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At the time of this analysis, there is no available documentation related to new KIM and only outdated documentation for old KIM.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.”

# An example

Warning, this example assumes knowledge of common KIM terminology. See the Definitions section for more info. This example assumes a certain permission configuration. KC’s final configuration may not match this example. The diagrams shown are a simplification of the physical database relationships.

**Use case: does an entity with principal id 1 have permission to create an award document in KRA under unit number 1234?**

Take a moment to analyze the following Java code. What KIM is doing behind the scenes will be explained next.

//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(“docType”, *“award”*);**

//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(“unitNumber”, *“1234”*);**

//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);**

First KIM uses the namespace code, permission template name, and permission attributes to find all the matching permissions under the requested template. For example:



Notice in the above diagram that 2 permissions exist (or are grouped) under the permission template but only one permission has a docType of award. In this case, we are assuming that a docType match is required in order to have a permission. This is completely customizable. To determine a match, the Permission Template’s serviceName (matching service) is used to find matching permissions. In fact it may choose to return a permission that is not even associated with a permission template at the database level. For simplicity, we will assume that the createService returns permissions that have a matching docType and are related to the create permission template.

Now that KIM has found a permission that matches the namespace code, permission template name, and permission attributes, KIM checks to see if the passed in principal has that permission.

It does this by getting all the roles associated with that permission. Then it checks if a principal is a part of that role.

Let’s assume 2 roles are returned that are associated with the create permission and the principal with id 1.



For each Role returned (in this case 2) KIM will invoke the KIMType Service to decide if there is a match (again the service is customizable). Let’s assume the service just makes sure that the unitNumber is a match with the qualifying attributes passed in by the client. In this case the principal is authorized because the principal is a member of the create role AND that association has a role member attribute of unitNumber=1234.

That is authorization in a nutshell. There is much more that could happen but this is one of the simplier use-cases. Phew!

Geoff: Do you think we are going to need to write a custom service for any Permission Templates or Kim Types? Probably not, right?

# 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.
* **Qualifying 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.
* **Regular Attribute:** A Name/Value pair used to include additional meta-data about a KIM object (role, person, etc.). This data is accessible to the consumer of KIM APIs. These attributes are only available in KC’s KIM implementation.
* **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. Geoff: I thought delegation was a way for one principal to 'grant' his/her roles to another principal. Is this correct or is it something else?

# 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 + attribute encoding scheme is complex relative to the task at hand.

* KIM no longer defines a namespace in a table. Many of the KIM tables still have a namespace code but there is no referential integrity to make 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. In addition, 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 when reading/writing to the Service’s member data. 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. Geoff: Is this fixed by: https://test.kuali.org/jira/browse/KULRICE-3221
* 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. This is more of an FYI then serious problem.
* 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. Geoff: Are these PKs not synthetic keys generated by sequences or something?
* Old KIM had the concept of “attributes (regular)” and “qualified attributes”. An attribute is a piece of metadata related to the item it is associated with. For example: the attribute “role type” is a piece of meta-data about a role. Role Type is used to further categorize a role. 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 but this is not really the correct purpose for these attributes. Geoff: Do we actually use these in old KIM? In the Old KIM Tables Not Used section, a lot of Attributes tables are listed.
* 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? Geoff: storing descends, can this be another row in the role-member table?
* Authorization checking can be done by permission name or permission template name. I don’t fully understand the benefits of one over the other. Rice code seems to use templates. Hopefully, the rice team can help us with this.

# 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/KRACNV databases and the KC codebase was analyzed.

There are many Foreign Key Relationships in Old KIM. There is also bootstrap data in New KIM. Even though the Field Conversions show a mapping from old KIM to new KIM for primary Keys/foreign keys new key values may need to be chosen because of New KIM’s bootstrap data. Keep this in mind when doing the data migration.

Since KC’s KIM data doesn’t all fit perfectly into Rice KIM some migration information may need to change.

# 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 similar permissions and then a new permission for each existing permission. The more natural solution is option number 2. Geoff: Can our permissions fit into any existing bootstrap types/templates?

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 |  | discussed elsewhere (metadata problem) |
| DESCEND |  | discussed elsewhere (metadata problem) |
|  | 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. Geoff: What if we lose this data when we migrate. Will that be a problem? Is it used for anything now?

# 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 no 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. |

# Items to Delete

Now that all the data is migrated, it’s time to start hitting the delete key. The Following items will be removed:

Database Tables:

* KIM\_ATTRIBUTE\_TYPES\_T
* KIM\_GROUPS\_GROUPS\_T
* KIM\_GROUPS\_PERSONS\_T  
  KIM\_GROUPS\_T
* KIM\_GROUP\_ATTRIBUTES\_T
* KIM\_GROUP\_QUAL\_ATTR\_T
* KIM\_NAMESPACES\_T
* KIM\_NAMESPACE\_DFLT\_ATTRIBS\_T
* KIM\_PERMISSIONS\_T
* KIM\_PERSONS\_T
* KIM\_PERSON\_ATTRIBUTES\_T
* KIM\_PERSON\_QUAL\_ATTR\_T
* KIM\_ROLES\_GROUPS\_QUAL\_T
* KIM\_ROLES\_GROUPS\_T
* KIM\_ROLES\_PERMISSIONS\_T
* KIM\_ROLES\_PERSONS\_QUAL\_T
* KIM\_ROLES\_PERSONS\_T
* KIM\_ROLES\_T
* KIM\_ROLE\_ATTRIBUTES\_T
* UNIT\_ACL

Java classes (including test classes):

* org.kuali.kra.kim.\*
* org.kuali.kra.bo.UnitAclEntry

OJB Mapping for aforementioned tables

DataDictionary, DataDictionaryMaintennance, KEW files for:

* org.kuali.kra.kim.bo.\*
* org.kuali.kra.bo.UnitAclEntry

Spring Bean definitions:

* org.kuali.kra.kim.\*

# \*\*Insert Info Here about changes to Java Code, Xml, etc.\*\*

# Creating a Matching Service for Permission Template

This example creates a permission matching service that only finds permissions named “Foo” in namespace “Bar” and matching attributes or an attribute called “secretDocNumber” with a value of “1234”.

To reiterate a previous potential issue, this service must be remotely accessible from KIM for KIM to work as a central installation. This could have performance implications.

/\*\*

\* A match occurs only if the permission name is "Foo", the namespace code is "Bar"

\* and the doc number is "1234" or

\* and the requestedDetails match the details on the permission.

\*/

**public** **class** FooBarPermMatcher **extends** KimPermissionTypeServiceBase {

/\*\*

\* {@inheritDoc}

\*/

@Override

**protected** List<KimPermissionInfo> performPermissionMatches(AttributeSet requestedDetails,

List<KimPermissionInfo> permissionsList) {

List<KimPermissionInfo> matchingPermissions = **new** ArrayList<KimPermissionInfo>();

**for** (KimPermissionInfo permission : permissionsList) {

**if** (**this**.isFooBar(permission) && (**this**.isCorrectDocNumber(permission)

|| **this**.performMatch(requestedDetails, permission.getDetails()))) {

matchingPermissions.add(permission);

}

}

**return** matchingPermissions;

}

/\*\*

\* Checks for permission name "Foo", the namespace code "Bar"

\* **@param** permission the permission info

\* **@return** true if a foo bar perm.

\*/

**private** **boolean** isFooBar(KimPermissionInfo permission) {

**return** "Foo".equals(permission.getName()) && "Bar".equals(permission.getNamespaceCode());

}

/\*\*

\* Checks for whether the permission has an attribute called "secretDocNumber"

\* with a value of "1234"

\* **@param** permission the permission info

\* **@return** true if doc number attribute is a match.

\*/

**private** **boolean** isCorrectDocNumber(KimPermissionInfo permission) {

**return** "1234".equals(permission.getDetails().get("secretDocNumber"));

}

After creating this Service it must be registered in KIM. In order for KIM to use this service the Spring Service name must be place in the KIMType’s service name field for a Permission template. With this service in place, only permissions with the aforementioned criteria will be returned. What this means to the client application is that a user is NOT authorized if no permissions are returned.

# Creating a Matching Service for Role

Checks if a the value for the Foo attribute matches a role's Foo attribute qualifier.

To reiterate a previous potential issue, this service must be remotely accessible from KIM for KIM to work as a central installation. This could have performance implications.

/\*\*

\* Checks if a the value for the Foo attribute matches a role's

\* Foo attribute qualifier.

\*/

**public** **class** FooRoleTypeServiceImpl **extends** KimRoleTypeServiceBase {

/\*\*\*

\* {@inheritDoc}

\*/

@Override

**public** **boolean** doesRoleQualifierMatchQualification(AttributeSet qualification, AttributeSet roleQualifier) {

**return** StringUtils.*equals*(qualification.get("Foo"),

roleQualifier.get("Foo"));

}

}

After creating this Service it must be registered in KIM. In order for KIM to use this service the Spring Service name must be place in the KIMType’s service name field for a Role. doesRoleQualifierMatchQualification will return true only when the aforementioned criteria is true. What this means to the client application is that a user is NOT authorized if true is not returned (unless a qualifier match is not found elsewhere.

General comments:

It’s still a bit fuzzy, but this has made my understanding of KIM internal workings much more clear. Having this level of knowledge before we actually try to use this framework will be of great benefit I think.

There are a couple of general questions I still have. This doc talks a lot about the configurability of KIM – how it allows you to write custom services at various points, etc. But I wasn’t clear a lot of the time whether we were actually going to have to do this, or whether the default out of the box implementations would satisfy our requirements. And maybe we don’t know that yet but even in that case a statement like ‘it isn’t clear yet how much customization we’ll have to do’ would be useful. Maybe it’s just me but it seems like the matching service examples above could be abstracted to work for lots of different roles/permission templates, rather than having to write one for every role/permission template you want to add to the system. I’ll be a little surprised if it turns out KIM doesn’t provide that somehow.

Also the changes to java code section, it would be nice to have some info there as I’m sure you intend to provide. It doesn’t have to be detailed down to the individual method call level but it would be nice to have an idea of what KIM services we’ll be calling from our Authorizers. My general understanding is that while the KIM module contains lots of services, only a few of them (maybe only 1 or 2?) are intended to be called directly from client code.

Much thanks!