KC KIM Migration: KIM Exposed

The following is the conversion to migrate the KC KIM (old KIM) to Rice KIM (new KIM or KIM).

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At the time of this analysis, there is no available documentation related to new KIM and only outdated documentation for old KIM.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

# 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 callback service. Both are optional. Group, Role, Permission, Responsibility, and Delegation are KimTypes.
* **Callback service:** KIM will invoke the callback 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 callback service passing all the attribute values encoded into a single attribute. Calling KIM multiple times may have performance implications while defining a callback 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 is a namespace must define. **Is this something that KC needs?**
* To customize authorization, KIM has the idea of a callback 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 callback 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 callback 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 callback 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. See org.kuali.rice.kim.service.impl.RoleServiceImpl.

# 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);**