Document Authorizer 2

Rules validate after a user has entered data into a document and performed an event on the document. However, there are some actions that we don't want a user to do in the first place. For instance, let's say that a certain user cannot create a transactional document like the Cash Management document. If we wait until the user submits the document to validate that the user had permission to work on the document in the first place, we are bound to end up with some very frustrated users. What we need for this are not validations, but guards, to prevent the wrong action from taking place in the beginning. Thankfully, we have a class of guards, which are called document authorizers. Here, we explore what document authorizers do and how they prevent wrong events from happening.

Transactional documents will extend org.kuali.core.document.authorization.TransactionalDocumentAuthorizerBase. TransactionalDocumentAuthorizerBase in turn extends org.kuali.core.document.authorization.DocumentAuthorizerBase, which is the default implementation of org.kuali.core.document.authorization.DocumentAuthorizer. Because of this, some of the base implementations may be in TransactionalDocumentAuthorizerBase and some may be in DocumentAuthorizerBase; we'll cover the major methods in both, so that we know everything we can take advantage of in our own document authorizers.

Document, may I?

DocumentAuthorizer declares three methods that all return booleans: canInitiate(), canCopy(), and canViewAttachement(). canInitiate() and canCopy() are both passed the KEW document type name of a document and a UniversalUser record for the user trying to initiate or copy; canViewAttachment() is passed an attachment type name - the mime type of the attachment to view, a specific document with an attachment, and, once again, the UniversalUser record of the user attempting to view the attachment.

DocumentAuthorizerBase, then, has a default implementation to see if the given user can perform the requested action. Naturally, we can override that...but the DocumentAuthorizerBase does its check in a highly convenient way: it allows us to set up copy and initiate workgroups in the data dictionary. We'll look a bit more at authorizations when we go through the data dictionary, but we can take a quick look now at the general syntax. Here's the initiation authorization for the Cash Receipt document (well, and many others...):

```
<authorizations>
  <authorization action="initiate">
    <workgroups>
      <workgroup>kualiUniversalGroup</workgroup>
    </workgroups>
  </authorization>
</authorizations>
```

We can define multiple authorizations within the data dictionary, though each authorization has an associated action. Each authorization also has a workgroup; initiators must be a member of that workgroup. Naturally, here, we're using a special workgroup: kualiUniversalGroup, which means "everyone in the system." If we wanted to lock initialization of this document down a bit, though, we'd force initiators to be in a given specific workgroup.

By default, DocumentAuthorizerBase uses the authorizations for initialization as the authorization for copy as well.

canViewAttachment() currently isn't overridden in KFS, and the default implementation in KualiDocumentBase always returns true, so any user can see any attachment. Again, this need not be the case for different institutions with different needs.

getDocumentActionFlags()

Another common point of guarding users from doing the wrong actions is to prevent them from saving or submitting a document for routing before the document is "ready." DocumentAuthorizer provides a way to turn on and off buttons and even information areas on a document: getDocumentActionFlags(). The method is passed a document and the current user as parameters, and it returns an object of type org.kuali.core.document.authorization.DocumentActionFlags. DocumentActionFlags declares a number of getters that return booleans; if the boolean is true, then the document framework will display that button or section of information and if it is false, it won't. Here are the flags that can be set on DocumentActionFlags:

- canReload
- canSave
- canRoute
- canCancel
- canClose
- canBlanketApprove

- canApprove
- canDisapprove
- canFYI
- canCopy
- canAcknowledge
- canAnnotate
- canAdHocRoute
- canSupervise
- canPerformRouteReport
- hasAmountTotal

canReload, canSave, canRoute, canCancel, canClose, canBlanketApprove, canApprove, canDisapprove, canFYI, canAcknowledge, and canCopy all refer to the normal document action buttons on the bottom of every document - maintenance or transactional; setting these booleans allows us to turn on or off the buttons. The code in documentControls.tag is the primary users of these tags and renders the buttons.

canAnnotate allows us to add notes to a document or not. canAdHocRoute decides whether we can add ad-hoc routing to a document. canPerformRouteReport decides whether the document can show its routing tab - the trail of the document in workflow. canSupervise turns on and off the Supervisor Functions button.

Finally, hasAmountTotal tells the document framework if it should show a total for the document in the header tab.

Let's look at how an actual transactional document uses this, specifically the Research Routing Form:

```
1.  public DocumentActionFlags getDocumentActionFlags(Document document, UniversalUser
user) {

2.    DocumentActionFlags flags = super.getDocumentActionFlags(document, user);
3.    RoutingFormDocument routingFormDocument = (RoutingFormDocument) document;

4.    flags.setCanAcknowledge(false);

5.    if (!flags.getCanRoute() &&
routingFormDocument.isUserProjectDirector(user.getPersonUniversalIdentifier())
          &&
routingFormDocument.getDocumentHeader().getWorkflowDocument().stateIsSaved()) {
6.      flags.setCanRoute(true);
7.    }

8.    flags.setCanBlanketApprove(false);
9.    flags.setCanCancel(false);

10.   flags.setCanFYI(false);
11.   flags.setCanClose(false);
12.   flags.setCanSave(true);

13.   return flags;
14. }
```

Here, we can see that the Research Routing Form is pretty restrictive. On line 2, a new set of DocumentActionFlags is created from the **super** ver sion of the method. Then, on line 3, the document the method has been handed is cast to a RoutingFormDocument. On line 4 and lines 8 - 11, the acknowledge, blanket approve, cancel, FYI, and close flags are turned off always, while on line 12, the save flag is turned on always. On lines 5 - 7, we see the power of the method, because the method actively checks the document to see if it can be routed or not. If the **super** method has turned off the ability to route, the logic checks that the currently logged in user is a project director for the document and that the document has been saved in workflow; if that's true, then the logic allows routing anyway.

If a certain document defines a new action (e.g. return to sender), both the document action flags and the document authorizer classes may be extended to accommodate it. For example, the subclassed document action flags class will implement the methods `getCanReturnToSender()` and `setCanReturnToSender(boolean)`, and the authorizer's getDocumentActionFlags() method will set the value for that property appropriately.

getEditMode()

The best way to introduce getEditMode() is by acknowledging that it is one weird method. Accept this, for it is the only way to DocumentAuthorizer

enlightenment. Breathe in: it's weird! Breathe out: but we can handle it, and, anyway, it's used a lot in KFS. Very good.

getEditMode()'s purpose is to control document-granularity editing. For instance, let's say that a document is editable by the initiator, but isn't editable by any approvers as the document goes through workflow. For the initiator, there would be a "fullEntry" edit mode and for the approvers, there would be an "viewOnly" edit mode. Understandable? Great. Because it's nowhere near that easy, but that's the gist at least.

getEditMode() takes in as parameters a document and the UniversalUser record of the user attempting to access the document. It returns a Map. This Map has an entry for every edit mode in place on the current document for the current user. The edit mode forms a key. The value of the Map can be anything that won't be interpreted as "false" by JSTL - no empty Strings or Collections, numbers that equal 0, or nulls, please. Definitely not a boolean **false**. But other than that, anything goes.

We can find a full list of edit mode keys at org.kuali.core.authorization.AuthorizationConstants. Here's the ones we care about for all transactional documents:

- AuthorizationConstants.EditMode.UNVIEWABLE - this means that for the current user, the document as a whole cannot be viewed
- AuthorizationConstants.EditMode.VIEW_ONLY - this means that the current user can view the current document, but that's it; they can't edit any fields
- AuthorizationConstants.EditMode.FULL_ENTRY - this gives the user full power over the transactional document. They can edit. They can change. They burn and pillage. But naturally, without granting some user that much power, there wouldn't be any documents in the first place.
- AuthorizationConstants.TransactionalEditMode.EXPENSE_ENTRY and AuthorizationConstants.TransactionalEditMode.EXPENSE_SPECIAL_ENTRY - these are special modes created for purchasing and financial documents. For instance, the Disbursement Voucher document allows certain approvers to edit expense lines on the document, and therefore, there's an edit mode for EXPENSE_SPECIAL_ENTRY set for those circumstances.

You can have multiple edit modes, especially considering that different document types have created their own edit modes. In AuthorizationConstants, we can see that the Disbursement Voucher, Cash Management, Budget Adjustment, and Budget Construction Editing documents have all declared their own edit mode constants, and those are meant to be used in addition to the constants listed above. Naturally, if we return a Map of edit modes that declares both VIEW_ONLY and FULL_ENTRY, we're going to end up with some pretty inconsistent behavior, so it's likely a good idea to avoid that.

Another use of edit modes are to control masking of sensitive fields on maintenance documents. Sensitive fields on maintenance documents are associated with edit modes that do not necessarily correspond to the ones listed above. When a sensitive field is rendered, the framework will either render the field in plain text if the field edit mode is in the map, or render a masked value if the edit mode is not in the map. See the documentation about masking for more details.

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