HTML Form Entry Module Technical Overview

HTML Form Entry Process Flow

Form Display

The first step in displaying an HTML Form is the creation of a **FormEntrySession** instance, which holds, among other things:

- **HtmlForm** instance that holds the XML that defines the form associated with the session.
- **HtmlFormEntryGenerator** instance that provides methods to translate the HTML Form XML into HTML that can be displayed by a web browser.
- **FormEntryContext** instance that holds the context data around generating the HTML widgets.
- **FormSubmissionController** instance, initially empty, that holds the actions that need to be taken when a form is submitted.
- **FormSubmissionActions** instance, initially empty, that holds the details of what data objects need to be created or modified to commit the form to the database.

When the **FormEntrySession** is instantiated it first populates the **FormEntryContext** with any existing Patient or Encounter data associated with the form .

Next, the **FormEntrySession** generates the HTML to display using the **HtmlFormEntryGenerator**. It first calls the **applyMacros()**, **applyTemplates()**, and **applyTranslations()** methods of **HtmlFormEntryGenerator** to perform the substitutions required for any macros, templates (repeats), or translations defined in the form.

Then it calls the **applyTags(FormEntryContext,String)** method of **HtmlFormEntryGenerator** to do the bulk of the form generation work – converting all other HTMLFormEntry-specific tags to Html – and to populate the **FormSubmissionController** with all the actions needs to process the form. **applyTags** iterates through all the nodes in the **HtmlForm** xml; for all HtmlFormEntry-specific tags, it retrieves the appropriate **TagHandler** for the tag (tag to tag handler mappings are defined in moduleApplicationContext.xml) and uses that tag handler to process the tag.

A **TagHandler** serves two primary purposes:

- It creates an instance of the appropriate HtmlGeneratorElement implementation and calls it's **generateHtml(Context)** method to generate the Html input fields to substitute for the tag. When a **HtmlGeneratorElement** is instantiated, it creates any widgets needed to render the element, and registers the widgets with the **FormEntryContext**.
- It creates an instance of the appropriate FormSubmissionControllerAction implementation, and adds it to the **FormSubmissionController**. The **FormSubmissonController** will use this to handle the specifics of form submissions for this element.

Note that the **HtmlGeneratorElement** and the **FormSubmissionControllerAction** may be (and, in fact, are likely to be) an instantiation of a single object that supports both interfaces. (For example **EncounterDetailSubmissionElement**, **EnrollInProgramElement**, **ObsSubmissionElement**, **PatientDetailSubmissionElement**, and **PatientElement** all serve as the **HtmlGeneratorElement** and the **FormSubmissionControllerAction** for their respective elements.)

All the above steps happen during the instantiation of the **FormEntrySession**, so, to display form, a web controller can simply instantiate a **FormEntrySession** – passing as parameters the patient, encounter, mode (read, enter, or edit), and the **HtmlForm** to use – and then use the **FormEntrySession** as a backing object. The field **htmlToDisplay** of the **FormEntrySession** will contain the html to display in the view.

Form Submission

When a form is submitted, the web controller handling the submission retrieves the **FormSubmissionController** from the session and calls its **handleFormValidation(FormEntryContext, HttpServletRequest)** method to validate the form. The **FormSubmissionController**, in turn, iterates through all the **FormSubmissionControllerActions** associated with it and calls their **validateSubmission(FormEntryContext, HttpServletRequest)** methods to validate the individual elements in the form.

If validation in successful, the web controller then calls the **FormSubmissionController handleFormSubmission(FormEntrySession, HttpServletRequest)** method. Again, the **FormSubmissionController** iterates through all the **FormSubmissionControllerActions**, this time calling their **handleSubmission(FormEntrySession, HttpServletRequest)** methods to handle the submission of the individual elements in the form.

Significantly, the **handleFormSubmission** methods don't directly commit the changes to the database. Instead each **FormSubmissionControllerAction** adds the needed actions to commit the changes to the **FormSubmissionActions** instance associated with the session. (Yes, this is a little confusing ... there are two types of actions... a **FormSubmissionControllerAction** creates a new action, or set of actions, that are stored in **FormSubmissionActions**).

After the **FormSubmissionController** finishes executing all the **handleFormSubmission** methods without error, the web controller can call the **FormEntrySession applyActions()** method, which applies all the actions stored in the **FormSubmissionActions** instance and actually commits the changes to the database, creating or modifying any data objects as necessary.

That's a basic, but not exhaustive, overview of how the module work. We didn't discuss how obsgroups are implemented, or about the various classes in the schema package, among other things. Check out the javadocs for more information.

Other Features

Custom Tags

Custom tags can be registered by calling the **HtmlFormEntryService addHandler** method(). Modules can register new tags by calling this method... for an example, in the **HTML Form Flowsheet module** the **onLoad()** method in the **HtmlFormFlowsheetActivator** registers a handler for a new "htmlformflowsheet" tag. To create a custom tag, you will need to define implementations of **TagHandler**, **HtmlGeneratorElement**, and **FormSubmission ControllerAction** for this new tag. You can take a look at some of the existing tags for examples of how to do this.

Defining attribute descriptors for custom tags

Starting in HtmlFormEntry version 1.7.2, all tag handlers are required to implement a **getAttributeDescriptors** method. Attribute descriptors define what attributes a tag supports, to facilitate sharing tags via the metadata sharing module. When preparing a form for export, all metadata referenced by the form need to be included in the export package. For example, in the following tag, all the concepts referenced by id, uuid, or concept mapping need to be loaded and included:

```
<obs conceptId="1000" answerConceptIds="1001,XYZ:HT,1002,32296060-03-102d-b0e3-001ec94a0cc7"/>
```

Attribute descriptors provide a means for a tag handler to define how to find all the metadata that may be referenced within the tag. For example, the **Drug OrderTagHandler** defines the following descriptors:

```
protected List<AttributeDescriptor> createAttributeDescriptors() {

         List<AttributeDescriptor> attributeDescriptors = new ArrayList<AttributeDescriptor>();

         attributeDescriptors.add(new AttributeDescriptor("drugNames", Drug.class));
        attributeDescriptors.add(new AttributeDescriptor("discontinuedReasonConceptId", Concept.class));
         attributeDescriptors.add(new AttributeDescriptor("discontinueReasonConceptAnswers", Concept.class));

         return Collections.unmodifiableList(attributeDescriptors);
 }
```

The easiest way to add attribute descriptor support to a tag handler is for the tag handler to extend **AbstractTagHandler**, which provides a basic implementation of **getAttributeDescriptor** that returns null. To define attribute attribute descriptors, override the **AbstractTagHandler. createAttributeDescriptors** method as shown above for **DrugOrderTagHandler**. (Note that **SubstitutionTagHandler** now extends **AbstractTagHandler** , so any tag handler that currently extends **SubstutitionTagHandler** will have default attribute descriptor support built-in.)

Currently, the module supports exporting any openmrs metadata referenced by id, uuid, or name. It also supports exporting concepts referenced by concept mapping.

Note that currently, before exporting a form, all references to persons, roles, and patient identifier types are removed from the form, though this may become configurable in the future.

Note that to work with version 1.7.2, any existing custom tag handlers will need to be modified to implement **getAttributeDescriptors** (though if the custom tag handler already implements **SubstitutionTagHandler**, no change will be required).

In the future, attribute descriptors may be used for other purposes besides metadata sharing. For instance, the Html Form Entry Designer module could use the descriptors to determine the valid attributes for a tag.

Custom Content for Velocity Context

If your module instantiates a Spring bean that implements org.openmrs.module.htmlformentry.velocity.VelocityContextContentProvider then your bean's populateContext() method will be called each time a new HTML Form is opened (for entry, view, or edit). (Since HTML Form Entry 1.11)

See an example from the Kenya EMR module on github.