The SearchPanel component (replacement for LookupForm)

Using the SearchPanel

When data is stored in a database we want to be able to search for records. Searching is so common that DomUI has a special component that helps with creating database search screens: the SearchPanel. The Search Panel works on database *objects*, i.e. entity classes that are defined in Hibernate or JPA, or even with plain JDBC accessed objects (using DomUI's generic database layer). This means that working with the panel you stay inside the Java world.

Let's start with an example:

```
@Override public void createContent() throws Exception {
   ContentPanel cp = new ContentPanel();
   add(cp);

   SearchPanel<Invoice> lf = new SearchPanel<>(Invoice.class);
   cp.add(lf);
   lf.setClicked(a -> search(lf));

   lf.add().property("customer").control();        // Start with lookup by customer
   lf.add().property("total").control();           // Allow searching for a total
   lf.add().property("invoiceDate").control();     // And the date.
}
```

This example creates a search panel which searches for Invoice instances using the customer, total (amount) and invoiceDate properties. The actual searching and showing of the data is done by base class which will be shown at the end of this document.

This fragment creates the following UI (LIVE - click inside to play with it):

As can be seen, each property is presented on the form, in order. And for each property we have a special "control" which allows for input related to the type of the property. We see, in order:

- The customer property uses a LookupInput2, because it is actually a @ManyToOne relation (parent) of the Invoice class. This control lets you search for a Customer using a special UI.
- The "total" field is a numeric field. It uses a special *Lookup Control* called *NumberLookupControl* which allows searching for a number as follows:
  - Enter "> 1000" to find all records with an amount > 1000. Likewise you can use <, <=, >=.
  - Entering a single amount searches for an exact match
  - Entering 10% does a "like" query with all amounts that start by 10.
- The invoiceDate field creates another *Lookup Control* called *DateLookupControl*. This control shows two dates, and allows searched starting from, ending at or between the two dates.

The SearchPanel uses metadata to get the default label for properties, and it uses a registry of factories (LookupControlRegistry2) to find the best lookup control for a property, by type. The registry can be easily extended with your own lookup control factories.

You can control how data is shown using the builder pattern exhibited above. In that way you can change:

- The label by using label(String) or label(NodeContainer)
- The lookup hint (what is shown when hovering over the control) which also defaults to metadata
- The default value to use. This value will be used as the initial value of the control, and will also be used when the "reset" button is pressed on the panel.
- Whether a text search ignores case or not (defaults to true)
- The minimal input length for a control before the search is issued. This can be used to prevent large searches by specifying that at least 3 characters should be used for instance.
- Options specific for the control being created.

Using metadata with the panel

In the above example we specified what to search on by hand. This is often handy because it allows full control. But the form can also be populated automatically by using the metadata associated with the entity we look for. Take for example the following definition for Invoice's metadata:

```
@Entity
@Table(name = "Invoice")
@SequenceGenerator(name = "sq", sequenceName = "invoice_sq")
@MetaObject(defaultColumns = {                      // 20180203 Must have metadata for SearchPanel/LookupForm
tests.
   @MetaDisplayProperty(name = "customer.lastName", displayLength = 20)
   , @MetaDisplayProperty(name = "customer.firstName", displayLength = 10)
   , @MetaDisplayProperty(name = "invoiceDate")
   , @MetaDisplayProperty(name = "billingAddress", displayLength = 20)
   , @MetaDisplayProperty(name = "billingCity", displayLength = 10)
   , @MetaDisplayProperty(name = "total", displayLength = 10)
}
, searchProperties = {
   @MetaSearchItem(name = "invoiceDate")
   , @MetaSearchItem(name = "billingCity")
   , @MetaSearchItem(name = "customer")
}
)
```

The searchProperties above are the "default" properties to use inside the SearchPanel, and they take effect when no manual configuration of the SearchPanel is done, like this:

```
@Override public void createContent() throws Exception {
   ContentPanel cp = new ContentPanel();
   add(cp);

   SearchPanel<Invoice> lf = new SearchPanel<>(Invoice.class);
   cp.add(lf);
   lf.setClicked(a -> search(lf.getCriteria()));
}
```

In this case the metadata takes effect, resulting in:

<h3 id="TheSearchPanelcomponent(replacementforLookupForm)-Mixingmetadataanduserconfiguration">Mixing metadata and user configuration</h3> By default the SearchPanel does not use metadata when the panel is configured manually. To combine metadata with manually added controls call the following: <pre class="syntaxhighlighter-pre" data-syntaxhighlighter-params="brush: java; gutter: false; theme: Confluence" data-theme="Confluence">lf.addDefault();</pre> This adds the metadata to the definition as follows: When this call is done the metadata defined search items are added after what is already configured
When a user defined item is added for a given property then metadata for that same property is not used. This acts a bit special:
If the user defined item has been added before the call to addDefault() then the metadata items are added behind all defined items, and the metadata item with the existing property is just skipped
If the user defined item is added after the metadata items have been added (so after the call to addDefault()) then the user defined item replaces the metadata defined item. The net result is that the user defined item will be at the same position that the metadata item would have been. One word of warning though: it is dangerous to assume a lot about how the metadata for an object looks. So manipulating the result of the metadata a lot inside a form is madness: any time the metadata changes the form becomes unstable. If you need to make large changes to how a form would look when it only uses metadata consider configuring the thing completely - that makes you independent of metadata. <h2 id="TheSearchPanelcomponent(replacementforLookupForm)-CustomizingtheSearchPanel">Customizing the SearchPanel</h2><h3 id="TheSearchPanelcomponent(replacementforLookupForm)-Usingdefaultvalues">Using default values</h3> To have default values for controls we use the builder. To define a default value we need to know the data type of the control that is being used for searching. This data type is often not the same as the data type of the field we search on! This can be seen in the examples above: The lookup control for the "total amount" field behaves as a String, because you can input things like "> 1200". The actual value type for this control is NumberLookupValue, which contains:
Number from: the "from" amount or the first number entered in the string (always present)
QOperation fromOperation: an enum representing the possible operations to issue on that "from", like ">=" or "<".
We also have Number to and QOperation to which are used when there are two conditions in the field, like ">= 1000 < 10000"
The invoiceDate property uses a DateLookupControl which consists of two dates. Consequently it returns a datatype "DatePeriod" which consists of two Date fields (from and to) representing the values in both DateInput controls. To set a default value you must use the data types that are actually used by the control or things will fail. An example of using default values is this: <pre class="syntaxhighlighter-pre" data-syntaxhighlighter-params="brush: java; gutter: false; theme: Confluence" data-theme="Confluence">@Override public void createContent() throws Exception { ContentPanel cp = new ContentPanel(); add(cp); SearchPanel<Invoice> lf = new SearchPanel<>(Invoice.class); cp.add(lf); lf.setClicked(a -> search(lf.getCriteria())); //-- Find customer by ID Customer defaultCustomer = getSharedContext().get(Customer.class, Long.valueOf(10)); lf.add().property("customer").defaultValue(defaultCustomer).control(); // Default customer //-- Default the search total to >= 5.0 NumberLookupValue nlv = new NumberLookupValue(QOperation.GE, BigDecimal.valueOf(5.0)); lf.add().property("total").defaultValue(nlv).control(); // Allow searching for a total //-- Default the date to before 2010. DatePeriod period = new DatePeriod(null, DateUtil.dateFor(2010, 0, 1)); lf.add().property("invoiceDate").defaultValue(period).control(); // And the date. }</pre> We use the appropriate data type for the controls being used and define the value for the control from there. The net result is that the form shows with default search values loaded: <iframe src="https://etc.to/demo/to.etc.domuidemo.pages.

searchpanel.SearchPanelManual2.ui" width="1024" height="400" /> <h2 id="TheSearchPanelcomponent(replacementforLookupForm)-Intermezzo:SearchPanelunderthehood">Intermezzo: SearchPanel under the hood</h2> To have a SearchPanel we need the following parts: Each property we want to search on must have a Lookup Control which lets you enter the value to search for. Once a Lookup Value has been entered in a Lookup Control we need a LookupQueryBuilder to add the LookupValue to a QCriteria. And when all of the data is known we need a ISearchFormBuilder to add the control and its label to a form in some layout. The earlier LookupForm (now deprecated) combined all of this into a single cruddy interface. This made it very hard to customize. The SearchPanel separates all these concerns making it easier to change. <h3 id="TheSearchPanelcomponent(replacementforLookupForm)-LookupControls">Lookup Controls</h3> The search starts with a Lookup control. A Lookup Control is some class implementing IControl<> which can be used to enter data for a lookup. Some data types (as we've seen above) require a special control to be made to handle the search, as their search is somehow special. But many other data types can be looked up with normal controls: Any String can be searched on by using a normal Text2<String> control. Enumerable values like boolean and enum can be looked up by using a combo box (ComboLookup2) which just contains the values from the enumeration Parent relations can be looked up by a LookupInput as long as that LookupInput is configured to look for the specified parent entity. If you do not like the default controls used by the SearchPanel it is relatively easy to create your own: just create a new class implementing IControl<> and make it do what you want. This often also requires that you define the proper datatype for the control, which must be able to hold all of the information that you use to search for. Examples of Lookup controls can be found by looking at DomUI's own implementations under the lookupcontrols package inside the searchpanel package. Because LookupControls are just normal IControl<> instances it is very easy to manipulate them: you can add change listeners or play with their values just as you would do with "normal" controls. Making one search control "react" to changes of another is done by just adding a change listener! <h3 id="TheSearchPanelcomponent(replacementforLookupForm)-LookupQueryBuilders">Lookup Query Builders</h3> Once you have a value from a Lookup Control we need to somehow translate that value into a part of a QCriteria query. This is the responsibility of the ILookupQueryBuilder instances which are defined as follows: <pre class="syntaxhighlighter-pre" data-syntaxhighlighter-params="brush: java; gutter: false; theme: Confluence" data-theme="Confluence">public interface ILookupQueryBuilder<D> { @Nonnull <T> LookupQueryBuilderResult appendCriteria(@Nonnull QCriteria<T> criteria, @Nullable D lookupValue); }</pre> The type D is defined as the type of the value of the associated LookupControl. Implementations of this interface must convert the value entered by the user and represented by the lookupValue into something edible inside the QCriteria instance passed to the method. The simplest QueryBuilder implementation is ObjectLookupQueryBuilder which is defined as follows: <pre class="syntaxhighlighter-pre" data-syntaxhighlighter-params="brush: java; gutter: false; theme: Confluence" data-theme="Confluence">@DefaultNonNull final public class ObjectLookupQueryBuilder<D> implements ILookupQueryBuilder<D> { private final String m_propertyName; public ObjectLookupQueryBuilder(String propertyName) { m_propertyName = requireNonNull(propertyName); } @Override public <T> LookupQueryBuilderResult appendCriteria(QCriteria<T> criteria, @Nullable D value) { if(value == null || (value instanceof String && ((String) value).trim().length() == 0)) return LookupQueryBuilderResult.EMPTY; // Is okay but has no data // FIXME Handle minimal-size restrictions on input (search field metadata) //-- Put the value into the criteria.. if(value instanceof String) { String str = (String) value; str = str.trim().replace("*", "%") + "%"; // FIXME Do not search with wildcard by default 8-(criteria.ilike(m_propertyName, str); } else { criteria.eq(m_propertyName, value); // property == value } return LookupQueryBuilderResult.VALID; } }</pre>
 The thing works as follows: Instances are created with the property name that the search takes place on. Once it's time to search the code finds out if the value is a String. In that case it will default to an "ILIKE" operation inside the QCriteria. If it is not a String the value is treated as a literal that needs to be equal to the database field. This latter is used by all literal matches like: Searching for any enumerated value (boolean, enum) Searching for a parent relation (like the specific Customer) Because the value type can be anything the Query Builder can built very complex queries to do the actual searching. <h3 id="TheSearchPanelcomponent(replacementforLookupForm)-Usingyourownlookupcomponents">Using your own lookup components</h3> It is quite simple to replace the default components by your own. Depending on what you want you use /create a lookup component and you use/create a QueryBuilder. As an example we can replace the customer lookup with a Combobox (bad idea) easily as follows: <pre class="syntaxhighlighter-pre" data-syntaxhighlighter-params="brush: java; gutter: false; theme: Confluence" data-theme="Confluence">@Override public void createContent() throws Exception { ContentPanel cp = new ContentPanel(); add(cp); SearchPanel<Invoice> lf = new SearchPanel<>(Invoice.class); cp.add(lf); lf.setClicked(a -> search(lf.getCriteria())); //-- Create a combobox of customers QCriteria<Customer> q = QCriteria.create(Customer.class) // All customers with last names starting with A .ilike("lastName", "B%"); ComboLookup2<Customer> customerC = new ComboLookup2<>(q); customerC.setContentRenderer((node, value) -> node.add(value.getFirstName() + " " + value.getLastName())); lf.add().property("customer").control(customerC); lf.add().property("total").control(); // Allow searching for a total lf.add().property("invoiceDate").control(); // And the date. }</pre> In this case, because the control returns a value (Customer) that needs to be compared with equals we only pass a new control; the SearchPanel will use the ObjectLookupQueryBuilder to create the query. The net results looks like this: <iframe src="https://etc.to/demo/to.etc.domuidemo.pages.searchpanel.SearchPanelManual3.ui" width="1024" height="400" /> When you make a control that has a more complex value you need to create the LookupBuilder too. <h3 id="TheSearchPanelcomponent(replacementforLookupForm)-ComparisonwithLookupForm'scomponents">Comparison with LookupForm's components</h3> LookupForm combined everything about a single search property in a single interface (ILookupControlInstance). This interface was responsible for everything: the control to use, the search to perform, how to render the control and its label and the shoe size of the builder. To customize this was hard because everything needed to be constructed as one class. The controls used by this code were not real IControl instances so manipulating them was very special. In addition because both controls and search code needed to be together there was no clear separation of tasks which again reduced reusability - and caused some quite bad code in the process. The form that was constructed by the LookupForm was fixed as there was no reasonable way to change the layout without adding more crud to the LookupForm. The SearchPanels separates all of this into separate well-defined and reusable parts, and delegates "special use" into special parts - that themselves then become reusable again. <h3 id="TheSearchPanelcomponent(replacementforLookupForm)-LookupFactories">Lookup Factories</h3> When just defining properties to search for the SearchPanel will try to create the proper lookup controls and query builder by itself. It does that by asking the LookupControlRegistry2 class for control factories that can handle the specified property. A factory instance is defined as follows: <pre class="syntaxhighlighter-pre" data-syntaxhighlighter-params="brush: java; gutter: false; theme: Confluence" data-theme="Confluence">public interface ILookupFactory<D> { @Nonnull FactoryPair<D> createControl(@Nonnull SearchPropertyMetaModel spm); }</pre> and gets registered like this: <pre class="syntaxhighlighter-pre" data-syntaxhighlighter-params="brush: java; gutter: false; theme: Confluence" data-theme="Confluence">register(new DateLookupFactory2(), a -> Date.class.isAssignableFrom(a.getActualType()) ? 10 : 0); register(new EnumAndBoolLookupFactory2<>(), LookupControlRegistry2::scoreEnumerable); register(new NumberLookupFactory2(), pmm -> DomUtil.isIntegerType(pmm.getActualType()) || DomUtil.isRealType(pmm.getActualType()) || pmm.getActualType() == BigDecimal.class ? 10 : 0); register(new RelationLookupFactory2<>(), pmm -> pmm.getRelationType() == PropertyRelationType.UP ? 10 : 0); register(new RelationComboLookupFactory2<>(), pmm -> pmm.getRelationType() == PropertyRelationType.UP && Constants.COMPONENT_COMBO.equals(pmm.getComponentTypeHint()) ? 10 : 0); register(new StringLookupFactory2<>(), pmm -> 1); // Accept all</pre> The factory is combined with a lambda that checks the characteristics of the property against whatever the factory would accept. This comparison returns a score. The factory returning the highest > 0 score is the one that is asked to create the control and the query factory. This makes it very easy to create your own defaults for SearchPanel controls: just create a factory, register it and make it return the correct score when you recognise a property you'd want to handle. <h2 id="TheSearchPanelcomponent

(replacementforLookupForm)-Example:creatingyourowncomponentandbuilder">Example: creating your own component and builder</h2> In this example we are going to make a screen to search inside the Tracks table. We will use the EnumSetInput control to select zero to one Genre's from the database, then limit the search to those genres selected. The completed thing looks like this: <iframe src="https://etc.to/demo/to.etc.domuidemo.pages.searchpanel.SearchPanelCustomControl4.ui" width="1024" height="400" /> We will start with the page's code: <pre class="syntaxhighlighter-pre" data-syntaxhighlighter-params="brush: java; gutter: false; theme: Confluence" data-theme="Confluence">@Override public void createContent() throws Exception { ContentPanel cp = new ContentPanel(); add(cp); SearchPanel<Track> lf = new SearchPanel<>(Track.class); cp.add(lf); lf.setClicked(a -> search(lf.getCriteria())); //-- For Genre we will use a new control EnumSetInput<Genre> genreC = new EnumSetInput<>(Genre.class); List<Genre> genreList = getSharedContext().query(QCriteria.create(Genre.class)); genreC.setData(genreList); Set<Genre> def = new HashSet<>(); def.add(genreList.get(0)); def.add(genreList.get(1)); lf.add().property("genre").defaultValue(def).control(new EnumSetQueryBuilder<>("genre"), genreC); lf.add().property("name").control(); lf.add().property("album").control(); // Allow searching for a total }</pre> We add the usual panel, then we prepare the EnumSetInput: We read all Genre records from the database in genreList We prepare a default (just for show) of the 1st two genres returned Then we add the control for the Genre to the SearchPanel. The EnumSetInput returns a Set<Genre> for all of the selected items. This set of course is not understood by any of the existing query builders, so we made one ourself: the EnumSetQueryBuilder: <pre class="syntaxhighlighter-pre" data-syntaxhighlighter-params="brush: java; gutter: false; theme: Confluence" data-theme="Confluence">public class EnumSetQueryBuilder<V> implements ILookupQueryBuilder<Set<V>> { private final String m_propertyName; public EnumSetQueryBuilder(String propertyName) { m_propertyName = propertyName; } @Nonnull @Override public <T> LookupQueryBuilderResult appendCriteria(@Nonnull QCriteria<T> criteria, @Nullable Set<V> lookupValue) { if(lookupValue == null || lookupValue.isEmpty()) return LookupQueryBuilderResult.EMPTY; QRestrictor<T> or = criteria.or(); lookupValue.forEach(value -> or.eq(m_propertyName, value)); return LookupQueryBuilderResult.VALID; } }</pre> An instance of this class gets connected to the search item. It works as follows: The class gets instantiated with the property name to search for, which in this case will be "genre" inside the Track entity. When it is time to create the query the appendCriteria call gets called. It: Checks whether data was actually there. If not it returns the EMPTY indicator. This indicator is used when searching is only allowed with at least one clause filled in. For complex data you would also check the input here and throw a ValidationException if the data was wrong. We can now create the query: we add an or of all values in the set. <h2 id="TheSearchPanelcomponent(replacementforLookupForm)-Generatingtheform">Generating the form</h2> The search panel generates a form containing the labels and controls that are added. By default it generates a simple vertical form where each label/control pair are on a single line. How can we influence the form's layout? For that we need to know how the form gets generated. The SearchPanel uses an ISearchFormBuilder implementation as the thing that actually builds the form: <pre class="syntaxhighlighter-pre" data-syntaxhighlighter-params="brush: java; gutter: false; theme: Confluence" data-theme="Confluence">public interface ISearchFormBuilder { /** Defines the target node for the form to be built. */ void setTarget(NodeContainer target) throws Exception; void append(SearchControlLine<?> it) throws Exception; void finish() throws Exception; }</pre> All actions that create the form based UI are delegated to this interface. If you do nothing then SearchPanel uses the default implementation of this thing: DefaultSearchFormBuilder. This implementation is very basic: <pre class="syntaxhighlighter-pre" data-syntaxhighlighter-params="brush: java; gutter: false; theme: Confluence" data-theme="Confluence">public class DefaultSearchFormBuilder implements ISearchFormBuilder { @Nullable private FormBuilder m_builder; private Div m_target; @Override public void setTarget(NodeContainer target) throws Exception { Div root = m_target = new Div("ui-dfsb-panel"); target.add(root); Div d = new Div("ui-dfsb-part"); root.add(d); m_builder = new FormBuilder(d); } @Override public void append(SearchControlLine<?> it) throws Exception { NodeContainer label = it.getLabel(); if(null != label) fb().label(label); IControl<?> control = it.getControl(); fb().control(control); } public void addBreak() { NodeContainer target = requireNonNull(m_target); Div d = new Div("ui-dfsb-part"); target.add(d); m_builder = new FormBuilder(d); } @Override public void finish() throws Exception { m_builder = null; } @Nonnull public FormBuilder fb() { return requireNonNull(m_builder); } }</pre> It just uses a normal FormBuilder to create the form, and it has an extra "method" to allow the form to be split into multiple columns: the "addBreak" method. Controlling how the form looks can be done by creating your own implementation of ISearchFormBuilder. You can even make it the default layout by calling: <pre class="syntaxhighlighter-pre" data-syntaxhighlighter-params="brush: java; gutter: false; theme: Confluence" data-theme="Confluence">SearchPanel.setDefaultSearchFormBuilder(Supplier<ISearchFormBuilder> factory)</pre> so that the supplier returns your factory. To interact with your factory you can add special "actions" to the SearchForm's definition. For example, to use that "addBreak()" method we would code the following: <pre class="syntaxhighlighter-pre" data-syntaxhighlighter-params="brush: java; gutter: false; theme: Confluence" data-theme="Confluence">SearchPanel sp = new SearchPanel(Invoice.class); DefaultSearchFormBuilder bld = new DefaultSearchFormBuilder(); sp.setFormBuilder(bld); // Ensure that the DefaultFormBuilder is used //-- add first two props lf.add().property("type").control(typeC, new EnumSetQueryBuilder<>(Definition.pTYPE)); lf.add().property("type").control(); lf.add().action(() -> bld.addBreak()); lf.add().property("stage").control();</pre> The action will be executed in the order of adding the controls. You could of course also cast lf.getFormBuilder() to DefaultSearchFormBuilder instead of creating an instance yourself. This however will cause your code to break if the default ever changes.

   </body> </html>