Lookups 2

The lookup screen provides the ability to search for business objects using criteria. Lookup screens also provide the ability to create a new business object record, to edit or copy an existing record, and to drill down to obtain more information about a record. Naturally, the Kuali Nervous System provides ways to customize the lookup screen as well - Lookup Helpers. Here, we'll discover the common ways Lookup screens are declared andhow we can use a custom lookup for one of our own business objects.

Declaring a Lookup in the Data Dictionary

The lookup definition indicates which fields are used to perform a lookup, as well as which fields are listed in the search results. Lookup screens are those that are used to search for a particular row in a table.

```
<lookup>
    <lookupableID>accountLookupable</lookupableID>
    <title>Account Lookup</title>

    <menubar>
        <![CDATA[<a
href="maintenance.do?methodToCall=start&businessObjectClassName=org.kuali.module.chart
.bo.AccountGlobal"><img
src="${kr.externalizable.images.url}tinybutton-createnewglobal.gif" alt="create new
global" width="98" height="15"/></a>]]>
    </menubar>

    <instructions>Lookup an Account</instructions>

    <defaultSort attributeName="accountNumber" sortAscending="true" />

    <lookupFields>
        <lookupField attributeName="chartOfAccountsCode" required="false" />
        <lookupField attributeName="accountNumber" required="false" noLookup="true" />
        <lookupField attributeName="accountName" required="false" />
        <lookupField attributeName="accountFiscalOfficerSystemIdentifier"
required="false" />
        <lookupField attributeName="accountFiscalOfficerUser.personName"
required="false" />
    </lookupFields>

    <resultFields>
        <field attributeName="chartOfAccountsCode" />
        <field attributeName="accountNumber" />
        <field attributeName="accountName" />
        <field attributeName="accountFiscalOfficerUser.personUserIdentifier" />
        <field attributeName="accountFiscalOfficerUser.personName" forceInquiry="true"
/>
    </resultFields>
</lookup>
```

Like in the inquiry definition, the title is used as the page title. Instructions are used to generate help text when the user clicks on the help icon.

The `<lookupableID>` specifies which Spring bean implementing `Lookupable` is used to perform the lookup on the BO.

The `<defaultSort>` tag is used to indicate how the search results will be initially sorted. Consult DTD documentation to learn about how to specify multiple sorting attributes.

The `<lookupFields>` section is used to indicate what the search criteria are on the lookup screen.

The <resultFields> section indicates which fields will be rendered in the search results.

Like inquiries, note that a lookup field and result field may be the property of a reference. The system will perform the joins necessary, provided references are defined in OJB or the data dictionary.

There are other advanced options when defining lookups, including how to define default values for a field. Please consult the KNS file dataDictionary-1.0.dtd for more details.

Common Lookup Helpers Customizations

A lookup helper is simply a class that implements `org.kuali.core.lookup.LookupableHelperService`. `LookupableHelperService` defines a hefty twenty-eight methods to define, though. As we'll see, we don't have redefine all of those, and definitely, some are more used than others. Let's take a look at the most customized `LookupableHelperService` methods.

- getBusinessObjectClass() and setBusinessObjectClass() - lookups have a specific business object to look up. These methods obviously allow the `LookupableHelperService` implementation to set and get the class of the business object that will be looked up.
- getSearchResults() and getSearchResultsUnbounded() are the heart of the lookupable helper service. They actually go to the persistence store and return the results, based on a passed-in Map of field values to search on (for most searches, if the primary keys for the business object class are filled in, the specified business object will be returned (if it exists), and the other fields will be ignored). The difference between getSearchResults() and getSearchResultsUnbounded() is that getSearchResults() will only return a certain number of results, set by the parameter KR-NS / Lookup / RESULTS_LIMIT, while getSearchResultsUnbounded() will return everything.
- getMaintenanceUrl() returns the url of any associated maintenance document. Implementations of this method should find the maintenance document if it exists and then return the URL for the "Create New" button.
- getActionUrls() is similar to getMaintenanceUrl() is that it returns URLs to show for each result row, by default the "edit" and "copy" links. However, this just returns a big String with links in it, so we can return whatever "action" links we want to from this method. For instance, if we've got one of those weird situations where a "setup new from existing" action is warranted, we must override this method to create a link for that action. The action itself, though, is very flexible and many custom lookup helpers exist just to change this method.
- getReturnUrl() returns what the "return value" URL should be for a business object in a given result row, so that the business object can be returned to another document.
- getInquiryUrl() is handed the business object for a result row and a name of a property in that business object, and implementations of this method should determine what the URL for a inquiry link should be for the given property.

Together, these methods provide a lot of functionality for custom lookups to change. However, there's a lot of methods to implement in `LookupableHelperService`. We could avoid reduplicating code, if only we had something to inherit from....

Extending Lookupable Helpers

Thankfully, we do have something to inherit from: two helper classes that we can extend, `org.kuali.core.lookup.AbstractLookupableHelperServiceImpl` and `org.kuali.core.lookup.KualiLookupableHelperServiceImpl`. We'll also take a look at `LookupUtils`, which has handy static methods for looking up business objects.

`AbstractLookupableHelperServiceImpl` implements a large portion of the functionality required by `LookupableHelperService` (27 of the 28 methods, though the implementation of getSearchResultsUnbounded() is somewhat... deficient). The method that it doesn't implement - forcing the class to live up to its name and be **abstract** - is getSearchResults(). However, default implementations of the rest of `LookupableHelperService`'s methods exist within `AbstractLookupableHelperServiceImpl`.

`KualiLookupableHelperServiceImpl extends AbstractLookupableHelperServiceImpl` to make a concrete class - that is, it implements getSearchResults() and reimplements a more functional version of getSearchResultsUnbounded(). It defers, ultimately, to the default implementation of `org.kuali.core.service.LookupService`, which does basic searching in the persistence store, using OJB queries.

Typical customized lookup helpers should inherit from `KualiLookupableHelperServiceImpl` - typically, a persistence store search is what is desired in a lookup. However, let's say that we instead wanted a special custom lookup that used a Lucene store as its search source. If that was the case, then surely we'd extend `AbstractLookupableHelperServiceImpl`, since our getSearchResults() method would be a very different beast than `KualiLookupableHelperServiceImpl`'s.

Both `KualiLookupableHelperServiceImpl` and `AbstractLookupableHelperServiceImpl` defer a lot to the utility methods in `org.kuali.core.lookup.LookupUtils`. `LookupUtils` has methods for setting the proper quick finders for form attributes, returning nested business objects, and translating conversion fields.

Tying a custom lookup helper to a business object

You're half right when you guessed this had something to do with the data dictionary; there's some Spring configuration involved as well. Let's take a look at how the `org.kuali.module.chart.lookup.KualiAccountLookupableHelperServiceImpl` is used.

Since `KualiAccountLookupableHelperServiceImpl` is associated with accounts, let's look at the Account business object data dictionary configuration in org/kuali/module/chart/datadictionary. In that file, we find the following lines:

```
<lookup>
  <lookupableID>accountLookupable</lookupableID>
  <title>Account Lookup</title>
```

That lookupableID tag *seems* like where we should specify our lookupable helper service. However, accountLookup obviously isn't the name of the class, so what is it? To answer that mystery, let's take a look in org/kuali/module/chart/KualiSpringBeansChart.xml. There, we find the following bean definitions:

```
<bean id="accountLookupableHelperService"
class="org.kuali.module.chart.lookup.KualiAccountLookupableHelperServiceImpl"
 singleton="false" parent="lookupableHelperService" />

<bean id="accountLookupable" class="org.kuali.core.lookup.KualiLookupableImpl"
singleton="false">
  <property name="lookupableHelperService">
    <ref bean="accountLookupableHelperService" />
  </property>
</bean>
```

The original implementors of this feature were indeed legally prevented from making it easy. But, even given the government-enforced contortions, it's not too terribly hard. First of all, we need to create a bean for our lookupable helper class; this is the bean named "accountLookupableHelperService". Notice that it is not a singleton, and that it has a parent set, "lookupableHelperService"; when we make our own definitions, we will need the same configuration for those two properties. Then, we create the bean instance of `KualiLookupableImpl`, with our "accountLookupableHelperService" injected into its "lookupableHelperService" property. This bean has the name "accountLookupable," so naturally, this is the bean name that we set in the data dictionary. Having done that, we've tied our custom lookupable into the Lookup screen.

Altering Criteria

In the simplest case, all you may need to do is to change the lookup criteria. All criteria is passed in through the fieldValues Map. This Map is keyed by the property name (relative from the business object class). If you add entries to this map, the key must match a reachable property on the business object, (If the property contains a "." (a nested property), then there must be relationships defined in the ORM mapping layer or in the data dictionary for the base attribute.)

When putting values into the Map, be sure to use the proper case for the key. You can also remove values from the map as normal to prevent them from being used.

After manipulating the map, call `super.getSearchResults( fieldValues )` if you are able to use the base lookup functionality. If there's nothing else to do, return that results.

If you need to create a list to be used by the lookup service, you can append multiple values together in a single property with "|" as a delimiter. There are other special characters which can be used as well. See the table below:

Character	**Meaning**
|	or
&&	and
!	not
>	greater than
<	less than
>=	greater than or equal to

<=	less than or equal to
..	between
*	Any substring
_	Any single character

Examples
abc|def = abc OR def
!abc!def = NOT abc AND NOT def
>1000&&<10000 = greater than 1,000 AND less than 10,000
caaaaa..dzzzzzz = string between caaaaa AND dzzzzzz

Handling Results

Performing post-filtering on results must be done with care. If you use the default lookup service first, the results have already been limited to the configured maximum. You may want to use the `super.getSearchResultsUnbounded(fieldValues)` instead. However, you will need to make sure before allowing such an operation that there will not be too many results returned (such as all GL entries). Otherwise, you could crash the server.

You could also use the `BusinessObjectService`'s `findMatching()` method to get results. But, when filtering these to the given criteria, you will not be able to duplicate all the special functionality of the lookup service. Fortunately, most users don't use those. So, if you replace all instances of "**" in the string with ".**" and use regular expression matching, it will mostly behave like the users expect.

Performance Tip
If you use regular expressions to filter your results, generate the needed Pattern objects once prior to looping rather than using the matches() method on java.util.String. Creating the pattern can be an expensive process and only needs to be done once per search string.

Filtering Example

```
/* Before the loop */
if (StringUtils.isNotBlank(fieldValues.get("parameterDetailTypeCode"))) {
    String patternStr = fieldValues.get("parameterDetailTypeCode").replace("*",
".*").toUpperCase();
    try {
        detailTypeRegex = Pattern.compile(patternStr);
    }
    catch (PatternSyntaxException ex) {
        LOG.error("Unable to parse parameterDetailTypeCode pattern, ignoring.", ex);
    }
}
/* loop over the results */
for (ParameterDetailType pdt : components) {
    boolean includeType = true;
    if (detailTypeRegex != null) {
        includeType =
detailTypeRegex.matcher(pdt.getParameterDetailTypeCode().toUpperCase()).matches();
    }
    if (includeType) {
        if (totalCount < maxResultsCount) {
            baseLookup.add(pdt);
        }
        totalCount++;
    }
}
```

Limiting Results

If you are unable to use the lookup service (`super.getSearchResults(fieldValues)`) for the final results, then you should manually truncate the results to the configured size. You can get that value by calling `LookupUtils.getApplicationSearchResultsLimit()`.

However, if your results are greater than that maximum, you should wrap your truncated results in a `CollectionIncomplete` class. This class holds your results (it implements the `List` interface) and provides a setter for the full number of results. This is what allows the lookups to report that: 50,000 rows matched, 1,200 returned...

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