Kuali Rice 2.1.7 Technical Reference Guide

Table of Contents

List of Figures

List of Tables

List of Examples

Chapter 1. Global

Rice Client Overview

You can integrate your application with Rice using several methods, each described below.

Embedded

This method includes embedding some or all of the Rice services into your application. When using this method, a standalone Rice server for the Rice web application is still required to host the GUI screens and some of the core services.

To embed the various Rice modules in your application, you configure them in the RiceConfigurer using Spring. For more details on how to configure the RiceConfigurer for the different modules, please read the Configuration Section in the Technical Resource Guide for the module you want to embed.

Figure 1.1. Diagram of a sample embedded implementation
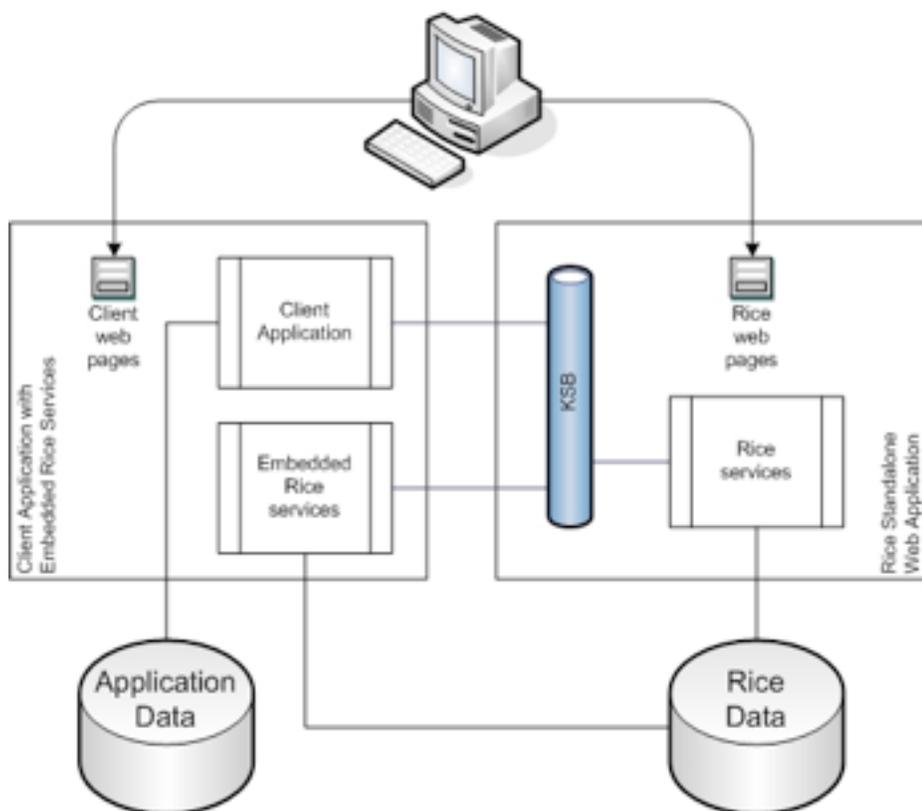
Advantages

• Integration of database transactions between client application and embedded Rice (via JTA)

• Performance: Embedded services talk directly to the Rice database

• No need for application plug-ins on the server

- Great for Enterprise deployment: It's still a single Rice web application, but scalability is increased because there are multiple instances of embedded services.

Disadvantages

- Can only be used by Java clients

- More library dependencies than the Thin Client method

- Requires client access to the Rice database

Bundled

This method includes the entire Rice web application and all services into your application. This method does not require a standalone Rice server.

Each of the Rice modules provides a set of JSPs and tag libraries that you include in your application. These are then embedded and hooked up as Struts Modules. For more details on how the web portion of each module is configured, please read the Configuration Guide for each of the modules.

Figure 1.2. Diagram of a sample bundled implementation
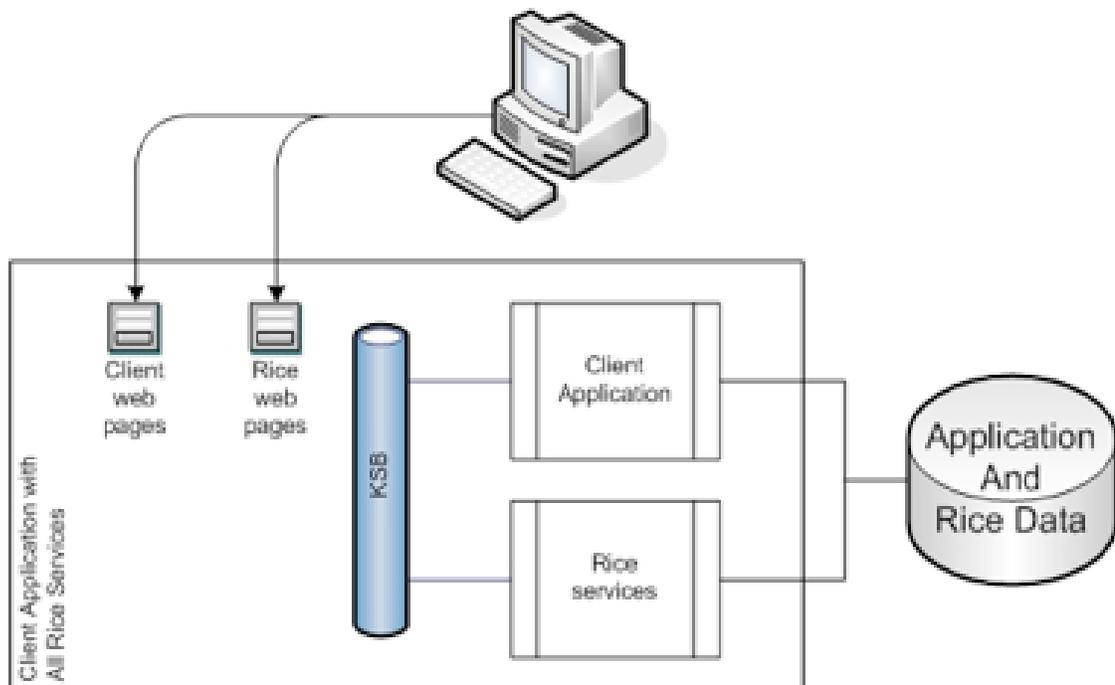
Advantages

- All the advantages of Embedded Method

- No need to deploy a standalone Rice server

- Ideal for development or quick-start applications

- May ease development and distribution

- Can switch to Embedded Method for deployment in an Enterprise environment

Disadvantages

- Not desirable for Enterprise deployment when more than one application is integrated with Rice

- More library dependencies than the Thin Client method and the Embedded Method (since it requires additional web libraries).

Thin Java Client

This method utilizes some pre-built classes to provide an interface between your application and web services on a standalone Rice server.

Many of the Rice services are exposed by the KSB as Java service endpoints. This means they use Java Serialization over HTTP to communicate. If desired, they can also be secured to provide access to only those callers with authorized digital signatures.

Figure 1.3. Diagram of a sample Thin Java Client implementation
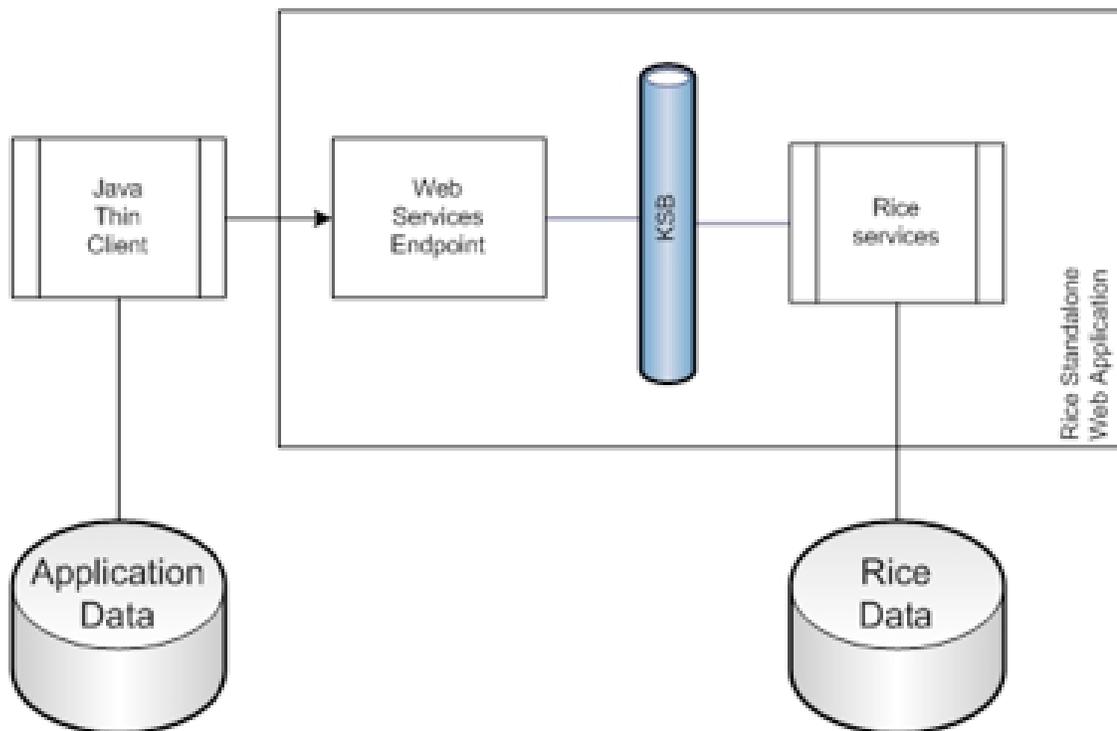
Advantages

- Relatively simple and lightweight configuration

- Fewer library dependencies

Disadvantages

- No transactional integration between client and server

- Plug-ins must be deployed to the server if custom Rice components are needed

Web Services

This means directly using web services to access a standalone Rice server. This method utilizes the same services as the Thin Java Client, but does not take advantage of pre-built binding code to access those services.

Advantages

- Any language that supports SOAP web services can be used

Disadvantages

- No transactional integration between client and server

- Plug-ins must be deployed to the server if custom Rice components are needed

- Web Services can be slower than other integration options

Global Configuration Parameters

Table 1.1. Global Configuration Parameters

Configuration Parameter	Description	Sample value
app.code	Together with environment, forms the app.context.name which then forms the application URL.	kr
application.id	The unique ID for the application. A value should be chosen which will be unique within the scope of Kuali Rice deployment and integration. There is no default for this value but it must be defined in order for portions of Kuali Rice to function properly.	
application.host	The name of the application server the application is being run on.	localhost
application.http.scheme	The protocol the application runs over.	http
cas.url	The base URL for CAS services and pages.	https://test.kuali.org/cas-stg
config.obj.file	The central OJB configuration file.	
config.spring.file	Used to specify the base Spring configuration file. The default value is "classpath:org/kuali/rice/kew/config/KEWSpringBeans.xml"	
credentialsSourceFactory	The name of the org.kuali.rice.core.security.credentials.CredentialsSourceFactory bean to use for credentials to calls on the service bus.	
datasource.accessToUnderlyingConnectionAllowed	Allows the data source's pool guard access to the underlying data connection. See: http://commons.apache.org/dbcp/apidocs/org/apache/commons/dbcp/BasicDataSource.html#isAccessToUnderlyingConnectionAllowed()	true
datasource.initialSize	The initial number of database connections in the data source pool. See: http://commons.apache.org/dbcp/apidocs/org/apache/commons/dbcp/ BasicDataSource.html#initialSize	7
datasource.minIdle	The number of connections in the pool which can be idle without new connections being created. See: http://commons.apache.org/dbcp/apidocs/org/apache/commons/dbcp/ BasicDataSource.html#minIdle	7
datasource.ojb.sequenceManager.className	The class used to manage database sequences in databases which do not support that feature. Default value is "org.apache.ojb.broker.platforms.KualiMySQLSequenceManagerImpl"	
datasource.pool.maxActive	The maximum number of connections allowed in the data source pool. See: http://commons.apache.org/dbcp/apidocs/org/apache/commons/dbcp/ BasicDataSource.html#maxActive	50

Configuration Parameter	Description	Sample value
environment	The name of the environment. This will be used to determine if the environment the application is working within is a production environment or not. It is also used generally to express the "name" of the environment, for instance in the URL.	dev
http.port	The port that the application server uses; it will be appended to all URLs within the application.	8080
log4j.settings.props	The log4j properties of the application, set up in property form.	
log4j.settings.xml	The log4j properties of the application, set up in XML form.	
rice.additionalSpringFiles	A comma delimited list of extra Spring files to load when the application starts.	
additional.config.locations	A comma delimited list of additional configuration file locations to load after the main configuration files have been loaded. Note that this parameter only applies to the Rice standalone server.	
rice.custom.ojb.properties	The file where OJB properties for the Rice application can be found. The default is "org/kuali/rice/core/ojb/RiceOJB.properties"	org/kuali/rice/core/ojb/RiceOJB.properties
rice.cache.disableAllCaches	Flag to disable all Spring caching in Rice	false
rice.cache.disabledCaches	Flag to disable specific Spring caches in Rice by name. The cache names should be comma separated.	http://rice.kuali.org/kim/v2_0/PermissionType, http://rice.kuali.org/kim/v2_0/TemplateType{Permission}
rice.logging.configure	Determines whether the logging lifecycle should be loaded.	false
rice.url	The main URL to the Rice application.	${application.url}/kr
security.directory	The location where security properties exist, such as the user name and password to the database.	/usr/local/rice/
transaction.timeout	The length of time a transaction has to complete; if it goes over this value, the transaction will be rolled back.	300000
version	The version of the Rice application.	03/19/2007 01:59 PM

Rice Service Architecture and Configuration Overview

This document describes how the Rice Service Architecture operates.

Overview

The Rice System consists of a stack of ResourceLoader objects that contain configuration information and expose service implementations (potentially from remote sources). Each module supplies its own Spring context containing it's services. These Spring contexts are then wrapped by a ResourceLoader which is used to locate and load those services.

Implementation Details

Rice is composed of a set of modules that provide distinct functionality and expose various services. Each module loads it's own Spring context which contains numerous services. These Spring contexts are wrapped by a ResourceLoader class that provides access to those services. A ResourceLoader is similar to Spring's BeanFactory interface, since you acquire instances of services by name. Rice adds several additional concepts, including qualification of service names by namespaces. When the RiceConfigurer is instantiated, it constructs a GlobalResourceLoader which contains an ordered chain of ResourceLoader instances to load services from:

Figure 1.4. Resource Loader Stack
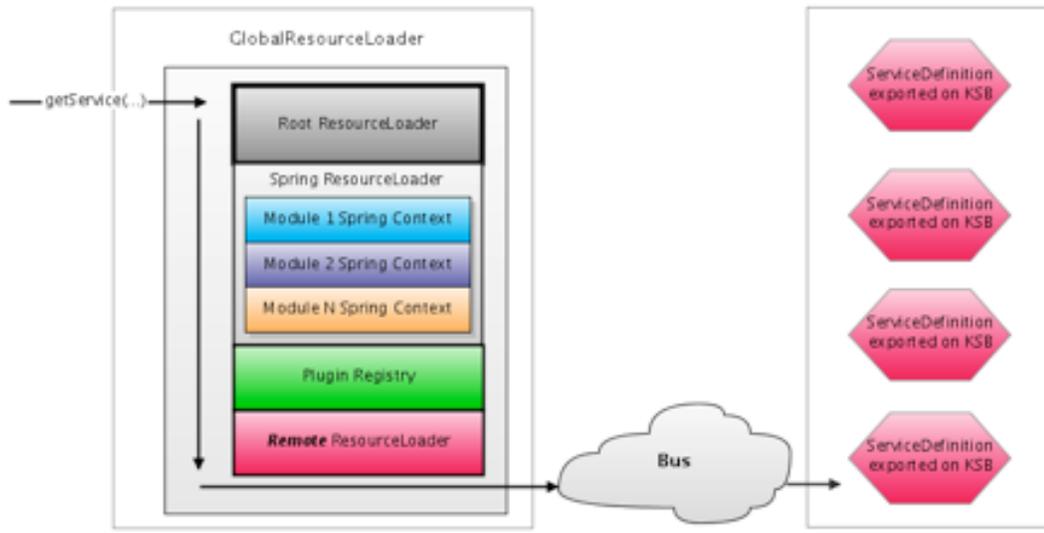
All application code should use the GlobalResourceLoader to obtain service instances. The getService(…) method iterates through each registered ResourceLoader to locate a service registered with the specified name. In it's default configuration, the GlobalResourceLoader contacts the following resource loaders in the specified order:

1. **Spring ResourceLoader** – wraps the spring contexts for the various Rice modules

2. **Plugin Registry** – allows for services and classes from to be loaded from packaged plugins

3. **Remote ResourceLoader** – integrates with the KSB ServiceRegistry to locate and load remotely deployed services

As shown above, the last ResourceLoader on the list is the one registered by KSB to expose services available on the service bus. It's important that this resource loader is consulted last because it gives priority to using locally deployed services over remote services (if the service is available both locally and remotely). This is meant to help maximize performance.

Thin Client Implementation

To implement a thin client version of Rice, modify the configuration files as per the following

config.xml:

```
<config>
    <param name="environment" override="false">dev</param>
    <param name="application.id">rice-remote-test-client</param>
    <param name="message.persistence">false</param>
    <param name="kim.mode">THIN</param>
    <param name="kew.mode">THIN</param>
    <param name="ksb.mode">THIN</param>
    <param name="standalone.application.id">TRAVEL</param>
 <param name="config.location">/root/kuali/main/${environment}/rice-remote-test-client-config.xml</param>
    <param name="config.location">classpath:META-INF/common-config-defaults.xml</param>
</config>
```

SpringBeans.xml:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.1.xsd">

<bean id="jtaTransactionManager" class="org.springframework.transaction.jta.JotmFactoryBean">
    <property name="defaultTimeout" value="3600"/>
</bean>

<bean id="bootstrapConfig" class="org.kuali.rice.core.impl.config.property.ConfigFactoryBean"
            p:initialize="true">
    <property name="configLocations">
   <list>
    <value>classpath:config.xml</value>
   </list>
 </property>
</bean>

<bean id="coreConfigurer" class="org.kuali.rice.core.impl.config.module.CoreConfigurer" depends-
on="bootstrapConfig"
           p:transactionManager-ref="jtaTransactionManager"
           p:userTransaction-ref="jtaTransactionManager"/>

<bean id="ksbConfigurer" class="org.kuali.rice.ksb.messaging.config.KSBConfigurer"/>

<bean id="kimConfigurer" class="org.kuali.rice.kim.config.KIMConfigurer"/>

<bean id="kewConfigurer" class="org.kuali.rice.kew.config.KEWConfigurer"/>

</beans>
```

Accessing Rice Services and Beans Using Spring

Rice Service as a Spring Bean

In addition to programmatically acquiring service references, you can also import Rice services into a Spring context with the help of the ResourceLoaderServiceFactoryBean:

```
<!-- import a Rice service from the ResourceLoader stack -->
<bean id="aRiceService" class="org.kuali.rice.resourceloader.support.ResourceLoaderServiceFactoryBean"/>
```

This class uses the GlobalResourceLoader to locate a service named the same as the ID and produces a bean that proxies that service. The bean can thereafter be wired in Spring like any other bean.

Using Annotations

Rice includes a Spring bean that extends the Spring auto-wire process (unlike the current version of Spring, the auto-wire process in the version of Spring that's included with Rice cannot be extended). With this bean configured into your application, you can use the @RiceService annotation to identify Rice services to auto-wire.

Add this bean definition to the top of your Spring configuration file to configure the Spring extension:

```
<bean class="org.kuali.rice.core.util.GRLServiceInjectionPostProcessor"/>
```

Add the @RiceService annotation to any field or method, following the normal Spring rules for injection annotations. The annotation requires a name property that specifies the name of the service to inject. If the

name requires a namespace other than the current context namespace, you must specify the namespace as a prefix (for example, "{KEW}actionListService").

```
@RiceService(name="workflowDocumentService")
protected WorkflowDocumentService workflowDocumentService;
```

Publishing Spring Services to the Global Resource Loader

In certain cases, it may be desirable to publish all beans in a particular Spring context to the Resource Loader stack. Fortunately, there is an easy way to accomplish this using the RiceSpringResourceLoaderConfigurer as shown below:

```
<!-- Publish all services from this Spring context to the GRL -->
<bean class="org.kuali.rice.core.resourceloader.RiceSpringResourceLoaderConfigurer"/>

<bean id="myService1" class="my.app.package.MyService1"/>

<bean id="myService2" class="my.app.package.MyService2"/>
```

In the above example, both myService1 and myService2 would be added to a Resource Loader that would be put at the top of the Resource Loader stack. The names of these services would be "myService1" and "myService2" with no namespace. To load these services you would use the following call to the Global Resource Loader:

```
MyService1 myService1 = GlobalResourceLoader.getService("myService1");
```

Customizing and Overriding Rice Services

Reasons for Overriding Services

The most common reason that one would want to override services in Kuali Rice is to customize the implementation of a particular service for the purposes of institutional customization.

A good example of this is the Kuali Identity Management (KIM) services. KIM is bundled with reference implementations that read identity (and other) data from the KIM database tables. In many cases an implementer will already have an existing identity management solution that they would like to integrate with. By overriding the service reference implementation with a custom one, it's possible to integrate with other institutional services (such as LDAP or other services).

Installing an Application Root Resource Loader

An alternative to using the RiceSpringResourceLoaderConfigurer to publish beans from a Spring context to the Rice Resource Loader framework is to inject a root Resource Loader into the RiceConfigurer.

You can create an implementation of ResourceLoader that returns a custom bean instead of the Rice bean, or you can use a built-in resource loader like the SpringBeanFactoryResourceLoader which wraps a Spring context in a ResourceLoader. Your configuration needs to inject this bean as the RootResourceLoader of the RiceConfigurer using the rootResourceLoader property, as shown below:

```
<!-- a Rice bean we want to override in our application -->
<bean id="overriddenRiceBean" class="my.app.package.MyRiceServiceImpl"/>
```

```
<!-- supplies services from this Spring context -->
<bean id="appResourceLoader"
 class="org.kuali.rice.core.resourceloader.SpringBeanFactoryResourceLoader"/>

<bean id="rice" class="org.kuali.rice.core.config.RiceConfigurer">
    <property name="rootResourceLoader" ref="appResourceLoader"/>
    ...
</bean>
```

Warning

Application Resource Loader and Circular Dependencies

Be careful when mixing registration of an application root resource loader and lookup of Rice services via the GlobalResourceLoader. If you are using an application resource loader to override a Rice bean, but one of your application beans requires that bean to be injected during startup, you may create a circular dependency. In this case, you have to make sure you are not unintentionally exposing application beans (which may not yet have been fully initialized by Spring) in the application resource loader, or you have to arrange for the GRL lookup to occur lazily, after Spring initialization has completed (either programmatically, or via some sort of proxy).

Replacing Rice Configuration Files

A Rice-enabled web application (including the Rice Standalone distribution) contains a RiceConfigurer (typically defined in a Spring XML file) that loads the Rice modules. You can override services from the various modules by injecting a list of additional spring files to load as in the following example:

```
<bean id="rice" class="org.kuali.rice.core.config.RiceConfigurer">
    ...
    <property name="additionalSpringFiles" ref="appResourceLoader">
        <list>
             <value>classpath:my/app/package/MyCustomSpringFile.xml</value>
        </list>
    </property>
    ...
</bean>
```

You will need to ensure that any Spring XML files and necessary classes they reference are in the classpath of your application. If you are overriding things in the Rice standlone application itself, then you would need to place classes in the **WEB-INF/classes** directory of the war and any jars in the **WEB-INF/lib** directory.

It's a standard behavior of Spring context loading that the last beans found in the context with a particular id will be the versions loaded during context initialization. The **additionalSpringFiles** property will put any Spring files specified at the end of the list loaded by the RiceConfigurer. So any beans defined in that file with the same id as beans in the internal Rice Spring XML files will effectively override the out-of-the-box version of those services.

When working with the packaged Rice standalone server, you won't have access to the Spring XML file which configures the RiceConfigurer. In this case, you can specify additional spring files using a configuration parameter in your Rice configuration XML, as in the following example:

```
<param name="rice.additionalSpringFiles"
value="classpath:my/app/package/MyCustomSpringFile.xml"/>
```

Eclipse and Rice

Warning

Recent change in Eclipse setup

Due to its unreliability, we have recently stopped relying on the Maven plugin for Eclipse to manage the project build path. Instead, we are using the eclipse:eclipse plugin for Maven to generate a static build path. Please note the changes in the Eclipse project setup.

Overview

This document describes how to set up an Eclipse environment for running Rice from source and/or for developing on the Kuali Rice project. To create your own Kuali Rice client application, see the instructions in Creating a Rice-Enabled Application.

Download the Tools

1. Install Java 5 SDK - http://java.sun.com.

2. Install the Eclipse Europa Bundle for Java Developers - http://www.eclipse.org/europa/

   - You need to allocate at least 768MB of memory for the Eclipse runtime and at least 512MB of memory for the JVM that Eclipse uses when it runs Java programs and commands.

   - Go to **Eclipse Preferences**.

   - On Windows: *Window --> Preferences --> Java --> Installed JREs*.

   - On Mac OS X: *Eclipse --> Preferences --> Java --> Installed JREs*.

   - Select the JRE and click **Edit**.

   - Add -Xmx768m to **Default VM Arguments**

3. Install Maven2 for command line usage:

   - Download Maven2.0.9 from http://maven.apache.org/download.html.

   - Install Maven2 into **C:\maven** on Windows or **/opt/maven** on Linux. This directory is called the Maven Root directory.

   - Register Maven on your computer's **PATH** so that it can be invoked as an executable without have to run the **mvn** command from the **<maven_root>/bin** directory all of the time.

   - Set the **M2_HOME** environment variable on your system to the location of your Maven2 installation.

4. Update **.m2** repository directory (WINDOWS ONLY) By default (on Windows) maven places the **.m2** repo directory in the user directory inside the **Documents and Settings** folder. The space characters can cause issues. To avoid them we need to do the following:

   a. Figure out where you want your local maven repository to be stored, i.e. **C:\work\m2**

   b. Make sure you turn off eclipse if it has auto updating maven turned on.

c. Move everything from your old maven directory to your new one. This will save you a considerable amount of time. If you do not do this then maven will re-download all repositories to the new location.

d. Update your settings.xml file. This should be located in **C:\Documents and Settings\user \.m2\settings.xml**. Add this line to the file somewhere inside the <settings> tag:

```
<localRepository>C:\work\m2</localRepository>
```

Import rice into Eclipse as a project (Source distribution only)

Note: You only need to follow these instructions if you downloaded the source distribution of Rice as a zip file. If you are a contributing developer who will be committing code to CVS, please skip this step (Importing rice into Eclipse as a Project) and go to the next one instead.

1. Open Eclipse.

2. Choose *File --> Import --> Existing Projects into Workspace*.

Figure 1.5. Root Directory Selection
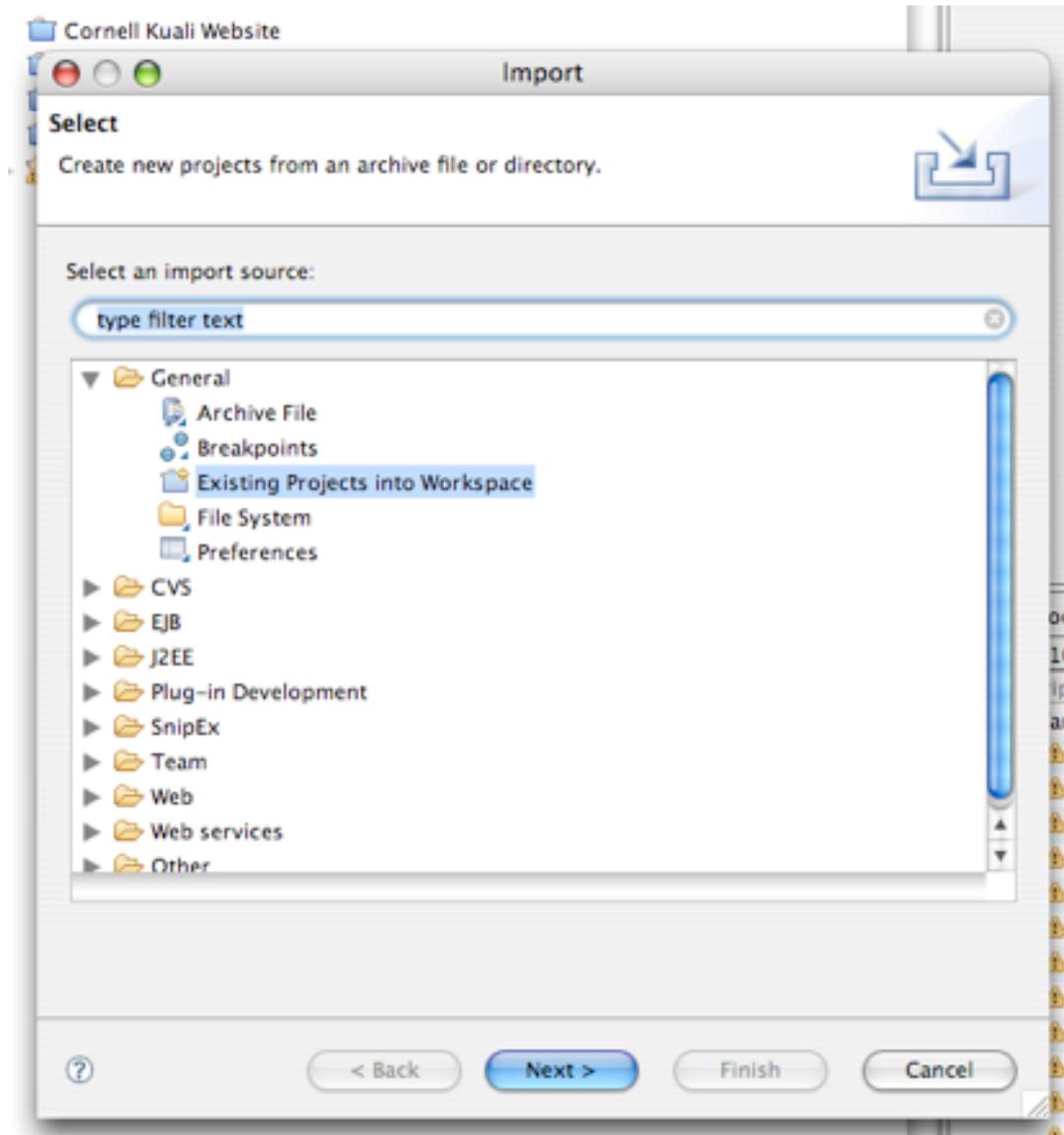
3. Browse for and select **/java/projects/rice** (or where ever you unzipped the source distribution to) as the root project directory and click Finish.

Figure 1.6. Root Directory Selection Continued
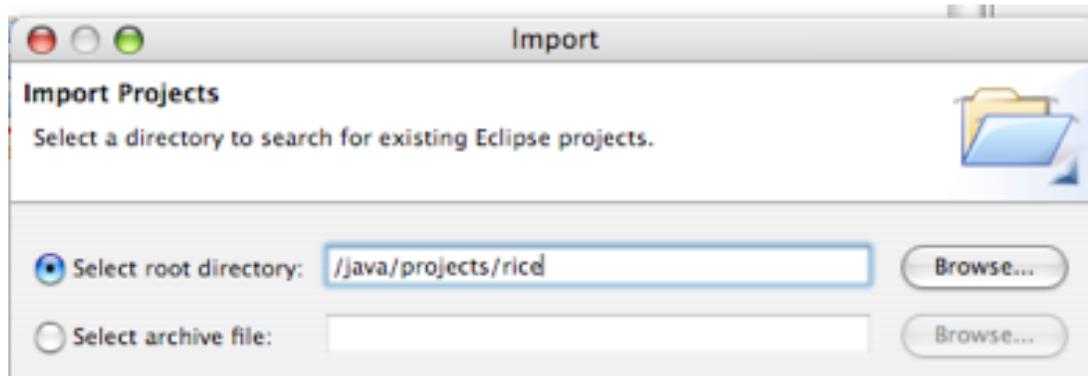
Check out the Rice code (Non-source SVN distribution only)

Note: You do not need to perform the steps in this section if you have downloaded the source distribution of Rice as a zip file.

1. We recommend installing Subclipse as a plugin from your Eclipse instance (http://subclipse.tigris.org/install.html)

2. Set up a new SVN repository in Eclipse: http://svn.kuali.org/repos/rice

3. Check out the Rice code from the appropriate branch of code (i.e. branches/rice-release-1-0-0-br)

Set up database drivers

Oracle

1. If this is the first time you've set up Eclipse to work with Rice, Maven won't find the Oracle drivers in the Kuali repository.

2. If you do not already have an Oracle driver saved in **/java/drivers** as **ojdbc14.jar**, you can download one from http://www.oracle.com/technology/software/tech/java/sqlj_jdbc/index.html. Save it as **/java/drivers/ojdbc14.jar**

3. Run this command from the command line (this should all be on one line when you enter it):

   UNIX

```
mvn install:install-file -DgroupId=com.oracle -DartifactId=ojdbc14
 -Dversion=10.2.0.3.0 -Dpackaging=jar -Dfile=/java/drivers/ojdbc14.jar
```

   Windows

```
mvn install:install-file -DgroupId=com.oracle -DartifactId=ojdbc14
 -Dversion=10.2.0.3.0 -Dpackaging=jar -Dfile=c:/java/drivers/ojdbc14.jar
```

Or, run the equivalent Ant target:

```
ant install-oracle-jar
```

Other databases

The driver for MySQL is already referenced by the Kuali Rice project. Rice does not have out-of-the-box support for other RDBMS at this point in time. However, if you want to use other databases, it is possible to add database support for that particular database as long as it's supported by the Apache OJB project (http://db.apache.org/ojb).

Set up Eclipse for Maven

If this is the first time you are using Eclipse with a project build path generated by the eclipse:eclipse Maven plugin, you need to define the **M2_REPO** Classpath Variable in your Eclipse: *Java > Build Path > Classpath Variable*, under the Preferences menu.

Figure 1.7. Eclipse Classpath Variables
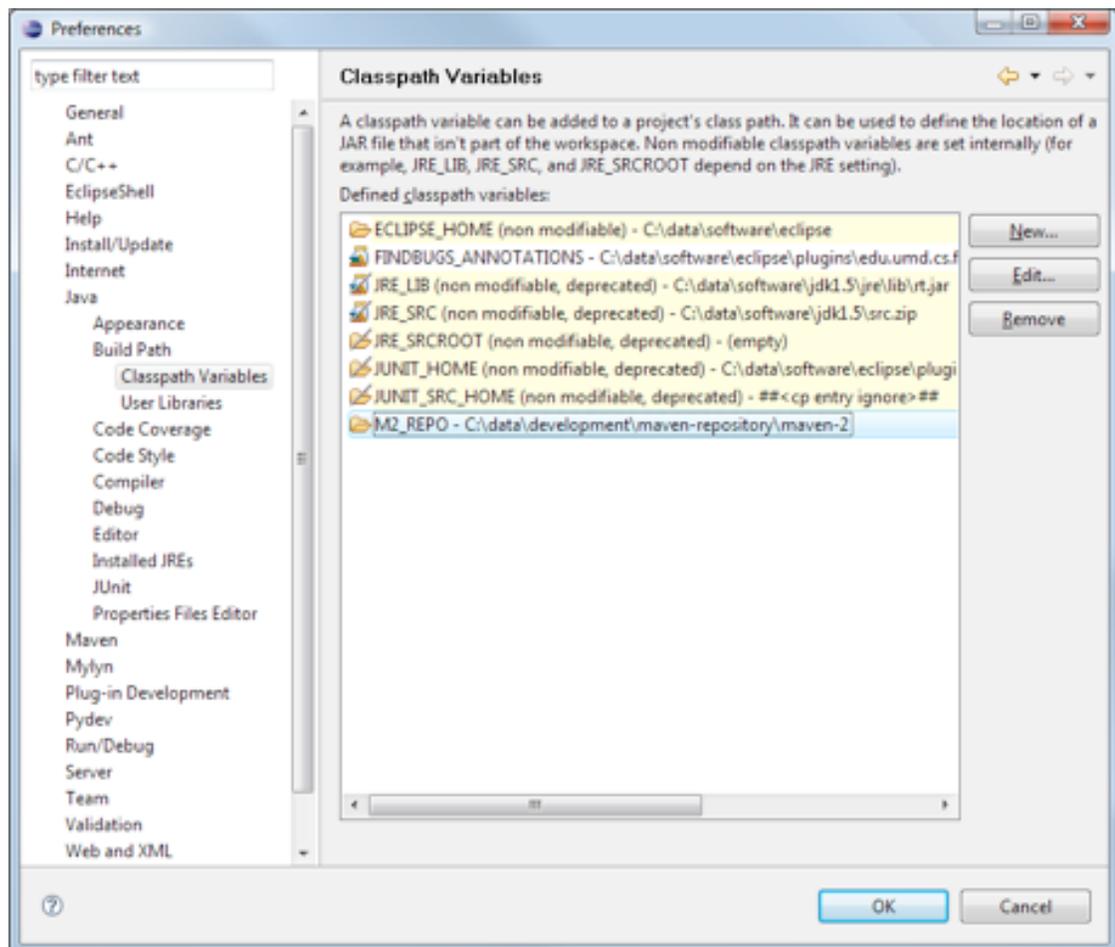
The Rice project contains auto-generated build path entries that rely on the presence of this M2_REPO variable to determine the location of dependency libraries.

Rebuild Rice

1. If dependency libraries have been added or removed from the Rice project, including the first time you check out Rice, you should run the retrieve-maven-dependencies Ant target to pull down all necessary libraries.

   Note: For the Maven2 Ant tasks to work, Ant has to know where your Maven2 home is. If you have set the **M2_HOME** variable in your system environment, it will be recognized automatically. If not, or if for some reason you want to use a different location (e.g., if you want to have multiple Maven installations), then you can set the **maven.home.directory** property in **/root/kuali-build.properties**.

2. Add the **build.xml** file in the root of the Rice project to your Ant view, or open a shell to the Rice project directory and run the retrieve-maven-dependencies target. You should see Maven retrieving any required dependencies. If you are running this Ant task in Eclipse, then you must recognize the **PATH** environment variable under *Run > External Tools > Open External Tools Dialog > Environment*.

3. Optionally, if you have trouble running this Ant target, you can just run an **mvn compile** from the command line to invoke a Maven compilation. This will download all dependencies into your local maven repository.

4. Execute a clean build of the project in Eclipse:

Figure 1.8. Eclipse Clean Build
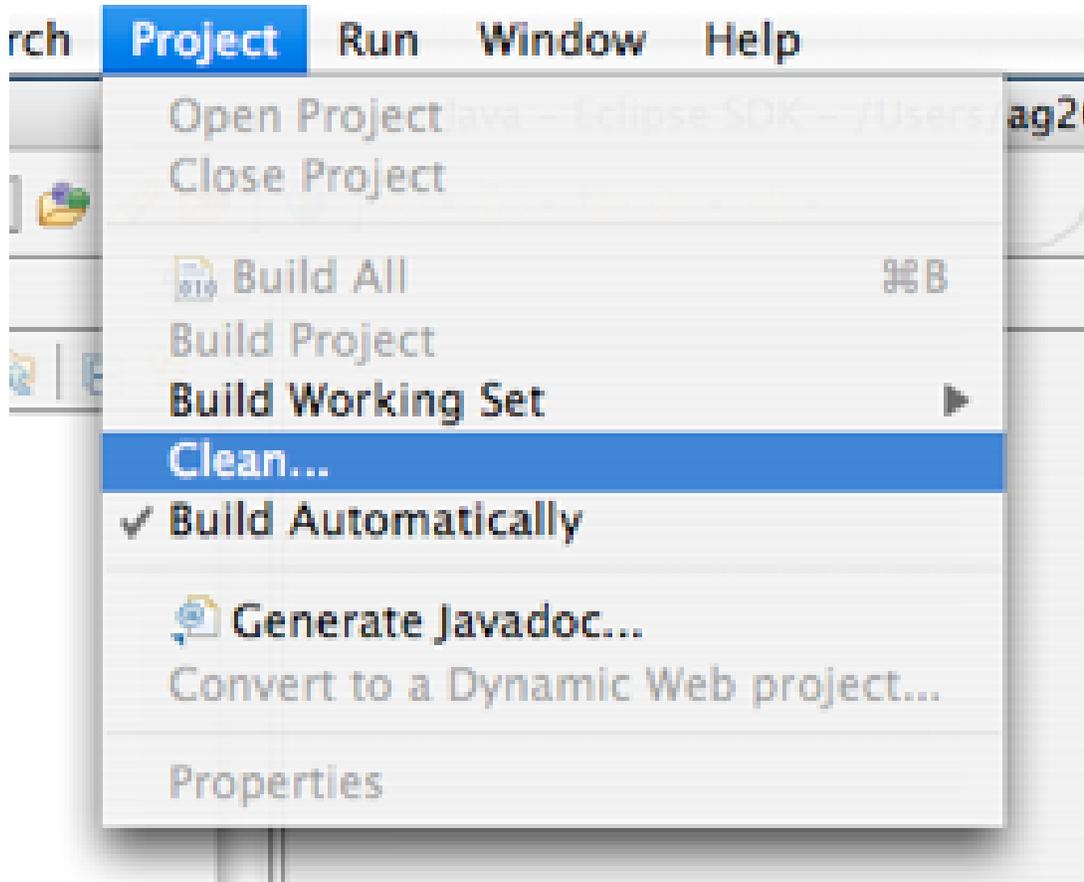
5. If your build was previously broken due to the **M2_REPO** classpath variable being undefined or due to missing libraries, it should now have been built successfully.

Install the database

To set up the database, please follow the instructions in the Installation Guide under Preparing the Database.

Installing the appropriate configuration files

Note: Be sure to use an appropriate editor such as vi or Notepad when editing configuration files. For example, we have found that WordPad can corrupt the configuration file.

To install the configuration file for the Kuali Rice sample application, you can do an Ant-based setup or a manual setup.

Ant-based setup

1. Execute the **prepare-dev-environment** Ant target in the **build.xml** file located in the root of the project.

2. This creates: **<user home>/kuali/main/dev/sample-app-config.xml**

Manual setup

1. Copy the **config/templates/sample-app-config.template.xml** file to **<user home>/kuali/main/dev/ sample-app-config.xml**.

   - For Windows, your user home is: **C:\Documents and Settings\<user name>**

   - For Unix/Linux, your user home is: **/home/<user name>**

   - For Mac OS X, your user home is: **/Users/<user name>**

2. Add the appropriate database parameters to **<user home>/kuali/main/dev/sample-app-config.xml**

   - Oracle

```
<param name="datasource.url">jdbc:oracle:thin:@localhost:1521:XE</param>
<param name="datasource.username">oracle.username</param>
<param name="datasource.password">oracle.password</param>
```

   - MySQL

```
<param name="datasource.url">jdbc:mysql://localhost:3306/kulrice</param>
<param name="datasource.username">mysql.username</param>
<param name="datasource.password">mysql.password</param>
```

Run the sample web application

   - Back in Eclipse, locate and run the rice - JettyServer.launch file:

Figure 1.9. Eclipse Jetty Launch
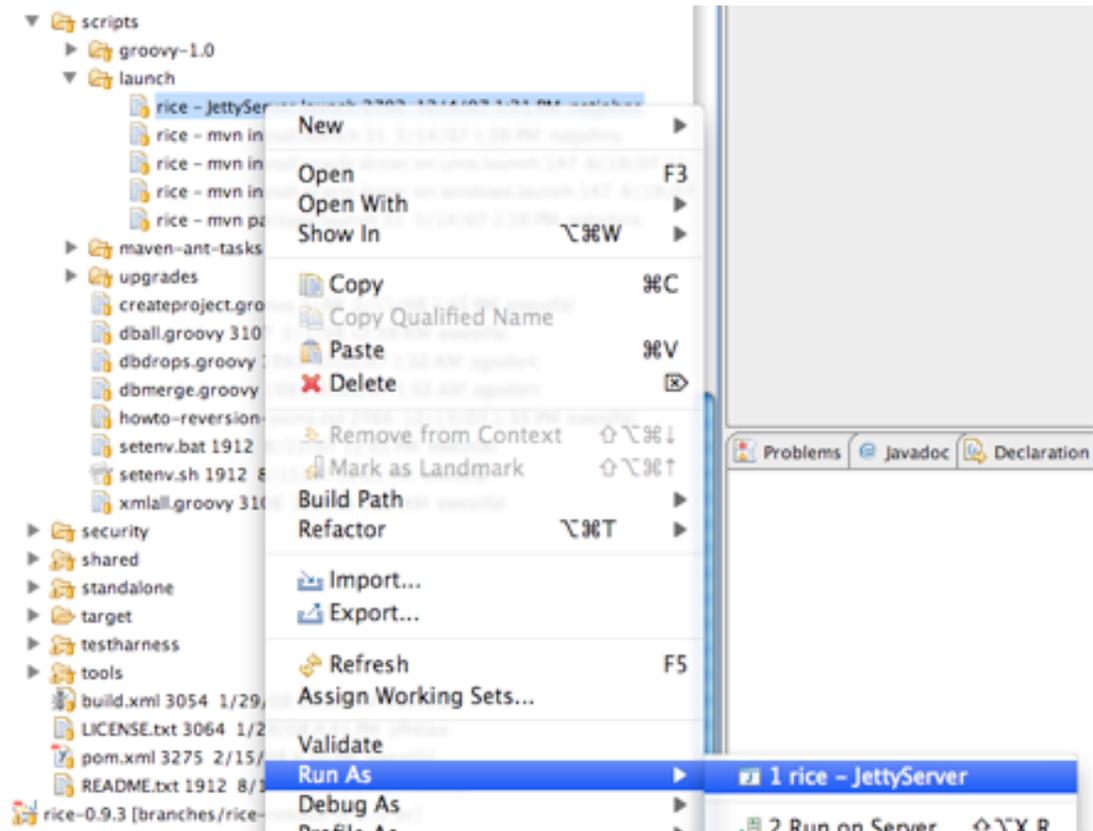
- Point your browser to the following url: http://localhost:8080/kr-dev

Changing Rice project dependencies

If you change any of the dependencies in any of the Rice **pom.xml** files, you must run the **update-eclipse-classpath** Ant target to regenerate the top-level Eclipse **.classpath** file for the project.

Figure 1.10. Update Eclipse Classpath
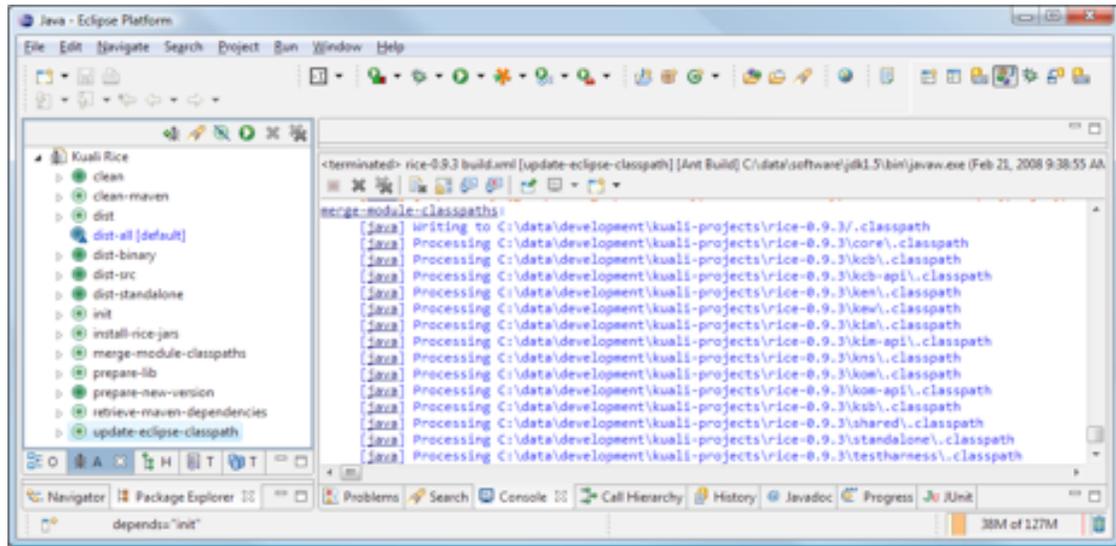
If you change the dependencies and commit the change, when others update their local source copy they must run the corresponding retrieve-maven-dependencies target again.

Note

Refresh your Eclipse project if dependencies (and therefore the Eclipse.classpath file) have changed.

Other Notes

Settings.xml warning

If this is the first time that you have installed the Maven plugin into your Eclipse environment, you may need to add a file called **settings.xml** in your **<user home>/.m2** directory.

The easiest way to tell if you need to do this is that there will be a warning in the console after building, stating that the settings.xml file is missing. All you need to do is create a settings.xml file with this content:

```
<settings/>
```

Rebuild, and the warning should no longer appear.

Note

You do **NOT** ever need to run any of the context menu Maven commands from inside Eclipse.

You do **NOT** need to run any Maven commands from the command line.

The Eclipse Maven2 plugin is a little bit flaky sometimes. You might need to close Eclipse to flush its memory.

Default workspace JDK not 1.5

If your default workspace JDK is not 1.5, you need to reconfigure the Maven external tools definitions for Rice this way:

1. Open *Run->External Tools->External Tools Dialog...* menu item.

2. Find the m2 build category.

3. Select each preconfigured Rice external tool configuration, select the JRE tab, and ensure the JRE is set to 1.5.

Using a custom maven repository location

The default Maven2 repository location is in your user directory; however, if you have a pre-existing repository (or for some other reason don't want it in your user directory), you can alter Maven2's repository location. The current version of the Maven2 plugin has a bug that does not allow this to work (see http://jira.codehaus.org/browse/MNGECLIPSE-314), but the 0.0.11 development version available from the update site http://m2eclipse.codehaus.org/update-dev/ allows you to specify a custom local repository.

Note

If you make this change, you may have to delete and re-add the Maven Managed Dependencies library to your project build path if you have an existing, invalid, Maven-managed dependencies library.

Setting JDK Compliance version

If your default workspace JDK is not 1.5, then you also need to set the JDK compliance level to the appropriate version for the project. You can find this by right-clicking on the *Project -> Properties -> Java Compiler -> Compiler* compliance level. Be sure the **Enable project specific settings** checkbox is checked.

Turn off validation

Be sure to turn off validation at the project level by right-clicking on the Project, then clicking *Properties -> Validation -> Suspend all Validators*. This can be adjusted once a successful Rice project is up and running.

ORA-12519, TNS:no appropriate service handler found

If you start seeing **java.sql.SQLException: Listener refused the connection with the following error: ORA-12519, TNS:no appropriate service handler found**, there are a couple of things that may remedy the problem.

1. Increase the Oracle XE connection limit:

```
alter system set processes=150 scope=spfile;
alter system set sessions=150 scope=spfile;
```

2. Lower the pool size in your rice config.xml:

```
<param name="datasource.pool.maxSize">10</param>
```

Disconnect any other clients and then restart Oracle-XE.

Creating Rice Enabled Applications

Creating a Rice Client Application Project Skeleton

In order to install a Rice client as a standalone server, please see the [installation guide instructions for Standalone Server Setup](#) section in the Installation Guide.

Reorder Eclipse Classpath

Once you have completed the installation, you will need to import your project into eclipse and reorder the eclipse classpath to account for a change in how the classpath was generated by maven. Navigate to your project properties and select the Order and Export tab from the Java Build Path project property. There will be an entry for JRE System Library at the bottom of the list that should be moved to the very top.

Rice Configuration System

The Rice Configuration System is an XML-based solution which provides capabilities similar to Java property files, but also adds some additional features. The configuration system lets you:

- Configure keys and values

- Aggregate multiple files using a single master file

- Build parameter values from other parameter values

- Use the parameters in Spring

- Override configuration values

Configuring Keys and Values

Below is an example of a configuration XML file. Note that the white space (spaces, tabs, and new lines) is stripped from the beginning and end of the values.

```
<config>
    <param name="client1.location">/var/lib/jenkins/workspace/rice-2.1.7-site-deploy/src/test/clients/
TestClient1</param>
    <param name="client2.location">/var/lib/jenkins/workspace/rice-2.1.7-site-deploy/src/test/clients/
TestClient2</param>
    <param name="ksb.client1.port">9913</param>
    <param name="ksb.client2.port">9914</param>
    <param name="ksb.testharness.port">9915</param>
    <param name="threadPool.size">1</param>
    <param name="threadPool.fetchFrequency">3000</param>
    <param name="bus.refresh.rate">3000</param>
    <param name="keystore.alias">rice</param>
    <param name="keystore.password">super-secret-pw</param>
    <param name="keystore.file">/var/lib/jenkins/workspace/rice-2.1.7-site-deploy/src/test/resources/keystore</
param>
</config>
```

Here is an example of the Java code required to parse the configuration XML file and convert it into a Properties object:

```
Config config = new SimpleConfig(configLocations, properties);
```

```
config.parseConfig();
```

In the sample above, configLocations is a List<String> containing file locations using the standard Spring naming formats (examples: **file:/whatever** and **classpath:/whatever**). The variable properties is a Properties object containing the default property values.

Here is an example of retrieving a property value from Java code:

```
String val = ConfigContext.getCurrentContextConfig().getProperty("keystore.alias");
```

Aggregating Multiple Files

The Rice Configuration System has a special parameter, config.location, which you use to incorporate the contents of another file. Typically, you use this to include parameters that are maintained by system administrators in secure locations. The parameters in the included file are parsed as if they had been in the original file at that place. Here is an example:

```
<config>
    <param name="config.location">file:/my_secure_dir/my_secure_file.xml</param>
</config>
```

Building Parameter Values from Other Parameters

Once you have defined a parameter, you can use it in the definition of another parameter. For example:

```
<config>
    <param name="apple">red delicious</param>
    <param name="taste">yummy yummy</param>
    <param name="apple.taste">${apple} ${taste}</param>
</config>
```

When this example is parsed, the value of the parameter **apple.taste** will be set to **red delicious yummy yummy**.

Using the Parameters in Spring

Because the parameters are converted into a Properties object, you can retrieve the complete list of parameters using this code:

```
config.getProperties()
```

You typically use this in Spring to parse a configuration and put its properties in a PropertyPlaceHolderConfigurer so that the parameters are available in the Spring configuration file:

```
<bean id="config" class="org.kuali.rice.core.config.spring.ConfigFactoryBean">
    <property name="configLocations">
        <list>
            <value>classpath:my-config.xml</value>
        </list>
    </property>
</bean>

<bean id="configProperties"
 class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
    <property name="targetObject" ref="config" />
```

```
        <property name="targetMethod" value="getProperties" />
</bean>

<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="properties" ref="configProperties" />
</bean>
```

Once this is complete, the configuration parameters can be used like standard Spring tokens in the bean configurations:

```
<bean id="dataSource" class="org.kuali.rice.core.database.XAPoolDataSource">
        <property name="transactionManager" ref="jotm" />
        <property name="driverClassName" value="${datasource.driver.name}" />
        <property name="url" value="${datasource.url}" />
        <property name="maxSize" value="${datasource.pool.maxSize}" />
        <property name="minSize" value="${datasource.pool.minSize}" />
        <property name="maxWait" value="${datasource.pool.maxWait}" />
        <property name="validationQuery" value="${datasource.pool.validationQuery}" />
        <property name="username" value="${datasource.username}" />
        <property name="password" value="${datasource.password}" />
</bean>
```

Initializing the Configuration Context in Rice

The Config object can be injected into the RiceConfigurer that's configured in Spring and it will initialize the configuration context with those configuration parameters.

This is done as follows:

```
<bean id="config" class="org.kuali.rice.core.config.spring.ConfigFactoryBean">
        ...
</bean>

<bean id="rice" class="org.kuali.rice.core.config.RiceConfigurer">
        <property name="rootConfig" ref="config"/>
</bean>
```

Overriding Configuration Values

The primary purpose of overriding configuration values is to provide a set of default values in a base configuration file and then provide a separate file that overrides the values that need to be changed. You can also update a parameter value multiple times in the same file. Parameter values can be changed any number of times; the last value encountered while parsing the file will be the value that is retained.

For example, when parsing the file:

```
<config>
        <param name="taste">yummy yummy</param>
        <param name="taste">good stuff</param>
</config>
```

The final value of the parameter **taste** will be **good stuff** since that was the last value listed in the file.

As another example, when parsing the file:

```
<config>
        <param name="taste">yummy yummy</param>
        <param name="apple.taste">apple ${taste}</param>
        <param name="taste">good stuff</param>
```

```
</config>
```

The final value of the parameter **apple.taste** will be **apple yummy yummy**. This demonstrates that parameters that appear in the value are replaced by the current value of the parameter at that point in the configuration file.

Additionally, you can define certain parameters in such that they won't override an existing parameter value if it's already set.

As an example of this, consider the following configuration file:

```
<config>
    <param name="taste" override="false">even yummier</param>
    <param name="brand.new.param" override="false">brand new value</param>

</config>
```

If this file was loaded into a configuration context that had already parsed our previous example, then it would notice that the **taste** parameter has already been set. Since **override** is set to false, it would not override that value with **even yummier**. However, since **brand.new.param** had not been defined previously, it's value would be set.

Data Source and JTA Configuration

The Kuali Rice software require a Java Transaction API (JTA) environment in which to execute database transactions. This allows for creation and coordination of transactions that span multiple data sources. This feature is something that would typically be found in a J2EE application container. However, Kuali Rice is designed in such a way that it should not require a full J2EE container. Therefore, when not running the client or web application inside of an application server that provides a JTA implementation, you must provide one. The default JTA environment that Kuali Rice uses is JOTM. There are other open-source options available, such as Atomikos TransactionsEssentials, and there are also commercial and open source JTA implementations that come as part of an application server (i.e. JBoss, WebSphere, GlassFish). Alternatively, Kuali Rice can be configured to use Bitronix.

If installing Rice using the standalone server option and a full Java application server is not being utilized, then the libraries required for JTA will need to be moved to the servlet server which is being used. These libraries have already been retrieved by Maven during project set up; it is a simple matter of moving them from the Maven repository to the libraries directory of the servlet server. Assuming, for instance, that Tomcat is being used, the following files need to be copied from the Maven repository to **$TOMCAT_HOME/common/lib**:

- **{Maven repository home}/repository/javax/transaction/jta/1.0.1B/jta-1.0.1B.jar**

- **{Maven repository home}/repository/jotm/jotm/2.0.10/jotm-2.0.10.jar**

- **{Maven repository home}/repository/jotm/jotm_jrmp_stubs/2.0.10/jotm_jrmp_stubs-1.0.10.jar**

- **{Maven repository home}/repository/xapool/xapool/1.5.0-patch3/xapool-1.5.0-patch3.jar**

- **{Maven repository home}/repository/howl/howl-logger/0.1.11/howl-logger-0.1.11.jar**

- **{Maven repository home}/repository/javax/resource/connector-api/1.5/connector-api-1.5.jar**

- **{Maven repository home}/repository/javax/resource/connector/1.0/connector-1.0.jar**

- **{Maven repository home}/repository/org/objectweb/carol/carol/2.0.5/carol-2.0.5.jar**

Additionally, the **{Rice project home}config/jotm/carol.properties** configuration file needs to be moved to $TOMCAT_HOME/common/classes, this time from the built Rice project.

Configuring JOTM

Configure the JOTM transaction manager and user transaction objects as Spring beans in your application's Spring configuration file. Here is an example:

```
<bean id="transactionManagerXAPool" class="org.springframework.transaction.jta.JotmFactoryBean">
    <property name="defaultTimeout" value="3600"/>
</bean>

<alias name="transactionManagerXAPool" alias="jtaTransactionManager"/>
<alias name="transactionManagerXAPool" alias="jtaUserTransaction"/>
```

You can use these beans in the configuration of Spring's JTA transaction manager and the Rice configurer. This configuration might look like the following:

```
<bean id="springTransactionManager" class="org.springframework.transaction.jta.JtaTransactionManager">
    <property name="userTransaction">
        <ref local="userTransaction" />
    </property>
    <property name="transactionManager">
        <ref local="jtaTransactionManager" />
    </property>
</bean>

<bean id="rice" class="org.kuali.rice.core.config.RiceConfigurer">
    <property name="transactionManager" ref="jtaTransactionManager" />
    <property name="userTransaction" ref="jtaUserTransaction" />
    ...
</bean>
```

Configuring JOTM Transactional Data Sources

JTA requires that the datasources that are used implement the XADataSource interface. Some database vendors, such as Oracle, have pure XA implementations of their datasources. However, internally to Rice, we use wrappers on plain datasources using a library called XAPool. When configuring transactional data sources that will be used within JOTM transactions, you should use the org.kuali.rice.core.database.XAPoolDataSource class provided with Rice. Here is an example of a Spring configuration using this data source implementation:

```
<bean id="myDataSource" class="org.kuali.rice.core.database.XAPoolDataSource">
    <property name="transactionManager" ref="jtaTransactionManager" />
    <property name="driverClassName" value="${datasource.driver.name}" />
    <property name="url" value="${datasource.url}" />
    <property name="maxSize" value="${datasource.pool.maxSize}" />
    <property name="minSize" value="${datasource.pool.minSize}" />
    <property name="maxWait" value="${datasource.pool.maxWait}" />
    <property name="validationQuery" value="${datasource.pool.validationQuery}" />
    <property name="username" value="${datasource.username}" />
    <property name="password" value="${datasource.password}" />
</bean>
```

Configuring JTOM Non-Transactional Data Sources

When using the built-in instance of the Quartz scheduler that Rice creates, you will need to inject a non-transactional data source into the RiceConfigurer in addition to the JTA transactional instance. This is to prevent deadlocks in the database and is required by the Quartz software (the Quartz web site has

an [FAQ entry](#) with more details on the problem). Here is an example of a non-transactional data source configuration:

```
<bean id="nonTransactionalDataSource"
 class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName" value="${datasource.driver.name}"/>
    <property name="url" value="${datasource.url}"/>
    <property name="maxActive" value="${datasource.pool.maxActive}"/>
    <property name="minIdle" value="7"/>
    <property name="initialSize" value="7"/>
    <property name="validationQuery" value="${datasource.pool.validationQuery}"/>
    <property name="username" value="${datasource.username}" />
    <property name="password" value="${datasource.password}" />
    <property name="accessToUnderlyingConnectionAllowed"
 value="${datasource.dbcp.accessToUnderlyingConnectionAllowed}"/>
</bean>
```

You need to either inject this non-transactional data source into the Quartz SchedulerFactory Spring bean (if you are explicitly defining it) or into the rice bean in the Spring Beans config file as follows:

```
<bean id="rice" class="org.kuali.rice.config.RiceConfigurer">
    ...
    <property name="nonTransactionalDataSource" ref="nonTransactionalDataSource" />
    ...
</bean>
```

Configuring Bitronix

Configure the [Bitronix](#) transaction manager and user transaction objects as Spring beans in your application's Spring configuration file. Here is an example:

```
<bean id="btmConfig" factory-method="getConfiguration"
        class="bitronix.tm.TransactionManagerServices" lazy-init="true"/>
<bean id="transactionManagerBitronix" class="bitronix.tm.TransactionManagerServices"
        factory-method="getTransactionManager" depends-on="btmConfig" destroy-method="shutdown" lazy-
init="true"/>

<alias name="transactionManagerBitronix" alias="jtaTransactionManager"/>
<alias name="transactionManagerBitronix" alias="jtaUserTransaction"/>
```

You can use these beans in the configuration of the Rice configurer. This configuration might look like the following:

```
<bean id="rice" class="org.kuali.rice.core.config.RiceConfigurer">
    <property name="transactionManager" ref="jtaTransactionManager" />
    <property name="userTransaction" ref="jtaUserTransaction" />
  ...
</bean>
```

Configuring Bitronix Transactional Data Sources

An example configuration of Btironix Transactional Data Sources:

```
<bean id="riceDataSourceBitronixXa" class="bitronix.tm.resource.jdbc.PoolingDataSource" init-method="init"
        destroy-method="close" lazy-init="true">
    <property name="className" value="oracle.jdbc.xa.client.OracleXADataSource" />
    <property name="uniqueName" ref="ds-random-string" />
    <property name="minPoolSize" value="${datasource.pool.minSize}" />
    <property name="maxPoolSize" value="${datasource.pool.maxSize}" />
    <property name="useTmJoin" value="true" />
```

```
        <property name="testQuery" value="${datasource.pool.validationQuery}" />
        <property name="allowLocalTransactions" value="true" />
        <property name="driverProperties">
          <props>
            <prop key="URL">${datasource.url}</prop>
            <prop key="user">${datasource.username}</prop>
            <prop key="password">${datasource.password}</prop>
          </props>
        </property>
</bean>

<bean id="ds-random-string" class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
        <property name="staticMethod" value="org.apache.commons.lang.RandomStringUtils.randomAlphanumeric"/>
        <property name="arguments"><list><value>20</value></list></property>
</bean>
```

Configuring Bitronix Non-Transactional Data Sources

Notice the addition of the driverClassName prop in the dirverProperties in the non-transaction data source configuration

```
<bean id="riceDataSourceBitronix" class="bitronix.tm.resource.jdbc.PoolingDataSource" init-method="init"
        destroy-method="close" lazy-init="true">
    <property name="className" value="bitronix.tm.resource.jdbc.lrc.LrcXADataSource" />
    <property name="uniqueName" ref="ds-random-string" />
    <property name="minPoolSize" value="${datasource.pool.minSize}" />
    <property name="maxPoolSize" value="${datasource.pool.maxSize}" />
    <property name="useTmJoin" value="true" />
    <property name="testQuery" value="${datasource.pool.validationQuery}" />
    <property name="allowLocalTransactions" value="true" />
    <property name="driverProperties">
      <props>
        <prop key="Url">${datasource.url}</prop>
        <prop key="driverClassName">${datasource.driver.name}</prop>
        <prop key="user">${datasource.username}</prop>
        <prop key="password">${datasource.password}</prop>
      </props>
    </property>
</bean>

<bean id="ds-random-string" class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
    <property name="staticMethod" value="org.apache.commons.lang.RandomStringUtils.randomAlphanumeric"/>
    <property name="arguments"><list><value>20</value></list></property>
</bean>
```

Version Compatibility

Commitment to Compatibility in Kuali Rice

From version 2.0 of Kuali Rice up to at least version 3.0, the project is committed to providing what it refers to as "middleware" or "client-server" service compatibility. This essentially means that an application which is a client of the Kuali Rice Standalone Server (either it's services or it's database) should be able to continue to function properly even if the Rice Standalone Server or it's database is upgraded to a newer version.

More information on the scope of version compatibility in Kuali Rice can be found in the Kuali Rice Version Compatibility Statement.

Keeping Your Client Application Compatible

There are a few rules that a client application using the Kuali Rice apis must following in order to ensure the client application remains compatible once the Kuali Rice Standalone Server is updated. These rules

only apply in situations where there is a standalone instance of Kuali Rice which is being integrated with. In the case that an application is running Kuali Rice "bundled", then compatibility is not a concern since that application forms a single software bundle with Kuali Rice included.

First, client-server compatibility only pertains to the components of Rice which have client-server interaction, that includes the following components and their sub-modules:

- Core Service

- KSB

- KEW

- KIM

- KEN

- KRMS

- Location

There are some components of Rice which are "framework-only" and don't contain any client-server remoting components. These include:

- Core

- KNS

- KRAD

There are, additionally, some modules of Rice which only run as part of the standalone server (or in bundled mode) and those include eDocLite and the various "web" modules of Rice.

Only the first set of "client-server" Rice components are presently under the constraints of version compatibility.

A summary of the rules a client application needs to follow in order to ensure they remain version compatible is as follows:

- If integrated with a standlone Kuali Rice server, do not configure any of the Kuali Rice components with a run mode of *LOCAL*. The *LOCAL* run mode is only for a fully bundled configuration of Kuali Rice as it interacts directly with all of the Kuali Rice database tables instead of using remotely accessible services.

- In application code, only use classes in the api or framework modules of the "client-server" components. This should be evident from the package names for the module as they should have "api" or "framework" in the package name (i.e. `org.kuali.rice.kew.api.*` and `org.kuali.rice.kim.framework.*`).

- Do not write custom code which interacts directly with the Kuali Rice database tables which are part of any of the previously mentioned "client-server" components.

- When writing code against the Rice apis or frameworks, be sure to read the javadocs and be sure to conform to the contracts specified therein.

- When implementing "callback" services, ensure that you are using the `CallbackServiceExporter` properly as specified in [the section called "CallbackServiceExporter"](#).

Chapter 2. KEN

KEN Overview

What is KEN?

Kuali Enterprise Notification is a form of communication between distributed systems that allows messages to be sent securely and consistently. These messages act as notifications upon receipt and are processed asynchronously within the service layer. The following architectural diagram represents the flow of messages in a typical Rice Environment.

Figure 2.1. KEN Message Flow
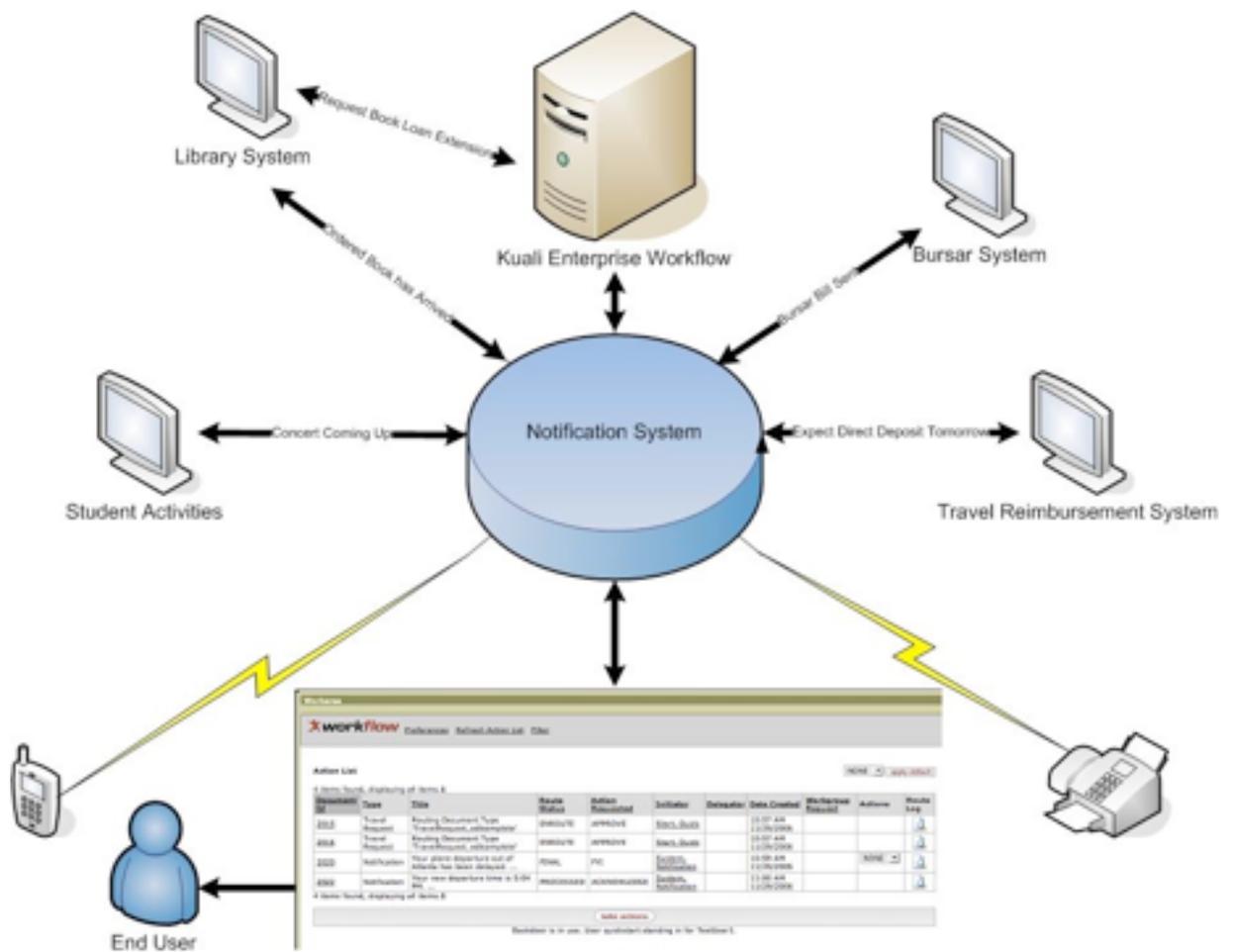
From a developer's perspective the diagram below helps to represent the inner workings of how KEN stores data from the Data Modeling Layer into the Persistence Layer.

Figure 2.2. KEN Message Storage
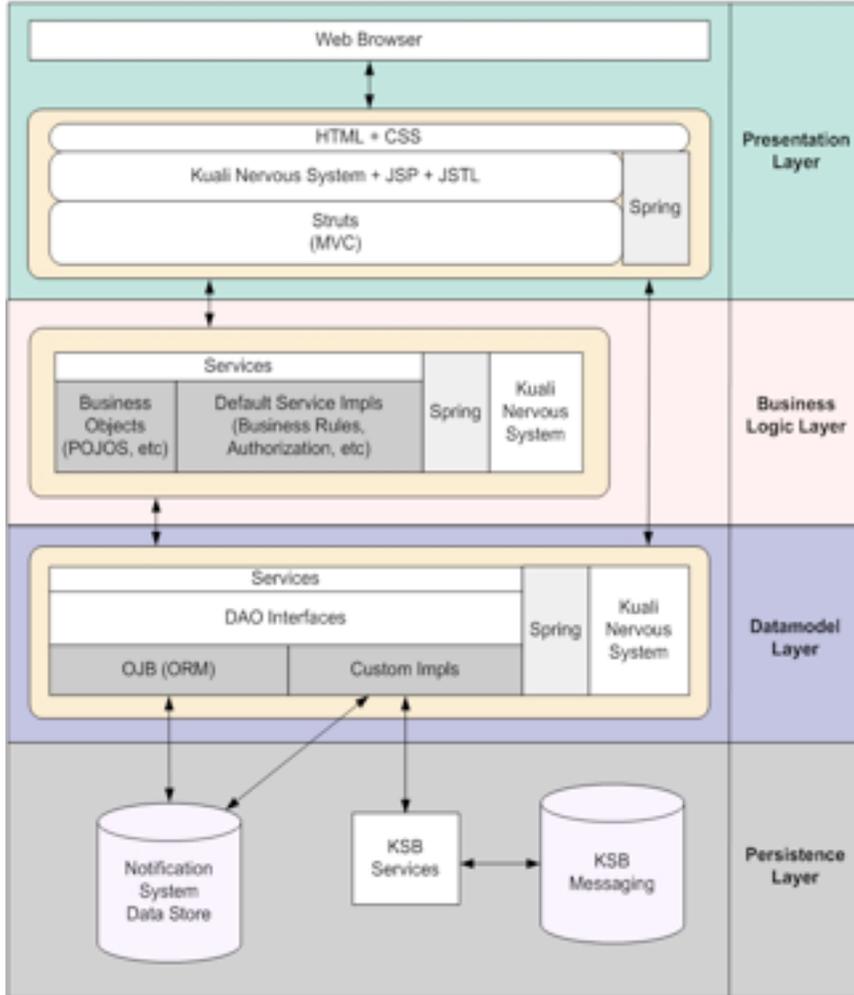
The following sections of documentation aim at describing the inner workings of KEN as well as how those pieces interact with Rice, specifically KEW. KEN itself is an interface that sits on top of KEW's API. This allows for registration and publishing of notifications, which then flow through KEW to result in a KEW action request. See KEW Overview for more information. In addition to the action list, KEW can be optionally configured to forward these requests to the Kuali Communications Broker or KCB for short. This module is logically related to KEN and handles dispatching messages based on the user preferences. Once messages are dispatched, a response or acknowledgement can be created.

KEN Configuration Parameters

Table 2.1. KEN Core Parameters

Configuration Parameter	Description	Default value
ken.url	The base URL of the KEN webapp; this should be changed when deploying for external access	${application.url}/ken
notification.resolveMessageDeliveriesJob.startDelayMS	The start delay (in ms) of the job that resolves message deliveries	5000
notification.resolveMessageDeliveriesJob.intervalMS	The interval (in ms) between runs of the message delivery resolution job	10000

Configuration Parameter	Description	Default value
notification.processAutoRemovalJob.startDelayMS	The start delay (in ms) of the job that auto-removes messages	60000
notification.processAutoRemovalJob.intervalMS	The interval (in ms) between runs of the message auto-removal job	60000
notification.quartz.autostartup	Whether to automatically start the KEN Quartz jobs	true
notification.concurrent.jobs	Whether the invocation of a KEN Quartz job can overlap another KEN Quartz job running concurrently	true
ken.system.user	The principal name of the user that KEN should use when initiating KEN-originated documents	notsys
kcb.url	The base URL of the KCB (notification broker) webapp	${application.url}/kcb
kcb.messaging.synchronous	Whether notification messages are processed synchronously	false
kcb.messageprocessing.startDelayMS	The start delay (in ms) of the job that processes notification messages	50000
kcb.messageprocessing.repeatIntervalMS	The interval (in ms) between runs of the notification message processing job	30000
kcb.quartz.group	Group name of the KCB Quartz job	KCB-Delivery
kcb.quartz.job.name	Name of the KCB Quartz job	MessageProcessingJobDetail
kcb.maxProcessAttempts	Maximum number of times that KCB will attempt to process a notification message	3
notification.processUndeliveredJob.intervalMS	The elapsed time, in milliseconds, between runs of the KEN process undelivered notifications job.	10000
notification.processUndeliveredJob.startDelayMS	The elapsed time, in milliseconds, between the start of the application and the first run of the KEN process undelivered notifications job.	10000

Note

As of Rice 1.0.1, The parameter **kcb.smtp.host** is no longer used. The smtp server settings that are required for sending email notifications with KEN are documented in the Kuali Enterprise Workflow (KEW) Technical Reference Guide under **Email Configuration.**

KEN Channels

A KEN Channel is correlated to a specific type of notification. An example of a Channel's use may be to send out information about upcoming Library Events or broadcast general announcements on upcoming concerts. Channels are subscribed to in the act of receiving notifications from a publisher or producer. They can also be unsubscribed to and removed from the data store from within the UI. The Channel Definitions are stored in the database table KREN_CHNL_T. The columns are listed as follows:

Table 2.2. KREN_CHNL_T

Column	Description
CHNL_ID	Identifier for the Channel
NM	Name of the Channel represented in the UI
DESC_TXT	Description of the Channel
SUBSCRB_IND	Determines if the Channel can or cannot be subscribed to from the UI. This also determines if the channel will be displayed in the UI
VER_NBR	Version Number for the Channel

Channel Subscription

Channels can be subscribed to through the UI and also through the direct access to the data store. To add a channel that can be subscribed to simply run the following SQL statement against the data store customizing value entries to your needs:

```
INSERT INTO KREN_CHNL_T (CHNL_ID,DESC_TXT,NM,SUBSCRB_IND,VER_NBR)
  VALUES (2,'This channel is used for sending out information about Library Events.','Library Events
 Channel','Y',
1)
```

KEN Producers

A KEN Producer submits notifications for processing through the system. An example of a Producer would be a mailing daemon that represents messages sent from a University Library System.

Characteristics of a Producer:

• Producers create and send notifications to a specific destination through various Channels.

• Each Producer contains a list of Channels that it may send notifications to.

• Producer Definitions are stored in the database table KREN_PRODCR_T.

Table 2.3. KREN_PRODCR_T

Column	Description
CNTCT_INFO	The email address identifying the Producer of the Notification.
DESC_TXT	A Description of the Producer.
NM	Name of the Producer.
PRODCR_ID	The Producer's Channel Identifier. See the KREN_CHNL_PRODCR_T table found in the database for more information on how Producers link to Channels.
VER_NBR	Version Number for the Producer.

Adding Producers

The Producer can be added through direct access to the data store. To add a Producer run the following SQL statement against the data store customizing value entries to your needs:

```
INSERT INTO KREN_PRODCR_T (CNTCT_INFO,DESC_TXT,NM,PRODCR_ID,VER_NBR)
  VALUES ('kuali-ken-testing@cornell.edu','This producer represents messages sent from the general message
 sending forms.','Notification System',1,1)
```

KEN Content Types

Overview

A Content Type is part of the message content of a notification that may be sent using KEN. It can be as simple as a single message string, or something more complex, such as an event that might have a date associated with it, start and stop times, and other metadata you may want to associate with the notification.

KEN is distributed with two Content Types: Simple and Event.

Warning

It is strongly recommended that you leave these two Content Types intact, but you can use them as templates for creating new Content Types.

Every notification sent through KEN must be associated with a **registered** Content Type. Registration of Content Types requires administrative access to the system and is described in the KEN Content Types section in the User Guide. The rest of this section describes the Content Type attributes that are required for registration.

Content Type Attributes

A Content Type is represented as a *NotificationContent* business object and consists of several attributes, described below:

id - Unique identifier that KEN automatically creates when you add a Content Type

name - This is a unique string that identifies the content. For example, *ItemOverdue* might be the *name* used for a notification Content Type about an item checked out from the campus library.

description - This is a more verbose description of the Content Type. For example, "Library item overdue notices" might be the *description* for *ItemOverdue*.

namespace - This is the string used in the XSD schema and XML to provide validation of the content, for example, *notification/ContentTypeItemOverdue*. The XSD namespace is typically the *name* attribute concatenated to the *notification/ContentType* string. Note how it is used in the **XSD** and **XSL** examples below.

xsd - The XSD attribute contains the complete W3C XML Schema compliant code.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- This schema defines a generic event notification type in order for it to be accepted into the system. -->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:c="ns:notification/common"
        xmlns:ce="ns:notification/ContentTypeItemOverdue"
        targetNamespace="ns:notification/ContentTypeItemOverdue"
        attributeFormDefault="unqualified"
        elementFormDefault="qualified">
    <annotation>
        <documentation xml:lang="en">Item Overdue Schema</documentation>
    </annotation>
    <import namespace="ns:notification/common" schemaLocation="resource:notification/notification-common" />

    <!-- The content element describes the content of the notification.  It contains a message (a simple
 String) and a message element -->
    <element name="content">
        <complexType>
            <sequence>
                <element name="message" type="c:LongStringType"/>
                <element ref="ce:event"/>
            </sequence>
        </complexType>
    </element>

    <!-- This is the itemoverdue element.  It describes an item overdue notice containing a summary,
 description, location, due date, and the amount of the fine levied -->
    <element name="itemoverdue">
        <complexType>
            <sequence>
                <element name="summary" type="c:NonEmptyShortStringType" />
                <element name="description" type="c:NonEmptyShortStringType" />
                <element name="location" type="c:NonEmptyShortStringType" />
                <element name="dueDate" type="dateTime" />
                <element name="fine" type="decimal" />
            </sequence>
```

```
            </complexType>
        </element>
</schema>
```

xsl - The XSD attribute contains the complete XSL code that will be used to transform a notification in XML to html for rendering in an Action List.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- style sheet declaration: be very careful editing the following, the
 default namespace must be used otherwise elements will not match -->
<xsl:stylesheet

    version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:n="ns:notification/ContentTypeEvent"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="ns:notification/ContentTypeItemOverdue resource:notification/ContentTypeItemOverdue"
    exclude-result-prefixes="n xsi">



    <!-- output an html fragment -->
    <xsl:output method="html" indent="yes" />


    <!-- match everything -->
    <xsl:template match="/n:content" >
        <table class="bord-all">
            <xsl:apply-templates />
        </table>
    </xsl:template>


    <!--  match message element in the default namespace and render as strong -->
    <xsl:template match="n:message" >
        <caption>
            <strong><xsl:value-of select="." disable-output-escaping="yes"/></strong>
        </caption>
    </xsl:template>


    <!-- match on itemoverdue in the default namespace and display all children -->
    <xsl:template match="n:itemoverdue">
        <tr>
            <td class="thnormal"><strong>Summary: </strong></td>
            <td class="thnormal"><xsl:value-of select="n:summary" /></td>
        </tr>
        <tr>
            <td class="thnormal"><strong>Item Description: </strong></td>
            <td class="thnormal"><xsl:value-of select="n:description" /></td>
        </tr>
        <tr>
            <td class="thnormal"><strong>Library: </strong></td>
            <td class="thnormal"><xsl:value-of select="n:location" /></td>
        </tr>
        <tr>
            <td class="thnormal"><strong>Due Date: </strong></td>
            <td class="thnormal"><xsl:value-of select="n:startDateTime" /></td>
        </tr>
        <tr>
            <td class="thnormal"><strong>Fine: </strong></td>
            <td class="thnormal">$<xsl:value-of select="n:fine" /></td>
        </tr>
    </xsl:template>
</xsl:stylesheet>
```

KEN Notifications

This document provides information about the attributes of a Notification. These attributes are elements such as message content, who is sending the notification, who should receive it, etc. Kuali Enterprise

Notification (KEN) supports an arbitrary number of Content Types, such as a simple message or an event notification. Each Content Type consists of a common set of attributes and a content attribute.

Common Notification Attributes

Table 2.4. Common Notification Attributes

Name	Type	Required	Description	Example
channel	string	yes	• Name of a channel • Must be registered	Library Events
producer	string	yes	• Name of the producing system • Must be registered and given authority to send messages on behalf of the *<Library Events>* channel	Library Calendar System
senders	a list of strings	yes	A list of the names of people on whose behalf the message is being sent	TestUser1, TestUser2
recipients	a list of strings	yes	A list of the names of groups or users to whom the message is being sent	library-staff-group, TestUser1, TestUser2
deliveryType	string	yes	fyi or ack	fyi
sendDateTime	datetime	no	When to send the notification	2006-01-01 00:00:00.0
autoRemoveDateTime	datetime	no	When to remove the notification	2006-01-02 00:00:00.0
priority	string	yes	An arbitrary priority; these must be registered in KEN; the system comes with defaults of *normal*, *low*, and *high*	normal
contentType	string	yes	Name for the content; KEN comes set up with *simple* and *event*; new contentTypes must be registered in KEN	simple
content	see below	yes	The actual content	see below

Message Content

Notifications are differentiated using the *contentType* attribute and the contents of the *content* element. The *content* element can be as simple as a message string or it may be a complex structure. For example, a simple notification may only contain a message string, whereas an *Event* Content Type might contain a summary, description, location, and start and end dates and times. Examples of the *Simple* and *Event* Content Types:

Sample XML for a Simple Notification

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- A Simple Notification Message -->
<notification xmlns="ns:notification/NotificationRequest"
    xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
    xsi:schemaLocation="ns:notification/NotificationRequest
    resource:notification/NotificationRequest">
    <!-- this is the name of the notification channel -->
    <!-- that has been registered in the system -->
    <channel>Campus Status Announcements</channel>

    <!-- this is the name of the producing system -->
    <!-- the value must match a registered producer -->
    <producer>Campus Announcements System</producer>

    <!-- these are the people that the message is sent on -->
    <!-- behalf of -->
```

```
    <senders>
        <sender>John Fereira</sender>
    </senders>


    <!-- who is the notification going to? -->
    <recipients>
        <group>Everyone</group>
        <user>jaf30</user>
    </recipients>


    <!--  fyi or acknowledge -->
    <deliveryType>fyi</deliveryType>


    <!-- optional date and time that a notification should be sent -->
    <!-- use this for scheduling a single future notification to happen -->
    <sendDateTime>2006-01-01T00:00:00</sendDateTime>


    <!-- optional date and time that a notification should be removed -->
    <!-- from all recipients' lists, b/c the message no longer applies -->
    <autoRemoveDateTime>3000-01-01T00:00:00</autoRemoveDateTime>


    <!-- this is the name of the priority of the message -->
    <!-- priorities are registered in the system, so your value -->
    <!-- here must match one of the registered priorities -->
    <priority>Normal</priority>


    <title>School is Closed</title>


    <!-- this is the name of the content type for the message -->
    <!-- content types are registered in the system, so your value -->
    <!-- here must match one of the registered contents -->
    <contentType>Simple</contentType>


    <!-- actual content of the message -->
    <content xmlns="ns:notification/ContentTypeSimple"
        xsi:schemaLocation="ns:notification/ContentTypeSimple
        resource:notification/ContentTypeSimple">

        <message>Snow Day! School is closed.</message>
    </content>
</notification>
```

Sample XML for an Event Notification

```
<?xml version="1.0" encoding="UTF-8"?>


<notification xmlns="ns:notification/NotificationMessage"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="ns:notification/NotificationMessage
    resource:notification/NotificationMessage">
    <!-- this is the name of the notification channel -->
    <!-- that has been registered in the system -->
    <channel>Concerts Coming to Campus</channel>


    <!-- this is the name of the producing system -->
    <!-- the value must match a registered producer -->
    <producer>Campus Events Office</producer>


    <!-- these are the people that the message is sent on -->
    <!-- behalf of -->
    <senders>
        <sender>ag266</sender>
```

```
        <sender>jaf30</sender>
    </senders>


    <!-- who is the notification going to? -->
        <recipients>
            <group>Group X</group>
            <group>Group Z</group>
            <user>ag266</user>
            <user>jaf30</user>
            <user>arh14</user>
        </recipients>


    <!--  fyi or acknowledge -->
    <deliveryType>fyi</deliveryType>


    <!-- optional date and time that a notification should be sent -->
    <!-- use this for scheduling a single future notification to happen -->
    <sendDateTime>2006-01-01 00:00:00.0</sendDateTime>


    <!-- optional date and time that a notification should be removed -->
    <!-- from all recipients' lists, b/c the message no longer applies -->
    <autoRemoveDateTime>2007-01-01 00:00:00.0</autoRemoveDateTime>


    <!-- this is the name of the priority of the message -->
    <!-- priorities are registered in the system, so your value -->
    <!-- here must match one of the registered priorities -->
    <priority>Normal</priority>


    <!-- this is the name of the content type for the message -->
    <!-- content types are registered in the system, so your value -->
    <!-- here must match one of the registered contents -->
    <contentType>Event</contentType>


    <!-- actual content of the message -->
    <content>
        <message>CCC presents The Strokes at Cornell</message>


        <!-- an event that it happening on campus -->
        <event xmlns="ns:notification/ContentEvent"
            xsi:schemaLocation="ns:notification/ContentEvent
            resource:notification/ContentEvent">
            <summary>CCC presents The Strokes at Cornell</summary>
            <description>blah blah blah</description>
            <location>Barton Hall</location>
            <startDateTime>2006-01-01T00:00:00</startDateTime>
            <stopDateTime>2007-01-01T00:00:00</stopDateTime>
        </event>
    </content>
</notification>
```

Notification Response

When KEN sends a notification, it always returns a response. This is an outline in XML of that response:

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
    <status>success</status>
</response>
```

Enterprise Notification Priority

Managing Priorities

There is no user interface page to manage priorities so you must make changes to the list of priorities in the **kren_prio_t** table using SQL.

The table has these columns:

Table 2.5. KREN_PRIO_T

Name	Type	Max Size	Required	Attribute
PRIO_ID	Numeric	8	Yes	ID
NM	Text	40	Yes	Name
DESC_TXT	Text	500	Yes	Description
PRIO_ORD	Numeric	4	Yes	Order
VER_NBR	Numeric	8	Yes	Version

Example 2.1. Example – This is an example of how to add a Priority into the table:

```
INSERT INTO kren_prio_t (PRIO_ID, NM, DESC_TXT, PRIO_ORD, VER_NBR) VALUES (8, 'Bulk', 'Mass notifications', 6,
 1);
```

KEN Delivery Types

This section describes Kuali Enterprise Notification (KEN) Delivery Types, or what are sometimes called Message Deliverers. A Message Deliverer Plugin is the mechanism used to deliver a notification to end users. All notifications sent through KEN appear in the Action List for each recipient for which the notification is intended. This message also contains an Email Delivery Type that allows you to send end users a notification summary as an email message. Note that for a Delivery Type other than the default (KEWActionList), the content of the notification is typically just a summary of the full notification.

Implementing the Java Interface

Creating a new Delivery Type primarily involves implementing a Java interface called **org.kuali.rice.kew.deliverer.NotificationMessageDeliverer**. The source code of the interface:

```
/*
 * Copyright 2007 The Kuali Foundation
 *
 * Licensed under the Educational Community License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.opensource.org/licenses/ecl2.php
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.

 */
package org.kuali.rice.ken.deliverer;
```

```
import java.util.HashMap;

import java.util.LinkedHashMap;


import org.kuali.rice.ken.bo.NotificationMessageDelivery;
import org.kuali.rice.ken.exception.ErrorList;

import org.kuali.rice.ken.exception.NotificationAutoRemoveException;
import org.kuali.rice.ken.exception.NotificationMessageDeliveryException;
import org.kuali.rice.ken.exception.NotificationMessageDismissalException;


/**
 * This class represents the different types of Notification Delivery Types that the system can handle.
 * For example, an instance of delivery type could be "ActionList" or "Email" or "SMS".  Any deliverer
 implementation
 * adhering to this interface can be plugged into the system and will be automatically available for use.
 * @author Kuali Rice Team (kuali-rice@googlegroups.com)
*/

public interface NotificationMessageDeliverer {
  /**
   * This method is responsible for delivering the passed in messageDelivery record.
   * @param messageDelivery The messageDelivery to process
   * @throws NotificationMessageDeliveryException
   */

 public void deliverMessage(NotificationMessageDelivery messageDelivery) throws
 NotificationMessageDeliveryException;
   /**
    * This method handles auto removing a message delivery from a person's list of notifications.
    * @param messageDelivery The messageDelivery to auto remove
    * @throws NotificationAutoRemoveException
    */

 public void autoRemoveMessageDelivery(NotificationMessageDelivery messageDelivery) throws
 NotificationAutoRemoveException;
   /**
    * This method dismisses/removes the NotificationMessageDelivery so that it is no longer being presented to
 the user
    * via this deliverer.  Note, whether this action is meaningful is dependent on the deliverer
 implementation.  If the
    * deliverer cannot control the presentation of the message, then this method need not do anything.
    * @param messageDelivery the messageDelivery to dismiss
    * @param the user that caused the dismissal; in the case of end-user actions, this will most likely be the
 user to
    *   which the message was delivered (user recipient in the NotificationMessageDelivery object)
    * @param cause the reason the message was dismissed
    */


 public void dismissMessageDelivery(NotificationMessageDelivery messageDelivery, String user, String cause)
 throws NotificationMessageDismissalException;
```

KEN: Sending a Notification

The Kuali Enterprise Notification system (KEN) provides for a way to programmatically send a notification. An application may construct a notification using the KEN web service API.

Send a Notification Using the Web Service API

To send a notification using the web service API, the notification must be constructed as an XML document that validates against a schema for a specific Content Type. For more detail, see the Notifications documentation.

To validate your notification XML, you must construct the XSD schema filename. To construct this file name, append the Content Type value to *ContentType*.

For example, if you create a new Content Type for a library book overdue notification, then the *contentType* element value should be *OverdueNotice* and the schema file you created for validation of the notification XML should be **ContentTypeOverdueNotice.xsd**. This XML schema should be declared as a namespace in the **content** element of the notification XML. Out of the box, KEN comes with *Simple* and *Event* Content Types.

Web Service URL

By default, the Notification Web Service API may be accessed at: http://yourlocalip:8080/remoting/soap/ken/v2_0/sendNotificationService

A WSDL may be obtained using the following URL: http://yourlocalip:8080/remoting/soap/ken/v2_0/sendNotificationService?wsdl

Note

In the URLs above, replace yourlocalip with the hostname where KEN is deployed.

Exposed Web Services

Initially, KEN exposes a web service method to send a notification. The *sendNotification* method is a simple String In/String Out method. It accepts one parameter (*notificationMessageAsXml*) and returns a notificationResponse as a String. For the format of the response, see the *Notification Response* document in the TRG for KEN.

Calling the *sendNotification* Service from JAVA

First, create a String that includes the XML content for the notification, as described in the Notification Message document of the TRG for KEN. In the following example code, the XML representation of the notification is read as a file from the file system in the main method, and the code calls the *MySendNotification* method to invoke the Notification web service.

A SOAP style web services binding stub is available in the **notification.jar** file, as described above in the **Dependencies** section.

You may use this code as a template for sending a notification using the web service:

```
package edu.cornell.library.notification;


import org.apache.commons.io.IOUtils;
import org.kuali.notification.client.ws.stubs.NotificationWebServiceSoapBindingStub;


import java.io.IOException;

import java.io.InputStream;
import java.net.URL;


public class MyNotificationWebServiceClient   {
  private final  static String WEB_SERVICE_URL = "http://localhost:8080/notification/services/Notification";

  public static void MySendNotification(String notificationMessageAsXml) throws Exception {
    URL url = new URL(WEB_SERVICE_URL);
    NotificationWebServiceSoapBindingStub stub = new NotificationWebServiceSoapBindingStub(url, null);
    String responseAsXml = stub.sendNotification(notificationMessageAsXml);
    // do something useful with the response
    System.out.println(responseAsXml);
  }
```

```
 public static void main(String[] args) {
   InputStream notificationXML =
MyNotificationWebServiceClient.class.getResourceAsStream("webservice_notification.xml");
   String notificationMessageAsXml = "";
   try {
     notificationMessageAsXml = IOUtils.toString(notificationXML);
   } catch (IOException ioe) {
     throw new RuntimeException("Error loading webservice_notification.xml");
   }

   try {
     MySendNotification(notificationMessageAsXml);
   } catch (Exception ioe) {
     throw new RuntimeException("Error running webservice");
   }
 }

}
```

KEN Authentication

Web

KEN can support any Web Sign On technology that results in the population of the HttpServletRequest remote user variable, exposed via the getRemoteUser accessor.

public java.lang.String getRemoteUser()

Returns the login of the user making this request, if the user has been authenticated, or null if the user has not been authenticated. Whether the user name is sent with each subsequent request depends on the browser and type of authentication.

Returns: A *String* specifying the login of the user making this request, or *null*

The generic KEN release comes configured with CAS.

Web Services

Web service authentication is part of the development process and is not implemented by the standalone release of Rice. The notification web service is Axis-based.

Chapter 3. KEW

What is Kuali Enterprise Workflow?

What is workflow, in general?

Workflow is a very general term and means different things in different contexts. For example, it may mean the sequence of approvals needed for a Leave Request or it may refer to a complex scientific procedure.

In our context of enterprise applications within a higher education institution, we're usually talking about business process management when we discuss workflow. Usually, this revolves around business rules, authorizations, and routing for approval.

A simple example is a leave request system. It needs some workflow to get the necessary people (supervisor, etc.) to approve it. This is one example of the routing and approval side of a workflow.

You may also have business rules in workflow that dictate that some people get automatic approval for leave requests. This is a business rule detail that workflow executes by automatically routing these types of requests past the approval steps.

What is Kuali Enterprise Workflow, in particular?

The Kuali Enterprise Workflow (KEW) product revolves around routing and approval of documents. It is a stand-alone *workflow engine* upon which you can integrate other enterprise applications to do routing and approvals.

In addition, KEW contains an **eDocLite** system. This is a mechanism to create simple data-entry forms directly in KEW. You can also create routing rules around eDocLite forms. eDocLite forms are the rough equivalent of the basic, one- or two-page forms that are commonly used to process business and get signature approvals.

The benefit of eDocLite in KEW is that it does *not* require a separate application. You can use eDocLite in KEW simply by setting up the forms that your institution or department needs.

Overall, KEW is based on documents. In KEW, each document has a collection of transactions or *things to be done*. Each transaction is approved or denied individually in KEW.

For example, John Doe may use a *Leave Request* document in KEW to ask for a week off in June. The KEW Leave Request document contains enough information for his supervisor to make a decision about John's leave. (The document may use data keys to retrieve external information, such as John's past Leave Requests and available hours.) Once John submits his Leave Request, KEW routes it to John's supervisor for approval. Depending on how John's department has configured KEW for routing Leave Requests, after John's supervisor approves or denies his request, KEW may route it to more people for further action to be taken.

Once John's Leave Request document is processed, it triggers a **PostProcessor**, which can peform any desired additional processing. This is most commonly used to "finalize" the business transaction once all approvers have signed off on it. In this particular example, it might call another service that would update records in the Leave Request application's database, indicating that the individual has successfully scheduled leave during that time period.

In addition, the KEW **PostProcessor** contains hooks for all the stages that a document goes through. For example, an external application may use a KEW workflow for routing and approval of documents, and that application may take action at each change in state of a routed document.

What problems or functions does KEW solve?

The primary benefit of KEW workflow is the correct routing for approval of documents. It enforces your business-specific rules about who needs to approve what documents, in which scenarios.

Simple Workflow Example

Leave Request: Each person has one other person (possibly more) who needs to approve his or her leave requests. In this context, KEW is the system that manages both the approval structure and the leave requests themselves (the actual approvals).

More Complex Workflow Example

Purchasing Desktop Computers: You may need several business rules in KEW for this, such as a rule to enforce:

1. A strategic alliance requires that you buy from one vendor unless there is a justification to not do so

2. General purchasing approval by the Purchasing Department is required when the cost of the purchase exceeds a certain limit

3. Approval by the account owners who fund the purchase is required

In this example, KEW requires an approval if:

• The strategic alliance is not used

• The cost limit for Purchasing Department approval is exceeded

The workflow also requires an approval by the signer (or delegate) for each spending account that you use for the purchase.

In KEW, **Approval Types** are set up such as account approver, supervisor, or organizational/department hierarchy approver. An Approval Type contains the applicable routing and approval rules. Once you create an approval type, those routing and approval rules are available for other workflow clients and scenarios. This creates a *tipping point* situation, in which the more applications and business processes you set up through workflow, the easier it gets to do new ones.

In addition, KEW can help you with distributed management of approval structures. Each group at your institution (each college, unit, division, etc.) can create their own approval and workflow structure for their group, and you can centrally manage the workflow above those groups. This allows groups to manage their own internal controls and structures, while still being subject to higher-level institutional controls.

What problems does KEW NOT solve?

KEW is not a general-purpose application builder. For complex applications, you need to develop applications separately and then integrate them with KEW. For simple forms or documents that need approval, you can use **eDocLite**, but this only works in simple cases, analogous to a one- or two-page paper form that requires signatures. It is important to note, however, that Kuali Rice does include a framework called the Kuali Nervous System (KNS) that can be used to facilitate the development of more complex applications and includes built-in integration with KEW.

KEW is not a general-purpose business rules engine. For example, it does not know that a continuation account must be specified when an account is closed. Those types of rules are the responsibility of the

application itself to manage. However, this is not a clear-cut line, as KEW does manage business rules that directly affect routing and approval.

KEW is not an Organization Hierarchy manager. For example, it will not automatically manage your organizational hierarchies and internal structures. However, integration with these hierarchies and structures can be accomplished using KEW, and leveraging such hierarchies for routing and approval is a very common need for many applications.

With which applications can KEW integrate?

Nearly anything, in theory. In the current version of KEW, any application can access KEW if it can:

- Do Java method calls, or

- Do remote method invocation, or

- Do web-services calls, or

- Communicate with the Kuali Service Bus (KSB)

(The recommended cross-platform integration method is over web services.)

Can I use KEW without building an entire application?

Yes, absolutely!

KEW is an incredibly powerful platform for routing and approval for enterprise (i.e., large) applications. However, it also includes **eDocLite**, which makes it easy to develop simple business-process forms and run them through KEW. In this situation, in its most simple form, you can do all of your work within KEW, and most of that work is in developing your form configurations. If needed, the eDocLite process can also hook into a post-processor to take an action once a document's approvals are complete.

Steps to Building a KEW Application

Preface

In its simplest form, KEW is merely a set of services that can be used to submit documents to a workflow engine and then interact with those documents as the progress through the routing process. Therefore, there are many different ways to build an application that uses KEW. Kuali Rice itself has a few built-in solutions (eDocLite and KNS) that make it easier to build applications that use KEW. Alternatively, an application can be built from scratch or retrofitted to use KEW.

In this section, we will look at some common approaches to designing and building an application which leverages KEW. However, it is by no means exhaustive and is simply meant to get you started and give you ideas as you embark upon development of your own applications that use Kuali Enterprise Workflow.

Initial Steps - Determine the Routing Rules

Determine to whom you want to route the document and when it should be routed. For example, in the **Travel Request Sample Workflow Client Application**, the steps in the routing process are:

1. Someone submits a travel request for a traveler

2. Traveler receives an *Approve Action Item*

3. Traveler's supervisor receives *Approve Action Item*

4. Traveler's dean/director receives *Acknowledge Action Item*

5. Fiscal Officer for account(s) receives *Approve Action Item*

Configure the Process Definition

In KEW, process definitions are attached to **Document Types**. The Document Type allows for configuration of various pieces of the business process in addition to the process definition.

The Document Type is defined in XML format. KEW can ingest files containing this Document Type configuration to set up the specified workflows and then executes the workflows based on that configuration.

One example of routing configuration is the Travel Request application. The Document Type configuration is defined in the following four XML files:

TravelRoutingConfiguration.xml - Defines the **travelDocument** Document Type, including *PostProcessor*, *docHandler*, and *routeNodes*:

```
<?xml version="1.0" encoding="UTF-8"?>
<data xmlns="ns:workflow" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="ns:workflow
 resource:WorkflowData">
    <documentTypes xmlns="ns:workflow/DocumentType" xsi:schemaLocation="ns:workflow/DocumentType
 resource:DocumentType">
        <documentType>
            <name>TravelRequest</name>
            <description>Create a New Travel Request</description>
            <label>Travel Request</label>
            <postProcessorName>org.kuali.rice.kns.workflow.postprocessor.KualiPostProcessor</postProcessorName>
            <superUserGroupName namespace="TVL">SuperUserGroup</superUserGroupName>
            <blanketApproveGroupName namespace="TVL">BlanketApproveGroup</blanketApproveGroupName>
            <defaultExceptionGroupName namespace="TVL">ExceptionGroup</defaultExceptionGroupName>

            <docHandler>${application.url}/travelDocument2.do?methodToCall=docHandler</docHandler>
            <routePaths>
                <routePath>
                    <start name="Initiated" nextNode="DestinationApproval" />
                    <requests name="DestinationApproval" nextNode="TravelerApproval" />
                    <requests name="TravelerApproval" nextNode="SupervisorApproval" />
                    <requests name="SupervisorApproval" nextNode="AccountApproval" />
                    <requests name="AccountApproval" />
                </routePath>
            </routePaths>
            <routeNodes>
                <start name="Initiated">
                    <activationType>P</activationType>
                </start>
                <requests name="DestinationApproval">
                    <ruleTemplate>TravelRequest-DestinationRouting</ruleTemplate>
                </requests>
                <requests name="TravelerApproval">
                    <ruleTemplate>TravelRequest-TravelerRouting</ruleTemplate>
                </requests>
                <requests name="SupervisorApproval">
                    <ruleTemplate>TravelRequest-SupervisorRouting</ruleTemplate>
                </requests>
                <requests name="AccountApproval">
                    <ruleTemplate>TravelRequest-AccountRouting</ruleTemplate>
                </requests>
            </routeNodes>
        </documentType>
    </documentTypes>
```

```
                                                                                       </data>
```

TravelRuleAttributes.xml – Defines the attributes used by the Workflow Engine to determine to whom to route to next:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<data xmlns="ns:workflow" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="ns:workflow
 resource:WorkflowData">
    <ruleAttributes xmlns="ns:workflow/RuleAttribute" xsi:schemaLocation="ns:workflow/RuleAttribute
 resource:RuleAttribute">
        <ruleAttribute>
            <name>EmployeeAttribute</name>
            <className>edu.sampleu.travel.workflow.EmployeeAttribute</className>
            <label>Employee Routing</label>
            <description>Employee Routing</description>
            <applicationId>TRAVEL</applicationId>
            <type>RuleAttribute</type>
        </ruleAttribute>

        <ruleAttribute>
            <name>AccountAttribute</name>
            <className>edu.sampleu.travel.workflow.AccountAttribute</className>
            <label>Account Routing</label>
            <description>Account Routing</description>
            <applicationId>TRAVEL</applicationId>
            <type>RuleAttribute</type>
        </ruleAttribute>
    </ruleAttributes>
</data>
```

TravelRuleTemplates.xml - Defines the **RuleTemplates** that represent each routeNode listed in the Document Type configuration:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<data xmlns="ns:workflow" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="ns:workflow
 resource:WorkflowData">
    <ruleTemplates xmlns="ns:workflow/RuleTemplate" xsi:schemaLocation="ns:workflow/RuleTemplate
 resource:RuleTemplate">
        <ruleTemplate allowOverwrite="true">
            <name>TravelRequest-DestinationRouting</name>
            <description>Destination Routing</description>
            <attributes>
                <attribute>
                    <name>DestinationAttribute</name>
                </attribute>
            </attributes>
        </ruleTemplate>
        <ruleTemplate allowOverwrite="true">
            <name>TravelRequest-TravelerRouting</name>
            <description>Traveler Routing</description>
            <attributes>
                <attribute>
                    <name>EmployeeAttribute</name>
                </attribute>
            </attributes>
        </ruleTemplate>
        <ruleTemplate allowOverwrite="true">
            <name>TravelRequest-SupervisorRouting</name>
            <description>Supervisor Routing</description>
            <attributes>
                <attribute>
                    <name>EmployeeAttribute</name>
                </attribute>
            </attributes>
        </ruleTemplate>
        <ruleTemplate allowOverwrite="true">
            <name>TravelRequest-AccountRouting</name>
            <description>Travel Account Routing</description>
            <attributes>
                <attribute>
```

```
                    <name>AccountAttribute</name>
                </attribute>
            </attributes>
        </ruleTemplate>
    </ruleTemplates>
    </data>
```

TravelRules.xml - Defines the rules (a rule is a combination of *Document Type*, *Rule Template* and *Responsibilities*) that the workflow engine uses to determine to whom to route to next:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<data xmlns="ns:workflow" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="ns:workflow
 resource:WorkflowData">
    <rules xmlns="ns:workflow/Rule" xsi:schemaLocation="ns:workflow/Rule resource:Rule">
        <rule>
            <name>TravelRequest-DestinationLasVegas</name>
            <documentType>TravelRequest</documentType>
            <ruleTemplate>TravelRequest-DestinationRouting</ruleTemplate>
            <description>Destination Rule</description>
            <ruleExtensions>
                <ruleExtension>
                    <attribute>DestinationAttribute</attribute>
                    <ruleTemplate>TravelRequest-DestinationRouting</ruleTemplate>
                    <ruleExtensionValues>
                        <ruleExtensionValue>
                            <key>destination</key>
                            <value>las vegas</value>
                        </ruleExtensionValue>
                    </ruleExtensionValues>
                </ruleExtension>
            </ruleExtensions>
            <responsibilities>
                <responsibility>
                    <principalName>user4</principalName>
                    <actionRequested>A</actionRequested>
                </responsibility>
            </responsibilities>
        </rule>
        <rule>
            <name>TravelRequest-EmployeeRole</name>
            <documentType>TravelRequest</documentType>
            <ruleTemplate>TravelRequest-TravelerRouting</ruleTemplate>
            <description>Traveler Routing</description>
            <responsibilities>
                <responsibility>
                    <role>edu.sampleu.travel.workflow.EmployeeAttribute!employee</role>
                    <actionRequested>A</actionRequested>
                </responsibility>
            </responsibilities>
        </rule>
        <rule>
            <name>TravelRequest-SupervisorRole</name>
            <documentType>TravelRequest</documentType>
            <ruleTemplate>TravelRequest-SupervisorRouting</ruleTemplate>
            <description>Supervisor Routing</description>
            <responsibilities>
                <responsibility>
                    <role>edu.sampleu.travel.workflow.EmployeeAttribute!supervisr</role>
                    <actionRequested>A</actionRequested>
                </responsibility>
            </responsibilities>
         </rule>
        <rule>
            <name>TravelRequest-DirectorRole</name>
            <documentType>TravelRequest</documentType>
            <ruleTemplate>TravelRequest-SupervisorRouting</ruleTemplate>
            <description>Dean/Director Routing</description>
            <responsibilities>
                <responsibility>
                    <role>edu.sampleu.travel.workflow.EmployeeAttribute!director</role>
                    <actionRequested>K</actionRequested>
                </responsibility>
            </responsibilities
        </rule>
```

```
        <rule>
            <name>TravelRequest-FiscalOfficerRole</name>
            <documentType>TravelRequest</documentType>
            <ruleTemplate>TravelRequest-AccountRouting</ruleTemplate>
            <description>Fiscal Officer Routing</description>
            <responsibilities>
                <responsibility>
                    <role>edu.sampleu.travel.workflow.AccountAttribute!FO</role>
                </responsibility>
            </responsibilities>
        </rule>
    </rules>
</data>
```

Client PlugIn Steps

Your plugin should contain Java classes that correspond to the attributes defined in the XML configuration file. The Travel Request Sample Client contains two attribute classes: *EmployeeAttribute* and *AccountAttribute*. Each of these classes implements these two interfaces:

```
org.kuali.rice.kew.rule.RoleAttribute
org.kuali.rice.kew.rule.WorkflowAttribute
```

Using the *EmployeeAttribute* as an example, here are the implementations for the *RoleAttribute* interface:

getRoleNames() - Returns a list of role names to display on the routing rule GUI in the KEW web application:

```
private static final Map ROLE_INFO;

static {

    ROLE_INFO = new TreeMap();
    ROLE_INFO.put(EMPLOYEE_ROLE_KEY, "Employee");
    ROLE_INFO.put(SUPERVISOR_ROLE_KEY, "Supervisor");
    ROLE_INFO.put(DIRECTOR_ROLE_KEY, "Dean/Director");
}

public List getRoleNames() {
    List roleNames = new ArrayList();
    for (Iterator iterator = roles.keySet().iterator(); iterator.hasNext();) {
        String roleName = (String) iterator.next();
        roleNames.add(new Role(getClass(), roleName, roleName));
    }
    return roleNames;
}
```

getQualifiedRoleNames() - Returns a list of strings that represents the qualified role name for the given roleName and XML docContent which is attached to the workflow document:

```
/**
     * Returns a String which represent the qualified role name of this role for the given
       * roleName and docContent.
       * @param roleName the role name (without class prefix)
       * @param documentContent the document content
*/

public List<String> getQualifiedRoleNames(String roleName, DocumentContent documentContent) {
    List<String> qualifiedRoleNames = new ArrayList<String>();
    Map<String, List<String>> qualifiedRoles = (Map<String, List<String>>)roles.get(roleName);
    if (qualifiedRoles != null) {
        qualifiedRoleNames.addAll(qualifiedRoles.keySet());
    } else {
        throw new IllegalArgumentException("TestRuleAttribute does not support the given role " + roleName);
```

```
    }
    return qualifiedRoleNames;
}
```

resolveQualifiedRole() - Returns a list of workflow users that are members of the given Qualified Role. (Used to help determine to whom to route the document.):

```
/**
  * Returns a List of Workflow Users which are members of the given qualified role.
  * @param routeContext the RouteContext
  * @param roleName the roleName (without class prefix)
  * @param qualifiedRole one of the the qualified role names returned from the {@link
 #getQualifiedRoleNames(String, DocumentContent)} method
  * @return ResolvedQualifiedRole containing recipients, role label (most likely the roleName), and an
 annotation
*/

public ResolvedQualifiedRole resolveQualifiedRole(RouteContext routeContext, String roleName, String
 qualifiedRole) {
    ResolvedQualifiedRole resolved = new ResolvedQualifiedRole();
    Map<String, List<String>> qualifiedRoles = (Map<String, List<String>>)roles.get(roleName);

    if (qualifiedRoles != null) {
        List<String> recipients = (List<String>)qualifiedRoles.get(qualifiedRole);
        if (recipients != null) {
            resolved.setQualifiedRoleLabel(qualifiedRole);
            resolved.setRecipients(convertPrincipalIdList(recipients));
        } else {
            throw new IllegalArgumentException("TestRuleAttribute does not support the qualified role " +
 qualifiedRole);
        }

    } else {
        throw new IllegalArgumentException("TestRuleAttribute does not support the given role " + roleName);
    }
    return resolved;
}
```

Using the *EmployeeAttribute* example, here are the implementations for the *WorkflowAttribute* interface:

getRoutingDataRows() – Returns a list of *RoutingDataRows* that contain the user interface level presentation of the *ruleData* fields. KEW uses the ruleData fields to determine where a given document would be routed according to the associated rule:

```
public List<Row> getRoutingDataRows() {
    List<Row> rows = new ArrayList<Row>();
    List<Field> fields = new ArrayList<Field>();
    fields.add(new Field("Traveler username", "", Field.TEXT, false, USERID_FORM_FIELDNAME, "", false, false,
 null, null));
    rows.add(new Row(fields));
    return rows;
}
```

getDocContent() - Returns a string containing this Attribute's *routingData* values, formatted as a series of XML tags:

```
public String getDocContent() {
    String docContent = "";

    if (!StringUtils.isBlank(_uuid)) {
        String uuidContent = XmlUtils.encapsulate(UUID_PARAMETER_TAGNAME, _uuid);

        docContent = _attributeParser.wrapAttributeContent(uuidContent);
    }

    return docContent;
```

```
}
```

validateRoutingData() - Validates *routingData* values in the incoming map and returns a list of errors from the routing data. (The user interface calls **validateRoutingData()** during rule creation.):

```
public List validateRoutingData(Map paramMap) {
    List errors = new ArrayList();

    String principalName = StringUtils.trim((String) paramMap.get(PRINCIPAL_NAME_FORM_FIELDNAME));
    if (isRequired() && StringUtils.isBlank(principalName)) {
        errors.add(new WorkflowServiceErrorImpl("principalName is required",
 "accountattribute.principalName.required"));
    }

    if (!StringUtils.isBlank(principalName)) {
        KimPrincipalInfo principal =
 KIMServiceLocator.getIdentityService().getPrincipalByPrincipalName(principalName);
        if (principal == null) {
             errors.add(new WorkflowServiceErrorImpl("unable to retrieve user for principalName '" +
 principalName + "'", "accountattribute.principalName.invalid"));
        }
    }
    if ( errors.size() == 0 ) {
        _principalName = principalName;
    }
    return errors;
}
```

Build PostProcessor and Services

The PostProcessor class should implement the interface:

```
org.kuali.rice.kew.postprocessor.PostProcessorRemote
```

You should use this interface for business logic that should execute when the document transitions to a new status or when actions are taken on the document. The PostProcessor for the Travel Request Client is the class:

```
org.kuali.rice.kns.workflow.postprocessor.KualiPostProcessor
```

that implements the doRouteStatusChange() method to update the status of the travel document in the Travel database. The KualiPostProcessor in this case is the standard PostProcessor used on all documents that are built on the KNS framework.

Package PlugIn

Depending on how the application has been developed (i.e. embedded workflow engine vs. using the engine as a remote service) it may be necessary to package components like the PostProcessor into a plug-in. See the Workflow PlugIn Guide for details on how to do this.

Client Web Application Steps

Build the Web Application

Begin to build a Kuali Enterprise Workflow the same as you build any other Java-enabled web application. You build it with all the business logic for the application and, for example, communication to the workflow engine using web services.

As an example, the Travel Request Client Web Application uses Struts, Spring, and OJB.

Build the Service that Connects to the Workflow Engine

For the rest of this section, this guide refers to the Java application communicating with the Kuali Enterprise Workflow as the *Client Application*. The Client Application needs a service that will interact with the workflow system. This service will perform actions such as locating a document in the workflow system and routing the document.

Below are examples from the Travel Request Sample Client. The methods in the *TravelDocumentService* class find a *TravelDocument* in the workflow system, save and route a *TravelDocument*, and validate a *TravelDocument*.

findByDocHeaderId() - Finds a Document in the workflow engine:

```
public TravelDocument findByDocHeaderId(Long docHeaderId, String principalId) {
    if (docHeaderId == null) {
        throw new IllegalArgumentException("invalid (null) docHeaderId");
    }
    TravelDocument result = travelDocumentDao.findByDocHeaderId(docHeaderId);
    if (result != null) {
        // convert DocAccountJoins into FinancialAccounts
        ArrayList accounts = new ArrayList();
        for (Iterator joins = result.getDocAccountJoins().iterator(); joins.hasNext();) {
            DocumentAccountJoin join = (DocumentAccountJoin) joins.next();

            FinancialAccount account = financialAccountService.findByAccountNumber(join.getAccountNumber());

            accounts.add(account);
        }
        result.setFinancialAccounts(accounts);
        try {
            WorkflowDocument document = new WorkflowDocument( principalId, result.getDocHeaderId());
        } catch (WorkflowException e) {
            LOG.error("caught WorkflowException: ", e);
            throw new RuntimeException(e);
        }
    }
    return result;
}
```

The **TravelDocumentServiceImpl** class populates the attribute values on the workflow document (Employee, Account) that will be used for future routing. It does this by calling its *getEmployeeAttribute*() and *getAccountAttribute*() methods and adding the results to the workflow document by calling the *addAttributeDefinition*() method.

```
private WorkflowAttributeDefinitionVO getEmployeeAttribute(TravelDocument travelDocument) {
    WorkflowAttributeDefinitionDTO attrDef = new
 WorkflowAttributeDefinitionDTO("edu.sampleu.travel.workflow.EmployeeAttribute");
    String principalName = travelDocument.getTravelerUsername();
    attrDef.addConstructorParameter(principalName);
    return attrDef;
}



private List getAccountAttributes(TravelDocument travelDocument) {
    List accounts = travelDocument.getFinancialAccounts();
    List accountAttributes = new ArrayList();
    for (Iterator accountIterator = accounts.iterator(); accountIterator.hasNext();) {
        WorkflowAttributeDefinitionDTO attrDef = new
 WorkflowAttributeDefinitionDTO("edu.sampleu.travel.workflow.AccountAttribute");
        FinancialAccount account = (FinancialAccount)accountIterator.next();
        attrDef.addConstructorParameter(account.getAccountNumber());
        accountAttributes.add(attrDef);
    }
    return accountAttributes;
```

```
}
```

Build the Action Class with Workflow Lifecycle Methods

In the Travel Request Sample Client, the **WorkflowDocHandlerAction** struts action class calls the workflow lifecycle methods (approve, acknowledge, etc.) on the workflow document.

WorkflowDocHandlerAction - Take approve action. (Each workflow action - acknowledge, complete, etc. - is like this):

```
public ActionForward approve(ActionMapping mapping, ActionForm form, HttpServletRequest request,
 HttpServletResponse response) throws Exception {
    LOG.info("entering approve() method ...");
    DocHandlerForm docHandlerForm = (DocHandlerForm) form;
    WorkflowDocument document = docHandlerForm.getWorkflowDocument()
 document.approve(docHandlerForm.getAnnotation());
    saveDocumentActionMessage("general.routing.approved", request);
    LOG.info("forwarding to actionTaken from approve()");
    return mapping.findForward("actionTaken");

}
```

Set up the **WorkflowDocument** in the **initializeBaseFormState()** method of the **DispatchActionBase** from which the Struts action classes inherit. Obtain the workflow document with this line of code:

```
String principalId = getUserSession(request).getPrincipalId();
WorkflowDocument document = new WorkflowDocument(principalId, docId);
```

Package the Web Application

Package the Client Application (client web application) for deployment the way you normally package web applications. The Travel Request Sample Web Application does this with an Ant build script. The *dist* step of the *build.xml* script builds the *SampleWorkflowClient.war* file.

Final Steps

Deploy the PlugIn

Deploy the plugin to your workflow installation. Copy the plugin directory structure to your application plugins directory. Please see the Workflow Plugin Guide for more information.

Deploy the Client Web Application

Deploy the Client Web Application to your Application server the way you normally deploy web applications.

KEW Configuration

KEW Integration Options

The following integration options are available to applications integrating with KEW:

- **Embedded** - The KEW engine is embedded into a Java application. The Standalone Rice Server is required.

- **Bundled** - Same as Embedded mode except that the entire KEW web application is also embedded into the Java application. The Standalone Rice Server is not required.

- **Remote Java Client** – A Java client is used which relies on the service bus to communicate with a Standalone Rice Server's KEW services.

- **Thin Java Client** - A thin Java client is used which communicates with a Standalone Rice Server over remote service calls.

- **Web Services** - Interacts directly with web services on a Standalone Rice Server.

Table 3.1. Advantages/Disadvantages of KEW Integration Options

Integration Option	Advantages	Disadvantages
Embedded	• Integration of database transactions between client application and embedded KEW (via JTA) • Performance - Embedded client talks directly to database • No need for application plug-ins on the server • Great for Enterprise deployment, there is still a single shared Standalone Rice web application but scalability is increased because of multiple Workflow Engines	• Can only be used by Java clients • More library dependencies than the Thin Client model • Requires client application to establish connections to Kuali Rice database
Bundled	• All the advantages of Embedded Mode • No need to deploy a standalone Rice server • Ideal for development or "quickstart" applications • Application can be bundled with Rice for ease of development/distribution • Can switch to Embedded Mode for deployment in an Enterprise environment	• Not desirable for Enterprise deployment where more than one application is integrated with Rice and KEW • More library dependencies than the Thin Client model and Embedded Mode (additional web libraries)
Remote Java Client	• Relatively simple configuration • Client can access more external KEW services from the Standalone Rice Server than the Thin Java Client, and yet the client does not need to have an embedded KEW engine	• Requires client application to be KSB-enabled, unlike the Thin Java Client • Cannot be used by KNS-enabled client applications
Thin Java Client	• Relatively simple configuration • Fewer Library Dependencies	• No transactional integration between client and server • Plug-ins must be deployed to the server if custom routing components are needed
Web Services	• Any language which supports web services can be used	• No transactional integration between client and server • Plug-ins must be deployed to the server if custom routing components are needed • Web Services can be slower than other integration options

Standalone Server

To effectively use any of the KEW integration modes besides bundled, a Standalone Rice Server will need to be deployed.

Embedded Deployment Diagram

Here is a diagram illustrating what a sample embedded deployment might look look.

Figure 3.1. Embedded Deployment Diagram example
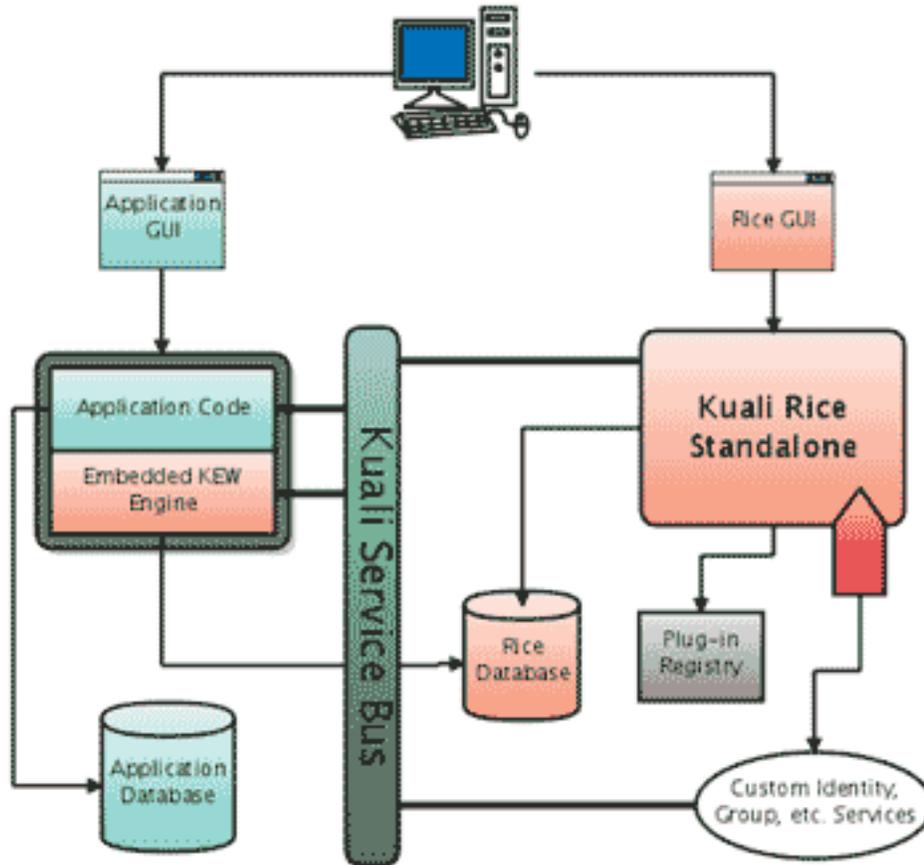
Bundling the KEW Application

web.xml

Bundled mode is the same as *embedded* mode except that the client application embeds the entire Kuali Rice system within it (including the web application). The embedding of the web application portion is accomplished by utilizing Struts Modules.

Configuration is the same as *embedded* mode, with the exception of loading the web application portions in the **web.xml**:

```
<filter>
    <filter-name>UserLoginFilter</filter-name>
    <filter-class>org.kuali.rice.kew.web.UserLoginFilter</filter-class>

</filter>


<filter-mapping>
    <filter-name>UserLoginFilter</filter-name>
    <servlet-name>action</servlet-name>
</filter-mapping>


<servlet>
```

```
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    .. other struts configuration if applicable
    <init-param>
        <param-name>config/en</param-name>
        <param-value>/en/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <load-on-startup>0</load-on-startup>
</servlet>


<servlet>
    <servlet-name>remoting</servlet-name>
    <servlet-class>org.kuali.rice.ksb.messaging.servlet.KSBDispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>



<servlet>
    <servlet-name>export</servlet-name>
    <servlet-class>org.kuali.rice.kew.export.web.ExportServlet</servlet-class>
</servlet>


<servlet>
    <servlet-name>attachment</servlet-name>
    <servlet-class>org.kuali.rice.kew.notes.web.AttachmentServlet</servlet-class>
</servlet>



<servlet>
    <servlet-name>edoclite</servlet-name>
    <servlet-class>org.kuali.rice.kew.edl.EDLServlet</servlet-class>
</servlet>



<servlet-mapping>
    <servlet-name>remoting</servlet-name>
    <url-pattern>/remoting/*</url-pattern>
</servlet-mapping>


<servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>

<servlet-mapping>


    <servlet-name>export</servlet-name>
    <url-pattern>/export/*</url-pattern>
</servlet-mapping>


<servlet-mapping>

    <servlet-name>attachment</servlet-name>
    <url-pattern>/en/attachment/*</url-pattern>
</servlet-mapping>


<servlet-mapping>
    <servlet-name>edoclite</servlet-name>
    <url-pattern>/en/EDocLite</url-pattern>
</servlet-mapping>
```

org.kuali.rice.kew.web.UserLoginFilter – This filter is used to assist the KEW bundled web application in determining who the authenticated user is. Specifically, the login filter invokes the KIM identity management service to determine the identity of the authenticated user.

Typically, a previously executed filter will challenge the user on entry to a Rice web page for their authentication credentials using CAS or some other form of single sign on (SSO) authentication system.

For development and testing purposes, Rice provides a simple filter implementation that will present a simple sign on screen. This screen displays only a single login entry field and submit button. The user can enter their username (no password) and press the submit button, and the system authenticates the user for entry into the system.

This can be configured as follows in the **web.xml**:

```
<filter>
    <filter-name>LoginFilter</filter-name>
    <filter-class>org.kuali.rice.kew.web.UserLoginFilter</filter-class>

</filter>


<filter-mapping>
    <filter-name>LoginFilter</filter-name>
    <servlet-name>action</servlet-name>
</filter-mapping>
```

and in the **rice-config.xml**:

```
<param name="filter.login.class">org.kuali.rice.kew.web.DummyLoginFilter</param>
<param name="filtermapping.login.1">/*</param>
```

org.apache.struts.action.ActionServlet - The Struts servlet which loads the KEW Struts module. The module name should be 'en'. Struts only allows a single Action Servlet so if you are using Struts in your application, all of your Struts modules will need to be configured using the init-param elements in this servlet definition.

org.kuali.rice.ksb.messaging.servlet.KSBDispatcherServlet - A servlet which dispatches http requests for the Kuali Service Bus (see KSB documentation for more details). The servlet mapping here should correspond to the **serviceServletUrl** configuration parameter for the KSBConfigurer.

org.kuali.rice.kew.export.web.ExportServlet - serves exports of lookup results as XML files

org.kuali.rice.kew.notes.web.AttachmentServlet - serves attachments that have been attached to documents using the KEW Notes and Attachments framework

org.kuali.rice.kew.edl.EDLServlet - The servlet used to interact with eDocLite documents. See eDocLite documentation for more information.

Bundled Deployment Diagram

Figure 3.2. Bundled deployment diagram
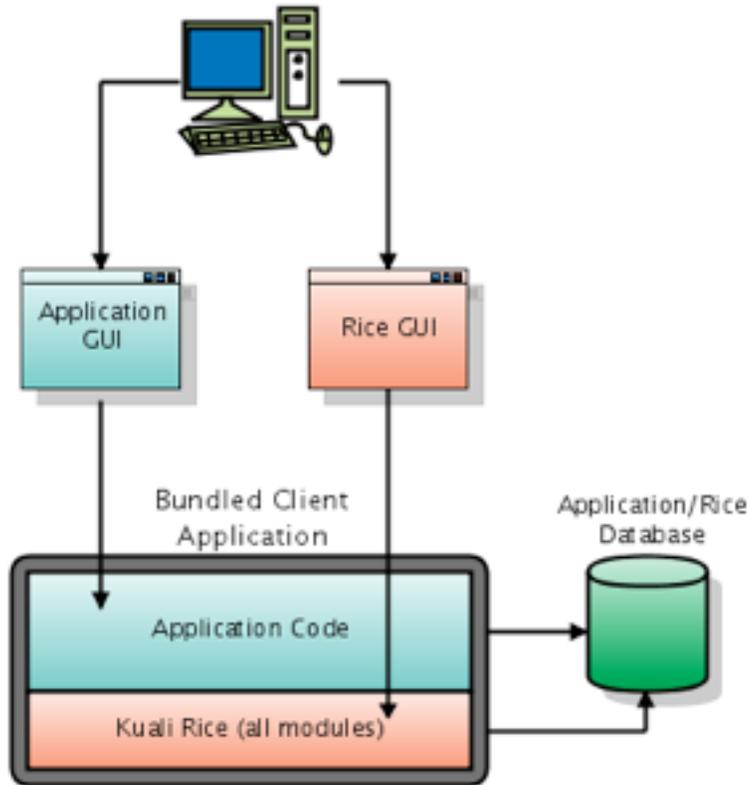
Using the Remote Java Client

Along with the previous embedded configurations, KEW also allows for Remote Java Clients, which communicate with KEW services that are available on the service bus. Configuration of the remote client is similar to that of the embedded client, except that no embedded KEW engine gets set up; instead, the client relies on the service bus for accessing the KEW services of the Standalone Rice Server.

Caution

Limitations of Remote KEW Java Clients:

At present, KNS-enabled Java clients cannot be used as Remote KEW Java Clients.

Using the Thin Java Client

In addition to the embedded configurations discussed previously, KEW also provides a thin java client which can be used to talk directly to two KEW services exposed on the service bus.

These KEW services are:

• WorkflowDocumentService - provides methods for creating, loading, approving and querying documents

- WorkflowUtilityService - provides methods for querying for various pieces of information about the KEW system

Additionally, access to two KIM services is required, as Principal and Group information is needed to use many of the methods in the KEW services above.

These KIM services are:

- kimIdentityService - provides methods to query for Principal and Entity information

- kimGroupService - provides methods to query for Group information

Of course, this configuration requires Standalone Rice Server deployment. The workflow engine deployed within Standalone Rice Server is used for processing documents that integrate using a thin client.

These services are exposed on the KSB as Java services, meaning they use Java Serialization over HTTP to communicate. Optionally, the KEW services can also be secured to provide access to only those callers with authorized digital signatures (note that secure access is required for the KIM services). In order to configure the thin client, the following configuration properties need to be defined.

Required Thin Client Configuration Properties

Table 3.2. Required Thin Client Configuration Properties

Property	Description
encryption.key	The secret key used by the encryption service; Must match the setting on the standalone server
keystore.alias	Alias of the application's key within the keystore
keystore.file	Path to the application's keystore file
keystore.password	Password to the keystore and the key with the configured alias
workflowdocument.javaservice.endpoint	Endpoint URL for the Workflow Document service
workflowutility.javaservice.endpoint	Endpoint URL for the Workflow Utility service
identity.javaservice.endpoint	Endpoint URL for the KIM identity service
group.javaservice.endpoint	Endpoint URL for the KIM group service

Note

It is simplest to use an identical keystore file and configuration in your thin client application to that on your standalone server.

Optional Thin Client Configuration Properties

Table 3.3. Optional Thin Client Configuration Properties

Property	Description
secure.workflowdocument.javaservice.endpoint	true/false value indicating if endpoint is secured (defaults to true); Must match the setting on the standalone server
secure.workflowutility.javaservice.endpoint	true/false value indicating if endpoint is secured (defaults to true); Must match the setting on the standalone server

Thin Client Spring Configuration

Here is the Spring configuration for a thin client in **ThinClientSpring.xml**:

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
    <!-- point Rice to the file containing your configuration params -->
    <!-- which should include a parameter setting kew.mode to "THIN" -->
    <bean id="config" class="org.kuali.rice.core.config.spring.ConfigFactoryBean">
        <property name="configLocations">
            <list>
                <value>classpath: yourThinClientApp-config.xml</value>
            </list>
        </property>
    </bean>
    <!-- Pull your configuration params out as Properties -->
    <bean id="configProperties"
class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
        <property name="targetObject" ref="config" />
        <property name="targetMethod" value="getProperties" />
    </bean>
    <!-- expose configuration params to Spring -->
    <bean class=
"org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="properties" ref="configProperties" />
    </bean>
    <!-- The RiceConfigurer that sets up thin client mode -->
    <bean id="rice" class="org.kuali.rice.kew.config.KEWConfigurer">
        <!-- inject the "config" bean into our configurer -->
        <property name="rootConfig" ref="config" />
    </bean>
</beans>
```

For more details on configuring Rice for a thin client, please see the [Thin Client Implementation](#) subsection of this Technical Reference Guide.

Endpoint URLs

Since KEW and KIM use the KSB to expose their services, the endpoint URLs are the same as those exported by the KSB.

An example configuration for these might be:

```
<param name=
"workflowdocument.javaservice.endpoint">http://yourlocalip/kr-dev/remoting/WorkflowDocumentActionsService</
param>
<param name=
"workflowutility.javaservice.endpoint">http://yourlocalip/kr-dev/remoting/WorkflowUtilityService</param>
<param name=
"identity.javaservice.endpoint">http://yourlocalip/kr-dev/remoting/kimIdentityService</param>
<param name=
"group.javaservice.endpoint">http://yourlocalip/kr-dev/remoting/kimGroupService</param>
```

Thin Client Deployment Diagram

Here is a diagram showing what a thin client deployment might look like.

Figure 3.3. Thin client deployment diagram
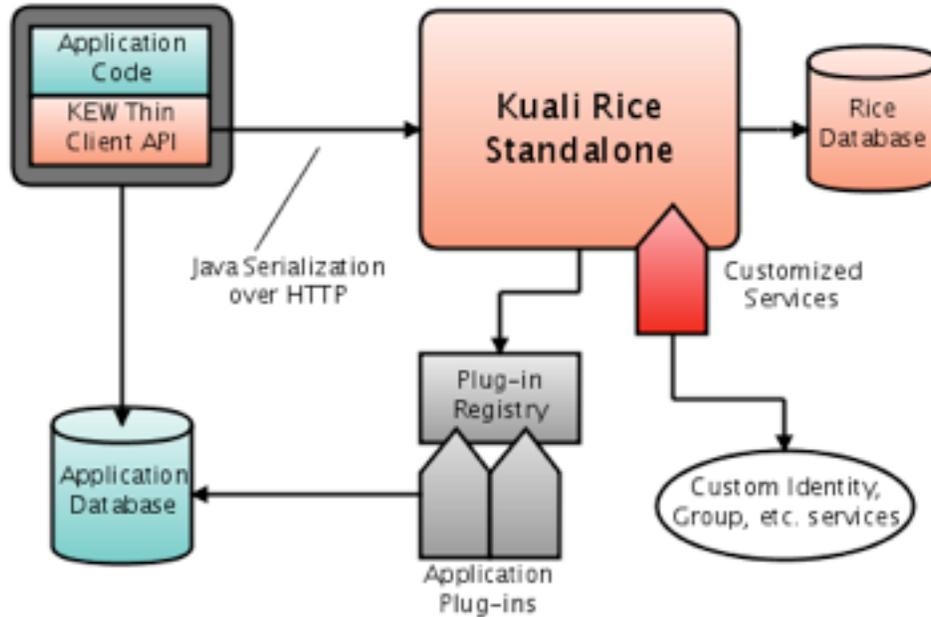
Picture of an Enterprise Deployment

As can be seen from the various integration options described, a KEW Enterprise Deployment (and Kuali Rice in general) might very well be a distributed environment with multiple systems communicating with each other.

The diagram below shows what a typical Enterprise deployment of Kuali Rice might look like.

Figure 3.4. Typical enterprise deployment of Kuali Rice
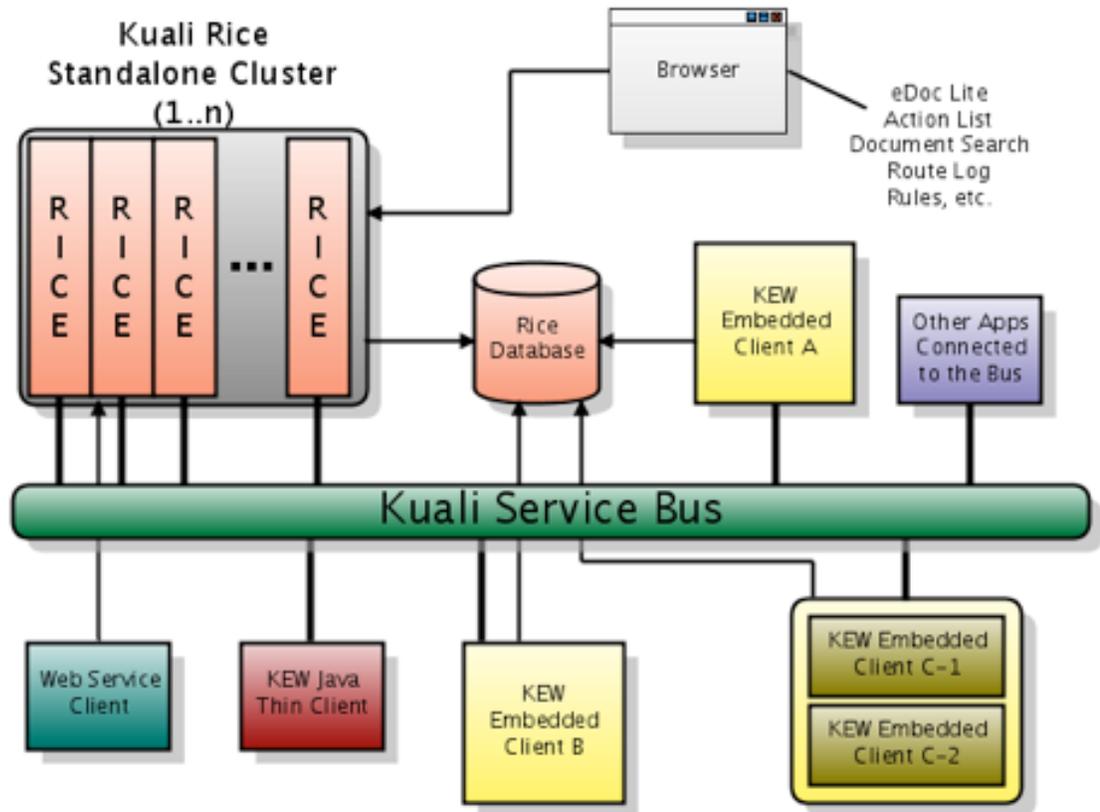
KEW Core Parameters

The display below includes those basic set of parameters for **rice-config.xml** as the minimal parameters to startup the Rice software. These parameters are a beginning reference to you for modification to your **rice-config.xml**.

Note

Please verify that your application.url and database username/password are set correctly.

Table 3.4. KEW Core Parameters

Core	Description	Examples\Values
workflow.url	URL to the KEW web module (i.e., ${application.url}/en)	
plugin.dir	Directory from which plugins will be loaded	
attachment.dir.location	Directory where attachments will be stored	

As a minimum, you must enable the dummy login filter by adding these lines to the rice-config.xml file for default login screen:

```
<param name="filter.login.class">org.kuali.rice.kew.web.DummyLoginFilter</param>
<param name="filtermapping.login.1">/*</param>
```

KEW Configuration Properties

Table 3.5. KEW Configuration Properties

Property	Description	Default
actionlist.outbox	Determines if the KEW actionlist "outbox" (ie, the actions already completed) will be viewable by users of the Rice application.	false
actionlist.outbox.default.preference.on	Determines if the KEW actionlist "outbox" is the default mode for viewing the action list.	false
base.url	Base URL under which Action List and other KEW screens can be found	Example: if your action list URL is http://yourlocalip/en/ActionList.do, set this property to http://yourlocalip/
client.protocol	Same as clientProtocol property on KEWConfigurer, this property can be configured in either place	embedded
data.xml.root.location	The temporary location of files being processed by the KEW XmlPollingService	/tmp/${environment}/kew/xml
document.lock.timeout	Used by the Oracle database platform to determine how long database locks on the document header are used	
email.reminder.lifecycle.enabled	If true, turns on timed job to send out regular e-mails to remind users of actions still waiting in their action list	
extra.classes.dir	Directory where classes for KEW plugins are located	
extra.lib.dir	Directory where libraries for KEW plugins are located	
kew.mode	The mode that KEW will run in; choices are "local", "embedded", "remote", or "thin"	local
kew.url	The base URL of KEW services and pages	${application.url}/kew
plugin.dir	Directory to load plugins from if the Plugin Registry is enabled	
plugin.registry.enabled	If set to true, then the Plugin Registry will be enabled and any available plugins will be loaded (see Workflow Plugin Guide)	false
attachment.dir.location	When using the attachments system, this is the directory where attachments will be stored	
data.xml.loaded.location	Directory path where the XML Loader will store successfully loaded XML files	
data.xml.pending.location	Directory path where the XML Loader will look for files to ingest	
data.xml.pollIntervalSecs	Interval in seconds that the XML Loader will poll the pending directory for new XML files to load	
data.xml.problem.location	Directory path where the XML Loader will put XML files it failed to load	
datasource.platform	The fully qualified class name of an implementation of the org.kuali.rice.core.database.platform.Platform interface	
default.note.class	The fully qualified class name of the default implementation of org.kuali.rice.kew.notes.CustomNoteAttribute to use for the Notes system	org.kuali.rice.kew.notes.CustomNoteAttributeImpl
edl.config.loc	Location to load the EDocLite component configuration from	classpath:META-INF/EDLConfig.xml
embedded.server	Indicates if an embedded instance is supposed to behave like a standalone server. See additional notes below under embedded.server	false
Identity.useRemoteServices	Configuration parameter that governs whether a number of common identity services (user and	

Property	Description	Default
	group service) are exported or retrieved via the bus. If this flag is set to true then: 1. user and group service will NOT be published the bus, and 2. CoreResourceLoader will short-circuit the resource loader stack lookup and go directly to the bus to obtain these services, circumventing any beans that may be defined by local modules.	
initialDelaySecs	Delay in seconds after system starts up to begin the XML Loader polling	
rice.kew.enableKENNotification	Determines if KCB notifications should be sent for KEW events when Action Item events occur	true
rice.kew.struts.config.files	The struts-config.xml configuration file that the KEW portion of the Rice application will use	/kew/WEB-INF/struts-config.xml
workflow.documentsearch.base.url	The URL for the document search page	${workflow.url}/DocumentSearch.do?docFormKey=88888888&returnLocation=${application.url}/portal.do&hideReturnLink=true
xml.pipeline.lifecycle.enabled	If set to true, will poll a directory for new Rice configuration XML and ingest any new XML placed in that directory	false

The 'embedded.server' Parameter

If embedded.server parameter is enabled (set to true), then two additional features will be loaded when KEW is started:

1. XML Loader

2. Email Reminders

The XML Loader will poll a directory for XML files to ingest into the system (as configured by the data.xml.* properties).

The Email Reminders will handle sending Daily and Weekly batch emails for users that have their preferences set accordingly.

The 'datasource.platform' Parameter

KEW requires and uses the database platform implementation in order to function. These may be be implemented differently for each support database management system.

The current functional implementations of this platform are:

• org.kuali.rice.core.database.platform.OraclePlatform

• org.kuali.rice.core.database.platform.Oracle9iPlatform (deprecated and just an alias for the OraclePlatform)

• org.kuali.rice.core.database.platform.MySQLPlatform

Custom Servlet Filters

When running a Standalone Rice Server, you may want to implement your own filters for authentication purposes. The system comes with a special filter that will read filter definitions and mappings from the configuration system.

The Bootstrap Filter is a generic filter that is applied to all web requests, which then delegates to any filters and are setup through the default configuration. This mechanism allows registration of institution-specific filters without the necessity of modifying the web application configuration file (/WEB-INF/web.xml) within the standalone webapp.

Filter syntax is as follows:

```
<param name="filter.filter name.class">class name of filter</param>
```

filter name is an arbitrary name for your filter:

```
<param name="filter.myfilter.class">edu.institution.organization.MyFilter</param>
```

Any number of configuration parameters may be defined for a given filter as follows:

```
<param name="filter.filter name.filter param name">filter param value</param>
```

For example:

```
<param name="filter.myfilter.color">red</param>

<param name="filter.myfilter.shape">square</param>
```

For custom filters to be invoked, they must first be mapped to requests. That is done via the filter mapping parameter:

```
<param name="filtermapping.filter name.optional order index">path matching expression</param>
```

filter name is the name of your previously defined filter, *optional order index* is an optional integer used to specify the position of the filter in the invocation order, and *path matching expression* is a Servlet-specification-compatible url pattern.

```
<param name="filtermapping.myfilter.1">/special/path/</param>
```

If an order index is not specified, it is assumed to be 0. Filters with equivalent order are ordered arbitrarily with relation to each other (not in order of filter or mapping definition). A full example follows:

```
<param name="filter.myfilter.class">edu.institution.organization.MyFilter</param>

<param name="filter.myfilter.color">red</param>

<param name="filter.myfilter.shape">square</param>
<param name="filter.securityfilter.class">edu.institution.organization.SecurityFilter</param>

<param name="filter.securityfilter.secretKey">abracadabra</param>

<param name="filter.compressionfilter.class">edu.institution.organization.CompressionFilter</param>
<param name="filter.compressionfilter.compressLevel">5</param>
<param name="filtermapping.securityfilter.1">/secure/</param>

<param name="filtermapping.myfilter.2">/special/path/</param>
```

```
<param name="filtermapping.compressionfilter.3">/*</param>
```

Email Configuration

KEW can send emails to notify users about items in their Action List (depending on user preferences). Email in KEW uses the JavaMail library. In order to configure email, you will need to configure the appropriate JavaMail properties. A list of those properties can be found at the end of the page at the following url: http://java.sun.com/products/javamail/javadocs/javax/mail/package-summary.html

In addition to these standard JavaMail properties, you can also set the following optional properties to configure simple SMTP authentication.

Table 3.6. Optional Properties to Configure Simple SMTP Authentication

Property	Description	Examples/Values
mail.transport.protocol	The protocol used to sending mail	smtp
mail.smtp.host	This is the host name of the SMTP	smtp.secureserver.net
mail.smtp.username	The username used for access to the SMTP server	
mail.smtp.password	The password used for access to the SMTP server	

Of course, if the authentication required by your mail server is beyond the abilities of the above configuration, it is possible to override the *enEmailService* loaded by the KEW module and implement a custom email service.

In order for KEW to send out emails, several steps need to be done. In order to have KEW send out any emails, the "SEND_EMAIL_NOTIFICATION_IND" KNS System Parameter needs to be set to 'Y'. For emails to real people, the environment code must be set to 'prd'. If this is not set to 'prd', an email can still be sent out to a test address. This test address is set by the KNS System Parameter, "EMAIL_NOTIFICATION_TEST_ADDRESS". Emails sent in a test system will only be sent to the address specified by the EMAIL_NOTIFICATION_TEST_ADDRESS. The "from" address may also be set with a System Parameter. To do this, set the "FROM_ADDRESS" System Parameter to the email address you want the KEW emails sent from. If the FROM_ADDRESS parameter doesn't exist or isn't set, it will default to "admin@localhost".

Periodic Email Reminders

KEW can send emails on a nightly or weekly basis to remind users about items in their Action List (depending on user preferences). The following set of parameters configures whether the processes to send these reminders will run, and at what time(s) of day they will do so.

Table 3.7. Configuration Parameters for Email Reminders

Property	Description	Examples/Values
email.reminder.lifecycle.enabled	Enable periodic KEW reminder emails	true
dailyEmail.active	Enable daily reminder emails	true
dailyEmail.cronExpression	Configures the schedule on which the daily reminder emails are sent – see org.quartz.CronExpression, org.quartz.CronTrigger for information about the format for this parameter	0 0 1 * * ?
weeklyEmail.active	Enable weekly reminder emails	true
weeklyEmail.cronExpression	Configures the schedule on which the weekly reminder emails are	0 0 2 ? * 2

Property	Description	Examples/Values
	sent – see org.quartz.CronExpression, org.quartz.CronTrigger for information about the format for this parameter	

Workflow Preferences Configuration

Workflow users have the ability to update their preferences by going to the "User Preferences" page. The default values for many of these preferences can now be configured.

For example, institutions will commonly override the default action list email preference. By default it's set to "immediate," but it can be configured to "no", "daily", "weekly", or "immediate." The user will still be able to override the defaults on their User Preferences screen.

Here a list of workflow preferences that can be configured:

```
<!-- Default Option for Action List User Preferences. -->

<param name="userOptions.default.color">white</param>
<!-- email options: no, daily, weekly, immediate -->
<param name="userOptions.default.email" >immediate</param>
<param name="userOptions.default.notifyPrimary" >yes</param>
<param name="userOptions.default.notifySecondary" >no</param>
<param name="userOptions.default.openNewWindow" >yes</param>
<param name="userOptions.default.actionListSize" >10</param>
<param name="userOptions.default.refreshRate" >15</param>
<param name="userOptions.default.showActionRequired" >yes</param>

<param name="userOptions.default.showDateCreated" >yes</param>
<param name="userOptions.default.showDocumentType" >yes</param>
<param name="userOptions.default.showDocumentStatus" >yes</param>
<param name="userOptions.default.showInitiator" >yes</param>
<param name="userOptions.default.showDelegator" >yes</param>
<param name="userOptions.default.showTitle" >yes</param>
<param name="userOptions.default.showWorkgroupRequest" >yes</param>
<param name="userOptions.default.showClearFYI" >yes</param>
<param name="userOptions.default.showLastApprovedDate" >no</param>
<param name="userOptions.default.showCurrentNode" >no</param>
<param name="userOptions.default.useOutBox" >yes</param>
<!-- delegatorFilterOnActionList: "Secondary Delegators on Action List Page" or "Secondary Delegators only on
 Filter Page" -->
<param name="userOptions.default.delegatorFilterOnActionList" >Secondary Delegators on Action List Page</param>
<param name="userOptions.default.primaryDelegatorFilterOnActionList" >Primary Delegates on Action List Page</
param>
```

Outbox Configuration

The **Outbox** is a standard feature on the **Action List** and is visible to the user in the UI by default. When the Outbox is turned on, users can access it from the Outbox hyperlink at the top of the Action List.

The Outbox is implemented by heavily leveraging existing Action List code. When an **Action Item** is deleted from the Action Item table as the result of a user action, the item is stored in the **KEW_OUT_BOX_ITM_T** table, using the **org.kuali.rice.kew.actionitem.OutboxItemActionListExtension** object. This object is an extension of the **ActionItemActionListExtension**. The separate object exists to provide a bean for OJB mapping.

The Workflow Preferences determine if the Outbox is visible and functioning for each user. The preference is called **Use Outbox**. In addition, you can configure the Outbox at the KEW level using the parameter tag:

```
<param name="actionlist.outbox">true</param>
```

When the Outbox is set to *false*, the preference for individual users to configure the Outbox is turned off. By default, the Outbox is set to true at the KEW level. You can turn the Outbox off (to hide it from users) by setting the property below to *false*:

```
<param name="actionlist.outbox.default.preference.on">false</param>
```

This provides backwards compatibility with applications that used earlier versions of KEW.

Notes on the Outbox:

• Actions on saved documents are not displayed in the Outbox.

• The Outbox responds to all saved Filters and Action List Preferences.

• A unique instance of a document only exists in the Outbox. If a user has a document in the Outbox and that user takes action on the document, then the original instance of that document remains in the Outbox.

Implementing KEW at your institution

In addition to the previous discussion of KEW configuration, there are a few other aspects relevant to implementing KEW at your institution.

Bootstrap data

Because the operation of parts of KEW is dependent on a set of Document Types and Attributes being available within the system, there is some bootstrap XML that you will want to import. The easiest way to do this is to import the files in the following locations using the XML Ingester:

• kns/src/main/config/xml/RiceSampleAppWorkflowBootstrap.xml

• kew/src/main/config/bootstrap/edlstyle.xml

• kew/src/main/config/bootstrap/widgets.xml

These files include the following:

• Application constants: cluster-wide configuration settings

• Core document types and rules: a few primordial document types and rules are required for the system to function

• Default "eDocLite" styles: these are required if you wish to use eDocLite

• Default admin user and workgroup: these are depended upon (at the moment) by the core document types and rules, as well as referred to by the default application constants

Application constants you may want to change:

• Config.Application.AdminUserList: this should be set to a space-delimited set of administrative user names

• Workflow.AdminWorkgroup: this should be set to an institutional admin workgroup; if the default KEW workgroup service is used, this can be left to the default, WorkflowAdmin

- Config.Mailer.FromAddress: this should be changed to an address specific to your institution, e.g. kew@your-university.edu

- HelpDeskActionList.helpDeskActionListName: set to an workgroup at your institution

- ApplicationContext: set to the context path of the KEW application, if it differs from the environment default, e.g. "en-prod" instead of "en-prd"

In the core document types and rules config, you will need to change:

- superUserWorkgroupName, blanketApproveWorkgroupName, and exceptionWorkgroup: should be set to the administrative group at your institution. If you are using the default workgroup service, this can be left as WorkgroupAdmin

- ensure all docHandler elements, if they specify a URL, specify: "${base.url}/en-dev/Workgroup.do?methodToCall=docHandler", and ensure that the base.url config parameter is specified in your configuration (as mentioned above)

KEW Administration Guide

This guide provides information on administering a Kuali Enterprise Workflow (KEW) installation. Out of the box, KEW comes with a default setup that works well in development and test environments. However, when moving to a production environment, this setup requires adjustments. This document discusses basic administration as well as instructions for working with some of KEW's administration tools.

Configuration Overview

You configure KEW primarily through the **workflow.xml** file. Please see the KEW Configuration Parameters guide for more information on initial configuration of a KEW installation.

Application Constants

Application Constants are the configuration elements in KEW. Each constant is modifiable at system runtime; any changes take effect immediately in KEW. Application Constants are stored in a cluster-safe cache and propagated across all machines when change occurs. For more information about Application Constants, please refer to Application Constants.

Production Environments

When rolling KEW out into a production environment, there are application constants which you may need to change:

- **ActionList.sendEmailNotification** - This is usually set to false in test environments so emails aren't generated during testing. Usually, this is set this to true in a production environment to allow email notifications. You also need to ensure that your email service is configured properly to allow KEW to send notifications.

- **ApplicationContext** - In a production environment, this is usually something like en-prd. You must set this value correctly so that KEW's email notifications contain valid links.

- **Backdoor.ShowbackDoorLogin** - The backdoor login allows users to masquerade as other users for testing purposes. It is recommended that you set this value to false in a production environment.

- **RouteManagerPool.numWorkers** – The appropriate value for this depends on the capabilities of your production hardware. If it's set too high, KEW may use so much of the CPU that other applications running on the same machine are adversely impacted.

- **RouteManagerQueue.waitTime** - In test environments, users tend to be more sensitive to immediate feedback since they may be testing processes over the course of a couple minutes that, in practice, occur over a number of days. In test environments, it is recommended that you keep this value low. In a production environment, you can reasonably increase this value without affecting the speed at which documents are routed. This reduces thrashing on the route queue.

- **RouteQueue.isRoutingByIPNumber** - If you are running your production KEW system in a clustered environment, set this value to false. This allows processing of documents in the queue to be distributed across the entire cluster, which enhances routing performance and facilitates load balancing.

- **RouteQueue.maxRetryAttempts** - As with the *RouteManagerQueue.waitTime* constant, in a test environment it is important to find out as quickly as possible if a document is going to go into exception routing (usually indicating a problem in that document's routing setup). In a production environment, it may make sense to allow a longer period before a document goes into exception routing. This constant, in combination with the *RouteQueue.timeIncrement* constant, determines how long it takes a document to be put into exception routing.

- **RouteQueue.timeIncrement** - Increasing this value results in a longer time before a document goes into exception routing.

XML Ingestion

KEW relies on XML for data population and routing configuration. **XML Ingester** is available from the **Administrator** channel in the portal. This allows import of various KEW components from XML, such as DocumentTypes, RuleAttributes, Rules, Workgroups, and more.

Uploading an eDocLite form

To upload XML, go to *Ingester UI* and select the XML file that you want to import:

Figure 3.5. Ingester

After upload, notice the red arrow and the statement, *Ingested xml doc: <name of file>*:

Figure 3.6. Ingestion Complete
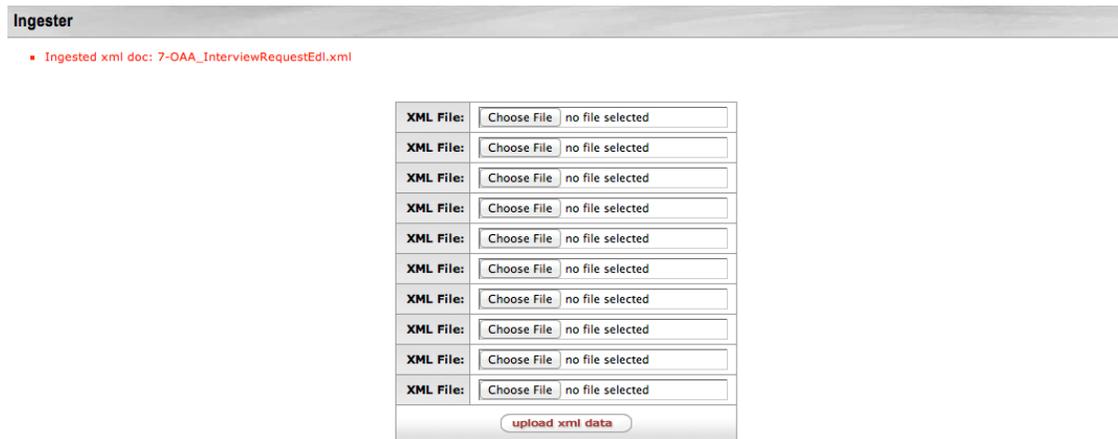
Message Queue Administration

The Message Queue is the main scheduling mechanism in KEW. You use it to schedule documents for asynchronous routing and to queue arbitrary units of work. When KEW places a document into exception routing, it may become stuck after a series of failed attempts. You can use the Route Queue UI to resolve this issue, as well as to fix new entries, if needed.

Examining the Message Queue

The main Message Queue screen:

Figure 3.7. Message Queue Screen
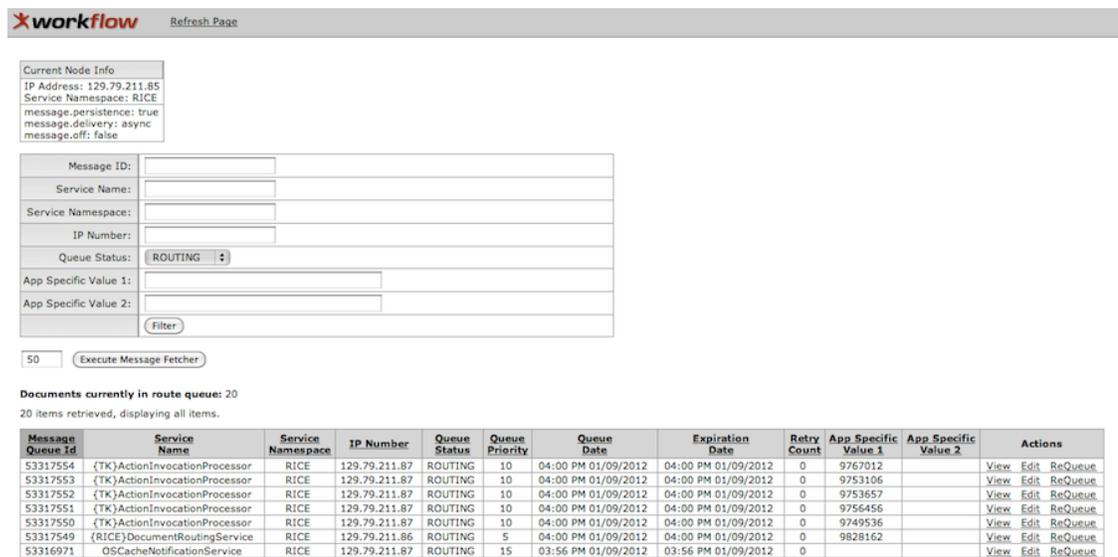
Examining this sample screen, we see there are 20 entries in the message queue, one on each row. The columns display information about each entry:

- **Message Queue Id** - The primary key of this route queue entry in the data store

- **Service Name**

- **Service Namespace**

- **IP Number** - The IP address of the machine on which the entry was created. In the environment pictured, we have three machines in our cluster. The IP number shows from which machine each entry was queued up.

- **Queue Status** – The entry can be in a state of QUEUED, ROUTING, or EXCEPTION:

  - A QUEUED entry is waiting for a worker thread to pick it up.

  - A ROUTING entry currently has a worker working on it.

  - An EXCEPTION entry has a problem and the route manager cannot access it. An administrator manually sets an EXCEPTION status to suspend a route queue entry until a problem can be diagnosed.

- **Queue Priority** - The priority of the entry in the queue, where entries with the lowest number are processed first

- **Queue Date** - The date that KEW should process this queue entry. If the queue checker runs and discovers the queue date for an entry is equal to or earlier than the current time, it processes that entry.

- **Expiration Date**

- **Retry Count** - The number of times KEW has attempted to process the entry

- **App Specific Value 1** - The parameters to be passed to the Route Queue processor such as document ID

- **App Specific Value 2** - The parameters to be passed to the Route Queue processor

- **Action** - The Edit link in the Action column allows you to edit the route queue entry.

Once a message entry has been successfully processed, it is deleted from the queue.

Diagnosing and Fixing Problems

Sometimes it is necessary to manually edit a route queue entry that is *halted inside of the queue*. This situation might happen when:

- KEW encounters an error trying to put the document into exception routing. This could occur if there is a database error or the document's *PostProcessor* throws an exception when it's notified of a status change

- KEW is improperly shut down in the middle of an entry being processed

- The database goes down while an entry is being processed

In all cases, the status of the entry is **ROUTING**, but there is no longer a worker thread processing the entry. Currently, KEW doesn't implement any auto-detection of failure cases. To put one of these entries in a state where it can be picked up by the route manager again, simply click the **Edit** link and set the entry's status back to **QUEUED**. Here's a screen shot of the **Route Queue Entry - Edit** screen:

Figure 3.8. Route Queue Entry Edit Screen

Use the Queue Status dropdown list to change the status of the entry. You may also want to set the Retry Count to zero to allow you to diagnose the problem before the document goes into exception routing.

KEW System Parameters

System Parameters Covered

Table 3.8. KEW System Parameters

Name	Value	Description
MAX_MEMBERS_PER_PAGE	20	The maximum number of role or group members to display at once on their documents. If the number is above this value, the document will switch into a paging mode with only this many rows displayed at a time.
PREFIXES	Ms;Mrs;Mr;Dr	
SUFFIXES	Jr;Sr;Mr;Md	
CHECK_ENCRYPTION_SERVICE_OVERRIDE_IND	Y	Flag for enabling/disabling (Y/N) the demonstration encryption check.
DATE_TO_STRING_FORMAT_FOR_FILE_NAME	yyyyMMdd	A single date format string that the DateTimeService will use to format dates to be used in a file name when DateTimeServiceImpl.toDateStringForFilename(Date) is called. For a more technical description of how characters in the parameter value will be interpreted, please consult the Java Documentation for java.text.SimpleDateFormat. Any changes will be applied when the application is restarted.
DATE_TO_STRING_FORMAT_FOR_USER_INTERFACE	MM/dd/yyyy	A single date format string that the DateTimeService will use to format a date to be displayed on a web page. For a more technical description of how characters in the parameter value will be interpreted, please consult the Java Documentation for java.text.SimpleDateFormat. Any changes will be applied when the application is restarted.
DEFAULT_COUNTRY	US	Used as the default country code when relating records that do not have a country code to records that do have a country code, e.g. validating a zip code where the country is not collected.
ENABLE_DIRECT_INQUIRIES_IND	Y	Flag for enabling/disabling direct inquiries on screens that are drawn by the nervous system (i.e. lookups and maintenance documents)
ENABLE_FIELD_LEVEL_HELP_IND	N	Indicates whether field level help links are enabled on lookup pages and documents.

Name	Value	Description
MAX_FILE_SIZE_DEFAULT_UPLOAD	5M	Maximum file upload size for the application. Must be an integer, optionally followed by "K", "M", or "G". Only used if no other upload limits are in effect.
SENSITIVE_DATA_PATTERNS	[0-9]{9};[0-9]{3}-[0-9]{2}-[0-9]{4}	A semi-colon delimited list of regular expressions that identify potentially sensitive data in strings. These patterns will be matched against notes, document explanations, and routing annotations.
STRING_TO_DATE_FORMATS	MM/dd/yy;MM-dd-yy;MMMM dd, yyyy;MMddyy	A semi-colon delimited list of strings representing date formats that the DateTimeService will use to parse dates when DateTimeServiceImpl.convertToSqlDate(String) or DateTimeServiceImpl.convertToDate(String) is called. Note that patterns will be applied in the order listed (and the first applicable one will be used). For a more technical description of how characters in the parameter value will be interpreted, please consult the Java Documentation for java.text.SimpleDateFormat. Any changes will be applied when the application is restarted.
STRING_TO_TIMESTAMP_FORMATS	MM/dd/yyyy hh:mm a	A semi-colon delimited list of strings representing date formats that the DateTimeService will use to parse date and times when DateTimeServiceImpl.convertToDateTime(String) or DateTimeServiceImpl.convertToSqlTimestamp(String) is called. Note that patterns will be applied in the order listed (and the first applicable one will be used). For a more technical description of how characters in the parameter value will be interpreted, please consult the Java Documentation for java.text.SimpleDateFormat. Any changes will be applied when the application is restarted.
TIMESTAMP_TO_STRING_FORMAT_FOR_FILE_NAME	yyyyMMdd-HH-mm-ss-S	A single date format string that the DateTimeService will use to format a date and time string to be used in a file name when DateTimeServiceImpl.toDateTimeStringForFilename(Date) is called. For a more technical description of how characters in the parameter value will be interpreted, please consult the Java Documentation for java.text.SimpleDateFormat. Any changes will be applied when the application is restarted.
TIMESTAMP_TO_STRING_FORMAT_FOR_USER_INTERFACE	MM/dd/yyyy hh:mm a	A single date format string that the DateTimeService will use to format a date and time to be displayed on a web page. For a more technical description of how characters in the parameter value will be interpreted, please consult the Java Documentation for java.text.SimpleDateFormat. Any changes will be applied when the application is restarted.
ACTIVE_FILE_TYPES	collectorInputFileType; procurementCardInputFileType; enterpriseFeederFileSetType; assetBarcodeInventoryInputFileType; customerLoadInputFileType	Batch file types that are active options for the file upload screen.
SCHEDULE_ADMIN_GROUP	KR-WKFLW:WorkflowAdmin	The workgroup to which a user must be assigned to modify batch jobs.
DEFAULT_CAN_PERFORM_ROUTE_REPORT_IND	N	If Y, the Route Report button will be displayed on the document actions bar if the document is using the default DocumentAuthorizerBase.getDocumentActionFlags to set the canPerformRouteReport property of the returned DocumentActionFlags instance.
EXCEPTION_GROUP	KR-WKFLW:WorkflowAdmin	The workgroup to which a user must be assigned to perform actions on documents in exception routing status.
MAX_FILE_SIZE_ATTACHMENT	5M	Maximum attachment uploads size for the application. Used by KualiDocumentFormBase. Must be an integer, optionally followed by "K", "M", or "G".

Name	Value	Description
PESSIMISTIC_LOCK_ADMIN_GROUP	KFS:KUALI_ROLE_SUPERVISOR	Workgroup which can perform admin deletion and lookup functions for Pessimistic Locks.
SEND_NOTE_WORKFLOW_NOTIFICATION_ACTIONS	K	Some documents provide the functionality to send notes to another user using a workflow FYI or acknowledge functionality. This parameter specifies the default action that will be used when sending notes. This parameter should be one of the following 2 values: "K" for acknowledge or "F" for "fyi". Depending on the notes and workflow service implementation, other values may be possible.
SESSION_TIMEOUT_WARNING_MESSAGE_TIME	5	The number of minutes before a session expires. That user should be warned when a document uses pessimistic locking.
SUPERVISOR_GROUP	KR-WKFLW:WorkflowAdmin	Workgroup which can perform almost any function within Kuali.
MULTIPLE_VALUE_RESULTS_EXPIRATION_SECONDS	86400	Lookup results may continue to be persisted in the DB long after they are needed. This parameter represents the maximum amount of time, in seconds, that the results will be allowed to persist in the DB before they are deleted from the DB.
MULTIPLE_VALUE_RESULTS_PER_PAGE	100	Maximum number of rows that will be displayed on a look-up results screen.
RESULTS_DEFAULT_MAX_COLUMN_LENGTH	70	If a maxLength attribute has not been set on a lookup result field in the data dictionary, then the result column's max length will be the value of this parameter. Set this parameter to 0 for an unlimited default length or a positive value (i.e. greater than 0) for a finite max length.
RESULTS_LIMIT	200	Maximum number of results returned in a look-up query.
MAX_AGE	86400	Pending attachments are attachments that do not yet have a permanent link with the associated Business Object (BO). These pending attachments are stored in the attachments.pending.directory (defined in the configuration service). If the BO is never persisted, then this attachment will become orphaned (i.e. not associated with any BO), but will remain in this directory. The PurgePendingAttachmentsStep batch step deletes these pending attachment files that are older than the value of this parameter. The unit of this value is seconds. Do not set this value too short, as this will cause problems attaching files to BOs.
NUMBER_OF_DAYS_SINCE_LAST_UPDATE	1	Determines the age of the session document records that the step will operate on, e.g. if this parameter is set to 4, the rows with a last update timestamp older that 4 days prior to when the job is running will be deleted.
CUTOFF_TIME	02:00:00:AM	Controls when the daily batch schedule should terminate. The scheduler service implementation compares the start time of the schedule job from quartz with this time on day after the schedule job started running.
CUTOFF_TIME_NEXT_DAY_IND	Y	Controls whether when the system is comparing the schedule start day & time with the scheduleStep_CUTOFF_TIME parameter, it considers the specified time to apply to the day after the schedule starts.
STATUS_CHECK_INTERVAL	30000	Time in milliseconds that the scheduleStep should wait between iterations.
ACTION_LIST_DOCUMENT_POPUP_IND	Y	Flag to specify if clicking on a Document ID from the Action List will load the Document in a new window.
ACTION_LIST_ROUTE_LOG_POPUP_IND	N	Flag to specify if clicking on a Route Log from the Action List will load the Route Log in a new window.
EMAIL_NOTIFICATION_TEST_ADDRESS		Default email address used for testing.

Name	Value	Description
HELP_DESK_NAME_GROUP	KR-WKFLW:WorkflowAdmin	The name of the group who has access to the "Help Desk" feature on the Action List.
PAGE_SIZE_THROTTLE		Throttles the number of results returned on all users Action Lists, regardless of their user preferences. This is intended to be used in a situation where excessively large Action Lists are causing performance issues.
SEND_EMAIL_NOTIFICATION_IND	N	Flag to determine whether or not to send email notification.
KIM_PRIORITY_ON_DOC_TYP_PERMS_IND	N	Flag for enabling/disabling document type permission checks to use KIM Permissions as priority over Document Type policies.
MAXIMUM_NODES_BEFORE_RUNAWAY		The maximum number of nodes the workflow engine will process before it determines the process is a runaway process. This is to prevent infinite "loops" in the workflow engine.
SHOW_ATTACHMENTS_IND	Y	Flag to specify whether or not a file upload box is displayed for KEW notes which allows for uploading of an attachment with the note.
SHOW_BACK_DOOR_LOGIN_IND	Y	Flag to show the backdoor login.
TARGET_FRAME_NAME	iframe_51148	Defines the target iframe name that the KEW internal portal uses for its menu links.
DOCUMENT_SEARCH_POPUP_IND	Y	Flag to specify if clicking on a Document ID from Document Search will load the Document in a new window.
DOCUMENT_SEARCH_ROUTE_LOG_POPUP_IND	N	Flag to specify if clicking on a Route Log from Document Search will load the Route Log in a new window.
FETCH_MORE_ITERATION_LIMIT		Limit of fetch more iteration for document searches.
RESULT_CAP		Maximum number of documents to return from a search.
DOCUMENT_TYPE_SEARCH_INSTRUCTION	Enter document type information below and click search.	Instructions for searching document types.
DEBUG_TRANSFORM_IND	N	Defines whether the debug transform is enabled for eDocLite.
USE_XSLTC_IND	N	Defines whether XSLTC is used for eDocLite.
IS_LAST_APPROVER_ACTIVATE_FIRST_IND		A flag to specify whether the WorkflowInfo.isLastApproverAtNode(...) API method attempts to active requests first, prior to execution.
REPLACE_INSTRUCTION	Enter the reviewer to replace.	Instructions for replacing a reviewer.
FROM_ADDRESS	rice.test@kulai.org	Default from email address for notifications. If not set, this value defaults to admin@localhost.
NOTE_CREATE_NEW_INSTRUCTION	Create or modify note information.	Instructions for creating a new note.
RESTRICT_DOCUMENT_TYPES		Comma separated list of Document Types to exclude from the Rule Quicklinks.
CUSTOM_DOCUMENT_TYPES		Defines custom Document Type processes to use for certain types of routing rules.
DELEGATE_LIMIT	20	Specifies that maximum number of delegation rules that will be displayed on a Rule inquiry before the screen shows a count of delegate rules and provides a link for the user to show them.
GENERATE_ACTION_REQUESTS_IND	Y	Flag to determine whether or not a change to a routing rule should be applied retroactively to existing documents.
ROUTE_LOG_POPUP_IND	F	Flag to specify if clicking on a Route Log from a Routing Rule inquiry will load the Route Log in a new window.
RULE_CACHE_REQUEUE_DELAY	5000	Amount of time after a rule change is made before the rule cache update message is sent.

Name	Value	Description
RULE_CREATE_NEW_INSTRUCTION	Please select a rule template and document type.	Instructions for creating a new rule.
RULE_LOCKING_ON_IND	Y	Defines whether rule locking it enabled.
RULE_SEARCH_INSTRUCTION	Use fields below to search for rules.	Instructions for the rule search.
RULE_TEMPLATE_CREATE_NEW_INSTRUCTION	Enter a rule template name and description. Please select all necessary rule attributes for this template.	Instructions for creating new rule templates.
RULE_TEMPLATE_SEARCH_INSTRUCTION	Use fields below to search for rule templates.	Instructions for the rule template search.
NOTIFY_EXCLUDED_USERS_IND		Defines a group name (in the format "namespace:name") which contains members who should never receive notification action requests from KEW. Notification requests in KEW are generated when someone disapproves or blanket approves are exist to notify other approvers that these actions have taken place. The most common use for this is in the case of "system" users who participate in workflow transactions. In these cases, since they aren't actual users who would be checking their action list, it doesn't make sense to send them requests since they won't ever be fulfilled.

Defining Workflow Processes Using Document Types

A Document Type is an object that brings workflow components together into a cohesive unit (routing configuration). One of its primary responsibilities is to define the routing path for a document. The routing path is the process definition for the document. It can consist of various types of nodes that perform certain actions, such as sending action requests to responsible parties, transmitting emails, or splitting the route path into parallel branches.

In addition to the routing path, it contains the Post Processor which receives event callbacks from the engine, the DocHandler which is the access point into the client application from the Action List and Access Control for certain actions. It can also define various policies that control how documents of that type are processed by the workflow engine.

This document has four parts:

1. A detailed explanation of the common fields in the **Document Type** XML definition

2. An example of each Document Type with a description of each field in it

3. Descriptions of the Document Type *policies*

4. A description of inheritance as applied to Document Types

There are some common attributes in every **Document Type**, but each **Document Type** can be customized to provide different functions.

• Document Types

• Document Type Policies

• Inheritance

Common Fields in Document Type XML Definition

Table 3.9. Common Fields in Document Type XML Definition

Field	Description
name	The name of the Document Type
parent	The parent Document Type of this Document Type. Each Child Document Type inherits the attributes of its parent Document Type.
description	The description of the Document Type; its primary responsibilities.
label	The label of the Document Type, how it's recognized
postProcessorName	The name of the postProcessor that takes charge of the routing for this Document Type
postprocessor	A component that gets called throughout the routing process and handles a set of standard events that all eDocs (electronic documents) go through.
authorizer	A component that gets called during the routing process to perform authorization checks. Applications can customize this component on a per-doctype basis.
superUserGroupName	The name of a workgroup whose members are the super users of this Document Type. Super users of this Document Type can execute a super user document search on this Document Type.
blanketApproveGroupName	The name of a workgroup whose members have the blanketapprove rights over this Document Type.
defaultExceptionGroupName	The name of the workgroup whose members receive an exception notice when a document of this Document Type encounters an exception in its routing.
docHandler	The DocHandler that handles the routing of this Document Type
active	A true or false indicator for the active status of this document
validApplicationStatuses	The set of valid application document statuses for this document type. If this optional configuration is set, the application document status will only allow the specified values to be set.
policies	The policies that apply to this Document Type
policy	The policy that applies to this Document Type. Use this when there is only one policy for the Document Type. **value**: A true or false indicator for whether the action for the policy will be taken
routingVersion	*This field exists only for backward compatibility with older versions of KEW.* Originally, KEW only supported sequential routing paths (as opposed to those with splits and joins). The KEW **getDocRouteLevel()** API returns an integer that represents the numerical step in the routing process. This number only has meaning for those documents that define sequential routing. • A document with a routingVersion of "1" will keep track of the route level number. • A document with a routingVersion of "2" (the default, unless explicitly defined in the Document Type configuration) will *NOT* keep track of the route level number and an exception will be thrown if code attempts to access that value. *New Document Type definitions do NOT need, and should NOT have, this flag defined.*
routePaths	The routing paths for this Document Type
routePath	The routing path for this Document Type. Use this field when there is just one routing path for this Document Type.
routeNode	A point or node on the routing path of this Document Type
routeModule	The most basic module; it allows KEW to generate Action Requests
start	The starting node of this Document Type during routing
requests	The requested next node in the routing of this Document Type
activationType	The activation type of the next node that is requested by this Document Type. There are two activation types: • **P**: Parallel: Multiple nodes in the routing process are activated at the same time

Field	Description
	• **S**: Serial or Sequential: The nodes in the routing process are activated one at a time
	• **R**: Priority-Parallel: The multiple nodes with the same priority are activated at the same time before moving to the next priority
ruleTemplate	The ruleTemplate that applies to the routing node in this Document Type
split	The routing path splits into branches and can continue on any of them at a split.
branch	One of the branches in the routing path.
join	The point in the routing path where the split branches join together.
process	There is a sub-process in the routing path; in other words, some nodes in the routing path will activate a sub-process.
simple	A new node in the routing path
	• **type**: The type of the new routing node
	• **value**: The value of the new routing node
	• **message**: The message associated with the new routing node
	• **level**: The routing level of the new routing node
	• **log**: The log name of the new routing node
dynamic	This changes the node to dynamic when it transitions to the next node in the routing path; therefore, the routing path is dynamic rather than static.

Document Types

Document Type Examples

BlanketApproveTest

```
<documentType>
    <name>BlanketApproveTest</name>
    <description>BlanketApproveTest</description>
    <label>BlanketApproveTest</label>
    <postProcessorName>org.kuali.rice.kew.postprocessor.DefaultPostProcessor</postProcessorName>
    <superUserGroupName namespace="KR-WKFLW" >TestWorkgroup</superUserGroupName>
    <blanketApproveGroupName namespace="KR-WKFLW">TestWorkgroup</blanketApproveGroupName>
    <defaultExceptionGroupName namespace="KR-WKFLW"> TestWorkgroup</defaultExceptionGroupName>
    <docHandler>_blank</docHandler>
    <active>true</active>
    <policies>
        <policy>
            <name>DEFAULT_APPROVE</name>
            <value>false</value>
        </policy>
    </policies>

</documentType>
```

- **name**: This is the Document Type for Blanket Approve Test.

- **description**: This Document Type is used to test the Blanket Approve function.

- **label**: This Document Type is recognized as the BlanketApproveTest type.

- **postProcessorName**: The postProcessor for this Document Type is org.kuali.rice.kew.postprocessor.DefaultPostProcessor.

- **superUserGroupName**: The super users for this Document Type are members of the TestWorkgroup.

- **blanketApproveGroupName**: The members of the TestWorkgroup have blanketApproval right on this type of document.

- **defaultExceptionGroupName**: The members of the TestWorkgroup will receive an exception notice for documents of this Document Type.

- **docHandler**: The Doc Handler for this type of document is _blank.

- **active**: This Document Type is currently Active. In other words, it is in use.

- **Policies** for this Document Type contains two policies: The DEFAULT_APPROVE policy is set false by default. In other words, the default approve action on this type of document is NOT to approve it.

BlanketApproveSequentialTest

```
<documentType>
    <name>BlanketApproveSequentialTest</name>
    <parent>BlanketApproveTest</parent>
    <description>BlanketApproveSequentialTest</description>
    <label>BlanketApproveSequentialTest</label>
    <postProcessorName>org.kuali.rice.kew.postprocessor.DefaultPostProcessor</postProcessorName>
    <superUserGroupName namespace="KR-WKFLW">WorkflowAdmin</superUserGroupName>
    <blanketApproveGroupName namespace="KR-WKFLW">TestWorkgroup</blanketApproveGroupName>
    <defaultExceptionGroupName namespace="KR-WKFLW">WorkflowAdmin</defaultExceptionGroupName>

    <docHandler>_blank</docHandler>
    <active>true</active>
    <routePaths>
        <routePath>
            <start name="AdHoc" nextNode="WorkflowDocument" />
            <requests name="WorkflowDocument" nextNode="WorkflowDocument2" />
            <requests name="WorkflowDocument2" nextNode="Acknowledge1" />
            <requests name="Acknowledge1" nextNode="Acknowledge2" />
            <requests name="Acknowledge2" />
        </routePath>
    </routePaths>
    <routeNodes>
        <start name="AdHoc">
            <activationType>P</activationType>
        </start>
        <requests name="WorkflowDocument">
            <activationType>P</activationType>
            <ruleTemplate>WorkflowDocumentTemplate</ruleTemplate>
        </requests>
        <requests name="WorkflowDocument2">
            <activationType>P</activationType>
            <ruleTemplate>WorkflowDocument2Template</ruleTemplate>
        </requests>
        <requests name="Acknowledge1">
            <activationType>P</activationType>
            <ruleTemplate>Ack1Template</ruleTemplate>
        </requests>
        <requests name="Acknowledge2">
            <activationType>P</activationType>
            <ruleTemplate>Ack2Template</ruleTemplate>
        </requests>
    </routeNodes>
</documentType>
```

- **name**: This is the Document Type for Blanket Approve Sequential Test. There is a sequence of routing nodes, and no routing node can be skipped.

- **parent**: The parent Document Type is BlanketApproveTest. This Document Type inherits the policies that BlanketApproveTest has.

- **description**: This Document Type is used to test the Blanket Approve Sequential function.

- **label**: This Document Type is recognized as the blanketApproveSequentialTest type.

- **postProcessorName**: The postProcessor for this Document Type is org.kuali.rice.kew.postprocessor.DefaultPostProcessor.

- **superUserGroupName**: The super users for this Document Type are members of the WorkflowAdmin.

- **blanketApproveGroupName**: The members of the TestWorkgroup have blanketApprove right on this type of document.

- **defaultExceptionGroupName**: The members of the WorkflowAdmin will receive an exception notice for documents of this Document Type.

- **docHandler**: The Doc Handler for this type of document is _blank.

- **active**: This Document Type is currently active. In other words, it is in use.

- **routePath**: The routing path for this Document Type is: AdHoc -> WorkflowDocument -> WorkflowDocument2 -> Acknowledge1 -> Acknowledge2.

- **routeNode**: Based on the routePath, there are five nodes in the routing of this Document Type:

  - The starting node for this Document Type is AdHoc. On the initiation of a document of this type, the postProcessor in Kuali Enterprise Workflow (KEW) activates the node, AdHoc.

  - The next node in the routing for this Document Type is WorkflowDocument. On request, the node is activated and applies the rules in rule template, WorkflowDocumentTemplate.

  - The next node in the routing for this Document Type is WorkflowDocument2. On request, the node is activated and applies the rules in rule template, WorkflowDocument2Template.

  - The next node in the routing for this Document Type is Acknowledge1. On request, the node is activated and applies the rules in rule template, Ack1Template.

  - The next node in the routing for this Document Type is Acknowledge2. On request, the node is activated and applies the rules in rule template, Ack2Template.

Figure 3.9. BlanketApproveSequentialTest Workflow
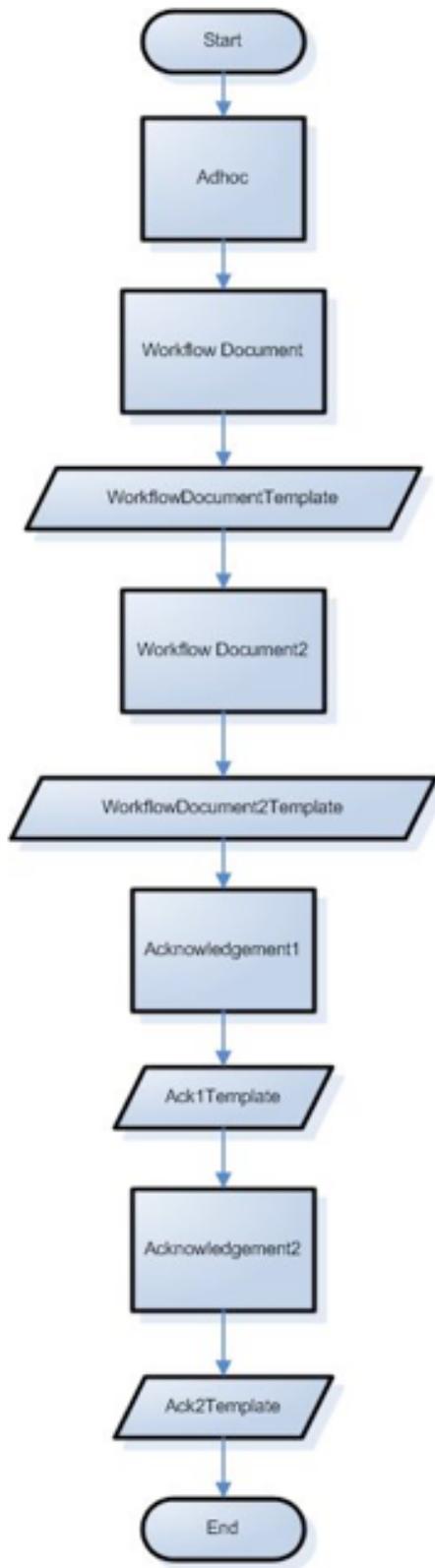
BlanketApproveParallelTest

```
<documentType>
    <name>BlanketApproveParallelTest</name>
    <parent>BlanketApproveTest</parent>
    <description>BlanketApproveParallelTest</description>
    <label>BlanketApproveParallelTest</label>
    <postProcessorName>org.kuali.rice.kew.postprocessor.DefaultPostProcessor</postProcessorName>
    <superUserGroupName namespace="KR-WKFLW" >TestWorkgroup</superUserGroupName>
    <blanketApproveGroupName namespace="KR-WKFLW">TestWorkgroup</blanketApproveGroupName>
    <defaultExceptionGroupName namespace="KR-WKFLW"> TestWorkgroup</defaultExceptionGroupName>
    <docHandler>_blank</docHandler>
    <active>true</active>
    <routePaths>
        <routePath>
            <start name="AdHoc" nextNode="WorkflowDocument" />
            <requests name="WorkflowDocument" nextNode="Split" />
            <split name="Split" nextNode="WorkflowDocumentFinal">
                <branch name="B1">
                    <requests name="WorkflowDocument2-B1" nextNode="WorkflowDocument3-B1" />
                    <requests name="WorkflowDocument3-B1" nextNode="Join" />
                </branch>
                <branch name="B2">
                    <requests name="WorkflowDocument3-B2" nextNode="WorkflowDocument2-B2" />
                    <requests name="WorkflowDocument2-B2" nextNode="Join" />
                </branch>
                <branch name="B3">
                    <requests name="WorkflowDocument4-B3" nextNode="Join" />
                </branch>
                <join name="Join" />
            </split>
            <requests name="WorkflowDocumentFinal" nextNode="Acknowledge1" />
            <requests name="Acknowledge1" nextNode="Acknowledge2" />
            <requests name="Acknowledge2" />
        </routePath>
    </routePaths>
    <routeNodes>
        <start name="AdHoc">
            <activationType>P</activationType>
        </start>
        <requests name="WorkflowDocument">
            <activationType>P</activationType>
            <ruleTemplate>WorkflowDocumentTemplate</ruleTemplate>
        </requests>
        <split name="Split" />
        <requests name="WorkflowDocument2-B1">
            <activationType>P</activationType>
            <ruleTemplate>WorkflowDocument2Template</ruleTemplate>
        </requests>
        <requests name="WorkflowDocument2-B2">
            <activationType>P</activationType>
            <ruleTemplate>WorkflowDocument2Template</ruleTemplate>
        </requests>
        <requests name="WorkflowDocument3-B1">
            <activationType>P</activationType>
            <ruleTemplate>WorkflowDocument3Template</ruleTemplate>
        </requests>
        <requests name="WorkflowDocument3-B2">
            <activationType>P</activationType>
            <ruleTemplate>WorkflowDocument3Template</ruleTemplate>
        </requests>
        <requests name="WorkflowDocument4-B3">
            <activationType>P</activationType>
            <ruleTemplate>WorkflowDocument4Template</ruleTemplate>
        </requests>
        <join name="Join" />
        <requests name="WorkflowDocumentFinal">
            <activationType>P</activationType>
            <ruleTemplate>WorkflowDocumentFinalTemplate</ruleTemplate>
        </requests>
        <requests name="Acknowledge1">
            <activationType>P</activationType>
            <ruleTemplate>Ack1Template</ruleTemplate>
        </requests>
        <requests name="Acknowledge2">
            <activationType>P</activationType>
            <ruleTemplate>Ack2Template</ruleTemplate>
```

```
        </requests>
      </routeNodes>
</documentType>
```

- **name**: This is the Document Type for Blanket Approve Parallel Test. At some point in the routing, the route path may split and a node can be skipped if another parallel node takes action on the document.

- **Parent**: The parent Document Type is BlanketApproveTest. This Document Type inherits the routing that exists for BlanketApproveTest.

- **description**: This Document Type is used to test the Blanket Approve Parallel function.

- **label**: This Document Type is recognized as the blanketApproveParallelTest type.

- **postProcessorName**: The postProcessor for this Document Type is *org.kuali.rice.kew.postprocessor.DefaultPostProcessor*.

- **superUserGroupName**: The super users for this Document Type are members of the TestWorkgroup.

- **blanketApproveGroupName**: The members of the TestWorkgroup have blanketApproval right on this type of document.

- **defaultExceptionGroupName**: The members of the TestWorkgroup will receive an exception notice for documents of this Document Type.

- **docHandler**: The Doc Handler for this type of document is *_blank*.

- **active**: This Document Type is currently active. In other words, it is in use.

- **routePath**: The routing path for this Document Type is: AdHoc -> WorkflowDocument -> split -> B1\B2\B3 -> Join -> WorkflowDocumentFinal -> Acknowledge1 -> Acknowledge2.

- **routeNode**: Based on the routePath, there are six nodes in the routing of this Document Type:

  - The starting node for this Document Type is **AdHoc**. On the initiation of a document of this type, the postProcessor in KEW activates the node, **AdHoc**.

  - The next node in the routing for this Document Type is **WorkflowDocument**. On request, the node is activated and applies the rules in rule template, **WorkflowDocumentTemplate**. Then, the routing path splits into three branches for the next node.

    - One branch is B1. On request, the node **WorkflowDocument2-B1** is activated and applies the **WorkflowDocument2Template**. The next node in this branch is **WorkflowDocument3-B1**. On request, the node is activated and applies the **WorkflowDocument3Template**.

    - One branch is B2. On request, the node **WorkflowDocument3-B2** is activated and applies the **WorkflowDocument3Template**. The next node in this branch is **WorkflowDocument2-B2**. On request, the node is activated and applies the **WorkflowDocument2Template**.

    - One branch is B3. On request, the node **WorkflowDocument4-B3** is activated and applies the **WorkflowDocument4Template**.

  - Then, the routing path joins and the route merges back together into one route.

  - The next node in the routing for this Document Type is **WorkflowDocumentFinal**. On request, the node is activated and applies the rules in rule template, **WorkflowDocumentFinalTemplate**.

  - The next node in the routing for this Document Type is **Acknowledge1**. On request, the node is activated and applies the rules in rule template, **Ack1Template**.

- The next node in the routing for this Document Type is **Acknowledge2**. On request, the node is activated and applies the rules in rule template, **Ack2Template**.

Figure 3.10. BlanketApproveParallelTest Workflow

NotificationTest

```
<documentType>
    <name>NotificationTest</name>
    <description>NotificationTest</description>
    <label>NotificationTest</label>
    <postProcessorName>org.kuali.rice.kew.postprocessor.DefaultPostProcessor</postProcessorName>
    <superUserGroupName namespace="KR-WKFLW" >TestWorkgroup</superUserGroupName>
    <blanketApproveGroupName namespace="KR-WKFLW">TestWorkgroup</blanketApproveGroupName>
    <defaultExceptionGroupName namespace="KR-WKFLW"> TestWorkgroup</defaultExceptionGroupName>
    <docHandler>_blank</docHandler>
    <active>true</active>
    <routePaths>
        <routePath>
            <start name="AdHoc" nextNode="NotifyFirst" />
            <requests name="NotifyFirst" nextNode="Split" />
            <split name="Split" nextNode="NotifyFinal">
                <branch name="LeftBranch">
                    <requests name="NotifyLeftBranch" nextNode="Join" />
                </branch>
                <branch name="RightBranch">
                    <requests name="NotifyRightBranch" nextNode="Join" />
                </branch>
                <join name="Join" />
            </split>
            <requests name="NotifyFinal" />
        </routePath>
    </routePaths>
    <routeNodes>
        <start name="AdHoc">
            <activationType>P</activationType>
        </start>
        <requests name="NotifyFirst">
            <activationType>P</activationType>
            <ruleTemplate>NotifyFirstTemplate</ruleTemplate>
        </requests>
        <split name="Split" />
        <requests name="NotifyLeftBranch">
            <activationType>P</activationType>
            <ruleTemplate>NotifyLeftBranchTemplate</ruleTemplate>
        </requests>
        <requests name="NotifyRightBranch">
            <activationType>P</activationType>
            <ruleTemplate>NotifyRightBranchTemplate</ruleTemplate>
        </requests>
        <join name="Join" />
        <requests name="NotifyFinal">
            <activationType>P</activationType>
            <ruleTemplate>NotifyFinalTemplate</ruleTemplate>
        </requests>
    </routeNodes>
</documentType>
```

- **name**: This is the Document Type for Notification Test. At some point in the routing, the route path may split, and a node can be skipped if another notification node takes action on the document.

- **description**: This Document Type is used to test the notification function.

- **label**: This Document Type is recognized as the NotificationTest type.

- **postProcessorName**: The postProcessor for this Document Type is **org.kuali.rice.kew.postprocessor.DefaultPostProcessor**.

- **superUserGroupName**: The super users for this Document Type are members of the TestWorkgroup.

- **blanketApproveGroupName**: The members of the TestWorkgroup have blanketApproval right on this type of document.

- **defaultExceptionGroupName**: The members of the TestWorkgroup will receive an exception notice for documents of this Document Type.

- **docHandler**: The Doc Handler for this type of document is **_blank**.

- **active**: This Document Type is currently active. In other words, it is in use.

- **routePath**: The routing path for this Document Type is: AdHoc -> NotifyFirst -> split -> LeftBranch \RightBranch -> Join -> NotifyFinal.

- **routeNode**: Based on the routePath, there are four nodes in the routing of this Document Type:

    - o The starting node for this Document Type is **AdHoc**. On the initiation of a document of this type, the postProcessor in KEW activates the node, **AdHoc**.

    - The next node in the routing for this Document Type is **NotifyFirst**. On request, the node is activated and applies the rules in rule template, **NotifyFirstTemplate**. Then the routing path splits into two branches for the next node.

        - One branch is **LeftBranch**. On request, the node is activated and applies the **NotifyLeftBranchTemplate**.

        - One branch is **RightBranch**. On request, the node is activated and applies the **NotifyRightBranchTemplate**.

    - Then the routing path joins together again.

    - The next node in the routing for this Document Type is **NotifyFinal**. On request, the node is activated and applies the rules in rule template, **NotifyFinalTemplate**

Figure 3.11. NotificationTest Workflow

NotificationTestChild

```
<documentType>
    <name>NotificationTestChild</name>
    <parent>NotificationTest</parent>
    <description>NotificationTest</description>
    <label>NotificationTest</label>
    <postProcessorName>org.kuali.rice.kew.postprocessor.DefaultPostProcessor</postProcessorName>
    <superUserGroupName namespace="KR-WKFLW" >TestWorkgroup</superUserGroupName>
    <blanketApproveGroupName namespace="KR-WKFLW">TestWorkgroup</blanketApproveGroupName>
    <defaultExceptionGroupName namespace="KR-WKFLW"> TestWorkgroup</defaultExceptionGroupName>
    <docHandler>_blank</docHandler>
    <active>true</active>
    <policies>
        <policy>
            <name>SEND_NOTIFICATION_ON_SU_APPROVE</name>
            <value>true</value>
        </policy>
```

```
        </policies>
</documentType>
```

- **name**: This is the Document Type for Notification Test Child.

- **Parent**: The parent Document Type is NotificationTest. This Document Type inherits the routing that NotificationTest has.

- **description**: This Document Type is used to test the Notification function.

- **label**: This Document Type is recognized as the NotificationTest type.

- **postProcessorName**: The postProcessor for this Document Type is **org.kuali.rice.kew.postprocessor.DefaultPostProcessor**.

- **superUserGroupName**: The super users for this Document Type are members of the TestWorkgroup.

- **blanketApproveGroupName**: The members of the TestWorkgroup have blanketApproval right on this type of document.

- **defaultExceptionGroupName**: The members of the TestWorkgroup will receive an exception notice for documents of this Document Type.

- **docHandler**: The Doc Handler for this type of document is **_blank**.

- **active**: This Document Type is currently active. In other words, it is in use.

- **Policy**: There is only one policy that applies to this Document Type: SEND_NOTIFICATION_ON_SU_APPROVE. This policy currently applies to this Document Type. In other words, a notification will be sent to the designated two users when a SuperUser approves a document of this type.

BlanketApproveMandatoryNodeTest

```
<documentType>
    <name>BlanketApproveMandatoryNodeTest</name>
    <parent>BlanketApproveTest</parent>
    <description>BlanketApproveMandatoryNodeTest</description>
    <label>BlanketApproveMandatoryNodeTest</label>
    <postProcessorName>org.kuali.rice.kew.postprocessor.DefaultPostProcessor</postProcessorName>
    <superUserGroupName namespace="KR-WKFLW" >TestWorkgroup</superUserGroupName>
    <blanketApproveGroupName namespace="KR-WKFLW">TestWorkgroup</blanketApproveGroupName>
    <defaultExceptionGroupName namespace="KR-WKFLW"> TestWorkgroup</defaultExceptionGroupName>
    <docHandler>_blank</docHandler>
    <active>true</active>
    <routePaths>
        <routePath>
            <start name="AdHoc" nextNode="WorkflowDocument" />
            <requests name="WorkflowDocument" nextNode="WorkflowDocument2" />
            <requests name="WorkflowDocument2" nextNode="Acknowledge1" />
            <requests name="Acknowledge1" nextNode="Acknowledge2" />
            <requests name="Acknowledge2" />
        </routePath>
    </routePaths>
    <routeNodes>
        <start name="AdHoc">
            <activationType>P</activationType>
        </start>
        <requests name="WorkflowDocument">
            <activationType>P</activationType>
            <ruleTemplate>WorkflowDocumentTemplate</ruleTemplate>
            <mandatoryRoute>true</mandatoryRoute>
        </requests>
        <requests name="WorkflowDocument2">
            <activationType>P</activationType>
```

```
                    <ruleTemplate>WorkflowDocument2Template</ruleTemplate>
                    <mandatoryRoute>true</mandatoryRoute>
                    <finalApproval>true</finalApproval>
            </requests>
            <requests name="Acknowledge1">
                    <activationType>P</activationType>
                    <ruleTemplate>Ack1Template</ruleTemplate>
            </requests>
            <requests name="Acknowledge2">
                    <activationType>P</activationType>
                    <ruleTemplate>Ack2Template</ruleTemplate>
            </requests>
        </routeNodes>
</documentType>
```
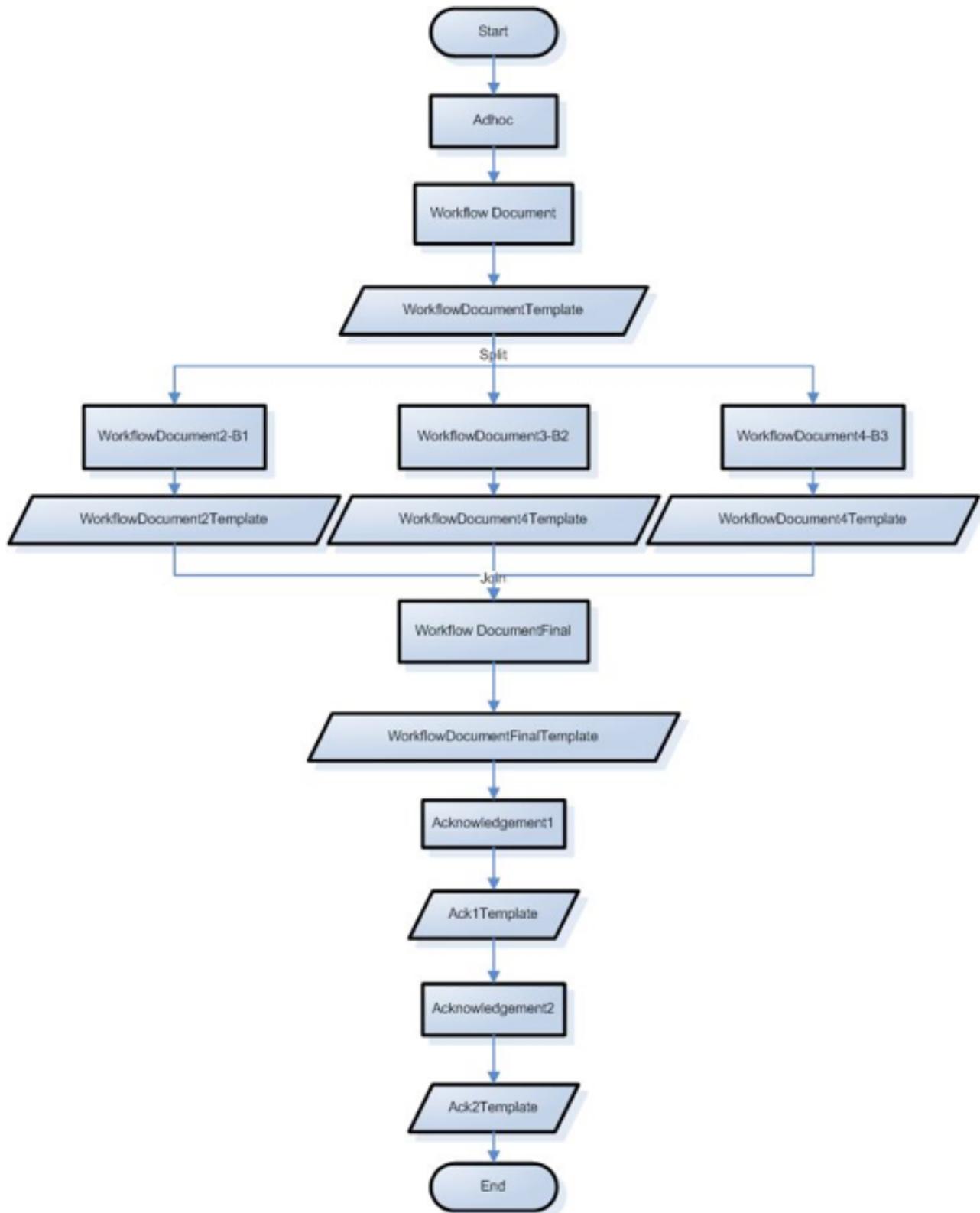
- **name**: This is the Document Type for Blanket Approve Mandatory Node Test.

- **Parent**: The parent Document Type is BlanketApproveTest. This Document Type inherits the policies that NotificationTest has.

- **description**: This Document Type is used to test the Blanket Approve Mandatory Node.

- **label**: This Document Type is recognized as the BlanketApproveMandatoryNodeTest type.

- **postProcessorName**: The postProcessor for this Document Type is **org.kuali.rice.kew.postprocessor.DefaultPostProcessor**.

- **superUserGroupName**: The super users for this Document Type are members of the TestWorkgroup.

- **blanketApproveGroupName**: The members of the TestWorkgroup have blanketApproval right on this type of document.

- **defaultExceptionGroupName**: The members of the TestWorkgroup will receive an exception notice for documents of this Document Type.

- **docHandler**: The Doc Handler for this type of document is _blank.

- **active**: This Document Type is currently active. In other words, it is in use.

- **routePath**: The routing path for this Document Type is: AdHoc -> WorkflowDocument -> WorkflowDocument2 -> Acknowledge1 -> Acknowledge2.

- **routeNode**: Based on the routePath, there are five nodes in the routing of this Document Type:

  - The starting node for this Document Type is **AdHoc**. On the initiation of a document of this type, the postProcessor in KEW activates the node **AdHoc**.

  - The next node in the routing for this Document Type is **WorkflowDocument**. On request, the node is activated, applies the rules in rule template, **WorkflowDocumentTemplate**, and sets the mandatory route as true. In other words, the document must route through this node.

  - The next node in the routing for this Document Type is **WorkflowDocument2**. On request, the node is activated, applies the rules in rule template, **WorkflowDocument2Template**, and sets the mandatory route as **true**. In other words, the document must route through this node.

  - The next node in the routing for this Document Type is **Acknowledge1**. On request, the node is activated and applies the rules in rule template, **Ack1Template**.

  - The next node in the routing for this Document Type is **Acknowledge2**. On request, the node is activated and applies the rules in rule template, **Ack2Template**.

Figure 3.12. Blanket Approve Mandatory Test
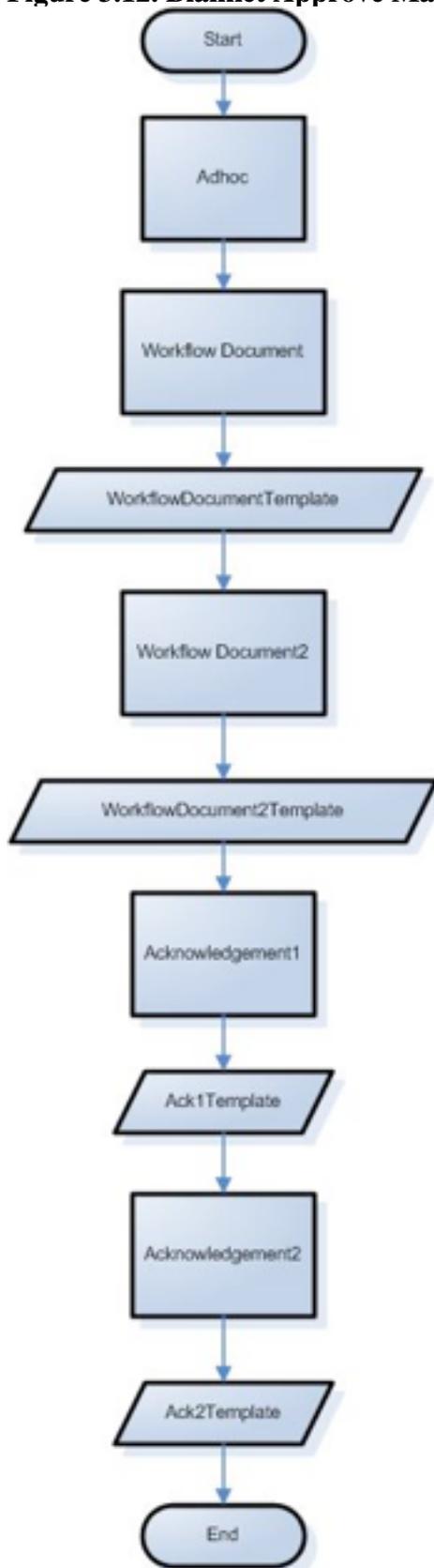
SaveActionEventTest

```
<documentType>
    <name>SaveActionEventTest</name>
    <description>SaveActionEventTest</description>
    <label>SaveActionEventTest</label>
 <postProcessorName>org.kuali.rice.kew.postprocessor.DefaultPostProcessor</postProcessorName>
    <superUserGroupName namespace="KR-WKFLW" >TestWorkgroup</superUserGroupName>
    <blanketApproveGroupName namespace="KR-WKFLW">TestWorkgroup</blanketApproveGroupName>
    <defaultExceptionGroupName namespace="KR-WKFLW"> TestWorkgroup</defaultExceptionGroupName>
    <docHandler>_blank</docHandler>
    <active>true</active>
    <policies>
        <policy>
            <name>DEFAULT_APPROVE</name>
            <value>false</value>
        </policy>
    </policies>
    <routePaths>
        <routePath>
            <start name="AdHoc" nextNode="WorkflowDocument" />
            <requests name="WorkflowDocument" />
        </routePath>
    </routePaths>
    <routeNodes>
        <start name="AdHoc">
            <activationType>P</activationType>
        </start>
        <requests name="WorkflowDocument">
            <activationType>P</activationType>
            <ruleTemplate>WorkflowDocumentTemplate</ruleTemplate>
        </requests>
    </routeNodes>
</documentType>
```
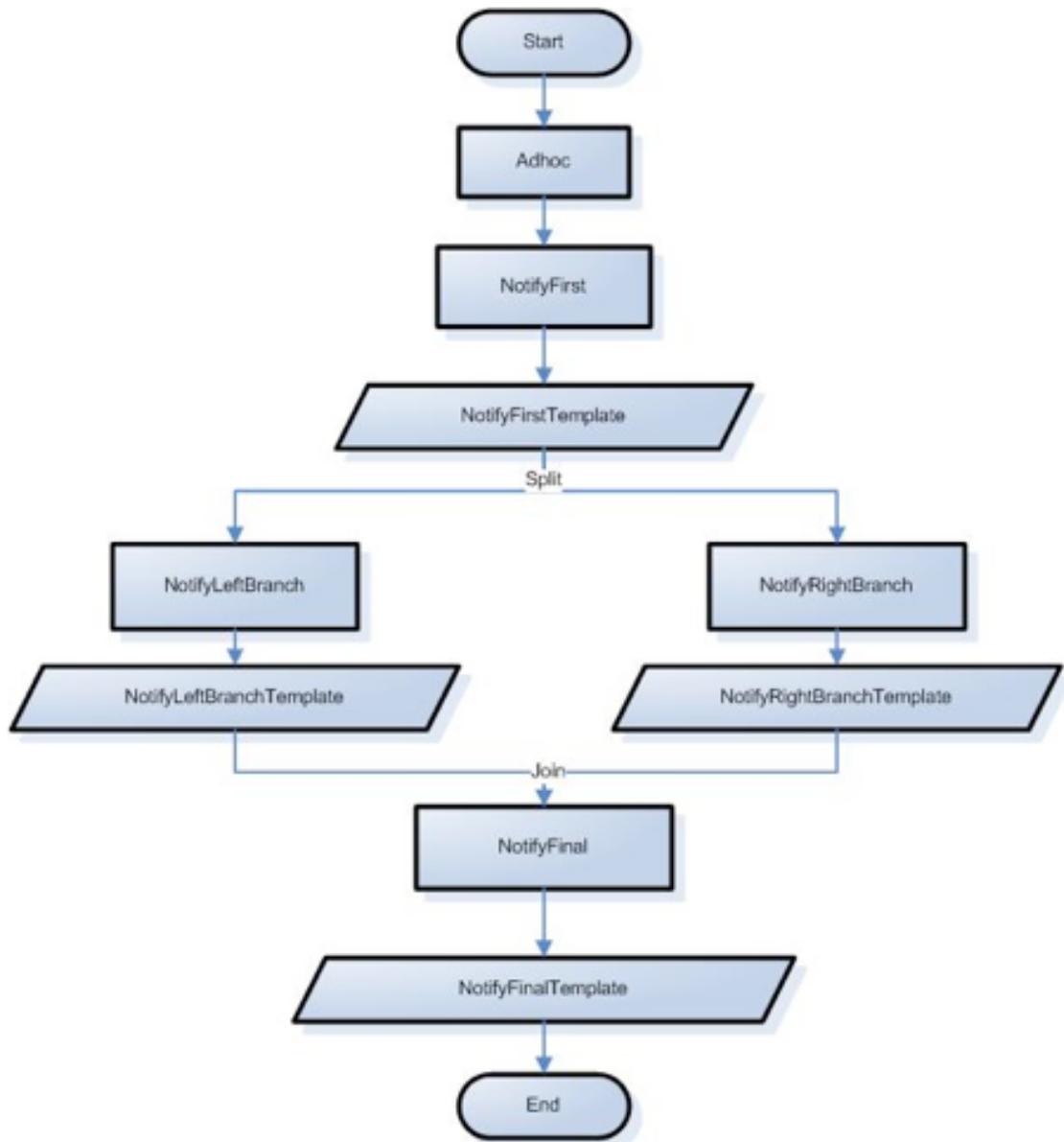
- **name**: This is the Document Type for Save Action Event Test.

- **description**: This Document Type is used to test the Blanket Approve Mandatory Node.

- **label**: This Document Type is recognized as the SaveActionEventTest type.

- **postProcessorName**: The postProcessor for this Document Type is **org.kuali.rice.kew.postprocessor.DefaultPostProcessor**.

- **superUserGroupName**: The super users for this Document Type are members of the TestWorkgroup.

- **blanketApproveGroupName**: The members of the TestWorkgroup have blanketApproval right on this type of document.

- **defaultExceptionGroupName**: The members of the TestWorkgroup will receive an exception notice for documents of this Document Type.

- **docHandler**: The Doc Handler for this type of document is **_blank**.

- **active**: This Document Type is currently active. In other words, it is in use.

- **Policies** for this Document Type: The DEFAULT_APPROVE policy is set **false** by default. In other words, the default approve action on this type of document is NOT to approve it.

- **routePath**: The routing path for this Document Type is: AdHoc -> WorkflowDocument.

- **routeNode**: Based on the routePath, there are two nodes in the routing of this Document Type:

  - The starting node for this Document Type is **AdHoc**. On the initiation of a document of this type, the postProcessor in KEW activates the node **AdHoc**.

- The next node in the routing for this Document Type is **WorkflowDocument**. On request, the node is activated and applies the rules in rule template **WorkflowDocumentTemplate**.

Figure 3.13. Save Action Event Test
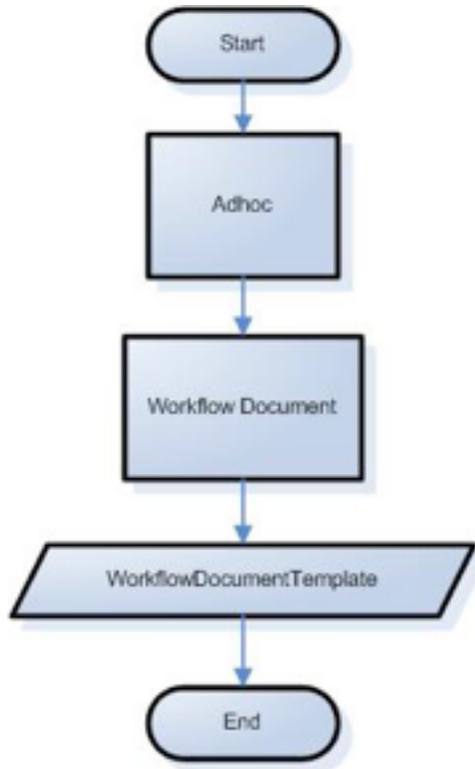
SaveActionEventTestNonInitiator

```
<documentType>
    <name>SaveActionEventTestNonInitiator</name>
    <description>SaveActionEventTest With No Initiator Only Save Required</description>
    <label>SaveActionEventTestNonInitiator</label>
    <postProcessorName>org.kuali.rice.kew.postprocessor.DefaultPostProcessor</postProcessorName>
    <superUserGroupName namespace="KR-WKFLW" >TestWorkgroup</superUserGroupName>
    <blanketApproveGroupName namespace="KR-WKFLW">TestWorkgroup</blanketApproveGroupName>
    <defaultExceptionGroupName namespace="KR-WKFLW"> TestWorkgroup</defaultExceptionGroupName>
    <docHandler>_blank</docHandler>
    <active>true</active>
    <policies>
        <policy>
            <name>DEFAULT_APPROVE</name>
            <value>false</value>
        </policy>
        <policy>
            <name>INITIATOR_MUST_SAVE</name>
            <value>false</value>
        </policy>
    </policies>
    <routePaths>
        <routePath>
            <start name="AdHoc" nextNode="WorkflowDocument" />
            <requests name="WorkflowDocument" />
        </routePath>
    </routePaths>
    <routeNodes>
        <start name="AdHoc">
            <activationType>P</activationType>
```

```
        </start>
        <requests name="WorkflowDocument">
            <activationType>P</activationType>
            <ruleTemplate>WorkflowDocumentTemplate</ruleTemplate>
        </requests>
    </routeNodes>
</documentType>
```

- **name**: This is the Document Type for Save Action Event Test Non Initiator.

- **description**: This Document Type is used to test the saving of an action event by non-initiator.

- **label**: This Document Type is recognized as the SaveActionEventTestNonInitiator type.

- **postProcessorName**: The postProcessor for this Document Type is **org.kuali.rice.kew.postprocessor.DefaultPostProcessor**.

- **superUserGroupName**: The super users for this Document Type are members of the TestWorkgroup.

- **blanketApproveGroupName**: The members of the TestWorkgroup have blanketApproval right on this type of document.

- **defaultExceptionGroupName**: The members of the TestWorkgroup will receive an exception notice for documents of this Document Type.

- **docHandler**: The Doc Handler for this type of document is **_blank**.

- **active**: This Document Type is currently active. In other words, it is in use.

- **Policies** for this Document Type:

  - The DEFAULT_APPROVE policy is set **false** by default. In other words, the default approve action on this type of document is NOT to approve it.

  - The INITIATOR_MUST_SAVE policy is set **false** by default. In other words, the initiator does NOT have to save the document for the non-initiator to save the actions on it.

- **routePath**: The routing path for this Document Type is: AdHoc -> WorkflowDocument.

- **routeNode**: Based on the routePath, there are two nodes in the routing of this Document Type:

  - The starting node for this Document Type is **AdHoc**. On the initiation of a document of this type, the postProcessor in KEW activates the node **AdHoc**.

  - The next node in the routing for this Document Type is **WorkflowDocument**. On request, the node is activated and applies the rules in rule template, **WorkflowDocumentTemplate**.

Figure 3.14. Save Action Even Test: Non-Initiator
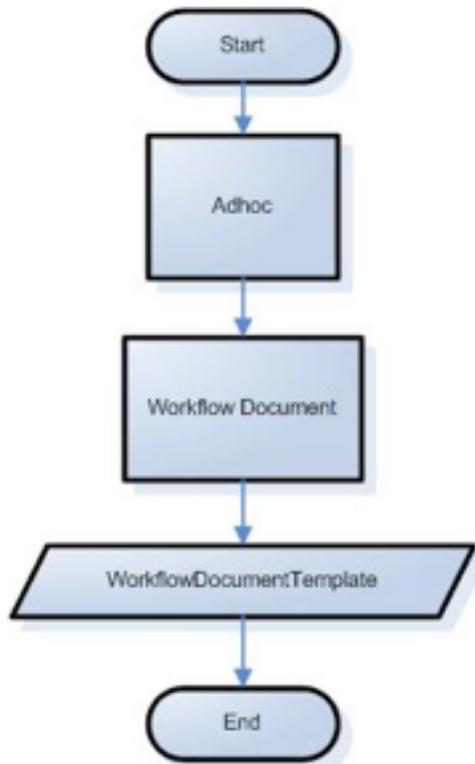
TakeWorkgroupAuthorityDoc

```
<documentType>
    <name>TakeWorkgroupAuthorityDoc</name>
    <description>TakeWorkgroupAuthority Action Test</description>
    <label>TakeWorkgroupAuthorityDoc</label>
    <postProcessorName>org.kuali.rice.kew.postprocessor.DefaultPostProcessor</postProcessorName>
    <superUserGroupName namespace="KR-WKFLW" >TestWorkgroup</superUserGroupName>
    <blanketApproveGroupName namespace="KR-WKFLW">TestWorkgroup</blanketApproveGroupName>
    <defaultExceptionGroupName namespace="KR-WKFLW"> TestWorkgroup</defaultExceptionGroupName>
    <docHandler>_blank</docHandler>
    <active>true</active>
    <policies>
        <policy>
            <name>DEFAULT_APPROVE</name>
            <value>false</value>
        </policy>
    </policies>
    <routePaths>
        <routePath>
            <start name="AdHoc" nextNode="WorkgroupByDocument" />
            <requests name="WorkgroupByDocument" />
        </routePath>
    </routePaths>
    <routeNodes>
        <start name="AdHoc">
            <activationType>P</activationType>
        </start>
        <requests name="WorkgroupByDocument">
            <activationType>P</activationType>
            <ruleTemplate>WorkgroupByDocument</ruleTemplate>
        </requests>
    </routeNodes>
</documentType>
```

- **name**: This is the Document Type for Take Workgroup Authority Doc.

- **description**: This Document Type is used to decide authorized workgroups by Document Type.

- **label**: This Document Type is recognized as the TakeWorkgroupAuthorityDoc type.

- **postProcessorName**: The postProcessor for this Document Type is **org.kuali.rice.kew.postprocessor.DefaultPostProcessor**.

- **superUserGroupName**: The super users for this Document Type are members of the TestWorkgroup.

- **blanketApproveGroupName**: The members of the TestWorkgroup have blanketApproval right on this type of document.

- **defaultExceptionGroupName**: The members of the TestWorkgroup will receive an exception notice for documents of this Document Type.

- **docHandler**: The Doc Handler for this type of document is **_blank**.

- **active**: This Document Type is currently active. In other words, it is in use.

- **Policies** for this Document Type: The DEFAULT_APPROVE policy is set **false** by default. In other words, the default approve action on this type of document is NOT to approve it.

- **routePath**: The routing path for this Document Type is: AdHoc -> WorkflowDocument.

- **routeNode**: Based on the routePath, there are two nodes in the routing of this Document Type:

  - The starting node for this Document Type is **AdHoc**. On the initiation of a document of this type, the postProcessor in KEW activates the node **AdHoc**.

  - The next node in the routing for this Document Type is **WorkflowDocument**. On request, the node is activated and applies the rules in rule template, **WorkflowDocumentTemplate**.

Figure 3.15. Take Workgroup Authority
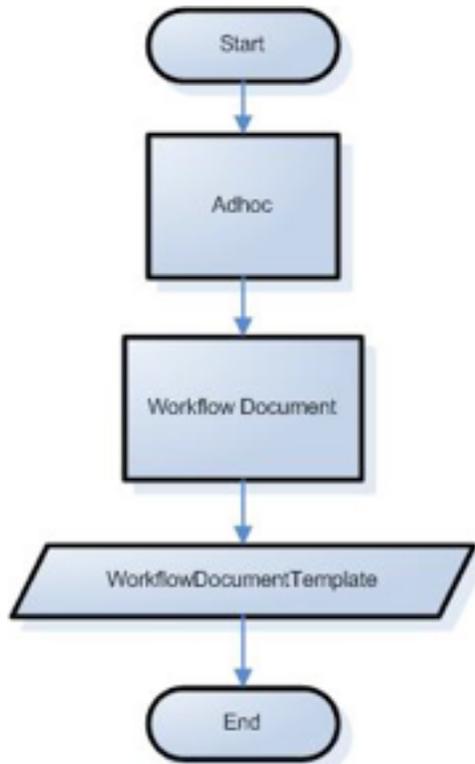
MoveSequentialTest

```
<documentType>
    <name>MoveSequentialTest</name>
    <parent>BlanketApproveTest</parent>
    <description>Move Sequential Test</description>
    <label>Move Sequential Test</label>

    <postProcessorName>org.kuali.rice.kew.postprocessor.DefaultPostProcessor</postProcessorName>
    <superUserGroupName namespace="KR-WKFLW" >TestWorkgroup</superUserGroupName>
    <blanketApproveGroupName namespace="KR-WKFLW">TestWorkgroup</blanketApproveGroupName>
    <defaultExceptionGroupName namespace="KR-WKFLW"> TestWorkgroup</defaultExceptionGroupName>
    <docHandler>_blank</docHandler>
    <active>true</active>
    <routePaths>
        <routePath>
            <start name="AdHoc" nextNode="WorkflowDocument" />
            <requests name="WorkflowDocument" nextNode="WorkflowDocument2" />
            <requests name="WorkflowDocument2" nextNode="Acknowledge1" />
            <requests name="Acknowledge1" nextNode="Acknowledge2" />
            <requests name="Acknowledge2" />
        </routePath>
    </routePaths>
    <routeNodes>
        <start name="AdHoc">
            <activationType>P</activationType>
        </start>
        <requests name="WorkflowDocument">
            <activationType>P</activationType>
            <ruleTemplate>WorkflowDocumentTemplate</ruleTemplate>
        </requests>
        <requests name="WorkflowDocument2">
            <activationType>P</activationType>
            <ruleTemplate>WorkflowDocument2Template</ruleTemplate>
        </requests>
        <requests name="Acknowledge1">
```

```
            <activationType>P</activationType>
            <ruleTemplate>Ack1Template</ruleTemplate>
        </requests>
        <requests name="Acknowledge2">
            <activationType>P</activationType>
            <ruleTemplate>Ack2Template</ruleTemplate>
        </requests>
    </routeNodes>
</documentType>
```

- **name**: This is the Document Type for Move Sequential Test.

- **Parent**: The parent Document Type is BlanketApproveTest. This Document Type inherits the policies that BlanketApproveTest has.

- **description**: This Document Type is used to test Move Sequence.

- **label**: This Document Type is recognized as MoveSequentialTest type.

- **postProcessorName**: The postProcessor for this Document Type is **org.kuali.rice.kew.postprocessor.DefaultPostProcessor**.

- **superUserGroupName**: The super users for this Document Type are members of the TestWorkgroup.

- **blanketApproveGroupName**: The members of the TestWorkgroup have blanketApproval right on this type of document.

- **defaultExceptionGroupName**: The members of the TestWorkgroup will receive an exception notice for documents of this Document Type.

- **docHandler**: The Doc Handler for this type of document is **_blank**.

- **active**: This Document Type is currently active. In other words, it is in use.

- **routePath**: The routing path for this Document Type is: AdHoc -> WorkflowDocument -> WorkflowDocument2 -> Acknowledge1 -> Acknowledge2.

- **routeNode**: Based on the routePath, there are five nodes in the routing of this Document Type:

  - The starting node for this Document Type is **AdHoc**. On the initiation of a document of this type, the postProcessor in KEW activates the node **AdHoc**.

  - The next node in the routing for this Document Type is **WorkflowDocument**. On request, the node is activated, applies the rules in rule template, **WorkflowDocumentTemplate**, and sets the mandatory route as **true**. In other words, the document must route through this node.

  - The next node in the routing for this Document Type is **WorkflowDocument2**. On request, the node is activated, applies the rules in rule template, **WorkflowDocument2Template**, and sets the mandatory route as **true**. In other words, the document must route through this node.

  - The next node in the routing for this Document Type is **Acknowledge1**. On request, the node is activated and applies the rules in rule template, **Ack1Template**.

  - The next node in the routing for this Document Type is **Acknowledge2**. On request, the node is activated and applies the rules in rule template, **Ack2Template**.

Figure 3.16. Move Sequential Test
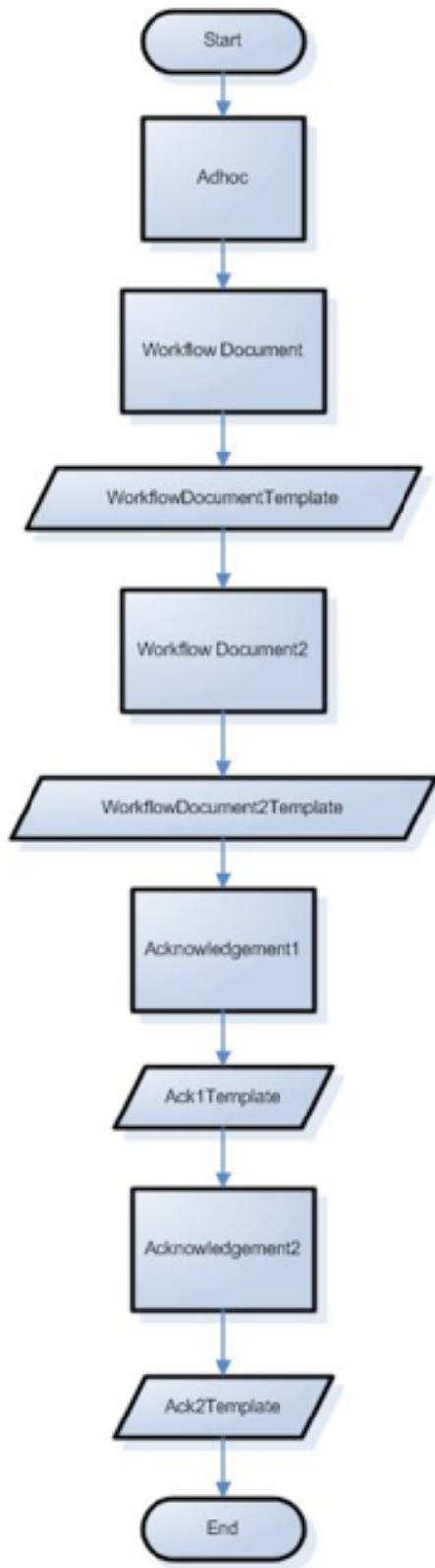
MoveInProcessTest

```
<documentType>
    <name>MoveInProcessTest</name>
    <parent>BlanketApproveTest</parent>
    <description>Move In Process Test</description>
    <label>Move In Process Test</label>
    <postProcessorName>org.kuali.rice.kew.postprocessor.DefaultPostProcessor</postProcessorName>
    <superUserGroupName namespace="KR-WKFLW" >TestWorkgroup</superUserGroupName>
    <blanketApproveGroupName namespace="KR-WKFLW">TestWorkgroup</blanketApproveGroupName>
    <defaultExceptionGroupName namespace="KR-WKFLW"> TestWorkgroup</defaultExceptionGroupName>
    <docHandler>_blank</docHandler>
    <active>true</active>
    <routePaths>
        <routePath>
            <start name="AdHoc" nextNode="WorkflowDocument" />
            <requests name="WorkflowDocument" nextNode="MyRadSubProcess" />
            <process name="MyRadSubProcess" nextNode="WorkflowDocumentFinal" />
            <requests name="WorkflowDocumentFinal" />
        </routePath>
        <routePath processName="MyRadSubProcess" initialNode="WorkflowDocument2">
            <requests name="WorkflowDocument2" nextNode="WorkflowDocument3" />
            <requests name="WorkflowDocument3" />
        </routePath>
    </routePaths>
    <routeNodes>
        <start name="AdHoc">
            <activationType>P</activationType>
        </start>
        <requests name="WorkflowDocument">
            <activationType>P</activationType>
            <ruleTemplate>WorkflowDocumentTemplate</ruleTemplate>
        </requests>
        <process name="MyRadSubProcess" />
        <requests name="WorkflowDocument2">
            <activationType>P</activationType>
            <ruleTemplate>WorkflowDocument2Template</ruleTemplate>
        </requests>
        <requests name="WorkflowDocument3">
            <activationType>P</activationType>
            <ruleTemplate>WorkflowDocument3Template</ruleTemplate>
        </requests>
        <requests name="WorkflowDocumentFinal">
            <activationType>P</activationType>
            <ruleTemplate>WorkflowDocumentFinalTemplate</ruleTemplate>
        </requests>
    </routeNodes>
</documentType>
```

- **name**: This is the Document Type for Move In Process Test.

- **Parent**: The parent Document Type for this Document Type is BlanketApproveTest. This Document Type inherits the policies that BlanketApproveTest has.

- **description**: This Document Type is used to test Move In Process.

- **label**: This Document Type is recognized as the MoveInProcessTest type.

- **postProcessorName**: The postProcessor for this Document Type is **org.kuali.rice.kew.postprocessor.DefaultPostProcessor**.

- **superUserGroupName**: The super users for this Document Type are members of the TestWorkgroup.

- **blanketApproveGroupName**: The members of the TestWorkgroup have blanketApproval right on this type of document.

- **defaultExceptionGroupName**: The members of the TestWorkgroup will receive an exception notice for documents of this Document Type.

- **docHandler**: The Doc Handler for this type of document is **_blank**.

- **active**: This Document Type is currently active. In other words, it is in use.

- **routePath**: The routing path for this Document Type is: AdHoc -> WorkflowDocument -> MyRadSub Process -> WorkflowDocument2 -> WorkflowDocument3 -> WorkflowDocumentFinal. There is a sub-process MyRadSubProcess in this path.

- **routeNode**: As can be seen from the routePath, there are five nodes in the routing of this Document Type:

  - The starting node for this Document Type is **AdHoc**. On the initiation of a document of this type, the postProcessor in KEW activates the node **AdHoc**.

  - The next node in the routing for this Document Type is **WorkflowDocument**. On request, the node is activated, applies the rules in rule template, **WorkflowDocumentTemplate**, and initiates a sub process MyRadSubProcess.

  - The next node in MyRadSubProcess for this Document Type is **WorkflowDocument2**. On request, the node is activated and applies the rules in rule template, **WorkflowDocument2Template**.

  - The next node in MyRadSubProcess for this Document Type is **WorkflowDocument3**. O the request, the node is activated and applies the rules in rule template, **WorkflowDocument3Template**.

  - The next node in the routing for this Document Type is **WorkflowDocumentFinal**. On request, the node is activated and applies the rules in rule template **WorkflowDocumentFinalTemplate**.

Figure 3.17. Move In Process Test
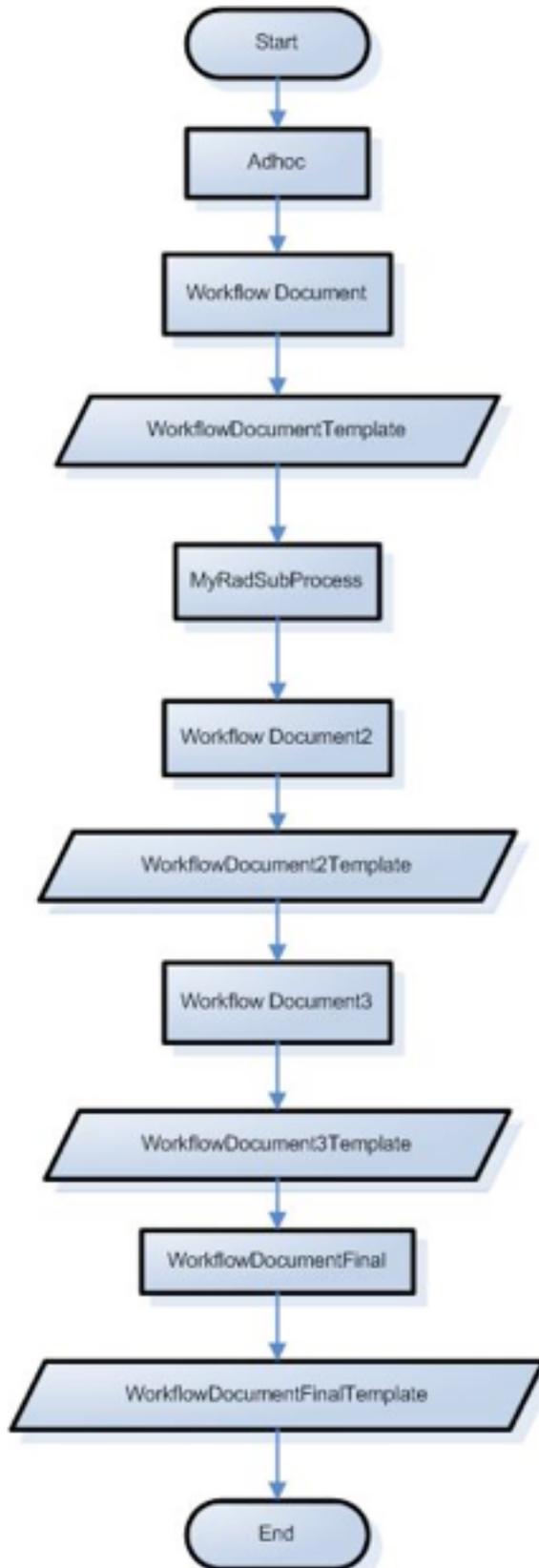
AdhocRouteTest

```
<documentType>
 <name>AdhocRouteTest</name>
 <description>AdhocRouteTest</description>
 <label>AdhocRouteTest</label>

 <postProcessorName>org.kuali.rice.kew.postprocessor.DefaultPostProcessor</postProcessorName>
 <superUserGroupName namespace="KR-WKFLW" >TestWorkgroup</superUserGroupName>
 <blanketApproveGroupName namespace="KR-WKFLW">TestWorkgroup</blanketApproveGroupName>
 <defaultExceptionGroupName namespace="KR-WKFLW"> TestWorkgroup</defaultExceptionGroupName>
 <docHandler>_blank</docHandler>
 <active>true</active>
 <routePaths>
  <routePath>
   <start name="AdHoc" nextNode="One" />
   <requests name="One" />
  </routePath>
 </routePaths>
 <routeNodes>
  <start name="AdHoc">
   <activationType>P</activationType>
  </start>
  <requests name="One">
   <activationType>S</activationType>
   <ruleTemplate>WorkflowDocumentTemplate</ruleTemplate>
  </requests>
 </routeNodes>
</documentType>
```

- **name**: This is the Document Type for Adhoc Route Test.

- **description**: This Document Type is used to test Ad Hoc Route.

- **label**: This Document Type is recognized as the AdhocRouteTest type.

- **postProcessorName**: the postProcessor for this Document Type is **org.kuali.rice.kew.postprocessor.DefaultPostProcessor**.

- **superUserGroupName**: The super users for this Document Type are members of the TestWorkgroup.

- **blanketApproveGroupName**: The members of the TestWorkgroup have blanketApproval right on this type of document.

- **defaultExceptionGroupName**: The members of the TestWorkgroup will receive an exception notice for documents of this Document Type.

- **docHandler**: The Doc Handler for this type of document is **_blank**.

- **active**: This Document Type is currently active. In other words, it is in use.

- **routePath**: The routing path for this Document Type is: AdHoc -> One.

- **routeNode**: Based on the routePath, there are two nodes in the routing of this Document Type:

  - The starting node for this Document Type is **AdHoc**. On the initiation of a document of this type, the postProcessor in KEW activates the node **AdHoc**.

  - The next node in the routing for this Document Type is **One**. On request, the node is activated by the type **S** and applies the rules in rule template, **WorkflowDocumentTemplate**.

Figure 3.18. Adhoc Route Test
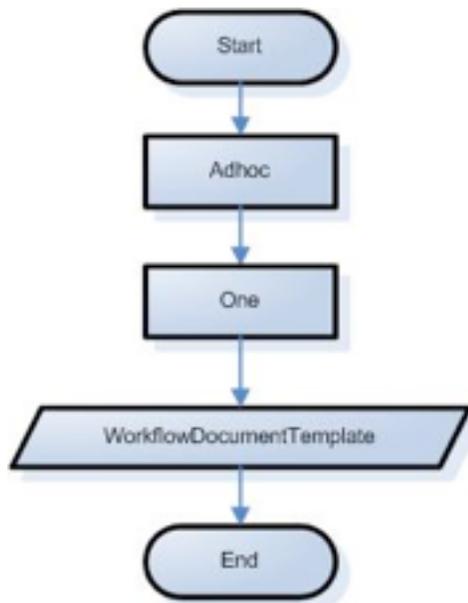
PreApprovalTest

```
<documentType>
    <name>PreApprovalTest</name>
    <description>PreApprovalTest</description>
    <label>PreApprovalTest</label>
    <postProcessorName>org.kuali.rice.kew.postprocessor.DefaultPostProcessor</postProcessorName>
    <superUserGroupName namespace="KR-WKFLW" >TestWorkgroup</superUserGroupName>
    <blanketApproveGroupName namespace="KR-WKFLW">TestWorkgroup</blanketApproveGroupName>
    <defaultExceptionGroupName namespace="KR-WKFLW"> TestWorkgroup</defaultExceptionGroupName>
    <docHandler>_blank</docHandler>
    <active>true</active>
    <routePaths>
        <routePath>
            <start name="AdHoc" nextNode="PreApprovalTestOne" />
            <requests name="PreApprovalTestOne" />
        </routePath>
    </routePaths>
    <routeNodes>
        <start name="AdHoc">
            <activationType>P</activationType>
        </start>
        <requests name="PreApprovalTestOne">
            <activationType>S</activationType>
            <ruleTemplate>WorkflowDocumentTemplate</ruleTemplate>
        </requests>
    </routeNodes>
</documentType>
```

- **name**: This is the Document Type for PreApprovalTest.

- **description**: This Document Type is used to test Pre-Approval.

- **label**: This Document Type is recognized as the PreApprovalTest type.

- **postProcessorName**: The postProcessor for this Document Type is **org.kuali.rice.kew.postprocessor.DefaultPostProcessor**.

- **superUserGroupName**: The super users for this Document Type are members of the TestWorkgroup.

- **blanketApproveGroupName**: The members of the TestWorkgroup have blanketApproval right on this type of document.

- **defaultExceptionGroupName**: The members of the TestWorkgroup will receive an exception notice for documents of this Document Type. • docHandler: The Doc Handler for this type of document is **_blank**.

- **active**: This Document Type is currently active. In other words, it is in use.

- **routePath**: The routing path for this Document Type is: AdHoc -> PreApprovalTestOne.

- **routeNode**: Based on the routePath, there are two nodes in the routing of this Document Type:

  - The starting node for this Document Type is **AdHoc**. On the initiation of a document of this type, the postProcessor in KEW activates the node **AdHoc**.

  - The next node in the routing for this Document Type is **PreApprovalTestOne**. On request, the node is activated by the type **S** and applies the rules in rule template, **WorkflowDocumentTemplate**.

Figure 3.19. PreApproval Test
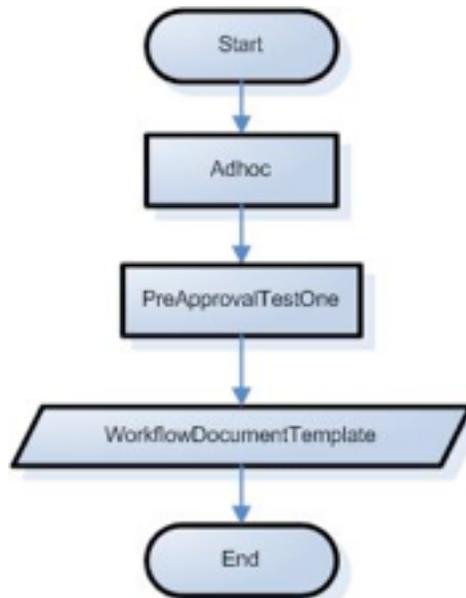
VariablesTest

```
<documentType>
    <name>VariablesTest</name>
    <description>VariablesTest</description>
    <label>VariablesTest</label>

    <postProcessorName>org.kuali.rice.kew.postprocessor.DefaultPostProcessor</postProcessorName>
    <superUserGroupName namespace="KR-WKFLW" >TestWorkgroup</superUserGroupName>
    <blanketApproveGroupName namespace="KR-WKFLW">TestWorkgroup</blanketApproveGroupName>
    <defaultExceptionGroupName namespace="KR-WKFLW"> TestWorkgroup</defaultExceptionGroupName>
    <docHandler>_blank</docHandler>
    <active>true</active>
    <routePaths>
        <routePath>
            <start name="AdHoc" nextNode="setStartedVar" />
            <simple name="setStartedVar" nextNode="setCopiedVar"/>
            <simple name="setCopiedVar" nextNode="PreApprovalTestOne"/>
            <requests name="PreApprovalTestOne" nextNode="setEndedVar"/>
```

```
            <simple name="setEndedVar" nextNode="setGoogleVar"/>
            <simple name="setGoogleVar" nextNode="setXPathVar"/>
            <simple name="setXPathVar" nextNode="resetStartedVar"/>
            <simple name="resetStartedVar" nextNode="logNode"/>
            <simple name="logNode" nextNode="logNode2"/>
            <simple name="logNode2"/>
        </routePath>
    </routePaths>
    <routeNodes>
        <start name="AdHoc">
            <activationType>P</activationType>
        </start>
        <simple name="setStartedVar">
            <type>org.kuali.rice.kew.engine.node.var.SetVarNode</type>
            <name>started</name>
            <value>startedVariableValue</value>
        </simple>
        <simple name="setCopiedVar">
            <type>org.kuali.rice.kew.engine.node.var.SetVarNode</type>
            <name>copiedVar</name>
            <value>var:started</value>
        </simple>
        <requests name="PreApprovalTestOne">
            <activationType>S</activationType>
            <ruleTemplate>WorkflowDocumentTemplate</ruleTemplate>
        </requests>
        <simple name="setEndedVar">
            <type>org.kuali.rice.kew.engine.node.var.SetVarNode</type>
            <name>ended</name>
            <value>endedVariableValue</value>
        </simple>
        <simple name="setGoogleVar">
            <type>org.kuali.rice.kew.engine.node.var.SetVarNode</type>
            <name>google</name>
            <value>url:http://google.com</value>
        </simple>
        <simple name="setXPathVar">
            <type>org.kuali.rice.kew.engine.node.var.SetVarNode</type>
            <name>xpath</name>
            <value>xpath:concat(local-name(//documentContent),$ended)</value>
        </simple>
        <simple name="resetStartedVar">
            <type>org.kuali.rice.kew.engine.node.var.SetVarNode</type>
            <name>started</name>
            <value>aNewStartedVariableValue</value>
        </simple>
        <simple name="logNode">
            <type>org.kuali.rice.kew.engine.node.LogNode</type>
            <message>var:xpath</message>
        </simple>
        <simple name="logNode2">
            <type>org.kuali.rice.kew.engine.node.LogNode</type>
            <level>ErRoR</level>
            <log>Custom.Logger.Name</log>
            <message>THAT'S ALL FOLKS</message>
        </simple>
    </routeNodes>
</documentType>
```

- **name**: This is the Document Type for VariablesTest.

- **description**: This Document Type is used to test Variables.

- **label**: This Document Type is recognized as the VariablesTest type.

- **postProcessorName**: The postProcessor for this Document Type is **org.kuali.rice.kew.postprocessor.DefaultPostProcessor**.

- **superUserGroupName**: The super users for this Document Type are members of the TestWorkgroup.

- **blanketApproveGroupName**: The members of the TestWorkgroup have blanketApproval right on this type of document.

- **defaultExceptionGroupName**: The members of the TestWorkgroup will receive an exception notice for documents of this Document Type.

- **docHandler**: The Doc Handler for this type of document is **_blank**.

- **active**: This Document Type is currently active. In other words, it is in use.

- **routePath**: The routing path for this Document Type is: AdHoc -> setStartedVar -> setCopiedVar -> preApprovalTestOne -> setEndedVar -> setGoogleVar -> setXPathVar -> resetStartedVar -> logNode -> logNode2.

- **routeNode**: Based on the routePath, there are ten nodes in the routing of this Document Type:

  - The starting node for this Document Type is **AdHoc**. On the initiation of a document of this type, the postProcessor in KEW activates the node **AdHoc**.

  - The next node in the routing for this Document Type is **setStartedVar**.

    - Its type is **org.kuali.rice.kew.engine.node.var.SetVarNode**

    - Its name is **started**.

    - Its value is **startedVariableValue**.

  - The next node in the routing for this Document Type is **setCopiedVar**.

    - Its type is **org.kuali.rice.kew.engine.node.var.SetVarNode**.

    - Its name is **copiedVar**.

    - The value that it is copying is var:**started**.

  - The next node in the routing for this Document Type is **preApprovalTestOne**. On request, the node is activated by the type **S** and applies the rules in rule template **WorkflowDocumentTemplate**.

  - The next node in the routing for this Document Type is **setEndedVar**

    - Its type is **org.kuali.rice.kew.engine.node.var.SetVarNode**.

    - Its name is **ended**.

    - Its value is **endedVariableValue**.

  - The next node in the routing for this Document Type is **setGoogleVar**.

    - Its type is **org.kuali.rice.kew.engine.node.var.SetVarNode**.

    - Its name is **google**. It links to **http://google.com**.

  - The next node in the routing for this Document Type is **setXpathVar**.

    - Its type is **org.kuali.rice.kew.engine.node.var.SetVarNode**.

    - Its name is **xpath**.

    - It adds **//documentContent** to the current path.

  - The next node in the routing for this Document Type is **resetStartedVar**.

- Its type is **org.kuali.rice.kew.engine.node.var.SetVarNode**.

- Its name is **started**.

- It resets the started node at a new node, **aNewStartedVariableValue**.

- The next node in the routing for this Document Type is **logNode**.

  - Its type is **org.kuali.rice.kew.engine.node.LogNode**.

  - It sends a message about the xpath of the variables at **var:xpath**.

- The next node in the routing for this Document Type is **logNode2**.

  - Its type is **org.kuali.rice.kew.engine.node.LogNode**.

  - Its level is **ErRoR**.

  - It opens the log **Custom.Logger.Name**.

  - It returns a message **THAT'S ALL FOLKS**.

Figure 3.20. Variables Test
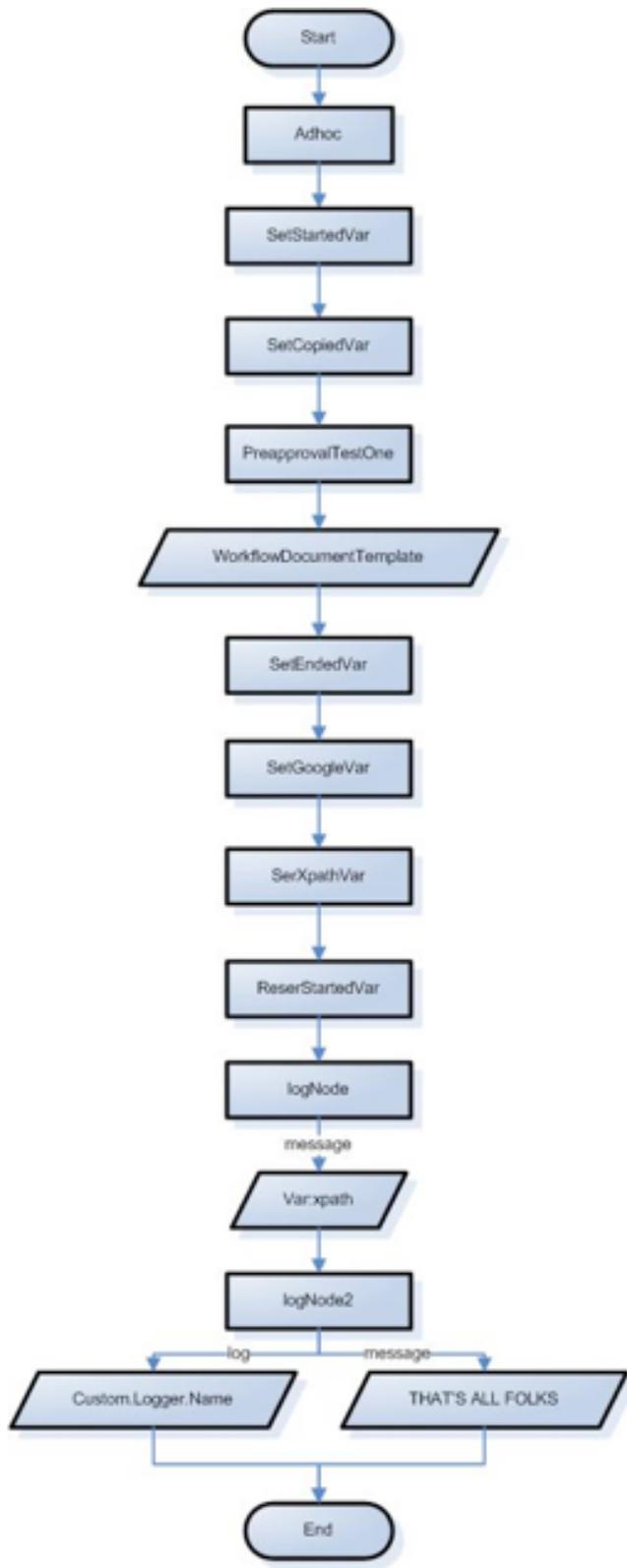
SUApproveDocumentNotifications

```
<documentType>
    <name>SUApproveDocumentNotifications</name>
    <parent>SUApproveDocument</parent>
    <description>SUApproveDocumentNotifications</description>
    <label>SUApproveDocumentNotifications</label>

    <postProcessorName>org.kuali.rice.kew.postprocessor.DefaultPostProcessor</postProcessorName>
    <superUserGroupName namespace="KR-WKFLW" >TestWorkgroup</superUserGroupName>
    <blanketApproveGroupName namespace="KR-WKFLW">TestWorkgroup</blanketApproveGroupName>
    <defaultExceptionGroupName namespace="KR-WKFLW"> TestWorkgroup</defaultExceptionGroupName>
    <docHandler>_blank</docHandler>
    <active>true</active>
    <policies>
        <policy>
            <name>SEND_NOTIFICATION_ON_SU_APPROVE</name>
            <value>true</value>
        </policy>
    </policies>
</documentType>
```

- **name**: This is the Document Type for SuperUser Approve Document Notifications.

- **description**: This Document Type is used to test the SuperUser Approve Document Notifications.

- **label**: This Document Type is recognized as the SUApproveDocumentNotifications type.

- **postProcessorName**: The postProcessor for this Document Type is **org.kuali.rice.kew.postprocessor.DefaultPostProcessor**.

- **superUserGroupName**: The super users for this Document Type are members of the TestWorkgroup.

- **blanketApproveGroupName**: The members of the TestWorkgroup have blanketApproval right on this type of document.

- **defaultExceptionGroupName**: The members of the TestWorkgroup will receive an exception notice for documents of this Document Type.

- **docHandler**: The Doc Handler for this type of document is _blank.

- **active**: This Document Type is currently active. In other words, it is in use.

- There is just one **policy** for this Document Type: The SEND_NOTIFICATION_ON_SU_APPROVE policy is set true by default. In other words, notifications will be automatically sent on SuperUser's approval.

Document Type Authorizer

The Document Type Authorizer is a component that gets called during the routing process to perform authorization checks. Applications can customize this component, for example to introduce custom role qualifiers or permission details, on a per-doctype basis by registering a custom

```
org.kuali.rice.kew.framework.document.security.DocumentTypeAuthorizer
```

implementation.

The DocumentTypeAuthorizer will be called to make the following checks:

- canInitiate

- canBlanketApprove

- canCancel

- canRecall

- canSave

- canRoute

- canSuperUserApproveSingleActionRequest

- canSuperUserApproveDocument

- canSuperUserDisapproveDocument

Document Type Policies

Document Type Policies affect workflow routing behavior.

Current Document Type polices:

- DISAPPROVE

- DEFAULT_APPROVE

- DOCUMENT_STATUS_POLICY

- INITIATOR_MUST_ROUTE

- INITIATOR_MUST_SAVE

- INITIATOR_MUST_CANCEL

- INITIATOR_MUST_BLANKET_APPROVE

- LOOK_FUTURE

- SEND_NOTIFICATION_ON_SU_APPROVE

- SUPPORTS_QUICK_INITIATE

- NOTIFY_ON_SAVE

- blanketApprovePolicy

- ALLOW_SU_POST_PROCESSOR_OVERIDE

- NOTIFY_COMPLETED_ON_RETURN

- NOTIFY_PENDING_ON_RETURN

- RECALL_NOTIFICATION

- ALLOW_SU_FINAL_APPROVAL

- SEND_NOTIFICATION_ON_SU_DISAPPROVE

Document Type Policies defined in the Document Type XML have this structure:

```
<documentType>
    <name>...</name>
    <policies>
        <policy>
            <name>DEFAULT_APPROVE</name>
            <value>true</value>
        </policy>
        <policy>
            <name>LOOK_FUTURE</name>
            <value>false</value>
        <policy>
        <policy>
            <name>DOCUMENT_STATUS_POLICY</name>
            <stringValue>APP</stringValue>
        </policy>
    </policies>
</documentType>
```

DISAPPROVE

The **DISAPPROVE** policy determines whether a document will discontinue routing (transactions). When a document has been disapproved, the document initiator and previous approvers will receive notice of this disapproval action.

DEFAULT_APPROVE

The **DEFAULT_APPROVE** policy determines whether a document will continue processing with or without any approval requests. If a document is set to have no approval requests, its put into exception routing. Then, the document will continue to route to the exception workgroup associated with the last route node or to the workgroup defined in the **defaultExceptionWorkgroupname**.

DOCUMENT_STATUS_POLICY

The **DOCUMENT_STATUS_POLICY** policy sets whether to display the KEW Route Status, the Application Document Status, or Both in the Route Log. Valid policy values are: **KEW**, **APP**, or **BOTH**.

The set of valid statuses for a given document type may be defined. If defined, only those values are allowed as valid statuses. These will also be used to populate a multi-select box on the doc search screen if this doc type is selected (see [Customizing Document Search: Application Document Status](#)). If not defined, any string with a length of up to 64 characters may be used, and a text input field is used on the doc search screen. An example configuration follows.

```
<validApplicationStatuses>
    <status>Initiated</status>
    <status>Validated</status>
    <status>Awaiting Content Aproval</status>
    <status>Org Review</status>
    <status>Approved</status>
    ...
</validApplicationStatuses>
```

Additionally, the valid statuses may be grouped into named categories for display and search purposes. If defined, these categories will display in doc search (again, if the document type is selected) under the application document status multi-select as headings under which the individual statuses are grouped.

These categories can be selected as well, which has an equivalent effect on the search to individually selecting all of the statuses within the category (see Customizing Document Search: Application Document Status). Note that not all statuses need be grouped within categories, as demonstrated below by the "Approved" status below.

```
<validApplicationStatuses>
    <category name="Pre-Submit">
        <status>Initiated</status>
        <status>Validated</status>
    </category>
    <category name="In Process">
        <status>Awaiting Content Aproval</status>
        <status>Org Review</status>
    </category>
    <status>Approved</status>
    ...
</validApplicationStatuses>
```

In the process definition section, automatic status updates may be assigned to occur on route node transition. Please see the section on the routePaths in this guide.

INITIATOR_MUST_ROUTE

The **INITIATOR_MUST_ROUTE** policy sets the rule that the user who initiates the document must route it.

INITIATOR_MUST_SAVE

The **INITIATOR_MUST_SAVE** policy sets the rule that the user who initiated the document will be the only one authorized to **save** the document.

INITIATOR_MUST_CANCEL

The **INITIATOR_MUST_CANCEL** policy sets the rule that the user who initiated the document will be the only one authorized to **cancel** the document.

INITIATOR_MUST_BLANKET_APPROVE

The **INITIATOR_MUST_BLANKET_APPROVE** policy sets the rule that the user who initiated the document is the only one authorized to **blanket approve** the document.

LOOK_FUTURE

The **LOOK_FUTURE** policy determines whether the document can be brought into a simulated route from the route log. This policy simulates where the document would end up if it completed the route.

SEND_NOTIFICATION_ON_SU_APPROVE

The **SEND_NOTIFICATION_ON_SU_APPROVE** policy indicates to KEW that it is to send a notification on SuperUser approval.

SUPPORTS_QUICK_INITIATE

The **SUPPORTS_QUICK_INITIATE** policy indicates whether the Document Type is displayed on the Quick Links, so that users can quickly initiate instances of the document.

NOTIFY_ON_SAVE

The **NOTIFY_ON_SAVE** policy indicates whether a notification should be sent in when a save action is applied to this Document Type.

blanketApprovePolicy

The **blanketApprovePolicy** policy indicates who can **blanket approve** a workflow document. Its values are either ANY or NONE.

• ANY means that anybody can blanket approve the document.

• NONE means that no one can blanket approve the document.

Alternatively, the configuration of the document can be set up to specify a blanketApproveWorkgroupName. blanketApproveWorkgroupName indicates that members of that workgroup can blanket approve the document. You can specify either blanketApprovePolicy OR blanketApproveWorkgroupName in the Document Type.

Since the blanket approve policy is not a true/false policy (like the others), it is specified as an element in the Document Type XML:

```
<documentType>
    <name>...</name>
    .
    .
    .
    <blanketApprovePolicy>NONE</blanketApprovePolicy>
</documentType>
```

ALLOW_SU_POST_PROCESSOR_OVERIDE

There is currently the ability to override the "Perform Post Processor Logic" on the "Super User Action on Action Requests" page. This functionality is configurable by document type and as such allows for inheritance.

By default, the ALLOW_SU_POST_PROCESSOR_OVERIDE it's set to true. The checkbox appears on the super user screen as:

Figure 3.21. Super User Action on Requests

In order to turn off the post processor check box, you would add this to the documentType definition:

```
<policies>
    <policy>
        <name>ALLOW_SU_POSTPROCESSOR_OVERRIDE</name>
        <value>false</value>
    </policy>
</policies>
```

Recall From Routing

Three Document Type policies affect Recall behavior. These policies are defined in the respective the DocumentType XML. The following two policies apply to Return-To-Previous actions as well as Recall actions:

NOTIFY_COMPLETED_ON_RETURN - Default: false toggles whether to notify previous router log participants with FYIs when a document is recalled. This does not affect notifications to pending approvers which are always sent.

Example:

```
<policy>
    <name>NOTIFY_COMPLETED_ON_RETURN</name>
    <value>true</value>
</policy>
```

NOTIFY_PENDING_ON_RETURN - Default: true toggles whether to notify pending approvers with FYIs when a document is recalled. This does not affect notifications to prior approvers.

Example:

```
<policy>
    <name>NOTIFY_PENDING_ON_RETURN</name>
    <value>true</value>
</policy>
```

The following policy is Recall-specific:

RECALL_NOTIFICATION - Default: false/none

Example:

```
<policy>
    <name>RECALL_NOTIFICATION</name>
    <value>true</value>
    <recipients xmlns:r="ns:workflow/Rule" xsi:schemaLocation="ns:workflow/Rule resource:Rule"
 xmlns:dt="ns:workflow/DocumentType">
        <r:principalName>quickstart</r:principalName>
        <r:user>quickstart</r:user>
        <role namespace="KR-SYS" name="Technical Administrator"/>
    </recipients>
</policy>
```

ALLOW_SU_FINAL_APPROVAL

Setting this policy to false disallows Super User approval on final nodes of the document.

```
<policies>
            <policy>
            <name>ALLOW_SU_FINAL_APPROVAL</name>
            <value>false</value>
            </policy>
            </policies>
```

SEND_NOTIFICATION_ON_SU_DISAPPROVE

By default, acknowledgedments are not sent on Super User Disapproval like they are for normal Disapprove actions. This policy can be used to enable sending of acknowledgements upon Super User Disapproval.

```
<policies>
            <policy>
            <name>SEND_NOTIFICATION_ON_SU_DISAPPROVE</name>
            <value>true</value>
            </policy>
            </policies>
```

Inheritance

Document Types can specify a parent Document Type. This allows them to be included in a Document Type hierarchy from which certain behavior can be inherited from their parent Document Type.

Inheritable Fields

These fields are inherited:

- **superUserGroupName**: Indicates members of the workgroup who can perform SuperUser actions on the document

- **blanketApproveGroupName**: Indicates members of the workgroup that can blanket approve the document.

- **notificationFromAddress**: Sends a notice to the sender when the transfer of the document is completed.

- **messageEntity**: A head and body of the message.

- **policies**: Indicates a set of rule(s) applied in the document. For each policy, True means policy DOES apply, False means policy does NOT apply.

- **searchable attributes**: Constraint(s) assigned as the searchable criteria for a document.

- **route paths/route nodes**: Designated traveling points before the document reaches its destination in a routing process.

Special notes about inheritance:

1. **Policies**: In the Policies section, there are multiple Document Type policies (INITAITOR_MUST_ROUTE, DEFAULT_APPROVE, etc). Each policy can be overridden on an **individual basis**. In contrast to the route path, there is no need to override the entire **policies** section for a Document Type. For more detailed information about Document Type policies, please see Document Type Policies (above) in this document.

2. **Route paths/ route nodes**: To override the route path and route node definitions of a parent Document Type, you must override ALL route node and route path definitions. You cannot inherit and use just part of a route path; it's all or nothing.

Document Type hierarchy and the Rules Engine

The Rules Engine follows these rules to determine its rule evaluation set for a Document Type at a particular node:

1. The Rules Engine looks at the Rule Template name of the current node and selects all rules with that template and that document's Document Type. It adds those rules to the rule evaluation set.

2. If the Document Type has a parent Document Type, it selects all rules with that template and that parent Document Type and adds those to the rule evaluation set.

3. Its repeats step two until it reaches the root of the Document Type hierarchy.

4. The final rule evaluation set includes all of these rules.

Defining Workflow Processes Using PeopleFlow - a new feature in KEW

PeopleFlow is our Kuali Rice instantiation of the "maps" concept included in the original Coeus. For all intents and purposes it's a prioritized list of people to send requests to. PeopleFlow gives you a new type of request activation strategy called "priority-parallel" to activate requests generated from a PeopleFlow in the appropriate order. Essentially, it's like a mini people-based workflow that doesn't require you to specify a KEW node in the document type for each individual who might need to approve or be notified. You can define "Stops" in a PeopleFlow, where everything in the same stop proceeds in parallel, but all must be done within the stop before proceeding to the next stop.

You can call/execute a PeopleFlow from within a KEW workflow node directly, or you can invoke the Kuali Rules Managment System (KRMS) engine from an application and any PeopleFlows that get selected during rule execution, defined in a KRMS agenda, will be called. In this way, you can integrate business rules across applications and workflows.

The same PeopleFlow that defines a routing order among a set of persons, groups or roles can be called by KRMS rules, with the KRMS rules defining which type of request to pass to the PeopleFlow (for example, an "approval" routing action or a "notification").

KRMS is also a new feature in Rice 2.0. See the KRMS Technical Guide for more information on KRMS.

You can define a PeopleFlow (simple workflow) via a maintenance document. See the KEW Users' Guide for additional details on defining a PeopleFlow.

Technical Information about PeopleFlow

(decide what needs to go here -- architecture, data model, api, troubleshooting, etc.?)

KEW Routing Components and Configuration Guide

KEW has several components that you can use to configure routing. Typically a single application will write a set of these components for reuse across multiple Document Types. These components are wired together using an XML configuration file that you need to import into KEW. See Importing XML Files to KEW for more information.

This document looks at defining the routing components available in KEW and how to use these components to make a cohesive routing setup.

- **RouteModule** - The most basic module; it allows KEW to generate Action Requests

- **RuleAttribute** - A component that fits into KEW's rule system. These rules are used to build routing paths for documents. They function for users across the organization and for multiple applications.

- **XML RuleAttribute** – Similar in functionality to a RuleAttribute but built using XML only

- **RoleAttribute** - A component that fits into KEW's rule system, but which is a pointer to outside data. See Built-in Roles and Nodes for more information on implementing a RoleAttribute.

- **PostProcessor** - A component that gets called throughout the routing process and handles a set of standard events that all eDocs (electronic documents) go through.

- **DocumentType Authorizer** - A component that gets called during the routing process to perform authorization checks. Applications can customize this component on a per-doctype basis.

These components are contained in a Document Type that is defined in XML. A Document Type is the prototype for eDocs. Below is the Document Type configuration that explains how KEW uses the eDoc rule:

```
<?XML version="1.0" encoding="UTF-8"?>
<data XMLns="ns:workflow" XMLns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="ns:workflow
 resource:WorkflowData">
    <documentTypes XMLns="ns:workflow/DocumentType" xsi:schemaLocation="ns:workflow/DocumentType
 resource:DocumentType">
        <documentType>
            <name>YOURSERVICE-DOCS.RuleDocument</name>
            <parent>YOURSERVICE-DOCS</parent>
            <description>Add/Modify Workflow rules</description>
            <label>Add/Modify Workflow rules</label>
            <postProcessorName>your.package.routetemplate.RulePostProcessor</postProcessorName>
            <authorizer>your.package.CustomDocumentTypeAuthorizer</authorizer>
            <superUserGroupName>WorkflowAdmin</superUserGroupName>
            <blanketApproveGroupName>IU-WORKFLOW-RULE-BLANKET-APPROVERS</blanketApproveGroupName>
            <defaultExceptionGroupName>YOUR_EXCEPTION_TEAM</defaultExceptionGroupName>
            <docHandler>https://yourlocalIP/en-prd/Rule.do?methodToCall=docHandler</docHandler>
            <notificationFromAddress>...@yourEmailServerIP.edu</notificationFromAddress>
            <active>true</active>
            <routingVersion>1</routingVersion>
            <routePaths>
                <routePath>
                    <start name="Adhoc Routing" nextNode="Rule routing Route Level" />
                    <requests name="Rule routing Route Level" />
                </routePath>
            </routePaths>
            <routeNodes>
                <start name="Adhoc Routing">
                    <activationType>S</activationType>
                    <mandatoryRoute>false</mandatoryRoute>
                    <finalApproval>false</finalApproval>
                </start>
```

```
                <requests name="Rule routing Route Level">
                    <activationType>S</activationType>
                    <ruleTemplate>RuleRoutingTemplate</ruleTemplate>
                    <mandatoryRoute>true</mandatoryRoute>
                    <finalApproval>false</finalApproval>
                </requests>
            </routeNodes>
        </documentType>
    </documentTypes>
</data>
```

Configuration Steps

Let's go through the configuration step-by-step and explain what all the pieces mean:

DocumentTypeName Definition

```
<name>YOURSERVICE-DOCS.RuleDocument</name>
<parent>YOURSERVICE-DOCS</parent>
<description>Add/Modify Workflow rules</description>
<label>Add/Modify Workflow rules</label>
```

The section above defines the Document Type's name, its **parent**, **description**, and **label**. The **name** is used by the client application's API to communicate with KEW. Here is a sample of code from the client application's API communicating with KEW:

```
WorkflowDocument document = new WorkflowDocument(new NetworkIdVO("username"), "DocumentTypeName");
document.routeDocument("user inputted annotation");
```

The above code will route a document in KEW.

- The string **DocumentTypeName** exists in KEW and you define it using the **<name>** element.

- The **parent** element gives the Document Type a parent Document Type. Use this for inheritance of routing configuration and policies.

- **Description** is defined as shown. The document's *Description* is displayed on the Document Type report.

- **Label** is typically the forward-facing name for the Document Type. The label is displayed to the user when an eDoc is in their Action List and they use it when they search for an eDoc using DocSearch.

PostProcessor Class

```
<postProcessorName>your.package.routetemplate.RulePostProcessor</postProcessorName>
```

The element above determines which class to use for the PostProcessor for this particular Document Type. This component receives event notifications as eDocs travel through routing.

DocumentTypeAuthorizer Class

```
<authorizer>your.package.CustomDocumentTypeAuthorizer</authorizer>
```

The element above determines which class to use for the DocumentType Authorizer for this particular Document Type. This component performs authorization checks as the eDoc travels through routing.

Managed Workgroups

```
<superUserWorkgroupName>WorkflowAdmin</superUserWorkgroupName>
<blanketApproveWorkgroupName>WorkgroupBlanketApprovers</blanketApproveWorkgroupName>

<defaultExceptionWorkgroupName>WorkflowAdmin</defaultExceptionWorkgroupName>
```

This section sets KEW managed workgroups in several roles in the Document Type.

- **SuperUserWorkgroupName** defines the workgroup that determines whether a person is allowed to take Super User Actions on a document through the Super User interface.

- The content of element **blanketApproveWorkgroupName** determines which people have access to blanket approve a document.

- **defaultExceptionWorkgroup** determines to which workgroup to send an eDoc of this type if it goes into exception routing. This is an optional element. You can also define Exception Workgroups with a route node.

docHandler

```
<docHandler>https://yourlocalIP/en-prd/Rule.do?methodToCall=docHandler</docHandler>
```

The docHandler tells KEW where to forward users when they click an eDoc link. See Document Search for more information.

notificationFromAddress

```
<notificationFromAddress>...@yourEmailServerIP</notificationFromAddress>
```

When KEW sends an email notification to a user regarding a document of this type, the From address on the message is the address specified here. This is helpful because users will often reply to the messages they receive from KEW, and this allows their responses to go to an appropriate address for the Document Type. This is an optional element. If it is not defined here, KEW uses the default From address. See the Installation Guide for more detail.

active

```
<active>true</active>
```

Use active to define the activeness of a Document Type. KEW does not allow anyone to create eDocs of an inactive Document Type.

routePaths

```
<routePaths>
```

```
    <routePath>
        <start name="Adhoc Routing" nextNode="Rule routing Route Level" />
        <requests name="Rule routing Route Level" />
    </routePath>
</routePaths>
```

The above defines the path an eDoc will travel as it progresses through its life. **Start** and **Requests** are some of the standard node types used. There is only one stop each eDoc must make as it travels through workflow. The eDoc starts at the step **Adhoc Routing** and then progresses to the request node named **Rule routing Route Level**.

Additionally, an automatic progression of the application document status may be configured to occur on route node transition with the addition of the nextAppDocStatus attribute in the elements of a routePath:

```
<routePaths>
    <routePath>
        <start name="Initiated" nextNode="DestinationApproval" nextAppDocStatus="Approval in Progress"/>
        <requests name="DestinationApproval" nextNode="TravelerApproval" nextAppDocStatus="Submitted"/>
        <requests name="TravelerApproval" nextNode="SupervisorApproval" />
        <requests name="SupervisorApproval" nextNode="AccountApproval" />
        <requests name="AccountApproval" />
    </routePath>
</routePaths>
```

This section only defines the path the eDocs will travel, and optionally the application document status transitions. The nodes themselves are defined below.

Node Definition XML

```
<routeNodes>
    <start name="Adhoc Routing">
        <activationType>S</activationType>
        <mandatoryRoute>false</mandatoryRoute>
        <finalApproval>false</finalApproval>
    </start>
    <requests name="Rule routing Route Level">
        <activationType>S</activationType>
        <ruleTemplate>RuleRoutingTemplate</ruleTemplate>
        <mandatoryRoute>true</mandatoryRoute>
        <finalApproval>false</finalApproval>
    </requests>
</routeNodes>
```

This is the node definition XML. This determines certain behaviors each node can have.

Activation Type determines if Approve requests are activated all at once or one at a time. Any given requests node can generate multiple rules that can then generate multiple requests. The ActivationType value specifies if *all* action requests generated for *all* fired rules are activated immediately (P = *parallel* activation), or if the set of action requests generated by each rule are activated one after the other, according to rule order (S = *sequential* activation). However, to activate requests starting with those with the smallest priority *and* to active all those requests in parallel the activation type of (R = *priority-parallel* activation). Once all requests are approved, then the next priority will be activated. This is essentially a hyprid of the traditional sequential and parallel activation types. Activation type is only relevant when multiple rules are generated.

Figure 3.22. Parallel and Sequential Activation Types
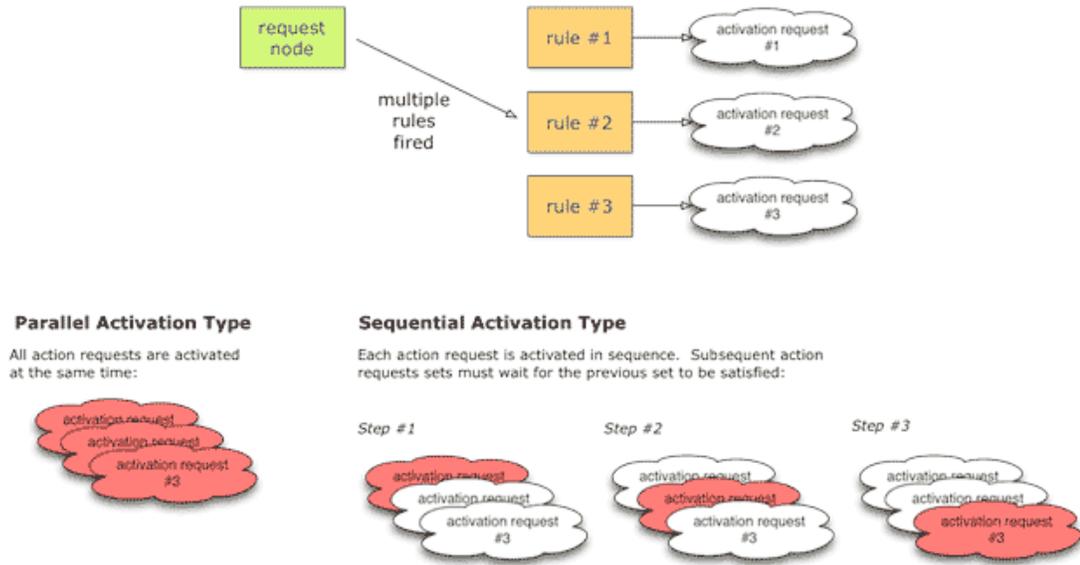
Figure 3.23. Parallel-Priority Activation Type
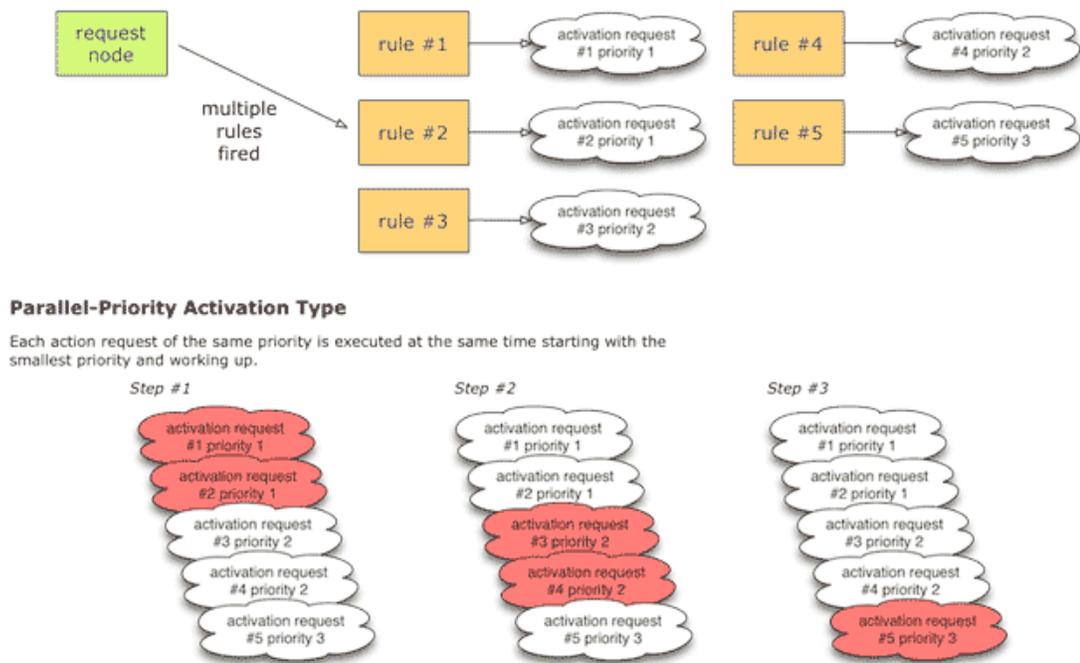
The **mandatoryRoute** key determines if it's mandatory to generate approval requests. If a route node is *mandatory* and it doesn't generate an *approve* request, the document is put in exception routing.

The **finalapproval** key determines if this node should be the last node that has an *approve* request. If approvals are generated after this step, the document is thrown into exception routing.

Finally, there is a request node named *Rule routing Route Level* with a key called **ruleTemplate**. This is our hook into the rule system for KEW:

```
<ruleTemplate>RuleRoutingTemplate</ruleTemplate>
```

And this is our hook into a route module:

```
<routeModule>package.your.ARouteModule</routeModule>
```

KEW contacts the route module when the document enters that route node and the route module returns Action Requests for KEW to deliver.

Rule Attributes

If the application integrating with KEW is using Rules to contain the routing data and **RuleAttributes** for document evaluation, then the routing configuration requires more XML. Below is an XML snippet that defines **RuleAttribute**; this is written in Java.

```
<?xml version="1.0" encoding="UTF-8"?>
<data xmlns="ns:workflow" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="ns:workflow
 resource:WorkflowData">
    <ruleAttributes XMLns="ns:workflow/RuleAttribute" xsi:schemaLocation="ns:workflow/RuleAttribute
 resource:RuleAttribute">
        <ruleAttribute>
            <name>RuleRoutingAttribute</name>
            <className>org.kuali.rice.kew.rule.RuleRoutingAttribute</className>
            <label>RuleRoutingAttribute</label>
            <description>RuleRoutingAttribute</description>
            <type>RuleAttribute</type>
        </ruleAttribute>
    </ruleAttributes>
</data>
```

The above defines a **RuleAttribute** called *RuleRoutingAttribute*. *RuleRoutingAttribute* maps to the Java class **org.kuali.rice.kew.rule.RuleRoutingAttribute**. The *type* of this attribute is a **RuleAttribute**; essentially this means the RuleAttribute's behavior is determined in a Java class. There are also RuleAttributes made entirely from XML, but programming attributes is outside the scope of this Guide.

Rule Templates

Finally, we need to tie the **RuleAttribute** to the Document Type. This is done using the **RuleTemplate** and it is defined using XML. The **RuleTemplate** schema below provides further explanation:

```
<?xml version="1.0" encoding="UTF-8"?>

<data xmlns="ns:workflow" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="ns:workflow
 resource:WorkflowData">
    <ruleTemplates XMLns="ns:workflow/RuleTemplate" xsi:schemaLocation="ns:workflow/RuleTemplate
 resource:RuleTemplate">
        <ruleTemplate>
            <name>RuleRoutingTemplate</name>
            <description>RuleRoutingTemplate</description>
            <attributes>
                <attribute>
                    <name>RuleRoutingAttribute</name>
                    <required>true</required>
                </attribute>
            </attributes>
        </ruleTemplate>
    </ruleTemplates>
</data>
```

Note

Notice that the name of this RuleTemplate, *RuleRoutingTemplate*, matches the name given in the **ruleTemplate** element in the Document Type route node declaration. Also, notice that the **RuleAttribute** made above is referenced in the **RuleTemplate** above in the *attributes* section.

```
<attributes>
    <attribute>
        <name>RuleRoutingAttribute</name>
        <required>true</required>
    </attribute>
</attributes>
```

The **RuleTemplate** is the join between **RuleAttributes** and Document Types. In this way, we can reuse the same attribute declaration (and therefore Java logic) across Document Types.

Once the XML, condensed into a single file, is uploaded into KEW, eDocs of this type can be created and routed from a client application.

All the content in the code examples above is aggregated into a single file below with a single surrounding data tag:

```
<?xml version="1.0" encoding="UTF-8"?>
<data xmlns="ns:workflow" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="ns:workflow
 resource:WorkflowData">
    <ruleAttributes xmlns="ns:workflow/RuleAttribute" xsi:schemaLocation="ns:workflow/RuleAttribute
 resource:RuleAttribute">
        <ruleAttribute>
            <name>RuleRoutingAttribute</name>
            <className>org.kuali.rice.kew.rule.RuleRoutingAttribute</className>
            <label>foo</label>
            <description>foo</description>
            <type>RuleAttribute</type>
        </ruleAttribute>
    </ruleAttributes>
    <ruleTemplates xmlns="ns:workflow/RuleTemplate" xsi:schemaLocation="ns:workflow/RuleTemplate
 resource:RuleTemplate">
        <ruleTemplate>
            <name>RuleRoutingTemplate</name>
            <description>RuleRoutingTemplate</description>
            <attributes>
                <attribute>
                    <name>RuleRoutingAttribute</name>
                    <required>true</required>
                </attribute>
            </attributes>
        </ruleTemplate>
    </ruleTemplates>
    <documentTypes xmlns="ns:workflow/DocumentType" xsi:schemaLocation="ns:workflow/DocumentType
 resource:DocumentType">
        <documentType>
            <name>EDENSERVICE-DOCS.RuleDocument</name>
            <parent>EDENSERVICE-DOCS</parent>
            <description>Add/Modify Workflow rules</description>
            <label>Add/Modify Workflow rules</label>
            <postProcessorName>org.kuali.rice.kew.postprocessor.RulePostProcessor</postProcessorName>
            <superUserGroupName namespace=KR-WKFLW">WorkflowAdmin</superUserGroupName>
            <blanketApproveGroupName namespace=KR-WKFLW">WorkflowAdmin</blanketApproveGroupName>
            <defaultExceptionGroupName></defaultExceptionGroupName>
            <docHandler>https://yourlocalIP/en-prd/Rule.do?methodToCall=docHandler</docHandler>
            <active>true</active>
            <routingVersion>1</routingVersion>
            <routePaths>
                <routePath>
                    <start name="Adhoc Routing" nextNode="Rule routing Route Level" />
                    <requests name="Rule routing Route Level" />
                </routePath>
            </routePaths>
            <routeNodes>
```

```
                    <start name="Adhoc Routing">
                        <activationType>S</activationType>
                        <mandatoryRoute>false</mandatoryRoute>
                    </start>
                    <requests name="Workflow Document Routing">
                        <activationType>S</activationType>
                        <ruleTemplate>RuleRoutingTemplate</ruleTemplate>
                        <mandatoryRoute>true</mandatoryRoute>
                    </requests>
                </routeNodes>
            </documentType>
        </documentTypes>
</data>
```

Routing Rules

There is a separate User Guide on how to use the Rule UI. This will show you how to create a Rule as well as modify and delete.

InitiatorRoleAttribute

InitiatorRoleAttribute is a RoleAttribute that exposes an INITIATOR abstract role that resolves to the initiator of the document.

Table 3.10. InitiatorRoleAttribute

Name	Address
Class	InitiatorRoleAttribute
Package	org.kuali.rice.kew.rule
Full	org.kuali.rice.kew.rule.InitiatorRoleAttribute

RoutedByUserRoleAttribute

RoutedByUserRoleAttribute is a RoleAttribute that exposes the user who routed the document.

Table 3.11. RoutedByUserRoleAttribute

Name	Address
Class	RoutedByUserRoleAttribute
Package	org.kuali.rice.kew.rule
Full	org.kuali.rice.kew.rule.RoutedByUserRoleAttribute

NoOpNode

NoOpNode is a SimpleNode implementation that is a code structure example, but has no functionality.

Table 3.12. NoOpNode

Name	Address
Class	NoOpNode
Package	org.kuali.rice.kew.engine.node
Full	org.kuali.rice.kew.engine.node.NoOpNode

RequestActivationNode

RequestActivationNode is a SimpleNode that activates any requests on it. It returns true when there are no more requests that require activation.

In *RequestActivationNode*, the *activateRequests* method activates the Action Requests that are pending at this route level of the document. The requests are processed by Priority and then by Request ID. The requests are activated implicitly according to the route level.

Acknowledgement Requests do not cause processing to stop. Only Action Requests for Approval or Completion cause processing to stop at the current document's route level. Inactive requests at a lower level cause a routing exception.

Table 3.13. RequestActivationNode

Name	Address
Class	RequestActivationNode
Package	org.kuali.rice.kew.engine.node
Full	org.kuali.rice.kew.engine.node.RequestActivationNode

NetworkIdRoleAttribute

NetworkIdRoleAttribute is a RoleAttribute that routes the request to a NetworkID specified in the document content.

Table 3.14. NetworkIdRoleAttribute

Name	Address
Class	NetworkIdRoleAttribute
Package	org.kuali.rice.kew.engine.node
Full	org.kuali.rice.kew.engine.node.NetworkIdRoleAttribute

UniversityIdRoleAttribute

UniversityIdRoleAttribute is a RoleAttribute that routes requests to an Empl ID specified in the document content.

Table 3.15. UniversityIdRoleAttribute

Name	Address
Class	UniversityIdRoleAttribute
Package	org.kuali.rice.kew.engine.node
Full	org.kuali.rice.kew.engine.node.UniversityIdRoleAttribute

SetVarNode

SetVarNode is a SimpleNode that allows you to set document variables.

The definition of *SetVarnode* takes these configuration parameter elements:

- **Name**: The name of the variable to set

- **Value**: The value to which to set the variable. This value is parsed according to Property/ PropertyScheme syntax. The default *PropertyScheme* is *LiteralScheme*, which evaluates the value simply as a literal; it won't do anything but return the value.

Table 3.16. SetVarNode

Name	Address
Class	SetVarNode
Package	org.kuali.rice.kew.engine.node.var

Name	Address
Full	org.kuali.rice.kew.engine.node.var.SetVarNode

Routing Configuration using KIM Responsibilities

In addition to routing workflow based on users and workgroups using routing rules, you can also route workflow based on KIM responsibilities. This allows you to utilize group membership and role assignments to manage who is permitted to perform approvals.

Route Node Definition

In review, you define a rule-based routing node with XML similar to:

```
<requests name="Rule routing Route Level">
    <activationType>S</activationType>
    <ruleTemplate>RuleRoutingTemplate</ruleTemplate>
    <mandatoryRoute>true</mandatoryRoute>
    <finalApproval>false</finalApproval>
</requests>
```

A routing node that uses KIM responsibilities can replace a rule-based routing node. You define it with XML similar to:

```
<role name="Purchasing">
    <qualifierResolverClass>
org.kuali.rice.kns.workflow.attribute.DataDictionaryQualifierResolver
</qualifierResolverClass>
    <activationType>P</activationType>
</role>
```

Node Name

You name the routing node with the name attribute, just like for a rule-based routing node.

Qualifier Resolver

The qualifier resolver finds any qualifiers that need to be used while matching the responsibility. You can specify it in either of two ways:

- **<qualifierResolver>name</qualifierResolver>** names a rule attribute which identifies the class to use

- **<qualifierResolverClass>class.name</qualifierResolverClass>** provides the fully-qualified name of the Java class to use

Other Options

You can specify **<responsibilityTemplateName>name</responsibilityTemplateName>** to identify the responsibility template to use. This option is not usually used since all of the responsibilities provided with KIM use a template named **Review**.

You can specify **<namespace>name</namespace>** to identify the name space for the responsibility. This option is usually not used since all of the responsibilities provided with KIM use a name space of **KR-WKFLW**.

Matching Routing Nodes to Responsibilities

The KIM responsibility template **Review** defines two details:

• The name of the document type

• The name of the routing node

When you define a responsibility in KIM using this template, you specify a value for each of these details. When a document is routed using responsibility-based routing nodes, KIM receives the type of the document being routed and the name of the node; it then locates any responsibilities which have the same routing node name and either the same document type name or the name of a parent document type (all the way up to the top of the hierarchy). The list of people who gets the request consists of anyone who has been assigned a role with any of the matching responsibilities.

Using the Workflow Document API

Overview

This document explains features of the workflow document API. There are two interfaces in KEW that allow you to create a document for delivery through workflow. The **WorkflowDocument** interface is designed to create a new document in the workflow system once an action has been taken, such as sending ad hoc requests. The **WorkflowInfo** interface is actually a convenience class for client applications that query workflow. Both classes assist with implementing connections to KEW.

WorkflowDocument

The process for this section of the API involves creating the initial **WorkflowDocument** using a constructor to create a new routable document in KEW. Once the object is defined, it initializes by loading an existing *routeHeaderId* or by constructing an empty document of a specified *documentType*. A number of methods can be invoked once initialization is complete and details of how those methods would be invoked are outlined primarily in the Java Documentation at https://test.kuali.org/rice/rice-api-1.0-javadocs/.

Document content methods modify the properties of a document's content. A specific case is *addAttributeDefinition()*, where a *WorkflowAttribute* is used to generate attribute document content that will be appended to the existing document content. Another case is adding a searchable attribute definition with the *addSearchableDefinition()* method. More information on the various constructors, methods, and objects relating to the WorkflowDocument class is available in the Java documentation found at https://test.kuali.org/rice/rice-api-1.0-javadocs/org/kuali/rice/kew/service/WorkflowDocument.html.

WorkflowInfo

This class is the second client interface to KEW. The first time this object is initialized, the client configuration is accessed to determine how to connect to KEW. Methods invoked from this class can grab the routing header information based on the principleId, or return a set of Action Requests for a document that's in route based on the *routeHeaderId*, the *nodeName* and the *principalId*. More information on the various constructors, methods, and objects relating to the *WorkflowInfo* class is available in the Java documentation found at https://test.kuali.org/rice/rice-api-1.0-javadocs/org/kuali/rice/kew/service/WorkflowInfo.html.

Creating an eDocLite Application

Overview

eDocLite is a simple, form-based system that is built into Kuali Enterprise Workflow (KEW). It facilitates rapid development and implementation of simple documents and validation rules using XML. Use it for simple documents with simple route paths. You can integrate it with larger applications using a database layer post-processor component.

eDocLite uses an XSLT style sheet for custom presentation and XML to define form fields. The actual form display is called an EDL. This diagram shows how these objects are related:

Key Ideas:

- Rapid implementation and development solution for simpler documents

- Easily re-configured

- Easily manageable

- Entirely web-based from design/development and user perspectives

- No java code required for developments; only XML with optional javascript for client side editing (workflow handles execution)

- Some validation javascript is automatically generated like regular expression editing and 'required field checking'.

Figure 3.24. EDL Controller Chain
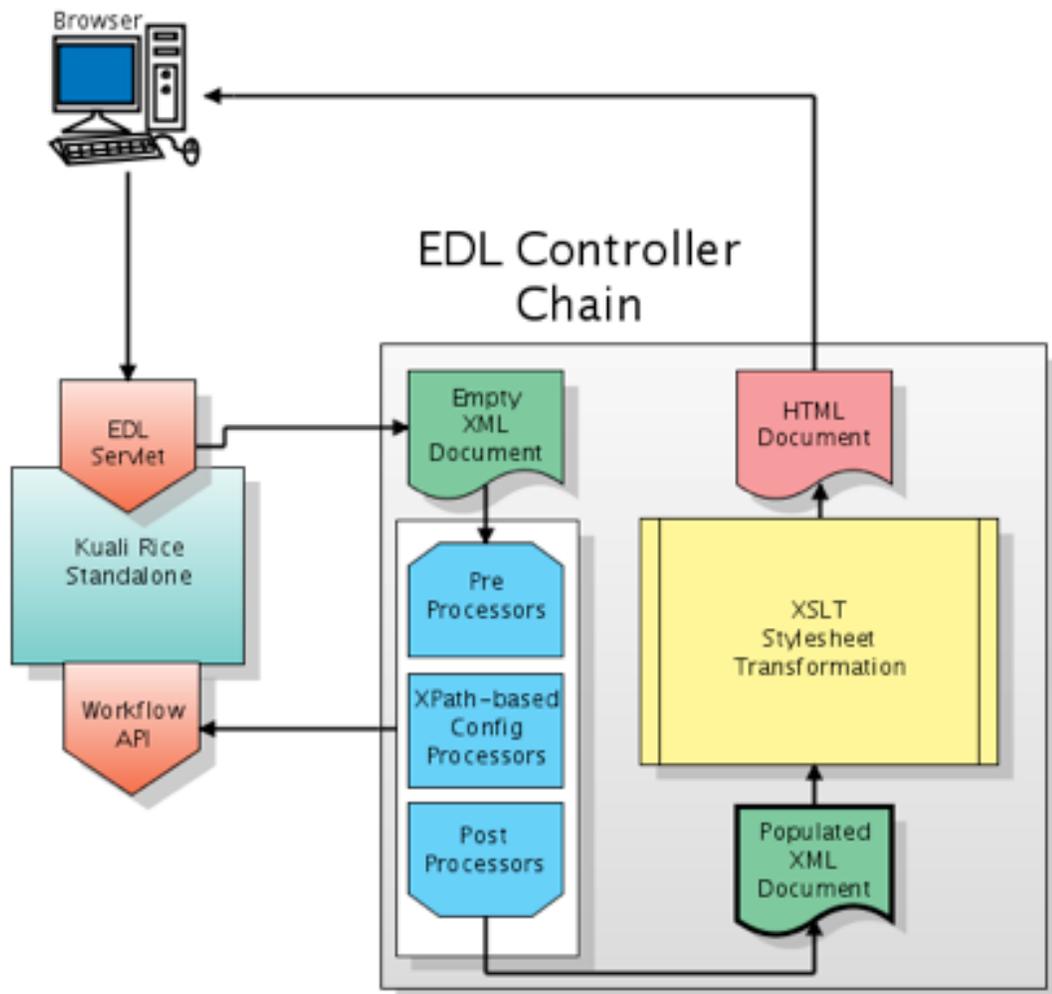
Components

Field Definitions

You need to define eDocLite fields to capture data that is passed to the server for storage.

Key Information about eDocLite fields:

• Save eDocLite data fields as key value pairs in two columns of a single database table.

• Use the xml element name as the key.

• You do not need to make any database-related changes when building eDocLite web applications.

• Store documents by document number.

• Make all field names unique within a document type.

The code example below focuses on the EDL section of the eDocLite form definition. The file Edoclite.xsd found in source under the impl/src/main/resources/schema/ directory describes the xml rules for this section.

Note that the first few lines proceeding *<edl name="eDoc.Example1.Form"* relate to namespace definitions. These are common across all eDocLites, so this guide does not discuss them.

In this example, any XML markup that has no value shown or that is not explained offers options that are not important at this time.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<data xmlns="ns:workflow" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="ns:workflow
 resource:WorkflowData">
    <edoclite xmlns="ns:workflow/eDocLite " xsi:schemaLocation="ns:workflow/eDocLite  resource:eDocLite ">

    <edl name="eDoc.Example1.Form" title="Example 1">
        <security />
        <createInstructions>** Questions with an asterisk are required.</createInstructions>
        <instructions>** Questions with an asterisk are required.</instructions>
        <validations />
        <attributes />
        <fieldDef name="userName" title="Full Name">
            <display>
                <type>text</type>
                <meta>
                    <name>size</name>
                    <value>40</value>
                </meta>
            </display>
            <validation required="true">
                <message>Please enter your full name</message>
            </validation>
        </fieldDef>
        <fieldDef name="rqstDate" title="Requested Date of Implementation:">
            <display>
                <type>text</type>
            </display>
            <validation required="true">
                <regex>^[0-1]?[0-9](/|-)[0-3]?[0-9](/|-)[1-2][0-9][0-9][0-9]$</regex>
                <message>Enter a valid date in the format mm/dd/yyyy.</message>
            </validation>
        </fieldDef>
        <fieldDef name="requestType" title="Request Type:">
            <display>
                <type>radio</type>
                <values title="New">New</values>
                <values title="Modification">Modification</values>
            </display>
            <validation required="true">
                <message>Please select a request type.</message>
            </validation>
        </fieldDef>
        <fieldDef attributeName="EDL.Campus.Example" name="campus" title="Campus:">
            <display>
                <type>select</type>
                <values title="IUB">IUB</values>
                <values title="IUPUI">IUPUI</values>
            </display>
            <validation required="true">
                <message>Please select a campus.</message>
            </validation>
        </fieldDef>
        <fieldDef name="description" title="Description of Request:">
            <display>
                <type>textarea</type>
                <meta>
                    <name>rows</name>
                    <value>5</value>
                </meta>
                <meta>
                    <name>cols</name>
                    <value>60</value>
                </meta>
```

```
                    <meta>
                        <name>wrap</name
                        <value>hard</value>
                    </meta>
                </display>
                <validation required="false" />
            </fieldDef>
            <fieldDef name="fundedBy" title="My research/sponsored program work is funded by NIH or NSF.">
                <display>
                    <type>checkbox</type>
                    <values title="My research/sponsored program work is funded by NIH or NSF.">nihnsf</values
                </display>
            </fieldDef>
            <fieldDef name="researchHumans" title="My research/sponsored program work involves human subjects.">
                <display>
                    <type>checkbox</type>
                    <values title="My research/sponsored program work involves human subjects.">humans</values>
                </display>
            </fieldDef>
        </edl>
        </eDocLite>
</data>
```

In the EDL XML file, field definition is embodied in the **edl** element. This element has a **name** attribute that is used to identify this file as a definition of an EDL form. It often has a **title** for display purposes.

Examination of this code shows that

- Individual fields have names, titles, and types. The types closely match html types.

- You can easily use simple validation attributes and sub-attributes to ensure that a field is entered if required and that an appropriate error message is presented if no value is provided by the web user.

- Regular expressions enhance the edit criteria without using custom JavaScript. (There are several ways that you can invoke custom JavaScript for a field, but they are not shown in this example.)

- An important field named campus has syntax that defines the value used to drive the routing destination. (In more complex documents, several fields are involved in making the routing decision.)

XSLT Style Sheet

The next section of the EDL XML file is the XSLT style sheet. It renders the EDL that the browser will present and contains logic to determine how data is rendered to the user.

A major workhorse of the XSLT code is contained in a style sheet library called **widgets.xml**. In the example below, it's included in the style sheet using an *xsl:include* directive.

Workflow Java classes have API's that offer methods that supply valuable information to the XSLT style sheet logic. XML allows you to interrogate the current value of *EDL*-defined fields, and it provides a variety of built-in functions.

Together, these helpers allow the eDocLite style sheet programmer to focus on rendering fields and titles using library (widget) calls and to perform necessary logic using the constructs built into the XML language(**if**, **choose…when**, etc.).

This is the area of eDocLite development that takes the longest and is the most tedious. Much of what the eDocLite style sheet programmer writes focuses on which fields and titles appear, in what order, to which users, and whether the fields are *readOnly*, *editable*, or *hidden*.

Below is the style sheet section of the EDL XML form for our example. It contains embedded comments.

```
<!-- widgets is simply more xslt that contains common functionality that greatly simplifies html rendering
It is somewhat complicated but does not require changes or full understanding unless enhancements are required.
  -->
```

```
<xsl:include href="widgets" />
<xsl:output indent="yes" method="html" omit-xml-declaration="yes" version="4.01" />


<!-- variables in the current version of xslt cannot be changed once set. Below they are set to various values
 often fed by java classes or to
values contained in workflow xml. Not all of these are used in this form but are shown because often they can
 be useful
The ones prefixed with my-class are methods that are exposed by workflow to eDocLite .-->
<xsl:variable name="actionable" select="/documentContent/documentState/actionable" />
<xsl:variable name="docHeaderId" select="/documentContent/documentState/docId" />
<xsl:variable name="editable" select="/documentContent/documentState/editable" />
<xsl:variable name="globalReadOnly" select="/documentContent/documentState/editable != 'true'" />
<xsl:variable name="docStatus" select="//documentState/workflowDocumentState/status" />
<xsl:variable name="isAtNodeInitiated" select="my-class:isAtNode($docHeaderId, 'Initiated')" />
<xsl:variable name="isPastInitiated" select="my-class:isNodeInPreviousNodeList('Initiated', $docHeaderId)" />
<xsl:variable name="isUserInitiator" select="my-class:isUserInitiator($docHeaderId)" />
<!-- <xsl:variable name="workflowUser" select="my-class:getWorkflowUser().authenticationUserId().id()" /> This
 has a unique implementation at IU -->
<xsl:param name="overrideMain" select="'true'" />


<!-- mainForm begins here. Execution of stylesheet begins here. It calls other templates which can call other
 templates.
Position of templates beyond this point do not matter. -->
<xsl:template name="mainForm">
    <html xmlns="">
        <head>
            <script language="javascript" />
            <xsl:call-template name="htmlHead" />
        </head>
        <body onload="onPageLoad()">
            <xsl:call-template name="errors" />
            <!-- the header is useful because it tells the user whether they are in 'Editing' mode or 'Read
 Only' mode. -->
            <xsl:call-template name="header" />
            <xsl:call-template name="instructions" />
            <xsl:variable name="formTarget" select="'eDocLite '" />
            <!-- validateOnSubmit is a javascript function (file: edoclite1.js) which supports edoclite forms
 and can be somewhat complicated
                but does not
 require modification unless enhancements are required. -->
            <form action="{$formTarget}" enctype="multipart/form-data" id="edoclite" method="post"
onsubmit="return validateOnSubmit(this)">
                <xsl:call-template name="hidden-params" />
                <xsl:call-template name="mainBody" />
                <xsl:call-template name="notes" />
                <br />
                <xsl:call-template name="buttons" />
                <br />
            </form>
            <xsl:call-template name="footer" />
        </body>
    </html>
</xsl:template>


<!-- mainBody template begins here. It calls other templates which can call other templates. Position of
 templates do not matter. -->
<xsl:template name="mainBody">
    <!-- to debug, or see values of previously created variables, one can use the following format.
        for example, uncomment the following line to see value of $docStatus. It will be rendered at the top
 of the main body form. -->
    <!-- $docStatus=<xsl:value-of select="$docStatus" /> -->
    <!-- rest of this all is within the form table -->
    <table xmlns="" align="center" border="0" cellpadding="0" cellspacing="0" class="bord-r-t" width="80%">
        <tr>
            <td align="left" border="3" class="thnormal" colspan="1">
<br />
<h3>
My Page
<br />
EDL EDoclite Example
</h3>
<br />
</td>
            <td align="center" border="3" class="thnormal" colspan="2">
<br />
<h2>eDocLite Example 1 Form</h2></td>
        </tr>
```

```
        <tr>
            <td class="headercell5" colspan="100%">
<b>User Information</b>
</td>
        </tr>
        <tr>
            <td class="thnormal">
                <xsl:call-template name="widget_render">
                    <xsl:with-param name="fieldName" select="'userName'" />
                    <xsl:with-param name="renderCmd" select="'title'" />
                </xsl:call-template>
                <font color="#ff0000">*</font>
            </td>
            <td class="datacell">
                <xsl:call-template name="widget_render">
                    <xsl:with-param name="fieldName" select="'userName'" />
                    <xsl:with-param name="renderCmd" select="'input'" />
                    <xsl:with-param name="readOnly" select="$isPastInitiated" />
                </xsl:call-template>
            </td>
        </tr>
        <tr>
            <td class="headercell5" colspan="100%">
<b>Other Information</b>
</td>
        </tr>
        <tr>
            <td class="thnormal">
                <xsl:call-template name="widget_render">
                    <xsl:with-param name="fieldName" select="'rqstDate'" />
                    <xsl:with-param name="renderCmd" select="'title'" />
                </xsl:call-template>
                <font color="#ff0000">*</font>
            </td>
            <td class="datacell">
                <xsl:call-template name="widget_render">
                    <xsl:with-param name="fieldName" select="'rqstDate'" />
                    <xsl:with-param name="renderCmd" select="'input'" />
                    <xsl:with-param name="readOnly" select="$isPastInitiated" />
                </xsl:call-template>
            </td>
        </tr>
        <tr>
            <td class="thnormal">
                <xsl:call-template name="widget_render">
                    <xsl:with-param name="fieldName" select="'campus'" />
                    <xsl:with-param name="renderCmd" select="'title'" />
                </xsl:call-template>
                <font color="#ff0000">*</font>
            </td>
            <td class="datacell">
                <xsl:call-template name="widget_render">
                    <xsl:with-param name="fieldName" select="'campus'" />
                    <xsl:with-param name="renderCmd" select="'input'" />
                    <xsl:with-param name="readOnly" select="$isPastInitiated" />
                </xsl:call-template>
            </td>
        </tr>
        <tr>
            <td class="thnormal">
                <xsl:call-template name="widget_render">
                    <xsl:with-param name="fieldName" select="'description'" />
                    <xsl:with-param name="renderCmd" select="'title'" />
                </xsl:call-template>
            </td>
            <td class="datacell">
                <xsl:call-template name="widget_render">
                    <xsl:with-param name="fieldName" select="'description'" />
                    <xsl:with-param name="renderCmd" select="'input'" />
                    <xsl:with-param name="readOnly" select="$isPastInitiated" />
                </xsl:call-template>
            </td>
        </tr>
        <tr>
            <td class="thnormal" colspan="2">
<b>(Check all that apply)</b>
</td>
```

```
            </tr>
            <tr>
                <td class="datacell" colspan="2">
                    <xsl:call-template name="widget_render">
                        <xsl:with-param name="fieldName" select="'fundedBy'" />
                        <xsl:with-param name="renderCmd" select="'input'" />
                        <xsl:with-param name="readOnly" select="$isPastInitiated" />
                    </xsl:call-template>
                    <br />
                    <xsl:call-template name="widget_render">
                        <xsl:with-param name="fieldName" select="'researchHumans'" />
                        <xsl:with-param name="renderCmd" select="'input'" />
                        <xsl:with-param name="readOnly" select="$isPastInitiated" />
                    </xsl:call-template>
                    <br />
                </td>
            </tr>
            <tr>
                <td class="headercell1" colspan="100%">
<b>Supporting Materials</b></td>
            </tr>
            <tr>
                <td class="thnormal" colspan="100%">Use the Create Note box below to attach supporting materials to
 your request. Notes may be added with or without attachments. Click the red 'save' button on the right.</td>
            </tr>
        </table>
        <br xmlns="" />
</xsl:template>
<xsl:template name="nbsp">
    <xsl:text disable-output-escaping="yes">&amp;nbsp;</xsl:text>
</xsl:template>
</xsl:stylesheet>
</style>
```

The beginning portion of this style sheet defines some XSL variables that are often useful to drive logic choices. For simplicity, this example uses very little logic.

The *isPastInitiated* variable drives whether a user-defined EDL field renders readOnly or not.

The *mainform* often serves to call some common widget templates that add canned functionality. The *mainform* then calls the *mainBody* template, which creates the html to render the EDL-defined fields. The *mainform* then (optional) calls the notes, buttons, and footer templates.

The majority of your programming effort goes into the *mainBody*, where calls to *widget_render* generate much of the field-specific title and value information. Various options can be passed into *widgets_render* to allow client events to be executed. The *mainBody* is usually one or more html tables and sometimes makes calls to programmer-defined sub-templates. The XSLT stylesheet generates the HTML rendered by the browser.

The main and repeating theme of the example involves calling *widget_render* with the title of an EDL field, followed by calling *widget_render* again with the input field. Widgets are a wrapper for XSLT stylesheets that offer the ability to create HTML. Paramters offer different ways to render HTML when making calls to widgets. Note that the variable value *$isPastInitiated* is passed as a parameter to widgets_render so that the html *readOnly* attribute is generated when the form is past the initiator's node.

Lazy importing of EDL Styles

You can configure Rice to lazily import an eDocLite style into the database on demand by setting a custom configuration parameter.

• Create a custom stylesheet file, e.g. myricestyle.xml containing a style with a unique name, e.g. "xyzAppStyle" and store it in a location that is locally accessible to your application server.

- Set a configuration parameter named **edl.style.<style-name>** with the value being a path to the file containing your style. Following the example above, you would name your parameter "edl.style.xyzAppStyle".

The stylesheet file could referenced could contain a full EDL, or be a standalone EDL style. On first use of that named style by an EDL, the file will be parsed and the named style will be imported into the database. The following example contains just an eDocLite XSL stylesheet:

```
<data xmlns="ns:workflow" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="ns:workflow
 resource:WorkflowData">
    <edoclite xmlns="ns:workflow/EDocLite" xsi:schemaLocation="ns:workflow/EDocLite resource:EDocLite">
        <style name="xyzAppStyle">
            <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xmlns:wf="http://
xml.apache.org/xalan/java/org.kuali.rice.kew.edoclite.WorkflowFunctions">
                <!-- your custom stylesheet -->
            </xsl:stylesheet>
        </style>
    </edoclite>

</data>
```

Note that in a default Rice installation (starting in version 1.0.2), the "widgets" style is lazily imported using this mechanism. In common-config-defaults.xml (which is located in the rice-impl jar), the following parameter is defined:

```
<param name="edl.style.widgets" override="false">classpath:org/kuali/rice/kew/edl/default-widgets.xml</param>
```

If you wanted to override that file, you could define your own parameter in your Rice XML configuration file using the above example as a template, but removing the override="false" attribute.

Document Type

A *document type* defines the workflow process for an eDocLite. You can create hierarchies where Child document types inherit attributes of their Parents. At some level, a document type specifies routing information. The document type definition for our first example follows. It contains routing information that describes the route paths possible for a document.

```
<?xml version="1.0" encoding="UTF-8"?>
<data xmlns="ns:workflow" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="ns:workflow
 resource:WorkflowData">
    <documentTypes xmlns="ns:workflow/DocumentType" xsi:schemaLocation="ns:workflow/DocumentType
 resource:DocumentType">
        <documentType>
            <name>eDoc.Example1Doctype</name>
            <parent>eDoc.Example1.ParentDoctype</parent>
            <description>eDoc.Example1 Request DocumentType</description>
            <label>eDoc.Example1 Request DocumentType</label>
            <postProcessorName>org.kuali.rice.kew.edl.EDocLitePostProcessor</postProcessorName>
            <superUserGroupName namespace="KUALI">eDoc.Example1.SuperUsers</superUserGroupName>
            <blanketApprovePolicy>NONE</blanketApprovePolicy>
            <defaultExceptionGroupName namespace="KUALI">eDoc.Example1.defaultExceptions</
defaultExceptionGroupName>
            <docHandler>${workflow.url}/EDocLite</docHandler>
            <active>true</active>
            <routingVersion>2</routingVersion>
            <routePaths>
                <routePath>
                    <start name="Initiated" nextNode="eDoc.Example1.Node1" />
                    <requests name="eDoc.Example1.Node1" />
                </routePath>
            </routePaths>
            <routeNodes>
                <start name="Initiated">
                    <activationType>P</activationType>
```

```
                    <mandatoryRoute>false</mandatoryRoute>
                    <finalApproval>false</finalApproval>
                </start>
                <requests name="eDoc.Example1.Node1">
                    <activationType>P</activationType>
                    <ruleTemplate>eDoc.Example1.Node1</ruleTemplate>
                    <mandatoryRoute>false</mandatoryRoute>
                    <finalApproval>false</finalApproval>
                </requests>
             </routeNodes>
        </documentType>
    </documentTypes>
</data>
```

The Parent element refers to a hierarchical order of the document types. Usually, you create one Root document type with limited but common information. Then, under that, you create more specific document types. In our example, there are only two levels.

The Root document type definition for our first example:

```
<?xml version="1.0" encoding="UTF-8"?>
<data xmlns="ns:workflow" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="ns:workflow
 resource:WorkflowData">
    <documentTypes xmlns="ns:workflow/DocumentType" xsi:schemaLocation="ns:workflow/DocumentType
 resource:DocumentType">
        <documentType>
            <name>eDoc.Example1.ParentDoctype</name>
            <description>eDoc.Example1 Parent Doctype</description>
            <label>eDoc.Example1 Parent Document</label>
            <postProcessorName>org.kuali.rice.kew.edl.EDocLitePostProcessor</postProcessorName>
            <superUserGroupName namespace="KUALI">eDoc.Example1.SuperUsers</superUserGroupName>
            <blanketApprovePolicy>NONE</blanketApprovePolicy>
            <docHandler>${workflow.url}/EDocLite</docHandler>
            <active>true</active>
            <routingVersion>2</routingVersion>
            <routePaths />
        </documentType>
    </documentTypes>
</data>
```

A Child document type can inherit most element values, although you must define certain element values, like *postProcessor*, for each Child document type.

A brief explanation of elements that are not intuitive is below. You can find additional element explanations by reading the Document Type Guide.

Parent DocType

postProcessorName - Use the default, as shown above, unless special processing is needed.

blanketApprovePolicy – When specified as NONE, this means that a user cannot click a single button that satisfies multiple levels of approval.

dochandler - Use the default, as shown above, so URLs are automatically unique in each environment, based on settings in the Application Constants (i.e., unique in each Test environment and unique again in Production).

active - Set this element to *false* to disable this feature.

routingVersion - Use the default, as shown above.

Child DocType

name - The name value must exactly match the value in the *EDL Association* document type element.

parent - The parent value must exactly match the name value of the parent document type.

superUserGroupName - A group of people who have special privileges that can be defined using the management service that's part of the KIM module.

defaultExceptionGroupName - A group of people who address a document of this type when it goes into Exception routing

routePaths and **routePath** - The initial elements that summarize the routing path the document will follow. In our example, an initiator fills out an eDocLite form. When the initiator submits that form, where it is routed depends on the value in the *Campus* field. There is only one destination node in our first example. The submitted form goes to either the IUB person or the IUPUI person, depending on the selection in the *Campus* field.

In most cases, a workgroup of people is the destination for an EDL form, not a single person. Workgroups are used as destinations because anyone in the workgroup can open the document, edit it, and click an **Action** button that routes the document to the next node. This prevents delays when someone is out of the office and a document awaits their action.

When the initiator submits the document, KEW adds that document to the Action List of the destination person or workgroup. The destination person or workgroup can then open the document, edit it (if any fields are available for editing), and click an **Action** button such as **Approve**, which routes the document onward. In our case, there is no further destination, so when the destination person or workgroup approves the document, the document becomes **Final** (it is finished). Some real-life examples have ten or more nodes for approvals or other actions. A document may bypass some of those nodes, depending on data placed into the form by previous participants.

routeNodes- Redefines the route path.

activationType

- **P** stands for *parallel* and is almost always used. This value makes more sense when considered from a *target node* perspective. From that perspective, it means that if a workgroup of people all received the document in their Action List, any one, in any order, can approve it. Once it is approved by anyone in the workgroup, it is routed to the next node, and KEW removes the document from the Action List of all the people in the workgroup. activationType

- **S** stands for *sequential* and is reserved for special cases where rules can specify that two or more people in a workgroup must take Action on a document, in a specific order, before KEW will route the document to the next node.

mandatoryRoute - Use *false* unless there is a special condition to solve. When this parameter is set to *true*, the document goes into exception routing if an approve request isn't generated by the ruleTemplate. This means that you are only expecting an *approve*, and nothing else.

finalApproval - Use false unless there is a special condition to solve. When this parm is set to true, the document goes into exception routing if approves are generated after this route node. This means this must be the last Action, or it will go into exception routing. (Be careful, because if this parameter is set to true and a user clicks a Return to Previous button, then the next action button clicked sends the document into exception handling.)

requests name= "..." - Defines the name of the node

ruleTemplate - A named entity type that helps define which routing rule fires. In our example, the *ruleTemplate* name is the same as the *request* name. These field values do NOT need to be the same. They are simply identifiers.

Rule Attributes

The RuleAttribute is a mechanism that can relate directly to an edl field. Most rule attributes are of the xml rule attribute type. This type uses an xpath statement which is used by the workflow engine to match to a rule that fires or does not fire.

In the below example, it can be seen that the edl defined field named 'campus' and its permissible values are defined. Then in the xpathexpression element says; when the value in the edl field named 'campus' matches the rule that contains 'IUB' the rule will fire. Or when the value in the edl field named 'campus' matches the rule that contains 'IUPUI' that rule will fire instead. Rules firing route a document to a person or a workgroup of people.

To make another rule attribute for a different field, clone this one, change all references to the field 'campus' to your different edl field name. Then cut and paste in the values section. Then in the edl definition, the new field must carry the extra syntax 'attributeName='. For example the eld definition for campus looks like this:

```
<fieldDef name="campus" title="Campus" workflowType="ALL">
```

Rule Routing

```
<?xml version="1.0" encoding="UTF-8"?>
<data xmlns="ns:workflow" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="ns:workflow
 resource:WorkflowData">
    <ruleAttributes xmlns="ns:workflow/RuleAttribute" xsi:schemaLocation="ns:workflow/RuleAttribute
 resource:RuleAttribute">
        <ruleAttribute>
            <name>EDL.Campus.Example</name>
            <className>org.kuali.rice.kew.rule.xmlrouting.StandardGenericXMLRuleAttribute</className>
            <label>EDL Campus Routing</label>
            <description>EDL School Routing</description>
            <type>RuleXmlAttribute</type>
            <routingConfig>
                <fieldDef name="campus" title="Campus" workflowType="ALL">
                    <display>
                        <type>select</type>
                        <values title="IUB">IUB</values>
                        <values title="IUPUI">IUPUI</values>
                    </display>
                    <validation required="false" />
                    <fieldEvaluation>
                        <xpathexpression>//campus = wf:ruledata('campus')</xpathexpression>
                    </fieldEvaluation>
                </fieldDef>
                <xmlDocumentContent>
                    <campus>%campus%</campus>
                </xmlDocumentContent>
            </routingConfig>
        </ruleAttribute>
    </ruleAttributes>

</data>
```

Rule attributes can have a different types such a searchable, but this type does not have to do with routing. Instead it relates to additional columns that are displayed in doc search for a particular doc type.

Ingestion Order

Many components can go in at any time, but it is advisable to follow a pattern to minimize the conflicts that can occur. A few pieces are co-dependent.

1. Basic Components:

2. Widgets.xml (If changed or not previously in the environment)

3. Kim Group(s)

4. Rule Attributes

5. Rule Template(s)

6. Parent Doctype (often no routing so data is more generic, but do put routing here if children will use common routing.)

7. Children Doctype(s) (routing defined here or on Parent)

8. EDL Form

9. Rule routing rule (Used if rules are created; explained later- 1 per parent doctype)

10.Rules (Create or Ingest)

11.Anything else - Like optional custom Email Stylesheet

Customizing Document Search

Each document carries an XML payload that describes metadata. You can specify pieces of that metadata to be indexed and searched on. This area focuses on the interface for searching through that data. For each **Document Search** page, you must setup the XML configuration files to define the search criteria and result fields.

Custom Search Screen

As an example of customizing a Document Search screen, we'll use a customized **Offer Request** screen:

Figure 3.25. Custom Search Screen:Offer Request Example

What are custom document search attributes?

Custom document search attributes are associated with a document type. They specify which pieces of document data will be made searchable for documents of that type. When you take action on a document in the workflow engine, a background process extracts the custom search attributes from the document and adds them to a database table where they can be queried as part of a custom document search. These

custom search attributes are defined and associated along with document types in WorkflowData XML files, and are added to Rice via the XML ingester. They are defined within (using XPath notation) /data/ruleAttributes/ruleAttribute tags, and are associated with specific document types within /data/documentTypes/documentType/attributes/attribute tags.

A custom search attribute's logic is defined in a Java class that implements the SearchAttribute interface. A SearchableAttribute implementation defines:

- What parts of the document content will be made searchable

- Which fields will be present in the document search interface

- Which columns will be shown in the search results

- What is considered valid user input for the custom search fields

There is a built in SearchAttribute implementation, SearchableXMLAttribute, that is highly configurable via XML and will meet most requirements. If there is need for more complex or specific behavior, a custom SearchAttribute implementation can be written and utilized as well.

DocumentSearchAttributes is much like **XMLRuleAttributes**, except that DocumentSearchAttributes is responsible for drawing input fields on the Document Search form and collecting data for the query, as opposed to analyzing data for routing evaluation (done by XMLRuleAttributes).

Hide Search Fields and Result Columns

In a search configuration, the **<visibility>** tag lets you configure search criteria to be included or excluded from the entry of search criteria or from the search results. You can use the **<visibility>** tag on all field(s) and column(s) in the Document Search results except for **Document Id** and **Route Log**, which must always be visible.

Hide a result column

```
<visibility>
    <column visible="false"/>

</visibility>
```

Hide a search field

```
<visibility>
    <field visible="false"/>
</visibility>
```

Field and column visibility based on workgroup membership

Use code like this in the XML file to display column(s) and field(s) based on the user's workgroup:

```
<visibility>
    <field>
        <isMemberOfWorkgroup>WorkflowAdmin</isMemberOfWorkgroup>
    </field>
    <column>
        <isMemberOfWorkgroup>WorkflowAdmin</isMemberOfWorkgroup>
    </column>
</visibility>
```

The example above indicates that the field and column only display for users who are a member of the workgroup, *WorkflowAdmin*.

Configure visibility for both field and column

A shortcut to configure the visibility for both fields and columns is the **<fieldAndColumn>** tag. A <fieldAndColumn> example:

```
<visibility>
    <fieldAndColumn>
        <isMemberOfWorkgroup>WorkflowAdmin</isMemberOfWorkgroup>
    </fieldAndColumn>
</visibility>
```

No field visibility

Declaring **<type>** as hidden is equivalent to setting visibility to false. An example of **<type>** and **<visibility>**, equivalent to a hidden field:

```
<searchingConfig>
    <fieldDef name="department" title="Department">
        <display>
            <type>text</type>
        </display>
        <visibility>
            <field visible="false"/>
        </visibility>
        <fieldEvaluation>
            <xpathexpression>normalize-space(substring-before(//department, ' '))</xpathexpression>
        </fieldEvaluation>
    </fieldDef>

</searchingConfig>


<!-- The above is equivalent to the following searching configuration -->

<searchingConfig>
    <fieldDef name="department" title="Department">
        <display>
            <type>hidden</type>
        </display>
        <fieldEvaluation>
            <xpathexpression>normalize-space(substring-before(//department, ' '))</xpathexpression
>
        </fieldEvaluation>
    </fieldDef>

</searchingConfig>
```

Configure Lookup Function

To make a lookupable available on the Document Search screen, you can use the **<quickfinder>** tag in the attribute definition. You can use the terms *quickfinder*, *lookup*, and *lookupable* interchangeably.

For example, you could set up an organizational hierarchic concept such as **Charts and Orgs** to implement a search. You could set up the code to perform this search using the **ChartOrgLookupableImpl** institutional plugin. This is an example of a standard lookupable component.

In the institutional plug-in, **ChartOrgLookupableImpl** is identified in the LookupableServiceExtension by the name of **ChartOrgLookupableImplservice**. **ChartOrgLookupableImpl** exposes two return parameters, which are:

- **Fin_coa_cd**: Represents the chart code

- **Org_cd**: Represents the organization code

An XML example of setting up a lookupable on the Document Search screen: **ChartOrgSearchAttribute.xml**

```xml
<ruleAttribute>
    <name>ChartOrgSearchAttribute</name>
    <className>org.kuali.rice.kew.docsearch.xml.StandardGenericXMLSearchableAttribute</className>
    <label>TestQuickfinderSearchAttribute</label>
    <description>TestQuickfinderSearchAttribute</description>
    <type>SearchableXmlAttribute</type>
    <searchingConfig>
        <fieldDef name="chart" title="Chart">
            <display>
                <type>text</type>
            </display>
            <quickfinder service="ChartOrgLookupableImplService" appliesTo="fin_coa_cd" draw="false"/>
            <fieldEvaluation>
                <xpathexpression>//chart</xpathexpression>
            </fieldEvaluation>
        </fieldDef>
        <fieldDef name="org" title="Organization">
            <display>
                <type>text</type>
            </display>
            <quickfinder service="ChartOrgLookupableImplService" appliesTo="org_cd" draw="true"/>
            <fieldEvaluation>
                <xpathexpression>//org</xpathexpression>
            </fieldEvaluation>
        </fieldDef>
        <xmlSearchContent>
            <chartOrg>
                <chart>%chart%</chart>
                <org>%org%</org>
            </chartOrg>
        </xmlSearchContent>
    </searchingConfig>
</ruleAttribute>
```

In the XML example above, there are two **<quickfinder>** tags representing the **Chart (fin_coa_cd)** and **Org (org_cd)** search. Notice the **draw** attribute for the **Org (org_cd)** search is set **true**. This means that a search icon will be displayed on the Document Search screen. Based on the XML code above, the final Document Search screen looks like this:

Figure 3.26. Custom Document Search: Department Example

Application Document Status

If the <validApplicationStatuses> configuration is specified in the document type definition, then setting the Document Type on the Document Search page will display a multi-select input titled "Application Document Status" that allows you to search by application statuses or status categories.

Figure 3.27. Document Search Screen: Application Document Status Example

In the figure above, **Pre-Submit** is a category of statuses containing **Initiated** and **Validated** which are individual statuses. Selecting **Pre-Submit** and searching will return identical results to selecting both **Initiated** and **Validated** and then searching.

Please see Document Type Policies: DOCUMENT_STATUS_POLICY for configuration details.

Define Keyword Search

XMLSearchableAttributeStdFloatRang is an XML searchable attribute that enhances the keyword search function. It provides multiple searchable elements for a user to select under the <searchingConfig> section. This example is the XMLSearchableAttributeStdFloatRang attribute in the default setting:

```
<ruleAttribute>
    <name>XMLSearchableAttributeStdFloatRange</name>
    <className>org.kuali.rice.kew.docsearch.xml.StandardGenericXMLSearchableAttribute</className>
    <label>XML Searchable attribute</label>
    <description>XML Searchable attribute</description>
    <type>SearchableXmlAttribute</type>
    <searchingConfig>
        <fieldDef name="testFloatKey" title="Float in the Water">
            <display>
                <type>text</type>
            </display>
            <searchDefinition dataType="float">
                <rangeDefinition inclusive="false">
                    <lower label="starting"/>
                    <upper label="ending"/>
                </rangeDefinition>
            </searchDefinition>
            <fieldEvaluation>
                <xpathexpression>//putWhateverWordsIwantInsideThisTag/testFloatKey/value</xpathexpression>
            </fieldEvaluation>
```

```
        </fieldDef>
        <xmlSearchContent>
            <putWhateverWordsIwantInsideThisTag>
                <testFloatKey>
                    <value>%testFloatKey%</value>
                </testFloatKey>
            </putWhateverWordsIwantInsideThisTag>
        </xmlSearchContent>
    </searchingConfig>
</ruleAttribute>
```

Caution

Cautions about the <searchingConfig> section:

1. **<searchDefinition>** identifies the search data type and search ranges.

2. **<rangeDefinition>** contains both the **<lower>** and **<upper>** elements that set up the parameters for the range search.

3. If you set the **<display><type>** tag to be date, then KEW automatically sets: **<searchDefinition dataType="datetime">**.

4. If the data type that you enter is not a *datetime*, then KEW sets all *datePicker* attributes to *false*.

5. Based on the **dataType** you enter, *datePicker* changes the default setting to either *true* or *false*.

6. To use a range search, you can either set **<searchDefinition rangeSearch="true">** or put the tag **<rangeDefinition>** under the **<searchDefinition>** tag. Either way, KEW will force a range search.

Custom Search Criteria Processing

URL Parameter Options

You can modify the search criteria and the display of the search screen by passing in URL parameters. Only use this method when the configuration desired is *preferable* and not *required*. If a particular piece of the search criteria is *required*, please see the section below titled, Using a Custom Search Criteria Processor.

Force the link to display the Detailed Search screen

Use the parameter **isAdvancedSearch** and set the value to **YES**.

Show or Hide All Criteria and/or the Workflow Header Bar

The default value of each of these parameters must be set to true to show both the criteria and the header bar.

• To hide the header bar, use the URL parameter **headerBarEnabled** and set the value to *false*.

• To hide the search criteria (including the buttons), use the URL parameter **searchCriteriaEnabled** and set the value to *false*.

Passing in Common Search Criteria Values

Common search criteria fields can be populated by supplying their values in the URL query parameters. For example, the following URL specifies a search on **KualiNotification** documents with initiator **user1**:

```
http://yourlocalip:8080/DocumentSearch.do?documentTypeName=KualiNotification&initiatorPrincipalName=user1
```

Common search criteria fields include:

- **documentTypeName** - the document type name

- **documentId** - the document id

- **initiatorPrincipalName** - the initiator principal name

- **dateCreated** - the document creation date

- **approverPrincipalName** - the approver principal name (use with advanced search)

- **viewerPrincipalName** - the viewer principal name (use with advanced search)

- **applicationDocumentId** - the application-supplied document id (use with advanced search)

- **dateApproved** - the approval date (use with advanced search)

- **dateLastModified** - the last modified date (use with advanced search)

- **dateFinalized** - the finalization date (use with advanced search)

- **title** - the document title(use with advanced search)

For a comprehensive list of search criteria fields, consult the

```
org.kuali.rice.kew.impl.document.search.DocumentSearchCriteriaBo
```

class.

The CURRENT_USER variable

In addition to literal field values, the 'CURRENT_USER' special token is dynamically replaced with an identifier for the currently authenticated user when the search is executed. This value can be supplied in any field (typically a field that takes a principal name or id). Several variants allow embedding different types of user ids:

- **CURRENT_USER**, **CURRENT_USER.principalName**, **CURRENT_USER.authenticationId**, **CURRENT_USER.a** - the current user principal name

- **CURRENT_USER.principalId**, **CURRENT_USER.workflowId**, **CURRENT_USER.w** - the current user principal id

- **CURRENT_USER.emplId**, **CURRENT_USER.e** - the current user employee id

Example:

```
http://yourlocalip:8080/DocumentSearch.do?
documentTypeName=KualiNotification&initiatorPrincipalName=CURRENT_USER
```

Passing in Searchable Attribute Values

Searchable attributes can be specified via URL parameters by prefixing the searchable attribute field name with **documentAttribute.**.

Here is an example using two **<fieldDef>** objects with names *firstname* and *lastname*:

```
http://yourlocalip:8080/DocumentSearch.do?documentAttribute.firstname=John&documentAttribute.lastname=Smith
```

Using a Custom Search Criteria Processor

The best way to do custom criteria processing is to implement a custom class that extends the class **org.kuali.rice.kew.docsearch.DocumentSearchCriteriaProcessor**. This file is ingested as a Workflow Attribute in KEW, using the **<type>** of **DocumentSearchCriteriaProcessorAttribute**. Once the Workflow Attribute is ingested, you can set the name value of the ingested attribute on one or more document type xml definitions in the Attributes section. A document type can only have one Criteria Processor Attribute.

Creating a child class of the **DocumentSearchCriteriaProcessor** class, a client can override various methods to modify the behavior of the search. The **DocumentSearchCriteriaProcessor** class can access the WorkflowUser object of the user performing the search. By having access to these objects, a custom processor class could implement dynamic hiding and showing of specific criteria fields based on ordinary user's data or search field data.

Show or Hide All Criteria and/or the Workflow Header Bar

Here are some helpful methods that you may override from the **DocumentSearchCriteriaProcessor** class file to hide or display full criteria (including buttons) and/or the header bar:

- **isHeaderBarDisplayed()** – If this function returns *false*, KEW hides the header bar on both the advanced and basic search screens (default return value is *true*).

- **isBasicSearchCriteriaDisplayed()** – If this function returns *false*, KEW hides criteria on the basic search screen (default return value is *true*).

- **isAdvancedSearchCriteriaDisplayed()** – If this function returns *false*, KEW hides the criteria on the advanced search screen (default return value is *true*).

Hiding Specific Fields or Criteria Using Field Key Values

The DocumentSearchCriteriaProcessor class has methods that allow classes to extend from it for basic field display. This is based on static string key values and makes it easier for clients to allow basic field display or to hide particular fields, whether they are searchable attributes or standard Document Search fields.

You may override these methods from the **DocumentSearchCriteriaProcessor** class to do specific field hiding by returning a list of string keys:

- **getGlobalHiddenFieldKeys()** – This function returns a list of keys (strings) for fields to be hidden on both the basic and advanced search screen.

- **getBasicSearchHiddenFieldKeys()** – This function returns a list of keys (strings) for fields to be hidden on the basic search screen.

- **getAdvancedSearchHiddenFieldKeys()** – This function returns a list of keys (strings) for fields to be hidden on the advanced search screen.

You can find the standard Document Search field key names in the class file **org.kuali.rice.kew.docsearch.DocumentSearchCriteriaProcessor**. They are constants prefixed by

the text **CRITERIA_KEY_**. For example, the static criteria key for the **Document Id** field is **DocumentSearchCriteriaProcessor.CRITERIA_KEY_DOCUMENT_ID**.

A client can also use searchable attribute **<fieldDef>** name values to hide fields in the same way that you use constants. If a particular searchable attribute **<fieldDef>** name exists in a list returned by one of the above **hidden field key** methods, the criteria processor class overrides the default behavior of that **<fieldDef>** searchable attribute for visibility.

Here is a general example of a custom criteria processor class that extends **StandardDocumentSearchCriteriaProcessor**:

```
public class CustomDocumentSearchCriteriaProcessor extends DocumentSearchCriteriaProcessor {

/**

  *  Always hide the header bar on all search screens

  */

    @Override
    public boolean isHeaderBarDisplayed() {
        return Boolean.FALSE;

    }


/**
  *  Always hide all criteria and buttons on the advanced search screen

  */

    @Override
    public Boolean isAdvancedSearchCriteriaDisplayed() {
        return Boolean.FALSE;
    }


/**
  *  Hide the Initiator Criteria field on both Basic and Advanced Search screens
  */
    @Override
    public List<String> getGlobalHiddenFieldKeys() {
        List<String> hiddenKeys = super.getGlobalHiddenFieldKeys();
        hiddenKeys.add(DocumentSearchCriteriaProcessor.CRITERIA_KEY_INITIATOR);
        return hiddenKeys;
    }


/**
  *  Hide the Document Title criteria field on the basic search screen
  *  Hide the searchable attribute field with name 'givenname' on the basic search screen
  */
    @Override
    public List<String> getBasicSearchHiddenFieldKeys() {
        List<String> hiddenKeys = super.getAdvancedSearchHiddenFieldKeys();
        hiddenKeys.add(DocumentSearchCriteriaProcessor.CRITERIA_KEY_DOCUMENT_TITLE);
        hiddenKeys.add("givenname");
        return hiddenKeys;
    }


/**
  *  Hide the Document Title criteria field on the advanced search screen
  *  Hide the searchable attribute field with name 'givenname' on the basic search screen
  */

    @Override
    public List<String> getAdvancedSearchHiddenFieldKeys() {
        List<String> hiddenKeys = super.getAdvancedSearchHiddenFieldKeys();
        hiddenKeys.add(DocumentSearchCriteriaProcessor.CRITERIA_KEY_DOCUMENT_TITLE);
        hiddenKeys.add("givenname");
```

```
        return hiddenKeys;

    }
}
```

Custom Search Generation

The best way to do custom search generation or processing is to implement a custom class that extends the class **org.kuali.rice.kew.impl.document.lookup.DocumentSearchGenerator**. This file is ingested as a Workflow Attribute in KEW using the **<type>** value of DocumentSearchGeneratorAttribute. Once the Workflow Attribute is ingested, the name value of the ingested attribute can be set on one or more document type xml definitions in the Attributes section. A Document Type can only have one Search Generator Attribute.

Using an extension of the **DocumentSearchGenerator** class, a client has access to override various methods to modify the behavior of the search. Also, the **DocumentSearchGenerator** class has helper methods that may be used to get the WorkflowUser object of the user performing the search.

Implementing a Custom Result Set Limit

To implement a custom result set limit, simply override the method **getDocumentSearchResultSetLimit()** from the **StandardDocumentSearchGenerator** class.

Custom Search Results

You can create a Custom Search Result table using an XML rule attribute of the type **DocumentSearchRsultXMLResultProcessorAttribute**.

The *standard* Search Result table:

Figure 3.28. Standard Doc Search Results Set

The Standard Search Result fields:

- Document Id

- Document Type

- Title

- Status

- Initiator

- Date Created

- Route log

The fields of **Document Id** and **Route Log** are always shown in the farthest left and right columns of the Search Result table. These fields cannot be hidden. You can add both columns a second time in the XML search result attributes if needed.

Custom XML Document Search Result Processor Attribute

An example of a custom XML result processor:

```
<ruleAttribute>
    <name>KualiContractsAndGrantsDocSearchResultProcessor</name>

    <className>org.kuali.rice.kew.docsearch.xml.DocumentSearchXMLResultProcessorImpl</className>
    <label>Contracts &amp; Grants Document Search Result Processor</label>
    <description>Attribute to allow for custom search results for Contracts &amp; Grants documents</
description>
    <type>DocumentSearchXMLResultProcessorAttribute</type>
    <searchResultConfig overrideSearchableAttributes="false" showStandardSearchFields="false">
        <column name="docTypeLabel" />
        <column name="docRouteStatusCodeDesc" />
        <column name="initiator" />
        <column name="dateCreated" />
    </searchResultConfig>

</ruleAttribute>
```

The result of the code displayed above is a Search Result table with these columns:

- Document Id

- Document Type

- Status

- Initiator

- Date Created

- Route Log

The key for the search result customization is focused on the elements and column tag(s) under the <searchResultConfig>.

Attributes that are included in the <searchResultConfig> tag:

- **overrideSearchableAttributes**: The indicator of whether to display the column *name* attributes defined by the searchAttribute fieldDef 'name's configured by setting the *true* or *false*

  - *true*: Display the <column> *name* attributes based on searchAttribute fieldDef names.

  - *false*: Display the *name* based on the <column> attribute.

- **showStandardSearchFields**: The indicator of whether to display the standard search fields by setting the value *true* or *false*.

  - *true*: Display the search result with the standard result fields; the *name* attribute of the <column> tag should match the values in the java file *DocumentSearchResult.java*.

  - *false*: Display the search result based on the custom result fields.

Attributes that can be added in a <column> tag:

- **Name**: The key for connecting the value of a particular attribute. For example, *routeHeaderId* equals *Document Id*. For more information about the attribute key, please refer to the Key reference table below.

- **Title**: The title of the field

- **Sortable**: The indicator of whether to sort the search result by setting the value *true* or *false*

    - *true*: Sort option for this column is enabled to sort either alphabetically or numerically depending on attribute type.

    - *false*: Sort option for this column is disabled.

For <column> with sortable = *true*, the field title becomes a link and when a user clicks the link, KEW sorts the results by that column.

An example of a custom ruleAttribute:

```
<ruleAttribute>
    <name>KualiContractsAndGrantsDocSearchResultProcessor</name>

    <className>org.kuali.rice.kew.docsearch.xml.DocumentSearchXMLResultProcessorImpl</className>
    <label>Contracts &amp; Grants Document Search Result Processor</label>
    <description>Attribute to allow for custom search results for Contracts &amp; Grants documents</
description>
    <type>DocumentSearchXMLResultProcessorAttribute</type>
    <searchResultConfig overrideSearchableAttributes="true" showStandardSearchFields="false">
        <column name="docTypeLabel" />
        <column name="docRouteStatusCodeDesc" />
        <column name="initiator" />
        <column name="dateCreated" />
        <column name="proposal_number" />
        <column name="chart" />
        <column name="organization" />
        <column name="proposal_award_status" />
        <column name="agency_report_name" />
    </searchResultConfig>
</ruleAttribute>
```

Table 3.17. Key Reference Table: Default field names and reference keys

Field	Key
Document Id	routeHeaderId
Document Type	docTypeLabel
Title	documentTitle
Status	docRouteStatusCodeDesc
Initiator	initiator
Date Created	dateCreated
Route Log	routeLog

Custom Document Search Result Processor Class File

You may also use a custom Document Search Result Processor by extending the class **org.kuali.rice.kew.docsearch.StandardDocumentSearchResultProcessor** and overriding individual methods.

Differences between SearchableAttribute and RuleAttribute

- SearchableAttribute does NOT have a **workflowType** attribute in the field tag.

- For SearchableAttribute, **xpathexpression** indicates the value's location in the document; it does not use **wf:ruledata('')**. For RuleAttribute, xpathexpression is a Boolean expression.

- SearchableAttribute uses xmlSearchContent instead of xmlDocumentContent; xmlDocumentContent is for RuleAttribute.

Document Security

Kuali Enterprise Workflow provides a declarative mechanism to facilitate Document-level security for these three screens:

- Document Search

- Route Log

- Doc Handler Redirection

Overview

1. You can create a security definition on a **Document Type**, which allows you to apply varying levels and types of security.

2. This definition is inheritable through the Document Type hierarchy.

3. If security is defined on a Document Type, rows for that Document Type that are returned from a search apply the security constraints and filter the row if the constraints fail.

4. Security constraints are evaluated against a document when its **Route Log** is accessed. If the security constraints fail, the user receives a *Not Authorized* message.

5. Security constraints are evaluated against a document when a **Doc Handler** link is clicked from either the **Action List** or **Document Search**. If the security constraints fail, the user receives a *Not Authorized* message.

Security Definition

You can define the security constraints in the Document Type XML. Here's a sample of the XML format:

```
<documentType>
    ....
    <security>
        <securityAttribute class="org.kuali.security.SecurityFilterAttribute"/>
        <securityAttribute name="TestSecurityAttribute"/>
        <initiator>true</initiator>
        <routeLogAuthenticated>true</routeLogAuthenticated>
        <searchableAttribute idType="emplid" name="emplid"/>
        <group>MyWorkgroup</group>
        <role allowed="true">FACULTY</role>
        <role allowed="true">STAFF</role>
    </security>
    ....

</documentType>
```

There is an implicit **OR** in the evaluation of these constraints. Thus, the definition above states that the authenticated user has access to the document if:

- The attribute **org.kuali.security.SecurityFilterAttribute** defines the user as having access **OR**

- The attribute defined in the system by the name **TestSecurityAttribute** defines the user as having access **OR**

- The user is the initiator of the document **OR**

- The user is on the Route Log of the document **OR**

- The user's EMPL ID is equal to the searchable attribute on the document with the key of *emplid* **OR**

- The user is a member of the *MyWorkgroup* workgroup **OR**

- The user has the FACULTY role **OR**

- The user has the STAFF role

\<initiator\>

Validates that the authenticated user is or isn't the initiator of the document.

\<routeLogAuthenticated\>

Validates that the authenticated user is or isn't *Route Log Authenticated*.

Route Log Authenticated means that one of these is true:

1. The user is the initiator of the document.

2. The user has taken action on the document.

3. The user has received a request for the document (either directly or as the member of a workgroup).

Route Log Authenticated checks for security but **does not** simulate or check future requests.

\<securityAttribute\>

Validates based on a custom-defined class. Class must have implemented the *SecurityAttribute* interface class. There are two methods of defining a security attribute:

- KEW Attribute Name: Specify an already-defined attribute (via KEW XML ingestion) using the XML attribute *name* .

  (Use of applicationId in a ruleAttribute specification sets the id of the application which contains the implementation of the security attribute.)

```
<documentType>
    ....

    <security>
        <securityAttribute name="TestSecurityAttribute"/>
    </security>
    ....

</documentType>
```

- Class Name: Define the fully qualified class name using the XML attribute class.

(Use of Class Name is limited to classes which are locally defined.)

```
<documentType>
    ....
    <security>
        <securityAttribute class="org.kuali.security.SecurityFilterAttribute"/>
    </security>
    ...
.
</documentType>
```

<searchableAttribute>

Validate that the authenticated User ID of the given idType is equivalent to the searchable attribute field with the given name.

The following id types are valid:

- emplid

- authenticationid

- uuid

- workflowid

<group>

Validate that the authenticated user is a member of the workgroup with the given name.

<role>

Validate that the authenticated user has the given role. The existence and names of these roles are determined by your setup in KEW. (You can create these roles when you implement *WebAuthenticationService*.) Typically, the roles mirror your organization structure.

For example, you may choose to expose these roles:

- STAFF

- FACULTY

- ALUMNI

- STUDENT

- FORMER-STUDENT

- APPLICANT

- ENROLLED

- ADMITTED

- PROSPECT

- GRADUATE

- UNDERGRADUATE

If the role is marked as **allowed=true**, than anyone with that role passes the security constraint. If the role is marked as **allowed=false**, then if the individual has the given disallowed role but none of the allowed roles, he or she fails the security check.

Order of Evaluation

The security constraints are evaluated in the following order. If any single constraint passes, it bypasses evaluating the remaining constraints.

1. Security attribute

2. Initiator

3. Role

4. Workgroup

5. Searchable attribute

6. Route log authenticated

Security - Warning Messages

These security scenarios generate security warning messages:

Document Search

- If no rows are filtered because of security, the user sees the search result without any warning message on the **Document Search** page.

- If rows are filtered because of security, a red warning message on top of the **Document Search** page shows how many rows were filtered. For example, "19 rows were filtered for security purposes."

- If the initial result set returns more than the search result threshold (500 rows), and rows in the set subsequently get filtered because of security, then a red warning message shows how many rows were returned and filtered. For example, "Too many results returned, displaying only the first 450. 50 rows were filtered for security purpose. Please refine your search."

Route Log and Doc Handler

- If the defined security constraints stop a user from viewing a document, a red warning message shows at the top of the page if they attempt to access the Route Log. For example, "You are not authorized to access this portion of the application."

Service Layer

In an out-of-the-box installation of KEW, Document Security is handled by org.kuali.rice.kew.doctype.DocumentSecurityServiceImpl, which implements the org.kuali.rice.kew.doctype.DocumentSecurityService service interface.

Action List Configuration Guide

Outbox Configuration

The **Outbox** is a standard feature on the **Action List** and is visible to the user in the UI by default. When the Outbox is turned on, users can access it from the *Outbox* hyperlink at the top of the Action List.

The Outbox is implemented by heavily leveraging existing Action List code. When an **Action Item** is deleted from the Action Item table as the result of a user action, the item is stored in the **KEW_OUT_BOX_ITM_T** table, using the **org.kuali.rice.kew.actionitem.OutboxItemActionListExtension** object. This object is an extension of the **ActionItemActionListExtension**. The separate object exists to provide a bean for OJB mapping.

The Workflow Preferences determine if the Outbox is visible and functioning for each user. The preference is called **Use Outbox**. In addition, you can configure the Outbox at the KEW level using the parameter tag:

```
<param name="actionlist.outbox">true</param>
```

When the Outbox is set to *false*, the preference for individual users to configure the Outbox is turned off. By default, the Outbox is set to *true* at the KEW level. You can turn the Outbox off (to hide it from users) by setting the property below to *false*:

```
<param name="actionlist.outbox.default.preference.on">false</param>
```

This provides backwards compatibility with applications that used earlier versions of KEW.

Note

Notes on the Outbox:

- Actions on saved documents are not displayed in the Outbox.

- The Outbox responds to all saved *Filters* and Action List *Preferences*.

- A unique instance of a document only exists in the Outbox. If a user has a document in the Outbox and that user takes action on the document, then the *original instance* of that document remains in the Outbox.

Email Customization

KEW provides default email template for Action List notification messages that are sent. However, it is also possible to customize this either globally or on a Document Type by Document Type basis.

There are two ways to customize Action List emails:

1. Configure a CustomEmailAttribute

2. Creating a custom XSLT Stylesheet

To accomplish this, you must write a *CustomEmailAttribute* and configure it on the appropriate *DocumentType*.

Configure a CustomEmailAttribute

The *CustomEmailAttribute* interface provides two methods for adding custom content to both the subject and the body.

```
public String getCustomEmailSubject();

public String getCustomEmailBody();
```

Note that each of these values is appended to the end of either the subject or the body of the email. The rest of the email still uses the standard email content.

Also, when implementing one of these components, the document is made available to you as a **RouteHeaderDTO** and the action request related to the notification is made available as an **ActionRequestDTO**.

Once you have implemented the CustomEmailAttribute, you need to make it available to the KEW engine (either deployed in a plugin or available on the classpath when running embedded KEW).

Document Type Configuration

Once you make the email attribute component available to KEW, you need to configure it on the Document Type.

First, define the attribute:

```xml
<data xmlns="ns:workflow" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="ns:workflow
 resource:WorkflowData">
    <ruleAttributes xmlns="ns:workflow/RuleAttribute" xsi:schemaLocation="ns:workflow/RuleAttribute
 resource:RuleAttribute">
        <ruleAttribute>
            <name>MyCustomEmailAttribute</name>
            <className>my.package.MyCustomEmailAttribute</className>
            <label>MyCustomEmailAttribute</label>
            <description>My Custom Email Attribute</description>
            <type>EmailAttribute</type>
        </ruleAttribute>
    </ruleAttributes>
</data>
```

Next, update the Document Type definition to include the attribute:

```xml
<data xmlns="ns:workflow" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="ns:workflow
 resource:WorkflowData">
    <documentTypes xmlns="ns:workflow/DocumentType" xsi:schemaLocation="ns:workflow/DocumentType
 resource:DocumentType">
        <documentType>
            <name>MyDocType</name>
            <label>My Document Type</label>
            <postProcessorName>...</postProcessorName>
            <attributes>
                <attribute>
                    <name>MyCustomEmailAttribute</name>
                </attribute>
            </attributes>
            <routePaths>
                ...
            </routePaths>
            <routeNodes>
                ...
            </routeNodes>
        </documentType>
    </documentTypes>
```

```
</data>
```

These should be ingested using the XML Ingester. See Importing Files to KEW for more information on using the XML Ingester.

Create a Custom XSLT Style Sheet

Global Email Customization

A more convenient way to customize email content declaratively is to replace the global email XSLT style sheet in Rice. Do this by ingesting an XSLT style sheet with the name *kew.email.style*. This style sheet should take input of this format for reminder emails:

```
<!-- root element sent depends on email content requested by the system -->
<immediateReminder|dailyReminder|weeklyReminder actionListUrl="url to ActionList" preferencesUrl="url to
 Preferences"
applicationEmailAddress="configured KEW email address" env="KEW environment string (dev/test/prd)">

    <user> <!-- the principal who received the request -->
        <name>...</name>
        <principalName>...</principalName>
        <principalId>...</principalId>
        <firstName>...</firstName>
        <lastName>...</lastName>
        <emailAddress>...</emailAddress>
        ...
    </user>
    <actionItem>
        <!-- one top-level actionItem element sent for each ActionItem; for immediate email reminders, there
will only ever be one; for daily and weekly reminders, there may be several -->

        <!-- custom subject content produced by the CustomEmailAttribute associated with the DocumentType of
this ActionItem, if any -->
        <customSubject>...</customSubject>

        <!-- custom body content produced by the CustomEmailAttribute associated with the DocumentType of this
ActionItem, if any -->
        <customBody>...</customBody>

        <actionItem> <!-- the actual ActionItem -->
            <principalId>...</principalId>
            <groupId>...</groupId>
            <routeHeaderId>...</routeHeaderId>
            <actionRequestId>...</actionRequestId>
            <docTitle>...</docTitle>
            <actionItemId>...</actionItemId>
            <roleName>...</roleName>
            <dateAssigned>...</dateAssigned>
            <actionRequestCd>...</actionRequestCd>
            <docHandlerURL>...</docHandlerURL>
            <recipientTypeCode>...</recipientTypeCode>
            <actionRequestLabel>...</actionRequestLabel>
            <delegationType>...</delegationType>
            <docName>...</docName>
            <docLabel>...</docLabel>
        </actionItem>
        <actionItemPerson> <!-- see "user" element at the top, simliar content -->
        ...
        </actionItemPerson>
        <actionItemPrincipalId>...</actionItemPrincipalId>
        <actionItemPrincipalName>...</actionItemPrincipalName>

        <doc> <!-- the RouteHeader associated with this ActionItem -->
            <routeHeaderId>...</routeHeaderId>
            <docTitle>...</docTitle>
            <docContent>...</docContent>
            <initiatorWorkflowId>...</initiatorWorkflowId>
            <documentTypeId>...</documentTypeId>
            <docRouteStatusLabel>...</docRouteStatusLabel>
            <docRouteStatus>...</docRouteStatus>
```

```
                    <createDate>...</createDate>
                    ...
            </doc>
            <docInitiator>
                <principalName>...</principalName>
                <principalId>...</principalId>
                <entityId>...</entityId>
            </docInitiator>
            <documentType> <!-- DocumentType -->
                <name>...</name>
                <label>...</label>
                <description>...</description>
                <serviceNamespace>...</serviceNamespace>
                <notificationFromAddress>...</notificationFromAddress>
                <docHandlerUrl>...</docHandlerUrl>
                <documentTypeId>...</documentTypeId>
                ...
        </actionItem>

</immediateReminder|dailyReminder|weeklyReminder>
```

This format is used for feedback emails:

```
<!-- feedback form -->
<feedback actionListUrl="url to ActionList" preferencesUrl="url to Preferences"
 applicationEmailAddress="configured KEW email address" env="KEW environment string (dev/test/prd)">
    <networkId>...</networkId>
    <lastName>...</lastName>
    <routeHeaderId>...</routeHeaderId>
    <documentType>...</documentType>
    <userEmail>...</userEmail>
    <phone>...</phone>
    <timeDate>...</timeDate>
    <edenCategory>...</edenCategory>
    <comments>
        ...
    </comments>
    <pageUrl>...</pageUrl>
    <firstName>...</firstName>
    <exception>...</exception>
    <userName>...</userName>
</feedback>
```

In both cases, the output generated by the style sheet must be like this:

```
<email>
    <subject>... subject here ...</subject>
    <body>... body here ...</body>
</email>
```

You must then upload the custom style sheet into the style service using the standard KEW XML ingestion mechanism:

```
<?xml version="1.0" encoding="UTF-8"?>
<data xmlns="ns:workflow" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="ns:workflow
 resource:WorkflowData">
<styles xmlns="ns:workflow/Style" xsi:schemaLocation="ns:workflow/Style resource:Style">
    <style name="kew.email.style">
 <!-- A custom global email reminder stylesheet -->
        <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
            <xsl:strip-space elements="*"/>
            <xsl:template match="immediateReminder">
                ...
            </xsl:template>
            <xsl:template match="dailyReminder">
                ...
            </xsl:template>
            <xsl:template match="weeklyReminder">
```

```
                ...
            </xsl:template>
            <xsl:template match="feedback">
                ...
            </xsl:template>
        </xsl:stylesheet>
    </style>
</styles>
```

The global style sheet should handle all email content requests. You can use the standard *include* syntax to import an existing style sheet that may implement defaults.

DocumentType-Specific Email Customization

You can also customize immediate reminder email content on a per-DocumentType basis. To do so, define a custom email style sheet name on the DocumentType definition:

```
...
    <documentType>
        <name>SomeDoc</name>
        <description>a document with customized reminder email</description>
            ...
        <emailStylesheet>somedoc.custom.email.style</emailStylesheet>
            ...
    </documentType>
...
```

Then, upload a corresponding style sheet with a matching name, as above.

Document Link

Document Link Features

KEW provides an option for linking documents and BOs that are functionally related. The link between related documents is created and removed in a double link double delete fashion, which means: when a link is added/deleted from 1 document to another document, a link in the reverse direction is also added/deleted, this feature will garuantee that searching for linked documents can be done from either side of the link. Using this option, client applications can link documents by using document link API.

Document Link API

Document link API is exposed to the client through WorkflowDocument interface, below is the summary of the api:

1. get all links to orgn doc

   public List<DocumentLinkDTO> getLinkedDocumentsByDocId(Long id) throws WorkflowException

2. get the link from orgn doc to a specifc doc

   public DocumentLinkDTO getLinkedDocument(DocumentLinkDTO docLinkVO) throws WorkflowException

3. add a link by id

   public void addLinkedDocument(DocumentLinkDTO docLinkVO) throws WorkflowException

4. remove all links to this doc as orgn doc

   public void removeLinkedDocuments(Long docId) throws WorkflowException

5. remove the link to the specific doc

   public void removeLinkedDocument(DocumentLinkDTO docLinkVO) throws WorkflowException

Document Link API Example

It is pretty straightforward to use this api, below are some examples:

1. To add a link

```
WorkflowDocument doc = new WorkflowDocument(…);

DocumentLinkDTO testDocLinkVO = new DocumentLinkDTO()
testDocLinkVO.setOrgnDocId(Long.valueOf(5000));

testDocLinkVO.setDestDocId(Long.valueOf(6000));
doc.addLinkedDocument(testDocLinkVO);
```

2. To retrieve all links to a document

```
List<DocumentLinkDTO> links2 = doc.getLinkedDocumentsByDocId(Long.valueOf(5000));
```

3. To remove a link

```
doc.removeLinkedDocument(testDocLinkVO);
```

Reporting Guide

Reporting Features

KEW provides various options for reporting on and simulation of routing scenarios. There is a GUI for performing these reporting functions as well as an API that you can use to run routing reports against the system.

The Routing Report Screen

From the Rice main menu there is a link to the **Routing Report** screen. From this set of screens you can enter various criteria for running reports against the routing engine. The output of this reporting is a simulated view of the **Route Log**, displaying the result of the report.

The Report APIs

The KEW client API also provides facilities for running reports against the routing engine. At the core of KEW is a **Simulation Engine** that is responsible for running these types of reports. The method for executing these reports is on the Workflow Info object that is part of the client API. The method is defined:

```
public DocumentDetailVO routingReport(ReportCriteriaVO reportCriteria) throws WorkflowException;
```

This method takes the report criteria and returns the results of the routing report.

Report Criteria

The routing report operates under two basic modes:

1. Reports that run against existing Documents

2. Reports that simulate a Document from a Document Type

In each these cases there are certain properties that you need to set on the ReportCriteriaVO to obtain the desired results.

In the first case, the report runs against a document that has already been created in the system. This document already has a Document Id and may be en route. Using this style of reporting, you can run simulations to determine where the document will go in future route nodes. For example, to run a simulation against an existing document to determine to whom it will route in the future, execute this code:

Routing Report against a Document

```
WorkflowInfo info = new WorkflowInfo();
RoutingReportCriteriaVO criteria = new ReportCriteriaVO(new Long(1234));
DocumentDetailVO results = info.routingReport(criteria);
// examine results...
```

This runs a report against the document with ID 1234, starting at the active nodes of the document and continuing to the terminal nodes of the document. The DocumentDetailVO will contain the Action Requests generated during the report simulation.

You can also stop the report at a particular node or once Rice generates a request for a particular user. For example, to stop the report simulation at a node or when Rice generates a certain user's request, configure the report criteria like this:

Terminate Report at Node or User

```
WorkflowInfo info = new WorkflowInfo();

RoutingReportCriteriaVO criteria = new ReportCriteriaVO(new Long(1234), "MyNodeName");
criteria.setTargetUsers(new UserIdVO[] { new NetworkIdVO("ewestfal") });

DocumentDetailVO results = info.routingReport(criteria);
```

This executes the report until it reaches a node named **MyNodeName** or a request is generated for user **ewestfal**.

In the second style of reporting, the report is run against an arbitrary Document Type and the simulation engine creates a temporary document against which to run the report. When setting up the report criteria for these scenarios, you usually populate the XML content of the document on the criteria (provided that the routing of that document evaluates the XML). Also, the criteria need to be configured with the valid node names (or rule templates) against which the report should be run. For example, to run a Document Type report, you can invoke the routing report this way:

Report against a Document Type

```
WorkflowInfo info = new WorkflowInfo();
```

```
RouteReportCriteriaVO criteria = new ReportCriteriaVO("MyDocumentType");

criteria.setXmlContent("<accountNumber>1234</accountNumber>");

criteria.setNodeNames(new String[] { "MyNodeName" });
DocumentDetailVO results = info.routingReport(criteria);
```

The code above simulates the generation of requests for **MyDocumentType** at the **MyNodeName** node with the XML given. This sort of reporting is especially useful if you simply need to determine what rules in the rule system will fire and generate action requests under a particular scenario.

As an alternative to specifying the node names, you can also specify rule template names. This is simply another way to target a specific node in the document. It searches the Document Type definition for nodes with the specified rule templates and then runs the report against those nodes. Currently, the rule template must exist on a node in the Document Type definition or an error will be thrown. In the case of our previous example, you could simply change the line that sets the node names on the criteria to:

```
criteria.setRuleTemplateNames(new String[] { "MyRuleTemplate" });
```

As above, this is primarily useful for determining who will have requests generated to them from the KEW rule system.

Interpreting Report Results

As we've seen, the object returned by the Routing Report is an instance of DocumentDetailVO. This object extends RouteHeaderVO and provides three more pieces of data along with it:

1. An array of ActionRequestVO objects representing the action requests on the document

2. An array of ActionTakenVO objects representing the actions that have been performed against the document

3. An array of RouteNodeInstanceVO objects that represent nodes in the document path

For reporting, the most important piece of data here is typically the ActionRequestVO objects. After running a report, this array contains the Action Requests that were generated as the result of the simulation. So, for example, in the example above where we run a document type report against the **MyRuleTemplate** rule template, this array contains all of the Action Requests that were generated to users or workgroups during the report simulation.

Workflow Plugin Guide

Overview

Kuali Enterprise Workflow (KEW) has a plugin framework that allows you to load code into the core system without requiring changes to the KEW source code or configuration. This framework provides:

• A custom class loading space

• Hot deploy and reload capabilities

• Participation in Workflow's JTA transactions

• An application plugin for installation of routing components

Application Plugin

Use an application plugin to deploy an application area's routing components into Workflow. These routing components might include:

- Rule attributes

- Searchable attributes

- Post processors

- Route modules

If these components require access to a data source, then the application plugin also configures the data source and allows it to participate in Workflow's JTA transactions.

In addition to routing components, the application plugin can also configure a plugin listener and a Resource Loader. The Resource Loader is responsible for loading resources (both Java classes and service implementations) from the plugin and providing them to the core system.

Application plugins are hot-deployable, so a restart of the server is not required when they are added or modified. The core system searches for plugins in a directory configured in the application configuration (see KEW Module Configuration).

Plugin Layout

You build the plugin as a set of files and directories. You then zip this structure and place it in the appropriate Workflow plugin space. For application plugins, this directory location is defined in the core system configuration.

The name of the zip file (minus the .zip extension) is used as the name of the plugin. The Plugin Loader only looks for files that end in .zip when determining whether to load and hot-deploy a plugin.

In general, application plugins can be named as desired. However, there is one reserved plugin name:

shared - A special plugin that provides a shared classloading space to all plugins (see Plugin Shared Space).

The directory structure of a plugin is similar to that of a web application. It should have this structure:

```
classes/

lib/
META-INF/
    workflow.xml
```

- **classes** - All java .class files that are used by the plugin should reside in this directory

- **lib** - All .jar library files that are used by the plugin should reside in this directory

- **META-INF** - The **workflow.xml** configuration file must reside in this directory

Plugin Configuration

Application plugins usually provide a subset of the functionality that an institutional plugin provides, since the institutional plugin can provide core service overrides.

The plugin framework provides two configuration points:

1. Plugin XML Configuration (described below)

2. Transaction and DataSource Configuration

Plugin XML Configuration

The XML configuration is defined in a file called workflow.xml. The format of this file is relatively simple. An example workflow.xml file:

```
<plug-in>
    <param name="my.param.1">abc</param>
    <param name="my.param.2">123</param>
    <listener>
        <listener-class>org.kuali.rice.core.ApplicationInitializeListener</listener-class>
    </listener>
    <resourceLoader class="my.ResourceLoader"/>
</plug-in>
```

We'll explain each of these elements in more detail below:

Plugin Parameters

The parameter configuration uses XML as the syntax. These parameters are placed into a configuration context for the plugin. The configuration inherits (and can override) values from the parent configurations. The configuration hierarchy is core -> institutional plugin -> application plugins.

A plugin can access its configuration using this code:

```
org.kuali.rice.Config config = org.kuali.rice.Core.getCurrentContextConfig();
```

Plugin Listeners

You can define one or more listeners that implement the interface **org.kuali.rice.kew.plugin.PluginListener**. These can be used to receive plugin lifecycle notifications from KEW.

The interface defines two methods to implement:

• Invoked when a plugin starts up

```
public void pluginInitialized(Plugin plugin);
```

• Invoked when a plugin shuts down

```
public void pluginDestroyed(Plugin plugin);
```

It is legal to define more than one plugin listener. Plugin listeners are started in the order in which they appear in the configuration file (and stopped in reverse order).

Resource Loader

A plugin can define an instance of **org.kuali.rice.resourceloader.ResourceLoader** to handle the loading of classes and services. When KEW attempts to load classes or locate services, it searches the institutional

plugin, then the core, then any application plugins. It does this by invoking the **getObject(..)** and **getService(...)** methods on the plugin's ResourceLoader.

If no ResourceLoader is defined in the plugin configuration, then the default implementation **org.kuali.rice.resourceloader.BaseResourceLoader** is used. The BaseResourceLoader lets you examine the plugin's classloader for objects when requested (such as post processors, attributes, etc.). This is sufficient for most application plugins.

For more information on configuring service overrides in a plugin, see the Overriding Services with a ResourceLoader section below.

Configuring an Extra Classpath

Sometimes it is desirable to be able to point in a plugin to classes or library directories outside of the plugin space. This can be particularly useful in development environments, where the plugin uses many of the same classes as the main application that is integrating with Workflow. In these scenarios, configuring an extra Classpath may mean you don't need to jar or copy many common class files.

To do this, specify these properties in your plugin's **workflow.xml** file:

1. **extra.classes.dir** - Path to an additional directory of **.class** files or resources to include in the plugin's classloader

2. **extra.lib.dir** - Path to an additional directory of **.jar** files to include in the plugin's classloader

The classloader then includes these classes and/or lib directories into its classloading space, in the same manner that it includes the standard **classes** and **lib** directories. The classloader always looks in the default locations first, and then defers to the extra classpath if it cannot locate the class or resource.

Transaction and DataSource Configuration

The easiest method to configure Datasources and Transactions is through the Spring Framework. Here is a snippet of Spring XML that shows how to wire up a Spring Transaction Manager inside of a plugin:

```
<bean id="userTransaction" class="org.kuali.rice.jta.UserTransactionFactoryBean" />

<bean id="jtaTransactionManager" class="org.kuali.rice.jta.TransactionManagerFactoryBean" />

<bean id="transactionManager" class="org.springframework.transaction.jta.JtaTransactionManager">
    <property name="userTransaction" ref="userTransaction" />
    <property name="transactionManager" ref="jtaTransactionManager" />
    <property name="defaultTimeout" value="${transaction.timeout}"/>
</bean>
```

The factory beans in the above XML will locate the **javax.trasaction.UserTransaction** and **java.transaction.TransactionManager**, which are configured in the core system. These can then be referenced and injected into other beans (such as the Spring JtaTransactionManager).

Once you configure the transaction manager, you also need to configure any DataSources you require. Here's an example of configuring a DataSource that participates in Atomikos JTA transactions (the default Transaction Manager distributed with Rice Standalone).

```
<bean id="myDataSource" class="com.atomikos.jdbc.nonxa.NonXADataSourceBean">
    <property name="uniqueResourceName" value="myDataSource"/>
    <property name="driverClassName" value="..."/>
    <property name="url" value="..."/>
    ...
</bean>
```

So, the application can access it's datasource by either injecting it into Spring services or by fetching it directly from the Spring context.

You can find more information on configuring Rice DataSources and TransactionManagers in Datasource and JTA Configuration.

OJB Configuration within a Plugin

If your plugin needs to use OJB, there are a few other configuration steps that you need to take. First, in your Spring file, add the following line to allow Spring to locate OJB and the JTA Transaction Manager:

```
<bean id="ojbConfigurer" class="org.kuali.rice.ojb.JtaOjbConfigurer">
    <property name="transactionManager" ref="jtaTransactionManager" />

</bean>
```

Next, for OJB to plug into Workflow's JTA transactions, you need to modify some settings in the plugin's **OJB.properties** file (or the equivalent):

```
PersistenceBrokerFactoryClass=org.apache.ojb.broker.core.PersistenceBrokerFactorySyncImpl
ImplementationClass=org.apache.ojb.odmg.ImplementationJTAImpl
OJBTxManagerClass=org.apache.ojb.odmg.JTATxManager
ConnectionFactoryClass=org.kuali.rice.ojb.RiceDataSourceConnectionFactory
JTATransactionManagerClass=org.kuali.rice.ojb.TransactionManagerFactory
```

The first three properties listed are part of the standard setup for using JTA with OJB. However, there are custom Rice implementations:

- **org.kuali.rice.ojb.RiceDataSourceConnectionFactory**

- **org.kuali.rice.ojb.TransactionManagerFactory**

- **org.kuali.rice.ojb.RiceDataSourceConnectionFactory**

This OJB ConnectionFactory searches your Spring Context for a bean with the same name as your **jcd-alias**. Here is what an OJB connection descriptor might look like inside of a Workflow plugin:

```
<jdbc-connection-descriptor
    jcd-alias="myDataSource"
    default-connection="true"
    platform="Oracle9i"
    jdbc-level="3.0"
    eager-release="false"
    batch-mode="false"
    useAutoCommit="0"
    ignoreAutoCommitExceptions="false">

    <sequence-manager className="org.apache.ojb.broker.util.sequence.SequenceManagerNextValImpl" />
    <object-cache class="org.apache.ojb.broker.cache.ObjectCachePerBrokerImpl"/>
</jdbc-connection-descriptor>
```

Notice that the **jcd-alias** attribute matches the name of the DataSource Spring bean defined in the example above.

Another important thing to notice in this configuration is that **useAutoCommit** is set to 0. This tells OJB not to change the auto commit status of the connection because it is being managed by JTA.

Finally, when your plugin needs to use OJB, you need to use this:

```
org.kuali.rice.ojb.TransactionManagerFactory
```

This provides OJB with the **javax.transaction.TransactionManager** that was injected into your JtaOjbConfigurer, as in the example above.

Overriding Services with a ResourceLoader

For a service override, you need to define a custom ResourceLoader implementation and configure it in your workflow.xml plugin configuration file. The org.kuali.rice.resourceloader.ResourceLoader interface defines this relative method:

```
public Object getService(javax.xml.namespace.QName qname);
```

When KEW is searching for services, it invokes this method on its plugins' ResourceLoader implementations. The service name is a qualified name (as indicated by the use of **javax.xml.namespace.QName**), but for services being located from the core, service names typically contain only a local part and no namespace.

The easiest way to implement a custom ResourceLoader is to create a class that extends from **org.kuali.rice.resourceloader.BaseResourceLoader** and just override the **getService(QName)** method. The BaseResourceLoader provides standard functionality for loading objects from ClassLoaders, among other things.

For example, if you want to override the User Service, you might implement this ResourceLoader:

```
public class MyResourceLoader extends BaseResourceLoader {
    public MyResourceLoader() {
        super(new QName("MyResourceLoader"));

    }

    @Override
    public Object getService(QName serviceName) {
        if ("enUserOptionsService".equals(serviceName.getLocalPart()) {
            // return your custom implementation of org.kuali.rice.kew.useroptions.UserOptionsService
        } else if (...) {
            ...
        } else if (...) {
            ...
        }
        return super.getService(serviceName);
    }
}
```

In the next section, we'll look at some of the services commonly overridden in an institutional plugin

Commonly Overridden Services

In theory, you can override any service defined in the **org/kuali/workflow/resources/ KewSpringBeans.xml** file in the Institutional Plugin. What follows is a list of the most commonly overridden services:

Table 3.18. Commonly Overridden Services

Service Name	Interface	Description
enUserOptionsService	org.kuali.rice.kew.useroptions.UserOptionsService	Provides User lookup and searching services

Service Name	Interface	Description
IdentityHelperService	org.kuali.rice.kew.identity.service.IdentityHelperService	Interfaces with KIM identity management services
enEmailService	org.kuali.rice.kew.mail.service.impl.DefaultEmailService	Provides email sending capabilities
enNotificationService	org.kuali.rice.ken.service.NotificationService	Provides callbacks for notifications within the system
enEncryptionService	org.kuali.rice.core.service.EncryptionService	Allows for pluggable encryption implementations

User Service

The Workflow core uses the UserService to resolve and search for users. The UserService could be as simple as a static set of users or as complex and integrated as a university-wide user system. Your institution may choose how to implement this, as long as you provide capabilities for the ID types that you intend to use. At the very least, implementations are required for the **WorkflowUserId** and **AuthenticationUserId** types (and their corresponding VO beans). All of the UserId types must be unique across the entire set of users.

The **WorkflowUserId** is typically associated with a unique numerical sequence value and the **AuthenticationUserId** is typically the username or network ID of the user.

The default UserService implementation provides a persistent user store that allows you to create and edit users through the GUI. It also caches users for improved performance and implements an XML import for mass user import. Institutions usually override the default user service with an implementation that integrates with their own user repository.

IdentityHelper Service

The IdentityHelper service helps to interact with the KIM identity management services in the system. IdentityHelpers are identified in one of two ways:

1. **PrincipalId** - A numerical identifier for a KIM principal

2. **Group** – An object associated with a group of principal users numerical identifier assigned to a Workgroup

Both of these object variables are implemented in KEW in the IdentityHelperServiceImpl file.

Email Service

The Email service is used to send emails from KEW. You can configure the default implementation when you configure KEW (see KEW Configuration). However, if more custom configuration is needed, then you can override the service in the plugin.

For example, you could override this service if you need to make a secure and authorized SSL connection with an SMTP server because of security policies.

Notification Service

The Notification service is responsible for notifying users when they receive Action Items in their Action List.

The default implementation simply sends an email (using the EmailService) to the user according to the individual user's preferences. A custom implementation might also notify other (external) systems in addition to sending the email.

Encryption Service

The Encryption service is responsible for encrypting document content.

The default implementation uses DES to encrypt the document content. If the **encryption.key** configuration parameter is set as a *Base64* encoded value, then the document content is encrypted using that key. If it is not set, then document content will not be encrypted and will be stored in the database in plain text.

Plugin Shared Space

All plugins also load certain classes from a shared space. The shared space contains certain classes that link with certain libraries that might exist in each application or institutional plugin's classloader (such as OJB and Spring). Current classes that Workflow publishes in the **shared** space are those in the shared module of the Rice project (**rice-shared-version.jar**). This is important because some of these classes link with libraries like Spring or OJB and since the plugin needs its own copy of these libraries, it needs to ensure that it doesn't retrieve these classes from any classloader but it's own.

KEW Usage of the Kuali Service Bus

General Usage

The Kuali Enterprise Workflow engine makes use of both synchronous service endpoints and asynchronous messaging features from the Kuali Service Bus.

Most asynchronous processing that KEW does is implemented using asynchronous messaging on the service bus. This includes:

1. Workflow engine processing

2. Blanket approval orchestration

3. Action processing for actions taken directly from the Action List

4. Re-resolving actions requests resulting from a responsibility change

5. Sending email reminders

6. Distributed cache flush notifications

In each of these cases, there exists a service that processes asynchronous messages and performs the appropriate actions for each of these functions.

In terms of synchronous services, Kuali Enterprise Workflow publishes two different types of services. One is used for performing workflow document actions (such as creating, approving, disapproving, etc.). The other is used to perform various query or read-only operations against the workflow system.

Implications of using "Synchronous" KSB messaging with KEW

For general information on synchronous messaging and its implications in the KSB, please read "Implications of synchronous vs. asynchronous Message Deliver" in the KSB technical reference guide.

In terms of Kuali Enterprise Workflow, the usage of synchronous messing means that operations like workflow engine processing will happen immediately and synchronously at the time it's invoked.

The main implication here besides what is listed in the KSB documentation is that, since message exception handling isn't implemented, exception routing does not work when using synchronous KSB messaging.

This means that if this messaging model is being used in a batch job, or similar type of program, routing exceptions will need to be manually caught. If it's desired to place a document into exception status from here, there are methods on the KEW APIs to do this manually.

Chapter 4. KIM

Terminology

Principal

A principal represents an entity that can authenticate. In essence, you can think of a principal as an "account" or as an entity's authentication credentials. A principal has an ID that is used to uniquely identify it. It also has a name that represents the principal's username and is typically what is entered when authenticating. All principals are associated with one and only one entity.

Entity

An entity represents a person or system. Additionally, other "types" of entities can be defined in KIM. Information like name, phone number, etc. is associated with an entity. While an entity will typically have a single principal associated with it, it is possible for an entity to have more than one principal or even no principals at all (in the case where the entity does not actually authenticate).

Entities have numerous attributes associated with them, including:

- Names

- Addresses

- Phone Numbers

- Email Addresses

- Entity Type

- Affiliations

- Employment Information

- External Identifiers

- Privacy Preferences

Group

A group is a collection of principals. You can create a group using both direct principal assignment and nested group membership. All groups are uniquely identified by a namespace code plus a name. A principal or group is a "member" of a group if it is either directly assigned to the group or indirectly assigned (through a nested group membership). A principal or group is a "direct" member of another group only if it is directly assigned as a member of the group, and not through a nested group assignment.

Permission

A permission is the ability to perform an action. All permissions have a permission template. Both permissions and permission templates are uniquely identified by a namespace code plus a name. The permission template defines the coarse-grained permission and specifies what additional permission details

need to be collected on permissions that use that template. For example, a permission template might have a name of "Initiate Document," which requires a permission detail specifying the document type that can be initiated. A permission created from the "Initiate Document" template would define the name of the specific Document Type that can be initiated as a permission detail.

The **isAuthorized** and **isAuthorizedByTemplateName** operations on the **PermissionService** are used to execute authorization checks for a principal against a permission. Permissions are always assigned to roles (never directly to a principal or group). A particular principal will be authorized for a given permission if the principal is assigned to a role that has been granted the permission.

Responsibility

A responsibility represents an action that a principal is requested to take. This is used for defining workflow actions (such as approve, acknowledge, FYI) for which the principal is responsible. Responsibilities form the basis of the workflow engine routing process.

A responsibility is very similar to a permission in a couple of ways. First, responsibilities are always granted to a role, never assigned directly to a principal or group. Furthermore, similar to permissions, a role has a responsibility template. The responsibility template specifies what additional responsibility details need to be defined when the responsibility is created.

Role

You grant permissions and responsibilities to roles. Roles have a membership consisting of principals, groups, and/or other roles. As a member of a role, the associated principal has all permissions and responsibilities that have been granted to that role.

You can specify a qualification to any membership assignment on the role, which is extra information about that particular member of the role. For example, a person may have the role of "Dean" but that can be further qualified by the school they are the dean of, such as "Computer Science." You can pass qualifications as part of authorization checks to restrict the subset of roles to check.

Reference Information

There are several collections of reference information managed within KIM:

- Address type

- Affiliation type

- Citizenship status

- Email type

- Employment status

- Employment type

- Entity name type

- Entity type

- External identifier type

- Phone number type

Configuration Parameters

Table 4.1. KIM Configuration Parameters

Configuration Parameter	Description	Default value
kim.mode	The mode that KIM will run in; choices are "LOCAL", "EMBEDDED", or "REMOTE".	LOCAL
kim.soapExposedService.jaxws.security	Determines if KIM services published on the service bus will be secured	true
kim.url	The base URL of KIM services and pages.	${application.url}/kim

Services

KIM provides several service APIs with which client applications should interact. These are:

- **org.kuali.rice.kim.api.role.RoleService**

- **org.kuali.rice.kim.api.group.GroupService**

- **org.kuali.rice.kim.api.identity.IdentityService**

- **org.kuali.rice.kim.permission.PermissionService**

- **org.kuali.rice.kim.responsibility.ResponsibilityService**

- **org.kuali.rice.kim.service.PersonService**

These services act as client-side facades to the underlying KIM data and provide important features such as caching.

In the next few sections we will look in-depth at these services. However, for more details, please see the javadocs for these services and the services they delegate to.

Using the Services

All KIM clients should retrieve service instances using the KIM service locator class **KimApiServiceLocator**. This class contains static methods to retrieve the appropriate Spring bean for the service. An example of retrieving the **IdentityService** service is:

```
IdentityService idmSvc = KimApiServiceLocator.getIdentityService();
```

You would use a similar mechanism for retrieving references to the other KIM services.

IdentityService

The **IdentityService** is one of the services the client applications will interact with most frequently.

The **IdentityService** contains service methods that allow for the retrieval, creation, and upating of entity information.

Additionally, it also provides caching for the retrieval methods to help increase the performance of service calls for the client application.

Retrieving Principal Information

To retrieve the principal ID for a user, use the **getPrincipalByPrincipalName** method:

```
Principal info = identityService.getPrincipalByPrincipalName(principalName);
```

Note that KIM, by default, stores principal names in lower case; the PRNCPL_NM column of KRIM_PRNCPL_T must store values in lower case. If your institution's existing identity systems do not handle lowercase principal names, then there are three points to override that setting:

1. **org.kuali.rice.kim.impl.identity.IdentityServiceImpl** method **getPrincipalByPrincipalName** lowercases the principal name sent in; depending on how principals were integrated into the system it may not need to. Note that **IdentityServiceImpl** method **getPrincipalByPrincipalNameAndPassword** does not lowercase the principal name automatically.

2. **org.kuali.rice.kim.lookup.PersonLookableHelperServiceImpl** method **getSearchResults** also automatically lowercases any principal name sent in; that behavior may also need to be changed

3. Finally, the file **{Rice home}/impl/src/main/resources/org/kuali/rice/kim/bo/datadictionary/ KimBaseBeans.xml** hold the data dictionary attribute templates for principal name as **KimBaseBeans-principalName**. The **forceUppercase** attribute is set to false by default, but perhaps should be overridden to true, to force uppercase principal names.

Once these three points have been overridden, you'll be able to use uppercase principal names.

Retrieving Entity Default Information

To retrieve the default information for an entity, use one of the getEntityDefaultInfo methods:

```
EntityDefault infoByEntityId = identityService.getEntityDefault(entityId);

EntityDefault infoByPrincipalId = identityService.getEntityDefaultByPrincipalId(principalId);
```

Retrieving Reference Information

To retrieve information about a type or status code, use the getter for that type.

Types in KIM are:

- Address type

- Affiliation type

- Citizenship status

- Email type

- Employment status

- Employment type

- Entity name type

- Entity type

- External identifier type

- Phone type

For instance, to retrieve information on an address type code:

```
CodedAttribute addressType = identityService.getAddressType(code);
```

GroupService

Retrieving Group Membership Information

To retrieve a list of all groups in which a particular user is a member, use the **getGroupsForPrincipal** method:

```
List<Group> groups = groupService.getGroupsByPrincipalId(principalId);
```

To determine if a user is a member of a particular group, use the **isMemberOfGroup** method:

```
if (groupService.isMemberOfGroup(principalId, groupId)) {
    // Do something special
}
```

To get a list of all members of a group, use the **getMemberPrincipalIds** method:

```
List<String> members = groupService.getMemberPrincipalIds(groupId);
```

Retrieving Group Information

To retrieve information about a group, use the **getGroup** or **getGroupByNamespaceCodeAndName** methods, depending on whether you know the group's ID or name:

```
Group info = groupService.getGroup(groupId);
Group info = groupService.getGroupByNamespaceCodeAndName(namespaceCode, groupName);
```

PermissionService

Checking Permission

To determine if a user has been granted a permission, without considering role qualifications, use the **hasPermission** method:

```
if (permissionService.hasPermission(principalId, namespaceCode, permissionName)) {
    // Do the action
}
```

To determine if a user has been granted a permission, use the **isAuthorized** method:

```
if (permissionService.isAuthorized(principalId, namespaceCode, permissionName, qualification)) {
```

```
    // Do the action
}
```

Retrieving Permission Information

To retrieve a list of principals granted a permission (including any delegates), use the **getPermissionAssignees** method:

```
List<Assignee> people = permissionService.getPermissionAssignees(namespaceCode,
 permissionName, qualification);
```

To retrieve a list of permissions granted to a principal, use the **getAuthorizedPermissions** method:

```
List<Permission> perms = permissionService.getAuthorizedPermissions(principalId,
 namespaceCode, permissionName, qualification);
```

ResponsibilityService

Checking Responsibility

To determine if a user has a responsibility, use the **hasResponsibility** method:

```
if (responsibilityService.hasResponsibility(principalId, namespaceCode, responsibilityName, qualification)) {
    // Do the action

}
```

Retrieving Responsibility Information

To retrieve a list of roles associated with a responsibility, use the **getRoleIdsForResponsibility** method:

```
List<String> roleIds = responsibilityService.getRoleIdsForResponsibility(responsibilityId);
```

AuthenticationService

Checking Authentication

The **AuthenticationService** is somewhat different than the other services. The **AuthenticationService** is not typically deployed remotely (unlike the **IdentityService**, **GroupService**, etc.).

Instead, the role of this service is simply to extract the authenticated user's principal name from the **HttpServletRequest** and inform the client-side development framework (typically, the KNS) about this information. KIM itself does not implement full authentication services, but rather relies on other implementations (such as CAS or Shibboleth) to provide this functionality.

The client application can then establish a local session to store the information about the principal that authenticated. This will typically be used in subsequent calls to the KIM services, such as making authorization checks for the principal.

The reference implementation of the **AuthenticationService** simply extracts the REMOTE_USER parameter from the request and presents that as the principal name. This is often sufficient for many

authentication providers that are available. However, if necessary this reference implementation can be overridden.

There is only a single method on the **IdentityManagementService** related to authentication.

```
String principalName = authenticationService.getPrincipalName(request);
```

RoleService

In KIM, Roles are used as a way to associate principals, groups and other roles with permissions and responsibilities. It is therefore not a common or recommended practice to query for whether or not a principal is a member of a Role for the purposes of logic in a client application. It is recommended to use permissions and the **isAuthorized** check to perform this sort of logic.

However, in some cases, querying for this information may be desirable. Or, in even more common cases, one may want to use an API to add or remove members from a Role. These kinds of operations are the responsibility of the **RoleManagementService**. Like the **IdentityManagementService**, this service is a façade which provides caching and delegates to underlying services. Specifically, it delegates to:

- RoleService

Checking Role Assignment

To determine if a role is assigned to a principal, use the **principalHasRole** method:

```
if (roleService.principalHasRole(principalId, roleIds, qualifications)) {
    // Do something
}
```

Retrieving Role Information

To retrieve information on a role, use the **getRole** or **getRoleByName** method:

```
Role info = roleService.getRole(roleId);
Role info = roleService.getRoleByNamespaceCodeAndName(namespaceCode, roleName);
```

To retrieve the list of principal IDs assigned to a role, use the **getRoleMemberPrincipalIds** method:

```
Collection<String> principals = roleService.getRoleMemberPrincipalIds(namespaceCode, roleName, qualifications);
```

Updating Role Membership

To assign a principal to a role, use the **assignPrincipalToRole** method:

```
roleService.assignPrincipalToRole(principalId, namespaceCode, roleName, qualifications);
```

To remove a principal from a role, use the **removePrincipalFromRole** method:

```
roleService.removePrincipalFromRole(principalId, namespaceCode, roleName, qualifications);
```

Person Service

The **PersonService** is used to aggregate **Entity** and **Principal** data into a data structure called a **Person**. A person is essentially a flattened collection of the various attributes on an entity (name, address, principal id, principal name, etc). This is intended to allow client applications to more easily interact with the data in the underlying KIM data model for entities and principals.

Retrieving Personal Information

To retrieve information on a person by principal ID, use the **getPerson** method:

```
Person person = perSvc.getPerson(principalId);
```

To retrieve information on a person by principal name, use the **getPersonByPrincipalName** method:

```
Person person = perSvc.getPersonByPrincipalName(principalName);
```

In order to search for people by a given set of criteria you can use the **findPeople** method:

```
List<Person> people = perSvc.findPeople(criteria);
```

In this case, criteria is a **java.util.Map<String, String>** which contains key-value pairs. The key is the name of the Person property to search on, while the value is the value to search for.

KimTypeService Callbacks

Implementing Custom KIM Types

KIM uses the concept of "types" to define additional attributes for it's various objects (such as groups, roles, permissions, etc.) and to affect their behavior.

All custom type services must implement a sub-interface of `org.kuali.rice.kim.framework.type.KimTypeService` based on the kind of custom type being created and the KIM objects it will be related to. The current type services supported by KIM are as follows:

- `GroupTypeService`

- `RoleTypeService`

- `PermissionTypeService`

- `ResponsibilityTypeService`

- `DelegationTypeService`

In addition to the interfaces provided above, KIM provides a standard set of implementations of each of these which can be extended by your application in order to inherit standard default behavior (including integration with the KNS Data Dictionary for reading and defining custom attributes). More detailed

information about these base classes can be found in the KIM javadocs. Your custom type service class should extend the appropriate subclass and only override the methods necessary to implement your custom behavior. Use the methods in these classes as the basis for your custom code.

For example, you might define a custom `PermissionTypeService` by extending `org.kuali.rice.kns.kim.permission.PermissionTypeServiceBase` as follows:

```
import org.kuali.rice.kns.kim.permission.PermissionTypeServiceBase;

public class MyPermissionTypeService extends PermissionTypeServiceBase {

    @Override
    protected boolean performMatch(Map<String, String> inputMap, Map<String, String> storedMap) {
        if (some_condition_is_true) {
            // perform custom matching logic
            ...
        } else {
            return super.performMatch(inputMap, storedMap); // execute the default logic from base class
        }
    }

}
```

Detailed documentation on the specific methods which can be implemented on `KimTypeService` and it's various sub-interfaces can be found in the KIM javadocs.

Configuring Custom KIM Types

Groups, Roles, Permissions, Responsibilities, and Delegations can all have custom types in KIM. These custom types can be mapped back to the KIM type services that you create. In order to do this, there are a few things you must do:

- Register the KIM Type which points to your custom type service

- Update any of the "typed" KIM objects that you want to point to your new KIM type

- Publish your KIM type service so that it is available on the Kuali Service Bus and the Rice resource loader framework

Currently, there is no way to register a new KIM Type without updating the KIM database using SQL. Fortunately, this is a fairly simple thing to do. The database table storing KIM Types is named `KRIM_TYP_T`. An example of how to insert a new KIM Type into this table in Oracle is below:

```
INSERT INTO KRIM_TYP_T (
    KIM_TYP_ID,
    NMSPC_CD,
    NM,
    SRVC_NM,
    OBJ_ID)
VALUES (
    KRIM_TYP_ID_S.NEXTVAL,
    'MyNamespace',
    'MyPermissionType',
    '{http://myapp.myu.edu}myPermissionTypeService',
    SYS_GUID())
```

One of the most important things to note about this is the service name (SRVC_NM) column. As we can see in the example above, for this KIM type we are linking it to a service named {http://myapp.myu.edu}myPermissionTypeService. This is how KIM will look up your custom type

service whenever it needs to load and invoke it.[1] It does this through the Rice resource loading framework which includes locally available services defined in Spring as well as services published on the Kuali Service Bus. For KIM type services, it's generally required to deploy them onto the KSB because the user interface components of KIM will use these when determining which custom attributes may need to be displayed and collected on it's various screens.

More information on how to publish these services can be found in the next section.

Once the KIM type has been registered, it will be assigned an ID, this is the value of the `KIM_TYP_ID` column after the record has been inserted. This ID can then be used to associate the type with the appropriate and desired data elements in KIM.

For example, to associate the custom `PermissionTypeService` you created earlier with one of your permission templates, you can execute the following SQL (assuming the ID of your new KIM type is 10000):

```
UPDATE KRIM_PERM_TMPL_T SET KIM_TYP_ID = '10000'
WHERE NMSPC_CD = 'MyNamespace' AND NM = 'MyPermissionTemplate'
```

Once this is complete, any existing or new permissions you create with this template will use your custom KIM type and it's associated type service.

Publishing Custom KIM Types to the Kuali Service Bus

As mentioned previously, KIM type services should be published onto the Kuali Service Bus in order to allow the KIM user interface functionality (which is typically deployed on the Rice Standalone Server) to access the services remotely. Since KIM type services are considered "callback" services because of the fact that the standalone server makes callbacks to them, the `org.kuali.rice.ksb.api.bus.support.CallbackServiceExporter` should be used.

Information on how to export and publish a callback service can be found in the section called "CallbackServiceExporter".

Assuming you have already wired up your custom PermissionTypeService implementation in your Spring file under a bean id of "myPermissionTypeService", an example Spring configuration which will publish the service would look like the following:

```
<bean id="myPermissionTypeService.exporter"
      class="org.kuali.rice.ksb.api.bus.support.CallbackServiceExporter"
      p:callbackService-ref="myPermissionTypeService"
      p:serviceNameSpaceURI="http://myapp.myu.edu"
      p:localServiceName="myPermissionTypeService"
      p:serviceInterface="org.kuali.rice.kim.framework.permission.PermissionTypeService"/>
```

KIM Database Tables

Table Name Prefixes

The KIM tables in the Rice database are prefixed by KRIM, which stands for **K**uali **R**ice **I**dentity **M**anagement.

---

[1] While the service name here is a single string value, it will be parsed into a `javax.xml.namespace.QName` object using that classes `valueOf(...)` method. This means that, for our example of `{http://myapp.myu.edu}myPermissionTypeService`, it will get parsed into a QName which is equivalent to `new QName("http://myapp.myu.edu", "myPermissionTypeService")`.

Unmapped LAST_UPDT_DT Columns

Many of the KIM tables have an additional column called LAST_UPDTD_DT (of type DATE in Oracle, DATETIME in MySQL) that isn't mapped at the ORM layer. Using this column is entirely optional, and it is unmapped by design. Its purpose is to aid implementers with tracking changes, and with doing data synchronization or extracts against KIM tables. The following sample PL/SQL script (Oracle only) adds to all the tables that contain LAST_UPDATED_DT an insert and update trigger to populate it:

```
DECLARE
    CURSOR tables IS
        SELECT table_name
            FROM user_tab_columns
            WHERE column_name = 'LAST_UPDATE_DT'
            AND data_type LIKE 'DATE%'
            ORDER BY 1;
BEGIN
    FOR rec IN tables LOOP
        EXECUTE IMMEDIATE 'CREATE OR REPLACE TRIGGER '||LOWER( SUBSTR( rec.table_name, 1, 27) )||'_tr BEFORE
 INSERT OR UPDATE ON '
            ||LOWER( rec.table_name )||' FOR EACH ROW BEGIN :new.last_update_ts := SYSDATE; END;';
    END LOOP;

END;
/
```

Chapter 5. KNS

KNS Configuration Guide

The Kuali Nervous System (KNS) is, primarily, an application development framework. Each Rice client application can use the KNS to construct various screens and build pieces of the application with the built-in components and services that the KNS provides.

To this end, configuration of the KNS in a client application can be accomplished by following these steps:

1. Creation of database tables in the client application's database that the KNS requires to function.

2. Loading of the **KNSConfigurer** inside of the **RiceConfigurer** Spring bean. This includes configuring connections to the databases.

3. Loading of the KNS struts module for the various UI components that the KNS provides in addition to any filters or servlets that need to be defined in the client application's **web.xml**.

4. Creation of a **ModuleConfiguration** for the application which instructs the KNS about which Data Dictionary files and OJB repository mapping files to load.

5. Customization of the various configuration parameters that the KNS supports.

Database Creation

In order for the KNS services to work, many of them require the ability to access special tables within the client application's database. These tables are used to store various pieces of data; from notes and attachments on documents to maintenance document data and much more.

These tables are included as part of either the demo-client-dataset or the bootstrap-client-dataset. These datasets are provided with the Kuali Rice binary distributions and instructions on how to install them can be found in the Installation Guide.

Note

It's important to note that these tables should be installed in the client application's database, and not the Rice standalone server database.

KNSConfigurer and RiceConfigurer

As with the other modules, a **KNSConfigurer** needs to be injected into the **RiceConfigurer** in order to instruct Rice to initialize the KNS module. The main purpose of this is to allow for the **applicationDataSource** and the **serverDataSource** to be specified.

The **applicationDataSource** should point to the client application's database. That database should contain the tables from one of the client datasets.

The **serverDataSource** should point to the database of the Rice standalone server. This is used for allowing access to the various KNS central services that use data in the Rice server database. This includes such data as System Parameters, Namespaces, Campuses, States and Countries.

Here is an example of Spring configuration for a KNS client:

```
<bean id="rice" class="org.kuali.rice.core.config.RiceConfigurer">
    ...
    <property name="knsConfigurer">
        <bean class="org.kuali.rice.kns.config.KNSConfigurer"
>
             <property name="applicationDataSource" ref="applicationDataSource"/>
             <property name="serverDataSource" ref="riceServerDataSource"/>
        </bean>
    </property>
    ...
</bean>
```

Alternatively, you can just set the **dataSource** and **serverDataSource** on the **RiceConfigurer** itself and that will be used for the KNS **applicationDataSource** and **serverDataSource** respectively. This is useful when using the same database for all the different modules of Rice.

The **KNSConfigurer** supports some other properties as well. See the javadocs of **KNSConfigurer** for more information.

Configuring the KNS Web Application Components

Loading the KNS Struts Modules

The web application framework of the KNS is built on top of the [Apache Struts](#) framework. As a result of this, the web application components of the KNS are loaded into the client application as a struts module. The struts module and various pieces of the Rice web content can be found in the binary distribution. They should be copied into the root directory of your web application.

A special implementation of the Struts **ActionServlet** is provided to help with the loading of the struts modules. It can be configured in the application's **web.xml** as in the following example:

```
<servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>org.kuali.rice.kns.web.struts.action.KualiActionServlet</servlet-class>
    <init-param>
        <param-name>config</param-name>
        <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <load-on-startup>0</load-on-startup>
</servlet>
```

Notice the **init-param** above points to a Struts configuration file. This file is intended to be the struts configuration file for the client application. It's used by the KNS for doing redirects back to the main application and is also used for adding KNS-based screens within the client application. Specifically, this is where action mappings go if using the transactional document framework of the KNS.

There is an example file in the distributions under **config/examples/struts-config.example.xml**. This will need to be renamed to **struts-config.xml** and copied to your web application's **WEB-INF** directory. It can then be loaded using the KualiActionServlet as seen in the example above.

In this example file you will see a reference to a message resource properties file. As is the case with a standard Struts-based application, the resource properties file is used to load text strings for internationalization purposes. The KNS framework requires that at least the app.title property to be set, as in the following example:

```
app.title=Recipe Sample Application
```

Configuring KNS Servlet Context Listeners

The KNS framework requires a couple of ServletContextListener classes to be configured in the application's web.xml. These include:

- **org.kuali.rice.kns.web.listener.JstlConstantsInitListener**

- **org.kuali.rice.kns.web.listener.KualiHttpSessionListener**

These should be included in the **web.xml** *after* any listeners or servlets that might be used to actually initialize the Spring context that loads Rice.

Here is an example of what this configuration might look like in web.xml:

```
<listener>
    <listener-class>my.app.package.ListenerThatStartsRice</listener-class>

</listener>


<listener>
    <listener-class>org.kuali.rice.kns.web.listener.JstlConstantsInitListener</listener-class>

</listener>



<listener>
    <listener-class>org.kuali.rice.kns.web.listener.KualiHttpSessionListener</listener-class>
</listener>
```

Configuring KNS Message Resources

As of Rice version 1.0.1.1, messages are loaded through the new KualiPropertyMessageResourcesFactory. This class is a factory of KualiPropertyMessageResources, which takes in a comma delimited list of .properties files.

This is set up in the struts-config.xml files near the end of the file:

```
<message-resources factory="org.kuali.rice.kns.web.struts.action.KualiPropertyMessageResourceFactory"
 parameter="" />
```

When the parameter above is set to an empty string, Rice uses the default value of properties files. The default value is set by the rice.struts.message.resources property the common-config-defaults.xml file. This is the default setting:

```
<param name="rice.struts.message.resources">KR-
ApplicationResources,org.kuali.rice.kew.ApplicationResources,org.kuali.rice.ksb.messaging.ApplicationResources,KIM-
ApplicationResources" />
```

This can be overridden in rice-config.xml. This value should be in a comma delimited format. The list of files is loaded from left to right, with any duplicated properties being overridden in that order. Therefore, in the list default list if a property key in KR-ApplicationResources was duplicated in KIM-ApplicationResources, the value used would be the one set in KIM-ApplicationResources.

Configuring AJAX Support

The KNS uses [DWR](#) to provide AJAX support. In order to enable this, the **org.kuali.rice.kns.web.servlet.KualiDWRServlet** must be configured in the application's **web.xml** as follows:

```
<servlet>
    <servlet-name>dwr-invoker</servlet-name>
    <servlet-class>org.kuali.rice.kns.web.servlet.KualiDWRServlet</servlet-class>
    <init-param>
        <param-name>debug</param-name>
        <param-value>false</param-value>
    </init-param>
    <init-param>
        <param-name>springpath</param-name>
        <param-value>true</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>

...

<servlet-mapping>
    <servlet-name>dwr-invoker</servlet-name>
    <url-pattern>/dwr/*</url-pattern>
</servlet-mapping>
```

Module Configuration – Loading Data Dictionary and OJB Files

One of the most important pieces of the KNS framework is the Data Dictionary. It's used to define various pieces of metadata about business objects, maintenance documents, lookups, inquiries and more. These Data Dictionary files are authored in XML and are loaded using a **ModuleConfiguration**. Additionally, business objects in the KNS are mapped to the database using an object relational mapping library called [Apache OJB](#). The **ModuleConfiguration** is also used to load those mapping files.

A **ModuleConfiguration** is a bean wired in Spring XML that instructs the KNS to load various pieces of configuration for a particular module. A client application could create a single module or multiple modules, depending on how it is organized. This configuration allows for the specification of the following:

- The module's namespace

- The Data Dictionary files to load

- The OJB repository files to load

- The package prefix of business objects in this module

- Externalizable business object definitions

Here is an example of what this configuration might look like:

```
<bean id="sampleAppModuleConfiguration"
 class="org.kuali.rice.kns.bo.ModuleConfiguration">
    <property name="namespaceCode" value="tv"/>
    <property name="initializeDataDictionary" value="true"/>
    <property name="dataDictionaryPackages">
        <list>
            <value>classpath:edu/sampleu/travel/datadictionary</value>
        </list>
```

```
        </property>
        <property name="databaseRepositoryFilePaths">
            <list>
                <value>OJB-repository-sampleapp.xml</value>
            </list>
        </property>
        <property name="packagePrefixes">
            <list>
                <value>edu.sampleu.travel</value>
            </list>
        </property>
</bean>
```

When the module is initialized by the KNS, it will load all of the Data Dictionary files into the Data Dictionary service. Additionally, all OJB files will be loaded and merged into the main OJB repository. The **packagePrefixes** are used to identify which business objects this module is responsible for.

There are more configuration options on the **ModuleConfiguration**. See the javadocs on this class for more information.

KNS Configuration Parameters

The KNS supports numerous configuration parameters that can be set in the Rice configuration file. Below is a list of these with descriptions and defaults.

Table 5.1. KNS Configuration Parameters

Property	Description	Default
application.url	Base URL of the application. Example: http://localhost/kr-dev	${appserver.url}/${app.context.name}
attachments.directory	Directory in which to store attachments	/tmp/${environment}/attachments
attachments.pending.directory	Directory in which to store attachments on a document or object which have not yet been persisted	/tmp/${environment}/attachments/pending
classpath.resource.prefix	The location, in the classpath, of methods that may be called by DWR.	/WEB-INF/classes/
externalizable.help.url	Base URL at which web-based help content will be located	/${app.context.name}/kr/static/help/
externalizable.images.url	Base URL at which images are located	/${app.context.name}/kr/static/images/
kr.externalizable.images.url	Base URL at which images that are part of the standard Kuali Rice image set are stored	/${app.context.name}/kr/static/images/
kr.url	Base URL of the KNS struts module. Includes the various built-in GUI components such as lookups, inquiries, etc.	${application.url}/kr
production.environment.code	The environment code that will be used to identify this application as a "production" instance. Certain features are turned off in non-production instances (email, for example)	PRD
mail.relay.server	Name of the SMTP server to use for sending emails from the KNS	
kr.incident.mailing.list	The email address where exception and incident reports should be sent	
javascript.files	A comma-separated list of javascript files to load on every KNS-based web page	See impl/src/main/resources/resources/META-INF/common-config-defaults.xml in the source distribution
css.files	A comma-separated list of css files to load on every KNS-based web page	See impl/src/main/resources/resources/META-INF/common-config-defaults.xml in the source distribution

Property	Description	Default
enable.nonproduction.data.unmasking	If the current application is running in an non-production environment, this determines if all fields should be unmasked in the Nervous System, even if the field would otherwise be masked.	false
kns.cache.parameter.max.size	The maximum number of parameters that can be stored in the kns parameter cache	200
kns.cache.parameter.max.age.seconds	The maximum age (in seconds) of entries in the parameter cache	3600
kns.cache.nonDatabaseComponent.max.size	The maximum size of the cache that is used to store parameter components that don't come from the database (i.e. are loaded from the Data Dictionary and other locations)	50
kns.cache.nonDatabaseComponent.max.age.seconds	The maximum ago (in seconds) of entries in the parameter non-database component cache	3600
session.document.cache.size	The max size of the cache used to store document sessions	100
portal.javascript.files	A list of Javascript files to be included int the "portal", ie the frame around the application pages.	
portal.css.files	A list of CSS files to be used in the "portal", ie the frame around the application pages.	rice-portal/css/portal.css
rice.kns.struts.config.files	The struts-config.xml configuration file that the KNS portion of the Rice application will use.	/kr/WEB-INF/struts-config.xml
rice.kns.illegalBusinessObjectsForSave	A comma-separated list of business objects that the KNS should not be allowed to save	
rice.kns.illegalBusinessObjectsForSave.applyCheck	If set to true, the check for illegal business objects to save will be performed, if false, it will not	true
encryption.key	The DES key to use for encrypting data elements that are configured for encryption in the KNS	
rice.struts.message.resources	The key used to load message property files. The value should be a comma delimited list or properties files.	KR-ApplicationResources, org.kuali.rice.kew.ApplicationResources, org.kuali.rice.ksb.messaging.ApplicationResources, KIM-ApplicationResources

KNS Business Object Framework

Business Object Database Table Definition

Business object instances are typically java object representations of rows of a database table.

The addition of following columns to each database table is strongly suggested:

- Object ID

- Version number

The Object ID is used as a globally unique identifier (or GUID) of each row across all database tables. That is, every row in every table should have a different Object ID value. It is typically defined as a VARCHAR field of 36 characters, and should be named "OBJ_ID" in the database. A unique constraint should be applied to the object ID column, but must NOT be part of the primary key. The KNS system will assume that each row has a unique value.

The object ID value is automatically stored by the framework and/or the database layer.

KFS/Rice uses optimistic locking to provide concurrency control. Optimistic locking requires the use of a version number field, named "VER_NBR". On Oracle, the field is defined as a NUMBER(8,0). On MySQL, the field is defined as a DECIMAL(8). This column should NOT be part of the primary key.

About optimistic locking

Optimistic locking helps to prevent updates to stale data and consists of two steps:

1. Retrieval of a row from a database, including the value of the version number column

2. Updating/deleting a row from the database with the same primary key and version number criteria. If updating the table, the version number will be incremented by one.

The following series of steps demonstrates how optimistic locking works:

1. User A retrieves the row for chart code "BL". The row has version number of 3.

2. User A performs an update of the "BL" record. The SQL query that updates the record would read something like "UPDATE CA_CHART_T SET <some updates>, VER_NBR = 4 WHERE FIN_COA_CD = "BL" and VER_NBR = 3. (The "4" refers to the incremented version number.)

3. User B retrieves the row for chart code "BL". The version number is now 4.

4. User B performs an update of the "BL" record. The SQL query that updates the record would read something like "UPDATE CA_CHART_T SET <some updates>, VER_NBR = 5 WHERE FIN_COA_CD = "BL" and VER_NBR = 4. (The "5" refers to the incremented version number.)

The following series of steps demonstrates how optimistic locking prevents concurrency problems.

1. User A retrieves the row for chart code "BL". The row has version number of 3.

2. User B retrieves the row for chart code "BL". Like user A, the version number is 3.

3. User A performs a update of the "BL" record. The SQL query that updates the record would read something like "UPDATE CA_CHART_T SET <some updates>, VER_NBR = 4 WHERE FIN_COA_CD = "BL" and VER_NBR = 3. (The "4" refers to the incremented version number.)

4. User B performs a update of the "BL" record. The SQL query that updates the record would read something like what User A executed above (notice the version numbers). However, the previous step already updated the version number to 4 from 3, so this update does nothing (i.e. update row count = 0) because it was trying to update the BL chart with a version number of 3. The system detects the 0 update row count, and throws an OptimisticLockingException. This exception indicates that the system tried to update stale data.

Business Object Database Mapping

The default mapping library used by the KNS for this release is OJB from Apache. More information can be found on the OJB website: http://db.apache.org/ojb/.

Purpose of OJB mappings

OJB repository files map the following information:

1. The BusinessObject (BO) mapped to a given database table

2. The getter/setter method in the BO mapped to a given database column

3. The fields(s) comprising foreign keys between a business object and its reference(s)

OJB documentation

Currently, OJB is used as the underlying persistence layer. It converts database rows into java objects upon retrieval, and vice versa upon updates/deletes. This section assumes that the reader is familiar with the basic mapping constructs/principles described on these pages:

• http://db.apache.org/ojb/docu/guides/repository.html#class-descriptor-N104E3

• http://db.apache.org/ojb/docu/guides/repository.html#field-descriptor-N105C6

• http://db.apache.org/ojb/docu/guides/repository.html#field-descriptor-N105C6

• http://db.apache.org/ojb/docu/guides/repository.html#collection-descriptor-N10770

• http://db.apache.org/ojb/docu/guides/repository.html#foreignkey

• http://db.apache.org/ojb/docu/guides/repository.html#inverse-foreignkey

• http://db.apache.org/ojb/docu/guides/basic-technique.html

OJB field-level conversions

OJB provides a way to convert data before they are persisted to and retrieved from the database. This is accomplished by specifying a class that implements org.apache.ojb.broker.accesslayer.conversions.FieldConversion in the <field-descriptor> element.

The following are the more often used converters in KFS/Rice:

• org.kuali.core.util.OjbCharBooleanConversion: since boolean flags are typically stored as "Y" or "N" (i.e. strings) in the database but represented as booleans within business objects, this converter automatically allows converts between the string and the boolean representation

• org.kuali.core.util.OjbKualiEncryptDecryptFieldConversion: provides seamless encryption of values when persisting, and decryption when retrieving from the database. Beware that the business object itself holds an unencrypted value, and as such, care should be taken to ensure that unencrypted sensitive data are not exposed to unauthorized parties.

Both OJB and the KNS offer a number of FieldConversion implementations beyond these two for use in client applications.

Example converter declaration for a sample Business Object

```
<field-descriptor name="bankAccountNbr" column="BNK_ACCT_NBR" jdbc-type="VARCHAR"
 conversion="org.kuali.core.util.OjbKualiEncryptDecryptFieldConversion"/>
```

When to use OJB vs. data dictionary relationships

OJB relationships should be used to define relationships between tables that are guaranteed to exist within the same database.

For example, assume a sample Business Object class "Bank". The Bank class contains a BankType reference object. Typically a BankType class table would exist in the same database as the Bank class table. In this example the relationship between Bank and BankType can be defined by OJB. However, a "User" business object table typically will exist in an external system since it will likely be referenced by

more than one Rice client application. If a BO had a relationship with a "User" BO, the mapping would require that the relationship be set up via the data dictionary files (which will be discussed in detail later in this document). Any business object implementing the org.kuali.rice.kns.bo.ExternalizableBusinessObject interface needs to be related to via the data dictionary.

Example OJB Mapping

Here is an example directly from Rice in the file OJB-repository-kns.xml:

```xml
<class-descriptor class="org.kuali.rice.kns.bo.StateImpl" table="KR_STATE_T">
    <field-descriptor name="postalCountryCode" column="POSTAL_CNTRY_CD" jdbc-type="VARCHAR" primarykey="true"
 index="true" />
    <field-descriptor name="postalStateCode" column="POSTAL_STATE_CD" jdbc-type="VARCHAR" primarykey="true"
 index="true" />
    <field-descriptor name="postalStateName" column="POSTAL_STATE_NM" jdbc-type="VARCHAR" />
    <field-descriptor name="objectId" column="OBJ_ID" jdbc-type="VARCHAR" index="true" />
    <field-descriptor name="versionNumber" column="VER_NBR" jdbc-type="BIGINT" locking="true" />
    <field-descriptor name="active" column="ACTV_IND" jdbc-type="VARCHAR"
 conversion="org.kuali.rice.kns.util.OjbCharBooleanConversion"/>

    <reference-descriptor name="country" class-ref="org.kuali.rice.kns.bo.CountryImpl" auto-retrieve="true"
 auto-update="none" auto-delete="none">
        <foreignkey field-ref="postalCountryCode" />
    </reference-descriptor>
</class-descriptor>
```

In this OJB mapping, we can determine the following information:

1. The KR_STATE_T table is mapped to the org.kuali.rice.kns.bo.StateImpl business object

2. The POSTAL_CNTRY_CD column is mapped to the "postalCountryCode" property of the BO (i.e. accessed using the getPostalCountryCode and setPostalCountryCode methods), is a VARCHAR, is indexed, and is one of the fields in the primary key

3. The POSTAL_STATE_CD column is mapped to the "postalStateCode" property of the BO, is a VARCHAR, is indexed, and is one of the fields in the primary key

4. The OBJ_ID column is mapped to the "objectId" property, is indexed, and is a VARCHAR

5. The VER_NBR column is mapped to the "verionNumber" property, is a BIGINT, and is used for locking

6. The ACTV_IND column is mapped to the "active" property, is a VARCHAR, and uses the conversion class **org.kuali.rice.kns.util.OjbCharBooleanConversion**

We can determine the following information about the "country" reference object:

1. It is of type **org.kuali.rice.kns.bo.CountryImpl**

2. the auto-retrieve attribute is true: When the StateImpl is retrieved from OJB, the CountryImpl object will behave like it was retrieved as well (the **proxy** attribute of the 'field-descriptor' tag can be set to true or false to determine whether the CountryImpl is really retrieved when the account is retrieved or not)

3. the auto-update attribute is none: When the StateImpl is updated using OJB, the CountryImpl object will not be updated even if changes have been made to it

4. the auto-delete attribute is none: When the StateImpl is deleted using OJB, the CountryImpl object will not be deleted

5. The <foreignkey> tag specifies the fields in the StateImpl BO that are in a foreign key relationship and their order with the primary key fields in the CountryImpl BO. The CountryImpl BO has one

primary key field, and the value from StateImpl's "postalCountryCode" property is used as the value for CountryImpl's primary key value.

Example OJB Mapping for Collection Descriptor

A mapping may also define a collection-descriptor tag as follows:

```
<class-descriptor class="org.kuali.rice.kns.test.document.bo.AccountManager" table="TRV_ACCT_FO">
    <field-descriptor name="id" column="acct_fo_id" jdbc-type="BIGINT" primarykey="true" autoincrement="true"
 sequence-name="TRV_FO_ID_S" />
    <field-descriptor name="userName" column="acct_fo_user_name" jdbc-type="VARCHAR" />

    <collection-descriptor name="accounts" collection-
class="org.apache.ojb.broker.util.collections.ManageableArrayList" element-class-
ref="org.kuali.rice.kns.test.document.bo.Account" auto-retrieve="true" auto-update="object" auto-
delete="object" proxy="true" >
        <orderby name="accountNumber" sort="ASC" />
        <inverse-foreignkey field-ref="amId" />
    </collection-descriptor>
</class-descriptor>
```

We can determine the following information about the "accounts" collection reference:

1. The collection itself is of type org.apache.ojb.broker.util.collections.ManageableArrayList, which keeps track of which elements have been removed from the array, to help when deleting elements.

2. Each element of the collection is of type org.kuali.rice.kns.test.document.bo.Account.

3. The auto-retrieve attribute is true: when the AccountManager is retrieved from the database, the collection will be populated or behave as if it were populated upon accessing the collection. (the proxy setting determines whether the database is queried when the AccountManager is retrieved from the DB or whether it will retrieve from the DB only when the collection is accessed (i.e. lazy loading)).

4. The auto-update attribute is object: when the AccountManager is inserted or updated, the accounts collection is inserted or updated accordingly.

5. The auto-delete attribute is object: when the AccountManager is deleted, the corresponding accounts will be deleted as well.

6. The <orderby> tag specifies the sort order of elements in the collection. In this case, the account numbers will be in ascending order in the collection.

7. The <inverse-foreignkey> specifies the fields of the element BO (i.e. Account) that will match the primary key fields of the AccountManager BO. The "amId" attribute in the Account table will be used to find objects that match the primary key of the AccountManager object, or in this case the "id" attribute.

Business Object Java Definition

Business Objects are java classes that implement the **org.kuali.core.bo.BusinessObject** interface. However, a majority of business objects extend **org.kuali.core.bo.PersistableBusinessObjectBase**, which implements **org.kuali.core.bo.PersistableBusinessObject** and **org.kuali.core.bo.BusinessObject**. Business Objects which extend from the class **PersistableBusinessObjectBase** also have an advantage in that they will inherit getter and setter methods for the attributes 'version number' and 'object id'.

In each application, all simple class names (i.e. ignoring the package) should be unique. If multiple packages contain the same class name, the data dictionary may not load the duplicated classes properly.

Business objects need to implement getter and setter methods for each field that is mapped between java business objects and the database table (the mapping is described later). Therefore, if, in java, the ACCOUNT_NM database column is named "accountName", then the getter method should be called getAccountName and the setter should be setAccountName (i.e. the conventions follow the standard Java bean getters and setters practices).

Objects that extend **org.kuali.core.bo.BusinessObjectBase** must also implement the toStringMapper method, which returns a map of the BO's fields to be used in toString.

The **org.kuali.core.bo.PersistableBusinessObjectBase** class has several more methods that can be overridden that customize the behavior of the business object. Just a few examples are customizations that can be made upon persistence and retrieval of the business object, and how reference objects of the business object are refreshed, as well as other methods.

Reference Objects

A reference object is a member variable of a business object that also implements the BusinessObject interface. It refers to the database row referenced by the values in a foreign key relationship. For example, the CampusImpl BO/table has a column for a campus type code (CAMPUS_TYP_CD). Therefore, the CampusImpl BO may have a referenced CampusTypeImpl object, which represents the campus type row referred to by the campus' campus type code. Here is the CampusImpl OJB mapping:

```
<class-descriptor class="org.kuali.rice.kns.bo.CampusImpl" table="KRNS_CAMPUS_T">
    <field-descriptor name="campusCode" column="CAMPUS_CD" jdbc-type="VARCHAR" primarykey="true" index="true" /
>
    <field-descriptor name="campusName" column="CAMPUS_NM" jdbc-type="VARCHAR" />
    <field-descriptor name="campusShortName" column="CAMPUS_SHRT_NM" jdbc-type="VARCHAR" />
    <field-descriptor name="campusTypeCode" column="CAMPUS_TYP_CD" jdbc-type="VARCHAR" />
    <field-descriptor name="objectId" column="OBJ_ID" jdbc-type="VARCHAR" index="true" />
    <field-descriptor name="versionNumber" column="VER_NBR" jdbc-type="BIGINT" locking="true" />
    <field-descriptor name="active" column="ACTV_IND" jdbc-type="VARCHAR"
 conversion="org.kuali.rice.kns.util.OjbCharBooleanConversion" />
    <reference-descriptor name="campusType" class-ref="org.kuali.rice.kns.bo.CampusTypeImpl" auto-
retrieve="true" auto-update="none" auto-delete="none">
        <foreignkey field-ref="campusTypeCode" />
    </reference-descriptor>
</class-descriptor>
```

Here are bits of the CampusImpl class file:

```
public class CampusImpl extends PersistableBusinessObjectBase implements Campus, Inactivateable {
    private String campusCode;
    private String campusName;
    private String campusShortName;
    private String campusTypeCode;
protected boolean active;

    private CampusType campusType;
...
```

A collection reference is a member variable of a business object that implements java.util.Collection, with each element in the collection being a BusinessObject. A collection reference would be appropriate to model something like the list of Kuali Financial sub accounts of the Kuali Financial account business object.

A reference object or collection is defined in two steps:

1. A field in a business object is created for either the reference object or collection reference

2. A relationship is mapped within either OJB (See above) or the data dictionary (See below)

To refresh (or retrieve) a reference object is to reload the referenced row from the database, in case the foreign key field values or referenced data have changed.

For references mapped within the data dictionary, the framework does not have the logic to enable refreshing of a reference. The code must both implement the logic to refresh a data dictionary defined reference and the logic to invoke refreshing. A specific explanation can be found below.

Refreshing reference objects mapped in OJB

For references mapped within OJB, the framework automatically takes care of the logic to enable refreshing of a reference. Under certain circumstances, it's able to automatically refresh references upon retrieval of the main BO from the database, and refreshing can also be invoked manually.

Note that this means that if the value of a foreign key field is changed, the corresponding reference object is not refreshed automatically. Taking the CampusImpl BO example above, if the code alters the CampusImpl's campusTypeCode field, the framework will not automatically retrieve the new associated CampusTypeImpl BO reference object. To refresh the CampusImpl's CampusTypeImpl reference object with the new campus type code, refresh/retrieve must be manually called (see below).

Refreshing reference objects not mapped in OJB

For references with relationships that are not mapped in OJB, code will need to be written to accommodate refreshing. A common example of this is Person object references, because institutions may decide to use another source for Identity Management (e.g. LDAP).

Although there are alternative strategies for accommodating refreshing, typically getter methods of these non-OJB mapped reference objects include the code that retrieves the reference object from the underlying datasource.

In contrast to OJB-mapped references, note that this strategy allows for the automatic refreshing of reference objects when a foreign key field value has been changed. If, in our example using CampusImpl above, the reference object for CampusTypeImpl was not defined in OJB, the string campusTypeCode may be changed and that would be enough to alter the getter method for CampusTypeImpl to properly retrieve the correct row from the database.

Initializing collection references

Business objects fall into two broad, and for the most part mutually exclusive, categories: those that are edited by maintenance documents and those that are not. This section refers only to business objects that are edited by maintenance documents that have updatable collections.

When constructing this type of BusinessObject, initialize each of the updatable collection references to an instance of org.kuali.rice.kns.util.TypedArrayList. TypedArrayList is a subclass of ArrayList that takes in a java.lang.Class object in its constructor. All elements of this list must be of that type, and when the get(int) method is called, if necessary, this list will automatically construct items of the type to avoid an IndexOutOfBoundsException. Take the example below, the SummaryAccount BO contains an updatable reference to a list of PurApSummaryItem objects.

```
public class SummaryAccount {
    private List<PurApSummaryItem> items;


    public SummaryAccount() {
        super();
        items = new TypedArrayList(PurApSummaryItem.class);
    }
}
```

When a collection is non-updatable (i.e. read only from the database), it is not necessary to initialize the collection. OJB will take care of list construction and population.

Inactivateable Business Objects

Business objects that have active/inactive states should implement the Inactivateable interface:

```
public interface Inactivateable {

    public boolean isActive();

                /* Indicates whether the record is active or inactive.
 */
    public void setActive(boolean active);
    /* Sets the record to active or inactive.
 */
}
```

By implementing this interface, functionality such as default active checks and inactivation blocking in the maintenance framework can be taken advantage of.

InactivateableFromTo Business Objects

Business objects that have active from and to dates (effective dating) should implement the InactivateableFromTo interface:

```
public interface InactivateableFromTo extends Inactivateable {

    /* Sets the date for which record will be active
     *
@param from
* - Date value to set

    */
    public void setActiveFromDate(Date from);


    /* Gets the date for which the record become active
     *
 @return Date
    */
    public Date getActiveFromDate();


    /* Sets the date for which record will be active to
     * @param from
* - Date value to set
    */

public void setActiveToDate(Date to);


    /* Gets the date for which the record become inactive
     *
 @return Date
    */

public Date getActiveToDate();


/* Gets the date for which the record is being compared to in determining active/inactive
     *
 @return Date
    */
```

```
public Date getActiveAsOfDate();


    /* Sets the date for which the record should be compared to in determining active/inactive, if
     * not set then the current date will be used
     *
 @param activeAsOfDate
     * - Date value to set

*/

public void setActiveAsOfDate(Date activeAsOfDate);


}
```

Explanation of InactivateableFromTo fields

activeFromDate - The date for which the record becomes active (inclusive when checking active status).

activeToDate - The date to which the record is active (exclusive when checking active status).

active - The active field is calculated from the active from and to dates. If the active from date is less than or equal to current date (or from date is null) and the current date is less than the active to date (or to date is null) the active getter will return true, otherwise it will return false.

current - The current field is set to true for records with the greatest active from date less than or equal to the current date.

For example say we have two employee records:

- rec 1, empl A, active from 01/01/2010, active to 01/01/2011

- rec 2, empl A, active from 03/01/2010, active to 01/01/2011

With 03/01/2010 <= current date < 01/01/2011 both of these records will be active, however only rec 2 would be current since it has a later active begin date.

To determine the maximum active begin date, records are grouped by the fields declared in the data dictionary for the business object.

activeAsOfDate - By default when checking the active or current status the current date is used, however this field can be set to check the status as of another date.

For example say we have a record with active from date 01/01/2010 and active to date 06/01/2010, with the current date equal to 08/01/2010. With the active as of date empty, the current date will be used and this record will be determined inactive. However if we set the active as of date equal to 05/01/2010 (which falls between the active date range) and query, this record will be determined active.

Framework Support

Business objects that implement InactivateableFromTo can participate in default existence checks and inactivation blocking functionality. In addition, the lookup framework contains special logic for searching on InactivateableFromTo instances. This includes:

1. Translating criteria on the active field (active true or false) to criteria on the active to and from date fields

2. Translating criteria on the current field (current true of false) to criteria selecting the active record with the greatest active from date less than or equal to the active date

3. Handles the active as of date when doing active or current queries

InactivateableFromToService

For finding active and current InactivateableFromTo records InactivateableFromToService can be used. This service provides many methods for dealing with InactivateableFromTo objects in code.

Group by Attributes

In order to determine whether or not an InactivateableFromTo record is current, the framework must know what fields of the business object to group by (see 'current' in 'Explanation of InactivateableFromTo fields'). This is configured by setting the groupByAttributesForEffectiveDating property on the data dictionary BusinessObjectEntry.

Example:

```
<bean id="TravelAccountUseRate-parentBean" abstract="true" parent="BusinessObjectEntry">
    <property name="businessObjectClass" value="edu.sampleu.travel.bo.TravelAccountUseRate"/>
    <property name="inquiryDefinition">
        <ref bean="TravelAccountUseRate-inquiryDefinition"/>
    </property>
    <property name="lookupDefinition">
        <ref bean="TravelAccountUseRate-lookupDefinition"/>
    </property>
    <property name="titleAttribute" value="Travel Account Use Rate"/>
    <property name="objectLabel" value="Travel Account Use Rate"/>
    <property name="attributes">
        <list>
            <ref bean="TravelAccountUseRate-id"/>
            <ref bean="TravelAccountUseRate-number"/>
            <ref bean="TravelAccountUseRate-rate"/>
            <ref bean="TravelAccountUseRate-activeFromDate"/>
            <ref bean="TravelAccountUseRate-activeToDate"/>
            <ref bean="TravelAccountUseRate-activeAsOfDate"/>
            <ref bean="TravelAccountUseRate-active"/>
            <ref bean="TravelAccountUseRate-current"/>
        </list>
    </property>
    <property name="groupByAttributesForEffectiveDating">
        <list>
            <value>number</value>
        </list>
    </property>
</bean>
```

KNS Data Dictionary Overview

The data dictionary is the main repository for metadata storage and provides the glue to combining classes related to a single piece of functionality. The data dictionary is specified in XML and allows for quick changes to be made to functionality. The Data Dictionary files use the Spring Framework for configuration so the notation and parsing operation will match that of the files that define the module configurers.

The contents of the data dictionary are defined by two sets of vocabularies; the 'business object' and the 'document' data.

Business Object Data Dictionary

Business Object Data Dictionary entries provide the KNS framework extra metadata about a business object which is not provided by the persistence mapping or the class itself.

The business object data dictionary contains information about:

- Descriptive labels for each attribute in the business object (data dictionary terminology uses the term "attribute" to refer to fields with getter/setter methods).

- Metadata about each attribute

- How input fields on HTML pages should be rendered for an attribute (e.g. textbox, drop down, etc.)

- The data elements from the business object that are shown to users on the KNS Inquiry page

- The data elements of the business object that can be used as criteria or shown as result data in the KNS Lookup for the business object

The business object data dictionary does not contain information about:

- Which BO does a table correspond to (responsibility of persistence layer, e.g. OJB)

- How fields in the BO correspond to database columns (responsibility of persistence layer, e.g. OJB)

- The orientation of various fields on user interface screens

Note About Following Documentation

One thing to note is the use of 'abstract' parent beans within the Rice files. These are used to facilitate easy overriding of beans from Rice in a client application or a customized Rice standalone server installation. Take the following example where the "RealBean" may be defined within Rice:

```
<bean id="RealBean" parent="RealBean-parent" />

<bean id="RealBean-parent" abstract="true" />
```

Client applications overriding this bean definition should always retain the id "RealBean". This allows for any developer working with overriding data dictionary files to easily define an override using the following parent bean structure:

```
<bean id="RealBean" parent="RealBean-client-parent" />

<bean id="RealBean-client-parent" abstract="true" parent="RealBean-parent" >

<!-- any client overrides go here -->
</bean>
```

The setup above will take any configuration from the Rice defined "RealBean-parent" and allow the client developer to override individual properties inside the bean. Then when anything inside Rice or the client application references the data dictionary bean "RealBean" they will get the Rice defined values unless they were overridden by client application developers. See the Spring Framework documentation for more examples of this.

For the sake of this documentation, the abstract parent bean structure will be mostly ignored but its operation is consistent throughout all data dictionary files.

Data Dictionary File Layout

A sample Data Dictionary file to show typical organization of various beans that may be defined:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>
    <bean id="Account" parent="Account-parentBean"/>
```

```
            <bean id="Account-parentBean" abstract="true" parent="BusinessObjectEntry">
                <property name="businessObjectClass" value="org.kuali.kfs.coa.businessobject.Account"/>
                <property name="inquiryDefinition" ref="Account-inquiryDefinition"/>
                <property name="lookupDefinition" ref="Account-lookupDefinition"/>
                <property name="titleAttribute" value="accountNumber"/>
                <property name="objectLabel" value="Account"/>
                <!-- Attribute definition -->
                <property name="attributes">
                    <list>
                        <!-- list goes here -->
                    </list>
                </property>
                <!-- Collections -->
                <property name="collections">
                    <list>
                        <!-- list goes here -->
                    </list>
                </property>
                <!-- Relationships -->
                <property name="relationships">
                    <list>
                        <!-- list goes here -->
                    </list>
                </property>
                <!-- Inactivation blocking definitions -->
                <property name="inactivationBlockingDefinitions">
                    <list>
                        <!-- list goes here -->
                    </list>
                </property>
            </bean>
            <bean id="Account-inquiryDefinition" parent="Account-inquiryDefinition-parentBean"/>
            <!-- Definition of 'Account-inquiryDefinition-parentBean' bean goes here -->
            <bean id="Account-lookupDefinition" parent="Account-lookupDefinition-parentBean"/>
            <!--Definition of 'Account-lookupDefinition-parentBean' bean goes here -->

</beans>
```

A more specific Rice example might be the CampusImpl object (whose business object data dictionary file is Campus.xml). Here is the main bean definition from that file:

```
<bean id="Campus-parentBean" abstract="true" parent="BusinessObjectEntry">
    <property name="businessObjectClass" value="org.kuali.rice.kns.bo.CampusImpl"/>
    <property name="inquiryDefinition">
        <ref bean="Campus-inquiryDefinition"/>
    </property>
    <property name="lookupDefinition">
        <ref bean="Campus-lookupDefinition"/>
    </property>
    <property name="titleAttribute" value="campusCode"/>
    <property name="objectLabel" value="Campus"/>
    <property name="attributes">
        <list>
            <ref bean="Campus-campusCode"/>
            <ref bean="Campus-campusName"/>
            <ref bean="Campus-campusShortName"/>
            <ref bean="Campus-campusTypeCode"/>
            <ref bean="Campus-versionNumber"/>
        </list>
    </property>
</bean>
```

One of the main properties required is the **businessObjectClass** which defines the java implementation class that this business object data dictionary file will be used for.

The **inquiryDefinition** and the **lookupDefinition** will be covered later in this document but for now simply note that the property is using a <ref> tag to point to a bean id that exists elsewhere in this file.

The **titleAttribute** property defines the attribute of the business object that is the primary key. This is typically used to define which attribute can be used to display the inquiry page.

The **objectLabel** property is the label that will be used for all general business object references including where the system has collections of the business object.

Attribute Definition

Attribute definitions are used to provide metadata about the attributes (i.e. fields) of a business object. The following is a sampling of attribute definitions from the CampusImpl business object data dictionary file:

```
<bean id="Campus-campusCode-parentBean" abstract="true" parent="AttributeDefinition">
    <property name="forceUppercase" value="true"/>
    <property name="shortLabel" value="Campus Code"/>
    <property name="maxLength" value="2"/>
    <property name="validationPattern">
        <bean parent="AlphaNumericValidationPattern"/>
    </property>
    <property name="required" value="true"/>
    <property name="control">
        <bean parent="TextControlDefinition" p:size="2"/>
    </property>
    <property name="summary" value="Campus Code"/>
    <property name="name" value="campusCode"/>
    <property name="label" value="Campus Code"/>
    <property name="description" value="The code uniquely identifying a particular campus."/>
</bean>

<bean id="Campus-campusTypeCode-parentBean" abstract="true" parent="AttributeDefinition">
    <property name="forceUppercase" value="true"/>
    <property name="shortLabel" value="Type"/>
    <property name="maxLength" value="2"/>
    <property name="validationPattern">
        <bean parent="AlphaNumericValidationPattern"/>
    </property>
    <property name="required" value="true"/>
    <property name="control">
        <bean parent="SelectControlDefinition"
 p:valuesFinderClass="org.kuali.rice.kns.keyvalues.CampusTypeValuesFinder" p:includeKeyInLabel="false"/>
    </property>
    <property name="summary" value="Campus Type Code"/>
    <property name="name" value="campusTypeCode"/>
    <property name="label" value="Campus Type Code"/>
    <property name="description" value="The code identifying type of campus."/>
</bean>
```

In client applications, it is common that several business objects share a field representing the same type of data. For example, a country's postal code may occur in many different tables. In these circumstances, the use of a parent bean reference (parent="Country-postalCountryCode") definition allows the reuse of parts of a standard definition from the "master" business object. For instance, the StateImpl business object (business object data dictionary file State.xml) references the postalCountryCode property of the CountryImpl (business object data dictionary file Country.xml). Because the **postalCountryCode** fields in StateImpl and CountryImpl are identical, a simple attribute definition bean in the Business Object data dictionary file (State.xml) can be used:

```
<bean id="State-postalCountryCode" parent="Country-postalCountryCode-parentBean"/>
```

The definition of the **Country-postalCountryCode-parentBean** bean is seen inside the Country.xml file (for the CountryImpl business object):

```
<bean id="Country-postalCountryCode-parentBean" abstract="true" parent="AttributeDefinition">
    <property name="name" value="postalCountryCode"/>
    <property name="forceUppercase" value="true"/>
    <property name="label" value="Country Code"/>
    <property name="shortLabel" value="Country Code"/>
    <property name="maxLength" value="2"/>
```

```
    <property name="validationPattern">
        <bean parent="AlphaNumericValidationPattern"/>
    </property>
    <property name="required" value="true"/>
    <property name="control">
        <bean parent="TextControlDefinition" p:size="2"/>
    </property>
    <property name="summary" value="Postal Country Code"/>
    <property name="description" value="The code uniquely identify a country."/>
</bean>
```

This type of definition (defining the attribute definition once and reusing the bean as a parent bean) can be used inside common files as well. Rice has an AttributeReferenceDummy.xml business object data dictionary file as well as a java object AttributeReferenceDummy.java file. This file's sole purpose is to place commonly defined attributes such as **versionNumber** (which is common across all business objects) in a central location so that other business object attribute definitions can use them as parent beans. Here is how the Campus business object uses the version number attribute:

```
<bean id="Campus-versionNumber-parentBean" abstract="true" parent="AttributeReferenceDummy-versionNumber">
```

All business object data dictionary files need to have the version number field bean defined. This will verify that the UI will have the version number as a hidden field.

Business Object Data Dictionary Lookup Definition

Lookup Fields

A lookup definition contains a property called lookupFields which is made up of a list of FieldDefinitions. These specify the fields that will be displayed on a lookup form for that business object. A typical lookupField (shown here with the parent property for context) in the Spring configuration for a Business Object will look like this:

```
<property name="lookupFields">
    <list>
        ...
        <bean parent="FieldDefinition" p:attributeName="campusCode"/>
        ...
    </list>
</property>
```

Lookup default values

You can set a global default for that lookup field using the defaultValue property:

```
<bean parent="FieldDefinition" p:attributeName="campusCode" p:defaultValue="BL"/>
```

The effect of this is that every time the lookup for this Business Object is rendered, the campusCode text input will have "BL" in it.

Quickfinders

A quickfinder is a button that is rendered next to a lookup field which takes you to a lookup for a related Business Object which that field references, which in the case of this example would be to a Campus Business Object.

Quickfinder parameters

If a lookup field will have a quickfinder button on it due to a BO relationship, you may wish to set default values for certain fields on that related Business Object's lookup form, but only when the quickfinder from this Business Object is used.

```
<bean parent="FieldDefinition" p:attributeName="campusCode"
 p:quickfinderParameterString="campusTypeCode=P,active=Y" />
```

The effect of this is different than the defaultValue in that the defaults apply not to the lookup form Business Object that we are currently defining lookupFields for, rather for specific fields in the related Business Object that this lookupField (campusCode) references – but only when accessed through this quickfinder on our parent BO's lookup form.

Example LookupDefinition with defaultValue and quickfinderParameterString

This is perhaps better explained through a simple example with two BOs that have a relationship, Building and Campus. Here is the LookupDefinition for Building:

```
<bean id="Building-lookupDefinition-parentBean" abstract="true" parent="LookupDefinition" p:title="Building
 Lookup">
    ...
    <property name="lookupFields">
        <list>
            ...
            <bean parent="FieldDefinition" p:attributeName="campusCode"
 p:quickfinderParameterString="campusTypeCode=P,active=Y" defaultValue="BL"/>
            ...
        </list>
    </property>
    ...
</bean>
```

The defaultValue is a global default, so every time you view the Building BO's lookup it will have "BL" in the campusCode input.

The quickfinderParameterString is much more localized, so if you go directly to the Campus BO's lookup it will have no effect. However, if you go to the Building BO's lookup and click the quickfinder button next to its campusCode input, the Campus BO's lookup it will have a default of "P" in the campusTypeCode input, and a default of "Y" in the active input.

There is a related property for FieldDefinition that also applies to lookups, the quickfinderParameterStringBuilderClass. This lets you specify a class (which must implement the org.kuali.rice.kns.lookup.valueFinder.ValueFinder interface) which will dynamically construct a quickfinderParameterString each time a lookup is rendered. This might be useful if e.g. you wanted to populate a field in the related BO's lookup with the current date and time when it is accessed through the quickfinder.

It is not valid to have both the quickfinderParameterString and the quickfinderParameterStringBuilderClass defined on a single FieldDefinition, and you will get an exception during Data Dictionary validation if you do so.

Totals

Support exists in the lookup framework for totaling the lookup results. If the 'total' property is set to true on one or more FieldDefinition within the resultFields, the total line will be rendered and totals displayed for each field indicated.

Example:

```
<property name="resultFields" >
    <list>
        <bean parent="FieldDefinition" p:attributeName="kemid" />
        <bean parent="FieldDefinition"
 p:attributeName="kemidObj.shortTitle" />
        <bean parent="FieldDefinition"
 p:attributeName="kemidObj.purposeCode" />
        <bean parent="FieldDefinition"
 p:attributeName="availableIncomeCash" p:total="true" />
        <bean parent="FieldDefinition"
 p:attributeName="availablePrincipalCash" p:total="true" />
        <bean parent="FieldDefinition"
 p:attributeName="availableTotalCash" p:total="true" />
        <bean parent="FieldDefinition"
 p:attributeName="kemidObj.close" />
    </list>
</property>
```

An additional row will be added to the lookup result table with the totals for each of these columns indicated. The label for the total row will display in the first lookup column. By default this label is set to 'TOTALS' and can be changed in KR-ApplicationResources.properties.

Figure 5.1. Totals

The total line will not be displayed for the column if the column values are masked.

One limitation of the totaling functionality is it will not work with a column that has inquiry URLs. This is because of the need to have a numeric value to sum on and for fields with an inquiry the URL is put into the tag value along with the actual cell value.

Disabling Search Buttons

In certain cases the search and clear buttons for a lookup are not needed. Therefore these buttons can be disabled in one of two ways.

The first way is to disable the buttons through the data dictionary. This is done by setting the property disableSearchButtons to true in the data dictionary lookup definition:

```
<bean id="CustomerProfile-lookupDefinition" parent="CustomerProfile-lookupDefinition-parentBean"/>
<bean id="CustomerProfile-lookupDefinition-parentBean" abstract="true" parent="LookupDefinition">
<property name="title" value="Customer Profile Lookup"/>
<property name="disableSearchButtons" value="true"/>
```

The second way is to disable the buttons for a particular instance of a lookup by passing disableSearchButtons=true as a request URL parameter:

```
http://localhost:8080/kr-dev/lookup.do?disableSearchButtons=true&more parms ...
```

Note in this scenario other calls to the lookup without this parameter will have the search buttons rendered.

Merging Custom Attributes into Lookup Definitions

There are instances when an institution would choose to add custom attributes to existing data dictionary definitions in a client application (such as KFS or KC). The lookup and result fields representing these custom attributes can be arranged as desired using the **DataDictionaryBeanOverride**. In the example below, we wish to add a custom attribute named **Campus Code** to KFS's existing Account bean.

```
<beans>
    ...
    <bean id="Account" parent="Account-parentBean">
    <property name="attributes">
        <list merge="true">
            <!-- list goes here -->
            <bean id="Account.campusCode" parent="Account-CampusCode" p:name="Account.campusCode" />
            ...
        </list>
    </property>
    </bean>
    ...
</beans>
```

Once the custom attribute is defined, we create a bean that takes KFS's **Account-lookupDefinition** bean and modifies it such that **Campus Code** is displayed right after the **Sub-Fund Group Code** attribute in the Account lookup screen and search results.

```
<beans>
    ...
    <bean id="Account-lookupDefinition-override" parent="DataDictionaryBeanOverride">
        <property name="beanName" value="Account-lookupDefinition" />
        <property name="fieldOverrides">
            <list>
                <!-- Place Campus Code after Account Sub-Fund Group Code in the lookup -->
                <bean parent="FieldOverrideForListElementInsert">
                    <property name="propertyName" value="lookupFields" />
                    <property name="propertyNameForElementCompare" value="attributeName" />
                    <property name="element">
                        <bean parent="FieldDefinition" p:attributeName="subFundGroupCode" />
                    </property>
                    <property name="insertAfter">
                        <list>
                            <bean parent="FieldDefinition" p:attributeName="Account.campusCode" />
                        </list>
                    </property>
                </bean>
                <!-- Place Campus Code after Account Sub-Fund Group Code in the search results -->
                <bean parent="FieldOverrideForListElementInsert">
                    <property name="propertyName" value="resultFields" />
                    <property name="propertyNameForElementCompare" value="attributeName" />
                    <property name="element">
                        <bean parent="FieldDefinition" p:attributeName="subFundGroupCode" />
                    </property>
                    <property name="insertAfter">
                        <list>
                            <bean parent="FieldDefinition" p:attributeName="Account.campusCode" />
                        </list>
                    </property>
                </bean>
            </list>
        </property>
    </bean>
    ...
```

```
</beans>
```

Merging Custom Attributes into Inquiry Definitions

There are instances when an institution would choose to add custom attributes to existing data dictionary definitions in a client application (such as KFS or KC). The fields representing these custom attributes can be arranged on the inquiry screen as desired using the **DataDictionaryBeanOverride**. In the example below, we wish to add a custom attribute named **Campus Code** to KFS's existing Account bean.

```
<beans>
    ...
    <bean id="Account" parent="Account-parentBean">
        <property name="attributes">
            <list merge="true">
                <!-- list goes here -->
                <bean id="Account.campusCode" parent="Account-CampusCode" p:name="Account.campusCode" />
                    ...
            </list>
        </property>
    </bean>
    ...
</beans>
```

Once the custom attribute is defined, we create a bean that takes KFS's **Account-inquiryDefinition** bean and modifies it such that **Campus Code** is displayed right after the **Sub-Fund Group Code** attribute in the Account inquiry screen.

```
<beans>
    ...
    <bean id="Account-inquiryDefinition-override" parent="DataDictionaryBeanOverride">
        <property name="beanName" value="Account-inquiryDefinition" />
        <property name="fieldOverrides">
            <list>
                <!-- Place Campus Code after Account Sub-Fund Group Code in the Account Details section
 (inquirySections[0]) -->
                <bean parent="FieldOverrideForListElementInsert">
                    <property name="propertyName" value="inquirySections[0].inquiryFields" />
                    <property name="propertyNameForElementCompare" value="attributeName" />
                    <property name="element">
                        <bean parent="FieldDefinition" p:attributeName="subFundGroupCode" />
                    </property>
                    <property name="insertAfter">
                        <list>
                            <bean parent="FieldDefinition" p:attributeName="Account.campusCode" />
                        </list>
                    </property>
                </bean>
            </list>
        </property>
    </bean>
    ...
</beans>
```

Document Data Dictionary Overview

There are two different document types in KNS:

1. **Maintenance Documents**

   Maintenance Documents create, update, copy, or inactivate either a single business object or a collection of business objects. They are used to perform standard maintenance on data.

2. **Transactional Documents**

> Transactional Documents represent an action that will occur in the system. They are treated as one-shot documents and need not be edited and modified several times because of their approach in performing an action.

Comparison of Maintenance and Transactional Documents

Table 5.2. Comparison of Maintenance and Transactional Documents

	Transactional Documents	Maintenance Documents
SQL Table(s)	yes	yes
OJB Mapping(s) - repository.xml	yes	yes
Business Object(s)	yes	yes
Data Dictionary File(s)(XML)	Transactional Document DD File	Maintenance Document DD File Business Object DD File (discussed earlier)

Each type of dictionary defines properties such as authorizations, rules and workflow document types.

The following examples all follow the same structure with respect to the use of 'abstract' parent beans for Data Dictionary beans. A detailed description of their use and why Kuali uses this type of implementation can be found in the beginning of the 'Business Object Data Dictionary' section.

Maintenance Document Data Dictionary Overview

In general, documents have metadata associated with them, and the metadata for maintenance documents exists in the document's data dictionary configuration. The data dictionary can do practically everything for a maintenance document: it declares the user interface for the form, ties rules and document authorizers to the document as well as the document's workflow document type.

Below is an example of a Maintenance Document Data Dictionary file from the KNS module itself. It is for the Parameter object used within the KNS. The path (or package) **org/kuali/rice/kns/document/ datadictionary/** is where the **ParameterMaintenanceDocument** can be found in Rice if below is difficult to view.

```
<bean id="ParameterMaintenanceDocument" parent="ParameterMaintenanceDocument-parentBean"/>

<bean id="ParameterMaintenanceDocument-parentBean" abstract="true" parent="MaintenanceDocumentEntry">
    <property name="businessObjectClass" value="org.kuali.rice.kns.bo.Parameter"/>
    <property name="maintainableClass" value="org.kuali.rice.kns.document.ParameterMaintainable"/>
    <property name="maintainableSections">
        <list>
            <ref bean="ParameterMaintenanceDocument-EditParameter"/>
        </list>
    </property>
    <property name="defaultExistenceChecks">
        <list>
            <bean parent="ReferenceDefinition" p:attributeName="parameterNamespace"
 p:attributeToHighlightOnFail="parameterNamespaceCode"/>
            <bean parent="ReferenceDefinition" p:attributeName="parameterType"
 p:attributeToHighlightOnFail="parameterTypeCode"/>
        </list>
    </property>
    <property name="lockingKeys">
        <list>
            <value>parameterNamespaceCode</value>
            <value>parameterDetailTypeCode</value>
            <value>parameterApplicationNamespaceCode</value>
            <value>parameterName</value>
```

```
            </list>
        </property>

    <property name="documentTypeName" value="ParameterMaintenanceDocument"/>
    <property name="businessRulesClass" value="org.kuali.rice.kns.rules.ParameterRule"/>
    <property name="documentAuthorizerClass"
 value="org.kuali.rice.kns.document.authorization.MaintenanceDocumentAuthorizerBase"/>
    <property name="workflowProperties">
        <ref bean="ParameterMaintenanceDocument-workflowProperties"/>
    </property>
</bean>



<!-- Maintenance Section Definitions -->
<bean id="ParameterMaintenanceDocument-EditParameter" parent="ParameterMaintenanceDocument-EditParameter-
parentBean"/>

<bean id="ParameterMaintenanceDocument-EditParameter-parentBean" abstract="true"
 parent="MaintainableSectionDefinition">
    <property name="maintainableItems">
        <list>
            <bean parent="MaintainableFieldDefinition" p:required="true" p:name="parameterNamespaceCode"/>
            <bean parent="MaintainableFieldDefinition" p:required="true" p:name="parameterDetailTypeCode"/>
            <bean parent="MaintainableFieldDefinition" p:required="true"
 p:name="parameterApplicationNamespaceCode" p:defaultValue="KUALI"/>
            <bean parent="MaintainableFieldDefinition" p:required="true" p:name="parameterName"/>
            <bean parent="MaintainableFieldDefinition" p:required="false" p:name="parameterValue"/>
            <bean parent="MaintainableFieldDefinition" p:required="true" p:name="parameterDescription"/>
            <bean parent="MaintainableFieldDefinition" p:required="true" p:name="parameterTypeCode"/>
            <bean parent="MaintainableFieldDefinition" p:required="true" p:name="parameterConstraintCode"/>
        </list>
    </property>
    <property name="id" value="Edit Parameter"/>
    <property name="title" value="Edit Parameter"/>
</bean>



<!-- Exported Workflow Properties -->

<bean id="ParameterMaintenanceDocument-workflowProperties" parent="ParameterMaintenanceDocument-
workflowProperties-parentBean"/>

<bean id="ParameterMaintenanceDocument-workflowProperties-parentBean" abstract="true"
 parent="WorkflowProperties">
    <property name="workflowPropertyGroups">
        <list>
            <bean parent="WorkflowPropertyGroup">
                <property name="workflowProperties">
                    <list>
                        <bean parent="WorkflowProperty" p:path="oldMaintainableObject.businessObject"/>
                        <bean parent="WorkflowProperty" p:path="newMaintainableObject.businessObject"/>
                    </list>
                </property>
            </bean>
        </list>
    </property>
</bean>
```

Basic Setup

The first bean defined for the ParameterMaintenanceDocument data dictionary file is the main definition bean "ParameterMaintenanceDocument-parentBean". This bean uses the parent bean "MaintenanceDocumentEntry". This is how this particular business object is defined specifically as a Maintenance Document. Inside the "ParameterMaintenanceDocument-parentBean" bean we see several properties being set:

```
<property name="businessObjectClass" value="org.kuali.rice.kns.bo.Parameter"/>
<property name="maintainableClass" value="org.kuali.rice.kns.document.ParameterMaintainable"/>
```

First and foremost the Maintenance Document Data Dictionary file should define the business object that will be maintained by this particular document using the **businessObjectClass** property. In this example the fully qualified business object class is **kuali.rice.kns.bo.Parameter**.

The Maintenance Documents also need a maintainable class. This is defined using the **maintainableClass** property and in our **Parameter** business object example the custom class being used is **org.kuali.rice.kns.document.ParameterMaintainable**. If there are no customizations needed for the business object then the default class **org.kuali.rice.kns.maintenance.KualiMaintainableImpl** should be used. More will be discussed about custom maintainable classes later in this document.

Existence Checking

The next maintenance document specific tag is **defaultExistenceChecks**. Certain document validations are so omnipresent that they can simply be declared - typically validations that certain fields of a document are required. Here are the default existence checks for the **ParameterMaintenanceDocument**:

```
<property name="defaultExistenceChecks">
    <list>
        <bean parent="ReferenceDefinition" p:attributeName="parameterNamespace"
 p:attributeToHighlightOnFail="parameterNamespaceCode"/>
    </list>
</property>
```

Here we have just one default existence check. Default existence checks verify that the associated business object for the document actually exist. For instance, in the Parameter maintenance document, if a user enters a parameter namespace value that does not exist, the default existence check will display an error message next to the **parameterNamespaceCode** attribute field after the user attempts to save or submit.

The **defaultExistenceCheck** tag has a few different ways it can operate. All involve setting a list of beans that use the "ReferenceDefinition" parent bean. This bean is defined in Rice and can be used by any Maintenance Document Data Dictionary file. The properties that may be set for the "ReferenceDefinition" beans vary but the example shows the most common. The **attributeName** property is set to the KNS attribute name of the business object which must exist for the check to pass. In this case the **Namespace** object in KNS has a **namespaceCode** attribute. Likewise the **attributeToHighlightOnFail** refers to the attribute in the **Parameter** business object that is used to link to the reference business object. This is the field which will be highlighted on the user interface for the error to display. Of course, for this to work correctly, the foreign keys to the fields must be specified as required. That will come into play in section below about specifying the UI.

Locking keys

Since maintenance documents edit one or more business objects, there is the potential for race conditions. For example, if two business objects were created with the same primary key field and they were both sent into routing at the same time, the first document that is approved to 'Final' status in Workflow could potentially be overwritten in the database by the second document when it goes to 'Final' status. The KNS attempts to prevent these situations from arising by creating a pessimistic lock on each business object going through workflow as part of a maintenance document. In most cases, it uses the **lockingKeys** tag of the data dictionary for the maintenance document to create that locking representation. Here's the locking representation configuration for the **ParameterMaintenanceDocument**:

```
<property name="lockingKeys">
    <list>
        <value>parameterNamespaceCode</value>
        <value>parameterDetailTypeCode</value>
        <value>parameterApplicationNamespaceCode</value>
        <value>parameterName</value>
```

```
        </list>
</property>
```

Not surprisingly, the attributes listed in the example are also the primary keys for the **Parameter** business object. The locking keys above simply mean that once a certain Parameter is put into Workflow routing with a certain set of the fields above, another document with the same exact values for all the attributes above will be prevented from being put into Workflow. The fields used in a locking key can be anything, as long as it marks the business object uniquely. It makes sense, then, that most locking keys are simply the primary keys for the business object.

Defining the UI

Finally, the largest part of the maintenance document data dictionary: the definition of the UI through the **maintenanceSections** property. The UI of a maintenance document is made up of one or more maintainable sections. Each section is named, and each section creates a new tab as its visual representation on the web form. Here is the section list property being set on the "ParameterMaintenanceDocument-parentBean" bean (only one section in this document):

```
<property name="maintainableSections">
    <list>
        <ref bean="ParameterMaintenanceDocument-EditParameter"/>
    </list>
</property>
```

The list of beans is defined in the main Maintenance Document Entry bean while each 'Section Definition' bean is defined below in the file. Here is the **ParameterMaintenanceDocument** example of the "ParameterMaintenanceDocument-EditParameter" bean definition:

```
<bean id="ParameterMaintenanceDocument-EditParameter-parentBean" abstract="true"
 parent="MaintainableSectionDefinition">
    <property name="maintainableItems">
        <list>
            <bean parent="MaintainableFieldDefinition" p:required="true" p:name="parameterNamespaceCode"/>
            <bean parent="MaintainableFieldDefinition" p:required="true" p:name="parameterDetailTypeCode"/>
            <bean parent="MaintainableFieldDefinition" p:required="true"
 p:name="parameterApplicationNamespaceCode" p:defaultValue="KUALI"/>
            <bean parent="MaintainableFieldDefinition" p:required="true" p:name="parameterName"/>
            <bean parent="MaintainableFieldDefinition" p:required="false" p:name="parameterValue"/>
            <bean parent="MaintainableFieldDefinition" p:required="true" p:name="parameterDescription"/>
            <bean parent="MaintainableFieldDefinition" p:required="true" p:name="parameterTypeCode"/>
            <bean parent="MaintainableFieldDefinition" p:required="true" p:name="parameterConstraintCode"/>
        </list>
    </property>
    <property name="id" value="Edit Parameter"/>
    <property name="title" value="Edit Parameter"/>
</bean>
```

Each maintainable section is defined by using the parent bean "MaintainableSectionDefinition". These beans, in turn, are made up of several different properties including an **id**, **title**, and **maintainableItems**. The **maintainableItems** property is a list of maintainable fields. Each maintainable field bean uses the "MaintainableFieldDefinition" bean as its parent bean and lists the attribute that should be shown. That attribute itself has typically been defined in the data dictionary configuration for the business object (see Business Object Data Dictionary Definition below). There is also a **required** property which can be set to force extra validation, though all validations described in the attributes of the business object will also be checked.

While attributes default to using the definition set up in the data dictionary for a given field, there are a couple of behavior modifications that can be made. One of which appears above in the **required** property.

This can override the default required behavior as defined for the business object on the Business Object Data Dictionary file. Below are demonstrations of how some of the various changes that can be made could potentially be done for the **ParameterMaintenanceDocument** data dictionary file. For instance, default values for any field can be set by using the **defaultValue** property, like so:

```
<bean parent="MaintainableFieldDefinition">
    <property name="name" value="parameterApplicationNamespaceCode"/>
    <property name="required" value="true"/>
    <property name="defaultValue" value="KUALI"/>
</bean>
```

The example above example sets the default value of the **parameterApplicationNamespaceCode** attribute to "KUALI".

Another property that can be used to set a field with a default value in the maintenance document data dictionary maintainableField beans is the **defaultValueFinderClass** property. This property should be set to a class that implements the interface class **org.kuali.rice.kns.lookup.valueFinder.ValueFinder**. The interface has one method only: **getValue**(), which returns a String which will be set into the form in the User Interface. Here is an example (not from the **ParameterMaintenanceDocument** but from the **IdentityManagementGenericPermissionMaintenanceDocument**) on how to use the **defaultValueFinderClass** property:

```
<bean parent="MaintainableFieldDefinition">
    <property name="name" value="permissionId"/>
    <property name="unconditionallyReadOnly" value="true"/>
    <property name="required" value="true"/>
    <property name="defaultValueFinderClass"
 value="org.kuali.rice.kim.lookup.valuefinder.NextPermissionIdValuesFinder"/>
</bean>
```

The above example pulls the next available id from a class for one of the Kuali Identity Management documents. This is a very custom behavior for KIM but does highlight just one way the **defaultValueFinderClass** can be used.

One other large customization that can be made is to modify the way the lookup on a particular field operates. Lookups will be described in detail later in this documentation. Below is a simulated example that does not exist in the Rice code:

```
<bean parent="MaintainableFieldDefinition">
    <property name="name" value="reconcilerGroup.groupName"/>
    <property name="required" value="true"/>
    <property name="overrideFieldConversions" value="groupId:cardGroupId,groupName:reconcilerGroup.groupName"/>
    <property name="overrideLookupClass" value="org.kuali.rice.kim.bo.impl.GroupImpl"/>
</bean>
```

The **overrideLookupClass** property will set the business object class of the lookup that should be used. This means in our example above that the lookup for the field "reconcilerGroup.groupName" will use the **org.kuali.rice.kim.bo.impl.GroupImpl** class lookup. The **overrideFieldConversions** property is used to translate data attributes from the **overrideLookupClass** to fields that match the current Business Object class for which the maintenance document data dictionary file is for. These are separated with the colon character and a comma is used to delineate each field translation if more than one is to be listed. In the example above the 'groupId' field (which exists on the **org.kuali.rice.kim.bo.impl.GroupImpl** class) will be set into the 'cardGroupId' field (which should exist on the business object class of the current maintainable data dictionary file). In some instances the **overrideFieldConversions** may not be necessary if the field names are the same on the lookup's business object class and the data dictionary's business object class.

Additional MaintainableFieldDefinition Properties

For each MaintainableFieldDefinition bean defined in a maintenance document, there are a few fields that can help adjust the User Interface for a KNS client. Here is a sample example:

```
      ...
1    <property name="maintainableItems">
2        <list>
3            <bean parent="MaintainableFieldDefinition" p:name="Code" p:required="true" />
4            <bean parent="MaintainableFieldDefinition" p:name="ID" p:unconditionallyReadOnly="true" />
5            <bean parent="MaintainableFieldDefinition" p:name="Name" p:readOnlyAfterAdd="true" />
6            <bean parent="MaintainableFieldDefinition" p:name="Type" p:lookupReadOnly="true" />
7            <bean parent="MaintainableFieldDefinition" p:name="linkedJob" p:noLookup="true" />
      ...
```

In the example above on line 4 the field with **name** value "ID" has a property named **unconditionallyReadOnly** that is set to "true". This means the field will be read only and uneditable in the User Interface at all times regardless of document state. This could be helpful when setting a default value that the user entering the document is not allowed to change.

The property **readOnlyAfterAdd** set to "true" on line 5 for the "Name" field means that once the maintenance document for this business object has been successfully saved and routed through all appropriate approvals, the "Name" field will be read only. This is useful in certain instances when creating a new business object.

The property **lookupReadOnly** in line 6 is used to change the UI so that a lookup link will be presented for the field but the value that is displayed when returning an object from the lookup is read only. In the example above the "Type" variable will have a lookup (as defined by the Business Object Data Dictionary file… see the Business Object Data Dictionary section for more information) but the displayed value in the UI for "Type" will be uneditable by user entry. It may still be changed by going to the lookup link again.

The noLookup property shown in line 7 for the "linkedJob" field is a way to override the default functionality coming from the Business Object Data Dictionary file. If that DD file has a Lookup control element but the lookup need to be hidden on the Maintenance Document then this attribute allows for that functionality.

Collections

Some maintenance documents include collections of business objects. Below is an example from the RoutingRuleMaintenanceDocument data dictionary file from Rice:

```
<bean id="RoutingRuleMaintenanceDocument-PersonResponsibilities-parentBean" abstract="true"
 parent="MaintainableSectionDefinition">
    <property name="id" value="PersonsMaintenance"/>
    <property name="title" value="Persons"/>
    <property name="maintainableItems">
        <list>
            <bean parent="MaintainableCollectionDefinition">
                <property name="name" value="personResponsibilities"/>
                <property name="businessObjectClass" value="org.kuali.rice.kew.rule.PersonRuleResponsibility"/>
                <property name="summaryTitle" value="Person"/>
                <property name="summaryFields">
                    <list>
                        <bean parent="MaintainableFieldDefinition" p:name="principalName"/>
                        <bean parent="MaintainableFieldDefinition" p:name="actionRequestedCd"/>
                    </list>
                </property>
                <property name="maintainableFields">
                    <list>
                        <bean parent="MaintainableFieldDefinition" p:name="principalName" p:required="true"/>
```

```
                        <bean parent="MaintainableFieldDefinition" p:name="actionRequestedCd"
 p:required="true"/>
                        <bean parent="MaintainableFieldDefinition" p:name="priority" p:required="true"/>
                    </list>
                </property>
                <property name="duplicateIdentificationFields">
                    <list>
                        <bean parent="MaintainableFieldDefinition" p:name="principalName"/>
                        <bean parent="MaintainableFieldDefinition" p:name="actionRequestedCd"/>
                    </list>
                </property>
            </bean>
        </list>
    </property>
</bean>
```

To put a collection into a maintenance section, simply put an instance of a **MaintainableCollectionDefinition** bean in the list that is set into the **maintainableItems** property of the maintenance section.

The **MaintainableCollectionDefinition** bean must have a **name** property. The **name** property should match the attribute name of the collection being maintained on the original business object. The **businessObjectClass** property value specifies the class of the items in the collection.

The **maintainableFields** property inside the **MaintainableCollectionDefinition** bean works exactly like the previously described structure of the **maintainableFields** property inside the **MaintainableSectionDefinition** bean. The only difference is that the name property of each **MaintainableFieldDefinition** refers to an attribute of the **businessObjectClass** that is set on the **MaintainableCollectionDefinition** bean.

The **summaryTitle** and **summaryFields** properties are used for display purposes once a list element is added to the list on the UI screen. The specified data elements will show when the full detail of the collection item is hidden using the 'hide/show' button functionality of the KNS. Usually these fields are specific to what uniquely defines the business objects contained within the collection.

The **duplicateIdentificationFields** property is used to identify specifically the set of fields inside the collection element business object that cannot be duplicated in the list. In this way they act as mini-locks. They will prevent more than one list element with the same set of fields. For instance, in the example above, if a list element already exists with the **actionRequestedCd** 'A' and the **principalName** 'john' then another list element with those same values cannot be added.

There are also a few more advanced type attributes that can be used. Take the above example and the abbreviated alteration below.

```
      ...
1    <property name="maintainableItems">
2        <list>
3            <bean parent="MaintainableCollectionDefinition" >
4                <property name="name" value="personResponsibilities"/>
5                <property name="includeAddLine" value="false"/>
6                <property name="businessObjectClass" value="org.kuali.rice.kew.rule.PersonRuleResponsibility"/
>
7                <property name="maintainableFields">
8                    <list>
9                        <bean parent="MaintainableFieldDefinition" p:name="newCollectionRecord"/>
    ...
```

The property **includeAddLine** on line 5 above is used to remove the UI element that allows the users to add their own elements to the list. This is helpful in cases where the list of items may be statically generated by code internal to the business object containing the collection.

On line 9 in the example above, the addition of the **MaintainableFieldDefinition** with the **name** property value of "newCollectionRecord" is used to tell the maintenance framework that any records currently existing in the collection are permanent - that is, there should not be delete buttons associated with them. However, if the property **includeAddLine** is set to "false" (or omitted) in the **MaintainableCollectionDefinition** bean above, new lines could be added to the collection and each of the new lines could be deleted (though lines that had been previously saved and routed appropriately into the collection could not be deleted).

Alternate/Additional Display Properties

Within the business object frameworks (lookup, inquiry, and maintenance document) an alternate or additional property can be specified to display when a field is read-only. These properties are configured through the data dictionary as follows:

alternateDisplayAttributeName

This property specifies an attribute on the business object that should be displayed instead of the field attribute when the view is read-only. The property is available on the FieldDefinition for lookup result fields and inquiries, and on the MaintainbleFieldDefinition for maintenance documents. In the case of lookup result fields and inquiries this attribute will always be displayed since the view is always read-only. For maintenance documents, the field attribute will display when the document is editable, and the alternate attribute will display when the document is read-only.

```
<bean id="CustomerProfile-lookupDefinition" parent="CustomerProfile-lookupDefinition-parentBean"/>
<bean id="CustomerProfile-lookupDefinition-parentBean" abstract="true" parent="LookupDefinition">
    <property name="title" value="Customer Profile Lookup"/>

    <property name="defaultSort">
        <bean parent="SortDefinition">
            <property name="attributeNames">
                <list>
                    <value>id</value>
                </list>
            </property>
        </bean>
    </property>
    <property name="lookupFields">
        <list>
            <bean parent="FieldDefinition" p:attributeName="id"/>
            <bean parent="FieldDefinition" p:attributeName="chartCode"/>
            <bean parent="FieldDefinition" p:attributeName="unitCode"/>
            <bean parent="FieldDefinition" p:attributeName="subUnitCode"/>
            <bean parent="FieldDefinition" p:attributeName="active"/>
        </list>
    </property>
    <property name="resultFields">
        <list>
            <bean parent="FieldDefinition" p:attributeName="id" p:alternateDisplayAttributeName="customerName"/
>
            <bean parent="FieldDefinition" p:attributeName="customerShortName"/>
            <bean parent="FieldDefinition" p:attributeName="customerDescription"/>
            <bean parent="FieldDefinition" p:attributeName="contactFullName"/>
            <bean parent="FieldDefinition" p:attributeNa ="processingEmailAddr"/>
            <bean parent="FieldDefinition" p:attributeName="defaultPhysicalCampusProcessingCode"/>
            <bean parent="FieldDefinition" p:attributeName="active"/>
            <bean parent="FieldDefinition" p:attributeName="defaultChartCode"/>
        </list>
    </property>
</bean>
```

In the example above, for the result field 'id' we have specified an alternateDisplayAttibuteName equal to "customerName". When the results are rendered the value of customerName property will be displayed and not the value of the id property. This behavior is the same within an InquiryDefinition.

If specified on a MaintainableFieldDefinition, again the value for the alternateDisplayAttributeName attribute will be displayed; however any quickfinder or lookup URL will be built using the field property as usual. If the field is editable or hidden, the value of the field property will be used.

additionalDisplayAttributeName

This property behaves much like the alternateDisplayAttributeName, the only difference being the value of the additionalDisplayAttributeName attribute will be appended to the value of the field attribute, using '*-*' as a delimiter.

Neither the alternateDisplayAttributeName nor additionalDisplayAttributeName need to have an AttributeDefinition defined, however they must have an accessible getter in the business object.

Automatic Translation of KualiCode fields

If enabled, fields that have references to a KualiCode class will be found and the corresponding KualiCode name field will be set as the additionalDisplayAttributeName. The object property holding the reference must also prefix the field name. For example, a field name of 'defaultChartCode' and reference name of 'defaultChart' would match, again assuming the type of 'defaultChart' implements KualiCode.

This automatic translation of code fields is turned on by default in the Inquiry framework, but turned off by default in lookups and maintenance documents. It can be configured for each MaintenanceDocumentEntry, LookupDefinition, or InquiryDefinition with the property 'translateCodes'.

For example, in the MaintenanceDocumentEntry:

```
<bean id="CustomerProfileMaintenanceDocument-parentBean" abstract="true" parent="MaintenanceDocumentEntry">
    <property name="businessObjectClass" value="org.kuali.kfs.pdp.businessobject.CustomerProfile"/>
    <property name="maintainableClass"
 value="org.kuali.kfs.pdp.document.datadictionary.CustomerProfileMaintenanceDocumentMaintainableImpl"/>
    <property name="maintainableSections">
        <list>
            <ref bean="CustomerProfileMaintenanceDocument-EditCustomerProfileSection1"/>
            <ref bean="CustomerProfileMaintenanceDocument-EditCustomerProfileSection2"/>
            <ref bean="CustomerProfileMaintenanceDocument-EditCustomerProfileSection3"/>
            <ref bean="CustomerProfileMaintenanceDocument-EditCustomerBank"/>
        </list>
    </property>
    <property name="defaultExistenceChecks">
        <list>
            <bean parent="ReferenceDefinition" p:attributeName="defaultChart"
 p:attributeToHighlightOnFail="defaultChartCode"/>
            <bean parent="ReferenceDefinition" p:attributeName="defaultAccount"
 p:attributeToHighlightOnFail="defaultAccountNumber"/>
            <bean parent="ReferenceDefinition" p:attributeName="defaultObject"
 p:attributeToHighlightOnFail="defaultObjectCode"/>
            <bean parent="ReferenceDefinition" p:attributeName="defaultProcessingCampus"
 p:attributeToHighlightOnFail="defaultPhysicalCampusProcessingCode"/>
            <bean parent="ReferenceDefinition" p:attributeName="state"
 p:attributeToHighlightOnFail="stateCode"/>
            <bean parent="ReferenceDefinition" p:attributeName="postalCode"
 p:attributeToHighlightOnFail="zipCode"/>
            <bean parent="ReferenceDefinition" p:attributeName="country"
 p:attributeToHighlightOnFail="countryCode"/>
            <bean parent="ReferenceDefinition" p:attributeName="transactionType"
 p:attributeToHighlightOnFail="achTransactionType"/>
            <bean parent="ReferenceDefinition" p:collection="customerBanks" p:attributeName="disbursementType"
 p:attributeToHighlightOnFail="disbursementTypeCode"/>
            <bean parent="ReferenceDefinition" p:collection="customerBanks" p:attributeName="bank"
 p:attributeToHighlightOnFail="bankCode"/>
        </list>
    </property>
    <property name="lockingKeys">
        <list>
            <value>chartCode</value>
            <value>unitCode</value>
```

```
        <value>subUnitCode</value>
    </list>
</property>
<property name="translateCodes" value="true"/>
```

If alternateDisplayAttributeName is specified for a field then it will override the code translation (if applicable).

Note the Summarizable interface and SummarizableFormatter class were removed as part of this work. If an application class implemented Summarizable it should be changed to implement the KualiCode interface.

Dynamic read-only, hidden, and required Field states

Within the KNS lookup and maintenance frameworks there is support for dynamically altering the read-only, hidden, or required states of a field. This functionality is configured through the data dictionary and java code as follows:

Conditional Logic

Any conditional logic that is necessary to determine whether a field should be read-only, hidden, or required (and editable) is implemented with java code. For maintenance documents this code is placed in the presentation controller. The following methods are available for this purpose:

```
public Set<String> getConditionallyReadOnlyPropertyNames(MaintenanceDocument document)


public Set<String> getConditionallyRequiredPropertyNames(MaintenanceDocument document)

public Set<String>
getConditionallyHiddenPropertyNames(BusinessObject businessObject)
```

Each of these methods returns a Set of field names (prefixing for the maintainable is not necessary). These fields will then take on the state determined by the method. The first two methods take as a parameter the MaintenanceDocument instance which can be used to get the current values for one or more fields. The third method is more general (because it is used for inquires as well) and takes a BusinessObject instance as a parameter. Within the maintenance context this will again be the MaintenanceDocument and can be cast after doing an instanceof check.

Example:

```
@Override
public Set<String> getConditionallyRequiredPropertyNames(MaintenanceDocument document) {
    Set<String> required = new HashSet<String>();
    SubAccount subAccount = (SubAccount) document.getNewMaintainableObject().getBusinessObject();
    if (StringUtils.isNotBlank(subAccount.getFinancialReportChartCode()) &&
 subAccount.getFinancialReportChartCode().equals("BL")) {
        required.add("a21SubAccount.costShareChartOfAccountCode");
        required.add("a21SubAccount.costShareSourceAccountNumber");
    }
    return required;
}
```

Only fields that have conditional states need to be considered here. For fields that are always read-only, hidden, or required the corresponding properties on the MaintainableFieldDefinition can be set to true through the data dictionary.

Sections of the maintenance document can also be conditionally set to read-only or hidden by implementing the following methods within the presentation controller:

```
public Set<String> getConditionallyReadOnlySectionIds(
MaintenanceDocument document);
public Set<String> getConditionallyHiddenSectionIds(BusinessObject businessObject);
```

Any authorization restrictions will be applied after this logic by the document authorizer class.

For lookups conditional logic is implemented in the LookupableHelperService. Similar methods exist for determining the read-only, hidden, or required states:

```
public Set<String> getConditionallyReadOnlyPropertyNames();
public Set<String> getConditionallyRequiredPropertyNames();

public Set<String> getConditionallyHiddenPropertyNames();
```

Each of these methods returns a Set of field names. Code implemented within these methods has access to the lookupable helper properties. In particular the request parameters can be retrieved using getParameters(), and the current rows using getRows(). The following convenience method is also available for getting a property value from the field:

```
protected String getCurrentSearchFieldValue(String propertyName)
```

It is recommended to use this method to get a value for a property as opposed to the request parameters, since the values could be different. This is because the conditional logic is applied at the end of the lookup lifecycle and field values could have been cleared or set to other values by processing code. Therefore basing conditional logic off these values will correctly reflect the values being returned to the search fields.

Example:

```
@Override
public Set<String> getConditionallyHiddenPropertyNames() {
    Set<String> hiddenPropertyNames = new HashSet<String>();

    String employeeId = getCurrentSearchFieldValue(KIMPropertyConstants.Person.EMPLOYEE_ID);
    if (StringUtils.isNotBlank(employeeId)) {
        hiddenPropertyNames.add(KFSPropertyConstants.VENDOR_NUMBER);
        hiddenPropertyNames.add(KFSPropertyConstants.VENDOR_NAME);
    }
    return hiddenPropertyNames;
}
```

Trigger Fields

The second part to implementing conditional logic is indicating which fields should trigger a refresh (page post) when its value changes. The page post will call each of the conditional methods so when the page renders the read-only, required, and hidden attributes are set according to the new field value (Note all field values are available to the conditional methods regardless of which triggered the refresh). To indicate a field should trigger a refresh, set the triggerOnChange attribute to true on the MaintainableFieldDefinition:

```
<bean parent="MaintainableFieldDefinition" p:name="financialReportChartCode" p:triggerOnChange="true"/>
```

For lookups, set the triggerOnChange attribute to true on the lookup FieldDefinition within the lookupFields property:

```
<property name="lookupFields" >
    <list>
        <bean parent="FieldDefinition" p:attributeName="payeeTypeCode"/>
        <bean parent="FieldDefinition" p:attributeName="taxNumber" />
        <bean parent="FieldDefinition" p:attributeName="firstName" />
        <bean parent="FieldDefinition" p:attributeName="lastName" />
        <bean parent="FieldDefinition" p:attributeName="vendorNumber" p:triggerOnChange="true" />
        <bean parent="FieldDefinition" p:attributeName="vendorName" />
        <bean parent="FieldDefinition" p:attributeName="employeeId" p:triggerOnChange="true" />
        <bean parent="FieldDefinition" p:attributeName="entityId" p:triggerOnChange="true" />
        <bean parent="FieldDefinition" p:attributeName="active"/>
    </list>
</property>
```

There is no limit to the number of trigger fields specified for a maintenance document or lookup.

Note

JavaScript was implemented to set the focus back to the next field in the tab order (from the field that triggered the refresh) when the page refreshes. This will not work correctly if fields are inserted between the field that triggered a refresh and the next tab field (for instance if a field between these two was hidden or read-only, and becomes editable on refresh).

Configuring a KNS Client in Spring

The Kuali Nervous System (KNS) is installed as a Rice Module using Spring. The primary source for configuring Spring in KNS is the KnsTestSpringBeans.xml file located in the **/kns/src/test/ resources/** directory. This file uses the **PropertyPlaceholderConfigurer** bean to load tokens for runtime configuration using the source file *kns-test-config.xml* located in the **/kns/src/test/resources/META-INF** directory.

The *kns-test-config.xml file* contains this code snippet:

```
<param name="module.name">sample-app</param>
<param name="service.namespace">RICE</param>
<param name="filter.login.class">org.kuali.rice.kew.web.DummyLoginFilter</param>
<param name="filtermapping.login.1">/*</param>
<param name="config.location">classpath:META-INF/test-config-defaults.xml</param>
<param name="serviceServletUrl">http://localhost:9916/${app.context.name}/remoting/</param>
<param name="transaction.timeout">3600</param>

<param name="config.location">classpath:META-INF/common-config-test-locations.xml</param>

<param name="config.location">${alt.config.location}</param>
<param name="kns.test.port">9916</param>
```

This is a combination of key value pairs. When used in conjunction with Spring tokenization and the **PropertyPlaceholderConfigurer** bean, the parameter name must be equal to the key value in the Spring file so that the properties propagate successfully.

Spring JTA Configuration

When doing persistent messaging, it is best to use JTA as your transaction manager. This ensure the messages you are sending are synchronized with the current executed transaction in your application and also allows message persistence to be put in a different database than the application's logic, if needed. Currently, KNSTestSpringBeans.xml uses JOTM to configure JTA without an application server. Below is the bean definition for JOTM that can be found in Spring.

```
<bean id="transactionManagerXAPool" class="org.springframework.transaction.jta.JotmFactoryBean">
    <property name="defaultTimeout" value="${transaction.timeout}"/>

</bean>
<bean id="dataSource" class="org.kuali.rice.database.XAPoolDataSource">
    <property name="transactionManager" ref="transactionManagerXAPool" />
    <property name="driverClassName" value="${datasource.driver.name}" />
    <property name="url" value="${datasource.url}" />
    <property name="maxSize" value="${datasource.pool.maxSize}" />
    <property name="minSize" value="${datasource.pool.minSize}" />
    <property name="maxWait" value="${datasource.pool.maxWait}" />
    <property name="validationQuery" value="${datasource.pool.validationQuery}" />
    <property name="username" value="${datasource.username}" />
    <property name="password" value="${datasource.password}" />
</bean>
```

Configure the **TransactionManager**, **UserTransaction** and a **DataSource**. Use the Rice **XAPoolDataSource** class as your data source because it addresses some bugs in the **StandardXAPoolDataSource**, which extends from this class.

KNS Validation and Business Rules Framework

When actions are performed on documents, there is typically some validation to accomplish on the document; indeed, a great deal of the business logic for client application is stored in document validations. The KNS supports a standard framework for validations as well as a way to display errors to application end users.

Rules and Events

KNS validations are performed by rules classes, which respond to a specific application event. An event is an object which encapsulates contextual information about something which has been requested of a document. For instance, when a user on a maintenance document clicks a "Route" button to route the document into workflow, the web-layer controller creates an instance of **org.kuali.rice.kns.rule.event.RouteDocumentEvent** which holds the document which has just been routed. It then passes this event instance to **org.kuali.rice.kns.service.KualiRuleService**.

The KualiRuleService interrogates the data dictionary entry for the document to find a rules class. The event then invokes the rules class against itself. This is accomplished through a rule interface. Every event has an associated rule interface; the class of this interface is returned by the Event's **getRuleInterfaceClass()** method. The event will cast the business rule from the data dictionary to the interface which it expects, and then call a standard method against that interface.

An example will clarify this. RouteDocumentEvent expects rules implementing the rule interface **org.kuali.rice.kns.rule.RouteDocumentRule**, which extends the BusinessRule interface given above. RouteDocumentRule has a single method to implement:

```
public boolean processRouteDocument(Document document);
```

When the KualiRuleService gets the event, it finds the data dictionary entry for the given document and generates an instance of the business rules class associated with the document. It then hands that to the event, which attempts to perform the cast to RouteDocumentRule and call the processRouteDocument method:

```
public boolean invokeRuleMethod(BusinessRule rule) {
    return ((RouteDocumentRule) rule).processRouteDocument(document);
```

```
}
```

It then returns whatever was returned by the rule.

This brings up the question of what the processRouteDocument method should actually do. Rule methods need to accomplish two things:

1. Run the business logic associated with that event against the document. If the business logic decides the document is valid, then a true should be returned. If the business logic, contrarily, decides the document is not valid, a false is typically returned. The result of the method invocation then typically determines whether the given event will be completed. For instance, if **processRouteDocument** returns a false, then the document – which has only had a workflow route requested of it – will fail to route. It will instead return to the document screen.

2. Some kind of user message should be recorded in the **GlobalVariables.getMessages()** thread-local singleton. This singleton has three maps, accessible through the **getErrorMap()**, **getWarningMap()**, and **getInfoMap()** methods. These maps associate an attribute on the page which caused a failure with a user message explaining the problem. If a false is returned from the method, then it is generally expected that the failure will be recorded in the Error map.

An excellent example of this can be found in the sample "Recipe application" which ships with Rice, in edu.sampleu.recipe.document.rule.RecipeRules:

```
@Override
protected boolean processCustomSaveDocumentBusinessRules(MaintenanceDocument document) {
    boolean valid = super.processCustomSaveDocumentBusinessRules(document);
    if (valid) {
        valid &= validateIngredients(document);
    }
    return valid;
}

private boolean validateIngredients(MaintenanceDocument recipeDocument) {
    Recipe recipe = (Recipe) recipeDocument.getDocumentBusinessObject();
    String ingredients = recipe.getIngredients();
    RecipeCategory category = recipe.getCategory();
    if (category != null) {
        String categoryName = recipe.getCategory().getName();
        if(StringUtils.containsIgnoreCase(ingredients, "beef") && !StringUtils.equalsIgnoreCase(categoryName,
 "beef")) {
            putFieldError("categoryId", "error.document.maintenance.recipe.ingredients.beef");
            return false;
        }
    }
    return true;
}
```

In this example, the processCustomSaveDocumentBusinessRules is called when the document is saved. In turn, the validateIngredients method is called. It checks that if the category is not null, then if "beef" is among the ingredients, then the categoryName of the recipe must include the word "beef" in it. If that is the case, we see that the putFieldError – a convenience method – adds the user message to the "categoryId" attribute (meaning the error message will be displayed close to that attribute) and that false is returned, meaning that the save is not carried out.

Standard KNS Events

There are eight common KNS events which apply to every document – maintenance and transactional – built within client applications. For each, the KNS does an amount of standard validation, while leaving customization points so client applications can add more validation business logic. They are:

Table 5.3. KNS Events

Event	Calling circumstances	Rule interface and method called	Validation performed in DocumentRuleBase
org.kuali.rice.kns.rule.event.RouteDocumentEvent	Called when a document is routed to workflow.	org.kuali.rice.kns.rule.RouteDocumentRule#processRouteDocument	Performs standard data dictionary validation
org.kuali.rice.kns.rule.event.SaveDocumentEvent	Called when a document is saved.	org.kuali.rice.kns.rule.SaveDocumentRule#processSaveDocument	Performs standard data dictionary validation
org.kuali.rice.kns.rule.event.ApproveDocumentEvent	Called when a workflow action is taken against a document.	org.kuali.rice.kns.rule.ApproveDocumentRule#processApproveDocument	
org.kuali.rice.kns.rule.event.BlanketApproveDocumentEvent	Called when a document is blanket approved through workflow.	org.kuali.rice.kns.rule.ApproveDocumentRule#processApproveDocument	
org.kuali.rice.kns.rule.event.AddNoteEvent	Called when a note is added to a document.	org.kuali.rice.kns.rule.AddNoteRule#processAddNote	Validates the note (via the data dictionary)
org.kuali.rice.kns.rule.event.AddAdHocRoutePersonEvent	Called when an ad hoc Person to route to is added to a document.	org.kuali.rice.kns.rule.AddAdHocRoutePersonRule#processAdHocRoutePerson	Validates that the ad hoc route Person is valid – that the Person's record exists and that the Person has the permission to approve the document
org.kuali.rice.kns.rule.event.AddAdHocRouteWorkgroupEvent	Called when an Ad Hoc workgroup to route to is added to a document.	org.kuali.rice.kns.rule.AddAdHocRouteWorkgroupRoute#processAddAdHocRouteWorkgroup	Validates the ad hoc route workgroup – that the workgroup exists and that the workgroup has permission to receive an ad hoc request and approve the document.
org.kuali.rice.kns.rule.event.SendAdHocRequestsEvent	Called when the end user requests that ad hoc events be sent.		

Since the standard events have to perform standard validation, they have custom methods to override. For instance, org.kuali.rice.kns.rules.DocumentRuleBase has a method "processCustomRouteDocumentBusinessRules" and it is expected that client applications will override this method rather than processRouteDocumentBusinessRules directly.

Maintenance documents add another event to this: org.kuali.rice.kns.rule.event.KualiAddLineEvent. This is invoked when a new item is added to a collection on a maintenance document. The org.kuali.rice.kns.maintenance.rules.MaintenanceDocumentRuleBase also contains a number of useful utility methods which makes writing business rules for maintenance documents easier.

Notifying Users of Errors

When a validation results in some kind of text being displayed to the user, **GlobalVariables.getMessageMap()** is used to store that text and is inquired during rendering to make sure messages are correctly displayed. As mentioned previously, the MessageMap is made up of three different maps: one for errors, one for warnings, and one for information messages. Each map has a "put" command – for instance, putError; each has a "has" predicate, such as "hasErrors"; and each have the ability to get the properties with form the keys of the map as well as any messages associated with that property. Adding, an error message to the map is easy, as seen in this example from the IdentityManagementGroupDocument:

```
GlobalVariables.getMessageMap().putError("document.member.memberId", RiceKeyConstants.ERROR_EMPTY_ENTRY, new
 String[] {"Member Type Code and Member ID"});
```

The method takes the property that the error is most associated with, which determines where the text will be displayed (ie, at the top of the section which contains the given property); a key to the User Message containing the error; and an array of Strings which will be interpolated into the message using the standard Java **java.text.MessageFormat**.

Further details about the use of User Messages can be found in the KNS User Messages section.

Creating New Events

While the vast majority of maintenance documents in client applications will not have custom actions, it is common in transactional documents to have new events beyond the standard ones provided by the KNS framework. Basically, any button created on a transactional document – one which results in a call to a method in the transactional document's action class – may well have an event associated with it. In that case, there are three pieces to create for the rule: the new event, the rule instance which is called from that event, and the default implementation for that rule.

An example from Kuali Financial Systems 3.0 will illustrate how these are used. The Cash Control transactional document in the Accounts Receivable module has a collection of details, added via an "add" button. To validate that action, an event was created (this code has slightly been altered for the sake of illustration):

```
package org.kuali.kfs.module.ar.document.validation.event;

public final class AddCashControlDetailEvent extends KualiDocumentEventBase {
    private final CashControlDetail cashControlDetail;

    public AddCashControlDetailEvent(String errorPathPrefix, Document document, CashControlDetail
 cashControlDetail) {
        super("Adding cash control detail to document " + getDocumentId(document), errorPathPrefix, document);
        this.cashControlDetail = cashControlDetail;
    }

    public Class getRuleInterfaceClass() {
```

```
        return AddCashControlDetailRule.class;
    }

    public boolean invokeRuleMethod(BusinessRule rule) {
        return ((AddCashControlDetailRule)
 rule).processAddCashControlDetailBusinessRules((TransactionalDocument) getDocument(), this.cashControlDetail);
    }

}
```

The AddCashControlDetailEvent extends the KualiDocumentEventBase class, defined in the KNS. Note that it encapsulates the state to check – both the document at hand and the cash control detail which is being validated. Finally, it implements the two methods which make the rule work: the **getRuleInterfaceClass()** and the **invokeRuleMethod()**. This works precisely as it does in the KNS RouteDocumentEvent.

The AddCashControlDetailRule looks like this:

```
public interface AddCashControlDetailRule<F extends TransactionalDocument > {
    public boolean processAddCashControlDetailBusinessRules(F transactionalDocument, CashControlDetail
 cashControlDetail);
}
```

This is very straightforward. There is a rules class, in turn, which implements this interface. Finally, the rules have to be called; that occurs when an event is created and sent to the KualiRuleService, which is typically done in the web layer's controller. In our example, this occurs in the CashControlDocumentAction:

```
// apply rules for the new cash control detail
rulePassed &= ruleService.applyRules(new
 AddCashControlDetailEvent(ArConstants.NEW_CASH_CONTROL_DETAIL_ERROR_PATH_PREFIX, cashControlDocument,
 newCashControlDetail));
```

Now the new action will be validated properly.

KNS User Messages

Functional users need a simple way to change wording of messages used throughout a KNS client application. Those messages may even be in a language foreign to that of the Foundation shipped messages (which are shipped in English). To facilitate ease of message changing, the KNS builds functionality on top of the standard Java message properties mechanism.

Once the Rice application has been generated, in src/main/resources, there will be a file named configurationServiceData.xml. That file lists a number of properties files which will be loaded:

```
<configuration>
    <properties fileName="KR-ApplicationResources.properties" />
    <properties fileName="KIM-ApplicationResources.properties" />
</configuration>
```

Each of these files are listed relative to the src/main/resources directly. A property file simply relates messages to keys, like so (from the src/main/resources/KR-ApplicationResources.properties file):

```
document.question.cancel.text=Are you sure you want to cancel?

document.question.delete.text=Are you sure you want to delete?
```

```
document.question.deleteContext.text=Are you sure you want to delete [b]{0}[/b]?

document.question.disapprove.text=Are you sure you want to [b]disapprove[/b] this document?

document.question.saveBeforeClose.text=Would you like to save this document before you close it?
```

This is the standard Java property file format, with keys (for instance, "document.question.cancel.text") related to messages.

A message may also have escaped HTML tags and templated positions in the text for other Strings to be interpolated in. An example of this is found in the "document.question.deleteContext.text" message. The [b] and [/b] will be translated automatically to bold markup. The {0} will be replaced, if possible, by another String. An example of this will be covered below.

Rice Best Practices suggest that each module in the client application have a KeyConstants class which relates the names of user message keys to the String constants. **org.kuali.rice.kns.util.RiceKeyConstants** is the key constants class for the KNS.

Developers of client applications can also override pre-existing messages. Messages are loaded in the order listed in the configurationServiceData.xml file above, so client application specific files should be listed later in the file. Then, if the client application user message file redefines a user message using the same key, as so:

```
document.question.cancel.text=Canceling will lead to permanent disuse of this document.  Are you completely
 certain this is the action you want to take?
```

users will be treated to the longer, more worried user message.

Retrieving User Messages

Retrieving the text for user messages can be done in a number of ways, based on the context the user message occurs in. The easiest use case is to get the text of the message directly through the default implementation of **org.kuali.rice.kns.service.KualiConfigurationService**. It has a method, **getPropertyString**, which, when handed the key to the message, returns the message text.

```
final String message =
 KNSServiceLocator.getKualiConfigurationService().getPropertyString(RiceKeyConstants.ERROR_DATE_TIME);
```

This will return the String "{0} is not a valid date/time." Note that in this case, the String will not be interpolated; **java.text.MessageFormat** should be used to switch the {0} with an actual, useful String.

KualiConfigurationService also has a method, **getPropertyAsBoolean**, which translates the messages (regardless of case) of "true", "yes", "on", or "1" as a boolean true and everything else as a false.

Error Messages

The vast majority of user messages are warnings when an error occurs. Thankfully, as was seen in the section on validations, the KNS handles error messages through the user messages system. For instance, in this code:

```
GlobalVariables.getErrorMap().putError("someProperty", ClientApplicationConstants.ERROR_MESSAGE, new String[]
 { businessObject.getSomeProperty().toString() });
```

The error message displayed will be the one with the key held by **ClientApplicationConstants.ERROR_MESSAGE**, and the value of **businessObject.getSomeProperty().toString()** will be interpolated into the message.

The message must be in the user messages file loaded by KualiConfigurationService.

Struts Messages

User messages are also available to the web layer of transactional documents and user screens through the standard Struts **bean:message** tag.

Messages to be loaded to struts are configured via the client application's project configuration file, in the **rice.struts.message.resources** property, like so:

```
<param name="rice.struts.message.resources">KR-
ApplicationResources,org.kuali.rice.kew.ApplicationResources,org.kuali.rice.ksb.messaging.ApplicationResources,KIM-
ApplicationResources</param>
```

Again, the files are listed relative to the src/main/resources directory. There is nothing to prevent programmers from using one user message file for both the KualiConfigurationService messages and the Struts messages.

Once Struts has these messages loaded, it is easy to access them in a JSP page or jsp TAG file. Indeed, a great many of the delivered Rice tags make use of these message resources in order to display information, as seen from this sample from the standard **kul:page** tag:

```
<title><bean:message key="app.title" /> :: ${headerTitle}</title>
```

In this case, the user message – set in KR-ApplicationResources.properties – with the property key of "app.title" will be displayed (which, by default as "Kuali").

Developers curious about further information about the bean:message tag would be advised to read Struts' documentation of the feature: http://struts.apache.org/1.2.x/userGuide/struts-bean.html.

KNS Questions and Dialogs

Several use cases exist where extra document processing needs to occur between the submitting of a document for routing or approval and the validation on that document. For instance, a document may be created to purchase an airplane ticket. The initial submitter is not required to enter the airline that will be traveled on. However, if the initial submitter attempts to route the document without an airline being entered, a prompt can come up to ask if the submitter really meant to not enter the airline. If the answer is yes, the document will go on to validation; if the answer is no, then the document will return to allow the user to edit.

Prompting Before Validation

This kind of prompt is easily accomplished by giving the document an **org.kuali.rice.kns.rule.PromptBeforeValidation** implementation. This is done via the data dictionary:

```
<bean id="BudgetAdjustmentDocument" parent="AccountingDocumentEntry">
    <property name="documentTypeName" value="BA"/>
    <property name="documentClass" value="org.kuali.kfs.fp.document.BudgetAdjustmentDocument"/>
```

```
    <property name="promptBeforeValidationClass"
 value="org.kuali.kfs.fp.document.validation.impl.BudgetAdjustmentDocumentPreRules"/>
        ...
</bean>
```

The **PromptBeforeValidation** interface only has one method, **processPrompts**. It is responsible for holding the current form at a current point, rendering a question, getting the answer to that question, and applying that answer to the next forward. It provides a lot of flexibility.

If, however, all of the questions to ask the user can be formulated as yes/no questions, it is more advisable to simply extend **org.kuali.rice.kns.rules.PromptBeforeValidationBase** and override the **doPrompts** method. **PromptBeforeValidationBase** provides all the functionality necessarily to easily ask a yes/no question or even a series of yes/no questions.

Analysis of an example from KFS should help clarify how this works. **org.kuali.kfs.module.ar.document.validation.impl.CustomerPreRules** will be examined. Here is how it overrides the **doPrompts** method:

```
@Override
public boolean doPrompts(Document document) {
    boolean preRulesOK = true;
    preRulesOK &= conditionallyAskQuestion(document);
    return preRulesOK;
}
```

doPrompts takes the document to act upon as a parameter and it returns a boolean variable. If true is returned, the document will plow forward into validation. If false is returned, then the view should return to another forward. Which forward used will be soon revealed.

Given this information, it's obvious that the real work is occurring in **conditionallyAskQuestion**. And indeed it is:

```
protected boolean conditionallyAskQuestion(Document document) {
    MaintenanceDocument maintenanceDocument = (MaintenanceDocument) document;
    Customer newCostomer = (Customer) maintenanceDocument.getNewMaintainableObject().getBusinessObject();
    boolean shouldAskQuestion = maintenanceDocument.isNew() && checkIfOtherCustomerSameName(newCostomer);

    if (shouldAskQuestion) {
        String questionText =
 SpringContext.getBean(KualiConfigurationService.class).getPropertyString(ArKeyConstants.CustomerConstants.MESSAGE_CUSTOMER_WITH_
        boolean confirm =
 super.askOrAnalyzeYesNoQuestion(ArKeyConstants.CustomerConstants.GENERATE_CUSTOMER_QUESTION_ID, questionText);
        if (!confirm) {
            super.abortRulesCheck();
        }
    }
    return true;
}
```

The document in this example is a maintenance document, but the method works precisely the same given a transactional document.

Logic determines, in this case, if the customer is new and if it shares the name of an existing customer. If that is the case, then it asks a yes/no question about if the user meant to enter a second customer with the same name. Note that question text is specified via a User Message; this is a best practice.

The question is asked and the yes/no answer returned through the **super.askOrAnalyzeYesNoQuestion**. That needs to be handed an ID which uniquely represents every asking of this question – that, in conjunction with information from the document itself, is used to identify the user response, which ends up in the session. The other method argument is the question text itself.

It returns true or false. Note though, that if the response was false, that false is not returned, but instead a method **super.abortRulesCheck();** is called.

abortRulesCheck() is simply a convenience method that sets the forward to return to as the **BASIC_MAPPING**:

```
public void abortRulesCheck() {
    event.setActionForwardName(RiceConstants.MAPPING_BASIC);
    isAborting = true;
}
```

If application requirements determine that a "no" answer should navigate the user to a different mapping than "basic", then **abortRulesCheck** should not be used, but instead, a false should be returned from the method, and the correct action forward name should be set on the event property inherited from **PromptBeforeValidationBase**.

There is no limit to the number of times **super.askOrAnalyzeYesNoQuestion** can be called in a single pre-rules check; several questions can be chained together.

HTML Markup

In the question framework some markup support is present for formatting the question text. This markup follows a custom syntax as opposed to HTML. Standard HTML characters will be escaped in the question text. This is to prevent cross-site scripting attacks. The custom syntax for the supported tags is then translated to the corresponding HTML when rendering the question page.

The custom syntax uses brackets to indicate tags as opposed to the standard HTML left and right angle quote characters. Like HTML, an opening and closing tag must be present: e.g. [tag] ... [/tag]. The custom syntax does not support empty body tags: e.g. [tag/].

The following is a list of the tags supported along with the corresponding HTML translation.

```
* All 1 character HTML tags
Examples:
[p] ... [/p] translates to <p> .... </p>
[b] ... [/b] translates to <b> .... </b>


* All 2 character HTML tags
Examples:

[br] ... [/br] translates to <br> .... </br>
[tr] ... [/tr] translates to <tr> .... </tr>

[td] ... [/td] translates to <td> .... </td>

* The font tag with color specified as hex or by name
Examples:
[font #000000] ... [/font] translates to <font color="#000000"> .... </font>
[font red] ... [/font] translates to <font color="red"> .... </font>

* The table tag

Example:

[table] ... [/table] translates to <table> .... </table>

* The table tag with style class
Example:
[table questionTable] ... [/table] translates to <table class="questionTable"> .... </table>


* The td tag with style class
```

```
Example:
[td leftTd] ... [/td] translates to <td class="leftTd"> .... </td>
```

Note since the style tag is not allowed any CSS classes used must be declared in the Kuali style sheet (by default kuali.css). In addition be aware that the one and two character tags are not verified as valid HTML tags. In essence, the brackets are simply replaced by the angle quotes and outputted for these tags.

When forming the question text, consideration should be given to the text length. The question text is sent as one of the request parameters on the URL which is limited by the browser supported max length. Keeping the text under 1000 characters will be safe across all supported browsers.

Derived Values Setters

What about those instances when a client application has a document that needs to set values based on user input but which do not require any further user prompts before the document is validated? This is where **org.kuali.rice.kns.web.derviedvaluesetter.DerivedValuesSetter** steps in.

DerivedValuesSetter has one method:

```
public void setDerivedValues(KualiForm form, HttpServletRequest request);
```

Nothing is returned, and the arguments are basically the web form and the servlet request itself. Values can be gathered from either of those sources, and then values can be set anywhere on the form – though it would typically be expected that the document in the **KualiForm** would be where everything is set.

Actual examples of **DerivedValuesSetter** implementations is fairly rare. There is one example from KFS 3.0 which will be used as an example, associated with the Organization Maintenance Document. First, the **DerivedValuesSetter** is set in the data dictionary for the document:

```
<bean id="OrganizationMaintenanceDocument" parent="MaintenanceDocumentEntry">
    <property name="businessObjectClass" value="org.kuali.kfs.coa.businessobject.Organization"/>
    <property name="documentTypeName" value="ORGN"/>
    <property name="promptBeforeValidationClass"
 value="org.kuali.kfs.coa.document.validation.impl.OrgPreRules"/>
    <property name="derivedValuesSetterClass" value="org.kuali.kfs.coa.document.web.OrgDerivedValuesSetter"/>
        ...
</bean>
```

The actual **DerivedValuesSetter** itself attempts to use the **PostalCodeService** to set the city and state of the organization. Here's a simplified version:

```
public class OrgDerivedValuesSetter implements DerivedValuesSetter {
    public void setDerivedValues(KualiForm form, HttpServletRequest request) {
        final Organization newOrg = (Organization) ((MaintenanceDocumentBase)((KualiMaintenanceForm)
 form).getDocument()).getNewMaintainableObject().getBusinessObject();
        final String organizationZipCode = newOrg.getOrganizationZipCode();
        final String organizationCountryCode = newOrg.getOrganizationCountryCode();
        if (StringUtils.isNotBlank(organizationZipCode) && StringUtils.isNotBlank(organizationCountryCode)) {
            final PostalCode postalZipCode =
 SpringContext.getBean(PostalCodeService.class).getByPrimaryId(organizationCountryCode, organizationZipCode);
            if (ObjectUtils.isNotNull(postalZipCode)) {
                newOrg.setOrganizationCityName(postalZipCode.getPostalCityName());
                newOrg.setOrganizationStateCode(postalZipCode.getPostalStateCode());
            }
        }
    }

}
```

Here, the new Organization business object is pulled from the maintenance document, and from that, the zip code and country code are pulled. The code attempts to use the country and zip codes to find a postal code, and if one is found, it sets the city and state of the document.

Both **PromptBeforeValidation** and **DerivedValuesSetter** classes offer KNS client application developers the flexibility to prompt the user or set values on a document before that document goes into validation.

KNS Notes and Attachments

On most documents written for Rice client applications, there exists a tab at the bottom of the page, the Notes tab. This allows document editors to attach files to the page or write explanatory notes.

How are these notes supported?

org.kuali.rice.kns.bo.PersistableBusinessObject requires methods to add and programmatically manipulate notes on the object. Therefore, all persisting business objects in client applications support the addition of notes to them. This allows for a great amount of flexibility. A note, represented by objects of class **org.kuali.rice.kns.bo.Note**, hold both text and links to attachments—as well as the note's creator and the time it was created. Therefore, such text and attachments can be associated with any persisting business object.

However, most Rice applications use Notes mostly on documents. In this case, the Note is associated with **org.kuali.rice.kns.bo.DocumentHeader** objects – the header of the document. The **kul:notes** tag and **org.kuali.rice.kns.web.struts.action.KualiDocumentActionBase** jointly provide support for adding these kinds of notes.

The use of these notes are also authorized by a number of KIM permissions. Before notes are added, the user is checked for having the **KR-NS Add Note / Attachment** permission. These permissions should always have a permission attribute associated with document name; optionally, a permission attribute for **attachmentTypeCode** can be used.

There is also the **KR-NS Delete Note / Attachment** permission. Two permission attributes are required for this: both the document name, and a record for the **createdBySelfOnly** attribute (a boolean attribute that may prevent end users from deleting notes created by other end users).

Finally, there is the **KR-NS View Note / Attachment** permission. Just as with **Add Note / Attachment** permissions, it requires a document type name and can have an optional **attachmentTypeCode**.

Note's attachments are handled by **org.kuali.rice.kns.service.AttachmentService**. By default, they attempt to move attachments into a directory specified by the **attachments.directory** configuration property; under that, each object gets its own subdirectory, with the name of the subdirectory based on the objectId of the business object.

KNS Javascript Guide

The KNS provides a number of ways to integrate Javascript into maintenance and transactional documents. A configuration parameter allows a core set of Javascript files to be imported on all pages. External Javascript files specific to a limited set of documents can easily be imported into pages using the data dictionary. Several KNS tags also support response to Javascript events.

Setting the configuration parameter is easiest of all. In the **{project name}-config.xml** file for most client applications, there already exists a generated line which looks like this:

```
<param name="javascript.files">kr/scripts/core.js,kr/scripts/dhtml.js,kr/scripts/documents.js,kr/scripts/
my_common.js,kr/scripts/objectInfo.js</param>
```

These scripts will be pulled in on every page which uses the **kul:page** tag. Note that the file path is relative to the root path of the project. It bears mentioning, too, that the css.files property works the same way for CSS files:

```
<param name="css.files">kr/css/kuali.css</param>
```

It's not always the best idea to include Javascript pages, which the browser must parse, onto every single page. If only certain documents or even a single document needs a given Javascript file, it is easiest to simply tell the data dictionary entry to import the file. Here is an example from KFS's Account Maintenance Document (**AccountMaintenanceDocument.xml**):

```
<bean id="AccountMaintenanceDocument" parent="AccountMaintenanceDocument-parentBean"/>

<bean id="AccountMaintenanceDocument-parentBean" abstract="true" parent="MaintenanceDocumentEntry"
 p:businessObjectClass="org.kuali.kfs.coa.businessobject.Account"
 p:maintainableClass="org.kuali.kfs.coa.document.KualiAccountMaintainableImpl">

    ...
    <property name="webScriptFiles">
        <list>
             <value>../dwr/interface/SubFundGroupService.js</value>
             <value>../scripts/coa/accountDocument.js</value>
        </list>
    </property>
    ...
</bean>
```

Values are expected to be relative to the base application URL of the document. In this case of a maintenance document, the URL is /application-name/kr/maintenance.do and the javascript files are located under /application-name/scripts, hence the ".." in the directories.

Integrating Javascript with KNS tags

As will be covered in the KNS tags section, most controls in KFS documents are rendered using the **kul:htmlControlAttribute** tag. That tag has three attributes which will be passed on to the rendered HTML control: **onblur**, **onclick**, and **onchange**, which will be passed on to the rendered control. (Though there is an exception to keep in mind: radio buttons will render what was passed in the onchange attribute as **onclick**, to enhance support for a highly popular legacy browser.)

Extra buttons also support Javascript, specifically the "**onclick**" event handler. By setting the **extraButtonOnclick** property of an **org.kuali.rice.kns.web.ui.ExtraButton** object to the text that should appear in the button's **onclick** call, the developer gains the ability to react, with Javascript, to the button's click.

Incorporating AJAX

Finally, we want to make our maintenance documents as interactive as possible to facilitate efficient user experience. In this example, KFS's **AccountMaintenanceDocument** wants to instantly give an error to users if the sub fund group assigned to the account is restricted, based on other values of the account.

To accomplish this, in the data dictionary file for the **AccountMaintenanceDocument**, extra JavaScript files are imported.

```
<property name="webScriptFiles">
    <list>
        <value>../dwr/interface/SubFundGroupService.js</value>
        <value>../scripts/coa/accountDocument.js</value>
    </list>

</property>
```

The **../scripts/chart/accountDocument.js** is a JavaScript file that defines the functions **onblur_subFundGroup** and **checkRestrictedStatusCode_Callback**. **onblur_subFundGroup** uses the **SubFundGroupService**, and to that successfully, DWR needs to create a JavaScript/Java bridge for that access. That's the purpose of the inclusion of the **../dwr/interface/SubFundGroupService.js** file: it's not a real JavaScript file at all, but instead a bridge created on the fly by DWR.

Maintainable fields can then trip off the AJAX call when certain events happen:

```
<bean parent="MaintainableFieldDefinition" p:name="subFundGroupCode"
 p:required="true" p:webUILeaveFieldFunction="onblur_subFundGroup"
 p:webUILeaveFieldCallbackFunction="checkRestrictedStatusCode_Callback"/>
```

In the above example, when the user leaves the UI field for the sub-fund group code, the **onblur_subFundGroup** JavaScript function will be called, and that should populate the name of the sub-fund group in the page under the UI field.

KNS Data Masking

It is very common for business objects to have fields which are not viewable to all users. The KNS provides very easy ways to mask fields throughout client applications.

Naturally, since certain end users can see the field unmasked, certain other users can see the field partially masked, and a final group of users views a fully masked field, KNS data masking it integrated with KIM permissions. Specifically, there are two KIM permissions which are consulted by KNS data masking: **KR-NS Full Unmask Field** and **KR-NS Partial Unmask Field**. Both of these permissions have two related permission attribute records: one for the field name, and one for the business object component name. That masking will automatically be applied to every use of the business object's field: on inquiries, lookups, maintenance documents, transactional documents, and screens.

```
<bean id="IdentityManagementPersonDocument-taxId-parentBean" parent="AttributeDefinition" abstract="true"
 p:name="taxId" p:forceUppercase="true" p:label="Tax Identification Number" p:shortLabel="Tax Id"
 p:maxLength="100" p:required="false">
    <property name="control">
        <bean parent="TextControlDefinition" p:size="20"/>
    </property>
    <property name="attributeSecurity">
        <bean parent="AttributeSecurity">
             <property name="mask" value="true"/>
             <property name="maskFormatter">
                 <bean parent="MaskFormatterLiteral" p:literal="*********"/>
             </property>
        </bean>
    </property>
</bean>
```

Having a KIM permission set up is not enough, however. Client application developers also have to associate masking with the field of the business object in the business object's data dictionary. That is accomplished by specifying the an attribute security object with the attribute. **IdentityManagementPersonDocument**'s taxId attribute has an example of an attribute security declaration:

The taxId field has a **TextControlDefinition** for the control, and that is followed by the attribute security declaration.

The attribute security declaration has a parent of the "**AttributeSecurity**" bean. There are several boolean properties available within the AttributeSecurity bean, but the mask and partialMask properties are the most interesting. This declaration quite simply turns masking on – if the **AttributeSecurity** is left null or if masking or partialMask are false, then no masking will be applied to the attribute.

Also specified in the example is the **maskFormatter**. There is also a partialMaskFormatter which can be set. A bean of any class which implements **org.kuali.rice.kns.datadictionary.mask.MaskFormatter** can be used for this declaration. The KNS also provides two default implementations: **org.kuali.rice.kns.datadictionary.mask.MaskFormatterLiteral**, which simply replaces a value which should be masked by a literal String (in the example above, "*********"), and **org.kuali.rice.kns.datadictionary.mask.MaskFormatterSubString**, which replaces all but a substring of the masked value as a String (this would be useful in partial mask situations).

The final piece of the puzzle is to get the KNS to consult the KIM permission and the business object's data dictionary when deciding whether or not to mask the field. Of course, the KNS renders maintenance documents, inquiry pages, and lookups automatically – it is expected that masking will be consulted in those situations.

This leaves only the issue of transactional documents and screens, where a client application developer has to build JSP manually. The KNS provides a number of helper functions to do permission checks.

Table 5.4. KNS Helper Functions for Permission Checks

JSP Function	Call Example	Description
canFullyUnmaskField	**${kfunc:canFullyUnmaskField (businessObjectClassName, fieldName, kualiForm)}**	Checks KIM permissions to determine if the field can be fully unmasked by the current end user.
canPartiallyUnmaskField **${kfunc:canPartiallyUnmaskField (businessObjectClassName, fieldName, kualiForm)}**	Checks KIM permissions to determine if the field can be partially unmasked by the current end user.	
getFullyMaskedValue **${kfunc:getFullyMaskedValue (className, fieldName, kualiForm, propertyName)}**	Uses the AttributeSecurity declaration to determine the fully masked value.	
getPartiallyMaskedValue	**${kfunc:getPartiallyMaskedValue (className, fieldName, kualiForm, propertyName)}**	Uses the AttributeSecurity declaration to determine the partially masked value.

Of course, calling these functions – especially those which do KIM permission checks – can be computationally expensive. It is always better to check if masking has been turned on by checking the data dictionary attribute for the field first, like so:

```
<c:if test="${!empty attributeEntry.attributeSecurityMask && attributeEntry.attributeSecurityMask == true  }">
    <c:set var="displayMask" value="${kfunc:canFullyUnmaskField(className, fieldName,KualiForm)? 'false' :
 'true'}" />
</c:if>
```

Alternatively, application developers can simply use the **kul:htmlControlAttribute** tag – as is the recommended practice under any circumstance – to draw the field. **kul:htmlControlAttribute** already utilizes the functions described above to make sure the field is properly masked, and as such represents the easiest way to apply masking to fields in transactional documents and screens.

Further information about KIM permissions will be covered in KNS Authorizations. The **kul:htmlControlAttribute** tag will be covered in the section on Tag Libraries.

KNS Authorization

In most client applications, there's going to be a need to guard certain end users from certain functionality. Certain documents may be locked down and only accessible to a small group of users. A tab on a certain document may only be visible based on if a System Parameter is turned on. KNS provides a standard way to turn on and off functionality based on conditions like these.

There are two sides to this functionality. One side is that these authorizations are integrated with KIM. KNS provides a number of contexts where KIM permissions are called and checked, to see if the current user is permitted to perform the action. Examples of such actions are looking up business objects, initiating documents, adding notes to a document, using a screen, or viewing a field on an inquiry or a maintenance document.

The other side is business logic associated with such authorizations. For instance, KIM permissions may be set up to allow any user of the client application to initiate a given document. However, there may be a business requirement that the document can only be initiated in the month of June. Since KIM permissions cannot capture that kind of logic, KNS provides point where programmers can create such logic.

When building KNS documents, there are two classes associated with the document which make these authorizations: the Document Presentation Controller and the Document Authorizer.

The Document Presentation Controller is where business logic authorizations are handled. These classes must implement the **org.kuali.rice.kns.document.authorization.DocumentPresentationController**. There are also interfaces for MaintenanceDocumentPresentationController and TransactionalDocumentPresentationController, each tailored to their respective document families.

The Document Authorizer is the class that does the KIM permission checks. Once again, there is an interface, **org.kuali.rice.kns.document.authorization.DocumentAuthorizer**, which all document authorizers must implement, and it also has two sub-interfaces, MaintenanceDocumentAuthorizer and TransactionalDocumentAuthorizer.

In cases where an authorization is checked by both presentation controller and authorizer, the presentation controller is called first, and then it's result is somehow sent to the authorizer. For instance, DocumentPresentationController has a method, **getActions()**, which returns a Set of Strings, each representing a standard document action (for instance, the Route document action). That Set is then sent as an argument to the DocumentAuthorizer; the DocumentAuthorizer only performs KIM checks for the actions that have been handed to it.

The classes for both the document authorizer and presentation controller are set in the document in the data dictionary. Here's an example, from the sample travel application:

```
<bean id="TravelRequest" parent="TravelRequest-parentBean"/>

<bean id="TravelRequest-parentBean" abstract="true" parent="TransactionalDocumentEntry">
    <property name="documentTypeName" value="TravelRequest"/>
    <property name="documentClass" value="edu.sampleu.travel.document.TravelDocument2"/>
    <property name="documentAuthorizerClass"
 value="edu.sampleu.travel.document.authorizer.TravelDocumentAuthorizer"/>
    <property name="documentPresentationControllerClass"
 value="edu.sampleu.travel.document.authorizer.TravelDocumentPresentationController"/>
        ...
</bean>
```

The classes for the authorizer is given to the documentAuthorizerClass property of the main document bean, and the presentation controller class is specified in the documentPresentationControllerClass property. This is the same for maintenance documents as well. Once these are specified, the proper classes will be constructed at authorization invocation contexts automatically.

Common Document Authorizations

There are two authorizations which are common to all documents. In both cases, the document presentation controller is called and then the authorizer if needed.

The first common authorization is the document initialization authorization. DocumentPresentationController has this method to be overridden for business logic about when a document can be initialized:

```
public boolean canInitiate(String documentTypeName);
```

The DocumentAuthorizer also has a:

```
public boolean canInitiate(String documentTypeName, Person user);
```

The DocumentAuthorizer checks the KR-SYS Initiate Document permission.

The second common authorization is handled by DocumentPresentationController#getDocumentActions:

```
public Set<String> getDocumentActions(Document document, Person user, Set<String> documentActions);
```

It passes its result to DocumentAuthorizer#getDocumentActions:

This authorization actually handles many common authorizations which need to be passed to the document presentation layer. The Set returned by the DocumentAuthorizer is converted into a Map, where each element in the Set becomes a key of the Map. That Map can then be accessed in any web page or tag through the **KualiForm.documentActions** variable.

org.kuali.rice.kns.document.authorization.DocumentPresentationControllerBase defines a number of protected methods which are inquired when the Set returned by **getDocumentActions** is built. Builders of client applications are far more likely to override one of those helper methods than override **getDocumentActions** from scratch.

Table 5.5. Document Presentation Controller Methods

DocumentPresentationControllerBase method	Purpose	Related Authorizer Permission
protected boolean canEdit(Document document)	Determines if the document can be edited; if false is returned, then all fields are in a read only state	KR-NS Edit Document
protected boolean canAnnotate(Document document)	Determines if any ad hoc requests can be added to the document.	
protected boolean canReload(Document document)	Determines if the document can be reloaded from the database.	
protected boolean canClose(Document document)	Determines if the document can be closed, returning the end user to the portal.	
protected boolean canSave(Document document)	Determines if the document can be saved.	KR-WKFLW Save Document
protected boolean canRoute(Document document)	Determines if the document can be routed to workflow.	KR-WKFLW Route Document
protected boolean canCancel(Document document)	Determines if the document can be canceled.	KR-WKFLW Cancel Document
protected boolean canCopy(Document document)	Determines if the document can be used as the template for a new document.	KR-NS Copy Document

DocumentPresentationControllerBase method	Purpose	Related Authorizer Permission
protected boolean canPerformRouteReport(Document document)	Determines if the future requests workflow report can be viewed.	
protected boolean canAddAdhocRequests(Document document)	Determines if the document can have ad hoc routing requests added to it.	KR-NS Send Ad Hoc Request
protected boolean canBlanketApprove(Document document)	Determines if the document can be blanket approved.	KR-WKFLW Blanket Approve Document
protected boolean canApprove(Document document)	Determines if the document can be approved.	KR-NS Take Requested Action
protected boolean canDisapprove(Document document)	Determines if the document can be disapproved.	KR-NS Take Requested Action
protected boolean canSendAdhocRequests(Document document)	Determines whether the document will be allowed to send itself to KEW to fulfill ad hoc requests.	KR-NS Send Ad Hoc Request
protected boolean canSendNoteFyi(Document document)	Sends an FYI to previous approvers if a note is added.	KR-NS Send Ad Hoc Request
protected boolean canEditDocumentOverview(Document document)	Determines if the fields in the document overview (title, etc) can be edited.	KR-NS Edit Document
protected boolean canFyi(Document document)	Determines if the document can be FYI'd.	KR-NS Take Requested Action
protected boolean canAcknowledge(Document document)	Determines if the document can be acknowledged.	KR-NS Take Requested Action

DocumentAuthorizer also contains a number of methods which are not subject to document presentation controller input. These are:

Table 5.6. Document Authorizer Methods

Document Authorizer method	Description	KIM Permission Checked
public boolean canOpen(Document document, Person user);	Determines if the current user can open the document	KR-NS Open Document
public boolean canReceiveAdHoc(Document document, Person user, String actionRequestCode);	Determines if the person, for whom there is a proposal to add an ad hoc routing request, can receive that ad hoc routing request	KR-WKFLW Ad Hoc Review Document
public boolean canAddNoteAttachment(Document document, String attachmentTypeCode, Person user);	Determines if the current user can add a note to the document.	KR-NS Add Note / Attachment
public boolean canDeleteNoteAttachment(Document document, String attachmentTypeCode, String createdBySelfOnly, Person user);	Determines if the current user can delete a note on a document.	KR-NS Delete Note / Attachment
public boolean canViewNoteAttachment(Document document, String attachmentTypeCode, Person user);	Determines if the current user can view a note on the document.	KR-NS View Note / Attachment

Maintenance Document Authorizations

A couple of authorizations are specific to maintenance documents. The document presentation controller and document authorizer diverge somewhat on which methods they support to control authorizations in documents, so each will be treated separately, save for the one method they do share.

That one method is **canCreate**. Here is MaintenanceDocumentPresentationController's declaration of the method:

```
public boolean canCreate(Class boClass);
```

It takes the class of the business object, of which the end user is attempting to create a new record of through a maintenance document. Business logic can be written to determine if new records of the given class can be created through a maintenance document.

MaintenanceDocumentAuthorizerBase checks the KR-NS Create / Maintain Record(s) to see if a new business object of the type can be created for its corresponding check.

MaintenanceDocumentPresentationController has two methods which do not have parallels in the MaintenanceDocumentAuthorizer. They are:

```
public Set<String> getConditionallyReadOnlyPropertyNames(MaintenanceDocument document);

public Set<String> getConditionallyReadOnlySectionIds(MaintenanceDocument document);
```

To understand this, recall that if there is a field on a maintenance document which is unconditionally read only, that is set within the data dictionary file for that maintenance document. Of course, it brings up the question of what to do if a field or a section is read only some times based on certain business logic and editable at others.

The answer to this has been provided by the MaintenanceDocumentPresentationController. A Set of the property names of the fields or names of the sections which are, given the current condition of the MaintenanceDocument argument, currently read only is returned from the method.

MaintenanceDocumentAuthorizer's separate methods are similar to **canCreate**. The first is **canMaintain**, which determines if the current user can edit an already existing business object. There is also **canCreateOrMaintain**, which combines the KIM permission checks when the document is routed to make sure the routing is valid.

Finally, MaintenanceDocumentAuthorizer has a method:

```
public Set<String> getSecurePotentiallyReadOnlySectionIds();
```

Unlike most methods in MaintenanceDocumentAuthorizer, this method was actually specified to be overridden. It returns a Set of the names of sections on a maintenance document which may be read only based on the user.

Maintenance Document/Inquiry Authorizations

Because maintenance documents and inquiries are rendered using the same code, authorizations which control that rendering are shared between the two. There are two such permissions: KR-NS View Inquiry or Maintenance Document Field and KR-NS View Inquiry or Maintenance Document Section. Since maintenance documents allow editing in addition to viewing, there are two other permissions which control the ability of end users to edit: KR-NS Modify Maintenance Document Field and KR-NS Modify Maintenance Document Section.

These are used only as KIM permissions, and they are invoked directly within the rendering framework. Their purpose is as follows:

• KR-NS View Inquiry or Maintenance Document Section will only render a whole tab section to those with the permission.

  KR-NS Modify Maintenance Document Section will only allow edits for a whole tab section to those with the permission; otherwise, the fields within the section will be rendered as read only.

- KR-NS View Inquiry or Maintenance Document Field will only render a field to entities granted this permission.

- KR-NS Modify Maintenance Document Field will only allow edits of a field to entities granted this permission; the field will otherwise be rendered as read only.

If no KIM permission is specified for a given section or field, it is assumed that it is viewable on both the Inquiry and Maintenance Document and the field will be editable on the Maintenance Document.

There are no document presentation controller methods to override if the ability to view or edit parts of an inquiry or maintenance document based on business logic. If a client application has such a requirement, adventurous technical personnel are invited to look at **Maintainable#getRows** and **Inquirable#getRows**. The subject is otherwise outside the scope of this document.

Transactional Document Authorizations

There is only one major authorization which is added to TransactionalDocumentPresentationController and TransactionalDocumentAuthorizer: **getEditModes**. Much like **DocumentPresentationController#getDocumentActions()**, **TransactionalDocumentPresentationController#getEditModes()** takes as an argument the document the authorization is being asked of and returns a Set of Strings.

Unlike **DocumentPresentationController#getDocumentActions()**, though, **TransactionalDocumentPresentationController#getEditModes()** does not have a set of standard actions it returns. Instead, it is designed specifically to allow any kind of action through the web presentation layer. There, the edit modes can be checked and acted upon in document specific ways.

How is this helpful? In maintenance documents, since the KNS handles the rendering in a standard way, it is easy to turn sections on and off; KIM permissions or work through the maintainable can accomplish. In transactional documents in the other hand, rendering is more manual. However, **getEditModes** provides a way for the business logic layer to communicate information to the presentation layer.

To get the presentation layer to not display a section, then, a presentation controller might be written as so:

```
public ExampleDocumentPresentationController extends TransactionalDocumentPresentationController {
    public Set<String> getEditModes(Document document) {
        final ExampleDocument exampleDocument = (ExampleDocument)document;
        Set<String> editModes = new HashSet<String>();
        if (exampleDocument.dontShowExtraSection()) {
            editModes.add("NO_EXTRA_SECTION");
        }
    return editModes;
    }

}
```

Then, in Example.jsp, we may have code that looks like this:

```
<c:if test="${!KualiForm.editingMode['NO_EXTRA_SECTION']}">
    <kul:tab tabTitle="Extra Section" defaultOpen="true" tabErrorKey="${Constants.EXTRA_SECTION_ERRORS}">
        ...
    </kul:tab>

</c:if>
```

Edit Modes also go through the document authorizer, meaning that there is a permission associated with them: KR-NS Use Transactional Document. Expected permission details are the document type and the name of edit mode (in this example, "NO_EXTRA_SECTION").

Other Authorizations

Finally, there are two permissions which do not affect documents but only business objects. They are:

- KR-NS Look Up Records, which determines if records of the given type can be looked up by the current user. Client applications seeking to change this based on business logic would likely override the business object's implementation of **org.kuali.rice.kns.lookup.LookupableHelperService#getRows()**.

- KR-NS Inquire Into Records, which determines if the current user can inquire into records of the given business object. Client applications seeking to change this based on business logic would likely override the business object's implementation of **org.kuali.rice.kns.lookup.LookupableHelperService#getInquiryUrl()** or its implementation of **org.kuali.rice.kns.inquiry.Inquirable#getInquiryUrl(BusinessObject businessObject, String attributeName, boolean forceInquiry)**, depending on the use case.

Overriding Document Authorizers

Document authorizers handle their calls to Kuali Identity Management in standard ways already. Because this side of authorizations mostly relies on KIM configuration, there is very little reason to override Document Authorizers. In fact, such overrides only occur to accommodate one of the two following situations.

The first situation is when a client application-specific KIM permission which affects documents is invoked. In this case, it is a best practice to give developers the ability to change this logic through the document presentation controller, and then do the actual KIM permission call in the document authorizer. Document authorizers were designed to be standard permission invocation contexts, and using them as such makes development much easier.

The second situation is to add extra attributes to permission detail attribute sets, role qualifier attribute sets, or to both. These extra attributes are sent on every KIM permission call performed by the authorizer. The reason for doing this is to make sure that permissions and roles can qualify properly when the document authorizer performs its call.

For example, imagine a role where the users are qualified by a client application specific field. The document authorizer does not know where or how to gather the data for that field, and yet it must be sent to KIM for the role members to be resolved correctly. Therefore, the

```
protected void addRoleQualification(BusinessObject businessObject, Map<String, String> attributes)
```

method should be overridden, and the attributes argument should be filled with values from the businessObject (which may well be a document) to make sure the role is resolved correctly.

The same can be done for permission details:

```
protected void addPermissionDetails(BusinessObject businessObject, Map<String, String> attributes)
```

Finally, if a certain attribute is used both in finding the permission via the permission details and resolving the role, then the following method should be overridden; it will add the attribute to both:

```
protected void addStandardAttributes(Document document, Map<String, String> attributes)
```

KNS Exception Handling and Incident Reporting

Any complex Java system are subject to the occurrences of exceptions. From missed assignments which cause NullPointerExceptions to network issues which cause SQLExceptions be thrown, the unexpected happens—even in Rice applications.

Because of this, Rice builds on top of Struts' exception mechanism to provide an easy way for exceptions to be handled and for incidents to be reported to the proper maintenance group.

When a developer creates a Rice application, there should be several struts-config.xml files created. The developer's own struts-config.xml, of course, exists in {project_root}/src/main/webapp/WEB-INF. It will automatically be created with the following entry:

```
<global-exceptions>
    <exception type="java.lang.Throwable"
 handler="org.kuali.rice.kns.web.struts.pojo.StrutsExceptionIncidentHandler"
 key="meaningless" />
</global-exceptions>
```

This tells Struts that if any exceptions—or even Errors for that matter!—reach the Struts request processor, then it is to redirect the application to the **org.kuali.rice.kns.web.struts.pojo.StrutsExceptionIncidentHandler**. This handler, in turn, redirects to the following forward:

```
<action path="/kualiExceptionIncidentReport"
 type="org.kuali.rice.kns.web.struts.action.KualiExceptionHandlerAction">
    <forward name="basic" path="/kr/kualiExceptionIncidentReport.do" />
</action>
```

This forward does a number of things. First, it sends the exception to **org.kuali.rice.kns.service.KualiExceptionIncidentService#getExceptionIncident** to wrap the exception, and then reports the wrapped exception to **org.kuali.rice.kns.service.KualiExceptionIncidentService#report**.

In the default implementation, **org.kuali.rice.kns.service.KualiExceptionIncidentService#report** emails the mailing list specified in the **KualiExceptionIncidentServiceImpl.REPORT_MAIL_LIST** configuration parameter. The rest of the mail can be configured by overriding the service bean's message template:

```
<bean id="knsExceptionIncidentService"
 class="org.kuali.rice.kns.service.impl.KualiExceptionIncidentServiceImpl">
    <property name="mailService"><ref bean="mailService"/></property>
    <property name="messageTemplate">
        <bean class="org.kuali.rice.kns.mail.MailMessage">
            <!-- The property place holder below must be specified in common-config-default.xml or any other
 KNS configuration file -->
            <property name="fromAddress">
                <value>${kr.incident.mailing.list}</value>
            </property>
        </bean>
    </property>
</bean>
```

Then the action redirects the user to the error page. In production environments, this page simply notes that an error occurred and that it had been reported to the system's administrators. Helpfully, it also provides

a text box so the user can describe the steps leading up to the incident. In development environments, this page also displays the top stack trace of the exception which occurred.

With this reporting mechanism, incidents are properly reported and can be responded to and fixed.

KNS System Parameters

Often times, there are changes in functionality in a client application which functional users want to have control over without an undue technical burden. For instance, a certain set of documents may be associated with a bank; information about a bank is shown on the screen of each of the documents. If more documents are among those to show bank information, functional users would love it if they could just create one maintenance document and that change took effect. By coding with system parameters, such functionality is achievable within the KNS.

A System Parameter is simply a business object which holds text. That text will be used in one of three standard ways: simply as text itself, as an indicator of whether certain logic should be performed or not; or to see if a value from logic falls within a certain set of values. The advantage of using System Parameters is that they are easily changed since a maintenance document already exists as part of the KNS for them.

Parameters are used either for configuration, as described above, or for validation – for instance, if a field on a document can only have one of a certain number of values, and those values need to be changed by a functional user, then a System Parameter would be helpful.

It should be noted that the maintenance of System Parameters is only authorized to those granted the KR-NS Maintain System Parameter KIM permission.

Getting text from a system parameter

The data from a system parameter can be retrieved through the **ParameterService#getParameterValue** method, using the parameter's name to identify the parameter. The parameter's name has three components: a namespace, a parameter detail type code, and a name field.

The namespace matches a KNS module's namespace code, typically the namespace code of the module which invokes the parameter. For instance, parameters called within the KNS itself use the base namespace code of "KR-NS"

The name of the parameter should be unique within certain constraints: it must be unique with the namespace, the parameter detail type code, and, as will be covered below, the application namespace. This means that, for instance, if a client application is written with two modules, both modules could create a system parameter with the same name because they would have different namespace codes. Indeed, system parameters within the same module can be named the same thing if they have differing parameter detail type codes.

The parameter detail type code is the most difficult to understand. To understand why, the method signature of **ParameterService#getParameterValue** must be investigated.

```
public String getParameterValue(Class<? extends Object> componentClass, String parameterName);
```

Instead of a String namespaceCode and a String parameterDetailTypeCode, a Class is sent in. That class typically represents the class which will make use of this specific system parameter. From that class is determined both the namespace code and the parameter detail type code.

Finding the namespace code is typically done by looking at the package prefixes in the module configuration. If a class needs to be in a different namespace, it can have the

@org.kuali.rice.kns.service.ParameterConstants.NAMESPACE annotation can be used to specify something different.

There is also an **@org.kuali.rice.kns.service.ParameterConstants.COMPONENT** annotation which can be used to specify a specific parameter detail type code. If that is missing, though, then an algorithm inspects the class to see what parameter detail type code is most appropriate:

- If the class represents a transactional class, then the parameter detail type code will the sample name of the class with the trailing expected "Document" removed. For instance, **org.kuali.kfs.fp.document.DisbursementVoucherDocument** has a parameter detail type code of "DisbursementVoucher"

- If the class represents a business object, then the parameter detail type code will be the simple class name. Business object class "org.kuali.kfs.fp.businessobject.PayeeDetail" will have a parameter detail type code of "PayeeDetail"

- Any other class will use the simple class name. This particular behavior will eventually be deprecated.

Based on these standards, it should be easy to tell what the parameter detail type code for a given parameter should be.

The parameter's value is then a simple lookup using the class making the call to ParameterService and the name of the parameter:

```
final String parameterValue = KNSLocator.getParameterService().getParameterValue(this.getClass(),
 "SIMPLE_VALUE");
```

The parameterValue can then be used for whatever purpose the business logic requires.

Using an indicator parameter

An indicator parameter's text is either "Y" or "N"; invoking that parameter as an indicator parameter simply means that the text will be translated to its corresponding boolean value. It is accessed through **ParameterService#getIndicatorParameter**, which works much as **ParameterService#getParameterValue** does:

```
if (KNSLocator.getParameterService().getParameterIndicator(this.getClass(), "EXECUTE_LOGIC_IND")) {
    // do something...
}
```

Parameter Evaluators

Using parameter text is fine if there is only one value in the text. However, very often a parameter may need to be associated with several pieces of text.

For instance, the first example of the System Parameters section talked about having bank information applied to a collection of documents. It seems inefficient to create a bunch of indicator parameters for this. It would be better to create one parameter with a number of document types in the txt.

This is easily done. The standard way is to list the document types in the text, separated by semi-colons as so: **FirstDocumentType;SecondDocumentType;ThirdDocumentType**

While that could be retrieved via the ParameterService#getParameterValue method and then split, there's a much better way to examine the value: through the use of a parameter evaluator.

ParameterEvaluators are simply objects which take values from the environment and see if they made the constraints of the parameter. It will do the parsing of the parameter itself and then attempt to match that against an input value:

```
KNSServiceLocator.getParameterService().getParameterEvaluator(ParameterConstants.NERVOUS_SYSTEM_DOCUMENT,
  "BANK_DOCUMENT_TYPES", document.getDocumentType()).evaluationSucceeds();
```

This looks at the KR-NS / Document / BANK_DOCUMENT_TYPES parameter, splits its semi-colon valued, and then matches **document.getDocumentType()** against each of the values returned from the split.

The constraint code of the system parameter, mentioned earlier, is invoked at this point. **evaluationSucceeds()** will return true if document.getDocumentType() is within the values in the parameter and the parameter constraint code is "A" ("allow"). If, on the other hand, the constraint code is "D" ("deny") and the document type is matched in the parameter's values, a false will be returned – the document type sent in is denied by the parameter.

(Parameter accessed through **getParameterValue()** and **getIndicatorParameter()** should simply set their constraint code to "A").

System parameters used for validation can add errors if the evaluation fails through the parameter value:

```
KNSServiceLocator.getParameterService().getParameterEvaluator(ParameterConstants.NERVOUS_SYSTEM_DOCUMENT,
  "VALID_DOCUMENT_TYPES", document.getDocumentType()).evaluateAndAddError(document.getClass(),
  "errorPropertyName", "error.invalid.document.type.message");
```

In this example, if the value of **document.getDocumentType()** does not match the values in the parameter, an error will automatically be added to **errorPropertyName** on the document, and the user message with the key of **error.invalid.document.type.message** will be shown. Once against, the system parameter's constraint code is used to determine if the value succeeds or not.

The parameter evaluator can handle more complex situations as well. Take an example where a validation needs to check that, if a business object has a certain "dispersementCode", then a child business object has a specific "reimbursementCode". In this case, the system parameter's value might look like this: **A=Z**

This means that if the dispursementCode of the parent is A, then the reimbursementCode of the parent must be Z. This parameter can be used with the semi-colon to form a list: **A=Z;B=X;C=K**

The ParameterEvaluator call is again straightforward:

```
if (KNSServiceLocator.getParameterService().getParameterEvaluator(this.getClass(), "PARENT_CHILD_MATCH",
 parent.getDisbursementCode(), child.getReimbursementCode().evaluationSuceeds()) {
    // do something...
}
```

Here, getParameterEvalutor is given the parameter class, the name of the parameter, the code of the parent and then the code of the child, but works as ParameterEvaluator worked before.

What if the parent's disbursementCode allowed two different reimbursementCode's? Then the parameter's text would look like this: **A=Z,Y;B=X,Y;C=K,J,L**

Commas separate the child's distinct values. The invocation of the parameter evaluator is precisely the same as the call above:

ParameterService#getParameterValues() can return a parsed version of a multiple value parameter, and there is a version of **ParameterService#getParameterValue()** which takes in a constrained value for parameters in the form of "A=B"; if given the value "A", it will simply return "B".

Calling missing System Parameters

All of the methods which use a parameter's value – **ParameterService#getParameterValue**, **ParameterService#getIndicatorParameter**, and **ParameterService#getParameterEvaluator** – will throw an exception if the system parameter with the specified name cannot be found. If there is an expectation in the code that a parameter may not be found in the database, then it is advisable to call **ParameterService#parameterExists** method first. If the method returns true, then it is safe to use any of the methods above to utilize the parameter's value.

This is often useful in cases where there is a parameter that is different from document to document, but for which there exists a default fallback case. It would work like this:

```
final ParameterService parameterService = KNSLocator.getParameterService();
if (parameterService.parameterExists(document.getClass(), "EXAMPLE_VALUE")) {
    return parameterService.getParameterValue(document.getClass(), "EXAMPLE_VALUE");
} else {
    return parameterService.getParameterValue(ParameterConstants.NERVOUS_SYSTEM_DOCUMENT, "EXAMPLE_VALUE");
}
```

In this example, **ParameterService#parameterExists** is called to see if there's a parameter named "EXAMPLE_VALUE" with the namespace and parameter detail code of "document". If that exists, then it returns the value of that parameter. If it does not exist, it uses the more general KR-NS / Document / EXAMPLE_VALUE parameter.

Overriding Rice Parameters

Rice comes with a number of system parameters which affect KIM, the KNS, and KEW. They have namespace codes "KR-IDM", "KR-NS", and "KR-WKFLW" respectively. These provide defaults for Rice behavior which occurs in sample applications.

This poses a problem. If a client application is built to be used with a standalone Rice server, then each client application would have to share the defaults set in these system parameters. To allow client applications to have the ability to set these Rice system parameters separately from other client applications in a shared Rice server, the application namespace code field was added.

For instance, Rice applications come with a system parameter KR-NS / All / DEFAULT_COUNTRY which lists the default country code used in the application. If, for some reason, a client application needed a separate DEFAULT_COUNTRY, a new system parameter would need to be created through the maintenance document. The existing system parameter and the new system parameter would differ only in their values and in their application namespace codes.

All Rice system parameters come with the default Rice application namespace code of "KUALI". If the client application's version of the KR-NS / All / DEFAULT_COUNTRY had an application namespace code matching that of the app.namespace configuration property of the client application, then that would be used before the KR-NS / All / DEFAULT_COUNTRY parameter with the "KUALI" application namespace.

Building Screens using the KNS Tag Libraries

The Kuali Nervous System handles the rendering of several pieces of standard functionality: maintenance documents, inquiry pages, and lookups. However, that leaves two pieces of functionality where writing

JSP is required: on transactional documents and on non-document screens. However, even though JSP coding is required in these cases, the KNS still provides a wealth of rendering functionality through the use of tag libraries.

This section examines several categories of the most used tags that are provided by the KNS.

Implicit Variables

The KNS provides a number of implicit variables which can be used in the context of JSP pages. These variables exist to give the web layer the ability to read variables from the other KNS layers.

For instance, the variable Constants is used to give web layer developers access to **org.kuali.rice.kns.util.KNSConstants**, as so:

```
<c:if test="${KualiForm.documentActions[Constants.KUALI_ACTION_CAN_EDIT]}">
    Howdy, end user!  You can edit this page!
</c:if>
```

Client applications often overload this variable to hold not only KNS constants but application specific constants as well. There is also a **RiceConstants** variable which holds the constants in **org.kuali.rice.core.api.util.RiceConstants**, a **KewApiConstants** which holds all of the constants in **org.kuali.rice.kew.util.KewApiConstants**, and a **PropertyConstants** which holds the values in **org.kuali.rice.kns.util.KNSPropertyConstants**.

All configuration properties are loaded into a variable **ConfigProperties**

```
<p>
application namespace is <c:out value="${ConfigProperties['config.namespace']}" />
</p>
```

The entirety of the data dictionary as also exposed in the map constant **DataDictionary**. The keys for this map are either the simple class name of a business object:

```
<c:set var="countryBODataDictionaryEntry" value="${DataDictionary['CountryImpl']}" />
```

Or, for documents, the KEW document type name:

```
<c:set var="identityManagementPersonDocumentEntry"
 value="${DataDictionary['IdentityManagementPersonDocument']}" />
```

Data dictionary values can then be accessed via JSP EL.

The final implicit variable to mention is **KualiForm**. This is the Struts form for the current page. For JSP pages supporting transactional documents, values from the document can be read through **KualiForm**. As such, this implicit variable is practically the most used.

These implicit variables work together to support the various tags the KNS provides.

Tags for Layout

KNS applications have a standard look, and client application developers will want to preserve that look. KNS layout tags provide an easy way to use the KNS look and feel.

First of all, JSP pages using tag libraries need to have **@taglib** directives added to the page:

The KNS tag library is typically imported with the kul: prefix:

Thankfully, a collection of common JSTL, Struts, and Rice tags are readily imported using a single import at the top of any custom developed JSP page:

Having done that, the developer can use the **kul:page** tag to draw the main outline of the page, such as in this example, from KFS's Format Disbursements page (**formatselection.jsp**):

The **kul:page** simply draws the frame around the page. It has two required attributes: the **docTitle**, which is the title it will use for the page in the gray bar which runs along the top, and the **transactionalDocument** attribute, which should only be true if the JSP page is supporting a transactional document.

This example uses a number of other attributes as well. The **headerTitle** is what will show in the browser's title bar. **showDocumentInfo** will treat the page as a document page and will attempt to, for instance, show a link for document type. The **errorKey** is the key for errors which should be associated with the top level of the page (in this case, it likely should have been neglected). Finally, the **htmlFormAction** is the url to the action that the form within the page – every page is assumed to have HTML form data, so an HTML form variable is constructed for it – should post to.

There's also a convenience tag that encapsulates the **kul:page** with all of the attributes needed for documents turned on: the aptly named **kul:documentPage**, exemplified here from the sample travel app:

The only required attribute here is **documentTypeName**; the **docTitle** will display the label from the data dictionary entry associated with this document, and the value for the **transactionalDocument** attribute will also be determined from the data dictionary entry. All other attributes will simply be passed along to the **kul:page** tag.

The **kul:documentPage** tag makes sure that not only is the document title splashed across at the top of the page with a light gray, scrubbed looking background graphic, but also shows common read-only document information: document number, KEW workflow status, initiator, and when the document was created.

The next most distinctive visual feature of KNS pages are the tabs which visually organize related information (through headers, it organizes the information for sight disabled end users as well). The KNS provides a main tag to draw these: not surprisingly, it's the **kul:tab** tag.

Here is an example of the tag, again from **travelDocument2.jsp**, which is part of the sample travel application:

There is only one required attribute for the tab: **defaultOpen**, which declares whether the tab should be initially rendered as open or closed (all tabs can be opened or closed once rendered). However, this example gives us a number of other useful attributes as well. **tabTitle** is the name that will appear in the tab; while not required, best practice suggests that developers provide one so the tab have a label even when closed. **tabErrorKey** lists the keys that will be associated with this tab; when those errors are rendered, their messages will be associated with the given tab.

Another thing to notice in the example was the inclusion of a **div** with class "**tab-container**". In practice, practically all KNS tabs have such a tab included. This leads to the natural question of why the tab is not part of the tag itself.

The **div** with a class of "**h2-container**" draws a header stripe at the top of the tab, with a black background and white text. This distinctive visual element should be used to mark off sub-sections of the tab.

There is also a **kul:subtab**. This visually provides an in-set tab, typically set off with a stripe that has a gray background and bolded black text. KIM's Identity Management Person Document, has such an example. It includes the tag **personContact.tag**, which builds a tab:

This splits the various sub-sections into distinctive visual elements.

The only required attribute for the tag is width, which specifies the width of the sub-element (**kul:subtab**'s are sometimes shorter than their surrounding tab – while they are always rendered with some padding, the amount of padding and thus the amount of visual separation can be increased as width is decreased).

Sub tabs often have titles, specified through the **subTabTitle** attribute. Whereas all tabs have hide/show buttons, they can be turned off from sub tabs through the use of the **noShowHideButton** attribute.

Finally, sub tabs are often associated with lookups, they have two attributes, **lookedUpBODisplayName** and **lookedUpCollectionName**, which allow results of lookups to be displayed in the sub tab itself.

Astute readers will have noticed an important visual point about tabs: the tab is rendered with the tab title in an offset visual element (like the tab in a file folder) and behind it is the gray background of the tab above. However, the top tab does not have a tab above it. Therefore, for that special top tab, there is a **kul:tabTop** which is identical to the **kul:tab** tag, save that it visually looks like the top tab. Also, to round off the bottom tab, there is a tag, **kul:panelFooter**, which takes no attributes, which will round off the bottom corners of the set of tabs.

It should be noted that for documents, general practice is that the top tab provides the standard set of fields that all KNS documents have: the document description, which is a required field, as well as a text area for the document explanation and an internal Org Doc #. Since this is standard, the KNS provides a tag, **kul:documentOverview**, which displays these fields and which is commonly the top tab of the document (thus obviating the need for the developer to use the **kul:tabTop** tag).

Practically all documents will share this line of code as the top tab. The **editingMode** attribute is required, but will practically always be the value of **KualiForm.editingMode**.

Armed with these visual layout tags, client application developers are ready to start filling in pages with form controls.

Tags for Controls

Certainly, controls can be hard coded in JSP files as HTML. However, the KNS provides several tags which provide standard functionality to controls – thus preserving the flexibility of declaring control information in the data dictionary as well as supporting masking, accessibility, and a number of other concerns without the application developer needing to concern with those details.

The basic tag for showing a field is kul:htmlControlAttribute. Dozens of examples can be found in even the simplest Rice client application. Here is the tag being used in travelDocument2.jsp in the Rice sample travel application:

```
<table width="100%" border="0" cellpadding="0" cellspacing="0" class="datatable">
    <tr>
        <kul:htmlAttributeHeaderCell labelFor="document.traveler" attributeEntry="${travelAttributes.traveler}"
 align="left" />
        <td><kul:htmlControlAttribute property="document.traveler"
 attributeEntry="${travelAttributes.traveler}" readOnly="${readOnly}" /></td>
    </tr>
    <tr>
        <kul:htmlAttributeHeaderCell labelFor="document.origin" attributeEntry="${travelAttributes.origin}"
 align="left" />
        <td><kul:htmlControlAttribute property="document.origin" attributeEntry="${travelAttributes.origin}"
 readOnly="${readOnly}" /></td>
    </tr>
    ...
</table>
```

This example has two controls which will appear on the form: one for **document.traveler** and one for **document.origin**. This is set via the property attribute; that attribute is required. Also required is the

attributeEntry attribute, which takes in the DataDictionary attribute entry for the attribute that is being displayed:

```
<c:set var="travelAttributes" value="${DataDictionary.TravelRequest.attributes}" />
```

There are also many optional attributes. One is seen in both examples above: **readOnly**, which determines if the field will simply have a read only version of its value displayed, or a control will be displayed. This attribute allows a lot of flexibility about when a field will be **readOnly** or not. Typically, though, **readOnly** is determined based on the whether there's an action "can edit" in the form's **documentActions** map:

```
<c:set var="readOnly" value="${!KualiForm.documentActions[Constants.KUALI_ACTION_CAN_EDIT]}" />
```

As covered earlier, masking is handled automatically if the field is read only. If the value of the property should not be displayed at all, the attribute **readOnlyBody** can be set to true and the value of the tag's body is displayed if the control attribute is rendered read only.

Among the other optional attributes are html attributes which are applied directly to the drawn control, such as **onblur**, **onclick**, and **onchange**. There is a **styleClass**, which is where a CSS class can be specified to render the value or control in.

Note that the type of control is not specified here. The data dictionary entry will be referred to, and that control definition will used to determine which control will be rendered. Select controls will use a values finder to find the values to display in the drop down. This means that controls can be changed without altering the JSP, which is a major strength.

The only exception to be aware of is that if a text control document contains a date, there is an attribute, **datePicker**, should be set to true.

Also in the example, the tag **kul:htmlAttributeHeaderCell** is used. It displays the label for the field in a <td> cell. There aren't officially any required attributes, though one of the following three would have a value set: **attributeEntry**, **attributeEntryName**, and **literalLabel**. **literalLabel** will force the header cell to simply display the given String. **attributeEntry**, on the other hand, will use a data dictionary attribute to find an appropriate label; it needs to be handed the proper label much as the **kul:htmlControlAttribute** uses. **attributeEntryName** takes the full name of a data dictionary attribute (such as "**DataDictionary.TravelRequest.attributes.origin**"). The label will come from the data dictionary, though the tag will do all of the lookup itself.

There are a number of other attributes exist which control how the html of the <td> tag will render. **width**, **rowspan**, **colspan**, **align**, and **labelFor**, as well as several others exist to customize the look of the tag.

What if a label is being rendered outside a table? For that, there is a **kul:htmlAttributeLabel** tag. It allows **attributeEntry** and **attributeEntryName** attributes which work just as they do in **kul:htmlAttributeHeaderCell**. **literalLabel** is not supported (since it is assumed that a literal label would simply be written into the JSP).

This too has a number of other attributes. Developers should consider three of these attributes. **useShortLabel** uses the short label in the data dictionary attribute instead of the regular label. **noColon** is a boolean. If it is set to true, then there will not be a colon rendered after the label. Finally, **forceRequired** means that a symbol will let end users know that the field is required.

There is also a convenience tag which belongs on every JSP page supporting a transactional tag, right after the **kul:documentPage** tag: **kul:hiddenDocumentFields**. Here is its use in travelDocument2.jsp:

```
<kul:documentPage showDocumentInfo="true" htmlFormAction="travelDocument2"
```

```
  documentTypeName="TravelRequest" renderMultipart="true" showTabButtons="true" auditCount="0">
<kul:hiddenDocumentFields />
```

This will make sure that the **docId** and **document.documetNumber** will be preserved to repopulate the form after an action occurs on the document by creating HTML hidden controls to carry the values through the POST.

There are two optional attributes, used to ask for the saving of more variables. If **includeDocumentHeaderFields** has a value of true, it will make sure that **document.documentHeader.documentNumber** is saved. Setting **includeEditMode** will preserve the edit modes determined for the document.

Finally, **kul:errors** should be mentioned. As previously seen, errors are typically associated with pages and tabs via **errorKeys**. If an error should show up not associated with a page or a tab but rather with some other visual element, then the **kul:errors** tab can display those.

There are no required attributes. If only the errors with a certain set of keys should be displayed, then the **keyMatch** attribute should be set. Otherwise, all remaining messages will be rendered. Forcing the rendering of all remaining messages can be forced by setting the **displayRemaining** attribute to true. An **errorTitle**, **warningTitle**, and **infoTitle** can also be set to separate the message sections. Defaults are provided if these attributes are not set.

Tags for KNS Functionality

Developers of transactional documents or screens will often want to hook into KNS functionality, such as inquiries and lookups. A set of tags makes this easily accomplished.

For instance, in Rice client applications, many controls have a question mark icon next to them, which allows the user to do a lookup and return the value into the control. To get one of those to display, the **kul:lookup tag** must be utilized, precisely as it is on **travelRequest2.jsp**:

```
<kul:htmlControlAttribute property="travelAccount.number" attributeEntry="${accountAttributes.number}"
 readOnly="${readOnly}" />
<kul:lookup boClassName="edu.sampleu.travel.bo.TravelAccount" fieldConversions="number:travelAccount.number" />
<kul:directInquiry boClassName="edu.sampleu.travel.bo.TravelAccount"
 inquiryParameters="travelAccount.number:number" />
```

Right after the **travelAccount.number** control is rendered, the **kul:lookup** tag will render the question mark lookup icon.

It takes the class of the business object it will perform a lookup against through the required **boClassName** attribute. The **fieldConversions** attribute is not strictly required but often used: it is a list of attributes from the result business object matched by a colon with the field that it should populate in the document upon return. **kul:lookup** also has support for a **lookupParameters** tag, which will populate the lookup with values from the document. There are a number of other optional attributes as well.

Also in this example is the kul:directInquiry tag. If the travelAccount.number field is filled in, then clicking the directInquiry tag will open up an inquiry page for the value given.

It, too, needs the class of the business object it is inquiring on through the required **boClassName** attribute. The non-required **inquiryParameter** attribute tells the tag which values to take from the document to use as keys for the inquiry page.

What if the value is read only and an inquiry should be displayed? In that case, the **kul:inquiry** tag should be used. Here is an example from the KFS **procurementCardTransactions.tag**:

```
<kul:inquiry boClassName="org.kuali.kfs.fp.businessobject.ProcurementCardTransactionDetail"
 keyValues="documentNumber=${currentTransaction.documentNumber}&financialDocumentTransactionLineNumber=
${currentTransaction.financialDocumentTransactionLineNumber}" render="true">
    <bean:write name="KualiForm" property="document.transactionEntries[${ctr}].transactionReferenceNumber" />

</kul:inquiry>
```

The **kul:inquiry** works much like the <a> tag it renders. Any text within the body of the tag is rendered as the text for the link. It, too, requires the **boClassName** attribute which specifies which business object will be rendered on.

It also requires two other attributes, **keyValues** and **render**. **render** is an odd attribute. It decides whether the inquiry link will be rendered or not. This allows some display level logic to check whether the field should actually be rendered on. If render is false, then only the text of the tag's body will be rendered.

keyValues hands in the query string to pass to the inquiry page, theoretically with the keys the inquiry page will need to find the record to display.

kul:inquiry has no optional attributes.

A variation of the **kul:lookup** tag also exists, which supports multiple value lookups, **kul:multipleValueLookup**. Here is an example from KC's awardKeywords.tag:

```
<kul:multipleValueLookup boClassName="org.kuali.kra.bo.ScienceKeyword" lookedUpCollectionName="keywords"
 anchor="${tabKey}"/>
```

Once again, **boClassName** of the business object class to be looked up is a required attribute. Also required is the **lookedUpCollectionName** attribute. Once the multiple values are returned from the lookup, the KNS will attempt to populate the named collection on the document with the values.

In this example, **anchor** is an optional attribute. It gives the link to return to an anchor to navigate to when it returns to the page. This is helpful on long pages. There is also an attribute **lookedUpBODisplayName** which will control the label for the business object being looked up.

Last, but by no means least, among these tags is the reliable kul:documentControls tag. Every JSP supporting a transactional document will include this tag, as it draws the row of controls on the very bottom of the page, thereby allowing end users to route, save, approve, cancel, and otherwise work with the document. travelRequest2.jsp uses it:

```
<kul:panelFooter />
<kul:documentControls transactionalDocument="false" />
```

Properly utilized this control appears just beneath the kul:panelFooter. The only required attribute is the **transactionalDocument** attribute, though, ironically, that attribute is never used within the tag. It therefore does not matter if false or true is entered as the value.

The other main attributes to be aware of support adding extra buttons. There are two mechanisms. In the first, by specifying the **extraButtonSource**, **extraButtonProperty**, and **extraButtonAlt** attributes, a single extra button will be rendered. For the image source, it will use **extraButtonSource**, with the alternate text specified by **extraButtonAlt**. The **extraButtonProperty** specifies the property of action to call (for instance, the property of the route button is "**methodToCall.route**").

That's fine for one extra button, but what if multiple extra buttons need to be added? The KNS supports this as well. **org.kuali.rice.kns.web.struts.form.KualiForm** has a List property named **extraButtons**.

The List is made up of **org.kuali.rice.kns.web.ui.ExtraButton** objects. Each **ExtraButton** object, in turn, has an **extraButtonProperty**, **extraButtonSource**, and **extraButtonAltText** properties which can be set. Those properties have the same effect as the **extraButtonSource**, **extraButtonProperty**, and **extraButtonAlt** attributes covered above. **ExtraButton** objects have two extra properties as well: **extraButtonParams** and **extraButtonOnclick** which provide the ability to hand extra parameters to the action and the ability for javascript to react to the button click respectively.

The form can have its **extraButtons** list populated before reaching the presentation layer. Most often, this is accomplished by simply overriding the form's **getExtraButtons()** method. Then the extra buttons are simply sent from the form into the **kul:documentControls** tag, as so:

```
<kul:documentControls transactionalDocument="false" extraButtons="${KualiForm.extraButtons}" />
```

The **kul:documentControls** tag will then render all of the extra buttons. Given its extra flexibility, this is the preferred method of adding extra buttons.

Useful Pre-Created Tabs

Finally, the KNS provides a number of tabs that happen to exist on most documents.

For instance, practically every document has the ability to add notes. If that functionality is to be turned off, it's much easier to do in the data dictionary – so frankly, every document should have a place to enter notes. Documents should also have the route log of the document, and a place where ad hoc KEW recipients can be added. The KNS makes adding all of these tabs easy:

```
<kul:notes />
<kul:adHocRecipients />
<kul:routeLog />
```

The names of the tags are self-explanatory; and as easy as that, these three standard tabs have been added to the document.

Chapter 6. KRAD

KRAD Overview

New for Rice 2.0, the Kuali Rapid Application Development (KRAD) framework eases the development of enterprise web applications by providing reusable solutions and tooling that enable developers to build in a rapid and agile fashion. KRAD is a complete framework for web developers that provides infrastructure in all the major areas of an application (client, business, and data), and integrates with other modules of the Rice middleware project.

KRAD expands the Kuali development platform and will eventually replace the Kuali Nervous System (KNS). KRAD supports the KNS document types - Lookups, Inquiries, and Maintenance pages - while it also provides more flexibility in user interface layouts, for example, beyond the "vertical" tab section and collection layouts typical of KNS-based applications. In addition, KRAD eliminates the need for a transaction document type, as maintenance documents can now handle full transactional interactions.

KRAD differs from KNS in some key ways:

• The KNS look-and-feel was targetted at administrative users, KRAD enables rich web applications targetted at a wide range of user types.

• KNS has little built-in rich user interface support whereas KRAD includes this.

• KNS is Struts 1.x based whereas KRAD is Spring-MVC based.

   KRAD uses the following:

• Spring Beans and Expression Language

• Apache Tiles as the templating engine

• Fluid Skinning System for CSS

• jQuery as the javascript library, including jQuery UI widgets

• And other plugins providing functionality, such as AJAX

Key KRAD Features

Built upon a rich JQuery library of standards and Fluid Skinning System's (FSS) set of cascading style sheets, KRAD provides a set of rich user Interface components, such as the following. Note that jQuery themes are widget-oriented, while the FSS provides support for whole pages and applications, so they are compatible with each other.

The key KRAD User Interface Framework (UIF) components include, but are not limited to, the following:

• Navigation objects: Left menu and Horizontal tabs navigation

• Layout managers: Grid, Box, Table and Stacked

• Widgets: Light-box, Disclosure, Breadcrumb, Date picker, Growl, Direct inquiry, Inquiry, QuickFinder, RichTable, Suggest, Tabs, Tree

- Controls: Checkbox, Checkbox group, File, KIM Group, Hidden, Select, TextArea, Text, User

- Containers: Group, Link group, Navigation group, Tab group, Tree group

- Fields: Input field, Field Group, Action, Ajax Action, Blank, Data, Errors, Generic, Header, iFrame, Image, Label, Link, Lookup Input, Message

- View Types: Lookup, Inquiry, Maintenance, Transactional

- General Features: Constraints (simple, valid characters, case, must occurs, dependency, custom), Watermarks, Help summary & description, Messages (constraint, instructional, required, error, informational, warning), Validation (client-side, dirty fields validation, exception handling - incident page), Remote fields, Progressive Disclosure, Audo Code-name translation (auto-completion), Dialogs (questions and prompts), Focus and anchoring handling, Tabbing order, Field queries, Information properties, Hidden properties, Default Values, Disabled, Alternate and Additional Display Properties, Read-Only fields request override, Attribute security and masking, Add/Delete line handling, Form Edit Modes, Property editors, Property replacers, Component refresh, Component Modifiers, Collection filters, Show/Hide inactive, EL Language for XML Config, Support for all JS events, Integration with KIM and KEW.

For example, see the information below on KRAD's Input field, and how this field can be grouped with others of the constructs listed above to make for a richer UI experience than what was possible in KNS.

An Input field enables user input. This means that this "grouped" field control will display an entry field for user input, and can optionally include instructions, a watermark, constraint text, a lookup widget, inquiry widget, and/or help widget, and includes a place for error messages associated with the field to appear. This could be considered the most complex of all fields, and additional information on this field can be found in the Developers' Guide.
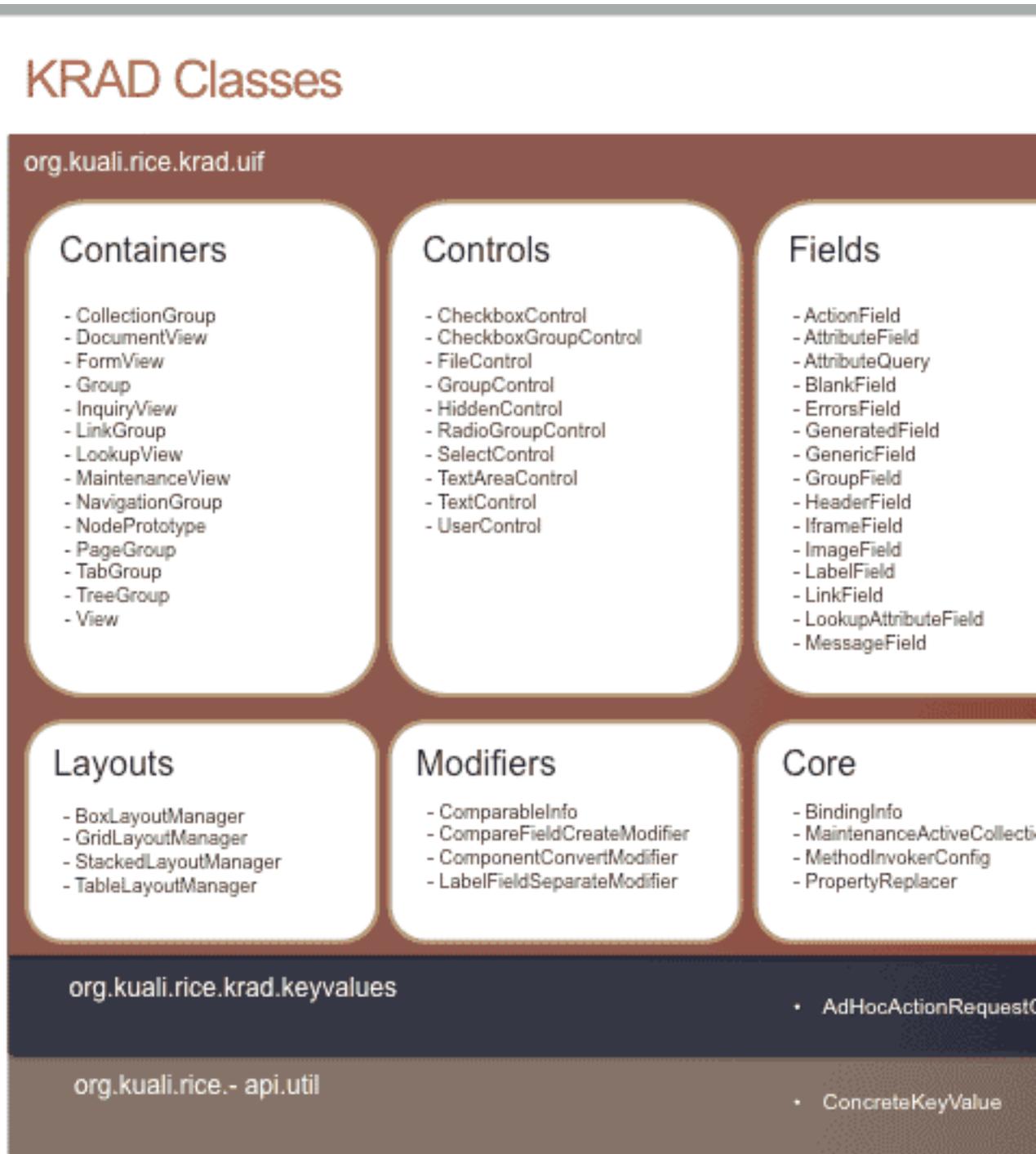
Figure 6.1. Input Field - Grouped
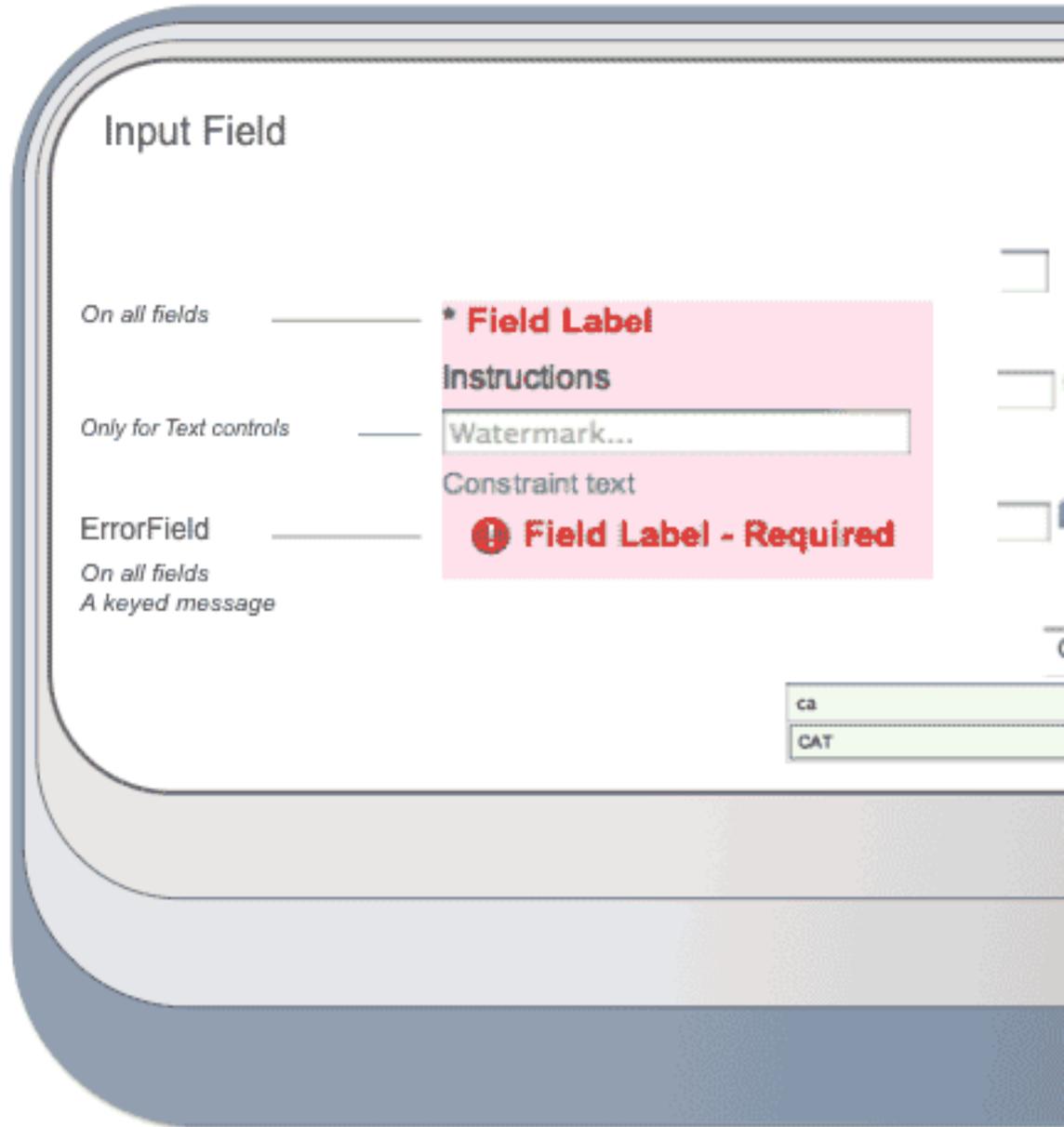
The information below provides additional conceptual and relational information on the KRAD architecture, classes and user interface patterns that are supported "out-of-the-box.
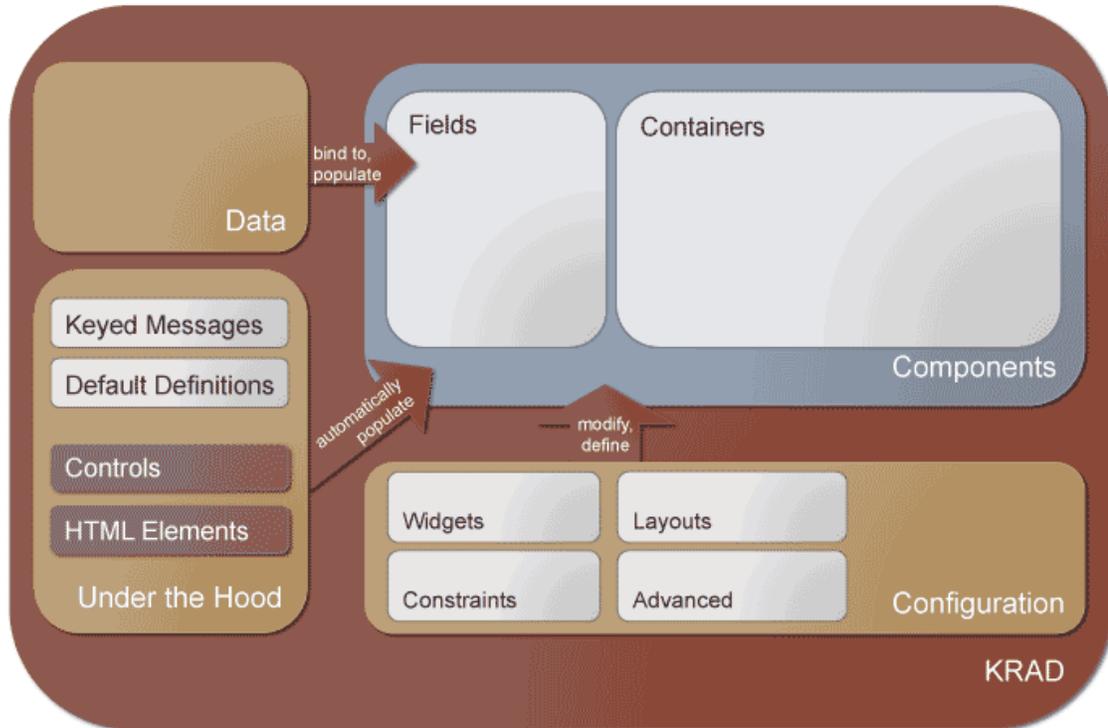
KRAD Conceptual view

Figure 6.2. KRAD Conceptual View

For additional high-level views of information on the layout managers and fields, see the KRAD Users' Guide.

KRAD Relational View

Figure 6.3. KRAD Relational View

KRAD Data Dictionary

(Need direction on what of the KNS information should be copied here and what other new information should be included.)

KRAD enhancements to the Data Dictionary include, but are not limited to, the following:

- Simple Constraints, Min/Max

- Valid Characters Constraints

- Dependency Constraints

- Lookup Constraints

- Conditional Logic Constraints

- Occurrences Constraints - Collection size constraints

- Constraints on the client side

- Changing Error Messages

- Custom Constraints

In the earliest versions of the Kuali Nervous System, it was recognized that forcing developers to write Java-based rules to check if a required field was filled in or if it matched a date pattern was a hefty load of work that easily could be transferred to the data dictionary.

Every AttributeDefinition defined for a property of a data object had the ability to be paired with a validation. For instance, let's take a generic date field from KFS's org/kuali/kfs/sys/businessobject/ datadictionary/GenericAttributes.xml file.

Code snippet example follows:

```
1.   <bean id="GenericAttributes-genericDate" parent="GenericAttributes-genericDate-parentBean"/>
2.   <bean id="GenericAttributes-genericDate-parentBean" abstract="true" parent="AttributeDefinition">
3.     <property name="name" value="genericDate"/>
4.     <property name="forceUppercase" value="false"/>
5.     <property name="label" value="Generic Date Style Attribute"/>
6.     <property name="shortLabel" value="GenericDate"/>
7.     <property name="maxLength" value="22"/>
8.     <property name="validationPattern" ref="DateValidation"/>
9.     <property name="control" ref="DateControl" />
10.    <property name="formatterClass" value="org.kuali.rice.kns.web.format.DateFormatter"/>
11.  </bean>
```

It's a simple enough example, but lines 7 and 8 pack quite a bit of power. Together, they limit the length of the field to a size which can fit in the database (evidently twenty-two characters) and they add the DateValidation, which requires that any user input fits a certain pattern defined as a regular expression. Two lines of configuration, and the developer gets a fair amount of error checking.

That's wonderful, of course, but it has limits. For example, there's no way to only run constraints based on the values present in other attributes. There wasn't a general way to enforce a data type for a user input value. There wasn't a way to say, for instance, that one or another field was required - either a field was required or it wasn't.

Such logic, not that much more complex, all required a Java-based rules solution. Much more complex logic is available than ever before. Not only that, but it can be enabled to work on the client side via JavaScript as well.

Finally, for even more flexibility, the processors which act on the constraints have been pulled out into injectable classes - meaning that applications can override the logic for a constraint if needed. Furthermore, constraints need not act only on AttributeDefinitions; new interfaces have been developed which allow any configuration class to participate in being validated. Obviously, there's a lot of functionality to cover - from the classic constraints which continue on in the framework to the powerful constraints that the Kuali Student team contributed to KRAD.

The information below includes an overview of the specific "built-in" KRAD constraints available to developers. We'll also cover the architecture of the constraint framework, with a special emphasis on how constraint logic may be overridden, how new constraints would be constructed, and how non-attributes could have Constraint logic built for them.

Information on each of the KRAD-packaged constraints is below, followed by a look at the constraint architecture itself.

Simple Constraints, Min / Max

As is covered in more detail in the Constraint Architecture section that follows this this documentation of the constraints packaged with KRAD, every constraint in KRAD implements the *org.kuali.rice.kns.datadictionary.validation.constraint.Constraint* interface. This interface is a simple marker interface. Children of that interface tend to define the data they would need from the configuration to figure out if the value put into the attribute is valid or not.

For instance, in the *GenericAttributes-genericDate* example in the introduction section above, the maxLength property is set to 22. One would expect a length-based constraint to require a getMaxLength() method which could then be fed to the Constraint to find the maximum length.

org.kuali.rice.kns.datadictionary.validation.constraint.SimpleConstraint defines what we might call a "nervous system classic" constraint. It is built from normal fields on AttributeDefinition - required; maxLength and minLength (the latter has been added as part of KRAD); exclusiveMin and exclusiveMax; and finally, minOccurs and maxOccurs, which will be covered in more detail below.

The required constraint, of course, means that some value must be set for the attribute. The maxLength and minLength attributes typically apply to String data, which must be a certain size. Likewise, exclusiveMin and exclusiveMax apply to numeric data which must fit within some set range.

Valid Characters Constraints

Another hold over from the Kuali Nervous System constraints, ValidCharactersConstraint exists to make sure that a String value matches against a regular expression. For instance, let's say that a KRAD application requires that all phone numbers must be in the form of (###) ###-#### (Evidently, the attribute does not yet accept international numbers...but as developers, we must rest assured that's coming, and is the requirement.)

In the data dictionary for that attribute, the following could be set.

Code snippet example follows:

```
1.  <bean id="DataObject-phoneNumber" parent="AttributeDefinition">
2.    <property name="name" value="phoneNumber" />
3.    <property name="validCharactersConstraint">
4.      <bean class="org.kuali.rice.kns.datadictionary.validation.constraint.ValidCharactersConstraint">
5.        <property name="value" value="\(\d{3}\) \d{3}-\d{4}" />
6.      </bean>
7.    </property>
8.  </bean>
```

In lines 3 through 7, we set the validCharactersConstraint property on the AttributeDefinition, handing the bean we just created a regex which should match the phone number pattern which the requirements say all phone numbers should match.

This regex is passed in as the value property to the ValidCharactersConstraint bean. A number of ValidCharacterConstraints are defined in *org/kuali/rice/kns/bo/datadictionary/ DataDictionaryBaseTypes.xml*. Among those are "UrlPatternConstraint", "DatePatternConstraint", "CreditcardPatternConstraint", "NonWhitespacePatternConstraint", "IntegerPatternConstraint", "PhoneUSPatternConstraint", and "TimePatternConstraint", as a mere sampling. As of the time of this writing, the constraints only worked for javascript side validation. However, work was being done to build server side equivalencies of all of these patterns.

Finally, note that the ValidCharactersConstraint has a second property, "jsValue". In most cases, Java's regular expression engine (ValidCharactersConstraint uses the built-in regular expression engine) will accept the same expressions as the JavaScript engine. That's good, because the same regular expression can be passed to the client and handled client side, as will be covered in more detail soon.

The best idea is to keep validation regular expressions to the use of broadly supported features (outside of POSIX, which Java supports but which most JavaScript engines do not), and keep on eye on engine comparison pages such as http://en.wikipedia.org/wiki/Comparison_of_regular_expression_engines. The KRAD team is attempting to avoid differences, though, and create a single pattern for both JavaScript and Java.

Dependency Constraints

Dependency constraints are used to define a set of PrerequisiteConstraint dependencies on an attribute. A PreRequisiteConstraint is simply used to denote that some other attribute be required. If the attribute is non-empty and has dependency constraints, each pre-requisite constraint attribute must also be non-empty. Note the prerequisite constraint is also used in the MustOccurConstraint. Unlike the MustOccurConstraint which requires that a minimum or maximum number of prerequisite constraints be satisfied, a dependency constraint requires that all pre-requisite constraints be satisfied.

A code snippet example follows:

```
1.  <bean id="DataObject-phoneNumber" parent="AttributeDefinition">
2.    <property name="name" value="phoneNumber"/>
3.    <property name="dependencyConstraints">
4.      <list>
5.        <bean class="org.kuali.rice.kns.datadictionary.validation.constraint.PrerequisiteConstraint"
 p:attributePath="phoneExtension" />
6.      </list>
7.    </property>
8.  </bean>
```

Lookup Constraints

These are constraints on values returned from lookups into an attribute. As of the time of this writing, they're still in process of implementation.

Conditional Logic Constraints

All of the constraints so far covered are static, in a fashion. Once declared, they will apply to their attributes no matter what. However, let's say that a constraint should only be tested when the attribute has a certain value. How could the constraint be turned off if that value isn't present and only be applied if the attribute has the given value?

The final constraint to look at is *org.kuali.rice.kns.datadictionary.validation.constraint.CaseConstraint*, which will turn on and off child constraints if attributes match certain values. The classic example of using this is in an international address form. If the country code is the United States, then the state code should be filled in as well. If the country code is for Canada or Turkey, a province should be filled in. That would be done via a configuration like this.

A code snippet example follows:

```
 1. <bean id="DataObject-countryCode" parent="AttributeDefinition">
 2.   <property name="name" value="countryCode"/>
 3.   <property name="caseConstraint">
 4.     <bean class="org.kuali.rice.kns.datadictionary.validation.constraint.CaseConstraint">
 5.       <property name="whenConstraint">
 6.         <list>
 7.           <bean class="org.kuali.rice.kns.datadictionary.validation.constraint.WhenConstraint">
 8.             <property name="values">
 9.               <list>
10.                 <value>US</value>
11.               </list>
12.             </property>
13.             <property name="constraint">
14.               <bean
 class="org.kuali.rice.kns.datadictionary.validation.constraint.PrerequisiteConstraint"p:attributePath="state" /
>
15.             </property>
16.           </bean>
```

```
17.              <bean class="org.kuali.rice.kns.datadictionary.validation.constraint.WhenConstraint">
18.            <property name="values">
19.              <list>
20.                 <value>CA</value>
21.                 <value>TR</value>
22.              </list>
23.            </property>
24.            <property name="constraint">
25.               <bean
 class="org.kuali.rice.kns.datadictionary.validation.constraint.PrerequisiteConstraint"p:attributePath="province"/
>
26.               </property>
27.             </bean>
28.           </list>
29.         </property>
30.       </bean>
31.     </property>
32. </bean>
```

Obviously, for such a powerful constraint, configuration becomes a bit more complex. A *CaseConstraint* has a List of *WhenConstraints*. *WhenConstraints* match values to constraints that should be run when the attribute's value matches the *WhenConstraint*'s values. Here, values are hard coded (lines 8 through 12 and lines 18 through 23) but they need not be. If the values are in other attributes, a List of valuePaths can be specified.

A *WhenConstraint* also has one child constraint to match. In both of the *WhenConstraints* above, a *PrerequisiteConstraint* is used to make sure that another attribute - either state or province - is non-empty (lines 14 and 25). Any Constraint could be used as the child of the *WhenConstraint* - a SimpleConstraint, another CaseConstraint, and so on. The ability to turn on and off constraints such can lead to very powerful validations being built directly in the DataDictionary.

Ocurrences Constraints

An occurrence constraint states that for a given attribute to be valid, a certain number of prerequisite conditions must be matched. A prerequisite condition simply means that another attribute with a specified attribute path is non-empty (so Strings must have some text in them; Collections must have at least one member; or the attribute must otherwise not be null). These constraints thus handle situations where one or more of a number of fields are required.

An occurrence constraint is specified via the MustOccurConstraint constraint. Let's say that an application requires either a phone number, an e-mail address, or a time for showing up be specified as contact information. The following example sets up that validation in the data dictionary, adding the error to the phone number attribute (though the same constraint could be copied to the other attributes just as easily).

A code snippet example follows:

```
1.    <bean id="DataObject-phoneNumber" parent="DataObjectEntry">
2.      <property name="objectClass" value="edu.sampleu.contact.ContactInformation" />
3.      <property name="mustOccurConstraints">
4.        <list>
5.          <bean class="org.kuali.rice.kns.datadictionary.validation.constraint.MustOccurConstraint">
6.            <property name="min" value="1" />
7.            <property name="max" value="3"/>
8.            <property name="prerequisiteConstraints">
9.              <list>
10.                <bean
 class="org.kuali.rice.kns.datadictionary.validation.constraint.PrerequisiteConstraint"p:attributePath="phoneNumber"/
>
11.                <bean
 class="org.kuali.rice.kns.datadictionary.validation.constraint.PrerequisiteConstraint"p:attributePath="emailAddress"/
>
12.                <bean
 class="org.kuali.rice.kns.datadictionary.validation.constraint.PrerequisiteConstraint"p:attributePath="showUpTime"/
>
```

```
13.             </list>
14.           </property>
15.         </bean>
16.       </list>
17.     </property>
18.   </bean>
```

Lines 1 and 2 surprising show that this constraint has been set at the DataObjectEntry level, not that of the AttributeDefinition. While MustOccurConstraints can be set in pretty much the same way on AttributeDefinitions, since several attributes are involved, it makes more sense to have the validation at a higher level. At the time of this writing, MustOccurConstraint is the only validation which can be set at the DataObjectLevel.

The min and max properties of lines 6 and 7 tell the constraint how many of the following properties must be present and the maximum number of filled in properties we expect. Here, the min is 1 - so at least one of the properties must be filled in - and the max is 3, so if all three are filled in, the validation will still work fine. If there was a desire to only have one attribute filled in, the max could have been set to 1.

The MustOccurConstraint has a list of prerequisiteConstraints - lines 8 through 14 - which describe which attributes are grouped by this constraint.

A MustOccurConstraint can also have a list of child MustOccurConstraints. Why would such a thing be desirable? Because it provides a way to set up nested validations. Let's say that, instead of specifying a show up time, we had an address which needed to be filled in. If that was the case, we'd need every field of the address - street, city, state, and zip filled in - for the constraint to pass. In that case, we would have left min and max at 1 and 3 respectively; but instead of line 12, we would have specified a value for the property mustOccurConstraints at line 15, and added a list of constraints asking for all the address attributes to be filled in.

Collection Size Constraints

Another common rule situation is when a collection is the child of a data object or document, and for that data object or document to be valid, a certain number of elements must be available in the collection. For instance, on an Add Course Document, one would expect the "courses" collection to have at least one course in it and to be less than the total number of courses a student is allowed to take in a semester or quarter. Therefore in the data dictionary entry for the data object, one adds a constraint as follows,

A code snippet example follows:

```
 1. <bean name="AddCourseDocument" parent="BusinessObjectEntry">
 2.    ...
 3.    <property name="collections">
 4.      <list>
 5.        <bean parent="CollectionDefinition" p:name="courses" p:label="Courses">
 6.          <property name="minOccurs" value="1" />
 7.          <property name="maxOccurs" value="74" />
 8.        </bean>
 9.      </list>
10.    </property>
11.    ...
12. </bean>
```

Evidently, some students can take up to seventy four classes. Busy student. CollectionSizeConstraint is handled as a special type of SimpleConstraint (though only for use with CollectionDefinitions). Simply set the minOccurs and maxOccurs for the attribute and there will be an error if the collection size falls outside those limits. Naturally, either the minOccurs or maxOccurs can be left out for collections which should be unbounded in either lower or upper size limit.

Constraints on the client side

One of the tasks that org.kuali.rice.kns.uif.field.AttributeField does in its Finalization stage is to convert constraints to JavaScript. For all of the following constraints, AttributeField automatically will push the Constraint to the client side:

- Exclusive Minimum and Inclusive Maximum constraints

- ValidCharactersConstraint

- CaseConstraint

- DependencyConstraint

- MustOccursConstraint

- PreRequisiteConstraint

When the user attempts to take action on the page, this level of Constraints will kick in - meaning that feedback comes much more quickly. These constraints will always be called when buttons such as save, submit, or approve - buttons where business logic would typically be evaluated - are clicked. They will also occur on onBlur's for most fields.

There are, however, certain constraints which apply to multiple fields: CaseConstraints and PreRequisiteConstraints, to name two instances. Which onBlur issues the error among all of those fields? Generally, KRAD attempts to not give an error until the user has gone past a point where she or he could have prevented said error. For instance, if oneField has a DependencyConstraint on two other fields, but those two other fields render later and lower on the page, then KRAD will associate the validation with the last, bottom-most field of those the constraint applies to. In an interesting corollary, KRAD will issue an error on the onFocus event for a field which has already been visited if an error occurs with that field.

Of course, the constraints are still run just the same on the server side once the page has been submitted; that way, if the user has scripting turned off, the constraints are still run and user input data gets validated.

Changing Error Messages

All of the covered constraints are associated with standard error messages. For instance, if a "required" constraint has been violated, the user will get the following message:

```
Phone Number is a required field.
```

This has taken the label from the attribute which violated the constraint and formatted that into the standard error.required message in the KR-ApplicationResources.properties file.

With most of the constraints, the message can be overridden on the constraint, by specifying the "labelKey" property. For instance, a configuration like this:

Code snippet example follows:

```
1.  <bean class="org.kuali.rice.kns.datadictionary.validation.constraint.MustOccurConstraint">
2.    <property name="min" value="1" />
3.    <property name="max" value="3" />
4.    <property name="labelKey" value="error.must.be.able.to.track.down" />
```

Instead of using the standard message, the error message that is shown will be the message associated with the "error.must.be.able.to.track.down" key. This allows for a great deal more flexibility in what message gets displayed - though, the classic messages will still show up as they always did if nothing else is specified.

Constraint Architecture (building a custom constraint)

The constraints that come standard with KRAD provide a lot of power through configuration. For example, validating user input will be easier than ever. And the constraint sub-system of KRAD was built with the realization that even more constraints will be added in the future. Because of that, there needs to be an easy way for Kuali application developers or even future versions of Rice to add new constraints into the system. And so, there is.

A constraint is a marker interface which is implemented by any Constraint bean (Java). These Constraint beans are purely configuration - they only hold what regex should be parsed against, if a field is required or not: basic information. The Constraint, in turn, is passed to an implementation of org.kuali.rice.kns.datadictionary.validation.processor.ConstraintProcessor.

Note that implementations of ConstraintProcessors can be genericized with both the type of value that the processor expects and the type of Constraint that the processor will work on. Most ConstraintProcessor implementations only genericize the Constraint, accepting any Object as a value to validate.

ConstraintProcessors have four methods:

- First, the getName() method returns the name that the constraint processor holds.

- The getConstraintType() method returns the implementation of Constraint that this processor has the business logic for.

- The isOptional() method returns a boolean: true if the processor can be turned off in certain situations by another piece of code, false otherwise.

  The only constraint which is currently optional is the ExistenceConstraint; it is turned off by passing a false in the doOptionalProcessing parameter of DictionaryValidationService#processConstraints.

- The final method is the one that contains the ConstraintProcessor's business logic: process. Process takes in DictionaryValidationResult, a value of some type, the Constraint information to apply to the value, and an AttributeValueReader if the value needs yet to be read; it returns an instance of org.kuali.rice.kns.datadictionary.validation.result.ProcessorResult.

ProcessorResults typically wrap instances of org.kuali.rice.kns.datadictionary.validation.result.ConstraintValidationResult. A ConstraintValidationResult encapsulates a number of possible outcomes for the validation, all generated by org.kuali.rice.kns.datadictionary.validation.result.DictionaryValidationResult.

DictionaryValidationResult's addError method, for example, returns a ConstraintValidationResult which contains an error about a constraint being broken. Likewise, DictionaryValidationResult's addSuccess method indicates that the result of the constraint test was positive - the value passed the constraint.

The other outcomes that DictionaryValidationResult can generate is addWarning - which gives an informative message that something is wrong with the attribute's value but which will not "fail"; addSkipped, which says that the value could not be tested and therefore the validation was not run; and finally addNoConstraint, which means that the constraint was configured in such way as to not run for the given value or at all.

DictionaryValidationResults wrap ConstraintValidationResults in a way which provides easy access to these results in the data dictionary. These ConstraintValidationResults are passed back to KRAD wrapped

within the ProcessorResults; the ProcessorResults then ensures that proper logic - whether that be the display of a message, the stopping of logic, or - if everything passed - carrying on with the transaction - occurs.

That covers ConstraintProcessors. Now on to how they are called from within an application. An implementation of org.kuali.rice.kns.service.DictionaryValidationService is responsible for checking all of the attributes which are passed in as part of a request into a KRAD form. The configuration of the default implementation of DictionaryValidationService has all of the ConstraintProcessors for KRAD passed into it. See the following code snippets. For example, if we assume the following 5 lines of code,

```
3.    <property name="collectionConstraintProcessors">
4.      <list>
5.      <bean class="org.kuali.rice.kns.datadictionary.validation.processor.CollectionSizeConstraintProcessor"/>
6.    </list>
7.    </property>
```

Lines 3 through 7 above are the collectionConstraintProcessors - constraints which apply to collections. Here is where CollectionSizeConstraint - the constraint that handles the maxOccurs and minOccurs constraint attributes - goes.

In line 5, the CollectionSizeConstraintProcessor is injected in. DictionaryValidationServiceImpl then matches the active Constraints on an attribute with the ConstraintProcessors passed in, and runs the logic against the constrained attribute. If the ConstraintProcessor acts only on a single attribute, it is passed into the elementConstraintProcessor property.

ConstraintProcessors are supplied to engines which validate against constraints - DictionaryValidationServiceImpl, for instance - via ConstraintProviders. Different implementations of the *org.kuali.rice.kns.datadictionary.validation.constraint.provider.ConstraintProvider* interface can exist; their job is to map an implementation of Constrainable (a simple interface all Constraints implement) to constraint processors, as can be seen in lines 29-79 below.

The usefulness of ConstraintProviders can be seen in the example. Lines 31-64 shows the mapping for the AttributeDefinitionConstraintProvider - constraints which can be run against an attribute definition. Lines 65-77 shows that only one constraint - the MustOccurConstraint - can be run for ObjectDictionaryEntryConstraintProviders, meaning this is the sole constraint supported by data dictionary entries for entire data objects.

<u>Code snippet example follows:</u>

```
1. <bean id="dictionaryValidationService"
class="org.kuali.rice.kns.service.impl.DictionaryValidationServiceImpl">
     ...
     contents trimmed
     ...
2.    <!-- Collection constraint processors are classes that determine if a feature of a collection of objects
satisfies some constraint -->
3.    <property name="collectionConstraintProcessors">
4.      <list>
5.      <bean class="org.kuali.rice.kns.datadictionary.validation.processor.CollectionSizeConstraintProcessor"/>
6.    </list>
7.    </property>
8.    <!-- Element constraint processors are classes that determine if a passed value is valid for a specific
constraint at the individual object or object attribute level -->
9.    <property name="elementConstraintProcessors">
10.     <list>
11.     <bean class="org.kuali.rice.kns.datadictionary.validation.processor.CaseConstraintProcessor"
12.          parent="mandatoryElementConstraintProcessor"/>
13.     <bean class="org.kuali.rice.kns.datadictionary.validation.processor.ExistenceConstraintProcessor"/>
14.     <bean class="org.kuali.rice.kns.datadictionary.validation.processor.DataTypeConstraintProcessor"
15.     parent="mandatoryElementConstraintProcessor"/>
```

```
16.    <bean class="org.kuali.rice.kns.datadictionary.validation.processor.RangeConstraintProcessor"
17.        parent="mandatoryElementConstraintProcessor"/>
18.    <bean class="org.kuali.rice.kns.datadictionary.validation.processor.LengthConstraintProcessor"
19.        parent="mandatoryElementConstraintProcessor"/>
20.    <bean class="org.kuali.rice.kns.datadictionary.validation.processor.ValidCharactersConstraintProcessor"
21.        parent="mandatoryElementConstraintProcessor"/>
22.    <bean class="org.kuali.rice.kns.datadictionary.validation.processor.PrerequisiteConstraintProcessor"
23.        parent="mandatoryElementConstraintProcessor"/>
24.    <bean class="org.kuali.rice.kns.datadictionary.validation.processor.MustOccurConstraintProcessor"
25.        parent="mandatoryElementConstraintProcessor"/>
26.    </list>
27.    </property>
28.    <!-- Constraint providers are classes that map specific constraint types to a constraint resolver, which
 takes a constrainable definition -->
29.    <property name="constraintProviders">
30.     <list>
31.      <bean
 class="org.kuali.rice.kns.datadictionary.validation.constraint.provider.AttributeDefinitionConstraintProvider">
32.       <!--
33.           individual constraint resolvers can be injected as a map keyed by constraint type as string, or
 the default
34.           resolvers can be instantiated into the map by adding 'init-method="init"' to the bean
 declaration above
35.           -->
36.           <property name="resolverMap">
37.     <map>
38.      <entry key="org.kuali.rice.kns.datadictionary.validation.constraint.CaseConstraint">
39.      <ref bean="dictionaryValidationCaseConstraintResolver"/>
40.     </entry>
41.     <entry key="org.kuali.rice.kns.datadictionary.validation.constraint.ExistenceConstraint">
42.      <ref bean="dictionaryValidationDefinitionConstraintResolver"/>
43.     </entry>
44.     <entry key="org.kuali.rice.kns.datadictionary.validation.constraint.DataTypeConstraint">
45.          <ref bean="dictionaryValidationDefinitionConstraintResolver"/>
46.     </entry>
47.     <entry key="org.kuali.rice.kns.datadictionary.validation.constraint.LengthConstraint">
48.      <ref bean="dictionaryValidationDefinitionConstraintResolver"/>
49.     </entry>
50.     <entry key="org.kuali.rice.kns.datadictionary.validation.constraint.ValidCharactersConstraint">
51.      <ref bean="dictionaryValidationValidCharactersConstraintResolver"/>
52.     </entry>
53.     <entry key="org.kuali.rice.kns.datadictionary.validation.constraint.PrerequisiteConstraint">
54.      <ref bean="dictionaryValidationPrerequisiteConstraintsResolver"/>
55.     </entry>
56.     <entry key="org.kuali.rice.kns.datadictionary.validation.constraint.MustOccurConstraint">
57.      <ref bean="dictionaryValidationMustOccurConstraintsResolver"/>
58.     </entry>
59.     <entry key="org.kuali.rice.kns.datadictionary.validation.constraint.CollectionSizeConstraint">
60.      <ref bean="dictionaryValidationDefinitionConstraintResolver"/>
61.     </entry>
62.          </map>
63.        </property>
64.      </bean>
65.      <bean
 class="org.kuali.rice.kns.datadictionary.validation.constraint.provider.ObjectDictionaryEntryConstraintProvider">
66.        <!--
67.       individual constraint resolvers can be injected as a map keyed by constraint type as string, or the
 default
68.       resolvers can be instantiated into the map by adding 'init-method="init"' to the bean declaration
 above
69.     -->
70.          <property name="resolverMap">
71.     <map>
72.      <entry key="org.kuali.rice.kns.datadictionary.validation.constraint.MustOccurConstraint">
73.      <ref bean="dictionaryValidationMustOccurConstraintsResolver"/>
74.     </entry>
75.     </map>
76.    </property>
77.    </bean>
78.     </list>
79.    </property>
80. </bean>
```

Other ConstraintProviders packed into Rice at the time of this writing are CollectionDefinitionConstraintProvider - constraints which work for collection

definitions; and ComplexAttributeDefinitionConstraintProvider, which supports constraints for "ComplexAttributeDefinitions" - data dictionary entries for attributes on one DataObject which are represented by another data object.

KRAD Business Objects?

(Need direction on what of the KNS information should be copied here and what new information should be included or if this section is not needed.)

KRAD Class Libraries?

(Need direction on what new information should be included here or if this section is not needed.)

Installing and Configuring KRAD

Before developing with KRAD and after installing and configuring Rice, here are the additional steps you'll need to follow to configure KRAD before starting to develop an application.

This information below assumes you already have Rice installed and configured for your database. Below are the additional tasks required to configure KRAD. For more information, see the KRAD Installation Guide and KRAD javadocs.

(TBD - Revise the section heads below as needed and then populate with info. Include instructions for setting up a Rice project, include assumptions for what is already done and not covered in the instructions, such as setting up all else needed for development environment - what are pre-reqs, what are co-reqs, etc..)

Configure Rice without KRAD (KNS Only)

In some cases it may be desirable to only use the KNS without KRAD. For example if you're timelines push a conversion to KRAD out into the future, you may see some benefits with startup performance and with memory usage.

You can override the kradApplicationModuleConfiguration bean to not include any of the files in the UIF folder. That is, you only need to include these files:

```
<property name="dataDictionaryPackages">
<list>
<value>classpath:org/kuali/rice/krad/bo/datadictionary/AdHocRoutePerson.xml</value>
<value>classpath:org/kuali/rice/krad/bo/datadictionary/AdHocRouteWorkgroup.xml</value>
<value>classpath:org/kuali/rice/krad/bo/datadictionary/Attachment.xml</value>
<value>classpath:org/kuali/rice/krad/bo/datadictionary/AttributeReferenceDummy.xml</value>
<value>classpath:org/kuali/rice/krad/bo/datadictionary/AttributeReferenceElements.xml</value>
<value>classpath:org/kuali/rice/krad/bo/datadictionary/BusinessObjectAttributeEntry.xml</value>
<value>classpath:org/kuali/rice/krad/bo/datadictionary/DataDictionaryBaseTypes.xml</value>
<value>classpath:org/kuali/rice/krad/bo/datadictionary/DocumentHeader.xml</value>
<value>classpath:org/kuali/rice/krad/bo/datadictionary/Note.xml</value>
<value>classpath:org/kuali/rice/krad/bo/datadictionary/NoteType.xml</value>
<value>classpath:org/kuali/rice/krad/bo/datadictionary/PessimisticLock.xml</value>
</list>
</property>
```

Likewise, this can be done for the 'baselinePackages' property on the dataDictionaryService bean.

Creating the KRAD database tables / connections to data?

KRAD Configurer and RiceConfigurer?

Configuring Spring and MVC?

Module Configuration – Loading Data Dictionary and OJB Files?

Other KRAD Configuration Parameters?

Building application pages using KRAD

This information assumes you've already installed Rice. Once Rice is installed and set up, you can use KRAD to build applications. You can use the code snippet templates covered below, and you can look through the codebase itself or the sample application to see the code snippets for each of the KRAD features, and then copy/paste them to use in your developing application.

KRAD Templates

Live templates contain predefined code fragments. You can use them to insert frequently-used or custom code constructs into your source code file quickly, efficiently, and accurately.

Loading the KRAD Templates

The following have been tested in the IntelliJ IDE.

Download the KRAD Templates File and place into the following location: (ACTION/TO-DO -- Need to specify the link where we will maintain this long-term!)

- Windows: <your home directory>\.<product name><version number>\config\templates

- Linux: ~\.<product name><version number>\config\templates

- MacOS: ~/Library/Preferences/<product name><version number>/templates

Using Templates

While in an XML file, type the template abbreviation and then the space key. The completion key (setup as space) can be changed if desired by going to settings-live templates. Your cursor will then be inserted into the location specified by the template (marked with the END variable).

Available KRAD Templates

Table 6.1. Available KRAD Templates

Abbreviation	Description	Code
@	Inserts the expression placeholders	@{$END}
action	Generates an action field	`<bean parent="ActionField" p:actionLabel="$END$" p:methodToCall=""/>`
alink	Generates an action link field	`<bean parent="ActionLinkField" p:actionLabel="$END$" p:methodToCall=""/>`
be	Generates a bean tag	`<bean parent="$END$"/>`
cc	Generates a checkbox control	`<bean parent="CheckboxControl"/>`
cgc	Generate a checkbox group control	`<property name="control"> <bean parent="CheckboxGroupControl"/> </property> <property name="optionsFinder"> <bean class="$END$"/> </property>`
dc	Generates a date control	`<bean parent="DateControl" p:size="$END$"/>`
fc	Generates a file control	`<bean parent="FileControl" p:size="$END$"/>`
fg	Generates a field group	`<bean parent="FieldGroup" p:label="$END$"> <property name="items"> <list> </list> </property> </bean>`
fi	Generates a field inquiry	`<property name="fieldInquiry.dataObjectClassName" value="$END$ $CLASS$"/> <property name="fieldInquiry.inquiryParameters" value=""/>`
fl	Generates a field lookup	`<property name="fieldLookup.dataObjectClassName" value="$END$ $CLASS$"/> <property name="fieldLookup.fieldConversions" value=""/> <property name="fieldLookup.lookupParameters" value=""/>`
fs	Generates a field suggest	`<property name="fieldSuggest.render" value="true"/> <property name="fieldSuggest.suggestQuery.dataObjectClassName" value="$END$ $CLASS$"/> <property name="fieldSuggest.sourcePropertyName" value=""/>`
group	Generates a group	`<bean id="$END$" parent="Group" p:title="" p:instructionalText=""> <property name="items"> <list> </list> </property> </bean>`
hfg	Generates a field group with horizontal layout	`<bean parent="HorizontalFieldGroup" p:label="$END$"> <property name="items"> <list> </list> </property> </bean>`
image	Generates an image field	`<bean parent="ImageField" p:label="$END$" p:altText="" p:source="@#ConfigProperties['krad.externalizable.images.url']"/>`
input	Generates an input field	`<bean parent="InputField" p:propertyName="$END$" p:label=""> <property name="control"> </property> </bean>`
link	Generates a link field	`<bean parent="LinkField" p:linkLabel="$END$" p:hrefText=""/>`
mess	Generates a message field	`<bean parent="MessageField" p:messageText="$END$"/>`
page	Generates a page	`<bean id="$END$" parent="Page" p:title=""> <property name="items"> <list> </list> </property> </bean>`
prop	Inserts a property tag	`<property name="$END$" value=""/>`
rc	Generates a radio group control	`<property name="control"> <bean parent="RadioGroupControl"/> </property> <property name="optionsFinder"> <bean class="$END$"/> </property>`
sc	Generates a select control	`<property name="control"> <bean parent="SelectControl"/> </property> <property name="optionsFinder"> <bean class="$END$"/> </property>`
section	Generates a section group	`<bean id="$END$" parent="GroupSection" p:title="" p:instructionalText=""> <property name="items"> <list> </list> </property> </bean>`
sstack	Generates a collection group section with stacked layout	`<bean id="END" parent="CollectionGroupSection" p:layoutManager.numberOfColumns="" p:title="" p:instructionalText=""> <property name="collectionObjectClass" value="$CLASS$"/> <property name="propertyName" value=""/> <property name="layoutManager.summaryTitle" value="" /> <property name="layoutManager.summaryFields" value="" /> <property name="items"> <list> </list> </property> </bean>`
stable	Generates a collection group section with table layout	`<bean id="END" parent="CollectionGroupSectionTableLayout" p:layoutManager.numberOfColumns="" p:title="" p:instructionalText=""> <property name="collectionObjectClass" value="$CLASS$"/> <property name="propertyName" value=""/> <property name="layoutManager.sequencePropertyName" value=""/> <property name="items"> <list> </list> </property> </bean>`

Abbreviation	Description	Code
stack	Generates a collection group with stacked layout	`<bean id="END" parent="CollectionGroup" p:layoutManager.numberOfColumns="" p:title="" p:instructionalText=""> <property name="collectionObjectClass" value="$CLASS$"/> <property name="propertyName" value=""/> <property name="layoutManager.summaryTitle" value="" /> <property name="layoutManager.summaryFields" value="" /> <property name="items"> <list> </list> </property> </bean>`
table	Generates a collection group with table layout	`<bean id="END" parent="CollectionGroupTableLayout" p:layoutManager.numberOfColumns="" p:title="" p:instructionalText=""> <property name="collectionObjectClass" value="$CLASS$"/> <property name="propertyName" value=""/> <property name="layoutManager.sequencePropertyName" value=""/> <property name="items"> <list> </list> </property> </bean>`
tac	Generates a text area control	`<bean parent="TextAreaControl" p:rows="$END$" p:cols=""/>`
tc	Generates a text control	`<bean parent="TextControl" p:size="$END$"/>`
view	Generates a view	`<bean id="END" parent="FormView"> <property name="title" value=""/> <property name="navigation"> <ref bean=""/> </property> <property name="items"> <list> </list> </property> <property name="additionalCssFiles" ref=""/> <property name="additionalJsFiles" ref=""/> <property name="viewHelperServiceClassName" value=""/> <property name="defaultBindingObjectPath" vaue=""/> <property name="formClass" value=""/> </bean>`

Creating your own Templates

See http://www.jetbrains.com/idea/webhelp/live-templates.html

Please post back and share!

Converting KNS pages to KRAD

(other? E/R diagrams?, binding paths?, pointer to javadocs?)

Chapter 7. KRMS

KRMS Overview

What is a Rule Management System, in general?

Wikipedia defines a business rule management system, in general, as follows: "a [software](#) system used to define, deploy, execute, monitor and maintain the variety and complexity of decision logic that is used by operational systems within an organization or enterprise. This logic, also referred to as [business rules](#), includes policies, requirements, and conditional statements that are used to determine the tactical actions that take place in applications and systems."

A key aspect of a rules management system is that it enables rules to be defined and maintained separately from application code. This modularity has the potential to reduce application maintenance costs, enable increased automation and application flexibility, and to enable business analysts and business process experts who are not developers and who reside outside of the IT organizations in the business departments themselves, to be more directly involved in creating and managing their rules.

A rules management system in general includes a repository of decision logic and a rules engine that can be executed by applications in a run-time environment. Again from wikipedia: "... provides the ability to: register, define, classify, and manage all the rules, verify consistency of rules definitions ("Gold-level customers are eligible for free shipping when order quantity > 10" and "maximum order quantity for Silver-level customers = 15"), define the relationships between different rules, and relate some of these rules to IT applications that are affected or need to enforce one or more of the rules."

What is Kuali's Rule Management System (KRMS), in particular?

Kuali's Rule Management System (KRMS) supports the creation, maintenance, storage and retrieval of business rules and agendas (ordered sets of business rules) within business contexts (e.g., for a particular department or for a particular university-wide process).

KRMS enables you to define a set of rules within a particular business unit or for a particular set of applications. These business rules test for certain conditions and define the set of actions that result when conditions are met. KRMS enables you to call and use this logic from any application, without having to re-write and manage all the rules' logic within the application.

Integration with organizational hierarchies and structures can be accomplished today using KEW for routing and approval, and KEW also has a legacy rule system of its own that can be used to make routing decisions. But before KRMS, managing general customizable business logic such as "if the transaction date is in the future OR the transaction date is less than the account activation date then flag the transaction for review" was the responsibility of the applications themselves. KRMS now offers a way to manage this type of logic externally in a repository that allows for business analysts to change it without having to modify application code.

Because KRMS is a general-purpose business rules management system, you can use it for many things, for example, you can define a rule to specify that when an account is closed, a continuation account must be specified. You can also define rules to manage your organizational hierarchies and internal structures. You can define compound propositional logic, for example, "Must meet":

• P1 - 12 credits of FirstYearScience (CLU set)

AND

- P2 - Completed CALC101 with grade >= B+

AND

- p3 - Average of B+ on last 12 credits

What problems or functions does KRMS solve?

KRMS gives business applications a powerful tool to externalize logic in places where customization will often be needed. This lowers the costs of adopting and administering the application by reducing the need for changes to the software itself, and allows the application to more fluidly reflect the institution's desired business processes.

There are a wide variety of actions that KRMS rules can be used to govern:

- Workflow Action rules - e.g. route an approval request

- Notification rules - e.g. send a notification to these people

- Validation rules - e.g. display this validation error message

- Questionnaire rules - e.g. administer this questionaire

- Custom-developed actions

For example, calling a KRMS set of rules (an agenda) from your application can result in routing a document to a PeopleFlow*, which is a new feature in KEW in Rice 2.0, or to any other action you define in KRMS.

* Essentially, it's like a mini people-based workflow that doesn't require you to specify a KEW node in the document type for each role, group or individual who might need to approve or be notified.

What problems does KRMS not address?

Some rule engines are built upon special algorithms that allow for [forward](#) or [backward chaining](#) (one example is [Rete](#)) that make them suitable for efficiently evaluating highly complex systems of what are known as production rules. The default engine implementation for KRMS is not designed upon such an algorithm, and it does not support either forward or backward chaining.

With which types of applications can KRMS integrate?

Any Rice-based application can use KRMS.

Can I use KRMS without building a Rice application?

The project has aspirations to increase Rice's modularity, and some strides have been made, but at the time of this writing the answer is no.

KRMS Concepts

## Namespaces, Contexts, Agendas, Rules and Propositions

Namespaces are the top level container in KRMS. They contain Contexts, KRMS Types, and all things related to Terms. There isn't a namespace entity in the KRMS schema, they are specified via namespace code fields on the applicable child entities.

Rules in KRMS are placed into ordered sets called Agendas. The order of the Rules in an Agenda determines the sequencing: which rule gets evaluated first, second and so on. The Agenda also enables you to include conditional branching logic between Rules.

In turn, Agendas are are created in Contexts, which may represent any categories of rules that are relevant within your institution. For example, they will frequently correspond to document types, but they could be more finely grained to encompass only a certain kind of rule that you might run, e.g. you might have a context called "Proposal Validations". In some university environments, the following might be relevant contexts: Awards, Proposals, IRB reviews, Course co-requisites, Course pre-requisites, Student plan evaluations, and so on.

Each Context contains its own agendas, and each Agenda contains its own rules. Rules aren't shared across agendas (though you can copy/paste, they become unique Rule instances), and Agendas aren't shared across Contexts. There is no Context hierarchy, that is, Agendas and Rules can't be inherited across contexts within any sort of hierarchy.

The following diagram outlines the hierarchy of entities in KRMS (note that some entities are omitted)
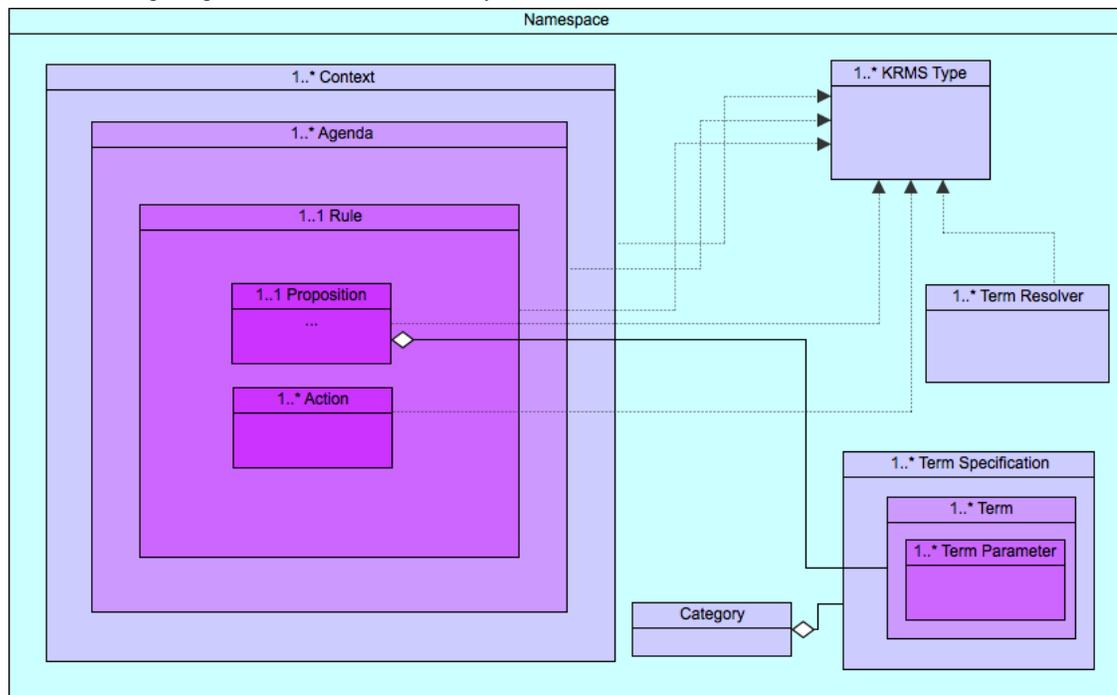
You'll also note that many of the entities in the above diagram are KRMS Types. In most cases (the notable exception is Context) what that means is that you can develop and integrate custom implementations of the engine objects associated with those entities. These include:

- Agendas with custom selection and execution code

- Actions with custom execution code

- Rules with custom evaluation and Action triggering code

- Propositions with custom evaluation code

- Term Resolvers with custom value resolution code

Propositions

Rules consist of propositions, and KRMS supports the following three main types of propositions:

1. Simple Propositions - a proposition of the form lhs op rhs where lhs=left-hand side, rhs=right-hand side, and op=operator

2. Compound Propositions - a proposition consisting of more than one simple proposition and a boolean algebra operator (AND, OR) between each of the simple propositions

3. Custom Propositions - a proposition which can optionally be parameterized by some set of values. Evaluation logic is implemented "by hand" and returns true or false.

The data model is designed in such a way to support each of these.

Next we'll look at each of the proposition tables in detail.

Proposition - krms_prop_t

Every proposition in the repository will have an entry in this table. Propositions are reference by a rule or another proposition (in the case of compound propositions). Propositions are never re-used across multiple rules.

Here is a summary of the non-common data elements in this proposition table:

Table 7.1. Non-common data elements in the proposition table

Column	Description
prop_id	A generated primary key identifier for the proposition
desc_txt	A plain-text description of the proposition
typ_id	Defines the PropositionType for the proposition. Defined in the krms_typ_t table.
dscrm_typ_cd	Discriminator type code which defines if the proposition is compound or simple. Valid values are C and S.

Proposition Parameters - krms_prop_parm_t

Each proposition can have zero or more parameters. The proposition parameter is the primary data element used to define the proposition. These parameters will be one of the following three types:

1. Constant Values

   - numbers

   - strings

   - dates

- etc.

2. Terms

  - data available in the execution environment and/or resolved by a term resolver

3. Functions

  - resolve to a value

  - could themselves take parameters of their own

  - typically defined externally to KRMS and then plugged in via a custom term resolver

4. Operators

  - one of a set of built-in "functions"

  - The full set of (currently) supported operators are as follows:

    - =

    - !=

    - >

    - <

    - >=

    - <=

To that end, the proposition parameter list should be modeled as a list in [Reverse Polish Notation](#) (RPN). This allows for arbitrary nesting of parameters, which may have parameters of their own. However, this requires that each function explicitly define the number of arguments that it expects. This will be specified when the function is defined, so the proposition system can assume this is available. This requirement does prohibit the use of functions that have a variable arity since the model currently does not have anyway to group parameters. So this will currently be unsupported.

Examples of proposition parameter lists defined using RPN are as follows:

- [campusCode, "BL", =] *equivalent to* campusCode="BL"

- [totalDollarAmount, availableAmount, >] *equivalent to* totalDollarAmount > availableAmount

- [award, getTotalDollarAmountForAward, award, getAvailableAmountForAward, >] *equivalent to* getTotalDollarAmount(award) > getAvailableAmountForAward(award)

In the cases above the following are constants:

- "BL"

The following are terms:

- campusCode

- totalDollarAmount

- availableAmount

- award

The following are functions:

- getTotalDollarAmountForAward

- getAvailableAmountForAward

And the following are operators:

- =

- >

Here is a summary of the non-common data elements in this proposition parameter table:

Table 7.2. Non-common data elements in the proposition parameter table

Column	Description
prop_parm_id	A generated primary key identifier for the proposition parameter
prop_id	The proposition which this parameter applies to
parm_val	the value of the parameter
parm_typ_cd	Indicates whether the parameter value represents a constant, term, or function. Valid values are C, T, F, O
seq_no	Defines the order of the parameter within the larger parameter list.

KRMS Administration Guide

(work in progress - content tdb. The below preface is patterned after the KEW TRG - what will admins need to administer for KRMS? I've put in some placeholder content-topics for a TOC skeleton.)

This guide provides information on administering a Kuali Rules Management System (KRMS) installation. Out of the box, KRMS comes with a default setup that works well in development and test environments. However, when moving to a production environment, this setup requires adjustments. This document discusses basic administration as well as instructions for working with some of KRMS' administration tools.

Initial Set up tasks

In this section we cover the types of tasks you'll need to do as a one-time setup at your institute in order for you and others to be able to define KRMS agendas for use by applications.

What do I have to install so that people can use KRMS?

What do I have to create or customize so that people can work with business contexts, agendas, and rules?

Below are the constructs you will need to point to or create for your institute:

- Use existing Namespaces or set up Namespaces for KRMS

- Use an existing Agenda Type service or set up an Agenda Type service for KRMS

- Use existing Types or set up Types for KRMS

- Use existing Contexts or configure new Contexts for KRMS

- Specify Terms

- Create Term Resolvers

- Create Parameterized Terms

Below are the instructions for doing these tasks.

Point to or Set up Namespaces

You can use existing Namespaces or set up Namespaces specifically for KRMS (include information on how to do both of these).

Point to or Set up an Agenda Type service for KRMS

You can use an existing Agenda Type service or set up an Agenda Type service specifically for KRMS (include information on how to do both of these).

For example, below is a code snippet for setting up the Agenda Type service:

```
<bean id="campusAgendaTypeService"
      class="edu.sampleu.krms.impl.CampusAgendaTypeService">
    <property name="configurationService" ref="configurationService"/>
</bean>
```

Point to or Set up the Types for KRMS

You can use existing Types or set up Types for KRMS (include information on how to do both of these).

Below is example SQL Server code to insert the Type into the Agenda Type service -- be sure to replace the content of the 2nd parenthetical expressions in each of the following examples with your defined values:

- First, add the Type(s) itself:

  insert into krms_typ_t (typ_id, nm, nmspc_cd, srvc_nm, actv, ver_nbr) values (*'T6', 'Campus Agenda', 'KRMS_TEST', 'campusAgendaTypeService', 'Y', 1*);

- Next, add the campus attribute(s) to the Campus Agenda Type:

  insert into krms_attr_defn_t (ATTR_DEFN_ID, NM, NMSPC_CD, LBL, CMPNT_NM, DESC_TXT) values (*'Q9901', 'Campus', 'KRMS_TEST', 'campus label', null, 'the campus which this agenda is valid for'*);

  insert into krms_typ_attr_t (TYP_ATTR_ID, SEQ_NO, TYP_ID, ATTR_DEFN_ID) values (*'T6A', 1, 'T6', 'Q9901'*);

Point to or Set up Contexts for KRMS

You can use existing Contexts or configure new Contexts for KRMS. There is graphical user interface support for configuring a new Context, through a maintenance page. For example, in the Rice demo / sample application, on the Main menu page, under KRMS Rules, select the Context Lookup.

You can search for existing Contexts or create a new one. To create a new one, select "Create New" at the top right on the context lookup page:

The resulting Context Maintenance screen enables you to define a new Context. The Context ID must be unique:

After creating your Context(s), you must 1) set "CampusAgendaType" as valid*, 2) set "Route to PeopleFlow" action as valid* for them, and 3) make the Type(s) you created valid for your Context(s). See the following examples, and replace the content of each of the 2nd parenthetical expressions with your defined values:

- insert into krms_cntxt_vld_agenda_t (cntxt_vld_agenda_id, cntxt_id, agenda_typ_id, ver_nbr) values (*'agendaid', 'contextid', 'agendatypeid', version#*);

- insert into krms_cntxt_vld_actn_t (cntxt_vld_actn_id, cntxt_id, actn_typ_id, ver_nbr) values (*'agendaid', 'contextid', 'agendatypeid', version#*);

- insert into krms_cntxt_vld_agenda_t (cntxt_vld_agenda_id, cntxt_id, agenda_typ_id, ver_nbr) values (*''agendaid', 'contextid', 'agendatypeid', version#*);

Specify the Terms for KRMS

You can point to existing terms or specify new terms for KRMS (include information on how to do both of these).

To specify newTerms, you will probably want to first create term categories. See the following examples, and replace the content of each of the 2nd parenthetical expressions with your defined values:

- Example - Generic Workflow Properties

- • insert into krms_ctgry_t (ctgry_id, nm, nmspc_cd, ver_nbr) values (*'CAT02', 'Workflow Document Properties', 'KR-SAP', '1'*);

- Example - Travel Account Properties

- • insert into krms_ctgry_t (ctgry_id, nm, nmspc_cd, ver_nbr) values (*'CAT03', 'Travel Account Properties', 'KR-SAP', '1'*);

And next, you can use existing Terms or configure new Terms for KRMS. There is graphical user interface support for configuring a new Term, through a maintenance page. For example, in the Rice demo / sample application, on the Main menu page, under KRMS Rules, select the Term Specification Lookup and, after completing that, select the Term Lookup.
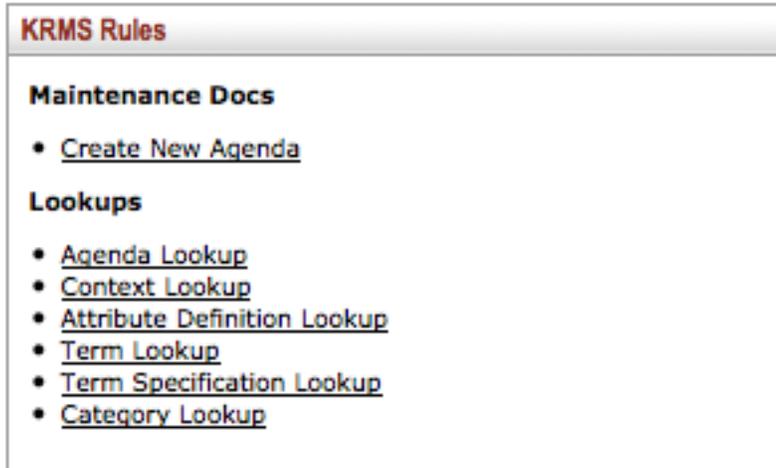
You can search for existing Term Specifications and Terms or create new ones. To create a new one, select "Create New" at the top right on the term specification lookup page or copy and then modify an existing one. See example Term Specification Lookup screen below:

If you copy an existing term specification, be sure to give it a new and unique name before you change and save or submit it. Below is a view of the term specification screen showing the types of attributes you can associate with it.

After creating your term specifications (your categories of terms), you can use the Term Lookup screen to add or create new terms. See the example Term Lookup screen below:

Figure 7.1. Term Lookup screen example

If you copy an existing term, be sure to change the name to a new and unique term before you save or submit it. Below is a view of the term specification screen showing the types of attributes you can associate with it.

Figure 7.2. Term specification screen example

Chapter 8. KSB

How to Use the KSB

Introduction

The Kuali Service Bus (KSB) is a lightweight service bus designed to allow developers to quickly develop and deploy services for remote and local consumption. You can deploy services to the bus using Spring or programmatically. Services must be named when they are deployed to the bus. Services are acquired from the bus using their name.

At the heart of the KSB is a service registry. This registry is a listing of all services available for consumption on the bus. The registry provides the bus with the information necessary to achieve load balancing, failover and more.

Bean Based Services

Typically, KSB programming is centered on exposing Spring-configured beans to other calling code using a number of different protocols. Using this paradigm the client developer and the organization can rapidly build and consume services, often a daunting challenge using other buses.

Figure 8.1. Overview of Supported Service Protocols
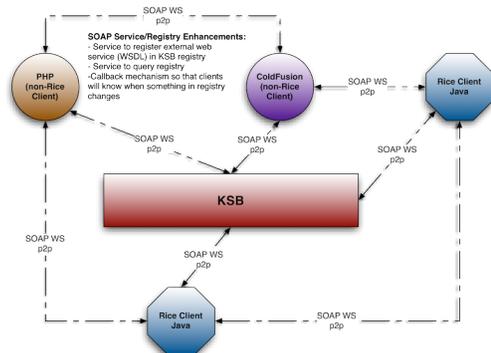
Diagram Notes

This drawing is conceptual and not representative of a true deployment architecture. Essentially, the KSB is a registry with service calling behavior on the client end (Java client). All policies and behaviors (Asynchronous as opposed to Synchronous) are coordinated on the client. The client offers some very attractive messaging features:

- **Synchronization** of message sending with currently running transaction (meaning all messages sent during a transaction are ONLY sent if the transaction is successfully committed)

- **Failover** - If a call to a service comes back with a 404 (or various other network-related errors), it will try to call other services of the same name on the bus. This is for both sync and async calls.

- **Load balancing** - Clients will round-robin call services of the same name on the bus. Proxy instances, however, are bound to single machines if you want to keep a line of communication open to a single machine for long periods of time.

- **Topics and Queues**

- **Persistent messages** - When using message persistence a message cannot be lost. It will be persisted until it is sent.

Details of Supported Service Protocols

Java Rice Client

As Consumer

If configured for the KSB, a Java Rice Client can invoke any service in the KSB Registry using these protocols:

1. Synchronously

   - SOAP WS p2p using KSB Spring configuration

   - Java call if it is within the same JVM

   - Spring HTTP Remoting

2. Asynchronously

   - Messaging Queues – As a Consumer, a Java Rice Client can invoke a one-shot deal for calling a KSB-registered service asynchronously

   - Java, SOAP, Spring HTTP Remoting

   - Messaging Topics - As a Consumer listening to a topic, the Java Rice Client will receive a broadcast message

As Producer

You can register Spring-defined services in the KSB Registry through the KSB Configurer. Consumers can call these services as described in other sections.

Any Java Client

As Consumer

A **Java Client**, regardless of whether or not it's a Rice Client configured for the KSB, can invoke any web service:

1. As a SOAP WS p2p using a straight-up WS call through CXF, Axis, etc. If the external web service is not registered on the KSB, the Java client must discover the service on its own.

2. Through Java if they are within the same JVM

3. Through Spring HTTP Remoting; you must know the endpoint URL of the service.

As Producer

1. Currently, you can't leverage the KSB and its registry for exposing any of its services. It is possible to bring up the registry and register services without the rest of the KSB.

2. A Java Client can expose its web services directly using XFire (CXF), Axis, etc.

3. You can bring up only the registry for discovery. However, the registry can't be a 'service;' it can only be a piece of code talking to a database.

Non-Java/Non-Rice Client

As Consumer

A **non-Java/non-Rice Client** that knows nothing about the KSB or its registry can only invoke web services synchronously using:

• SOAP WS p2p using straight-up WS call through native language-specific WS libs

• Discovery cannot be handled by leveraging the KSB Registry at this time.

As Producer

1. Currently cannot register services on KSB in registry

2. Can still produce services, but they can't be called leveraging the KSB; clients need to discover and invoke the services directly (on their own).

KSB Registry as a Service

As of the 2.0 version of Rice, the ServiceRegistry is now itself a service. In order to bring the registry online for the client application, the application needs to configure a URL similar to the following:

```
<param name="rice.ksb.registry.serviceUrl">http://localhost:8080/kr-dev/remoting/serviceRegistrySoap</param>
```

Currently, this connector is only configured to understand a SOAP interface to the service registry which is secured by digital signatures. This is the only type of interface to the registry that the standalone server currently publishes. Additionally, only a single URL to the registry can be configured at the current time. If someone wants to do load balancing amongst potential registry endpoints, then a hardware or software load balancer could be configured to do this.

Configuring the KSB Client in Spring

Overview

The Kuali Service Bus (KSB) is installed as a Kuali Rice (Rice) Module using Spring. Here is an example XML snippet showing how to configure Rice and KSB using Spring:

```
<beans>
  ...
  <bean id="coreConfigurer" class="org.kuali.rice.core.impl.config.module.CoreConfigurer">
    <property name="dataSource" ref="riceDataSource${connection.pool.impl}" />
    <property name="nonTransactionalDataSource" ref="riceNonTransactionalDataSource" />
    <property name="transactionManager" ref="transactionManager${connection.pool.impl}" />
    <property name="userTransaction" ref="jtaUserTransaction" />
  </bean>

  <bean id="ksbConfigurer" class="org.kuali.rice.ksb.messaging.config.KSBConfigurer"/>
```

```
</beans>
```

Spring Property Configuration

The *KSBTestHarnessSpring.xml* located in the project folder under /ksb/src/test/resources/ is a good starting place to explore KSB configuration in depth. The first thing the file does is use a PropertyPlaceholderConfigurer to bring tokens into the Spring file for runtime configuration. The source of the tokens is the xml file: ksb-test-config.xml located in the /ksb/src/test/resources/META-INF directory.

```
<bean id="config" class="org.kuali.rice.core.config.spring.ConfigFactoryBean">
    <property name="configLocations">
        <list>
            <value>classpath:META-INF/ksb-test-config.xml</value>
        </list>
    </property>
</bean>


<bean class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
    <property name="staticMethod"
 value="org.kuali.rice.core.impl.config.property.ConfigInitializer.initialize"/>
    <property name="arguments">
        <list>
            <ref bean="config"/>
        </list>
    </property>
</bean>


<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="properties" value="#{config.getProperties()}" />
</bean>
```

Note

- Properties are passed into the Rice configurer directly. These could be props loaded from Spring and injected into the bean directly.

- You could use the Rice configuration subsystem for configuration.

- A JTA TransactionManager and UserTransaction are also being injected into the CoreConfigurer.

As mentioned above, this allows tokens to be used in the Spring file. If you are not familiar with tokens, they look like this in the Spring file: ${datasource.pool.maxSize}

Let's take a look at the ksb-test-config.xml file:

```
<config>
    <param name="config.location">classpath:META-INF/common-derby-connection-config.xml</param>
    <param name="config.location">classpath:META-INF/common-config-test-locations.xml</param>
    <param name="client1.location">/var/lib/jenkins/workspace/rice-2.1.7-site-deploy/src/test/clients/
TestClient1</param>
    <param name="client2.location">/var/lib/jenkins/workspace/rice-2.1.7-site-deploy/src/test/clients/
TestClient2</param>
    <param name="ksb.client1.port">9913</param>
    <param name="ksb.client2.port">9914</param>
    <param name="ksb.testharness.port">9915</param>
    <param name="threadPool.size">1</param>
    <param name="threadPool.fetchFrequency">3000</param>
    <param name="bus.refresh.rate">3000</param>
    <param name="bam.enabled">true</param>
    <param name="transaction.timeout">3600</param>
    <param name="keystore.alias">rice<param>
```

```
    <param name="keystore.password">keystorepass</param>
    <param name="keystore.file">/var/lib/jenkins/workspace/rice-2.1.7-site-deploy/src/test/resources/keystore/
ricekeystore</param>
    <param name="keystore.location">/var/lib/jenkins/workspace/rice-2.1.7-site-deploy/src/test/resources/
keystore/ricekeystore</param>
    <param name="use.clearDatabaseLifecycle">true</param>
    <param name="use.sqlDataLoaderLifecycle">true</param>
    <!-- bus messaging props -->
    <param name="message.delivery">synchronous</param>
    <param name="message.persistence">true</param>
    <param name="useQuartzDatabase">false</param>
    <param name="config.location">${additional.config.locations}</param>
    <param name="config.location">${alt.config.location}</param>
</config>
```

This is an XML file for configuring key value pairs. When used in conjunction with Spring tokenization and the PropertyPlaceHolderConfigurer bean, the parameter name must be equal to the key value in the Spring file so that the properties propagate successfully.

Spring JTA Configuration

When doing persistent messaging it is best practice to use JTA as your transaction manager. This ensures that the messages you are sending are synchronized with the current executed transaction in your application. It also allows message persistence to be put in a different database than the application's logic if needed. Currently, *KSBTestHarnessSpring.xml* uses JOTM to configure JTA without an application server. Bitronix is another JTA product that could be used in Rice and you could consider using it instead of JOTM. Below is the bean definition for JOTM that you can find in Spring:

```
<bean id="jotm" class="org.springframework.transaction.jta.JotmFactoryBean">
    <property name="defaultTimeout" value="${transaction.timeout}"/>

</bean>
<bean id="dataSource" class="org.kuali.rice.database.XAPoolDataSource">
    <property name="transactionManager" ref="jotm" />
    <property name="driverClassName" value="${datasource.driver.name}" />
    <property name="url" value="${datasource.url}" />
    <property name="maxSize" value="${datasource.pool.maxSize}" />
    <property name="minSize" value="${datasource.pool.minSize}" />
    <property name="maxWait" value="${datasource.pool.maxWait}" />
    <property name="validationQuery" value="${datasource.pool.validationQuery}" />
    <property name="username" value="${datasource.username}" />
    <property name="password" value="${datasource.password}" />

</bean>
```

Bittronix's configuration is similar. Datasources for both are set up in org.kuali.rice.core.RiceDataSourceSpringBeans.xml. If using JOTM, use the Rice XAPoolDataSource class as your data source because it addresses some bugs in the StandardXAPoolDataSource, which extends from this class.

Put JTA and the Rice Config object in the CoreConfigurer

Next, you must inject the JOTM into the RiceConfigurer:

```
<bean id="rice" class="org.kuali.rice.core.impl.config.module.CoreConfigurer">
    <property name="dataSource" ref="dataSource" />
    <property name="transactionManager" ref="jotm" />
    <property name="userTransaction" ref="jotm" />
<...more.../>
```

Configuring JTA from an appserver is no different, except the TransactionManager and UserTransaction are going to be fetched using a JNDI FactoryBean from Spring.

Note

You set the serviceNamespace property in the example above by injecting the name into the RiceConfigurer. You can do this instead of setting the property in the configuration system.

Configuring KSB without JTA

You can configure KSB by injecting a PlatformTransactionManager into the KSBConfigurer.

- This eliminates the need for JTA. Behind the scenes, KSB uses Apache's OJB as its Object Relational Mapping.

- Before you can use PlatformTransactionManager, you must have a client application set up the OJB so that KSB can use it.

This is a good option if you are an OJB shop and you want to continue using your current setup without introducing JTA into your stack. Normally, when a JTA transaction is found, the message is not sent until the transaction commits. In this case, the message is sent immediately.

Let's take a look at the *KSBTestHarnessNoJtaSpring.xml* file. Instead of JTA, the following transaction and DataSource configuration is declared:

```
<bean id="ojbConfigurer" class="org.springmodules.orm.ojb.support.LocalOjbConfigurer" />


<bean id="transactionManager" class="org.springmodules.orm.ojb.PersistenceBrokerTransactionManager">
    <property name="jcdAlias" value="dataSource" />
</bean>


<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName">
        <value>${datasource.driver.name}</value>
    </property>
    <property name="url">
        <value>${datasource.url}</value>
    </property>
    <property name="username">
        <value>${datasource.username}</value>
    </property>
    <property name="password">
        <value>${datasource.password}</value>
    </property>
</bean>
```

The RiceNoJtaOJB.properties file needs to include the Rice connection factory property value:

```
ConnectionFactoryClass=org.kuali.rice.core.framework.persistence.ojb.RiceDataSourceConnectionFactory
```

Often, the DataSource is pulled from JNDI using a Spring FactoryBean. Next, we inject the DataSource and transactionManager (now a Spring PlatformTransactionManager).

```
<bean id="rice" class="org.kuali.rice.core.impl.config.module.CoreConfigurer">
    <property name="dataSource" ref="dataSource" />
    <property name="nonTransactionalDataSource" ref="dataSource" />
    ...
</bean
```

```
<bean id="ksbConfigurer" class="org.kuali.rice.ksb.messaging.config.KSBConfigurer">
    <property name="platformTransactionManager" ref="transactionManager" />
    <... more .../>
</bean>
```

Notice that the transactionManager is injected into the KSBConfigurer directly. This is because only KSB, and not Rice, supports this type of configuration. The DataSource is injected normally. When doing this, the OJB setup is entirely in the hands of the client application. That doesn't mean anything more than providing an OJB.properties object at the root of the classpath so OJB can load itself. KSB will automatically register its mappings with OJB, so they don't need to be included in the repository.xml file.

web.xml Configuration

To allow external bus clients to invoke services on the bus-connected node, you must configure the KSBDispatcherServlet in the web applications web.xml file. For example:

```
<servlet>
    <servlet-name>remoting</servlet-name>
    <servlet-class>org.kuali.rice.ksb.messaging.servlet.KSBDispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>
    <servlet-name>remoting</servlet-name>
    <url-pattern>/remoting/*</url-pattern>
</servlet-mapping>
```

This allows bus-exposed services to be accessed at a URL like **http://yourlocalip:8080/myapp/remoting/ [KSB:service name]**. Notice how this URL corresponds to the configured **serviceServletUrl** property on the KSBConfigurer.

Configuration Parameters

The service bus leverages the Rice configuration system for its configuration. Here is a comprehensive set of configuration parameters that you can use to configure the Kuali Service Bus:

Table 8.1. KSB Configuration Parameters

Parameter	Required	Default Value
bam.enabled	Whether Business Action Messaging is enabled	false
bus.refresh.rate	How often the service bus will update the services it has deployed in minutes.	60
dev.mode	no	false
message.persistence	no	true
message.delivery	no	asynch
message.off	no	false
ksb.mode	The mode that KSB will run in; choices are "local", "embedded", or "remote".	LOCAL
ksb.url	The base URL of KSB services and pages.	${application.url}/ksb
RouteQueue.maxRetryAttempts	no	5
RouteQueue.timeIncrement	no	5000
Routing.ImmediateExceptionRouting	no	false
RouteQueue.maxRetryAttemptsOverride	no	None

Parameter	Required	Default Value
rice.ksb.batch.mode	A service bus mode suitable for running batch jobs; it, like the KSB dev mode, runs only local services.	false
rice.ksb.struts.config.files	The struts-config.xml configuration file that the KSB portion of the Rice application will use.	/ksb/WEB-INF/struts-config.xml
rice.ksb.web.forceEnable	no	false
threadPool.size	The size of the KSB thread pool.	5
useQuartzDatabase	no	true
ksb.org.quartz.*	no	None
rice.ksb.config.allowSelfSignedSSL	no	false

dev.mode

Indicates whether this node should export and consume services from the entire service bus. If set to true, then the machine will not register its services in the global service registry. Instead, it can only consume services that it has available locally. In addition to this, other nodes on the service bus will not be able to "see" this node and will therefore not forward any messages to it.

message.persistence

If *true*, then messages will be persisted to the datastore. Otherwise, they will only be stored in memory. If message persistence is not turned on and the server is shutdown while there are still messages that need to be sent, those messages will be lost. For a production environment, it is recommended that you set message.persistence to *true*.

message.delivery

Can be set to either *synchronous* or *asynchronous*. If this is set to synchronous, then messages that are sent in an asynchronous fashion using the KSB API will instead be sent synchronously. This is useful in certain development and unit testing scenarios. For a production environment, it is recommended that you set message delivery to *asynchronous*.

Note

It is strongly recommended that you set **message.delivery** to *asynchronous* for all cases except for when implementing automated tests or short-lived programs that interact with the service bus.

message.off

If set to true, then asynchronous messages will not be sent. In the case that message persistence is turned on, they will be persisted in the message store and can even be picked up later using the Message Fetcher. However, if message persistence is turned off, these messages will be lost. This can be useful in certain debugging or testing scenarios.

RouteQueue.maxRetryAttempts

Sets the default number of retries that will be executed if a message fails to be sent. You can also customize this retry count for a specific service (see Exposing Services on the Bus).

RouteQueue.timeIncrement

Sets the default time increment between retry attempts. As with RouteQueue.maxRetryAttempts, you can also configure this at the service level.

Routing.ImmediateExceptionRouting

If set to *true*, then messages that fail to be sent will not be retried. Instead, their MessageExceptionHandler will be invoked immediately.

RouteQueue.maxRetryAttemptsOverride

If set with a number, it will temporarily set the retry attempts for ALL services going into exception routing. You can set the number arbitrarily high to prevent all messages in a node from making it to exception routing if they are having trouble. The message.off param produces the same result.

useQuartzDatabase

When using the embedded Quartz scheduler started by the KSB, indicates whether that Quartz scheduler should store its entries in the database. If this is true, then the appropriate Quartz properties should be set as well. (See ksb.org.quartz.* below).

ksb.org.quartz.*

Can be used to pass Quartz properties to the embedded Quartz scheduler. See the configuration documentation on the [Quartz site](#). Essentially, any property prefixed with *ksb.org.quartz.* will have the "ksb." portion stripped and will be sent as configuration parameters to the embedded Quartz scheduler.

rice.ksb.config.allowSelfSignedSSL

If *true*, then the bus will allow communication using the **https** protocol between machines with self-signed certificates. By default, this is not permitted and if attempted you will receive an error message like this:

Note

It is best practice to only set this to 'true' in non-production environments!

rice.ksb.web.forceEnable

publish the KSB user interface components (such as the Message Queue, Thread Pool, Service Registry screens) even when the ksb.mode is not set to *local*.

KSBConfigurer Properties

In addition to the configuration parameters that you can specify using the Rice configuration system, the KSBConfigurer bean itself has some properties that can be injected in order to configure it:

exceptionMessagingScheduler

By default, KSB uses an embedded Quartz scheduler for scheduling the retry of messages that fail to be sent. If desired, a Quartz scheduler can instead be injected into the KSBConfigurer and it will use that scheduler instead. See Quartz Scheduling for more detail.

messageDataSource

Specifies the javax.sql.DataSource to use for storing the asynchronous message queue. If not specified, this defaults to the DataSource injected into the RiceConfigurer.

If this DataSource is injected, then the registryDataSource must also be injected and vice-versa.

nonTransactionalMessageDataSource

Specifies the javax.sql.DataSource to use that matches the messageDataSource property. This datasource instance must not be transactional. If not specified, this defaults to the nonTransactionalDataSource injected into the RiceConfigurer.

registryDataSource

Specifies the javax.sql.DataSource to use for reading and writing from the Service Registry. If not specified, this defaults to the DataSource injected into the RiceConfigurer.

If this DataSource is injected, then the messageDataSource must also be injected and vice-versa.

services

Specifies a list of Java service definitions relating to SOAP to use as part of messaging.

KSB Configurer

The application needs to do one more thing to begin publishing services to the bus: Configure the KSBConfigurer object. This can be done using Spring or programmatically. We'll use Spring because it's the easiest way to get things configured:

```
<bean id="jotm" class="org.springframework.transaction.jta.JotmFactoryBean">
    <property name="defaultTimeout" value="${transaction.timeout}"/>
</bean>




<bean id="dataSource" class=" org.kuali.rice.core.database.XAPoolDataSource ">
    <property name="transactionManager" ref="jotm"/>
    <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
    <property name="maxSize" value="25"/>
    <property name="minSize" value="2"/>
    <property name="maxWait" value="5000"/>
    <property name="validationQuery" value="select 1 from dual"/>
    <property name="url" value="jdbc:oracle:thin:@LOCALHOST:1521:XE"/>
    <property name="username" value="myapp"/>
    <property name="password" value="password"/>
</bean>


<bean id="nonTransactionalDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
        <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
        <property name="url" value="jdbc:oracle:thin:@LOCALHOST:1521:XE"/>
        <property name="maxActive" value="50"/>
        <property name="minIdle" value="7"/>
        <property name="initialSize" value="7"/>
        <property name="validationQuery" value="select 1 from dual"/>
        <property name="username" value="myapp"/>
        <property name="password" value="password"/>
        <property name="accessToUnderlyingConnectionAllowed" value="true"/>
</bean>

<bean id="coreConfigurer" class="org.kuali.rice.core.impl.config.module.CoreConfigurer">
    <property name="dataSource" ref="datasource" />
    <property name="nonTransactionalDataSource" ref="nonTransactionalDataSource" />
    <property name="transactionManager" ref="jotm" />
    <property name="userTransaction" ref="jotm" />
</bean>

<bean id="ksbConfigurer" class="org.kuali.rice.ksb.messaging.config.KSBConfigurer"/>
```

The application is now ready to deploy services to the bus. Let's take a quick look at the Spring file above and what's going on there: The following configures JOTM, which is currently required to run KSB.

```
<bean id="jotm" class="org.springframework.transaction.jta.JotmFactoryBean" />
```

Next, we configure the XAPoolDataSource and the non transactional BasicDataSource. This is pretty much standard data source configuration stuff. The XAPoolDataSource is configured through Spring and not JNDI so it can take advantage of JTOM. Servlet containers, which don't support JTA, require this configuration step so the datasource will use JTA.

```
<bean id="dataSource" class=" org.kuali.rice.core.database.XAPoolDataSource ">
    <property name="transactionManager" ref="jotm"/>
    <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
    <property name="url" value="jdbc:oracle:thin:@LOCALHOST:1521:XE"/>
    <property name="maxSize" value="25"/>
    <property name="minSize" value="2"/>
    <property name="maxWait" value="5000"/>
    <property name="validationQuery" value="select 1 from dual"/>
    <property name="username" value="myapp"/>
    <property name="password" value="password"/>
</bean>

<bean id="nonTransactionalDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
    <property name="url" value="jdbc:oracle:thin:@LOCALHOST:1521:XE"/>
    <property name="maxActive" value="50"/>
    <property name="minIdle" value="7"/>
    <property name="initialSize" value="7"/>
    <property name="validationQuery" value="select 1 from dual"/>
    <property name="username" value="myapp"/>
    <property name="password" value="password"/>
    <property name="accessToUnderlyingConnectionAllowed" value="true"/>
</bean>
```

Next, we configure the bus:

```
<bean id="rice" class="org.kuali.rice.core.config.CoreConfigurer">
    <property name="dataSource" ref="dataSource" />
    <property name="nonTransactionalDataSource" ref="nonTransactionalDataSource" />
    <property name="transactionManager" ref="jotm" />
    <property name="userTransaction" ref="jotm" />
</bean>

<bean id="ksbConfigurer" class="org.kuali.rice.ksb.messaging.config.KSBConfigurer">
    <property name="registryDataSource" ref="dataSource" />
    <property name="bamDataSource" ref="dataSource" />
    <property name="messageDataSource" ref="dataSource" />
    <property name="nonTransactionalMessageDataSource" ref="nonTransactionalDataSource" />
</bean>
```

We are injecting JOTM, and the datasources. The injection of the KSBConfigurer class into the ksbConfigurer property tells this instance of Rice to start the Service Bus. The final necessary step is making sure the configuration parameter 'application.id' is set properly to some value that will identify all services deployed from this node as a member of this node.

At this point, the application is configured to use the bus, both for publishing services and to send messages to services. Usually, applications will publish services on the bus using the KSBConfigurer or the SoapServiceExporter classes. See Acquiring and invoking services for more detail.

Implications of "synchronous" vs. "asynchronous" Message Delivery

As noted in Configuration Parameters, it is possible to configure message delivery to run asynchronously or synchronously. It is imported to understand that asynchronous messing should be used in almost all cases.

Asynchronous messing will result in messages being sent in a separate thread after the original transaction that requested the message to be sent is committed. This is the appropriate behavior in a "fire-and-forget" messaging model. The option to configure message deliver as synchronous was added for two reasons:

1. To allow for the implementation of automated unit tests which could perform various tests without having to right "polling" code to wait for asynchronous messing to complete.

2. For short-lived programs (such as batch programs) which need to send messages. This allows for a guarantee that all messages will be sent prior to the application being terminated.

The second case is the only case where synchronous messaging should be used in a production setting, and even then it should be used with care. Synchronous message processing in Rice currently has the following major differences from asynchronous messaging that need to be understood:

1. Order of Execution

2. Exception Handling

Order of Execution

In asynchronous messaging, messages are queued up until the end of the transaction, and then sent after the transaction is committed (technically, they are sent **when** the transaction is committed).

In synchronous messaging, messages are processed **immediately** when they are "sent". This results in a different ordering of execution when using these two different messaging models.

Exception Handling

In asynchronous messaging, whenever there is a failure processing a message, an exception handler is invoked. Recovery from such failures can include resending the message multiple times, or recording and handling the error in some other way. Since all of this is happening after the original transaction was committed, it does not affect the original processing which invoked the sending of the message.

With synchronous messaging, since the message processing is invoked immediately and the calling code blocks until the processing is complete, any errors raised during messaging will be thrown back up to the calling code. This means that if you are writing something like a batch program which relies on synchronous messaging, you must be aware of this and add code to handle any errors if you want to deal with them gracefully.

Another implication of this is that message exception handlers will **not** be invoked in this case. Additionally, because an exception is being thrown, this will typically trigger a rollback in any transaction that the calling code is running. So transactional issues must be dealt with as well. For example, if the failure of a single message shouldn't cause the sending of all messages in a batch job to fail, then each message will need to be sent in it's own transaction, and errors handled appropriately.

Configuring Quartz for KSB

Quartz Scheduling

The Kuali Service Bus (KSB) uses Quartz to schedule delayed tasks, including retry attempts for messages that cannot be sent the first time. By default, KSB uses an embedded quartz scheduler that can be configured by passing parameters starting with "*ksb.org.quartz.*" into the Rice configuration.

If the application is already running a quartz scheduler, you can inject a custom quartz scheduler using code like this:

```
<bean class="org.kuali.rice.ksb.messaging.config.KSBConfigurer">
    ...
    <property name="exceptionMessagingScheduler">
        <bean class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
            ...
        </bean>
    </property>
</bean>
```

When you do this, KSB will not create an embedded scheduler but will instead use the one provided.

Acquiring and Invoking Services Deployed on KSB

Service invocation overview

1. Acquired and called directly

   - Automatic Failover

   - No Persistence

   - Direct call - Request/Response

2. Acquired and called through the MessageHelper

   - Automatic Failover

   - Message Persistence

   - KSB Exception Messaging

   - Callback Mechanisms

In the examples below, notice that the **client code is unaware of the protocol with which the underlying service is deployed**. Given a connector for a given protocol and a compatible service definition, you could move a service to different protocols as access needs change without affecting dependent client code.

Acquiring and invoking a service directly

The easiest way to call a service is to grab it and invoke it directly. This uses a direct request/response pattern and what you see is what you get. You will wait for the processing the call takes on the other side plus the cost of the remote connection time. Any exceptions thrown will come across the wire in a protocol-acceptable way.

This code acquires a SOAP-based service and calls it:

```
QName serviceName = new QName("testNameSpace", "soap-repeatTopic");

SOAPService soapService = (SOAPService) GlobalResourceLoader.getService(serviceName);
soapService.doTheThing("hello");
```

The SOAPService interface needs to be in the client classpath and bindable to the WSDL. The easiest way to achieve this in Java is to create a bean that is exported as a SOAP service. This is the server-side service declaration in a Spring file:

```
<bean id="ksbConfigurer" class="org.kuali.rice.ksb.messaging.config.KSBConfigurer">
  ...
  <property name="services">
    <list>
        <bean class="org.kuali.rice.ksb.api.bus.support.SoapServiceDefinition">
            <property name="service">
              <ref bean="soapService" />
            </property>
            <property name="localServiceName" value="soap-repeatTopic" />
            <property name="serviceNameSpaceURI" value="testNameSpace" />
            <property name="priority" value="3" />
            <property name="queue" value="false" />
            <property name="retryAttempts" value="1" />
        </bean>
        ...
    </list>
  </property>
</bean>
```

This declaration exposes the bean soapService on the bus as a SOAP available service. The Web Service Definition Language is available at the serviceServletUrl + serviceNameSpaceURI + localServiceName + ?wsdl.

This next code snippet acquires and calls a Java base service:

```
EchoService echoService = (EchoService)GlobalResourceLoader.getService(new QName("TestCl1", "echoService"));
String echoValue = "echoValue";
String result = echoService.echo(echoValue);
```

Again, the interface is all that is required to make the call. This is the server-side service declaration that deploys a bean using Spring's HttpInvoker as the underlying transport:

```
<bean id="ksbConfigurer" class="org.kuali.rice.ksb.messaging.config.KSBConfigurer">
  ...
  <property name="services">
    <list>
        <bean class="org.kuali.rice.ksb.api.bus.support.SoapServiceDefinition">
            <property name="service" ref="echoService" />
            <property name="serviceInterface"
 value="org.kuali.rice.ksb.messaging.remotedservices.EchoService" />
            <property name="localServiceName" value="soap-echoService" />
            <property name="busSecurity" value="false"></property>
        </bean>
        ...
    </list>
  </property>
</bean>
```

Below is a description of each property on the ServiceDefinition (JavaServiceDefinition and SOAPServiceDefinition):

Table 8.2. Properties of the ServiceDefinition

property	required	default	description
busSecurity	no	yes (JavaServiceDefinition), no (SOAPServiceDefinition)	For Java-based services, message is digitally signed before calling the service and verified at the node hosting the service. For SOAP services, WSS4J is used to digitally sign the SOAP request/response in accordance with the WS Security specification. More info on Bus Security here.
localServiceName	yes	none	The local name of the QName that makes up the complete service name.

property	required	default	description
messageExceptionHandler	no	DefaultMessageExceptionHandler	Name of the MessageExceptionHandler that is called when a service call fails. This is called after the retryAttempts or millisToLive policy of the service or Node has been met.
millisToLive	no	none	Used instead of retryAttempts. Only considered in case of error when invoking service. Defines how long the message should continue to be tried before being put into KSB Exception Messaging.
priority	no	5	Only applies when asynchronous messaging is enabled. The lower the priority is, the sooner the message will be executed. For example, if 100 *priority 10* messages are waiting for invocation and a *priority 5* message is sent, the *priority 5* message will be executed first.
queue	no	true	If *true*, the service will behave like a queue in that there is only one real service call when a message is sent. If *false*, the service will behave like a topic. All beans bound to the service name will be sent a message when a message is sent to the service. Use queues for operations you only want to happen once (for example, to route a document). Use topics for notifications across a cluster (for example, to invalidate cache entry).
retryAttempts	no	7	Determines the number of times a service can be invoked before being put into KSB Exception Messaging (the error state)
service	yes	none	The bean to be exposed for invocation on the bus
serviceEndPoint	no	serviceServletUrl + serviceName	This can be explicitly set to create an alternate service end point, different from the one the bus automatically creates.
serviceName	yes	serviceNameSpaceURI + localServiceName	If *localServiceName* and *serviceNameSpaceURI* are omitted, the QName of the service. This can be used instead of the *localServiceName* and *serviceNameSpaceURI* convenience methods.
serviceNameSpaceURI	no	messageEntity property or message.entity config param is used	The namespaceURI of the QName that makes up the complete service name. If set to "" (blank string) the property is NOT included in the construction of the QName representing the service and the service name will just be the localServiceName with no namespace.

Acquiring and invoking a service using messaging

To make a call to a service through messaging, acquire the service by its name using the MessageHelper:

```
QName serviceName = new QName("testAppsSharedQueue", "sharedQueue");

KEWSampleJavaService testJavaAsyncService = (KEWSampleJavaService)
 KsbApiServiceLocator.getMessageHelper().getServiceAsynchronously(serviceName);
```

At this point, the testJavaAsyncService can be called like a normal JavaBean:

```
testJavaAsyncService.invoke(new ClientAppServiceSharedPayloadObj("message content", false));
```

Because this is a queue, a single message is sent to one of the beans bound to the service name *new QName("testAppsSharedQueue", "sharedQueue")*. That 'message' is the call 'invoke' and it takes a

ClientAppServiceSharedPayloadObj. Typically, messaging is done asynchronously. Messages are sent when the currently running JTA transaction is committed - that is, the messaging layer automatically synchronizes with the current transaction. So, using JTA, even though the above is coded in line with code, invocation is normally delayed until the transaction surrounding the logic at runtime is committed.

When not using JTA, the message is sent asynchronously (by a different thread of execution), but it's sent ASAP.

To review, the requirements to use a service that is exposed to the bus on a different machine are:

1. The service name

2. The interface to which to cast the returned service proxy object

3. The ExceptionMessageHandler required by the service in case invocation fails

Note

Typically, service providers give clients a JAR with this content or organizations maintain a JAR with this content.

To complete the example: Below is the Spring configuration used to expose this service to the bus. This is taken from the file *TestClient1SpringBeans.xml*:

```
<!-- bean declaration -->
<bean id="sharedQueue" class=" org.kuali.rice.ksb.testclient1.ClientApp1SharedQueue" />

<bean id="ksbConfigurer" class="org.kuali.rice.ksb.messaging.config.KSBConfigurer">
  ...
  <property name="services">
    <list>
        <bean class=" org.kuali.rice.ksb.messaging.JavaServiceDefinition">
            <property name="service" ref="sharedQueue" />
            <property name="localServiceName" value="sharedQueue" />
            <property name="serviceNameSpaceURI" value="testAppsSharedQueue" />
        </bean>
        <... more .../>
    </list>
  </property>
</bean>
```

This is located in the Spring file of the application exposing the service (in other words, the location in which the actual invocation will occur). The client does not need a Spring configuration to invoke the service.

There are two messaging call paradigms, called *Topics* and *Queues*. When any number of services is declared a Topic, then those services are invoked at least once or multiple times. If any number of services is declared a Queue, then one and only one service name will be invoked.

Getting responses from service calls made with messaging

You can use Callback objects to get responses from service calls made using messaging. Acquiring a service for use with a Callback:

```
QName serviceName = new QName("TestCl1", "testXmlAsyncService");
SimpleCallback callback = new SimpleCallback();
KSBXMLService testXmlAsyncService = (KSBXMLService)
 KsbApiServiceLocator.getMessageHelper().getServiceAsynchronously(serviceName, callback);
```

```
testXmlAsyncService.invoke("message content");
```

When the service is invoked asynchronously, the AsynchronousCallback object's (the SimpleCallback class above) callback method is called.

When message persistence is turned on, this object is serialized with any method call made through the messaging API. Take into consideration that this object (and the result of a method call) may survive machine restart and therefore it's recommended that you NOT depend on certain transient in-memory resources.

Failover

Service call failover

Failover works the same whether making direct service calls or using messaging.

Services exported to the bus have automatic failover from the client's perspective. For example, if service A is deployed on machines 1 and 2 and a client happens to get a proxy that points to machine 1 but machine 1 crashes, the KSB will automatically detect that the exception is a result of some network issue and direct the call to machine 2. KSB then removes machine 1 from the registry so new clients to the bus don't try to acquire the service. When machine 1 returns to the network it will register itself with the service registry and therefore the bus.

When a message calls a service, the failover rules determine which service KSB assigns (topic or queue) to the message.

Failover with queues

Because queues require only one call between all beans bound to the queue, if a single call to a queue fails, failover to the next bean occurs. If successful, the call is done. If it is not successful, it continues until a suitable bean is found. If none is found, the message is marked for retry later. Eventually, the message either goes to KSB exception messaging or successfully completes.

Failover with topics

If a machine in a topic is unavailable, a failed call to that machine will continue to be retried until that call is successful or that call goes into KSB exception messaging.

KSB Exception Messaging

Exception Messaging is the set of services and configuration options that handle messages that cannot be delivered successfully. Exception Messaging is primarily used by configuring your service using the properties outlined in KSB Module Configuration. When services are configured to use message persistence and there is a problem invoking a service, the persisted message or service call is relied upon to make another call to that service until the call is either:

1. Successful

2. Certain configuration policies have been met and the message goes into the Exception state

The Exception state means that KSB can't doing anything more with this message. The message will not invoke properly. That generally means that some sort of technical intervention is required by both the consumer and the provider of the service to determine what the problem is.

All Exception behavior is configurable at the service level by setting the name of the class to be used as MessageExceptionHandler. This class determines what to do when a client of a service cannot invoke a message. The DefaultMessageExceptionHandler is enough to meet most requirements.

When a message is put into the Exception state, KSB puts it back into the message store and marks it with a status of 'E'. At that point, it is up to the person responsible for monitoring this node on the bus to determine what to do with the message.

Because the node exposing the service configures the MessageExceptionHandler, any clients depending on the service need that MessageExceptionHandler and any dependent code and configuration.

KSB Messaging Paradigms

KSB supports two types of messaging paradigms; Queues and Topics, and the differences are explained below. These are very similar to JMS messaging concepts. An open source solution was not used for KSB messaging because an open source JMS provider wasn't found that provided JTA synchronization, discovery, failover, and load balancing. Many claim such features, but when put to the test in real world scenarios (i.e., machines going down and coming back up, databases failing, network connectivity issues); none managed to reliably deliver messages.

The advantage here is that we can apply these messaging concepts to any support protocol with which we can communicate.

Queues

When any number of services is bound to a queue and a method is invoked, one and only one service gets the invocation.

Topics

When any number of services is bound to a topic and a method is invoked, all services are invoked AT LEAST once or multiple times.

Message Fetcher

org.kuali.rice.ksb.messaging.MessageFetcher is a Runnable that needs to be configured by the client application to retrieve stored messages from the database that weren't processed when the node went down. This can happen for many reasons. The machine can be under load and just crash.

When message persistence is enabled, a service that fails or throws an Exception stores preprocessed messages in the database until they can be resent. This makes certain that a crash or emergency restart of your machine will not result in message loss.

The KSB does not automatically fetch all these messages and attempt to invoke them when it starts, because often the KSB is started when the services the messages are bound for are not yet started. For now, you need to decide when to call the run method on the MessageFetcher. Because it's a Runnable, you could also put the MessageFetcher in the KSBThreadPool that is available on the KSBServiceLocator. You could wrap it in a TimerTask, etc. All that is required is this:

```
new MessageFetcher((Integer) null).run()
```

Unfortunately, the cast to Integer is required. The MessageFetcher also has a constructor that takes a long variable as a parameter. This can be used to pull any message in the message store and put it in memory for invocation. *Integer* is a fetch size; *null* means all.

Load Balancing

Load balancing between service calls is automatic. If there are multiple nodes that expose services of the same name, clients will randomly acquire proxies to each endpoint bound to that name.

Object Remoting

As of Rice 2.0, Object remoting support has been removed.

Publishing Services to KSB

You can publish Services on the service bus either by configuring them directly in the application's KSBConfigurer module definition, or by using the PropertyConditionalServiceBusExporter bean. In either case, a ServiceDefinition is provided that specifies various bus settings and the target Spring bean.

KSBConfigurer

A service can be exposed by explicitly registering it with the KSBConfigurer module, services property:

```
<bean class="org.kuali.rice.ksb.messaging.config.KSBConfigurer">
    <property name="serviceServletUrl" value="${base url}/MYAPP/remoting/" />
    ...
    <property name="services">
      <list>
          <bean class="org.kuali.rice.ksb.api.bus.support.SoapServiceDefinition">
              <property name="service">
                  <ref bean="mySoapService" />
              </property>
              <property name="serviceInterface"><value>org.myapp.services.MySOAPService</value></property>
              <property name="localServiceName" value="myExposedSoapService" />
          </bean>
          <bean class="org.kuali.rice.ksb.api.bus.support.JavaServiceDefinition">
              <property name="service">
                  <ref bean="myJavaService" />
              </property>
              <property name="serviceInterface">
                  <value>org.myapp.services.MyJavaService</value></property>
              <property name="localServiceName" value="myExposedJavaService" />
          </bean>
```

Service Exporter

You can also publish Services in any context using the ServiceBusExporter (or PropertyConditionalServiceBusExporter) bean. Note that KSBConfigurer must also be defined in your RiceConfigurer.

```
<bean id="myapp.serviceBus"
      class="org.kuali.rice.krad.config.GlobalResourceLoaderServiceFactoryBean">
    <property name="serviceName" value="rice.ksb.serviceBus"/>
</bean>

<bean id="myAppServiceExporter"
      class="org.kuali.rice.ksb.api.bus.support.ServiceBusExporter"
      abstract="true">
    <property name="serviceBus" ref="myapp.serviceBus"/>
</bean>
```

```
<bean id="myJavaService.exporter" parent="myAppServiceExporter">
    <property name="serviceDefinition">
        <bean class="org.kuali.rice.ksb.api.bus.support.JavaServiceDefinition">
            <property name="service">
                <ref bean="myJavaService" />
            </property>
            <property name="serviceInterface">
                <value>org.myapp.services.MyJavaService</value>
            </property>
            <property name="localServiceName" value="myExposedJavaService" />
        </bean>
    </property>
</bean>

<bean id="mySoapService.exporter" parent="myAppServiceExporter">
    <property name="serviceDefinition">
        <bean class="org.kuali.rice.ksb.api.bus.support.SoapServiceDefinition">
            <property name="service">
                <ref bean="mySoapService" />
            </property>
            <property name="serviceInterface">
                <value>org.myapp.services.MySOAPService</value>
            </property>
            <property name="localServiceName" value="myExposedSoapService" />
        </bean>
    </property>

</bean>
```

CallbackServiceExporter

The term "Callback Service" refers to services that client applications write and configure and which are used by various modules of Rice including KIM, KEW, and KRMS. Because of the naming convention on these, they are often referred to as "Type Services". These include:

- KIM

  - RoleTypeService

  - PermissionTypeService

  - GroupTypeService

  - etc.

- KRMS

  - ActionTypeService

  - PropositionTypeService

  - AgendaTypeService

  - etc.

- KEW

  - PeopleFlowTypeService

These are typically called back into from the Rice Standalone Server when needing information for rendering of various components in the server-side user interface. Additionally, in some cases they can also be used to provide custom processing hooks for different components of the various Kuali Rice frameworks.

Version Compatibility for Callback Services

Callback services (like all services in Rice) can be evolved over time and across versions. This means that new functionality might be added to them. Since the Rice Standalone Server interacts with these services remotely, it really needs to know what version of a particular callback service that the client application is running. They also must be published as the appropriate type of service endpoint that the standalone server knows how to talk to (i.e. SOAP instead of Java Serialization). Thankfully, the KSB service registry can store metadata about a service which includes the service version. However, in order to for this to work properly the client application must be sure they publish the service with a version that matches the version of Rice they are using.

In order to make this easier on client applications, a helper has been implemented which can be used for this purpose in Rice.

Callback Service Exporter Helper

There is a helper class which can be used by client applications to export these callback services onto the Kuali Service Bus. The class is `org.kuali.rice.ksb.api.bus.support.CallbackServiceExporter`. This is a class which can be wired up inside of a Spring context in order to publish a callback service to the KSB with the appropriate Rice version. The version of Rice is packaged up into the Rice jars inside of a file called *common-config-defaults.xml* and it uses the version that matches the version of Rice in the POM when the jar was packaged.

Typical configuration might look like the following:

```
<bean id="sampleAppPeopleFlowTypeService.exporter"
      class="org.kuali.rice.ksb.api.bus.support.CallbackServiceExporter"
      p:serviceBus-ref="rice.ksb.serviceBus"
      p:callbackService-ref="sampleAppPeopleFlowTypeService"
      p:serviceNameSpaceURI="http://rice.kuali.org/sample-app"
      p:localServiceName="sampleAppPeopleFlowTypeService"
      p:serviceInterface="org.kuali.rice.kew.framework.peopleflow.PeopleFlowTypeService"/>
```

The javadocs for `CallbackServiceExporter` provide more detail on the options for publishing of callback services.

ServiceDefinition properties

ServiceDefinitions define how the service is published to the KSB. Currently KSB supports three types of services: Java RPC (via serialization over HTTP), SOAP, and JMS.

Basic parameters

All service definitions support these properties:

Table 8.3. ServiceDefinition Properties

Property	Description	Required
Service	The reference to the target service bean	yes
localServiceName	The "local" part of the service name; together with a namespace this forms a qualified name, or QName	yes
serviceNameSpaceURI	The "namespace" part of the service name; together with a local name forms a qualified name, or QName	Not required; if omitted, the **Core.currentContextConfig().getMessageEntity()** is used when exporting the service

Property	Description	Required
serviceEndpoint	URL at which the service can be invoked by a remote call	Not required; defaults to the serviceServletUrl parameter defined in the Rice config
retryAttempts	Number of attempts to retry the service invocation on failure; for services with side-effects you are advised to omit this property	Not required; defaults to 0
millisToLive	Number of milliseconds the call should persist before resulting in failure	Not required; defaults to no limit (-1)
Priority	Priority	Not required; defaults to 5
MessageExceptionHandler	Reference to a MessageExceptionHandler that should be invoked in case of exception	Not required; default implementation handles retries and timeouts
busSecurity	Whether to enable bus security for the service	Not required; defaults to *ON*

ServiceNameSpaceURI/MessageEntity

ServiceNameSpaceURI is the same as the *Message Entity* that composes the qualified name under which the service is exposed. When omitted, this namespace defaults to the message entity configured for Rice (e.g., in the RiceConfigurer), thereby qualifying the local name. Note: Although this implicit qualification occurs during export, you must always specify an explicit message entity when acquiring a resource, for example:

```
GlobalResourceLoader.getService(new QName("MYAPP", "myExposedSoapService"))
```

SOAPServiceDefinition

Table 8.4. SOAPServiceDefinition

Property	Description	Required
serviceInterface	The interface to expose and from which to generate the WSDL	Not required; if omitted the first interface implemented by the class is used

JavaServiceDefinition

Table 8.5. JavaServiceDefinition

Property	Description	Required
serviceInterface	The interface to expose	Not required; if omitted, all application-layer interfaces implemented by the class are exposed
serviceInterfaces	A list of interfaces to expose	Not required; if omitted, all application-layer interfaces implemented by the class are exposed

Publishing Rice services

We show how you can "import" Rice services into the client Spring application context in Configuring KSB Client in Spring. Using this technique, you can also publish Rice services on the KSB:

```
<!-- import a Rice service from the ResourceLoader stack -->
<bean id="myapp.aRiceService" class="org.kuali.rice.krad.config.GlobalResourceLoaderServiceFactoryBean">
    <property name="serviceName" value="aRiceService"/>
</bean>


<!-- if Rice does not publish this service on the bus, one can explicitly publish it -->
<bean id="myAppServiceExporter"
      class="org.kuali.rice.ksb.api.bus.support.ServiceBusExporter"
      abstract="true">
```

```
    <property name="serviceBus" ref="myapp.serviceBus"/>
</bean>

<bean id="myJavaService.exporter" parent="myAppServiceExporter">
    <property name="serviceDefinition">
        <bean class="org.kuali.rice.ksb.api.bus.support.JavaServiceDefinition">
            <property name="service">
                <ref bean="aRiceService" />
            </property>
            <property name="serviceInterface" value="org.kuali.rice...SomeInterface" />
            <property name="localServiceName" value="aPublishedRiceService" />
        </bean>
    </property>
</bean>
```

Warning

Not all Rice services are intended for public use. Do not arbitrarily expose them on the bus

The ResourceLoader Stack

Overview

Rice is composed of a set of modules that provide distinct functionality and expose various services.

- Services in Rice are accessible by the **ResourceLoader**, which can be thought of as analogous to Spring's *BeanFactory* interface. (In fact, Rice modules themselves back ResourceLoaders with Spring bean factories.)

- Services can be acquired by name. (Rice adds several additional concepts, including qualification of service names by namespaces.)

- When the **RiceConfigurer** is instantiated, it constructs a GlobalResourceLoader that is composed of an initial *RootResourceLoader* (which may be provided by the application via the RiceConfigurer), as well as resource loaders supplied by each module:

Figure 8.2. Global Resource Loader
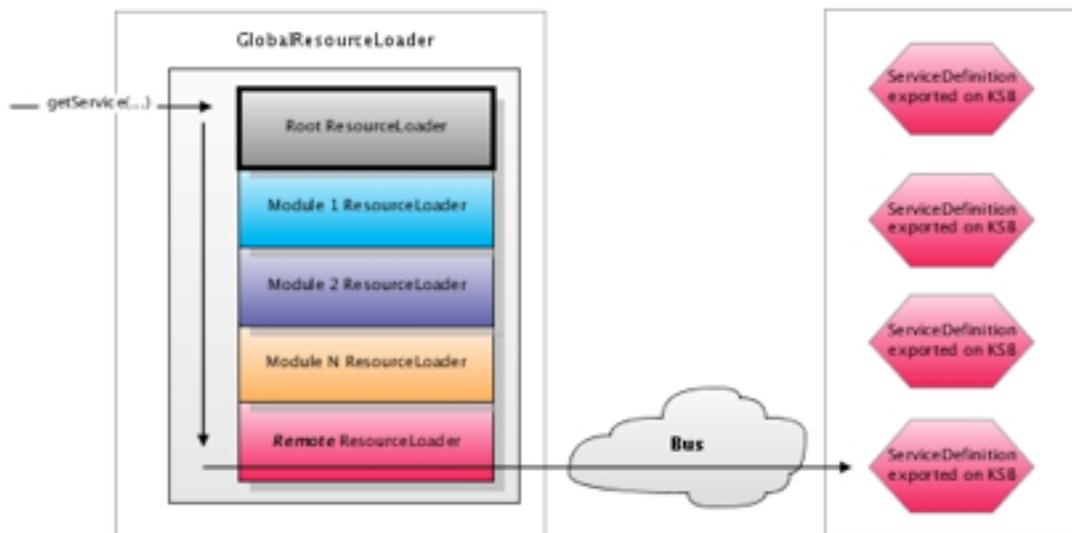
The **GlobalResourceLoader** is the top-level entry point through which all application code should go to obtain services. The **getService** call will iterate through each registered ResourceLoader, looking for the service of the specified name. If the service is found, it is returned, but if it is *not* found, ultimately the call will reach the **RemoteResourceLoader**. The Root ResourceLoader is registered by the KSB module that exposes services that have been registered on the bus.

Accessing and overriding Rice services and beans from Spring

ResourceLoaderFactoryBean

In addition to programmatically acquiring service references, you can also import Rice services into a Spring context with the help of the **GlobalResourceLoaderServiceFactoryBean**:

This bean is *bean-name-aware* and will produce a bean of the same name obtained from Rice's resource loader stack. The bean can then be wired in Spring like any other bean.

Installing an application root resource loader

Applications can install their own root ResourceLoader to override beans defined by Rice. To do so, inject a bean that implements the ResourceLoader interface into the RiceConfigurer rootResourceLoader property. For example:

```
<!-- a Rice bean we want to override in our application -->
<bean id="overriddenRiceBean" class="my.app.package.MyRiceServiceImpl"/>

<!-- supplies services from this Spring context -->
<bean id="appResourceLoader" class="org.kuali.rice.core.impl.resourceloader.SpringBeanFactoryResourceLoader"/>
<bean id="coreConfigurer" class="org.kuali.rice.core.impl.config.module.CoreConfigurer">
    <property name="dataSource" ref="standaloneDataSource" />
    <property name="transactionManager" ref="atomikosTransactionManager" />
    <property name="userTransaction" ref="atomikosUserTransaction" />
    <property name="rootResourceLoader" ref="appResourceLoader"/>
</bean>
```

Warning

Application ResourceLoader and Circular Dependencies

Be careful when mixing registration of an application root resourceloader and lookup of Rice services through the GlobalResourceLoader. If you are using an application resourceloader to override a Rice bean, but one of your application beans requires that bean to be injected during startup, you may create a circular dependency. In this case, you will either have to make sure you are not unintentionally exposing application beans (which may not yet have been fully initialized by Spring) in the application resourceloader, or you will have to arrange for the GRL lookup to occur lazily, after Spring initialization has completed (either programmatically or through a proxy).

Overriding Rice services: Alternate method

A Rice-enabled webapp (including the Rice Standalone distribution) contains a multiple module configurers, typically defined in an xml Spring context file. These load the Rice modules. Each module has its own ResourceLoader, which is typically backed by an XML Spring context file. Overriding and/or setting global beans and/or services (such as data sources and transaction managers) is done as described

above. However, because in each module services can be injected into each other, overriding module services involves overriding the respective module's Spring context file.

The cleanest way to do this is to set the rice.*.addtionalSpringFiles to an accessible spring beans file that overrides one or more spring beans in the existing module's context. Each rice module has a corresponding configuration parameter that can be pointed to a file that will override any existing services.

```
<param name="rice.kew.additionalSpringFiles">classpath:myapp/config/MyAppKewOverrideSpringBeans.xml</param>

<param name="rice.ksb.additionalSpringFiles">classpath:myapp/config/MyAppKsbOverrideSpringBeans.xml</param>

<param name="rice.krms.additionalSpringFiles">classpath:myapp/config/MyAppKrmsOverrideSpringBeans.xml</param>

<param name="rice.kim.additionalSpringFiles">classpath:myapp/config/MyAppKimOverrideSpringBeans.xml</param>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<!-- override of KNS encryption service -->
<beans>


    <!-- override encryption services -->
    <bean id="encryptionService" class="edu.my.school.myapp.service.impl.MyEncryptionServiceImpl" lazy-
init="true">
        <property name="cipherAlgorithm" value="${encryption.cipherAlg}"/>
        <property name="keyAlgorithm" value="${encryption.keyAlg}"/>
        <property name="key" value="${encryption.key}"/>
        <property name="enabled" value="${encryption.busEncryption}"/>
    </bean>

</beans>
```

KSB Security -- STILL NEEDS TO BE REVIEWED!!!!

Overview

Acegi handles the security layer for KSB. Acegi uses remote method invocation to hold the application's security context and to propagate this object through to the service layer.

Credentials types

There are several security types you can use to propagate the security context object:

- CAS

- USERNAME_PASSWORD

- JAAS

- X509

CredentialsSource

The CredentialsSource is an interface that helps obtain security credentials. It encapsulates the actual source of credentials. The two ways to obtain the source are:

- X509CredentialsSource - X509 Certificate

- UsernamePasswordCredentialsSource - Username and Password

KSB security: Server side configuration

Here is a code snippet that shows the changes needed to configure KSB security on the server side:

```
<bean id="ksbConfigurer" class="org.kuali.rice.ksb.messaging.config.KSBConfigurer">
    <!-- Other properties removed -->
    <property name="services">
        <list>
            <bean class="org.kuali.rice.ksb.api.bus.support.SoapServiceDefinition">
                <property name="service">
                    <ref bean="soapService" />
                </property>
                <property name="localServiceName" value="soapLocalName"/>
                <property name="serviceNameSpaceURI" value="soapNameSpace"/>
                <property name="serviceInterface" value="org.kuali.ksb.examples.SOAPEchoService"/>
                <property name="priority" value="3"/>
                <property name="retryAttempts" value="1" />
                <property name="busSecurity" value="false"></property>

                <!-- Valid Values: CAS, KERBEROS -->
                <property name="credentialsType" value="CAS"/>
            </bean>
            <bean class="org.kuali.rice.ksb.api.bus.support.JavaServiceDefinition">
                <property name="service" ref="echoService"></property>
                <property name="localServiceName" value="javaLocalName" />
                <property name="serviceNameSpaceURI" value="javaNameSpace"/>
                <property name="serviceInterface" value="org.kuali.ksb.examples.EchoService"/>
                <property name="priority" value="5" />
                <property name="retryAttempts" value="1" />
                <property name="busSecurity" value="true" />

                <!-- Valid Values: CAS, KERBEROS -->
                <property name="credentialsType" value="CAS"/>
            </bean>
            <!-- Other services removed -->
        </list>
    </property>
</bean>
```

KSB security: Client side configuration

```
<bean id="customCredentialsSourceFactory"
 class="edu.myinstituition.myapp.security.credentials.CredentialsSourceFactory" />

<bean id="coreConfigurer" class="org.kuali.rice.core.impl.config.module.CoreConfigurer">
    <!-- Other properties removed -->
    <property name="credentialsSourceFactory" ref="customCredentialsSourceFactory">
</bean>
```

KSB connector and exporter code

Connectors

Connectors are used by a client to connect to a service that is usually exposed through the KSB registry. The Service Connector factory provides a bean that holds a proxy to a remote service with some contextual information. The factory determines the type of proxy to invoke based on the service definition. The service definition used by the server is serialized to the database and de-serialized by the client. There are different types of connectors supported by KSB, most notable are SOAP and Java over HTTP.

Exporters

Services, when exported, can be secured using standard Acegi methods. A security manager and an interceptor help organize the set of Business Objects that are exported.

Security and Keystores

Generating the Keystore

For client applications to be able to consume secured services hosted from a Rice server, the implementer must generate a keystore. As an initial setup, KSB security relies on the creation of a keystore using the JVM keytool as follows:

Step 1: Create the Keystore

The first step is to create the keystore and generate a public-private key combination for the client application. When using secured services on the KSB, we require the client applications transfer their messages digitally signed so that Rice can verify the messages authenticity. This is why we must generate these keys.

Generate your initial Rice keystore as follows:

```
keytool -genkey -validity 9999 -alias rice -keyalg RSA -keystore rice.keystore -dname "cn=rice" -keypass r1c3pw
 -storepass r1c3pw
```

Caution

keypass and storepass should be the same.

r1c3pw is the password used for the provided example.

Step 2: Sign the Key

This generates the keystore in a file called "rice_keystore" in the current directory and generates an RSA key with the alias of "rice". Since there is no certificate signing authority to sign our key, we must sign it ourselves. To do this, execute the following command:

```
keytool  -selfcert -validity 9999 -alias rice -keystore rice.keystore -keypass r1c3pw -storepass r1c3pw
```

Step 3: Generate the Certificate

After the application's certificate has been signed, we must export it so that it can be imported into the Rice keystore. To export a certificate, execute the following command:

```
keytool  -export -alias rice -file rice.cert -keystore rice.keystore -storepass r1c3pw
```

Step 4: Import Application Certificates

The client application's certificate can be imported using the following command:

```
keytool  -import -alias rice -file client.application.cert.file -keystore rice.keystore -storepass r1c3pw
```

The keystore file will end up deployed wherever your keystores are stored so hang on to both of these files and don't lose them! Also, notice that we specified a validity of 9999 days for the keystore and cert. This is so you do not have to continually update these keystores. This will be determined by your computing standards on how you handle key management.

Configure KSB to use the keystore

The following params are needed in the xml config to allow the ksb to use the keystore:

```
<param name="keystore.file">/usr/local/rice/rice.keystore</param>
<param name="keystore.alias">rice</param>
<param name="keystore.password"> password </param>
```

- keystore.file - is the location of the keystore

- keystore.alias - is the alias used in creating the keystore above

- keystore.password - this is the password of the alias AND the keystore. This assumes that the keystore is up in such a way that these are the same.

BasicAuthenticationService

The **BasicAuthenticationService** allows services published on the KSB to be accessed securely with basic authentication. As an example, here is how the **Workflow Document Actions Service** could be exposed as a service with basic authentication.

- Add the following bean to a spring bean file that is loaded as a part of the KEW module.

```
<bean id="rice.kew.workflowDocumentActionServiceBasicAuthentication.exporter"
            parent="kewServiceExporter" lazy-init="false">
    <property name="serviceDefinition">
        <bean parent="kewService">
            <property name="service">
                <ref bean="rice.kew.workflowDocumentActionsService" />
            </property>
            <property name="localServiceName"
                value="workflowDocumentActionsService-basicAuthentication" />
            <property name="busSecurity"
                value="${rice.kew.workflowDocumentActionsService.secure}" />
            <property name="basicAuthentication" value="true" />
        </bean>
    </property>
</bean>
```

- Add the following bean to a spring bean file that is loaded as a part of the KSB module.

```
<bean class="org.kuali.rice.ksb.service.BasicAuthenticationCredentials">
    <property name="serviceNameSpaceURI"
        value="http://rice.kuali.org/kew/v2_0" />
    <property name="localServiceName"
        value="workflowDocumentActionsService-basicAuthentication" />
    <property name="username"
        value="${WorkflowDocumentActionsService.username}" />
    <property name="password"
        value="${WorkflowDocumentActionsService.password}" />
    <property name="authenticationService" ref="basicAuthenticationService" />
```

```
    </bean>
```

- Add the following config parameters to a secure file that is loaded when the application is started.

```
<param name="WorkflowDocumentActionsService.username">username</param>
<param name="WorkflowDocumentActionsService.password">pw</param>
```

- To verify the new service can be called, it can be tested using a tool such as soapUI. Here is an example call which will invoke the method **logAnnotation** on **WorkflowDocumentActionsServiceImpl**.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                xmlns:v2="http://rice.kuali.org/kew/v2_0">
  <soapenv:Header>
    <wsse:Security xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
secext-1.0.xsd"
             soapenv:mustUnderstand="1">
      <wsse:UsernameToken xmlns:wsu=
          "http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
          wsu:Id="UsernameToken-1815911473">
        <wsse:Username>username</wsse:Username>
        <wsse:Password Type=
          "http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-
profile-1.0#PasswordText">pw</wsse:Password>
      </wsse:UsernameToken>
    </wsse:Security>
  </soapenv:Header>
  <soapenv:Body>
    <v2:logAnnotation>
      <v2:documentId>123456</v2:documentId>
      <v2:principalId>admin</v2:principalId>
      <v2:annotation>Add this annotation please.</v2:annotation>
    </v2:logAnnotation>
  <soapenv:Body>
</soapenv:Envelope>
```

Queue and Topic invocation

When you deploy a service, you can configure it for queue or for topic invocation using the **setQueue** property on the **ServiceDefinition**. The default is to register it as a queue-style service. The distinction between queue and topic invocation occurs when there is more than one service registered under the same **QName**.

Queue invocation

Remote service proxies obtained through the resource loader stack using **getService(QName)** (ultimately through the **ServiceBus**) are inherently synchronous. In the presence of multiple service registrations, the **ServiceBus** will choose one at random.

When invoking services asynchronously through the **MessageHelper**, an asynchronous service call proxy will be constructed with all available service definitions. The **MessageServiceInvoker** is called to invoke each service. If the service is defined as a queue service, then the **ServiceBus** will be consulted in a similar fashion to determine a single service to call. After the first queue service invocation the **MessageServiceInvoker** will return.

Topic invocation

The simplest way to invoke a topic service is using the **MessageHelper** functions to invoke the service asynchronously. As described above for an asynchronous queue invocation, an asynchronous service call

proxy will be constructed with the list of all of the services registered as a topic under the given name. Each of these services will be independently obtained and invoked by the **MessageServiceInvoker**.

Invoking a topic synchronously, however, requires use of a synchronous service call proxy to aggregate all of the topic's services. This functionality is not directly available via the **ServiceBus** API because the **ServiceBus** acts as a facade for direct service invocation.

To invoke a topic synchronously, you can construct a **SynchronousServiceCallProxy** using **SynchronousServiceCallProxy.createInstance**, passing the list of **Endpoint** obtained using **ServiceBus.getEndpoints(QName)**. This is done, for example, by **MessageHelperImpl** when the bus has been forced into synchronous mode via the **message.delivery** config param.

The synchronous service call proxy is the same as the asynchronous service call proxy, except that it does not queue up the invocation, it will invoke it blockingly. The same queue/topic distinctions described above apply when you invoke a topic synchronously. Under the normal queue situation, use of the synchronous service call proxy is not necessary because, as mentioned above, remote services obtained through the **ServiceBus** are naturally synchronous. You can see this in the example below:

```
List<Endpoint> servicesToProxy = KsbApiServiceLocator.getServiceBus().getEndpoints(qname);

SynchronousServiceCallProxy sscp = return SynchronousServiceCallProxy.createInstance(servicesToProxy, callback,
 context, value1, value2);
```

KSB Parameters

Here is a comprehensive set of configuration parameters used to configure the Kuali Service Bus.

Core Parameters

Table 8.6. Core Parameters

Core	Description	Default
serviceServletUrl	URL that maps to the KSB Servlet. It handles incoming requests from the service bus.	${application.url}/remoting/
rice.ksb.config.allowSelfSignedSSL	Indicates if self-signed certificates are permitted for https communication on the service bus	false
application.id	Application identifier for client application	
keystore.file	Path to the keystore file to use for security	
keystore.alias	Alias of the standalone server's key	
keystore.password	Password to access the keystore and the server's key	
ksb.mode	Mode in which to load the KSB module	local
ksb.url	The URL of the KSB web application	${application.url}/ksb
rice.ksb.struts.config.files	The file that defines the struts context for the KRice KSB struts module	/ksb/WEB-INF/struts-config.xml
dev.mode	If *true*, application will not publish or consume services from the central service registry, but will maintain a local copy of the registry. This is intended only for client application development purposes.	false
bam.enabled	If *true*, will monitor and log the service calls made and general business activity performed to the database. *Recommendation*: Enable this only for testing purposes, as there is a significant performance impact when enabled.	false
message.persistence	If *true*, messages are stored in the database until sent. If *false*, they are stored in memory.	true
message.delivery	Specifies whether messages are sent synchronously are asynchronously. Valid values are *synchronous* or *async*	async

Core	Description	Default
message.off	If set to *true*, then messages will not be sent but will instead pile up in the message queue. Intended for development and debugging purposes only.	false
Routing.ImmediateExceptionRouting	If *true*, messages will go immediately to exception routing if they fail, rather than being retried	false
RouteQueue.maxRetryAttempts	Default number of times to retry messages that fail to be delivered successfully.	5
RouteQueue.maxRetryAttemptsOverride	If set, will override the max retry setting for ALL services, even if they have their own custom retry setting.	
ksb.org.quartz.*	Can define any property beginning with *ksb.org.quartz* and it will be passed to the internal KSB quartz configuration as a property beginning with *org.quartz* (more details below)	
useQuartzDatabase	If *true*, then Quartz scheduler in Rice will use a database-backed job store; if *false*, then jobs will be stored in memory	true

serviceServletUrl

The URL that resolves to the KSB servlet that handles incoming requests from the service bus. All services exported onto the service bus use this value to construct their endpoint URLs when they are published to the service registry. See section below on configuring the *KSBDispatcherServlet*. This parameter should point to the absolute URL of where that servlet is mapped. It should include a trailing slash.

application.id

An identifier that indicates the name of the *logical* node on the service bus. If the application is running in a cluster, this should be the same for each machine in the cluster. This is used for namespacing of services, among other things. All services exported from the client application onto the service bus use this value as their default namespace unless otherwise specified.

keystore.file, keystore.alias, keystore.password

See the documentation below on keystore management.

ksb.mode

Mode in which to load the KSB module. Valid values are *local* and *embedded*. There is currently no difference in how the KSB module loads based on these settings. It will always try to load the KSB struts module if a *KualiActionServlet* is configured.

ksb.url

The URL of the KSB web application screens

rice.ksb.struts.config.files

The file that defines the struts context for the KRice KSB struts module. The struts module is loaded automatically if a *KualiActionServlet* is configured in the *web.xml*.

dev.mode

Indicates whether this node should export and consume services from the entire service bus. If set to *false*, then the machine will not register its services in the global service registry. Instead, it can only consume services that it has available locally. In addition to this, other nodes on the service bus will not be able to "see" this node and will therefore not forward any messages to it.

message.persistence

If *true*, then messages will be persisted to the datastore. Otherwise, they will only be stored in memory. If message persistence is not turned on and the server is shutdown while there are still messages that need to be sent, those messages will be lost. For a production environment, it is recommended that message persistence be set to *true*.

message.delivery

Can be set to either *synchronous* or *async*. If this is set to *synchronous*, then messages that are sent in an asynchronous fashion using the KSB API will instead be sent synchronously. This is useful in certain development and unit testing scenarios. For a production environment, it is recommended that message delivery be set to *async*.

message.off

If set to *true* then asynchronous messages will not be sent. In the case that message persistence is turned on, they will be persisted in the message store and can even be picked up later using the Message Fetcher. However, if message persistence is turned off, these messages will be lost. This can be useful in certain debugging or testing scenarios.

RouteQueue.maxRetryAttempts

Sets the default number of retries that will be executed if a message fails to be sent. This retry count can also be customized for a specific service. (See Exposing Services on the Bus)

RouteQueue.timeIncrement

Sets the default time increment between retry attempts. As with *RouteQueue.maxRetryAttempts* this can also be configured at the service level.

RouteQueue.maxRetryAttemptsOverride

If set with a number, it will temporarily set the retry attempts for ALL services going into exception routing. A good way to prevent all messages in a node that is having trouble from making it to exception routing is by setting the number arbitrarily high. The *message.off* param does the same thing.

Routing.ImmediateExceptionRouting

If set to *true*, then messages that fail to be sent will not be re-tried. Instead their *MessageExceptionHandler* will be invoked immediately.

useQuartzDatabase

When using the embedded Quartz scheduler started by the KSB, indicates whether that Quartz scheduler should store its entries in the database. If this is *true*, then the appropriate Quartz properties should be set as well (see ksb.org.quartz.* below).

ksb.org.quartz.*

Can be used to pass Quartz properties to the embedded Quartz scheduler. See the configuration documentation on the Quartz site. Essentially, any property prefixed with **ksb.org.quartz.** will have the "*ksb.*" portion stripped and will be sent as configuration parameters to the embedded Quartz scheduler.

KSB Configurer Properties

In addition to the configuration parameters available in the KRice configuration system, the *KSBConfigurer* bean has some properties that can be injected to configure it:

exceptionMessagingScheduler

By default, the KSB uses an embedded Quartz scheduler for scheduling the retry of messages that fail to be sent. If desired, a Quartz scheduler can instead be injected into the KSBConfigurer and it will use that scheduler instead. See Quartz Scheduling for more detail.

messageDataSource

Specifies the **javax.sql.DataSource** to use for storing the asynchronous message queue. If not specified, this defaults to the DataSource injected into the RiceConfigurer.

If this DataSource is injected, then the registryDataSource must also be injected, and vice-versa.

registryDataSource

Specifies the **javax.sql.DataSource** to use for reading and writing from the Service Registry. If not specified, this defaults to the DataSource injected into the RiceConfigurer.

If this DataSource is injected, then the **messageDataSource** must also be injected, and vice-versa.

overrideServices

See Acquiring and invoking services

Services

See Acquiring and invoking services

JAX-RS / RESTful services

Rice now allows allows RESTful (JAX-RS) services to be exported and consumed on the Kuali Service Bus (KSB). For some background on REST, see http://en.wikipedia.org/wiki/Representational_State_Transfer.

For details on JAX-RS, see JSR-311.

Caveats

- The KSB does **not** currently support "busSecure" (digital signing of requests & responses) REST services. Attempting to set a REST service's "busSecure" property to "true" will result in a RiceRuntimeException being thrown. Rice can be customized to expose REST services in a secure way, e.g. using SSL and an authentication mechanism such as client certificates, but that is beyond the scope of this documentation.

- If the JAX-RS annotations on your resource class don't cover all of its public methods, then accessing the non-annotated methods over the bus will result in an Exception being thrown.

A Simple Example

To expose a simple JAX-RS annotated service on the bus, you can follow this recipe for your spring configuration (which comes from the Rice unit tests):

```xml
<!-- The service implementation you want to expose -->

<bean id="baseballCardCollectionService"
 class="org.kuali.rice.ksb.testclient1.BaseballCardCollectionServiceImpl"/>


<!-- The service definition which tells the KSB to expose our RESTful service -->
<bean class="org.kuali.rice.ksb.messaging.RESTServiceDefinition">
    <property name="serviceNameSpaceURI" value="test" />


    <!-- as noted earlier, the servicePath property of RESTServiceDefinition can't be set here  -->


    <!-- The service to expose.  Refers to the bean above -->
    <property name="service" ref="baseballCardCollectionService" />


    <!-- The "Resource class", the class with the JAX-RS annotations on it.  Could be the same as the  -->
    <!-- service implementation, or could be different, e.g. an interface or superclass     -->

    <property name="resourceClass"
value="org.kuali.rice.ksb.messaging.remotedservices.BaseballCardCollectionService" />


    <!-- the name of the service, which will be part of the RESTful URLs used to access it -->
    <property name="localServiceName" value="baseballCardCollectionService" />
</bean>
```

The following java interface uses JAX-RS annotations to specify its RESTful interface:

```java
// … eliding package and imports

@Path("/")
public interface BaseballCardCollectionService {
    @GET
    public List<BaseballCard> getAll();

    /**
      * gets a card by it's (arbitrary) identifier
      */
    @GET
    @Path("/BaseballCard/id/{id}")
    public BaseballCard get(@PathParam("id") Integer id);
    /**
      * gets all the cards in the collection with the given player name
      */
    @GET
    @Path("/BaseballCard/playerName/{playerName}")
    public List<BaseballCard> get(@PathParam("playerName") String playerName);


    /**
      * Add a card to the collection.  This is a non-idempotent method
      * (because you can add more than one of the same card), so we'll use @POST
      * @return the (arbitrary) numerical identifier assigned to this card by the service
      */
    @POST
    @Path("/BaseballCard")
    public Integer add(BaseballCard card);


    /**
      * update the card for the given identifier.  This will replace the card that was previously
      * associated with that identifier.
```

```
    */
    @PUT
    @Path("/BaseballCard/id/{id}")
    @Consumes("application/xml")
    public void update(@PathParam("id") Integer id, BaseballCard card);


    /**
     * delete the card with the given identifier.
     */
    @DELETE
    @Path("/BaseballCard/id/{id}")
    public void delete(@PathParam("id") Integer id);

    /**
     * This method lacks JAX-RS annotations
     */
    public void unannotatedMethod();
}
```

Acquisition and use of this service over the KSB looks just like that of any other KSB service. In the synchronous case:

```
BaseballCardCollectionService baseballCardCollection = (BaseballCardCollectionService)
 GlobalResourceLoader.getService(new QName("test", "baseballCardCollectionService");
);


List<BaseballCard> allMyMickeyMantles = baseballCardCollection.get("Mickey Mantle");
// baseballCardCollection.<other service method>(...)
// etc
```

Composite Services

It is also possible to aggregate multiple Rice service implementations into a single RESTful service where requests to different sub-paths off of the base service URL can be handled by different underlying services. This may be desirable to expose a RESTful service that is more complex than could be cleanly factored into a single java service interface.

The configuration for a composite RESTfull service looks a little bit different, and as might be expected given the one-to-many mapping from RESTful service to java services, there are some caveats to using that service over the KSB. Here is a simple example of a composite service definition (which also comes from the Rice unit tests):

```
<bean class="org.kuali.rice.ksb.messaging.RESTServiceDefinition">
    <property name="serviceNameSpaceURI" value="test" />
    <property name="localServiceName" value="kms" />
    <property name="resources">
        <list>
            <ref bean="inboxResource"/>
            <ref bean="messageResource"/>
        </list>
    </property>
    <property name="servicePath" value="/" />
</bean>


<!-- the beans referenced above are just JAX-RS annotated Java services -->
<bean id="inboxResource" class="org.kuali.rice.ksb.testclient1.InboxResourceImpl">
    <!-- ... eliding bean properties ... -->
</bean>
<bean id="messageResource" class="org.kuali.rice.ksb.testclient1.MessageResourceImpl">
    <!-- ... eliding bean properties ... -->

</bean>
```

As you can see in the bean definition above, the service name is kms, so the base service url would by default (in a dev environment) be **http://localhost:8080/kr-dev/remoting/kms/** . Acquiring a composite service such as this one on the KSB will actually return you a **org.kuali.rice.ksb.messaging.serviceconnectors.ResourceFacade**, which allows you to get the individual java services in a couple of ways, as shown in the following simple example:

```
ResourceFacade kmsService =
 (ResourceFacade) GlobalResourceLoader.getService(
new QName(NAMESPACE, KMS_SERVICE));


// Get service by resource name (url path)
InboxResource inboxResource = kmsService.getResource("inbox");
// Get service by resource class
MessageResource messageResource = kmsService.getResource(MessageResource.class);
```

Additional Service Definition Properties

There are some properties on the RESTServiceDefinition object that let you do more advanced configuration:

Providers

JAX-RS Providers allow you to define:

- ExceptionMappers which will handle specific Exception types with specific Responses.

- MessageBodyReaders and MessageBodyWriters that will convert custom Java types to and from streams.

- ContextResolver providers allow you to create special JAXBContexts for specific types, which will gives you fine control over marshalling, unmarshalling, and validation.

The JAX-RS specification calls for classes annotated with @Provider to be automatically used in the underlying implementation, but the CXF project which Rice uses under the hood does not (at the time of this writing) support this configuration mechanism, so this configuration property is currently necessary.

Extension Mappings

Ordinarily you need to set your ACCEPT header to ask for a specific representation of a resource. ExtensionMappings let you map certain file extensions to specific media types for your RESTful service, so your URLs can then optionally specify a media type directly. For example you could map the **.xml** extension to the media type **text/xml**, and then tag .xml on to the end of your resource URL to specify that representation.

Language Mappings

language mappings allow you a way to control the the Content-Language header, which lets you specify which languages your service can accept and provide.

Additional Information

For more information on what these properties provide, it may be helpful to consult the JAX-RS specification, or the CXF documentation.

Glossary

A

Action List
: A list of the user's notification and workflow items. Also called the user's Notification List. Clicking an item in the Action List displays details about that notification, if the item is a notification, or displays that document, if it is a workflow item. The user will usually load the document from their Action List in order to take the requested action against it, such as approving or acknowledging the document.

Action List Type
: This tells you if the Action List item is a notification or a more specific workflow request item. When the Action List item is a notification, the Action List Type is "Notification."

Action Request
: A request to a user or Workgroup to take action on a document. It designates the type of action that is requested, which includes:

- Approve: requests an approve or disapprove action.

- Complete: requests a completion of the contents of a document. This action request is displayed in the Action List after the user saves an incomplete document.

- Acknowledge: requests an acknowledgment by the user that the document has been opened - the doc will not leave the Action List until acknowledgment has occurred; however, the document routing will not be held up and the document will be permitted to transaction into the processed state if neccessary.

- FYI: a notification to the user regarding the document. Documents requesting FYI can be cleared directly from the Action List. Even if a document has FYI requests remaining, it will still be permitted to transition into the FINAL state.

Action Request Hierarchy
: Action requests are hierarchical in nature and can have one parent and multiple children.

Action Requested
: The action one needs to take on a document; also the type of action that is requested by an Action Request. Actions that may be requested of a user are:

- Acknowledge: requests that the users states he or she has reviewed the document.

- Approve: requests that the user either Approve or Disapprove a document.

- Complete: requests the user to enter additional information in a document so that the content of the document is complete.

- FYI: intended to simply makes a user aware of the document.

Action Taken
: An action taken on a document by a *Reviewer* in response to an Action Request. The Action Taken may be:

- Acknowledged: Reviewer has viewed and acknowledged document.

- Approved: Reviewer has approved the action requested on document.

- Blanket Approved: Reviewer has requested a blanket approval up to a specified point in the route path on the document.

- Canceled: Reviewer has canceled the document. The document will not be routed to any more reviewers.

- Cleared FYI: Reviewer has viewed the document and cleared all of his or her pending FYI(s) on this document.

- Completed: Reviewer has completed and supplied all data requested on document.

- Created Document: User has created a document

- Disapproved: Reviewer has disapproved the document. The document will not being routed to any subsequent reviewers for approval. Acknowledge Requests are sent to previous approvers to inform them of the disapproval.

- Logged Document: Reviewer has added a message to the Route Log of the document.

- Moved Document: Reviewer has moved the document either backward or forward in its routing path.

- Returned to Previous Node: Reviewer has returned the document to a previous routing node. When a Reviewer does this, all the actions taken between the current node and the return node are removed and all the pending requests on the document are deactivated.

- Routed Document: Reviewer has submitted the document to the workflow engine for routing.

- Saved: Reviewer has saved the document for later completion and routing.

- Superuser Approved Document: *Superuser* has approved the entire document, any remaining routing is cancelled.

- Superuser Approved Node: Superuser has approved the document through all nodes up to (but not including) a specific node. When the document gets to that node, the normal Action Requests will be created.

- Superuser Approved Request: Superuser has approved a single pending Approve or Complete Action Request. The document then goes to the next routing node.

- Superuser Cancelled: Superuser has canceled the document. A Superuser can cancel a document without a pending Action Request to him/her on the document.

- Superuser Disapproved: Superuser has disapproved the document. A Superuser can disapprove a document without a pending Action Request to him/her on the document.

- Superuser Returned to Previous Node: Superuser has returned the document to a previous routing node. A Superuser can do this without a pending Action Request to him/her on the document.

Activated	The state of an action request when it is has been sent to a user's Action List.
Activation	The process by which requests appear in a user's Action List
Activation Type	Defines how a route node handles activation of Action Requests. There are two standard activation types:

• Sequential: Action Requests are activated one at a time based on routing priority. The next Action Request isn't activated until the previous request is satisfied.

• Parallel: All Action Requests at the route node are activated immediately, regardless of priority |
Active Indicator	An indicator specifying whether an object in the system is active or not. Used as an alternative to complete removal of an object.
Ad Hoc Routing	A type of routing used to route a document to users or groups that are not in the Routing path for that Document Type. When the Ad Hoc Routing is complete, the routing returns to its normal path.
Annotation	Optional comments added by a *Reviewer* when taking action. Intended to explain or clarify the action taken or to advise subsequent Reviewers.
Approve	A type of workflow action button. Signifies that the document represents a valid business transaction in accordance with institutional needs and policies in the user's judgment. A single document may require approval from several users, at multiple route levels, before it moves to final status.
Approver	The user who approves the document. As a document moves through Workflow, it moves one route level at a time. An Approver operates at a particular route level of the document.
Attachment	The pathname of a related file to attach to a Note. Use the "Browse..." button to open the file dialog, select the file and automatically fill in the pathname.
Attribute Type	Used to strongly type or categorize the values that can be stored for the various attributes in the system (e.g., the value of the arbitrary key/value pairs that can be defined and associated with a given parent object in the system).
Authentication	The act of logging into the system. The Out of the box (OOTB) authenticaton implementation in Rice does not require a password as it is intended for testing puposes only. This is something that must be enabled as part of an implementation. Various authentication solutions exist, such as CAS or Shibboleth, that an implementer may want to use depending on their needs.
Authorization	Authorization is the permissions that an authenticated user has for performing actions in the system.
Author Universal ID	A free-form text field for the full name of the Author of the Note, expressed as "Lastname, Firstname Initial"

B

Base Rule Attribute	The standard fields that are defined and collected for every *Routing Rule* These include:

- Active: A true/false flag to indicate if the Routing Rule is active. If false, then the rule will not be evaluated during routing.

- Document Type: The *Document Type* to which the Routing Rule applies.

- From Date: The inclusive start date from which the Routing Rule will be considered for a match.

- Force Action: a true/false flag to indicate if the review should be forced to take action again for the requests generated by this rule, even if they had taken action on the document previously.

- Name: the name of the rule, this serves as a unique identifier for the rule. If one is not specified when the rule is created, then it will be generated.

- Rule Template: The Rule Template used to create the Routing Rule.

- To Date: The inclusive end date to which the Routing Rule will be considered for a match.

Blanket Approval	Authority that is given to designated *Reviewers* who can approve a document to a chosen route point. A Blanket Approval bypasses approvals that would otherwise be required in the *Routing* For an authorized Reviewer, the *Doc Handler* typically displays the Blanket Approval button along with the other options. When a Blanket Approval is used, the Reviewers who are skipped are sent Acknowledge requests to notify them that they were bypassed.
Blanket Approve Workgroup	A workgroup that has the authority to Blanket Approve a document.
Branch	A path containing one or more Route Nodes that a document traverses during routing. When a document enters a *Split Node* multiple branches can be created. A *Join Node* joins multiple branches together.
Business Rule	1. Describes the operations, definitions and constraints that apply to an organization in achieving its goals.
	2. A restriction to a function for a business reason (such as making a specific object code unavailable for a particular type of disbursement). Customizable business rules are controlled by Parameters.

C

Campus	Identifies the different fiscal and physical operating entities of an institution.
Campus Type	Designates a campus as physical only, fiscal only or both.
Cancel	A workflow action available to document initiators on documents that have not yet been routed for approval. Denotes that the document is void and should be disregarded. Canceled documents cannot be modified in any way and do not route for approval.

Canceled	A routing status. The document is denoted as void and should be disregarded.
CAS - Central Authentication Service	http://www.jasig.org/cas - An open source authentication framework. Kuali Rice provides support for integrating with CAS as an authentication provider (among other authentication solutions) and also provides an implementation of a CAS server that integrates with Kuali Identity Management.
Client	A Java Application Program Interface (API) for interfacing with the Kuali Enterprise Workflow Engine.
Client/Server	The use of one computer to request the services of another computer over a network. The workstation in an organization will be used to initiate a business transaction (e.g., a budget transfer). This workstation needs to gather information from a remote database to process the transaction, and will eventually be used to post new or changed information back onto that remote database. The workstation is thus a Client and the remote computer that houses the database is the Server.
Close	A workflow action available on documents in most statuses. Signifies that the user wishes to exit the document. No changes to Action Requests, Route Logs or document status occur as a result of a Close action. If you initiate a document and close it without saving, it is the same as canceling that document.
Comma-separated value	A file format using commas as delimiters utilized in import and export functionality.
Complete	A pending action request to a user to submit a saved document.
Completed	The action taken by a user or group in response to a request in order to finish populating a document with information, as evidenced in the Document Route Log.
Country Restricted Indicator	Field used to indicate if a country is restricted from use in procurement. If there is no value then there is no restriction.
Creation Date	The date on which a document is created.
CSV	See *comma-separated value*

D

Date Approved	The date on which a document was most recently approved.
Date Finalized	The date on which a document enters the FINAL state. At this point, all approvals and acknowledgments are complete for the document.
Deactivation	The process by which requests are removed from a user's *Action List*
Delegate	A user who has been registered to act on behalf of another user. The Delegate acts with the full authority of the Delegator. Delegation may be either *Primary Delegation* or *Secondary Delegation*.
Delegate Action List	A separate Action List for Delegate actions. When a Delegate selects a Delegator for whom to act, an Action List of all documents sent to the Delegator is displayed.

For both *Primary* and *Secondary Delegation* the Delegate may act on any of the entries with the full authority of the Delegator.

Disapprove	A workflow action that allows a user to indicate that a document does not represent a valid business transaction in that user's judgment. The initiator and previous approvers will receive Acknowledgment requests indicating the document was disapproved.
Disapproved	A status that indicates the document has been disapproved by an approver as a valid transaction and it will not generate the originally intended transaction.
Doc Handler	The Doc Handler is a web interface that a Client uses for the appropriate display of a document. When a user opens a document from the Action List or Document Search, the Doc Handler manages access permissions, content format, and user options according to the requirements of the Client.
Doc Handler URL	The URL for the *Doc Handler*.
Doc Nbr	See *Document Number*.
Document	Also see *E-Doc*.
	An electronic document containing information for a business transaction that is routed for Actions in KEW. It includes information such as Document ID, Type, Title, Route Status, Initiator, Date Created, etc. In KEW, a document typically has XML content attached to it that is used to make routing decisions.
Document Id	See *Document Number*.
Document Number	A unique, sequential, system-assigned number for a document
Document Operation	A workflow screen that provides an interface for authorized users to manipulate the XML and other data that defines a document in workflow. It allows you to access and open a document by Document ID for the purpose of performing operations on the document.
Document Search	A web interface in which users can search for documents. Users may search by a combination of document properties such as Document Type or Document ID, or by more specialized properties using the Detailed Search. Search results are displayed in a list similar to an Action List.
Document Status	See also *Route Status*.
Document Title	The title given to the document when it was created. Depending on the Document Type, this title may have been assigned by the Initiator or built automatically based on the contents of the document. The Document Title is displayed in both the Action List and Document Search.
Document Type	The Document Type defines the routing definition and other properties for a set of documents. Each document is an instance of a Document Type and conducts the same type of business transaction as other instances of that Document Type.

Document Types have the following characteristics:

- They are specifications for a document that can be created in KEW

- They contain identifying information as well as policies and other attributes

- They defines the Route Path executed for a document of that type (Process Definition)

- They are hierarchical in nature may be part of a hierarchy of Document Types, each of which inherits certain properties of its *Parent Document Type*.

- They are typically defined in XML, but certain properties can be maintained from a graphical interface

Document Type Hierarchy	A hierarchy of Document Type definitions. Document Types inherit certain attributes from their parent Document Types. This hierarchy is also leveraged by various pieces of the system, including the Rules engine when evaluating rule sets and KIM when evaluating certain Document Type-based permissions.
Document Type Label	The human-readable label assigned to a Document Type.
Document Type Name	The assigned name of the document type. It must be unique.
Document Type Policy	These advise various checks and authorizations for instances of a Document Type during the routing process.
Drilldown	A link that allows a user to access more detailed information about the current data. These links typically take the user through a series of inquiries on different business objects.
Dynamic Node	An advanced type of *Route Node* that can be used to generate complex routing paths on the fly. Typically used whenever the route path of a document cannot be statically defined and must be completely derived from document data.

E

ECL	1. An acronym for Educational Community License.
	2. All Kuali software and material is available under the Educational Community License and may be adopted by colleges and universities without licensing fees. The open licensing approach also provides opportunities for support and implementation assistance from commercial affiliates.
E-Doc	An abbreviation for electronic documents, also a shorthand reference to documents created with eDocLite.
eDocLite	A framework for quickly building workflow-enabled documents. Allows you to define document screens in XML and render them using XSL style sheets.
Embedded Client	A type of client that runs an embedded workflow engine.
Employee Status	Found on the Person Document; defines the employee's current employment classification (for example, "A" for Active).
Employee Type	Found on the Person Document; defines the employee's position classification (for example, "P" for Professional).

Entity	An Entity record houses identity information for a given Person, Process, System, etc. Each Entity is categorized by its association with an Entity Type.
Entity Attribute	Entities have directory-like information called Entity Attributes that are associated with them
	Entity Attributes make up the identity information for an Entity record.
Entity Type	Provides categorization to Entities. For example, a "System" could be considered an Entity Type because something like a batch process may need to interface with the application.
Exception	A workflow routing status indicating that the document routed to an exception queue because workflow has encountered a system error when trying to process the document.
Exception Messaging	The set of services and configuration options that are responsible for handling messages when they cannot be successfully delivered. Exception Messaging is set up when you configure KSB using the properties outlined in KSB Module Configuration.
Exception Routing	A type of routing used to handle error conditions that occur during the routing of a document. A document goes into Exception Routing when the workflow engine encounters an error or a situation where it cannot proceed, such as a violation of a Document Type Policy or an error contacting external services. When this occurs, the document is routed to the parties responsible for handling these exception cases. This can be a group configured on the document or a responsibility configured in KIM. Once one of these responsible parties has reviewed the situation and approved the document, it will be resubmitted to the workflow engine to attempt the processing again.
Extended Attributes	Custom, table-driven business object attributes that can be established by implementing institutions.
Extension Rule Attribute	One of the rule attributes added in the definition of a rule template that extends beyond the base rule attributes to differentiate the routing rule. A Required Extension Attribute has its "Required" field set to True in the rule template. Otherwise, it is an Optional Extension Attribute. Extension attributes are typically used to add additional fields that can be collected on a rule. They also define the logic for how those fields will be processed during rule evaluation.

F

Field Lookup	The round magnifying glass icon found next to fields throughout the GUI that allow the user to look up reference table information and display (and select from) a list of valid values for that field.
Final	A workflow routing status indicating that the document has been routed and has no pending approval or acknowledgement requests.
Flexible Route Management	A standard KEW routing scheme based on rules rather than dedicated table-based routing.
FlexRM (Flexible Route Module)	The Route Module that performs the Routing for any Routing Rule is defined through FlexRM. FlexRM generates Action Requests when a Rule matches the

data value contained in a document. An abbreviation of "Flexible Route Module." A standard KEW routing scheme that is based on rules rather than dedicated table-based routing.

Force Action

A true/false flag that indicates if previous Routing for approval will be ignored when an *Action Request* is generated. The flag is used in multiple contexts where requests are generated (e.g., rules, ad hoc routing). If Force Action is False, then prior Actions taken by a user can satisfy newly generated requests. If it is True, then the user needs to take another Action to satisfy the request.

FYI

A workflow action request that can be cleared from a user's Action List with or without opening and viewing the document. A document with no pending approval requests but with pending Acknowledge requests is in Processed status. A document with no pending approval requests but with pending FYI requests is in Final status. See also *Ad Hoc Routing* and *Action Request*.

G

Group

A Group has members that can be either *Principals* or other Groups (nested). Groups essentially become a way to organize Entities (via Principal relationships) and other Groups within logical categories.

Groups can be given authorization to perform actions within applications by assigning them as members of *Roles*.

Groups can also have arbitrary identity information (i.e., *Group Attributes* hanging from them. Group Attributes might be values for "Office Address," "Group Leader," etc.

Groups can be maintained at runtime through a user interface that is capable of workflow.

Group Attribute

Groups have directory-like information called Group Attributes hanging from them. "Group Phone Number" and "Team Leader" are examples of Group Attributes.

Group Attributes make up the identity information for a Group record.

Group Attributes can be maintained at runtime through a user interface that is capable of workflow.

H

Hierarchical Tree Structure

A hierarchical representation of data in a graphical form.

I

Initialized

The state of an Action Request when it is first created but has not yet been Activated (sent to a user's Action List).

Initiated

A workflow routing status indicating a document has been created but has not yet been saved or routed. A Document Number is automatically assigned by the system.

Initiator	A user role for a person who creates (initiates or authors) a new document for routing. Depending on the permissions associated with the Document Type, only certain users may be able to initiate documents of that type.
Inquiry	A screen that allows a user to view information about a business object.

J

Join Node	The point in the routing path where multiple branches are joined together. A Join Node typically has a corresponding *Split Node* for which it joins the branches.

K

KC - Kuali Coeus	TODO
KCA - Kuali Commercial Affiliates	A designation provided to commercial affiliates who become part of the Kuali Partners Program to provide for-fee guidance, support, implementation, and integration services related to the Kuali software. Affiliates hold no ownership of Kuali intellectual property, but are full KPP participants. Affiliates may provide packaged versions of Kuali that provide value for installation or integration beyond the basic Kuali software. Affiliates may also offer other types of training, documentation, or hosting services.
KCB – Kuali Communications Broker	KCB is logically related to KEN. It handles dispatching messages based on user preferences (email, SMS, etc.).
KEN - Kuali Enterprise Notification	A key component of the Enterprise Integration layer of the architecture framework. Its features include: • Automatic Message Generation and Logging • Message integrity and delivery standards • Delivery of notifications to a user's Action List
KEW – Kuali Enterprise Workflow	Kuali Enterprise Workflow is a general-purpose electronic routing infrastructure, or workflow engine. It manages the creation, routing, and processing of electronic documents (eDocs) necessary to complete a transaction. Other applications can also use Kuali Enterprise Workflow to automate and regulate the approval process for the transactions or documents they create.
KFS – Kuali Financial System	Delivers a comprehensive suite of functionality to serve the financial system needs of all Carnegie-Class institutions. An enhancement of the proven functionality of Indiana University's Financial Information System (FIS), KFS meets GASB and FASB standards while providing a strong control environment to keep pace with advances in both technology and business. Modules include financial transactions, general ledger, chart of accounts, contracts and grants, purchasing/accounts payable, labor distribution, budget, accounts receivable and capital assets.
KIM – Kuali Identity Management	A Kuali Rice module, Kuali Identity Management provides a standard API for persons, groups, roles and permissions that can be implemented by an institution. It also provdies an out of the box reference implementation that allows for a university to use Kuali as their Identity Management solution.

KNS – Kuali Nervous System	A core technical module composed of reusable code components that provide the common, underlying infrastructure code and functionality that any module may employ to perform its functions (for example, creating custom attributes, attaching electronic images, uploading data from desktop applications, lookup/ search routines, and database interaction).
KPP - Kuali Partners Program	The Kuali Partners Program (KPP) is the means for organizations to get involved in the Kuali software community and influence its future through voting rights to determine software development priorities. Membership dues pay staff to perform Quality Assurance (QA) work, release engineering, packaging, documentation, and other work to coordinate the timely enhancement and release of quality software and other services valuable to the members. Partners are also encouraged to tender functional, technical, support or administrative staff members to the Kuali Foundation for specific periods of time.
KRAD - Kuali Rapid Application Development	TODO
KRMS - Kuali Rules Management System	TODO
KS - Kuali Student	Delivers a means to support students and other users with a student-centric system that provides real-time, cost-effective, scalable support to help them identify and achieve their goals while simplifying or eliminating administrative tasks. The high-level entities of person (evolving roles-student, instructor, etc.), time (nested units of time - semesters, terms, classes), learning unit (assigned to any learning activity), learning result (grades, assessments, evaluations), learning plan (intentions, activities, major, degree), and learning resources (instructors, classrooms, equipment). The concierge function is a self-service information sharing system that aligns information with needs and tasks to accomplish goals. The support for integration of locally-developed processes provides flexibility for any institution's needs.
KSB – Kuali Service Bus	Provides an out-of-the-box service architecture and runtime environment for Kuali Applications. It is the cornerstone of the Service Oriented Architecture layer of the architectural reference framework. The Kuali Service Bus consists of: • A services registry and repository for identifying and instantiating services • Run time monitoring of messages • Support for synchronous and asynchronous service and message paradigms
Kuali	1. Pronounced "ku-wah-lee". A partnership organization that produces a suite of community-source, modular administrative software for Carnegie-class higher education institutions. See also *Kuali Foundation* 2. (n.) A humble kitchen wok that plays an important role in a successful kitchen.
Kuali Foundation	Employs staff to coordinate partner efforts and to manage and protect the Foundation's intellectual property. The Kuali Foundation manages a growing portfolio of enterprise software applications for colleges and universities. A lightweight Foundation staff coordinates the activities of Foundation members for critical software development and coordination activities such as source code control, release engineering, packaging, documentation, project management,

software testing and quality assurance, conference planning, and educating and assisting members of the Kuali Partners program.

Kuali Rice

Provides an enterprise-class middleware suite of integrated products that allow both Kuali and non-Kuali applications to be built in an agile fashion, such that developers are able to react to end-user business requirements in an efficient manner to produce high-quality business applications. Built with Service Oriented Architecture (SOA) concepts in mind, KR enables developers to build robust systems with common enterprise workflow functionality, customizable and configurable user interfaces with a clean and universal look and feel, and general notification features to allow for a consolidated list of work "action items." All of this adds up to providing a re-usable development framework that encourages a simplified approach to developing true business functionality as modular applications.

L

Last Modified Date

The date on which the document was last modified (e.g., the date of the last action taken, the last action request generated, the last status changed, etc.).

M

Maintenance Document

An e-doc used to establish and maintain a table record.

Message

The full description of a *notification message*. This is a specific field that can be filled out as part of the Simple Message or Event Message form. This can also be set by the programmatic interfaces when sending notifications from a client system.

Message Queue

Allows administrators to monitor messages that are flowing through the Service Bus. Messages can be edited, deleted or forwarded to other machines for processing from this screen.

N

Namespace

A Namespace is a way to scope both *Permissions* and *Entity Attributes* Each Namespace instance is one level of scoping and is one record in the system. For example, "KRA" or "KC" or "KFS" could be a Namespace. Or you could further break those up into finer-grained Namespaces such that they would roughly correlate to functional modules within each application. Examples could be "KRA Rolodex", "KC Grants", "KFS Chart of Accounts".

Out of the box, the system is bootstrapped with numerous Rice namespaces which correspond to the different modules. There is also a default namespace of "KUALI".

Namespaces can be maintained at runtime through a maintenance document.

Note Text

A free-form text field for the text of a Note

Notification Content

This section of a *notification message* which displays the actual full message for the notification along with any other content-type-specific fields.

Notification Message	The overall Notification item or Notification Message that a user sees when she views the details of a notification in her Action List. A Notification Message contains not only common elements such as Sender, Channel, and Title, but also content-type-specific fields.

O

OOTB	Stands for "out of the box" and refers to the base deliverable of a given feature in the system.
Optimistic Locking	A type of "locking" that is placed on a database row by a process to prevent other processes from updating that row before the first process is complete. A characteristic of this locking technique is that another user who wants to make modifications at the same time as another user is permitted to, but the first one who submits their changes will have them applied. Any subsequent changes will result in the user being notified of the optimistic lock and their changes will not be applied. This technique assumes that another update is unlikely.
Optional Rule Extension Attribute	An Extension Attribute that is not required in a Rule Template. It may or may not be present in a *Routing Rule* created from the Template. It can be used as a conditional element to aid in deciding if a Rule matches. These Attributes are simply additional criteria for the Rule matching process.
Org Doc #	The originating document number.
Organization	Refers to a unit within the institution such as department, responsibility center, campus, etc.
Organization Code	Represents a unique identifier assigned to units at many different levels within the institution (for example, department, responsibility center, and campus).

P

Parameter Component Code	Code identifying the parameter Component.
Parameter Description	This field houses the purpose of this parameter.
Parameter Name	This will be used as the identifier for the parameter. Parameter values will be accessed using this field and the namespace as the key.
Parameter Type Code	Code identifying the parameter type. Parameter Type Code is the primary key for its' table.
Parameter Value	This field houses the actual value associated with the parameter.
Parent Document Type	A Document Type from which another *Document Type* derives. The child type can inherit certain properties of the parent type, any of which it may override. A Parent Document Type may have a parent as part of a hierarchy of document types.
Parent Rule	A Routing Rule in KEW from which another Routing Rule derives. The child Rule can inherit certain properties of the parent Rule, any of which it may override. A Parent Rule may have a parent as part of a hierarchy of Rules.
Permission	Permissions represent fine grained actions that can be mapped to functionality within a given system. Permissions are scoped to *Namespace* which roughly correlate to modules or sections of functionality within a given system.

A developer would code authorization checks in their application against these permissions.

Some examples would be: "canSave", "canView", "canEdit", etc.

Permissions are aggregated by *Roles*.

Permissions can be maintained at runtime through a user interface that is capable of workflow; however, developers still need to code authorization checks against them in their code, once they are set up in the system.

Attributes

1. Id - a system generated unique identifier that is the primary key for any Permission record in the system

2. Name - the name of the permission; also a human understandable unique identifier

3. Description - a full description of the purpose of the Permission record

4. Namespace - the reference to the associated *Namespace*

Relationships

1. Permission to *Role* - many to many; this relationship ties a Permission record to a Role that is authorized for the Permission

2. Permission to *Namespace* - many to one; this relationship allows for scoping of a Permission to a Namespace that contains functionality which keys its authorization checking off of said

Person Identifier	The username of an individual user who receives the document ad hoc for the Action Requested
Person Role	Creates or maintains the list used in selection of personnel when preparing the Routing Form document.
Pessimistic Locking	A type of lock placed on a database row by a process to prevent other processes from reading or updating that row until the first process is finished. This technique assumes that another update is likely.
Plugins	A plugin is a packaged set of code providing essential services that can be deployed into the Rice standalone server. Plugins usually contains only classes used in routing such as custom rules or searchable attributes, but can contain client application specific services. They are usually used only by clients being implemented by the 'Thin Client' method
Post Processor	A routing component that is notified by the workflow engine about various events pertaining to the routing of a specific document (e.g., node transition, status change, action taken). The implementation of a Post Processor is typically specific to a particular set of Document Types. When all required approvals are completed, the engine notifies the Post Processor accordingly. At this point, the Post Processor is responsible for completing the business transaction in the manner appropriate to its Document Type.

Posted Date/Time Stamp	A free-form text field that identifies the time and date at which the Notes is posted.
Postal Code	Defines zip code to city and state cross-references.
Preferences	User options in an Action List for displaying the list of documents. Users can click the Preferences button in the top margin of the Action List to display the Action List Preferences screen. On the Preferences screen, users may change the columns displayed, the background colors by Route Status, and the number of documents displayed per page.
Primary Delegation	The Delegator turns over full authority to the Delegate. The Action Requests for the Delegator only appear in the Action List of the Primary Delegate. The Delegation must be registered in KEW or KIM to be in effect.
Principal	A Principal represents an *Entity* that can authenticate into the system. One can roughly correlate a Principal to a login username. Entities can exist in KIM without having permissions or authorization to do anything; therefore, a Principal must exist and must be associated with an Entity in order for it to have access privileges. All authorization that is not specific to *Groups* is tied to a Principal.
	In other words, an Entity is for identity while a Principal is for access management.
	Also note that an Entity is allowed to have multiple Principals associated with it. The use case typically given here is that a person may apply to a school and receive one log in for the application system; however, once accepted, they may receive their official login, but use the same identity information set up for their Entity record.
Processed	A routing status indicating that the document has no pending approval requests but still has one or more pending acknowledgement requests.

R

Recipient Type	The type of entity that is receiving an Action Request. Can be a user, workgroup, or role.
Required Rule Extension Attribute	An Extension Attribute that is required in a Rule Template. It will be present in every Routing Rule created from the Template.
Responsibility	See *Responsible Party*.
Responsibility Id	A unique identifier representing a particular responsibility on a rule (or from a *route module* This identifier stays the same for a particular responsibility no matter how many times a rule is modified.
Responsible Party	The Reviewer defined on a routing rule that receives requests when the rule is successfully executed. Each routing rule has one or more responsible parties defined.
Reviewer	A user acting on a document in his/her *Action List* and who has received an *Action Request* for the document.
Rice	An abbreviation for Kuali Rice.
Role	Roles aggregate *Permissions* When Roles are given to *Entities* (via their relationship with Principals) or *Groups* an authorization for all associated Permissions is granted.

Route Header Id	Another name for the *Document Id*.
Route Log	Displays information about the routing of a document. The Route Log is usually accessed from either the Action List or a Document Search. It displays general document information about the document and a detailed list of Actions Taken and pending *Action Requests* for the document. The Route Log can be considered an audit trail for a document.
Route Module	A routing component that the engine uses to generate action requests at a particular *Route Node*. *FlexRM* (Flexible Route Module) is a general Route Module that is rule-based. Clients can define their own Route Modules that can conduct specialized Routing based on routing tables or any other desired implementation.
Route Node	Represents a step in the routing process of a document type. Route node "instances" are created dynamically as a document goes through its routing process and can be defined to perform any function. The most common functions are to generate Action Requests or to split or join the route path.

- Simple: do some arbitrary work

- Requests: generate action requests using a Route Module or the Rules engine

- Split: split the route path into one or more parallel branches

- Join: join one or more branches back together

- Sub Process: execute another route path inline

- Dynamic: generate a dynamic route path

Route Path	The path a document follows during the routing process. Consists of a set of route nodes and branches. The route path is defined as part of the *document type* definition.
Route Status	The status of a document in the course of its routing:

- Approved: These documents have been approved by all required reviewers and are waiting additional postprocessing.

- Cancelled: These documents have been stopped. The document's initiator can 'Cancel' it before routing begins or a reviewer of the document can cancel it after routing begins. When a document is cancelled, routing stops; it is not sent to another Action List.

- Disapproved: These documents have been disapproved by at least one reviewer. Routing has stopped for these documents.

- Enroute: Routing is in progress on these documents and an action request is waiting for someone to take action.

- Exception: A routing exception has occurred on this document. Someone from the Exception Workgroup for this Document Type must take action on this document, and it has been sent to the Action List of this workgroup.

- Final: All required approvals and all acknowledgements have been received on these documents. No changes are allowed to a document that is in Final status.

- Initiated: A user or a process has created this document, but it has not yet been routed to anyone's Action List.

- Processed: These documents have been approved by all required users, and is completed on them. They may be waiting for Acknowledgements. No further action is needed on these documents.

- Saved: These documents have been saved for later work. An author (initiator) can save a document before routing begins or a reviewer can save a document before he or she takes action on it. When someone saves a document, the document goes on that person's Action List.

Routed By User

The user who submits the document into routing. This is often the same as the Initiator. However, for some types of documents they may be different.

Routing

The process of moving a document through its route path as defined in its Document Type. Routing is executed and administered by the workflow engine. This process will typically include generating Action Requests and processing actions from the users who receive those requests. In addition, the Routing process includes callbacks to the Post Processor when there are changes in document state.

Routing Priority

A number that indicates the routing priority; a smaller number has a higher routing priority. Routing priority is used to determine the order that requests are activated on a route node with sequential activation type.

Routing Rule

A record that contains the data for the *Rule Attributes* specified in a *Rule Template* It is an instance of a Rule Template populated to determine the appropriate Routing. The Rule includes the Base Attributes, Required Extension Attributes, Responsible Party Attributes, and any Optional Extension Attributes that are declared in the Rule Template. Rules are evaluated at certain points in the routing process and, when they fire, can generate Action Requests to the responsible parties that are defined on them.

Technical considerations for a Routing Rules are:

- Configured via a GUI (or imported from XML)

- Created against a Rule Template and a Document Type

- The Rule Template and it's list of Rule Attributes define what fields will be collected in the Rule GUI

- Rules define the users, groups and/or roles who should receive action requests

- Available Action Request Types that Rules can route

  - Complete

  - Approve

  - Acknowledge

  - FYI

- During routing, Rule Evaluation Sets are "selected" at each node. Default is to select by Document Type and Rule Template defined on the Route Node

- Rules match (or 'fire') based on the evaluation of data on the document and data contained on the individual rule

- Examples

  - If dollar amount is greater than $10,000 then send an Approval request to Joe.

  - If department is "HR" request an Acknowledgment from the HR.Acknowledgers workgroup.

Rule Attribute	Rule attributes are a core KEW data element contained in a document that controls its Routing. It participates in routing as part of a Rule Template and is responsible for defining custom fields that can be rendered on a routing rule. It also defines the logic for how rules that contain the attribute data are evaluated.

Technical considerations for a Rule Attribute are:

- They might be backed by a Java class to provide lookups and validations of appropriate values.

- Define how a Routing Rule evaluates document data to determine whether or not the rule data matches the document data.

- Define what data is collected on a rule.

- An attribute typically corresponds to one piece of data on a document (i.e dollar amount, department, organization, account, etc.).

- Can be written in Java or defined using XML (with matching done by XPath).

- Can have multiple GUI fields defined in a single attribute.

Rule QuickLinks	A list of document groups with their document hierarchies and actions that can be selected. For specific document types, you can create the rule delegate.
Rule Template	A Rule Template serves as a pattern or design for the routing rules. All of the Rule Attributes that include both Required and _Optional_ are contained in the Rule Template; it defines the structure of the routing rule of FlexRM. The Rule Template is also used to associate certain Route Nodes on a document type to routing rules.

Technical considerations for a Rule Templates are:

- They are a composition of Rule Attributes

- Adding a 'Role' attribute to a template allows for the use of the Role on any rules created against the template

- When rule attributes are used for matching on rules, each attribute is associated with the other attributes on the template using an implicit 'and' logic attributes

- Can be used to define various other aspects to be used by the rule creation GUI such as rule data defaults (effective dates, ignore previous, available request types, etc)

S

Save

A workflow action button that allows the Initiator of a document to save their work and close the document. The document may be retrieved from the initiator's Action List for completion and routing at a later time.

Saved

A routing status indicating the document has been started but not yet completed or routed. The Save action allows the initiator of a document to save their work and close the document. The document may be retrieved from the initiator's action list for completion and routing at a later time.

Searchable Attributes

Attributes that can be defined to index certain pieces of data on a document so that it can be searched from the *Document Search screen*.

Technical considerations for a Searchable Attributes are:

- They are responsible for extracting and indexing document data for searching

- They allow for custom fields to be added to Document Search for documents of a particular type

- They are configured as an attribute of a Document Type

- They can be written in Java or defined in XML by using Xpath to facilitate matching

Secondary Delegation

The Secondary Delegate acts as a temporary backup Delegator who acts with the same authority as the primary Approver/the Delegator when the Delegator is not available. Documents appear in the Action Lists of both the Delegator and the Delegate. When either acts on the document, it disappears from both Action Lists.

Secondary Delegation is often configured for a range of dates and it must be registered in KEW or KIM to be in effect.

Service Registry

Displays a read-only view of all of the services that are exposed on the Service Bus and includes information about them (for example, IP Address, or Endpoint URL).

Simple Node

A type of node that can perform any function desired by the implementer. An example implementation of a simple node is the node that generates Action Requests from route modules.

SOA

An acronym for Service Oriented Architecture.

Special Condition Routing

This is a generic term for additional route levels that might be triggered by various attributes of a transaction. They can be based on the type of document, attributes of the accounts being used, or other attributes of the transaction. They often represent special administrative approvals that may be required.

Split Node

A node in the routing path that can split the route path into multiple branches.

Spring

The Spring Framework is an open source application framework for the Java platform.

State

Defines U.S. Postal Service codes used to identify states.

Status

On an Action List; also known as Route Status. The current location of the document in its routing path.

Submit	A workflow action button used by the initiator of a document to begin workflow routing for that transaction. It moves the document (through workflow) to the next level of approval. Once a document is submitted, it remains in 'ENROUTE' status until all approvals have taken place.
Superuser	A user who has been given special permission to perform Superuser Approvals and other Superuser actions on documents of a certain Document Type.
Superuser Approval	Authority given Superusers to approve a document of a chosen Route Node. A Superuser Approval action bypasses approvals that would otherwise be required in the Routing. It is available in Superuser Document Search. In most cases, reviewers who are skipped are not sent Acknowledge Action Requests.
Superuser Document Search	A special mode of Document Search that allows Superusers to access documents in a special Superuser mode and perform administrative functions on those documents. Access to these documents is governed by the user's membership in the Superuser Workgroup as defined on a particular Document Type.

T

Thread pool	A technique that improves overall system performance by creating a pool of threads to execute multiple tasks at the same time. A task can execute immediately if a thread in the pool is available or else the task waits for a thread to become available from the pool before executing.
Title	A short summary of the notification message. This field can be filled out as part of the Simple Message or Event Message form. In addition, this can be set by the programmatic interfaces when sending notifications from a client system.

This field is equivalent to the "Subject" field in an email. |

U

URL	An acronym for Uniform Resource Locator.
User	A person who can log in and use the application. This term is synonymous with "Principal" in KIM. "Whereas Entity Id represents a unique Person, Principal Id represents a set of login information for that Person."

V

Viewer	A user(s) who views a document during the routing process. This includes users who have action requests generated to them on a document.

W

Web Service Client	A type of client that connects to a standalone KEW server using Web Services.
Wildcard	A character that may be substituted for any of a defined subset of all possible characters.
Workflow	Electronic document routing, approval and tracking. Also known as Workflow Services or Kuali Enterprise Workflow (KEW). The Kuali infrastructure service

that electronically routes an e-doc to its approvers in a prescribed sequence, according to established business rules based on the e-doc content. See also *Kuali Enterprise Workflow*.

Workflow Engine   The component of KEW that handles initiating and executing the route path of a document.

Workflow QuickLinks   A web interface that provides quick navigation to various functions in KEW. These include:

- Quick EDoc Watch: The last five Actions taken by this user. The user can select and repeat these actions.

- Quick EDoc Search: The last five EDocs searched for by this user. The user can select one and repeat that search.

- Quick Action List: The last five document types the user took action with. The user can select one and repeat that action.

X

XML   See also *XML Ingester*.

                                   1. An acronym for Extensible Markup Language.

                                   2. Used for data import/export.

XML Ingester   A workflow function that allows you to browse for and upload XML data.

XML RuleAttribute   Similar in functionality to a RuleAttribute but built using XML only