Rules 2

Overview

The Rules Framework allows you to code business rules against which your Document will be be validated whenever a specified Event is fired (such as on Save, Enroute, etc). If the validation does not pass, the Event is aborted and an error message specified by the programmer will be presented to the user. This page is a how-to guide for using this framework.

Business Rule Classes

Business rule classes are java classes that implement one or more event interfaces. Through the interface method implementations, business rules can be checked and errors returned.

Sample Business Rule Class

```
public class SampleRuleClass implements SaveDocumentRule, RouteDocumentRule {
    /*
     * @see
org.kuali.core.rule.SaveDocumentRule#processSaveDocument(org.kuali.core.document
.Document)
     *
     */
    public boolean processSaveDocument(Document document) {
        boolean isValid = true;

        isValid &= isDocumentOverviewValid(document);

        if (isValid) {
            isValid &= processCustomSaveDocumentBusinessRules(document);
        }

        return isValid;
    }

    /*
     *
     * @see
org.kuali.core.rule.RouteDocumentRule#processRouteDocument(org.kuali.core.docume
nt.Document)
     */
    public boolean processRouteDocument(Document document) {
        boolean isValid = true;

        isValid &= isDocumentAttributesValid(document);

        if (isValid) {
            isValid &= processCustomRouteDocumentBusinessRules(document);
        }

        return isValid;
    }

}
```

This class is setup to respond to SaveDocumentRuleEvent and RouteDocumentRuleEvent for the particular document it is registered with in the data dictionary.

> Note this class must implement the two interface methods *processSaveDocument(Document)* and *processRouteDocument(Document)*.

Registering your business rule class with a document type.

BudgetAdjustmentDocument.xml

```
<dictionaryEntry>
    <transactionalDocument>

<documentClass>org.kuali.module.financial.document.BudgetAdjustmentDocument</documentClass>

<businessRulesClass>org.kuali.module.financial.rules.BudgetAdjustmentDocumentRule</businessRulesClass>
        ...
```

The fully qualified class name of the business rule class must be specified with the <businessRuleClass> tag in the document's data dictionary file.

Each module contains a 'rules' package where business rule classes should be created.

See Validation and Error Handling 2#Adding Error Messages for information on adding errors if a business rule fails.

Other Examples:

Add Accounting Line Event

```
/**
     *
     * This method implements a custom business rule check that should be used when
adding a new accounting line to a transaction document.  This method
     * overrides the
TransactionalDocument.processCustomAddAccountingLineBusinessRules() method to add any
additional rules or validation calls that may be
     * necessary.
     *
     * @see
org.kuali.module.financial.rules.TransactionalDocumentRuleBase#processCustomAddAccount
ingLineBusinessRules(org.kuali.core.document.TransactionalDocument,
     *      org.kuali.core.bo.AccountingLine)
     */
    @Override
    public boolean processCustomAddAccountingLineBusinessRules(TransactionalDocument
transactionalDocument, AccountingLine accountingLine) {
        boolean allow = true;

        LOG.debug("validating accounting line # " +
accountingLine.getSequenceNumber());

        /* add and validation code inserted here */

        return allow;
    }
```

Route Document Event

```
/**
     * This method implements a custom business rule check that should be called when
the corresponding document type is routed.  This method overrides the
     * TransactionalDocumentRuleBase.processCustomRouteDocumentBusinessRules() method.
Additionally, this method calls the super method to perform any standard
     * validation or rule checks defined by the super, before performing any
additional custom rule checks.
     *
     * @see
org.kuali.module.financial.rules.TransactionalDocumentRuleBase#processCustomRouteDocum
entBusinessRules(org.kuali.core.document.Document)
     */
    @Override
    protected boolean processCustomRouteDocumentBusinessRules(Document document) {
        boolean isValid = super.processCustomRouteDocumentBusinessRules(document);
        BudgetAdjustmentDocument baDocument = (BudgetAdjustmentDocument) document;

        /* perform custom checks here */

        return isValid;
    }
```

Modifying Business Rules

Institutions may find that the default implementation of business rules within KFS is insufficient for their needs. In that case, institutions may hook in their own implementations of the business rules.

This section will override the Chart Maintenance Document's rule as an example for an institution that uses the "edu.sample" package to write its customization code.

Step 1 - Implementing the new business rules

Developers may choose to override a document's rule implementation class, or write a new class from scratch that implements the `BusinessRule` interface.

MyChartRule.java

```
package edu.sample.module.chart.rules;
import org.kuali.module.chart.rules.ChartRule;
import org.kuali.core.document.MaintenanceDocument;

public class MyChartRule extends ChartRule {
    protected boolean processCustomRouteDocumentBusinessRules(MaintenanceDocument
document) {
        // do some validation logic, and add errors to the error map as necessary
    }
}
```

Note above that the new business rule method is prefaced with *processCustom*. This is a convention that should be used for all custom

implementations of business rules. The methods specified by the Rule interfaces (i.e. without the *Custom* in the name) include some basic checks, and delegate the domain-specific checks by calling the *processCustom* method.

Because this file has package `edu.sample.module.chart.rules`, this file should be stored as `<KFS project root>/work/edu/sample/module/chart/rules/MyChartRule.java`.

Step 2 - Create a new Data Dictionary XML file to use the new business rules class

Copy the document's existing data dictionary XML file into the directory for your institution. For our example, the copied XML file should be stored under `<KFS root directory>/work/src/edu/sample/module/chart/datadictionary`. The "org/kuali" has been replaced with "edu/sample". Note the directory that was used, as the same directory name will be needed for Step 3.

Modify the `<businessRulesClass>` tag to specify the newly created class.

Modified Data Dictionary file with new rule class

```
<dictionaryEntry>
    <maintenanceDocument>
        <businessObjectClass>org.kuali.module.chart.bo.Chart</businessObjectClass>

<maintainableClass>org.kuali.core.maintenance.KualiMaintainableImpl</maintainableClass
>


<businessRulesClass>edu.sample.module.chart.rules.MyChartRule</businessRulesClass>
```

Step 3 - Create a overriding Spring module bean definition

Consult the module definition documentation for instructions on how to configure KFS to pick up the newly created data dictionary files.

Writing the rules

Before we start, a note on best practice: often times, save, route, and approve rules are pretty similar. Because of this, many rules classes create a central method, "processMaintenanceRules()" or some such, so that logic can be shared between the processCustomSaveDocumentBusinessRules and processCustomRouteDocumentBusinessRules, which returns a boolean to mark if the rules failed or not. The route rules typically return that boolean value; save rules usually always return true, because even if the rules aren't validating, that's rarely a reason to stop a business object from actually saving.

This practice is illustrated nicely by org.kuali.module.chart.rules.SubObjCdRule. This first method we want to look at is

```
public void setupConvenienceObjects() {
  oldSubObjectCode = (SubObjCd) super.getOldBo();
  newSubObjectCode = (SubObjCd) super.getNewBo();
}
```

This is a method that is called automatically during the rule class initiation. Basically, since instances of rule classes are created once and then not reused, it's fine to mess around with their encapsulated state - in this case, the old subobject code and the new subobject code.

The save rule is fairly simple:

```
protected boolean processCustomSaveDocumentBusinessRules(MaintenanceDocument document)
{
  boolean success = true;
  success &= checkExistenceAndActive();
  return success;
}
```

In this case, if the validation fails, saving the maintenance document should evidently not occur. Therefore, if the validation fails, the rule returns false. Again, SubObjCdRule is kind of different from most rules in preventing the same. Still, we can see that all the rule logic has been deferred to the checkExistenceAndActive() method. The routing rule is pretty similar:

```
protected boolean processCustomRouteDocumentBusinessRules(MaintenanceDocument
document) {
  boolean success = true;
  success &= checkExistenceAndActive();
  return success;
}
```

Extremely similar, actually. SubObjCdRule also has an approval rule which also calls getExistenceAndActive(); the approval method returns true no matter what getExistenceAndActive() returns. We see from this that deferring rule checks to a central method really pays off. If we need to add an extra rule to this maintenance document, we just need to make sure it is checked by getExistenceAndActive() and it will validate no matter what event is occurring to the maintenance document.

Therefore, without further ado, the actual rule, getExistenceAndActive():

```
protected boolean checkExistenceAndActive() {
  boolean success = true;

  // disallow closed accounts unless in certain orgs
  if (ObjectUtils.isNotNull(newSubObjectCode.getAccount())) {
    Account account = newSubObjectCode.getAccount();
    // if the account is closed
    if (account.isAccountClosedIndicator()) {
      putFieldError("accountNumber",
KFSKeyConstants.ERROR_DOCUMENT_SUBOBJECTMAINT_ACCOUNT_MAY_NOT_BE_CLOSED);
      success &= false;
    }
  }
  return success;
}
```

Basically, a rule is like any other predicate method; it's Java code, and therefore, as flexible as KFS gets. In this case, the method checks that the new sub object code has an account, and if it does, it creates an error, using the putFieldError method, into the document if that account is closed. It then returns false if an error occurred.

Just like with any rule, it is important to use the putFieldError method to create an error in the GlobalVariables.getErrorMap() map, so that the user gets a message about what went wrong. MaintenanceDocumentRulesBase has several helper methods that make it easy to add these messages; indeed, as we can see from this example, adding an error on a field requires only that we associate a message with a field (in this case, "accountNumber") and then give the error the key to a property in ApplicationResources.properties that has the correct error message (this is the KFSKeyConstants.ERROR_DOCUMENT_SUBOBJECTMAINT_ACCOUNT_MAY_NOT_BE_CLOSED).

As many rules as needed can be created; indeed, some rule classes, like org.kuali.module.chart.rule.AccountRule are massive and do a great deal of validation. It really depends on the functional needs of the business object maintained by the maintenance document framework.

Events

Requests on a document for which we need to write rules against are wrapped with an 'Event' object, then the business rule class of the document can respond to a type of event by implementing the event's interface method.
All context of the event (e.g. Document, AccountingLine, ...) are contained in the Event object which is passed to the implementing method of the business rule class.

Event Methods

```
/**
     * Returns the interface that classes must implement to recieve this event.
     *
     * @return
     */
    public Class getRuleInterfaceClass();

    /**
     * Invokes the event handling method on the rule object.
     *
     * @param rule
     * @return
     */
    public boolean invokeRuleMethod(BusinessRule rule);

    /**
     * This will return a list of events that are spawned from this event.
     *
     * @return
     */
    public List generateEvents();
```

Rules on documents validate the data in the document when certain events occur: when the document is submitted for routing, for instance, or saved. Transactional documents have access to very low-level rules, so let's take a look at those rules.

The base events

What are the base events that occur to every document, transactional or otherwise? They are the workflow events: saving a document, submitting a document to routing, approving a document, disapproving a document, and canceling a document. Since checking rules on a document after a cancel or disapprove only tends to annoy users, there is no rule checking for those events. However, rules do get validated for the other three.

org.kuali.core.rules.TransactionalDocumentRuleBase extends org.kuali.core.rules.DocumentRuleBase directly, so let's take a look at DocumentRuleBase. We note these methods:

```
public boolean processSaveDocument(Document document);

public boolean processRouteDocument(Document document);

public boolean processApproveDocument(ApproveDocumentEvent approveEvent);
```

And while approve document seems a bit different from the other two, we at least recognize these as methods corresponding to the base events we care about: save, route, and approve. Excellent! So, in the rules classes for our documents, we'll just override those methods, right?

No. That's kind of prone towards error. We want to make sure that we call the **super**.processSaveDocument method or whatnot, and the rules

classes have an easier way. Instead, we just call these methods:

```
protected boolean processCustomSaveDocumentBusinessRules(Document document);

protected boolean processCustomRouteDocumentBusinessRules(Document document);

protected boolean processCustomApproveDocumentBusinessRules(ApproveDocumentEvent
approveEvent);
```

Each of these methods are called by their corresponding process*Document method, and so there is no need to call **super** on these methods to get the parent's rules to validate as well. (Naturally, if the parent also implemented a processCustom*DocumentBusinessRules method, then we would have to call **super**; but that's not prone to error because...well, inheritance is a difficult business).

What does a rule do? Let's look at a simple route rule, defined in org.kuali.module.financial.rules.AdvanceDepositDocumentRule:

```
1.  protected boolean processCustomRouteDocumentBusinessRules(Document document) {
2.    boolean isValid = super.processCustomRouteDocumentBusinessRules(document);

3.    if (isValid) {
4.      isValid = isMinimumNumberOfAdvanceDepositsMet(document);
5.    }

6.    return isValid;
7.  }
```

What's happening in this rule? First of all, we can see that all of our rules methods return booleans. The boolean that is returned tells the document framework whether to continue on with its current operation or not. For instance, when we've got a routing rule, such as the one above, and that rule fails validation, then we don't want the document to route. And in that case, we'd return a **false** from the method. If we want the document framework to continue with the current event, then we return a **true**. Save rules often return **true**, simply because even if the validation failed, we often want to make certain that the document was persisted.

In this example, we see that the programmer was cautious; **super**.processCustomRouteDocumentBusinessRules gets called, just in case some logic is in it, and the result of that validation is stored in the "isValid" variable. If the **super** custom routing rules returned a true, then isValid is checked against another method, isMinimumNumberOfAdvanceDepositsMet(), and whatever the result of that is, isValid is set to. Then, the validation state isValid is returned from the method.

The actual rule "logic" is separated out into the isMinimumNumberOfAdvanceDepositsMet() method:

```
1.  private boolean isMinimumNumberOfAdvanceDepositsMet(Document document) {
2.    AdvanceDepositDocument ad = (AdvanceDepositDocument) document;

3.    if (ad.getAdvanceDeposits().size() == 0) {
4.      GlobalVariables.getErrorMap().putError(DOCUMENT_ERROR_PREFIX,
KFSKeyConstants.AdvanceDeposit.ERROR_DOCUMENT_ADVANCE_DEPOSIT_REQ_NUMBER_DEPOSITS_NOT_
MET);
5.      return false;
6.    }
7.    else {
8.      return true;
9.    }
10. }
```

Moving the rule logic to separate methods is a smart idea - just ask Martin Fowler. He's not here, however, so let's talk about why it's a good idea ourselves. First of all, it splits our code into more cleanly defined conceptual entities, which is always nice. Importantly, however, separating the

rule logic means that it can be easily called in multiple places. For instance, if the same rule is to be called in both the route rules and the approve rules, it's a lot easier to have it separated into a method, such as isMinimumNumberOfAdvanceDepositsMet in our example, that can be called by both.

Here, we see that we need at least one advance deposit on the document. If there aren't any advance deposits, then we return false; otherwise we return true, meaning that routing can actually occur. We see on failure though, one other thing this rule is doing:

```
GlobalVariables.getErrorMap().putError(DOCUMENT_ERROR_PREFIX,
KFSKeyConstants.AdvanceDeposit.ERROR_DOCUMENT_ADVANCE_DEPOSIT_REQ_NUMBER_DEPOSITS_NOT_
MET);
```

Obviously, if a validation fails, we want the user to know about that. In KFS 2.0, we do that by accessing the errorMap in the GlobalVariables thread local singleton, and then adding an error to it. The putError method takes the property name of the property "most responsible" for the error (the message will be shown on that property's tab and the property itself will be highlighted), and the key to the resource property defined in ApplicationResources.properties of the error message. Any other String parameters passed in to this method will be interpolated into the error message. In this example, the error will be shown in the "document." property, which will put the message at the very top of the screen. The error message property is named "error.document.advanceDeposit.requiredNumberOfAdvanceDepositsNotMet", which, when we look at ApplicationResources.properties is "This document must contain at least one advance deposit before it can be submitted." Having put the error into the errorMap of GlobalVariables and returning false, the rule is done. That's all that has to be done for a rule.

A final note: when a route or approve event occurs, a save event occurs simultaneously. Therefore, rules in processCustomSaveDocumentBusinessRules() will be called in the event of a processCustomRouteDocumentBusinessRules() or processCustomApproveDocumentBusinessRules() as well. However, often, even if validation fails, we very often want the document to be saved in the persistence store. Therefore, as noted, save rules often return true, no matter the result of the validation. Because of this, several documents perform similar checks in the route rules as they do in the save rules, with the difference being that the route rules return false when the validation fails.

Not-so-base events

There are three other events which we can add custom rules to:

```
protected boolean processCustomAddNoteBusinessRules(Document document, Note note);

protected boolean processCustomAddAdHocRoutePersonBusinessRules(Document document,
AdHocRoutePerson person);

protected boolean processCustomAddAdHocRouteWorkgroupBusinessRules(Document document,
AdHocRouteWorkgroup workgroup;
```

These events are a bit more obtuse; in fact, none of these three are ever used by any transactional document in KFS. Therefore, this discussion will be short, though these rules can be customized as easily as the save, route, and approve rules.

All transactional and maintenance documents give the ability for users to add notes, so that the document can be annotated as it passes through workflow. The processCustomAddNoteBusinessRules gets called whenever a new note is added to a document.

Instances of transactional and maintenance documents can be given ad-hoc routes too. For instance, a new accountant may want to send an accounting document to the person showing them the ropes, rather than straight to the account fiscal officers. To allow this, documents have "ad hoc" routes - routes that aren't typical but exist just for the given document. We can add either a person to route the document to or a workgroup to add the document to, and the processCustomAddAdHocRoutePersonBusinessRules and processCustomAddAdHocRouteWorkgroupBusinessRules rules let us validate those ad-hoc additions.

Again, none of these are being used by KFS currently...but there's always the future.

Inventing new events

What if we want our document to validate events that don't happen at the given event points? This is an important question with transactional documents; as we'll see when we look at how to define the User Interface, we can create new buttons. Transactional documents invent new events. How do we get those new events to live within the framework?

A good example of this can be seen in the Cash Receipt document. Cash Receipt documents have a collection of checks; checks can, therefore, be added, updated, and deleted in the document, and the Cash Receipt document wanted rules to make sure that, for instance, made sure a newly added check wasn't for a negative value. The programmers behind Cash Receipt had to create a new set of events.

There are three parts to creating a new rule for a new event. First of all, a rule interface must be defined. This should declare what method should be called on the business object class when the event occurs. Naturally, the rules class for the documents that have this event will need to provide an implementation. Second, an event class must be created. Finally, when the appropriate action occurs, the new event must be sent to an implementation of KualiRuleService, which calls the proper rules on the document for the given event. Let's look at how this works for the check events of Cash Receipt.

First, since there are three events to have rules for, there is an interface to kind of group all of those events together:

```
1. public interface CheckRule {
2. }
```

This isn't a necessary step, but it is a nice practice. Things get more interesting when we look at the org.kuali.module.financial.rule.AddCheckRule interface:

```
1. public interface AddCheckRule<F extends AccountingDocument> extends CheckRule {
2.    public boolean processAddCheckBusinessRules(F financialDocument, Check check);
3. }
```

The interface extends CheckRule and genericizes the fact that it is meant to work on a descendant of AccountingDocument. The interface then declares one method, processAddCheckBusinessRules, which takes in a financial document which extends AccountingDocument and a Check. As a note, most Rule interfaces declare one rule method.

This method is implemented in org.kuali.module.financial.rules.CashReceiptDocumentRule:

```
1. public boolean processAddCheckBusinessRules(AccountingDocument financialDocument,
Check check) {
2.   boolean isValid = validateCheck(check);

3.   return isValid;
4. }
```

The validateCheck method simply makes sure that the amount on the new check is not zero and is not negative:

```
1.  private boolean validateCheck(Check check) {
      // validate the specific check coming in
2.
SpringContext.getBean(DictionaryValidationService.class).validateBusinessObject(check)
;

3.    boolean isValid = GlobalVariables.getErrorMap().isEmpty();

      // check to make sure the amount is also valid
4.    if (check.getAmount().isZero()) {
5.      GlobalVariables.getErrorMap().putError(KFSPropertyConstants.CHECK_AMOUNT,
KFSKeyConstants.CashReceipt.ERROR_ZERO_CHECK_AMOUNT, KFSPropertyConstants.CHECKS);
6.      isValid = false;
7.    }
8.    else if (check.getAmount().isNegative()) {
9.      GlobalVariables.getErrorMap().putError(KFSPropertyConstants.CHECK_AMOUNT,
KFSKeyConstants.CashReceipt.ERROR_NEGATIVE_CHECK_AMOUNT, KFSPropertyConstants.CHECKS);
10.     isValid = false;
11.   }

12.   return isValid;
13. }
```

Again, a nice example of separating a rule's logic into a separate method.

The rule will get caused by an org.kuali.module.financial.rule.event.AddCheckEvent. Rule events ultimately have to implement the org.kuali.core.rule.event.KualiDocumentEvent, which has an abstract base implementation: org.kuali.core.rule.event.KualiDocumentEventBase.

Note: Name your new rule class with something that describes the action this event will represent, and append the name with Event.

Several of KualiDocumentEvent's methods are implemented by KualiDocumentEventBase. This is useful if your event only needs access to the document. If your event needs to maintain the state of another object, for instance a new account line, you should create that member inside your event along with the necessary constructors, getters, and setters, so we'll want to inherit from that. Those methods are:

- getDocument() - every rule event occurs to a document, and therefore, the event holds a pointer to the document that caused the rule event to be fired.
- getDescription() - a brief description of the event
- getName() - this returns the name of the event class itself
- getErrorPathPrefix() - this is the property prefix for the property where the event occurred
- validate() - this validates the event itself. KualiDocumentEventBase's implementation simply checks that the document isn't null at the time of event validation; if the document is null, it throws a big, fat IllegalArgumentException. Many other events override this method (making sure to call **super**), adding more validation.
- generateEvents() - an event may cause other events to fire. We've already seen an example of this: org.kuali.core.rule.event.RouteDocumentEvent generates a save event using this method. The default implementation returns an empty list.

Great. That leaves us two really important methods that our event classes must implement:

- getRuleInterfaceClass() - this returns the class of the related rule interface. So, for instance, the AddCheckEvent returns AddCheckRule. **class**.
- invokeRuleMethod() - this method is given a rule class as a parameter and then lets the check event invoke the proper rule method on the document. *Note: The rule class has already been checked by the rules service that it implements the interface for this event, so you can safely cast the rule object to the interface and call the interface method.*

Now, we have an event. But that's only half the battle. To instantiate the event, you'll need to make a call to KualiRuleService to get things rolling.

- You have to instantiate your Event manually, in the code (most likely in the action method), and then pass it into a call to applyRules(). Do this wherever you feel it's appropriate.

```
boolean rulePassed = SpringServiceLocator.getKualiRuleService().applyRules(new
InsertPeriodLineEventBase(budgetForm.getDocument(), budgetForm.getNewPeriod()));
```

You have a fully functioning new event. But don't forget to add the error tags to the corresponding JSP pages.

- Make sure you include the error tags on the JSP corresponding to the new messages you added.

All of this, though, begs a question: how do we get the event to force the rule to be called? I'm personally not ready to tackle that very tricky question right now, but I have no doubt that we'll take a look at that in transactional document Action classes. Somewhere along the way, we've got to look at how to prevent users from entering bad data without rules - by using document authorizers.

Some Events have already been set up by other developers, both in the Core project and in specific modules. So before you create a new Event, make sure one has not already been created for the action you want to capture. If it has, you're in luck - you can skip this step. If not, go ahead and create your Event.

Unable to render {include} The included page could not be found.