# Kuali Rice 2.3.0-M3-SNAPSHOT Kuali Service Bus

# Table of Contents

# List of Figures

# List of Tables

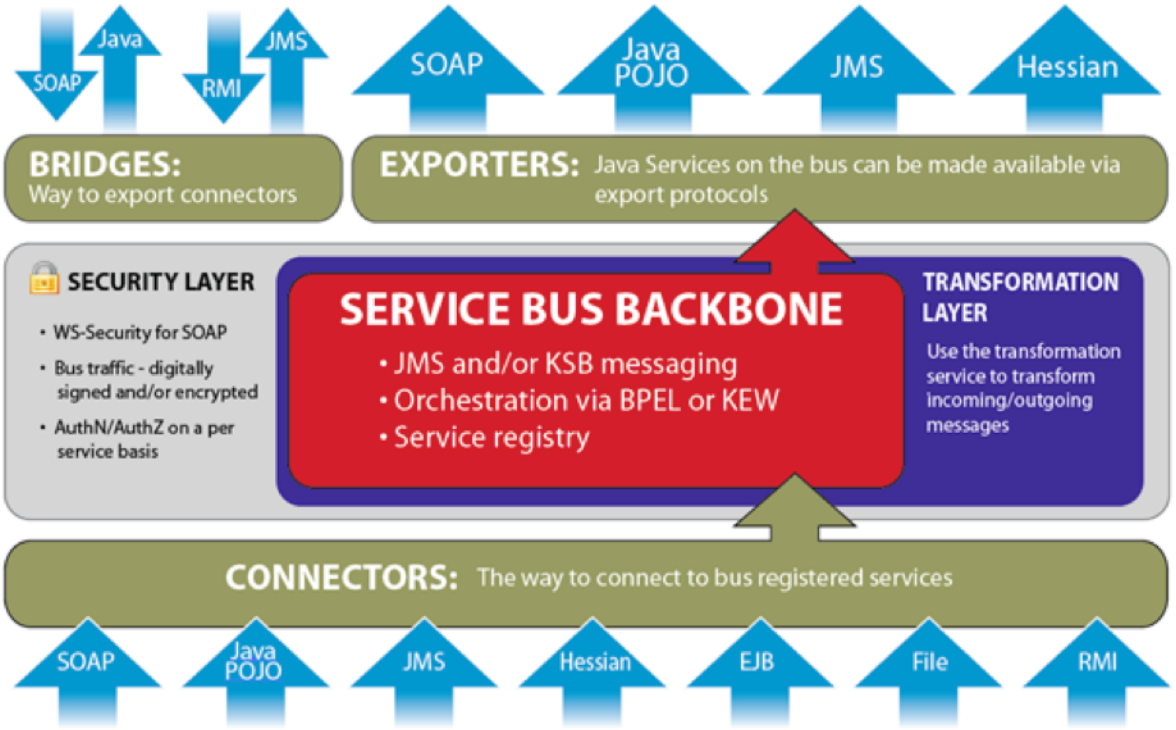
# Chapter 1. KSB Overview

## What is the Kuali Service Bus?

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.

**Figure 1.1. Kuali Service Bus**




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.

## Features

- **Transactional Asynchronous Messaging** - You can call services asynchronously to support a 'fire and forget' model of calling services. Messaging participates in existing JTA transactions, so that messages are not sent until the currently running transaction is committed and are not sent if the transaction is rolled back. You can increase the performance of service calling code because you are not waiting for a response.

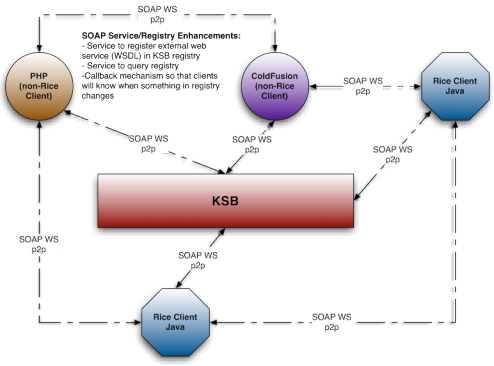
- **Synchronous Messaging** - Call any service on the bus using a request response paradigm.
- **Queue Style Messaging** - Supports executing Java services using message queues. When a message is sent to a queue, only one of the services listening for messages on the queue is given the message.
- **Topic Style Messaging** - Supports executing Java services using messaging topics. When a message is sent to a topic, all services that are listening for messages on the topic receive the message.
- **Quality of Service** - Determines how queues and topics handle messages that have problems. Time to live is supported, giving the message a configured amount of time to be handled successfully before exception handling is invoked for that message type. Messages can be given a number of retry attempts before exception handling is invoked. The delay separating each call increases. Exception handlers can be registered with each queue and topic for custom behavior when messages fail and Quality of Service limits have been reached.
- **Discovery** - Services are automatically discovered along the bus by service name. End-point URLs are not needed to connect to services.
- **Reliability** - Should problems arise, messages sent to services via queues or synchronous calls automatically fail-over to any other services bound to the same name on the bus. Services that are not available are removed from the bus until they come back online, at which time they will be rediscovered for messaging.
- **Persisted Callback** - Callback objects can be sent with any message. This object will be called each time the message is received with the response of the service (think topic as opposed to queue). In this way, we can deploy services for messaging that actually return values.
- **Primitive Business Activity Monitoring** - If turned on, each call to every service, including the parameters passed into that service, is recorded. This feature can be turned on and off at runtime.
- **Spring-Based Integration** - KSB is designed with Spring-based integration in mind. A typical scenario is making an existing Spring-based POJO available for remote asynchronous calls.
- **Programmatic Based Integration** - KSB can be configured programmatically if Spring configuration is not desired. Services can also be added and removed from the bus programmatically at runtime.

# 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.

# Overview of Supported Service Protocols

**Figure 1.2. Supported Service Protocols**




This drawing is conceptual and not representative of true deployment architecture. Essentially, the KSB is a registry with service-calling behavior on the client end (Java client). All policies and behaviors (async as opposed to sync) are coordinated on the client. The client offers some very attractive messaging features:

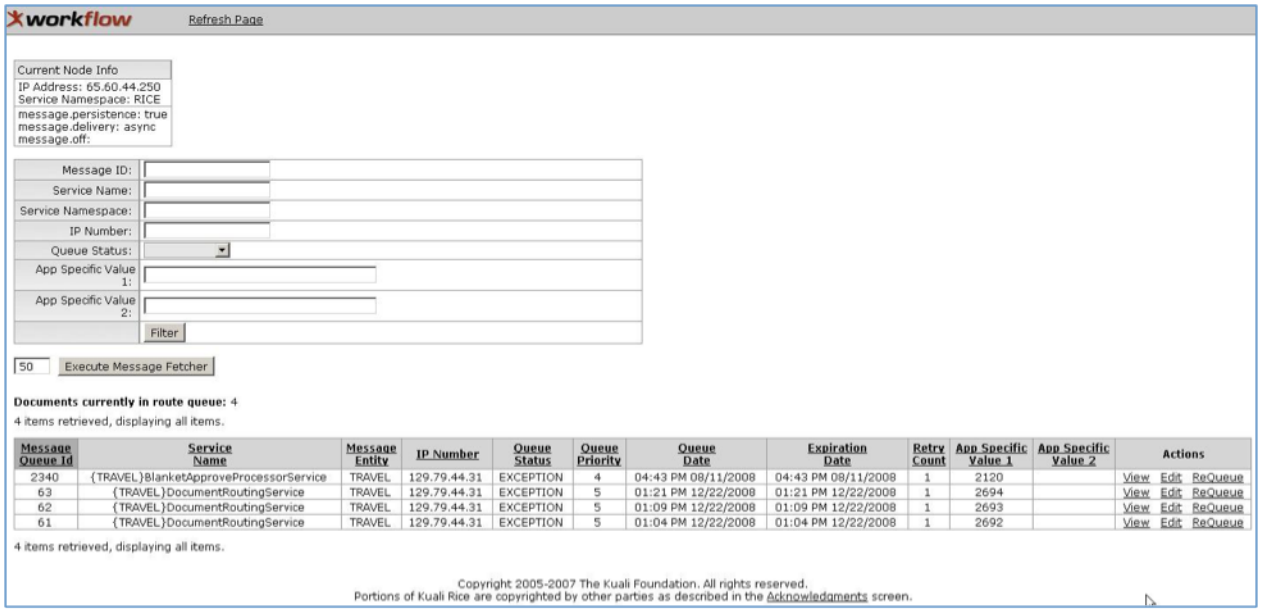
- **Synchronization** of message sending with currently running transaction (meaning all messages sent during a transaction are ONLY sent if the transaction is successfully committed)
- **Failover** - If a call to a service comes back with a 404 (or various other network-related errors), it will try 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.
- **Message Driven Service Execution** - Bind standard JavaBean services to messaging queues for message driven beans.

# Chapter 2. Message Queue

Use the Message Queue section to administer the KNS message queuing system. You can find it on the Administration menu.

It has three main sections: Current Node Info, the message filter and fetch section, and the Documents currently in route queue section.

**Figure 2.1. Message Queue: Documents Currently In Route**



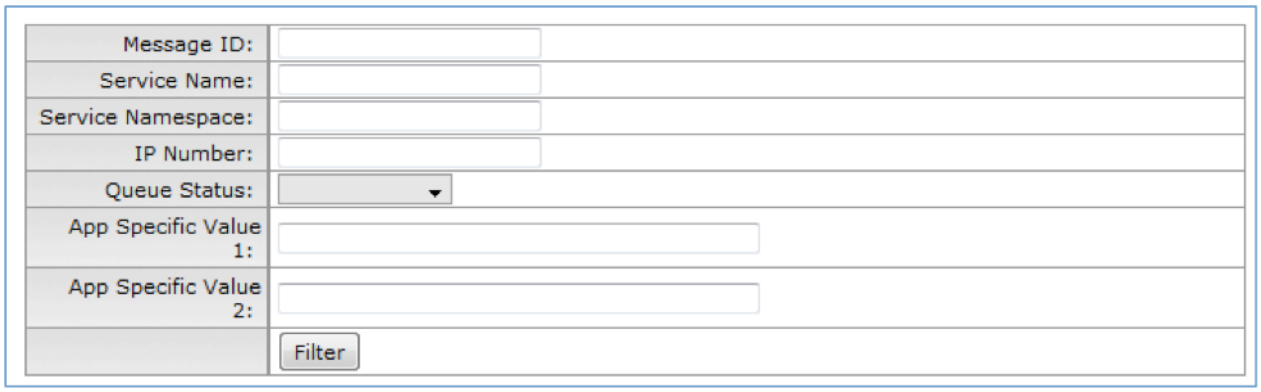

# Current Node Info

- **IP Address**: This value equals the IP address of the machine: Rice
- **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 in queue, 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 application interface (API) will 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.

# Message Filter and Fetch

The message filter and fetch section of the Message Queue screen lets you search for, filter, and/or isolate messages in the Documents in route queue. To use the Message Filter section, enter your criteria and click the Filter button:

### Figure 2.2. Message Filter Screen



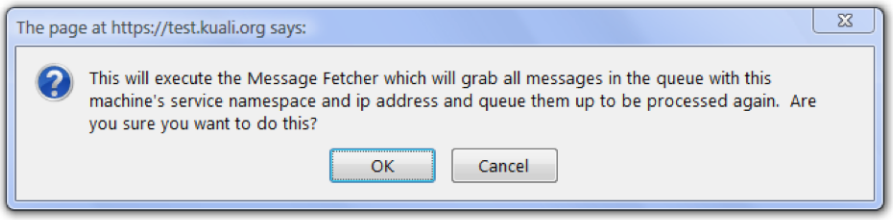

### Table 2.1. Message Filter Screen: Attributes

| Field | Description |
|---|---|
| Message ID | A unique 5-digit message queue identification number |
| Service Name | The name of the service |
| Application ID | The service container's identifier |
| IP Number | The message initiator's IP address |
| Queue Status | You can sort documents by the queue status. The queue status may be:<br>• QUEUED: The message is waiting for a worker thread to pick it up<br>• ROUTING: A worker is currently working on the message.<br>• EXCEPTION: There is a problem with the message and the route manager will ignore it. EXCEPTION status is typically set manually by the administrator to suspend a route queue entry until a problem can be diagnosed. |
| App Specific Value 1 | The specific value of a document |
| App Specific Value 2 | The specific value of a document |
| Filter Button | Click to execute the search filter |

The Execute Message Fetcher button retrieves all the messages in the route queue. You can adjust the number of messages requested by entering a number in the field left of the button.

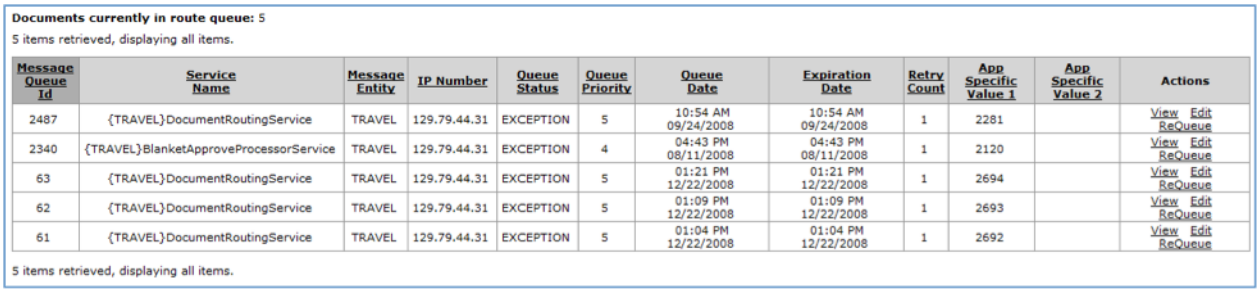
When you click the Execute Message Fetcher button, a dialog box appears, confirming that you want to execute this command:

### Figure 2.3. Execute Message Filter: Confirmation Screen




KSB displays the results of a search and/or filter at the bottom of the page in the Documents currently in route queue table.

### Figure 2.4. Documents In Route Queue



**Documents currently in route queue:** 5

5 items retrieved, displaying all items.

| Message Queue Id | Service Name | Message Entity | IP Number | Queue Status | Queue Priority | Queue Date | Expiration Date | Retry Count | App Specific Value 1 | App Specific Value 2 | Actions |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2487 | {TRAVEL}DocumentRoutingService | TRAVEL | 129.79.44.31 | EXCEPTION | 5 | 10:54 AM 09/24/2008 | 10:54 AM 09/24/2008 | 1 | 2281 | | View Edit ReQueue |
| 2340 | {TRAVEL}BlanketApproveProcessorService | TRAVEL | 129.79.44.31 | EXCEPTION | 4 | 04:43 PM 08/11/2008 | 04:43 PM 08/11/2008 | 1 | 2120 | | View Edit ReQueue |
| 63 | {TRAVEL}DocumentRoutingService | TRAVEL | 129.79.44.31 | EXCEPTION | 5 | 01:21 PM 12/22/2008 | 01:21 PM 12/22/2008 | 1 | 2694 | | View Edit ReQueue |
| 62 | {TRAVEL}DocumentRoutingService | TRAVEL | 129.79.44.31 | EXCEPTION | 5 | 01:09 PM 12/22/2008 | 01:09 PM 12/22/2008 | 1 | 2693 | | View Edit ReQueue |
| 61 | {TRAVEL}DocumentRoutingService | TRAVEL | 129.79.44.31 | EXCEPTION | 5 | 01:04 PM 12/22/2008 | 01:04 PM 12/22/2008 | 1 | 2692 | | View Edit ReQueue |

5 items retrieved, displaying all items.

# Documents Currently in Route Queue

### Table 2.2. Documents Currently in Route Queue: Attributes

| Field | Description |
|---|---|
| Message Queue ID | A unique 5-digit message queue identification number. This is the same as the Message ID in the Message Filter section. |
| Service Name | The name of the service |
| Message Entity | |
| IP Number | The message initiator's IP address |
| Queue Status | You can sort documents by the queue status. The queue status may be:<br>• QUEUED: The message is waiting for a worker thread to pick it up<br>• ROUTING: A worker is currently working on the message.<br>• EXCEPTION: There is a problem with the message and the route manager will ignore it. EXCEPTION status is typically set manually by the administrator to suspend a route queue entry until a problem can be diagnosed. |
| Queue Priority | The priority of the entry in the queue. Entries with the smallest number are processed first. |
| Queue Date | The date on which the queue entry should be processed. If the queue checker runs and discovers entries that have a queue date that are equal to or earlier than the current time, it processes them. The approximate time at which this screenshot was taken 4:53 PM. |
| Expiration Date | |
| Retry Count | |
| App Specific Value 1 | |
| App Specific Value 2 | |

| Field | Description |
|---|---|
| Actions | Click a link in this field to:<br>• **View**: View the detail message report<br>• **Edit**: Edit the settings of a Message Entry<br>• **ReQueue**: Enforce the routing process |

# View

When you click View in the Actions menu, KSB displays information about that message. Most of the initial information is the same as that displayed in the Documents currently in route queue table. Additional information on the View screen:

- **Message**

### Table 2.3. Message: Attributes

| Field | Description |
|---|---|
| Application ID: | The service container's identifier |
| Method Name: | |

- **Payload**

### Table 2.4. Payload: Attributes

| Field | Description |
|---|---|
| Payload Class | The class of the Payload |
| Method Name | The name of the method used in this document |
| ignoreStoreAndForward | A true and false indicator that ignores the store functions and forwards the message |
| ServiceInfo.messageEntryId | A unique 4-digit message entry identification number |
| ServiceInfo.ServiceNamespace | The application |
| ServiceInfo.serverIp | The server's IP address |
| ServiceInfo.ServiceName | The name of the service |
| ServiceInfo.endpointUrl | The web address of the service |
| ServiceInfo.queue | A true and false indicator that activates the queue or topic function:<br>• "True" uses the Queue method, which sends the message to one contact at a time<br>• "False" uses the Topic method, which sends the message to all contacts at once |
| ServiceInfo.alive | A true and false indicator that shows the activity state of the document |
| ServiceInfo.priority | The priority of the entry for execution. Entries with the smallest number are processed first |
| ServiceInfo.retryAttempts | How many times KSB will try to resend the message |
| ServiceInfo.millisToLive | An expiration indicator:<br>• 1 means the message never expires |
| ServiceInfo.messageExceptionHandler | This provides a reference the service can use to call back. |
| ServiceInfo.serviceclass | The name of the service class |
| ServiceInfo.busSecurity | A true and false indicator that assigns the security function |
| ServiceInfo.credentialsType | The credential type of the document |
| Arguments | The argument of this document |

- **Edit**

When you click **Edit** in the **Actions** menu, KSB displays the editable fields for that message. Fields on the Edit screen:

### Table 2.5. Edit Screen: Attributes

| Field | Description |
|---|---|
| Queue Priority | Change the queue priority by entering a positive number. A smaller number has higher priority for execution. |
| Queue Status | Change the status to Queued, Routing, or Exception. |
| Retry Count | Change the number of times KSB will retry. |
| IP Number | Change the initiator's IP address. |
| Service Name | Change the name of the service. |
| Message Entity | Change the message entity. |
| Method Name | Change the method. |
| App Specific Value 1 | Change the information for the specific value 1. |
| App Specific Value 2 | Change the information for the specific value 2. |

Functional links on the Edit page:

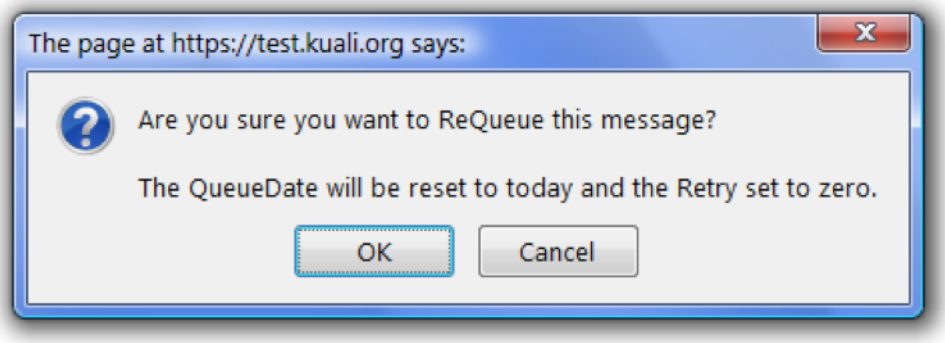
### Table 2.6. Edit Screen: Links

| Field | Description |
|---|---|
| Save Changes | Save the information you just changed. |
| Save Changes and Resubmit | Save the information you changed and resubmit the message. |
| Save and Forward | Save the message and send it to the next contact. |
| Delete | Delete the message. |
| Reset | Reload the previous settings. This undoes the changes that you made on this screen, as long as you haven't yet saved them. |
| Clear Message | Clear all information fields on this page. |

- **ReQueue**

When you click ReQueue in the Actions menu, KSB displays this pop-up message:

### Figure 2.5. Requeue Documents: Confirmation Screen

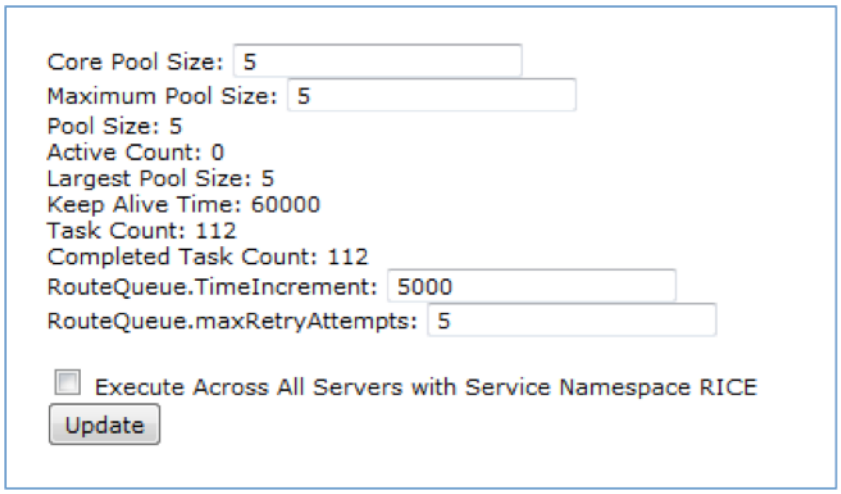


# Chapter 3. Thread Pool

**Thread pool** is a feature that improves overall system performance by creating a pool of threads which can be independently used by the system to execute multiple tasks at the same time. A task can execute immediately if there is a thread in the pool that is available. If no thread is available, the task waits for a thread to become available from the pool before executing.

The **Thread Pool** screen is accessed from the **Administration** menu. It tells you the current state of the Thread Pool and allows you to change four parameters for the Thread Pool. The core pool size, the maximum pool size, the RouteQueue.TimeIncrement and the RouteQueue.maxRetryAttempts.

### Figure 3.1. Thread Pool Administration Page




### Table 3.1. Thread Pool: Attributes

| Field | Description |
|---|---|
| Core Pool Size | A positive number equal to the core number of threads in the pool |
| Maximum Pool Size | A positive number equal to the maximum number of threads in the pool; when the Core Pool Size is larger than the Maximum Pool Size, Maximum Pool Size automatically sets the pool size equal to the Core Pool Size |
| Pool Size | The current number of threads in the pool |
| Active Count | The approximate number of threads that are actively executing tasks |
| Largest Pool Size | Maximum number of threads allowed in the Thread Pool |
| Keep Alive Time | The amount of time which threads in excess of the core pool size may remain idle before being terminated; measured in milliseconds; for example, 60,000 milliseconds = 60 seconds |
| Task Count | Number of tasks that have been scheduled for execution |
| Completed Task Count | Number of tasks that have completed execution |
| Execute Across All Servers with Application ID RICE | When you click this checkbox, then click the **Update** button, the update is applied across all servers. |
| Update button | Click the **Update** button to execute the changes you entered in the editable fields above. |

# Chapter 4. Service Registry

The Service Registry lists published and temporary services that are available for the local machine. You cannot configure the service registry here; this is only information about the registry.

Display this page by clicking the **Service Registry** link on the Rice **Administration** page.

At the top of the page, the **Current Node Info** table shows the settings and configuration of the local machine:

The returned table of services is divided into three sections:

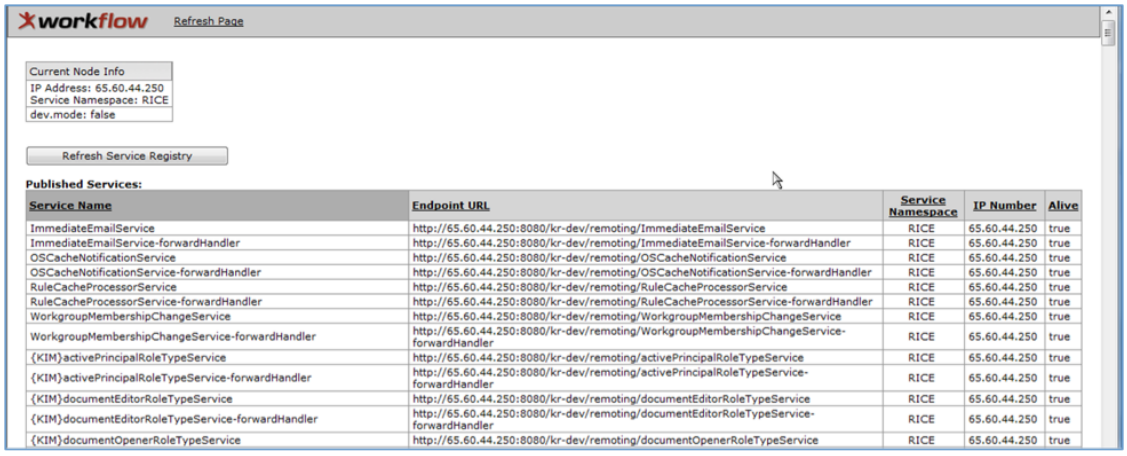
1. Published Services: Services in use by the local machine

2. Published Temp Services: Temporary services that are the result of Object Remoting. For more information about Object Remoting, please refer to the Object Remoting section of the KSB portion of the Technical Reference Guide.

3. All Registry Services

This screen print shows the top of a Service Registry page, with the **Current Node Info** table and the beginning of the **Published Services** table, as well as the refresh link and button:

### Figure 4.1. Service Registry




| Service Name | Endpoint URL | Service Namespace | IP Number | Alive |
|---|---|---|---|---|
| ImmediateEmailService | http://65.60.44.250:8080/kr-dev/remoting/ImmediateEmailService | RICE | 65.60.44.250 | true |
| ImmediateEmailService-forwardHandler | http://65.60.44.250:8080/kr-dev/remoting/ImmediateEmailService-forwardHandler | RICE | 65.60.44.250 | true |
| OSCacheNotificationService | http://65.60.44.250:8080/kr-dev/remoting/OSCacheNotificationService | RICE | 65.60.44.250 | true |
| OSCacheNotificationService-forwardHandler | http://65.60.44.250:8080/kr-dev/remoting/OSCacheNotificationService-forwardHandler | RICE | 65.60.44.250 | true |
| RuleCacheProcessorService | http://65.60.44.250:8080/kr-dev/remoting/RuleCacheProcessorService | RICE | 65.60.44.250 | true |
| RuleCacheProcessorService-forwardHandler | http://65.60.44.250:8080/kr-dev/remoting/RuleCacheProcessorService-forwardHandler | RICE | 65.60.44.250 | true |
| WorkgroupMembershipChangeService | http://65.60.44.250:8080/kr-dev/remoting/WorkgroupMembershipChangeService | RICE | 65.60.44.250 | true |
| WorkgroupMembershipChangeService-forwardHandler | http://65.60.44.250:8080/kr-dev/remoting/WorkgroupMembershipChangeService-forwardHandler | RICE | 65.60.44.250 | true |
| {KIM}activePrincipalRoleTypeService | http://65.60.44.250:8080/kr-dev/remoting/activePrincipalRoleTypeService | RICE | 65.60.44.250 | true |
| {KIM}activePrincipalRoleTypeService-forwardHandler | http://65.60.44.250:8080/kr-dev/remoting/activePrincipalRoleTypeService-forwardHandler | RICE | 65.60.44.250 | true |
| {KIM}documentEditorRoleTypeService | http://65.60.44.250:8080/kr-dev/remoting/documentEditorRoleTypeService | RICE | 65.60.44.250 | true |
| {KIM}documentEditorRoleTypeService-forwardHandler | http://65.60.44.250:8080/kr-dev/remoting/documentEditorRoleTypeService-forwardHandler | RICE | 65.60.44.250 | true |
| {KIM}documentOpenerRoleTypeService | http://65.60.44.250:8080/kr-dev/remoting/documentOpenerRoleTypeService | RICE | 65.60.44.250 | true |

To update the list of published services, use either the **Refresh Page** link in the header at the top of the page or the "Refresh Service Registry" button.

This screen print shows the point on a Service Registry page where KSB displays a notation that there are no published temporary services and the beginning of the **All Registry Services** table:

## Figure 4.2. Service Registry Results

**Published Services:**

| Service Name | Endpoint URL |
|---|---|
| {http://rice.kuali.org/core/v2_0}componentService | http://localhost:8080/bar/remoting/soap/core/v2_0/componentService |
| {http://rice.kuali.org/core/v2_0}coreServiceCacheAdminService | http://localhost:8080/bar/remoting/soap/core/v2_0/coreServiceCacheAdminService |
| {http://rice.kuali.org/core/v2_0}namespaceService | http://localhost:8080/bar/remoting/soap/core/v2_0/namespaceService |
| {http://rice.kuali.org/core/v2_0}parameterRepositoryService | http://localhost:8080/bar/remoting/soap/core/v2_0/parameterRepositoryService |
| {http://rice.kuali.org/core/v2_0}styleRepositoryService | http://localhost:8080/bar/remoting/soap/core/v2_0/styleRepositoryService |
| {http://rice.kuali.org/ken/v2_0}sendNotificationService | http://localhost:8080/bar/remoting/soap/ken/v2_0/sendNotificationService |
| {http://rice.kuali.org/kew/v2_0}actionInvocationQueue | http://localhost:8080/bar/remoting/soap/kew/v2_0/actionInvocationQueue |
| {http://rice.kuali.org/kew/v2_0}actionListCustomizationHandlerService | http://localhost:8080/bar/remoting/soap/kew/v2_0/actionListCustomizationHandlerService |
| {http://rice.kuali.org/kew/v2_0}actionListService | http://localhost:8080/bar/remoting/soap/kew/v2_0/actionListService |
| {http://rice.kuali.org/kew/v2_0}backdoorRestrictionPermissionTypeService | http://localhost:8080/bar/remoting/soap/kew/v2_0/backdoorRestrictionPermissionTypeService |
| {http://rice.kuali.org/kew/v2_0}documentAttributeIndexingQueue | http://localhost:8080/bar/remoting/soap/kew/v2_0/documentAttributeIndexingQueue |
| {http://rice.kuali.org/kew/v2_0}documentOrchestrationQueue | http://localhost:8080/bar/remoting/soap/kew/v2_0/documentOrchestrationQueue |
| {http://rice.kuali.org/kew/v2_0}documentProcessingQueue | http://localhost:8080/bar/remoting/soap/kew/v2_0/documentProcessingQueue |
| {http://rice.kuali.org/kew/v2_0}documentRefreshQueue | http://localhost:8080/bar/remoting/soap/kew/v2_0/documentRefreshQueue |
| {http://rice.kuali.org/kew/v2_0}documentSearchCustomizationHandlerService | http://localhost:8080/bar/remoting/soap/kew/v2_0/documentSearchCustomizationHandlerService |

Please note, you may have permissions that allow you to click on a row's Endpoint URL to view the WSDL fiels assoicated with the given service. In Internet Explorer or Firefox, this WSDL will be displayed normally in a separate window. In Google Chrome or Safari, however, you will need to click the link then right click to view the frame source to see the WSDL due to current restrictions in Chrome and Safari.

# Chapter 5. Pessimistic Locking

## Default Pessimistic Locking

### Warning

Only Transactional Documents may use the default Pessimistic Locking implementation.

To lock a document via the default Pessimistic Locking mechanism means that the document is locked by a user prior to any changes the user may perform. The traditional setup of documents in Rice is to lock them using Optimistic Locking where two users may edit a document at the same time. However, the first user to take an action that will save the document will 'win', and the second user will see an error saying that the document was edited by another user. For Pessimistic Locking, the first user who has edit privileges will get a lock on the document, and any subsequent users who should have edit privileges on the document, who try to view the document, will only get read-only access to the document, until the first user's lock is 'released'.

### Note

Pessimistic Locking is used in conjunction with standard Rice Optimistic Locking. Currently there is no way to use Pessimistic Locking instead of the default Optimistic Locking.

There are two places in Rice where Pessimistic Locks can be used:

- Locking for User Entry - locks are created by a user who has some type of entry privileges on the document
- Locking for Workflow Processing - locks are created when a workflow process is begun

## Locking for User Entry

The default implementation for locking a document for user entry tells the system to place a lock on a document if a user attempts to view it and that user has one or more 'entry' type edit modes as defined by the document's Document Authorizer class. Once the lock is placed, any other user who should have 'entry' privileges on the document will not be allowed to do so until the lock by the first user is released.

### Note

If the Transactional Document that will be using Pessimistic Locking has a custom Document Authorizer class and uses custom edit modes returned by the getEditMode(Document, UniversalUser) method, the custom authorizer class should also override the method isEntryEditMode(Map.Entry). See #Defining 'Entry' Edit Modes below.

## Document Configuration - Document

To enable Pessimistic Locking on a document the attribute 'usePessimisticLocking' must be set to 'true' in the transactional document's entry.

**Example:**

```
<dictionaryEntry>
    <transactionalDocument>
```

```
        ...
        <usePessimisticLocking>true</usePessimisticLocking>
        ...
    </transactionalDocument>
</dictionaryEntry>
```

# Customizing

For extremely complex customization that goes beyond what may be described in this document a client developer can look at the javadocs of the **`org.kuali.core.document.authorization.DocumentAuthorizerBase`** class paying special attention to the methods below:

- isLockRequiredByUser(Document, Map, UniversalUser)
- isEntryEditMode(Map.Entry)
- getEditModeWithEditableModesRemoved(Map)
- getEntryEditModeReplacementMode()
- createNewPessimisticLock(Document, Map, UniversalUser)

The completely override the lock handling the Document Authorizer method establishLocks(Document, Map, UniversalUser) can be overriden.

# Defining 'Entry' Edit Modes

If the Transactional Document that will be using Pessimistic Locking has a custom Document Authorizer class and uses custom edit modes returned by the **`getEditMode(Document, UniversalUser)`** method, the custom authorizer class should also override the method **`isEntryEditMode(Map.Entry)`** . If the entry parameter passed in is defined as a valid 'entry' mode then the method should return true.

# Using custom Lock Descriptors

The default Pessimistic Lock implementation does not use Lock Descriptors so only one person may have a lock on a single document at any one time. If something more custom is required Lock Descriptors can be used. The default implementation of a document that uses Pessimistic Locking and custom Lock Descriptors is that once a single user establishes a lock on a certain document with a certain lock descriptor... no other user can create a lock on that document with that descriptor. If another user needs that lock created they will have read only access on the document until the other user releases their lock.

**Example**

As an example, think of a document that has both an Delivery section and a Billing section. Perhaps a user 'fred' has access to edit the Delivery section but not the Billing section. Likewise, a user 'francine' has access to edit the Billing section but not the Delivery section. In this case it would be possible for both 'francine' and 'fred' to each have a lock on a single document since the data they have editable is mutually exclusive from the other. In this example 'fred' could have a Pessimistic Lock with a descriptor 'Delivery' while 'francine' could have a Pessimistic Lock with a 'Billing' descriptor.

To use lock descriptors the client application document should implement a custom Document Authorizer class if not done already (see Authorizers - Client Developer Guide (0.9.3) for more information). The authorizer class should override the useCustomLockDescriptors() method to return true. The method

getCustomLockDescriptor(Document, Map, UniversalUser) must also be overriden to return the value of the desired lock descriptor. It's up to the client to determine how to set these and what values to use.

# Locking for Workflow Processing

The default implementation for locking a document for processing by Workflow tells the system to place a lock on a document once a Workflow action is taken if that Workflow action is not contained in a list (see Default Workflow Actions that Don't Require Locks). The default user that will 'own' the lock will be the Rice System User. Once the lock is placed, any other user who should have 'entry' privileges on the document will not be allowed to do so until the lock is released. Locks for Workflow processing are released once the Workflow process completes successfully.

### Note

If a document that has a Pessimistic Lock for Workflow is not successfully processed and goes into Exception Routing, the document will stay locked by the Workflow process.

# Default Workflow Actions that Don't Require Locks

The following actions in Workflow will not set up a Pessimistic Lock for the coinciding process:

- Save
- Acknowledge
- Clear FYI
- Disapprove
- Canceled
- Log on Document

# Document Configuration - Workflow

To enable Pessimistic Locking for Workflow operations on a document the attribute **`useWorkflowPessimisticLocking`** must be set to 'true' in the transactional document's entry.

**Example**

```
<dictionaryEntry>
    <transactionalDocument>
        ...
        <useWorkflowPessimisticLocking>true</useWorkflowPessimisticLocking>
        ...
    </transactionalDocument>
</dictionaryEntry>
```

# Customizing

The Pessimistic Locking mechanism for Workflow processes has lock creation and lock releasing points that exist in a document's post processor methods. Specifically the method

**`doActionTaken(ActionTakenEventVO)`** in the **`DocumentBase`** class is used to create locks while the method **`afterWorkflowEngineProcess(boolean)`** in the same class is used to release locks. If a document overrides either of these methods or does not use the standard KualiPostProcessor implementation, the client will need to use the DocumentBase methods code in whatever method they implement if they would like Pessimistic Locking for Workflow.

# Using a Custom Lock Owner

The default owner of a Pessimistic Lock created for a Workflow process is the Rice System User. To change that a client can implement a custom Document Authorizer class and override the method **`getWorkflowPessimisticLockOwnerUser()`**. This method is used to get the lock owner for lock creation but also will be used to release the lock at the conclusion of the Workflow process. If a non-static user will be used a client may need to override the method **`releaseWorkflowPessimisticLocking(Document)`** to handle special cases.

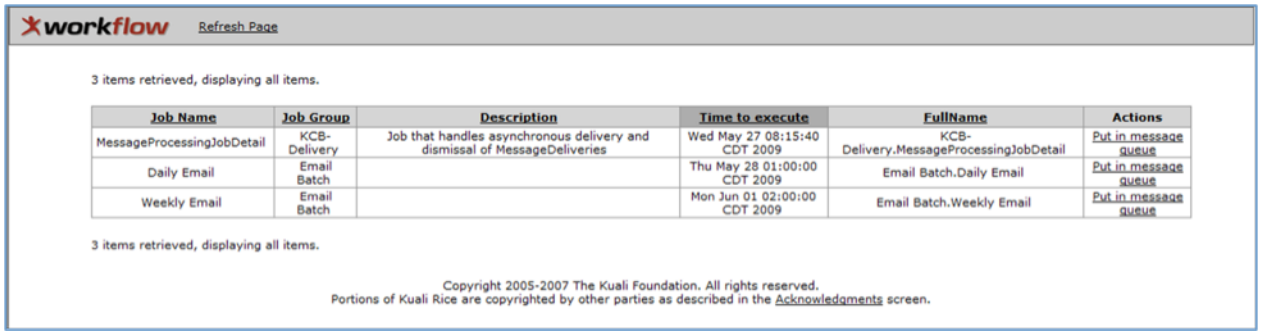
# Chapter 6. Quartz

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.

You can inject a custom quartz scheduler if the application is already running one. See the Technical Reference Guide for KSB, Configuring Quartz for KSB for more information.

Quartz is also known as the **Exception Routing Queue**.

### Figure 6.1. Exception Routing Queue




When you click the Quartz link on the Kuali Rice Portal Administration page, KSB displays the screen shown above. The contents of the table can be sorted in ascending or descending order by clicking on a column title. This technique works for all columns except Actions. The table contains this information on each job that is scheduled:

### Table 6.1. Exception Routing Queue: Attributes

| Field | Description |
|---|---|
| Job Name | Unique name for the job |
| Job Group | Classification of the job |
| Description | Text description of what this job does |
| Time to execute | The scheduled date and time for the job to occur |
| FullName | A more descriptive Job Name |
| Actions | Put in message queue effectively is a button that takes that message out of quartz and sends it back into the KSB to be retried without waiting until the scheduled time. |

# Chapter 7. Security

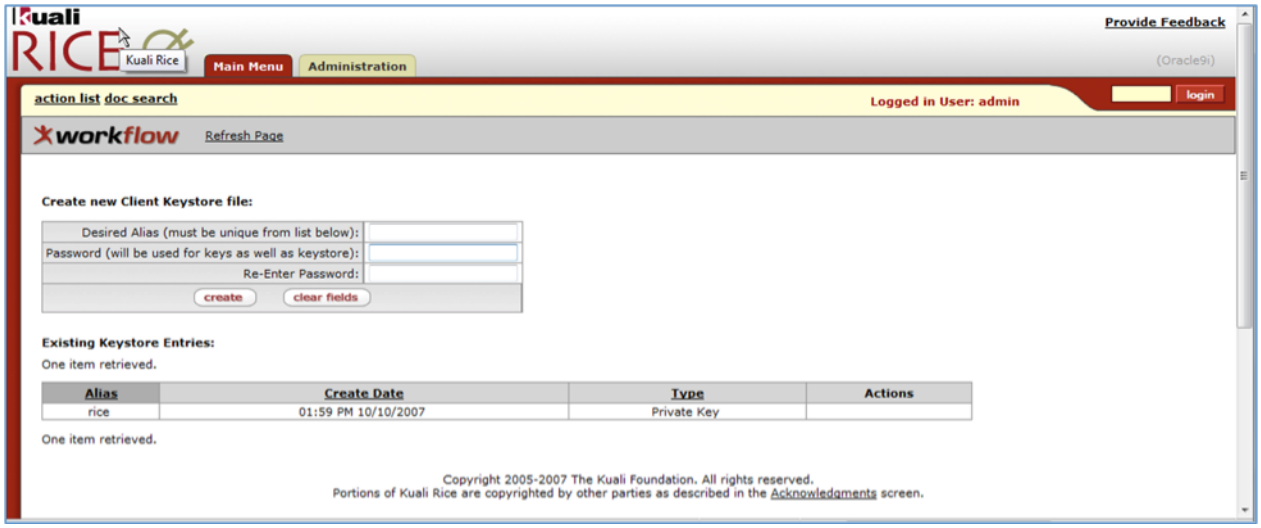
## Overview

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.

## Security Management

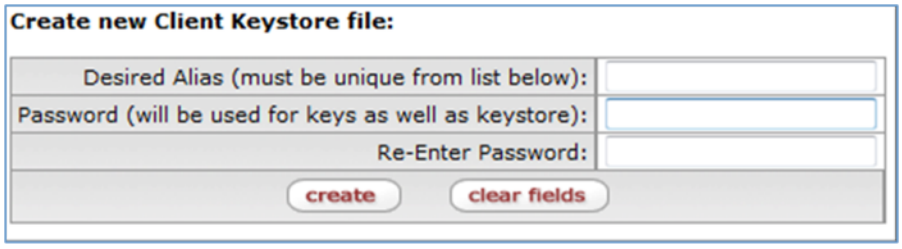
For client applications to consume secured services hosted from a standalone Rice server, the implementer must generate a keystore in KSB. KSB security relies on the creation of a keystore using the JVM keytool.

**Figure 7.1. Create Keystore**




To create a new Client Keystore file, complete all three fields and click the create button that is just below the fields:

**Figure 7.2. Create Keystore: File Section**




The **Desired Alias** (name for the new keystore you are creating) must be unique among your keystores. KSB automatically displays a list of existing Keystore entries for your reference below the *Create new Client Keystore file* table. The data in this list can be sorted in ascending or descending order by clicking the column heading for any column except *Actions*.

**Figure 7.3. Create Keystore: Existing Keystore Section**

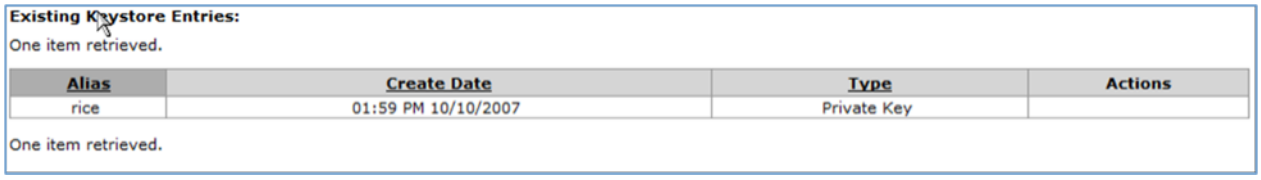



**Table 7.1. Existing Keystore Entries: Attributes**

| Field | Description |
|---|---|
| Alias | Keystore name |
| Create Date | Date and time the keystore was created |
| Type | The type of keystore |
| Actions | |

# 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>
```

# Chapter 8. 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.

# Chapter 9. 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-2.3-site-deploy/src/test/clients/
TestClient1</param>
    <param name="client2.location">/var/lib/jenkins/workspace/rice-2.3-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-2.3-site-deploy/src/test/resources/keystore/
ricekeystore</param>
    <param name="keystore.location">/var/lib/jenkins/workspace/rice-2.3-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 9.1. 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 |
| 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.

# Chapter 10. 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.

# Chapter 11. 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 11.1. 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. |
| 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 |

| property | required | default | description |
|---|---|---|---|
| | | | 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.<br><br>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.<br><br>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.

# Chapter 12. 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.

# Chapter 13. 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.

# Chapter 14. 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.

# Chapter 15. 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.

# Chapter 16. Object Remoting

As of Rice 2.0, Object remoting support has been removed.

# Chapter 17. 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.core.framework.resourceloader.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 17.1. 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 |
| 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) |

| Property | Description | Required |
|---|---|---|
| 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 17.2. 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 17.3. 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.core.framework.resourceloader.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

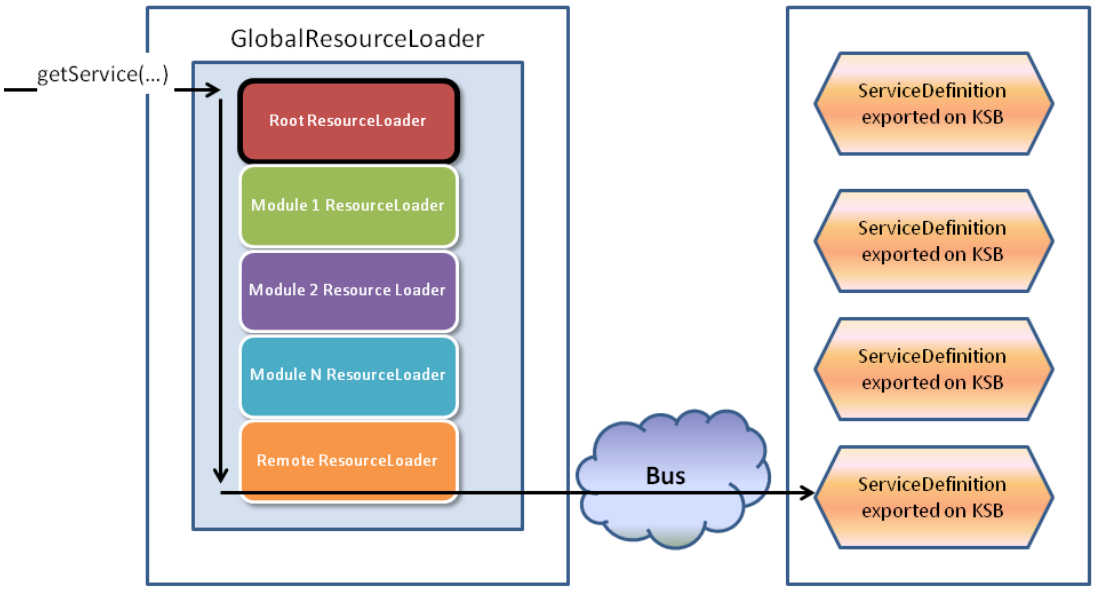
# Chapter 18. The ResourceLoader Stack

## 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 18.1. 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>
```

# Chapter 19. 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);
```

# Chapter 20. KSB Parameters

Here is a comprehensive set of configuration parameters used to configure the Kuali Service Bus.

# Core Parameters

### Table 20.1. 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.<br><br>*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 |
| 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

# Chapter 21. 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):

```
<!-- 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:

```
// … 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.

# Chapter 22. Using the KSB with bus security

### Warning

The information in this section is under development.

The Kuali Service Bus (KSB) includes web services already installed with a base Rice implementation. These services use WS-Security requiring the digital signing of each request providing a Signature and digest of the request. The digital signing ensures that the client application has the proper credentials to access the service.

# Rice Services

This documentation will illustrate how to interact with the a base KSB using the expected bus security. The examples will include a SoapUI client and a Java client application. Both of these examples can be used with the demo Rice implementation available at: http://demo.rice.kuali.org/portal.do

## Base Rice services

The list of Rice services is available from the Rice Administration menu:

- http://demo.rice.kuali.org/portal.do

  username: **admin**

- **Administration** tab
- **Service Bus** link from the Service Bus section

### Note

The demo Rice server is using the http protocol. The KSB Bus Security uses the keystore to produce a digest and digitally sign the request, but it does not encrypt the request. A production system should use the https protocol so that the request is encrypted as part of the transport.

## CampusService

For simplicity's sake the **CampusService.findAllCampuses** method will serve as the example. There are no query parameters and the answering web service response XML is easily human readable.

From the list of services on the demo Rice server, the CampusService has the following attributes:

- ServiceName:

```
{http://rice.kuali.org/location/v2_0}campusService
```

- Endpoint URL:

```
http://demo.rice.kuali.org/remoting/soap/location/v2_0/campusService
```

- Type: SOAP

  Because the type is SOAP, the WSDL can be found at http://demo.rice.kuali.org/remoting/soap/location/v2_0/campusService?wsdl

# Bus Security

The base SOAP services published on the KSB all rely on the use of the Rice Java Keystore included with the base Rice implementation. It is imperative that the client applications use the **same rice.keystore file** that the server is using. The Rice Administration menu provides an interface where an alias can be added to a new keystore and that new keystore is downloaded by your browser. See the notes on why this approach does not work for newly added aliases.

See this for information on obtaining a rice.keystore.

# Usage Examples

When consuming SOAP webservices with Java, two approaches are typical: a SoapUI client with tests and assertions, and a Java client application.

**SoapUI**

- Using SoapUI with rice.keystore bus security

**Java client application**

- Java client application using rice.keystore bus security

**SOAP request**

- What does the SOAP request look like, anyway?

# Obtaining rice.keystore

**Kuali Rice Java Keystore (jks)**

If you already have a running Rice instance, chances are you are already using, or have configured, a Rice keystore.

The links provided here are for convenience in finding the latest default keystore from source.

**Kuali locations**

- https://wiki.kuali.org/display/KULFOUND/Subversion
- https://svn.kuali.org/repos
- https://svn.kuali.org/repos/rice/trunk/rice-middleware/security/rice.keystore

# Using the KSB with bus security - new keystore aliases

Rice provides an interface for creating new aliases and producing new keystore files.

However, both the server and the client must be using the same **PrivateKeyEntry** or **trustedCertEntry**. These are identified inside of your keystore by the **owner, issuer,** and **serial number** values. So, in using this interface to create a new alias which produces a new keystore, you will not be able to use the new alias since the new keystore will not be used by the server. You can use the new keystore but only with the alias (representing the cert) that the server is expecting - namely 'rice'.

**keytool inspection of keystore**

```
$ keytool -list -v -keystore rice.keystore
Enter keystore password:

Keystore type: JKS
Keystore provider: SUN

Your keystore contains 1 entry

Alias name: rice
Creation date: Oct 10, 2007
Entry type: PrivateKeyEntry
Certificate chain length: 1
Certificate[1]:
Owner: CN=rice
Issuer: CN=rice
Serial number: 470d1315
Valid from: Wed Oct 10 10:59:49 PDT 2007 until: Sat Sep 22 10:59:49 PDT 2018
Certificate fingerprints:
         MD5:  53:73:B3:E6:39:56:73:AA:98:D4:A9:2D:C6:36:A2:DB
         SHA1: 86:4B:D2:54:39:D8:9B:B2:97:A9:B3:1A:32:B3:1F:12:83:A5:1F:4F
         Signature algorithm name: MD5withRSA
         Version: 1


*******************************************
*******************************************
```

### Note

The Issuer and Serial number are used in the digital signing of the SOAP Request. Here the Serial number is represented in hex, in the SOAP Request it is represented in base 10.

# KSB SoapUI Client

SoapUI is an excellent client for exercising, and automated testing of, SOAP services.

The steps necessary for a SoapUI Client include:

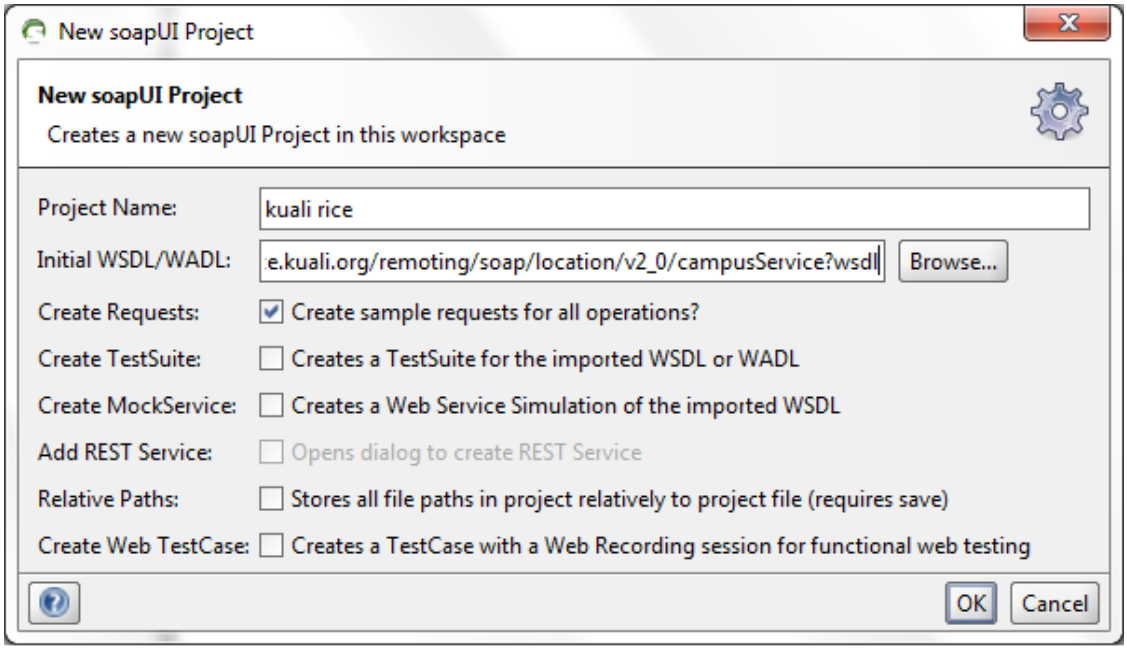
1. Creating the SoapUI project
2. Identification of the rice.keystore file to the SoapUI project definition
3. Configuring the use of the rice.keystore for outgoing requests
4. Associating the rice.keystore to a request

## Creating the SoapUI project

Create a new SoapUI project and provide the location of the WSDL.

```
http://demo.rice.kuali.org/remoting/soap/location/v2_0/campusService?wsdl
```

**Figure 22.1. Create a new SoapUI project**




# Identifying rice.keystore to the SoapUI project

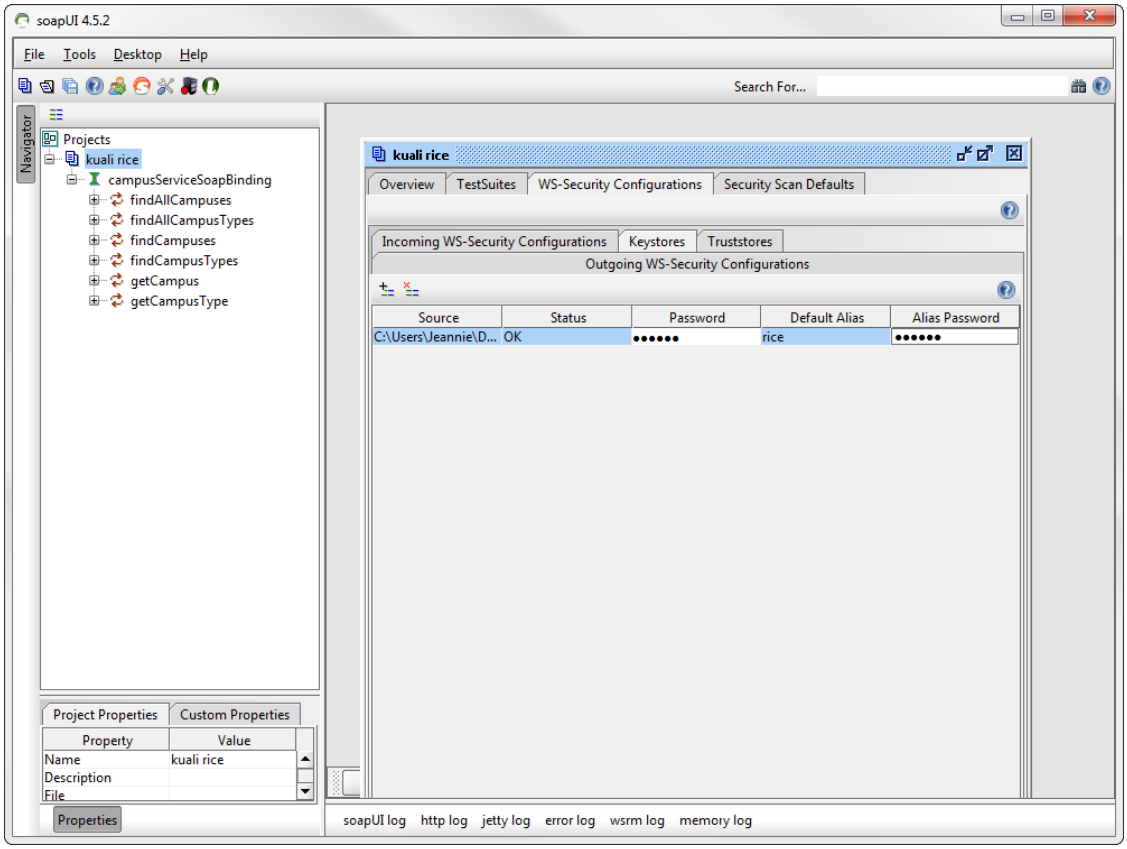
With the project highlighted, use the **Enter** key to bring up the project attributes.

- choose the **WS-Security Configurations** parent tab
- choose the **Keystores** child tab
- add our keystore

  1. Navigate to the location of your rice.keystore
  2. Provide the keystore password (r1c3pw is the default)
  3. Choose a default alias of **rice**
  4. Provide the alias password (also r1c3pw is the default)

The interface should report the Status of "OK" if the keystore is found and accessible with the keystore password provided.

**Figure 22.2. Identify rice.keystore to the SoapUI project**




# Configure using the keystore for outgoing requests

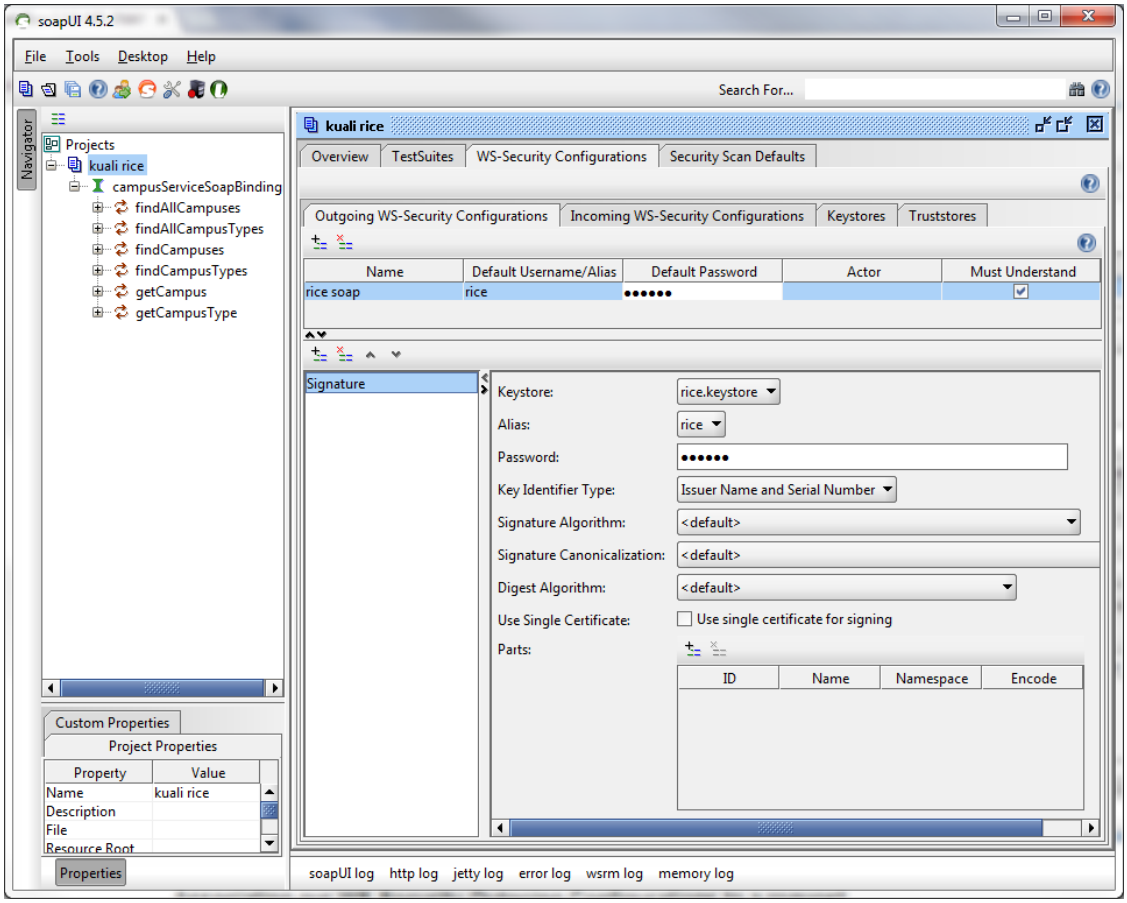
Still within the project attributes:

- choose the **WS-Security Configurations** parent tab
- choose the **Outgoing WS-Security Configurations** child tab
- add a new configuration
- provide an internal name for the configuration
- provide the alias name of **rice**
- provide the alias password (r1c3pw is the default)
- add a **Signature** action
  1. Choose our **rice.keystore** keystore
  2. Choose our **rice** alias
  3. Provide our alias password (r1c3pw)
  4. The **Key Identifier and Serial Number** should be defaulted and is the desired choice.

5. Leave the other entries as **default**

When done with the project attributes, close the attributes window.

**Figure 22.3. Identify rice.keystore to the SoapUI project**

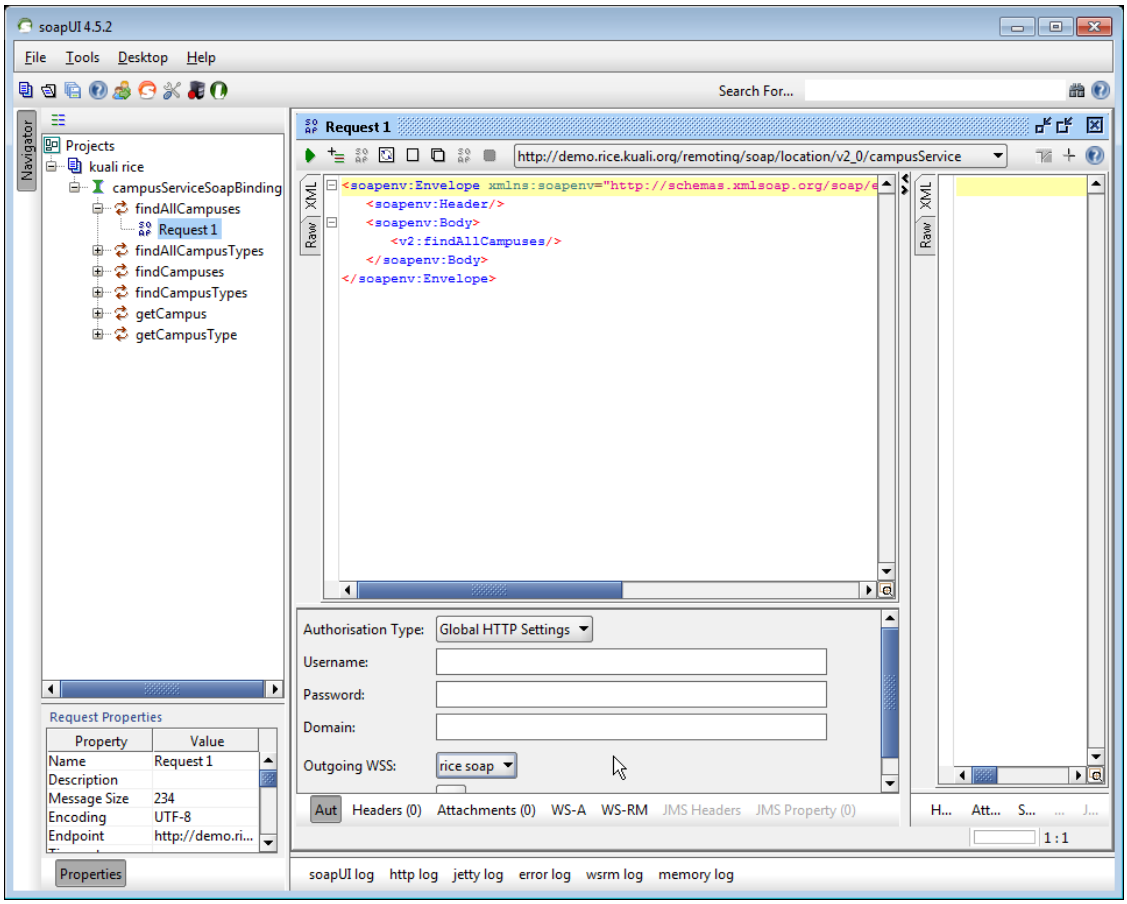



# Associating our WS-Security Outgoing Configurations to a request

Expand the project tree to find our desired request of **findAllCampuses**

- click on the **Request1** name
- find the **Aut** (short for Authentication) button at the bottom of the window
- choose our internal name for the outgoing configuration

**Figure 22.4. Associate Outgoing Config to a Request**

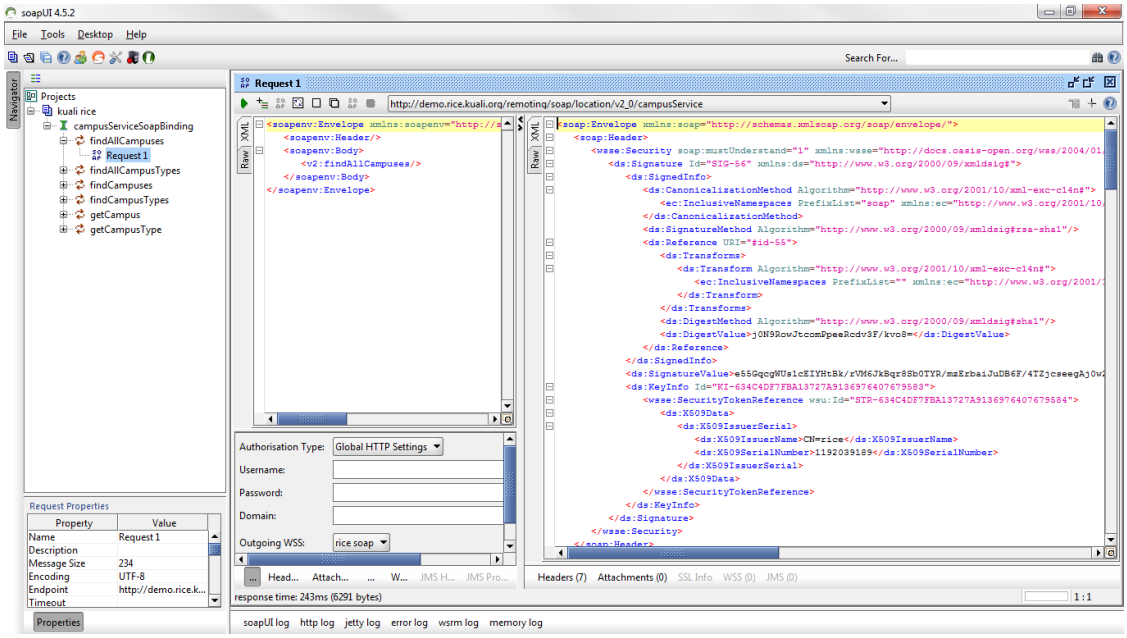



# Execute the request

Click the green triangle "run" button at the top of the window to execute the request

- the request should execute successfully and return an XML structure which includes attributes on each of the 12 campuses defined in the demo Rice data set

**Figure 22.5. Execute request**




# KSB Java Client

The generation of a Java client application is relatively easy using the capabilities of JDK1.6 and Apache CXF libraries.

The steps outlined below include:

1. Generating a web service client from the SOAP WSDL
2. Creating a Maven project including dependencies
3. Writing just enough Java code to employ the generated client and including the WS-Security headers using the keystore

# Generating the web service client

Using the JDK1.6+ tool **wsimport**, a Java client can be created directly from the WSDL definition of our desired service.

Use wsimport with the following parameters

- Keep to keep the generated source files
- Xnocompile since your .java files will probably be added and compiled inside of an IDE anyway
- Verbose to see the list of classes generated
- http://demo.rice.kuali.org/remoting/soap/location/v2_0/campusService?wsdl the WSDL location to the **campusService** service

Your output should look similar to this listing.

**wsimport**

```
$ wsimport -keep -Xnocompile -verbose http://demo.rice.kuali.org/remoting/soap/location/v2_0/campusService?wsdl
parsing WSDL...


generating code...

org\kuali\rice\core\v2_0\AbstractPredicate.java
org\kuali\rice\core\v2_0\AndType.java
org\kuali\rice\core\v2_0\CompositePredicateType.java
org\kuali\rice\core\v2_0\EqualIgnoreCaseType.java
org\kuali\rice\core\v2_0\EqualType.java
org\kuali\rice\core\v2_0\GreaterThanOrEqualType.java
org\kuali\rice\core\v2_0\GreaterThanType.java
org\kuali\rice\core\v2_0\IllegalArgumentFault.java
org\kuali\rice\core\v2_0\InIgnoreCaseType.java
org\kuali\rice\core\v2_0\InType.java
org\kuali\rice\core\v2_0\LessThanOrEqualType.java
org\kuali\rice\core\v2_0\LessThanType.java
org\kuali\rice\core\v2_0\LikeType.java
org\kuali\rice\core\v2_0\NotEqualIgnoreCaseType.java
org\kuali\rice\core\v2_0\NotEqualType.java
org\kuali\rice\core\v2_0\NotInIgnoreCaseType.java
org\kuali\rice\core\v2_0\NotInType.java
org\kuali\rice\core\v2_0\NotLikeType.java
org\kuali\rice\core\v2_0\NotNullType.java
org\kuali\rice\core\v2_0\NullType.java
org\kuali\rice\core\v2_0\ObjectFactory.java
org\kuali\rice\core\v2_0\OrType.java
org\kuali\rice\core\v2_0\QueryByCriteriaType.java
org\kuali\rice\core\v2_0\package-info.java
org\kuali\rice\location\v2_0\CampusQueryResultsType.java
org\kuali\rice\location\v2_0\CampusService.java
org\kuali\rice\location\v2_0\CampusService_Service.java
org\kuali\rice\location\v2_0\CampusType.java
org\kuali\rice\location\v2_0\CampusTypeQueryResultsType.java
org\kuali\rice\location\v2_0\CampusTypeType.java
org\kuali\rice\location\v2_0\FindAllCampusTypes.java
org\kuali\rice\location\v2_0\FindAllCampusTypesResponse.java
org\kuali\rice\location\v2_0\FindAllCampuses.java
org\kuali\rice\location\v2_0\FindAllCampusesResponse.java
org\kuali\rice\location\v2_0\FindCampusTypes.java
org\kuali\rice\location\v2_0\FindCampusTypesResponse.java
org\kuali\rice\location\v2_0\FindCampuses.java
org\kuali\rice\location\v2_0\FindCampusesResponse.java
org\kuali\rice\location\v2_0\GetCampus.java
org\kuali\rice\location\v2_0\GetCampusResponse.java
org\kuali\rice\location\v2_0\GetCampusType.java
org\kuali\rice\location\v2_0\GetCampusTypeResponse.java
org\kuali\rice\location\v2_0\ObjectFactory.java
org\kuali\rice\location\v2_0\RiceIllegalArgumentException.java
org\kuali\rice\location\v2_0\package-info.java
```

The classes are generated into two package structures which directly match the namespace declarations in the WSDL:

- org.kuali.rice.core.v2_0
- org.kuali.rice.location.v2_0

There are other command-line parameters for wsimport which gives you control over package names if you so desire.

# Create a Maven project

Maven is only a suggestion, of course. But, usage of Maven will ease the structure of your application and the dependencies.

Your pom.xml can be as simple as the example given here.

- cxf-rt-frontend-jaxws
- csf-rt-ws-security
- junit (test scope)

**pom.xml**

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>edu.somewhere</groupId>
  <artifactId>ksb-campusservice-client</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <dependencies>
    <dependency>
        <groupId>org.apache.cxf</groupId>
        <artifactId>cxf-rt-frontend-jaxws</artifactId>
        <version>2.7.3</version>
    </dependency>
    <dependency>
        <groupId>org.apache.cxf</groupId>
        <artifactId>cxf-rt-ws-security</artifactId>
        <version>2.7.3</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.11</version>
        <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Apache CXF provides very convenient classes and functionality to take the XML of the request and provide the digesting and signing. It is possible to perform those functions without CXF by using wss4j directly or even modifying the XML of the request by hand, but the Apache CXF classes make it much easier.

# Writing the Java code

The approach here is to write a client class that can be used as the Data Access Object (DAO) in your data access layer (DAL) of an enclosing application. This Maven project could be compiled as a .jar and included as a dependency for your other applications needing access to the Rice KSB services.

Your project will need to include the following:

- the generated classes, keeping their package structure intact, included at src/main/java
- **client-sign.properties** at src/main/resources
- **rice.keystore** included at src/main/resources

**client-sign.properties**

```
# Properties for the KSB Client Test classes

# properties for accessing the java keystore using Merlin
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=r1c3pw
org.apache.ws.security.crypto.merlin.keystore.alias=rice
org.apache.ws.security.crypto.merlin.keystore.file=rice.keystore
```

Your Java code will include the following:

- **KSBCampusServiceClient.java**

**KSBCampusServiceClient.java**

```
package edu.somewhere;

import java.net.URL;
import java.util.HashMap;
import java.util.Map;

import org.apache.cxf.endpoint.Client;
import org.apache.cxf.endpoint.Endpoint;
import org.apache.cxf.frontend.ClientProxy;
import org.apache.cxf.ws.security.wss4j.WSS4JOutInterceptor;
import org.apache.ws.security.handler.WSHandlerConstants;
import org.kuali.rice.location.v2_0.CampusService;
import org.kuali.rice.location.v2_0.CampusService_Service;

public class KSBCampusServiceClient
{

    public CampusService getCampusService( URL url )
    {
        CampusService_Service svc = new CampusService_Service();

        CampusService campusService = svc.getCampusServicePort();

        Client client = ClientProxy.getClient( campusService );
        Endpoint cxfEP = client.getEndpoint();

        Map<String, Object> outProps = new HashMap<String, Object>();
        outProps.put( WSHandlerConstants.ACTION, "Signature" );
        outProps.put( WSHandlerConstants.USER, "rice" );
        outProps.put( WSHandlerConstants.PW_CALLBACK_CLASS, KSBClientCallbackHandler.class.getName() );
        outProps.put( WSHandlerConstants.SIG_PROP_FILE, "client-sign.properties" );

        WSS4JOutInterceptor wssOut = new WSS4JOutInterceptor( outProps );

        cxfEP.getOutInterceptors().add( wssOut );

        return campusService;
    }
}
```

This is your public DAO class and it provides the wiring of the WS-Security Signature action to the WSS4JOutInterceptor. In other words, it takes your outbound XML request and properly adds the digest and signature values in the SOAP header. The key identifiers in this class include the **rice** user, the **callback handler** class, and the **properties** file for access to the keystore.

- **KSBClientCallbackHandler.java**

**KSBClientCallbackHandler.java**

```
package edu.somewhere;

import java.io.IOException;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;

import org.apache.ws.security.WSPasswordCallback;

public class KSBClientCallbackHandler implements CallbackHandler
{

    @Override
    public void handle( Callback[] callbacks ) throws IOException, UnsupportedCallbackException
    {
        for( Callback thisCallback : callbacks )
        {
            WSPasswordCallback pwcb = (WSPasswordCallback)thisCallback;
            String user = pwcb.getIdentifier();
            int usage = pwcb.getUsage();
```

```
            if( usage == WSPasswordCallback.SIGNATURE )
            {
                // this is to provide the password for the alias within the keystore
                //  - while it is the same value as the keystore name and password,
                //  - you could craft a different alias than the keystore user
                if( "rice".equals( user ) ) pwcb.setPassword( "r1c3pw" );
            }
        }
    }
}
```

This class is an implementation of a callback handler. The wss4j WS-Security function requires access to the keystore for purposes of hashing and signing the request. This is accomplished by providing a callback to a class which can provide the proper keystore credentials. This is the same technique used for simple username/password credentialing of simple WS-Security SOAP headers, but in our case this callback handler is asking for the password to the certificate identified by the alias within the keystore.

- **KSBCampusServiceClientTest.java**

```
KSBCampusServiceClientTest.java
package edu.somewhere;

import static org.junit.Assert.*;

import java.net.URL;

import org.junit.Test;
import org.kuali.rice.location.v2_0.CampusService;
import org.kuali.rice.location.v2_0.CampusType;
import org.kuali.rice.location.v2_0.FindAllCampusesResponse.Campuses;

import edu.somewhere.KSBCampusServiceClient;

public class KSBCampusServiceClientTest
{

    @Test
    public void campusServiceTest() throws Exception
    {
        KSBCampusServiceClient client = new KSBCampusServiceClient();

        CampusService svc = client.getCampusService( new URL( "http://demo.rice.kuali.org/remoting/soap/
location/v2_0/campusService?wsdl" ) );
        Campuses campuses = svc.findAllCampuses();

        assertEquals( 12, campuses.getCampus().size() );

        for( CampusType campus : campuses.getCampus() )
        {
            System.out.printf( "%s : %s : %s \n", campus.getCode(), campus.getShortName(), campus.getName() );
        }
    }
}
```

This class was written as a JUnit test class (located at src/test/java) but could easily be a class with a main() method as well. This serves as the example calling class employing the client's functionality.

# Complete sample application

The code included here is available from a public respository.

- https://bitbucket.org/majorbanzai/kuali-kode/src

  look for the ksbclient directory

- eclipse project

- maven project

```
get the code
$ hg clone https://bitbucket.org/majorbanzai/kuali-kode
$ cd kuali-kode/ksbclient
$ mvn test
```

- KSBCampusServiceClientTest integration test

```
KSBCampusServiceClientTest output
BL : BLOOMINGTON : BLOOMINGTON
BX : BLGTN OFF CA : BLGTN OFF CAMPUS
CO : COLUMBUS : COLUMBUS
EA : EA-RICHMOND : EAST-RICHMOND
FW : FORT WAYNE : FORT WAYNE
IN : INDIANAPOLIS : INDIANAPOLIS
KO : KOKOMO : KOKOMO
NW : NW-GARY : NORTHWEST-GARY
OC : OFF CAMPUS : OFF CAMPUS
SB : SOUTH BEND : SOUTH BEND
SE : SE-NEW ALBNY : SOUTHEAST-NEW ALBANY
UA : UNIVER ADMIN : UNIVERSITY ADMINISTRATION
```

# Using the KSB with bus security - SOAP request

## SOAP request with WS-Security header

What does the SOAP request look like, anyway?

Because the demo Rice server is using the http protocol, we can use our favorite network sniffer to watch the traffic. Your SOAP message looks like the following:

```
SOAP request
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:v2="http://rice.kuali.org/
location/v2_0">
   <soapenv:Header>
      <wsse:Security soapenv:mustUnderstand="1"
            xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"
            xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
         <ds:Signature Id="SIG-6" xmlns:ds="http://www.w3.org/2000/09/xmldsig#"> ❶
            <ds:SignedInfo> ❷
               <ds:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
                  <ec:InclusiveNamespaces PrefixList="soapenv v2"
                      xmlns:ec="http://www.w3.org/2001/10/xml-exc-c14n#"/>
               </ds:CanonicalizationMethod>
               <ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
               <ds:Reference URI="#id-5">
                  <ds:Transforms>
                     <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
                        <ec:InclusiveNamespaces PrefixList="v2"
                            xmlns:ec="http://www.w3.org/2001/10/xml-exc-c14n#"/>
                     </ds:Transform>
                  </ds:Transforms>
                  <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
                  <ds:DigestValue>Vw3UgSOjyJLj2Qapgiw4//qxsO0=</ds:DigestValue> ❸
               </ds:Reference>
            </ds:SignedInfo>
            <ds:SignatureValue>eYwbnLoVQNKHQxJjC1zolpxZ6mx1ixU4C+pJaVP6fLDGdnwxU1ueWRoXAbDVCKFOYrPpX7k6TGii ❹
Q9Zy4CeUiI7KBqqGpOlMbQ5avVjAk4AGSIo3f02cAtx3kwlD/Dyb+PuCmAVm0QGB8GOeWDsFfX6x
```

```
RJuO40x5n3tPtNXhq5U=
            </ds:SignatureValue>
            <ds:KeyInfo Id="KI-21FCA5124C428B1F2113651984409138">
               <wsse:SecurityTokenReference wsu:Id="STR-21FCA5124C428B1F2113651984409139">
                  <ds:X509Data>
                     <ds:X509IssuerSerial>
                         <ds:X509IssuerName>CN=rice</ds:X509IssuerName> ❺
                         <ds:X509SerialNumber>1192039189</ds:X509SerialNumber> ❻
                     </ds:X509IssuerSerial>
                  </ds:X509Data>
               </wsse:SecurityTokenReference>
            </ds:KeyInfo>
         </ds:Signature>
      </wsse:Security>
   </soapenv:Header>
   <soapenv:Body wsu:Id="id-5"
            xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
      <v2:findAllCampuses/>
   </soapenv:Body>
</soapenv:Envelope>
```

**Request breakdown**

❶ <ds:Signature> begins the signing of the request

❷ <ds:SignedInfo> is the signature package. Once generated, the following bytes all the way down to the </ds:SignatureValue> cannot be altered

❸ <ds:DigestValue> is the hash digest generated from the body of the request and the signature value

❹ <ds:SignatureValue> is the signature. In this case some of the bytes, when printed, represent a new-line character. Any modification of this stanza byte stream will invalidate the signature

❺ <ds:X509IssuerName> is the alias expected and must be in the keystores of the server and the client

❻ <ds:X509SerialNumber> is directly from the keystore and must match between the server and the client. Using the **keytool** tool, you can see this value (represented in hex)

The request itself is very small. The **<soapenv:Body>** stanza represents the actual request of calling the **findAllCampuses** method. The bus security starts inside the header with the **<wsse:Security>** stanza.

## Note

The X509IssuerName and X509SerialNumber come directly from the keystore. Here the serial number is in base 10, in the keystore the value is shown in hex.

# Chapter 23. Caching Infrastructure

## Overview

As we decrease the direct database access from Rice clients and expose services remotely, service-level caching becomes more important. Previously in Rice we didn't approach caching with a standard comprehensive solution. This was problematic for many reasons explained later in the document. As caching starts to take a greater role in Rice it is clear that we must have a well thought out plan for caching. The caching solution we are looking for and have implemented must have the following properties:

1. Usable by developers without introducing bugs
2. Current (not built on dead or dying technologies)
3. Concise (doesn't pollute the codebase with caching logic)
4. Flexible (works for most/many caching situations)
5. Supports client/server side caching
6. Tunable/customizable (max cache size, cache to disk, etc)
7. Supports distributed caching (will it work with the KSB?)
8. Performant
9. Usable by Kuali clients for their own caching needs not just Rice
10. Version compatible
11. Pluggable (allows using different caching implementations)

## Proposal that was Implemented

Spring 3.1 includes a declarative cache abstraction API. This is an annotation driven approach which significantly reduces caching logic. The only thing service authors should have to do is annotate service interfaces (or implementation code) with Spring cache annotations. For example:

```
@Cacheable(value="foo", key="#p0")
public Foo getFoo(String id);

@CacheEvict(value = "foos", allEntries=true)
public Foo updatefoo(Foo f);
```

Then the service implementation would look like:

```
public Foo getFoo(String id) {
  return getFooFromDB(id);
}

public Foo updatefoo(Foo f) {
  return updateinDB(f);
```

```
}
```

All the boilerplate caching logic has been magically melted away through the wonders of AOP proxies. When Spring creates a Spring managed service (bean) it will automatically return a proxy containing caching logic. This works great for most cases, but falls apart when clients are calling services remotely. This is because the remote proxy is not created by Spring, but is instead created by the KSB (ServiceConnectorFactory). In order to handle this case, we will need to directly cache proxy our remote proxies.

To make sure the annotations are actually being read by Spring, we must include the following in our Spring xml files:

```
<cache:annotation-driven />
```

and declare a cache manager like:

```
<bean id="cacheManager" class="org.springframework.cache.ehcache.EhcacheCacheManager" p:cache-
manager="ehcache"/>

<!-- Ehcache library setup -->
<bean id="ehcache" class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean" p:config-
location="ehcache.xml"/>
```

Due to the fact that Spring is using proxies, there is a slight overhead in going through an extra layer. This will probably not be a problem, but if it is, Spring provides the option to use aspectj and aspect weaving. This will remove the proxying at the expense of complexity.

**Figure 23.1. Cache Proposal**

# The Implementation

The above proposal has been put to code, which here is explained in more detail. To understand the various parts of the Spring Cache abstraction and the implementation it is recommended to read the Spring cache documentation before going any further.

## The Spring Parts

- **CacheManager:**An interface that defines a way to retrieve a particular cache. This cache manager has a name and manages one of more cache objects
- **Cache:**An interface that defines a data structure to hold objects to cache. The cache has a name and can be thought of as a Map-like structure. In fact, some cache implementations are backed by a java.util.Map.
- **Cacheable:**An annotation to use on a Spring-managed (or non-Spring-managed w/ Kuali extensions) bean to enable method caching. This annotation has two important parts. One or more cache name(s) to put the cached object in and the key to use for caching. Both should be present. It is recommended that cache keys be simple string (or primitive) values.
- **CacheEvict:**An annotation to use on a Spring-managed (or non-Spring-managed w/ Kuali extensions) bean to enable cache eviction. This annotation has several important parts. You must always specify

one or more cache name(s). You can optionally specify either a clearAll flag to force the entire cache to be cleared or you can specify a cache key so that only one item is cleared from the cache.

- **Spring annotation processor:**an xml snippet to enable Spring caching on Spring beans. You must specify the CacheManager to use for caching. There are several optional settings that can be used on this declaration which will not be explained here.

### Note

Due to the way Rice is using the Spring Expression Language with Cacheable and CacheEvict annotations, Rice must be compiled with debug symbols.

# The Kuali Parts

- **CacheService:**An interface that defines operations to invoke on a **local** cache. This is used in distributed cache operations. Currently only supports flush style operations.
- **CacheServiceImpl:**The default implementation of the CacheService. It contains a reference to a CacheManager and invokes caching operations on it. Most standard Kuali apps will have multiple CacheService endpoints remotely available.
- **DistributedCacheManagerDecorator:** A CacheManager that decorates an existing CacheManager. It adds distributed caching operations by retrieving a list of CacheServices deployed on the bus and calling each one asynchronously. In the future, this will only call CacheService endpoints that are interested in receiving a certain message. Although some of the diagrams on this page may suggest that the distributed cache messages execute immediately, they are actually queued up and sent in bulk at the end of a transaction. This means that our distributed caching is transaction aware. The queuing nature of this class helps decrease the chattiness of cache flush messages on the KSB.

### Note

Since all cache keys must generate stable soap values, all cache keys are coerced to a String by this decorator. This is why our cache keys should be primitive values; otherwise, we might be relying on unstable toString implementations.

- **CacheProxy:**A utility class provides an extension to the Spring cache abstraction. This allows the proxing on non-Spring managed beans with Spring caching behavior. This is used for client-side caching behavior for remote proxies. See Spring enhancement JIRA

# A Real Example

```
FooService.java
interface FooService {

  //demonstrates a simple argument
  @Cacheable(value=Foo.Cache.NAME, key="'id=' + #p0")
  Foo getFoo(String id);

  //demonstrates a complex argument - build a string. No using the actual object as key
  @Cacheable(value=Foo.Cache.NAME, key="'name=' + #p0.name + '|' + 'name=' + #p0.code")
  Foo getFooByNameAndCode(NameAndCode nc);

  //demonstrates no arguments.  making up a key
  @Cacheable(value=Foo.Cache.NAME, key="all")
  Collection<Foo> findAllFoos();

  //demonstrates a single evict.
  //We need to be careful here because if multiple
  //"keys" hold the FooType object then the allEntries must be true*
```

```
  @CacheEvict(value=FooType.Cache.NAME, key="'id=' + #p0")
  void updateNameOnFooType(String id, String name);

  //demonstrates a complete evict.
  @CacheEvict(value=Foo.Cache.NAME, allEntries=true)
  void addFoos(Collection<Foo> foos);
}
```

**FooSpringBeans.xml**

```
<beans>
  <!-- tell Spring to look for cache annotations and which CacheManager to use -->
  <cache:annotation-driven cache-manager="fooDistributedCacheManager" />

  <!--
     create a local CacheManager.  Cache operations on this CacheManager only happen
     against the application's local cache.
     Can use any cache implementation: java.util.concurrent, ehcache, etc.
   -->
  <bean id="fooLocalCacheManager" class="org.springframework.cache.support.SimpleCacheManager">
    <property name="caches">
      <set>
        <bean class="org.springframework.cache.concurrent.ConcurrentMapCacheFactoryBean"
          p:name="#{T(org.Kuali.Rice.module.api.foo.Foo$Cache).NAME}"/>
        <bean class="org.springframework.cache.concurrent.ConcurrentMapCacheFactoryBean"
          p:name="#{T(org.Kuali.Rice.module.api.foo.FooType$Cache).NAME}"/>
      </set>
    </property>
  </bean>

  <!--
       Wrap the local CacheManager in a decorator to enable distributed cache
       operations across the Kuali ecosystem.
   -->
  <bean id="fooDistributedCacheManager"
 class="org.Kuali.Rice.core.impl.cache.DistributedCacheManagerDecorator">
    <!-- the local CacheManager to wrap -->
    <property name="cacheManager" ref="sharedDataLocalCacheManager" />
    <!-- the ksb service to lookup and call CacheService Enpoints asynchronously -->
    <property name="messageHelper" ref="Rice.ksb.messageHelper" />
    <!-- the name of the endpoint to call.  Must be the same (for this module) across all applications in the
 Kuali ecosystem -->
    <property name="serviceName" value="{http://Rice.Kuali.org/foo/2_0}fooModuleCacheServiceSoap" />
    <!--
         how long to wait in milliseconds before flushing the distributed cache queue and sending distributed
 flush messages.
         defaults to 60000 (60 seconds).
     -->
    <property name="flushQueueMaxWait" value="${Rice.cache.flush.queue.max.wait}" />
  </bean>

  <!--
    Service that should be exposed on the ksb to receive messages from the distributed cache manager.
    Notice it handles calling into the *local* CacheManager.
   -->
  <bean id="fooCacheService" class="org.Kuali.Rice.core.impl.cache.CacheServiceImpl"
        p:cacheManager-ref="fooLocalCacheManager" />
</beans>
```

**FooServiceBusSpringBeans.xml**

```
<beans>
  <!-- export the CacheService on the service bus to receive distributed cache messages for the foo module-->
  <bean parent="fooRemoteServiceExporter">
    <property name="serviceBus" ref="Rice.ksb.serviceBus"/>
    <property name="serviceDefinition">
      <bean parent="fooJaxWsSoapService"
            p:service-ref="fooCacheService"
            p:localServiceName="fooCacheServiceSoap"/>
    </property>
    <property name="exportIf" value="fooCacheServiceSOAP.expose"/>
  </bean>
</beans>
```

# Standards and Rules

## Version Compatibility Rules

1. Cache Names cannot change (using the object's namespace is a good way to enforce this).
2. Cache Keys cannot change (may want to create a utility method for this on each object we are caching).
3. Always use simple keys (Strings or primitives).
4. When doing a single evict (allEntries=false), object can only be present with a single cache key. (*more on this below [81]).

## Suggested Standards

1. Only *effectively* immutable/thread-safe objects should be cached!
2. One cache manager per module KimCacheManager, KewCacheManager.
3. One cache per top-level object Permission, Responsibility, etc.
4. One remotely available CacheService per cacheManager (*more on this below [81]).
5. Use jdk style proxying (*more on this below [82]).
6. All Remotable services should cache.
7. Always annotate service interfaces so remote proxies automatically get client-side caching.

## Notes on Standards, Rules, etc.

**Many CacheService Enpoints:** One CacheService endpoint per CacheManager allows client apps to use Rice's caching infrastructure without sending distributed cache flush messages to apps that don't care. For example: KC exposes a remote service (AwardService) to KFS. KC hands KFS a fully cache annotated service interface. KFS and KC clusters can participate is distributed cache messages without bothering other Kuali apps that don't ever call the AwardService and don't have a AwardCacheService exposed remotely. Another interesting prospect is a Kuali ecosystem may have Rice installs with different "modules" enabled. This design allows the Rice installs to only receive messages for the modules they have enabled (XXXCacheService available).

**Spanning CacheManagers:** This design cannot currently handle flushing across CacheManagers. This is a current limitation although in practice it may not be needed. For example: Say the GenericType object is used and cached in KIM and KEW (KimCacheManager, KewCacheManager). If a Kim api updates the GenericType object the KimCacheManager will handle flushing the kim module cache but the KewCacheManager's cache will be stale.

One way we can handle this in the situations that we definitively need to access another cache manager, is to execute the following code in the service implementation (in normal cases this should be avoided):

```
GlobalResourceLoader.getService("alternateDistributedCacheManager").getCache("the_cache_to_retrieve").evict("the_specific_key");
```

**Same object, multiple cache keys:** See Spring enhancement JIRA #2 for more info. Seems like we will be doing a lot of @CacheEvict(value="cache_name" allEntries=true) because the same object may be present under multiple cache keys. Not exactly sure what to do about this...We could have a cache per method but that will be hard to manage. Maybe the underlying caching implementations can handle this for us?

One way we can handle this in the situations that we definitively want to avoid flushing an entire cache, is by executing the following code in the service implementation (in normal cases this should be avoided):

```
GlobalResourceLoader.getService("fooDistributedCacheManager").getCache("the_cache_to_retrieve").evict("the_specific_key");
```

**jdk proxying?** With the Spring caching abstraction, you can either proxy a service to inject the caching logic (like a decorator), or use bytecode weaving with aspectj. Proxying is a simpler solution while less performant than aspectj. Unless jdk proxying becomes a significant bottleneck (which seems doubtful), then using code weaving should be an option implementers can turn on but not enabled by default. Tuning the cache setting (like ehcache settings) is probably a more important thing to do than proxy versus code weaving.

**pushing/priming:** Distributed cache updates (pushing updates to clients), cache priming, or cache warming is currently not supported.

**where to cache?** Although we have primarily targeted our remotable services for caching, there is no reason why caching couldn't get used anywhere in Rice or a client application. We just need to be mindful of the version compatibility rules.

**caching mutable objects?** This depends on the implementation of the caching framework. If using ConcurrentHashMap as a caching implementation, then mutable values should NOT be cached. If using ehcache, then mutable values can be cached as long as the cache is configured correctly to do a defensive copy. The safest rule of thumb in Rice is to only store immutable values in a cache. This gives implementers the greatest flexibility in regards to what caching implementation to use.

**duplicate cache flush messages:** This is the biggest drawback to this design. The server has to be the entity to send out the distributed cache flush messages. Why? This is because the server knows if a destructive call succeeds and therefore causes a stale cache. Since the server does not know which client made the service request, the server will send out a cache flush message to the calling client even though the client already cleared his own cache. If there was some way to pass along the instanceId of the calling client, this could be avoided. It appears the RiceCacheAdministrator (RiceDistributedCacheListener) has the same limitation if used for client and server side caching. Maybe, the KSB could maintain a ThreadLocal variable that contains the calling client's applicationId, instanceId, etc. It could do this through some interceptor style pattern. The interceptor would need to make sure the variable is cleared even when exceptions happen. The thread local idea is kind of a code smell, but may be just what the doctor ordered in this case.

**make sure we support bundled:** This should be working now but we need to confirm that when in dev.mode in a bundled architecture, this still works correctly.

**no compile dependency on ehcache:** By using Spring's Cache Abstraction, there is no need to compile against any ehcache APIs. In fact, the maven dependency for ehcache is runtime only (which could even be switched to optional). It's important that we be mindful of this in the future because this allows implementers to switch ehcache for some other solution (like JBoss' native caching support).

**cache keys:** Cache keys should be made up of the important arguments to a method and optionally the method name. They key is meant to uniquely identify a method's return value in a cache. A few examples are:

```
@Cacheable(value= Group.Cache.NAME, key="'{getAttributes}' + 'groupId=' + #p0")
```

```
@Cacheable(value= Group.Cache.NAME, key="'id=' + #p0")
```

# Caching Administration UI

## Requirements

The caching UI should allow a system administrator visualize the "local" caches in a running instance of a cache enabled Kuali Application. The administrator should have the ability to trigger a **distributed** cache flush of cached item(s). To demonstrate the items that must be displayed on this UI see the following example:

- KimCacheManager
  - RoleCache
    - CacheEntry (id-1)
    - CacheEntry (id-2)
  - PermissionCache
- KewCacheManager
  - DocumentTypeCache
    - CacheEntry(ParameterDocumentType)

With the above example, an admin should be able to do the following:

- Flush All CacheManagers (KimCacheManager, KewCacheManager)
- Flush KimCacheManager
- Flush RoleCache in KimCacheManager
- Flush CacheEntry (id-1) in RoleCache in KimCacheManager

Access to the screen and flush actions must also be locked down through KIM Permissions.

## Non-requirements

- We have not identified the need to do a non-distributed flush through the UI (local flush).
- We have not identified the need to do a complete flush of all caches across the Kuali-ecosystem from a single point. For example: If you wanted to flush KFS specific cache you would have to login to the KFS admin screen to perform that action rather than pushing an uber-flush button from Rice.
- We have not identified the need to dynamically disable caching from a UI on a running application

# Putting it all together

Below are a couple pseudo examples of UML sequence diagrams to help illustrate a couple standard call flows.

**Figure 23.2. Standard call flow 1**

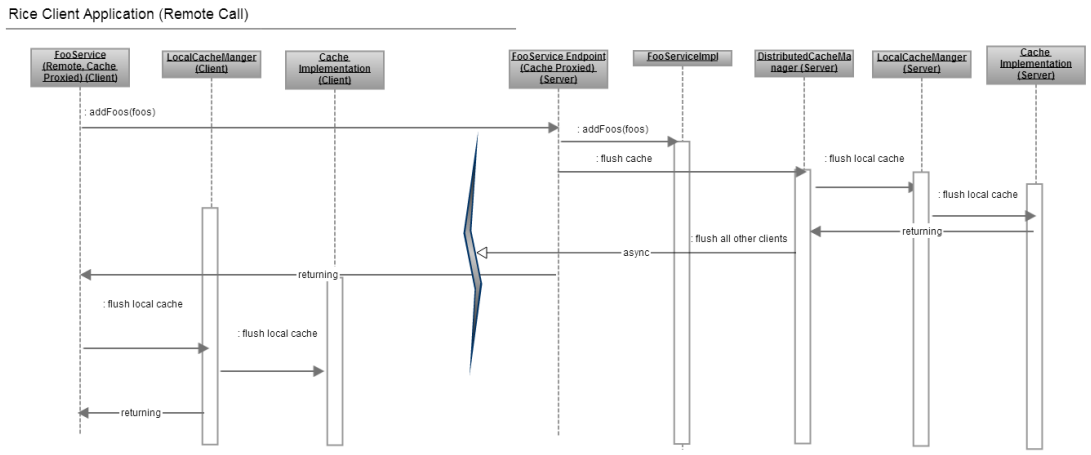
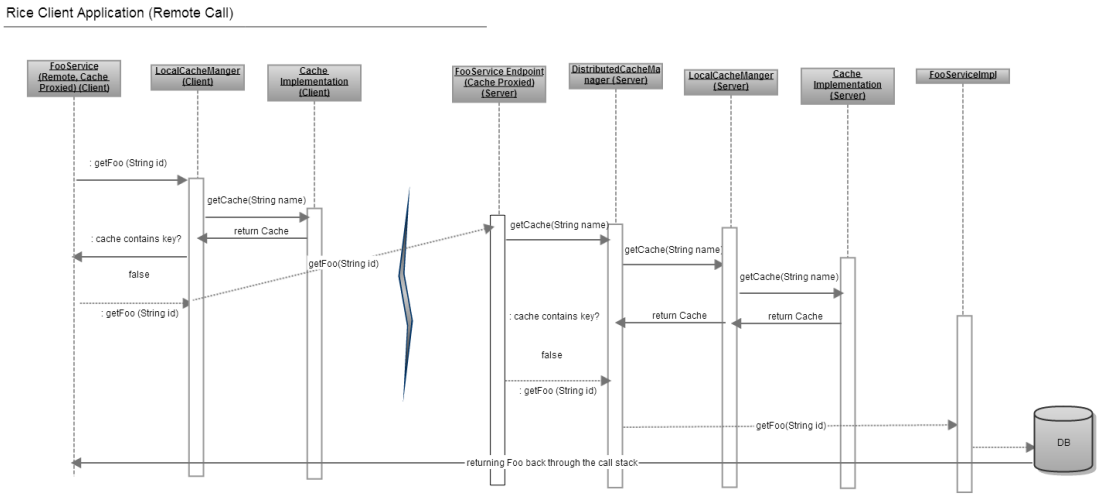



**Figure 23.3. Standard call flow 2**




# Implementation Plug Points

One critical piece of this design is the ability to plugin into different cache implementations with very little impact to the Rice codebase. Why would you want to do this? Simply put: some applications servers or infrastructures have alternative caching frameworks that have advantages over what we provide with Rice. In order to achieve this, the Rice team (and other Kuali apps) must make an effort to NOT directly use a caching framework in code, but to always go through Spring's caching abstraction. In Rice, we will achieve this by making our default caching implementation (ehcache) a runtime or optional dependency. Remember: the following hints for customization will have to be done for every module of Rice and every cache enabled Kuali app.

## Option 1: replacing the default caching implementation

To do this you must replace(or override) the following Spring entries for the **local** CacheManagers. For example:

```
<bean id="sharedDataLocalCacheManager" class="org.springframework.cache.ehcache.EhCacheCacheManager">
```

```
  <property name="cacheManager">
    <bean class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean"
       p:config-location="${shareddata.ehcache.config.location}"/>
  </property>
</bean>
```

Could be replaced with:

```
<!-- this assumes org.jboss.JBossCacheManager implements the org.springframework.cache.support.CacheManager
 interface -->
<bean id="sharedDataLocalCacheManager" class="org.jboss.JBossCacheManager">
  <!--...-->
</bean>
```

## Option 2: replacing the Distributed CacheManager

Doing Option 1 changes the caching implementation but still uses the Kuali Service Bus for transaction-aware flush messages. Many Caching implementations already provide these facilities. You could remove or replace the following:

```
<bean id="sharedDataDistributedCacheManager"
 class="org.Kuali.Rice.core.impl.cache.DistributedCacheManagerDecorator">
  <property name="cacheManager" ref="sharedDataLocalCacheManager" />
  <property name="messageHelper" ref="Rice.ksb.messageHelper" />
  <property name="serviceName" value="{http://Rice.Kuali.org/shareddata}sharedDataCacheServiceSoap" />
</bean>
```

Doing this may mean that the CacheService endpoints are no longer used so the following entries could be removed as well:

```
<bean id="sharedDataCacheService" class="org.Kuali.Rice.core.impl.cache.CacheServiceImpl"
  p:cacheManager-ref="sharedDataLocalCacheManager" />
```

```
<bean parent="sharedDataRemoteServiceExporter">
  <property name="serviceBus" ref="Rice.ksb.serviceBus"/>
  <property name="serviceDefinition">
    <bean parent="sharedDataJaxWsSoapService"
          p:service-ref="sharedDataCacheService"
          p:localServiceName="sharedDataCacheServiceSoap"/>
  </property>
  <property name="exportIf" value="sharedDataCacheServiceSOAP.expose"/>
</bean>
```

And finally, remember to update the cache section of the Spring files like the following:

```
<cache:annotation-driven cache-manager="jbossDistributedCacheManager" />
```

## Option 3: Aspect Weaving

There have been some concerns that Rice's choice to use jdk proxying may cause some overhead. To switch to aspect weaving which is more performant change the following:

```
<cache:annotation-driven cache-manager="sharedDataDistributedCacheManager" />
```

to

```
<cache:annotation-driven cache-manager="sharedDataDistributedCacheManager" mode="aspectj" />
```

You must also include the spring-aspectj.jar on the classpath.

# References


Kuali Rice JIRA

Design/Code Review

Spring Cache Abstraction

EhCache

Spring enhancement JIRA #1

Spring enhancement JIRA #2

Spring bug JIRA #1

Rice 2.0 Wiki - Compatibility Refactoring - Caching Infrastructure

# 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:<br>• Approve: requests an approve or disapprove action.<br>• 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.<br>• 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.<br>• 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:<br>• Acknowledge: requests that the users states he or she has reviewed the document.<br>• Approve: requests that the user either Approve or Disapprove a document.<br>• Complete: requests the user to enter additional information in a document so that the content of the document is complete.<br>• 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:<br>• Acknowledged: Reviewer has viewed and acknowledged document.<br>• 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:<br>• 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.<br>• 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*.<br><br>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.<br><br>Document Types have the following characteristics:<br><br>• 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<br><br>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 interfact 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.<br><br>Groups can be given authorization to perform actions within applications by assigning them as members of *Roles*.<br><br>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.<br><br>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.<br><br>Group Attributes make up the identity information for a Group record.<br><br>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:<br>• Automatic Message Generation and Logging<br>• Message integrity and delivery standards<br>• 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:<br>• A services registry and repository for identifying and instantiating services<br>• Run time monitoring of messages<br>• 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*<br>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".<br><br>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".<br><br>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.<br><br>In other words, an Entity is for identity while a Principal is for access management.<br><br>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.<br>• Simple: do some arbitrary work<br>• Requests: generate action requests using a Route Module or the Rules engine<br>• Split: split the route path into one or more parallel branches<br>• Join: join one or more branches back together<br>• Sub Process: execute another route path inline<br>• 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:<br>• Approved: These documents have been approved by all required reviewers and are waiting additional postprocessing.<br>• 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.<br>• Disapproved: These documents have been disapproved by at least one reviewer. Routing has stopped for these documents.<br>• Enroute: Routing is in progress on these documents and an action request is waiting for someone to take action.<br>• 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.<br>• 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*.<br><br>Technical considerations for a Searchable Attributes are:<br><br>• They are responsible for extracting and indexing document data for searching<br><br>• They allow for custom fields to be added to Document Search for documents of a particular type<br><br>• They are configured as an attribute of a Document Type<br><br>• 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.<br><br>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.<br><br>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:<br>• Quick EDoc Watch: The last five Actions taken by this user. The user can select and repeat these actions.<br>• Quick EDoc Search: The last five EDocs searched for by this user. The user can select one and repeat that search.<br>• 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*.<br>1. An acronym for Extensible Markup Language.<br>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 |