- WRITE: (jcr:write) permits the modification of the object itself. This includes all kind of more specific WRITE... permissions (also Bind/File, Unbind/Unfile, and Version), and READ.

- WRITEPROPERTY: (jcr:setProperty) permits the modification of properties (distinct from WRITECONTENT).
- WRITECONTENT: (jcr:setProperty on the jcr:content property) permits the modification of a document's content stream (distinct from WRITEPROPERTY).
- FILE: (jcr:addChildNodes on the parent) permits to add a document to a folder (bind in WebDAV).
- UNFILE: (jcr:removeChildNodes on the parent) permits to remove a document from a folder (unbind in WebDAV) , should not to be confused with DELETE (UNBIND does not permit the physical deletion of a document).
- VERSION: (jcr:addChildNodes on the version history) permits to create, retrieve and delete versions of this document (to be defined how that relates to the permissions on the specific version object).
- READ: (jcr:read) permits to read the object. This includes READPROPERTY and READCONTENT.
- READPROPERTY: (jcr:read on the properties) permits to read the objects properties, this includes READPOLICY.
- READCONTENT: (jcr:read on the jcr:content property) permits to read the document's content stream.
- READPOLICY: (jcr:getAccessControlPolicy) permits to read the policies assigned to a controllable object.

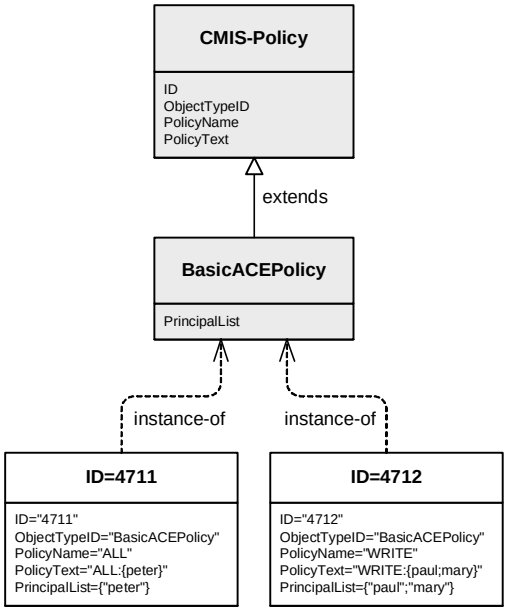
TBD: Are these permissions sufficient for applications? What about the Version permission - is it really needed? What about locking (if it should be added to CMIS) – will extended permissions be required then? What about the "include" relations for the extended permissions – if not specified: why define extended permissions at all (why not fall back to basic permissions only)?

Proposal: (see also *ACCESS CONTROL ENTRIES* above): ACEs are mapped to a set of policies, where each policy represents a permission, holding a a multivalue String property "principals", which in turn lists all the principal IDs having the permission represented by the property. The policies type should identify the type of ACL the ACE belongs to. The figure below illustrates a type hierarchy for the ACEs:



ACEs for Level-1-ACLs should have a type inheriting from GenericACEPolicy assigned. ACEs for Level-2-ACLs should have a type inheriting from BasicACEPolicy (to denote a basic permission, or ExtendedACEPolicy to denote an extended permission).

E.g. a policy type=BasicACLPolicy, name=READ with values (“john”,”mary”) for the property name=“principals” would represent the ACEs (john: READ) and (mary: READ).
An ACL like {(peter: ALL); (paul: WRITE); (mary: WRITE); (ALL: READ)} would be mapped to three policies: {type=“BasicACLPolicy”, name=”ALL”, property(“principals”=(“john”))}; {type=“BasicACLPolicy”, name=”WRITE”, property(“principals”=(“paul”,”mary”))}; {type=“BasicACLPolicy”, name=”READ”, property(“principals”=(“ALL”))}; the figure outlines the types and instances involved:



This proposal restricts to positive ACEs. Thus, a NONE permission is not required, as this is the default permission for a principal not being listed in an ACE (direct or via PrincipalSet, see below).

If no policy and therefore no ACL is assigned to an object, no restrictions apply. If policies, or specifically an ACL is applied to an object, access is restricted according to the ACL – if no rule exists which grants access for the given user, access is denied. More specifically for ACLs: If no ACE exists, where the given user matches[*)] the ACE's principal entry, access is denied.

[*)] Matching means something like

```
ace_principal = UserManagement.getPrincipalByID(ACE.getPrincipalID());
matches = ace_principal.equals(user)
          || ((ace_principal instanceof SecurityPrincipalSet)
              && ((SecurityPrinicpalSet)ace_principal).contains(user));
```

DATA MODEL

There are two basic options how to map ACLs to CMIS: Either mapping the ACLs to policies, or mapping ACLs to specific properties:

Option 1: Applying ACLs with policies

ACLs (or their ACEs) can be denoted as a special policy type (see *ACCESS CONTROL ENTRIES*Access Control Lists above).

The level of ACL supported by a repository can be returned by the getRepositoryInfo method of the Repository Service (see below). An ACL is a list of ACEs where each ACE consists of a principal associated with a permission (see *ACCESS CONTROL ENTRIES*, *PRINCIPAL* and *PERMISSIONS* above).

To apply an ACL to an object, the following steps would be required with the current methods of the Policy Service:

1. Create the required instances of the ACEs for the ACL and obtain the IDs for the ACEs of the ACL:
   Call `ObjectService.createPolicy()`with a type of `BasicACLPolicy` or `ExtendedACLPolicy`, the principal lists for the ACEs are passed as properties (for details see above). This returns the IDs for the ACEs Policy Objects representing the ACL.
2. Apply the ACEs of the ACL to an object:
   Call `PolicyService.applyPolicy()`with the ID of the target object and the IDs from step 1.

To retrieve an ACL for an object, the following steps are required:

1. Call `PolicyService.getAppliedPolicies()` with the object ID and get back a list of policy IDs.
2. For each returned ID check the policy type – `BasicACLPolicy` or `ExtendedACLPolicy` indicate the ACE-entries for a specific permission.
3. Calculate the permissions from the properties of the returned instances from step 2.

This requires that there can be *only one* set of policies with type `BasicACLPolicy` or its subtype `ExtendedACLPolicy` applied to an object. Additional policies might exist (even of type `GenericACEPolicy`, which represent more generic ACEs with permissions not specified by CMIS) - it would then be up to the repository if and how to map ACEs of generic ACLs to ACEs of basic or extended ACLs.

Comments: This approach fits nicely into the proposed policy concept and the typing model of CMIS. However the addidional indirection requiring an ID for each policy makes it more complicated than necessary for a client. IDs for the ACEs of an ACL might not be exposed by many of the existing APIs of common repositories, making it difficult to build a CMIS connector on top of an existing API.

Option 2: Applying ACLs using properties

ACLs would be exposed as properties with a fixed name then (e.g. "ACL"). Such an ACL property could be modeled using different approaches:

1. A multivalue type of a newly defined property type ACE (TBD: how to define this property type?).
2. A special string syntax in a multivalued string type (e.g. "(john, mary: read, version); (all: read)" – TBD).
3. A single valued property of type XML with an XML schema (TBD: XML schema for ACLs/ACEs)

The mechanism using the XML schema does not require an extension of the current CMIS specification. However it requires an XML parser/generator for a very basic task like setting/retrieving an ACL. And applications which are able to deal with policies as well, would require two calls (one for the ACL and one for the policies) to get the "complete" picture of applied security constraints for a specific object.

To apply an ACL to an object, the following steps are required:

1. Fill a property named "ACL" (depending on the type defined above, TBD).
2. Call the `ObjectService create…` or `.updateProperty()` method and add/update the property from step 1 to the list of other properties.

To retrieve an ACL from an object, the following steps are required:

1. Retrieve all the properties from the object with the given object id.
2. Calculate the permissions from the property named "ACL" by parsing the appropriate syntax (TBD).

Comments: This approach is much simpler to implement for repositories as well as clients. It does not require additional round trips and dealing with ids. It also gives repositories more flexibility how to implement ACLs and avoids the problem how to set an initial ACL at creation time. On the other side it does not make use of the intended policies for this purpose and defines a special semantic for a named property which might be considered a bad design.

Conclusions

TBD: What to use for ACLs: Policies or Properties?

The policy concept is very suitable for access controls mechanisms that have a more dynamic nature (and using e.g. XACML to describe them). It also very convenient for other mechanisms sitting on top of basic permissions (e.g. retention management, security clearance, supplemental markings, etc.). However for ACLs it can be used but imposes additional burden for applications and repositories with the current specification, if we stick to the current methods defined for the Object Service and Policy Service.

The property based approach is much simpler to use and implement – but provides less flexibility and is more difficult to understand when combined with policies.

Thus, we are facing two major challenges:

- The main problem to solve when mapping ACLs to policies is about how to avoid the "instantiation" of a policy for an ACL (which is kind of "anonymous" to the application, and is already specified by the ACEs which in turn are specified by the principals and permissions).
  ➔ As this proposal tries to focus on a mapping on ACLs to policies, we would require additional methods for the Object Service and Policy Service.

- The main problem to solve when mapping ACLs to properties is about how to mix ACL properties with policies, and how to distinguish ACLs capabilities from property handling capabilities then.

DISCOVERING ACL CAPABILITIES OF A REPOSITORY

RepositoryService.getRepositoryInfo must return some information, if the repository supports ACLs at all – either generic (Level 1), or specific (Level 2 – and then: what subset of the predefined set).

TBD: How to include the type of ACL support in the result from getRepositoryInfo?

Proposal: an additional property capabilityPolicySupportOptions should be returned by the Repository Service when getRepositoryInfo is called. If the property is not existing or empty, no policies are supported.

The property capabilityPolicySupportOptions is a multivalued String property, which can contain the following entries as values: "GenericPolicies" to indicate that repository specific policies are available, "XACMLPolicies" to indicate that XACML compliant policies are available, "GenericACLs" to indicate that generic ACLs (with a repository specific permission set, Level-1-ACLs) are available , and "DefaultACLs" to indicate that ACLs with the semantics as specified by CMIS are available (Level-2-ACLs).

TBD: How to expose the supported permissions in an hierarchy?

Proposal: an additional property capabilityACLPermissionSupportOptions must be returned by the Repository Service when getRepositoryInfo is called, and if the property capabilityPolicySupportOptions contained an entry for "GenericACLs" or "DefaultACLs".

The property capabilityACLPermissionSupportOptions is a XML property, where the XML contains the information about the applicable permission set and the hierarchy of permissions (see *PERMISSIONS* for the default set of permissions and the default hierarchy for the basic permissions).

TBD: A XML schema for specifying the permissions and their hierarchy relationship.

CHECKING PRIVILEGES

ObjectService.getAllowableAction should be used to check for the effective privileges, ObjectService.getAllowableAction would require more specification anyhow (e.g. is currently unclear, what has to be checked for a moveObject operation).

TBD: How to use getAllowableAction in detail? Clarification of the existing CMIS specification required.

TBD: Would there be any benefits if an application would be able to compute the allowed actions by knowing the ACLs?

DISCOVERING THE ACLS OF AN OBJECT

PolicyService.getAppliedPolicies should be used to retrieve the applied policies for an object.

TBD: Basic assumption is that ACLs are applied to documents and folders only.

TBD: getAppliedPolicies currently supports a filter as input parameter – but right now only Property Filters are described in more detail in the specification. Additional clarification required (maybe a more specific Policy Filter type or making policies a subtype of properties?).

APPLYING AN ACL TO AN OBJECT

PolicyService.applyPolicy should be used to to apply the ACEs which represent an ACL to an object.

TBD: No inheritance is assumed – i.e. applying a policy to a folder does not change the ACLs of existing documents within that folder. Furthermore, only "positive" ACEs are assumend, therefore no explicit ordering is required.

As outlined above, the ACEs would have to be created before via ObjectService.createPolicy first. Therefore, at least an additional method createACEPolicies() which takes a list of permissions + principal-lists and returns a set of IDs for the ACEs would be nice.

TBD: ObjectService.createPolicy or createACEPolicies() should not be required before – so how to extend PolicyService.applyPolicy?

Proposal: either an additional ACL Service, or an additional method PolicyService.applyACLPolicy(repositoryID, array<permission+principal-list>, targetObjectID) which allows an ACL to be applied for the given array of permissions + principal-lists lists.

TBD: Do we need a specific permission+principal-list container, or is it sufficient to map that to properties (where the name specifies the permission and the value is a multi valued string, listing the principal IDs of the principal-list)?

IMPLICATIONS WHEN CREATING OBJECTS

TBD: There might be some use cases, where an application requires an initial ACL to be applied to an object when it is created – even in a single step as an atomic action to avoid security risks caused by intermediate (potentially unkown) states.

Proposal: It should be possible to apply an initial ACL to an object when using the ObjectService.create... methods. This would require additional parameters for the create... methods (at least for createFolder and createDocument), which allows at least a list of IDs (for the ACEs building the ACL) to be passed as input.

Taking the considerations from *APPLYING AN ACL TO AN OBJECT* into account as well, this would result in create... methods, wich correspond to using the array<permission+principal-list> parameter as well.

If no initial ACL is provided, the logic of how the security settings for the newly created object are derived is up to the repository. This can be any kind of strategy, like an implicit ACL is defined by repository, or the ACL is inherited from a virtual root folder if no other ACL is applied by other inheritance rules, or just that at least current user has full access, etc..

OPERATIONS AND REQUIRED PERMISSIONS

The table below shows an overview for the proposed usage of the basic permissions per CMIS operation.

	Operation		
OBJECT SERVICE	Read	Write	All
`Create…`		parent	
`GetAllowableActions`			
`GetProperties`	object		
`GetContentStream`	document		
`UpdateProperties`		object	
`MoveObject`		object, parent, target	
`DeleteObject`			object
`DeleteTree` *(delete)*			object
`DeleteTree` *(unfile)*		document, parent	
`SetContentStream`		document	
`DeleteContentStream`		document	
REPOSITORY SERVICE	Read	Write	All
`GetRepositories`			
`getRepositoryInfo`			
`getTypes`			
`getTypeDefinition`			
NAVIGATION SERVICE	Read	Write	All
`GetDescendants`	folder		
`GetChildren`	folder		
`GetFolderParent`	folder		
`GetObjectParents`	object		
`GetCheckedoutDocuments`	document		
MULTIFILING SERVICE	Read	Write	All
`AddObjectToFolder`		folder, object	
`RemoveObjectFromFolder`		folder, object	

DISCOVERY SERVICE	Read	Write	All
`Query`	object		
VERSIONING SERVICE	Read	Write	All
`checkOut`		document	
`cancelCheckOut`		document	
`checkIn`		document	
`getAllVersions`	document		
RELATIONSHIP SERVICE	Read	Write	All
`getRelationships`			
POLICY SERVICE	Read	Write	All
`ApplyPolicy`			
`RemovePolicy`			
`GetAppliedPolicies`			

The table below shows an overview for the proposed usage of the extended permissions per CMIS operation.

	Operation									
OBJECT SERVICE	Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy
`Create…`				parent						
`GetAllowableActions`										
`GetProperties`		object								
`GetContentStream`			document							
`UpdateProperties`						object				
`MoveObject`				target	source	object				
`DeleteObject`									object	
`DeleteTree` *(delete)*									object	
`DeleteTree` *(unfile)*					folder	object				
`SetContentStream`							document			
`DeleteContentStream`							document			
REPOSITORY SERVICE	Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy
`GetRepositories`										

getRepositoryInfo										
getTypes										
getTypeDefinition										
NAVIGATION SERVICE	Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy
GetDescendants		folder								
GetChildren		folder								
GetFolderParent		folder								
GetObjectParents		object								
GetCheckedoutDocuments		document								
MULTIFILING SERVICE	Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy
AddObjectToFolder				folder		object				
RemoveObjectFromFolder					folder	object				
DISCOVERY SERVICE	Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy
Query		object								
VERSIONING SERVICE	Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy
checkOut								document		
cancelCheckOut								document		
checkIn								document		
getAllVersions		document								
RELATIONSHIP SERVICE	Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy
getRelationships										
POLICY SERVICE	Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy
ApplyPolicy										
RemovePolicy										
GetAppliedPolicies										

REQUIRED PERMISSIONS PER OPERATION

OBJECT SERVICE

CreateDocument Operation

Basic Permission Set		
Read	Write	All

If a parent folder is specified, Write permission is required on the parent folder.

TBD: What happens if no parent folder is specified? Possible solution: All unfiled documents have a virtual parent which contains the access rights for unfiled documents.

Extended Permission Set									
Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy

If a parent folder is specified, File permission is required on the parent folder.

CreateFolder Operation

Basic Permission Set		
Read	Write	All

Requires Write permission on the parent folder.

Extended Permission Set									
Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy

Requires File permission on the parent folder.

CreateRelationship Operation

Basic Permission Set		
Read	Write	All

TBD

Extended Permission Set									
Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy

TBD

CreatePolicy Operation

Basic Permission Set		
Read	Write	All

Requires Write permission on th specified object.

Extended Permission Set									
Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy

Requires WritePolicy permission on th specified object.

GetAllowableActions Operation

Basic Permission Set

Read	Write	All

For this operation no specific permission is required.

Extended Permission Set									
Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy

For this operation no specific permission is required.

GetProperties Operation

Basic Permission Set		
Read	Write	All

Requires Read permission for the specified object

Extended Permission Set									
Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy

Requires ReadProperty permission for the specified object

GetContentStream Operation

Basic Permission Set		
Read	Write	All

Requires Read permission for the specified object

Extended Permission Set									
Read	Read	Read	File	Unfile	Write	Write	Version	Delete	Write

Policy	Property	Content			Property	Content			Policy

Requires ReadContent permission for the specified object

UpdateProperties Operation

Basic Permission Set		
Read	Write	All

Requires Write permission for the specified object

Extended Permission Set									
Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy

Requires WriteProperty permission for the specified object

MoveObject Operation

Basic Permission Set		
Read	Write	All

Requires Write permission on the parent folder of the object to be moved, Write permission on the target folder and Write permission on the object to be moved.

Extended Permission Set									
Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy

Requires Unfile permission on the parent folder of the object to be moved, File permission on the target folder and WriteProperty permission on the object to be moved.

DeleteObject Operation

Basic Permission Set		
Read	Write	All

Requires All permission on the object to be deleted.

Extended Permission Set									
Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy

Requires Delete permission on the object to be deleted.

DeleteTree Operation

Basic Permission Set		
Read	Write	All

Depending on the specified parameters the operation can result in an unfiling or in a delete for several objects. The required permissions are differ for unfiling or deletion:

Delete:
For all objects to be deleted, All permission is required.

Unfile:
For non folder objects to be unfiled, Write permission is required on the object itself and its parent.

Extended Permission Set									
Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy

Delete:
For all objects to be deleted, Delete permission is required.

Unfile:
For non folder objects to be unfiled, WriteProperty permission is required on the object itself and Unfile permission on its parent.

SetContentStream Operation

Basic Permission Set		
Read	Write	All

Requires Write permission on the object to be updated.

Extended Permission Set									
Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy

Requires WriteContent permission on the object to be updated.

DeleteContentStream Operation

Basic Permission Set		
Read	Write	All

Requires Write permission on the object to be updated.

Extended Permission Set									
Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy

Requires WriteContent permission on the object to be updated.

REPOSITORY SERVICE

GetRepositories Operation

Basic Permission Set		
Read	Write	All

For this operation no specific permission is required.

Extended Permission Set									
Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy

For this operation no specific permission is required.

getRepositoryInfo Operation

Basic Permission Set		
Read	Write	All

For this operation no specific permission is required.

Extended Permission Set									
Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy

For this operation no specific permission is required.

getTypes Operation

Basic Permission Set		
Read	Write	All

For this operation no specific permission is required.

Extended Permission Set									
Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy

For this operation no specific permission is required.

getTypeDefinition Operation

Basic Permission Set		
Read	Write	All

For this operation no specific permission is required.

Extended Permission Set									
Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy

For this operation no specific permission is required.

NAVIGATION SERVICE

GetDescendants Operation

Basic Permission Set		
Read	Write	All

Requires Read permission on the specified folder. The method returns only those descendants where the user has Read access to.

Extended Permission Set									
Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy

Requires ReadProperty permission on the specified folder. The method returns only those descendants where the user has at least ReadProperty access to.

GetChildren Operation

Basic Permission Set		
Read	Write	All

Requires Read permission on the specified folder. The method returns only those children where the user has Read access to.

Extended Permission Set									
Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy

Requires ReadProperty permission on the specified folder. The method returns only those children where the user has at least ReadProperty access to.

GetFolderParent Operation

Basic Permission Set		
Read	Write	All

Requires Read permission on the specified folder and its parent. If the ancestor nodes are requested, the method returns only those ancestors, where the user has Read permission to.

Extended Permission Set									
Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy

Requires ReadProperty permission on the specified folder and its parent. If the ancestor nodes are requested, the method returns only those ancestors, where the user has ReadProperty permission to.

GetObjectParents Operation

Basic Permission Set		
Read	Write	All

Requires Read permission on the specified object. The method returns only those parents, where the user has Read access to.

Extended Permission Set									
Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy

Requires ReadProperty permission on the specified object. The method returns only those parents, where the user has ReadProperty access to.

GetCheckedoutDocuments Operation

Basic Permission Set		
Read	Write	All

Requires Read permission for the checked out documents. The method returns only those checked out documents where the user has Read permissions to.

Extended Permission Set									
Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy

Requires ReadProperty permission for the checked out documents. The method returns only those checked out documents where the user has at least ReadProperty permissions to.

MULTIFILING SERVICE

AddObjectToFolder Operation

Basic Permission Set		
Read	Write	All

Requires Write permission on the specified parent folder.

Extended Permission Set									
Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy

Requires File permission on the specified parent folder.

RemoveObjectFromFolder Operation

Basic Permission Set		
Read	Write	All

Requires Write permission on the specified parent folder and Write permission on the object to be removed.

Extended Permission Set									
Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy

Requires Unfile permission on the specified parent folder and Write permission on the object to be removed.

DISCOVERY SERVICE

Query Operation

Basic Permission Set		
Read	Write	All

A query always returns only those objects where the user has Read permission to.

Extended Permission Set									
Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy

A query always returns only those objects where the user has at least ReadProperty permission to.

VERSIONING SERVICE

checkOut Operation

Basic Permission Set		
Read	Write	All

Requires Write permission on the specified Object

Extended Permission Set									
Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy

Requires Version permission on the specified Object

cancelCheckOut Operation

Basic Permission Set		
Read	Write	All

Requires Write permission on the specified Object

Extended Permission Set

Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy

Requires Version permission on the specified Object

checkIn Operation

Basic Permission Set		
Read	Write	All

Requires Write permission on the specified Object

Extended Permission Set									
Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy

Requires Version permission on the specified Object

getAllVersions Operation

Basic Permission Set		
Read	Write	All

Requires Read permission on the specified Object

Extended Permission Set									
Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy

Requires ReadProperty permission on the specified Object. The Method returns only those versions where the user has at least ReadProperty Permission to.

RELATIONSHIP SERVICE

getRelationships Operation

Basic Permission Set		
Read	Write	All

TBD

Extended Permission Set									
Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy

TBD

POLICY SERVICE

ApplyPolicy Operation

Basic Permission Set		
Read	Write	All

Requires All permission on the specified Object

Extended Permission Set									
Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy

Requires WritePolicy permission on the specified Object

RemovePolicy Operation

Basic Permission Set		
Read	Write	All

Requires All permission on the specified Object

Extended Permission Set									
Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy

Requires WritePolicy permission on the specified Object

GetAppliedPolicies Operation

Basic Permission Set		
Read	Write	All

Requires Read permission on the specified Object

Extended Permission Set									
Read Policy	Read Property	Read Content	File	Unfile	Write Property	Write Content	Version	Delete	Write Policy

Requires ReadPolicy permission on the specified Object