KC KIM Migration: KIM Exposed

The following is the conversion to migrate the authZ components of KC KIM to Rice KIM. This does not cover entity migration.

# A quick note on the analysis

At the time of this analysis, there is no available documentation related to new KIM and only outdated documentation for old KIM. This analysis was done by looking through the KIM codebase and database schema, talking to various developers, and analyzing KFS’ use of KIM. The KIM codebase is not entirely the same between what KFS is using and what KC will be using which may lead to some peculiarities in this analysis.

This analysis assumes that KIM entity migration is complete.

# Migration Approach Overview

The basic approach of this migration is the following:

* Migrate AuthZ data
* Delete ALL KC KIM database tables, Java APIs, related XML, etc.
* Have KC authorization use KIM

This migration approach is much more invasive then the KC Person migration. Since KIM AuthZ has changed in very significant ways, the current KC APIs will not be preserved. Instead, all KC KIM logic will be removed and replaced with Rice KIM. Code changes should be isolated since most calls to KC KIM are done within “Authorizers.”

# Definitions

* **Old KIM:** Kuali Coeus’ KIM
* **New KIM:** Rice KIM also referred to as KIM
* **Principal:** Represents a KIM entity that can authenticate into a Kuali system.
* **Group:** Way to organize (categorize) Principals and Groups (nesting).
* **Role:** Representing a “function” that a role member may be able to do. Roles aggregate Permissions and Responsibilities. Roles have members that are Groups, Roles(nesting), or Principals.
* **Permission:** Defines a granular task that is non-workflow related. Assigned to a role(s). Checks authorization via attribute values.
* **Responsibility:** Defines a granular task that is workflow related. Assigned to a role(s). Checks authorization via attribute values.
* **KimType:** Defines the authorization criteria for elements of KIM. The criterion defines member attribute names and a Matching service. Both are optional. Group, Role, Permission, Responsibility, and Delegation are KimTypes.
* **Matching service:** KIM will invoke the Matching service defined in a KIM Types. Since many things are KIM Types, multiple services may get invoked while looking for authorization. These services decide whether a principal is a match for a KIM element (i.e. does principal have permission “foo”) which ultimately defines authorization.
* **Attribute:** A Name/Value pair used to check authorization. These attributes are defined in the database as attribute definitions. They are matched via static values specified in the KIM database if using the default Service implementations. If not using the defaults you can provide your old matching logic ignoring the attribute values stored in the database.
* **Namespace:** A way to organize elements of KIM based on System, Subsystem, module, etc. Examples: KFS, KRA, KRA-AWD. This also helps prevent name clashes between elements of KIM since each element is assigned to a namespace.
* **Template:** Permissions and Responsibilities have templates. These templates allow the reuse of permission/responsibility data within multiple permissions/responsibilities while specifying different attribute data values.
* **Delegation:**  Delegation provides a means to authorize a Principal for any permission under a role without the Principal having that explicit permission as long as they have the correct attributes/qualification. There are primary and secondary types. Not sure what this is.

# General Open Points/Notes/Strangeness

* The KIM API has a concept of attributes for auth matching. These attributes are sent to KIM through a Map<String, String>. Using a Map<String, String> is simple but breaks down for complex auth matching.

For example: imagine you wanted to check authorization for a permission called ‘edit\_foo’ with a “qualification” that the attribute called ‘baz’ must have the value ‘1’ or ‘2’. This check would require calling into KIM multiple times or defining a Matching service passing all the attribute values encoded into a single attribute. Calling KIM multiple times may have performance implications while defining a Matching service is complex relative to the task at hand.

* KIM no longer defines a namespace in a table. Many of the KIM table still have a namespace code but there is no referential integrity to makes sure that a namespace exists.
* KIM does not have concept of Namespace default attributes which are common attributes that all Entities in a namespace must define. **Is this something that KC needs?**
* To customize authorization, KIM has the idea of a Matching service that is defined in something called a KimType. Since Kuali projects can run in remote mode, these services must be exposed in a manner that KIM can access them remotely. This could create some performance challenges for applications using KIM.

For example: imagine a call to KIM (via a remote service). KIM does some processing and then determines that is must call a Matching service from KC. This service is then accessed remotely. A single flow like this may not be harmful; however, if multiple calls into KIM are necessary this may become problematic. Furthermore, the client (KC) will not know whether a Matching service will be invoked at the point and time where KIM is called. Furthermore, calls to KIM KC may be performant and later suffer due to the introduction of a custom, remote Matching service.

* The concrete KIM services (if configured as Singletons) make use of caching (using HashMap). Unfortunately there is no synchronization. This will cause more than just cache misses or phantom reads but can cause the Service to fail in unexpected ways. See org.kuali.rice.kim.service.impl.RoleServiceImpl.
* Some default Matching services use regex pattern matching on things like namespace code. This means that namespaces can be wildcarded. It also means that you CANNOT use regex meta-characters without the proper regex escape sequences. See NamespacePermissionTypeServiceImpl
* What is Rice’s strategy for adding new rows to rice tables? For example: Imagine release 1.0 of rice contains 10 KIM roles with primary keys of 1-10. KC adds a new role to KIM with id 11. When rice has a new release how is rice going to be able to add a new row without potentially conflicting with a custom KC role with the same primary key (11)? This is more of a generic question related to rice bootstrap data conflicting with application data.
* Old KIM had the concept of “attributes” and “qualified attributes”. An attribute is a piece of meta-data related to the item it is associated with. For example: the attribute “descend” is a piece of meta-data about a role. Descend is used to indicate whether members of this role have the option of access to subunits. This metadata does not affect whether a person is a part of a role or not. Qualified attributes on the other hand affect authorization by determining whether or not a person has certain qualifications. For example: a qualified attribute of kra.award=1 means that a person has a role if working with award number 1. New KIM only has the concept of qualified attributes. We can use these attributes to store metadata and define our own Matching services for matching but this is not really the correct purpose for these attributes.
* We need a way to easily store data about role-member data. For example: a role of “Creator” and member “quickstart” may have a unit.acl of 100000. This fits nicely into the member attribute data. How can we store active or descend information about this specific piece of member attribute data?

# KIM Permission Checking Walkthrough

//principal id is the id (in KIM) of the authenticated user

//can get the id by calling //GlobalVariables.getUserSession().getPrincipalId();

**String principalId = “1”;**

//the namespace the permission is “categorized” under

**String namespace = “KRA”;**

//the template name – not the permission name

**String permissionTemplateName = “create”;**

//Attributes used to find the permission

**AttributeSet permissionAttributes = new AttributeSet();**

**attributeSet.put(“foo”, *“bar”*);**

//Attributes used to “qualify” or further specify authorization criteria. //This assumes that the permission has been found and matched against the //permission attributes.

//The default implementation will cycle through any associated Groups, Roles, //Principals, Deletegates trying to match these qualifications. Any match //means the Principal is authorized. If qualifications are not specified //authorization is automatic assuming the permission was found.

**AttributeSet qualifyingAttributes = new AttributeSet();**

**attributeSet.put(“great\_band”, *“gnr”*);**

//A note on attribute matching

//KIM will use the service defined in the KRIM\_TYPE\_T table to perform //the //attribute match. The default implementation just checks that all //the //attributes are present as defined in the KRIM\_ATTR\_DEFN\_T and //have the //values defined in the associated KRIM\_PERM\_MBR\_ATTR\_DATA\_T

**KIMServiceLocator.getIdentityManagementService()**

**.isAuthorizedByTemplateName(principalId, namespace, permissionTemplateName, permissionAttributes, qualifyingAttributes);**

# Field Conversion Notes

A large portion of KC’s KIM Implementation is not used inside KC. As a result, the field conversions are incomplete because there is not data to convert. To determine what data was used and not used the KRADBA database and the KC codebase was analyzed.

# Old KIM Tables Not Used

The following tables are not used in KC. As a result, no migration analysis is documented.

KIM\_ATTRIBUTE\_TYPES\_T

KIM\_GROUPS\_T

KIM\_GROUPS\_ATTRIBUTES\_T

KIM\_GROUP\_QUAL\_ATTR\_T

KIM\_NAMESPACE\_DFLT\_ATTRIBS\_T

KIM\_PERSON\_ATTRIBUTES\_T

KIM\_PERSON\_QUAL\_ATTR\_T  
KIM\_ROLES\_GROUP\_QUAL\_T

KIM\_ROLES\_GROUPS\_T  
KIM\_ROLES\_PERSONS\_T  
KIM\_ROLE\_ATTRIBUTES\_T

# Field Conversion – KIM\_NAMESPACE\_T

No equivalent table exists for namespaces in KIM. KIM does reference “namespace codes” in many of the KIM tables. There are two solutions for this migration task. 1) Add a namespace table to Rice KIM, or 2) use the KIM\_NAMESPACE\_T.NAME column whenever referencing a namespace in KIM. For example: Currently in old KIM, the Permission “CREATE PROPOSAL” references the Namespace “KRA” via a foreign key. When we do the permissions migration the new KIM permission will reference the same namespace but by name and without a foreign key constraint.

# Field Conversion – KIM\_PERMISSIONS\_T

Every permission in the permissions table will have a permission template, permission, and KIM type.

For migration purposes we have a couple options. 1) We can have a new KIM type, template and permission for each old KIM permission, or 2) Create a generic KIM type, and template to categorize permission and then a new permission for each existing permission. The more natural solution is option number 2.

Using the example permissions, “CREATE\_PROTOCOL” and “CREATE\_PROPOSAL” for option number 2: First you would create a KIM Type entry defining a service to do permission matching. Then you would create a permission template called “CREATE” and associate it with the new KIM Type with a namespace of “KRA”. Finally, you would create two permissions referencing the CREATE template. These permissions would be called “CREATE\_PROTOCOL” and “CREATE\_PROPOSAL” but would be in their corresponding module namespaces (ex: KRA-PD, KRA-IRB).

|  |  |  |
| --- | --- | --- |
| KIM\_PERMISSIONS\_T | KIM table(s) | Notes |
| ID | KRIM\_PERM\_T.PERM\_ID | May have primary key conflicts with rice |
| NAME (VARCHAR(500)) | KRIM\_PERM\_T.NM (VARCHAR(100)) |  |
| DESCRIPTION (VARCHAR(4000)) | KRIM\_PERM\_T.DESC\_TXT (VARCHAR(400)) |  |
| NAMESPACE\_ID | KRIM\_PERM\_T.KRIM\_PERM\_T.NMSPC\_CD | Will now be a namespace code which is a module namespace |

As you can see the permission fits nicely in the new KIM table but several new tables must be populated.

# Field Conversion – KIM\_PERSONS\_T

|  |  |  |
| --- | --- | --- |
| KIM\_PERSONS\_T | KIM table(s) | Notes |
| ID | KRIM.PRNCPL\_T.PRNCPL\_ID |  |
| USERNAME (VARCHAR(500)) | KRIM.PRNCPL\_T.PRNCPL\_NM (VARCHAR(100)) |  |
| PASSWORD (VARCHAR(500)) | KRIM.PRNCPL\_T.PRNCPL\_PSWD (VARCHAR(400)) |  |
|  | KRIM.PRNCPL\_T.ACTV\_IND | set equal to related entry for PERSON.ACTIVE\_FLAG |
|  | KRIM.PRNCPL\_T.ENTITY\_ID | Set to related entity |

# Field Conversion – KIM\_ROLES\_PERMISSIONS\_T

|  |  |  |
| --- | --- | --- |
| KIM\_ROLES\_PERMISSIONS\_T | KIM table(s) | Notes |
| ROLE\_ID | KRIM\_ROLE\_PERM\_T.ROLE\_ID | Foreign key to KRIM\_ROLE\_T.ROLE\_ID |
| PERMISSION\_ID | KRIM\_ROLE\_PERM\_T.PERM\_ID | Foreign key to KRIM\_PERM\_T.PERM\_ID |
| ACTIVE\_FLAG | KRIM\_ROLE\_PERM\_T.ACTV\_IND |  |

# Field Conversion – KIM\_ROLES\_PERSONS\_QUAL\_T

|  |  |  |
| --- | --- | --- |
| KIM\_PERSONS\_QUAL\_T | KIM table(s) | Notes |
| ID | KRIM\_ROLE\_MBR\_ATTR\_DATA.ATTR\_DATA\_ID |  |
| ROLE\_PERSON\_ID | KRIM\_ROLE\_MBR\_ATTR\_DATA.TARGET\_PRIMARY\_KEY | Must have entry in KRIM\_ROLE\_MBR\_T associating the role and principal for this to work. |
| ATTRIBUTE\_NAME | KRIM\_ROLE\_MBR\_ATTR\_DATA.KIM\_TYPE\_ID, KRIM\_ROLE\_MBR\_ATTR\_DATA.ATTR\_DEFN\_ID | The KIM\_TYPE & DEFN define the specific attribute name that value is associated with |
| ATTRIBUTE\_VALUE | KRIM\_ROLE\_MBR\_ATTR\_DATA.ATTR\_VAL |  |
|  |  |  |

# Field Conversion – KIM\_ROLES\_T

|  |  |  |
| --- | --- | --- |
| KIM\_ROLES\_T | KIM table(s) | Notes |
| ID | KRIM\_ROLE\_T.ROLE\_ID | May have primary key conflicts with rice |
| NAME (VARCHAR(500)) | KRIM\_ROLE\_ T.PERM\_NM (VARCHAR(80)) |  |
| DESCRIPTION | KRIM\_ROLE \_T.DESC\_TXT |  |
| ROLE\_TYPE |  | Will discuss later |
| DESCEND |  | Will discuss later |
|  | NMSPC\_CD | Set to namespace that this applies to. Consider using a module namespace. |

# Field Conversion – KIM\_ROLES\_TYPE\_T

This is essentially metadata related to a role. The best solution is for rice to have a metadata facility for KIM types. Since this does not exist we could use nested roles, or encode the type in the name, or create our own KC tables to associate a role with a piece of data like type. These are not very good solutions compared to KIM supporting this requirement out of the box.

# Field Conversion – UNIT\_ACL

The unit can be stored as role-member attribute data easily. The descend and active flags are problematic in that there is now easy way to store a value associated with a specific role-member attribute data. We could encode two values into the ATTR\_VAL field or create our own table to keep track of the descent flag.

|  |  |  |
| --- | --- | --- |
| UNIT\_ACL | KIM table(s) | Notes |
| ID | KRIM\_ROLE\_MBR\_ATTR\_DATA.ATTR\_DATA\_ID |  |
| ROLE\_ID |  | Must have entry in KRIM\_ROLE\_MBR\_T associating the role and principal for this to work. |
| PERSON\_ID |  | Must have entry in KRIM\_ROLE\_MBR\_T associating the role and principal for this to work. |
| ATTRIBUTE\_VALUE | KRIM\_ROLE\_MBR\_ATTR\_DATA.ATTR\_VAL |  |
| DESCEND |  | There isn’t really a facility to support the descend flag for a specific piece of role-member attr data. |
| ACTIVE | KRIM\_ROLE\_MBR\_T.ACTV\_FROM, KRIM\_ROLE\_MBR\_T.ACTV\_TO | Could use these two column but this is not quite the same. This would make the entire role/mbr combination now active not a specific unit. |