Using Select With a List

Using Select With a List

The documentation for the Select Component and the Tapestry Tutorial provide simplistic examples of populating a drop-down menu (as the (X)HTML *Select* element) using comma-delimited strings and enums. However, most real-world Tapestry applications need to populate such menus using values from a database, commonly in the form of java.util.List objects. Doing so generally requires a SelectModel and a ValueEncoder bound to the Select component with its "model" and "encoder" parameters:

```
<t:select t:id="colorMenu" value="selectedColor" model="ColorSelectModel" encoder="colorEncoder" />
```

In the above example, ColorSelectModel must be of type SelectModel, or anything that Tapestry knows how to coerce into a SelectModel, such as a List or a Map or a "value=label,value=label,..." delimited string, or anything Tapestry knows how to coerce into a List or Map, such as an Array or a comma-delimited String.

SelectModel

A SelectModel is a collection of options (specifically OptionModel objects) for a drop-down menu. Basically, each option is a value (an object) and a label (presented to the user).

JumpStart Demos:
Total Control Object Select
ID Select
Easy ID Select

If you provide a property of type List for the "model" parameter, Tapestry automatically builds a SelectModel that uses each object's toString() for both the select option value and the select option label. For database-derrived lists this is rarely useful, however, since after form submission you would then have to look up the selected object using that label.

If you provide a Map, Tapestry builds a SelectModel that uses each item's key as the encoded value and its value as the user-visible label. This is more useful, but if you are going to build a copy of the list as a map just for this purpose, you may as well let Tapestry do it for you, using SelectModelFactory.

SelectModelFactory

To have Tapestry create a SelectModel for you, use the SelectModelFactory service. SelectModelFactory creates a SelectModel from a List of objects (of whatever type) and a label property name that you choose:

SelectWithListDemo.java (a page class)

```
@Property
private SelectModel colorSelectModel;
@Inject
SelectModelFactory selectModelFactory;
...
void setupRender() {
    // invoke my service to find all colors, e.g. in the database
    List<Color> colors = colorService.findAll();

    // create a SelectModel from my list of colors
    colorSelectModel = selectModelFactory.create(colors, "name");
}
```

The resulting SelectModel has a selectable option (specifically, an OptionModel) for every object in the original List. The label property name (the "name" property, in this example) determines the user-visible text of each menu option, and your ValueEncoder's toClient() method provides the encoded value (most commonly a simple number). If you don't provide a ValueEncoder, the result of the objects' toString() method (Color#toString() in this example) is used. Although not a recommended practice, you *could* set your toString() to return the object's ID for this purpose:

Color.java (partial)

```
...
@Override
public String toString() {
    return String.valueOf(this.getId());
}
```

But that is contorting the purpose of the toString() method, and if you go to that much trouble you're already half way to the recommended practice: creating a ValueEncoder.

ValueEncoder

In addition to a SelectModel, your Select menu is likely to need a ValueEncoder. While a SelectModel is concerned only with how to construct a Select menu, a ValueEncoder is used when constructing the Select menu *and* when interpreting the encoded value that is submitted back to the server. A ValueEncoder is a converter between the type of objects you want to represent as options in the menu and the client-side encoded values that uniquely identify them, and vice-versa.

Most commonly, your ValueEncoder's toClient() method will return a unique ID (e.g. a database primary key, or perhaps a UUID) of the given object, and its toValue() method will return the *object* matching the given ID by doing a database lookup (ideally using a service or DAO method).

JumpStart Demo:
Easy Object Select

If you're using one of the ORM integration modules (Tapestry-Hibernate, Tapestry-JPA, or Tapestry-Cayenne), the ValueEncoder is automatically provided for each of your mapped entity classes. The Hibernate module's implementation is typical: the primary key field of the object (converted to a String) is used as the client-side value, and that same primary key is used to look up the selected object.

That's exactly what you should do in your own ValueEncoders too:

ColorEncoder.java (perhaps in your com.example.myappname.encoders package)

```
public class ColorEncoder implements ValueEncoder<Color>, ValueEncoderFactory<Color> {

    @Inject
    private ColorService colorService;

    @Override
    public String toClient(Color value) {
        // return the given object's ID
        return String.valueOf(value.getId());
    }

    @Override
    public Color toValue(String id) {
        // find the color object of the given ID in the database
        return colorService.findById(Long.parseLong(id));
    }

    // let this ValueEncoder also serve as a ValueEncoderFactory
    @Override
    public ValueEncoder<Color> create(Class<Color> type) {
        return this;
    }
}
```

Alternatively, if you don't expect to need a particular ValueEncoder more than once in your app, you might want to just create it on demand, using an anonymous inner class, from the getter method in the component class where it is needed. For example:

SelectWithListDemo.java (a page class, partial)

```
    . . .

    public ValueEncoder<Color> getColorEncoder() {

        return new ValueEncoder<Color>() {

            @Override
            public String toClient(Color value) {
                // return the given object's ID
                return String.valueOf(value.getId());
            }

            @Override
            public Color toValue(String id) {
                // find the color object of the given ID in the database
                return colorService.findById(Long.parseLong(id));
            }
        };
    }
```

Notice that the body of this anonymous inner class is the same as the body of the ColorEncoder top level class, except that we don't need a `create` method.

Applying your ValueEncoder Automatically

If your ValueEncoder *implements ValueEncoderFactory* (as the ColorEncoder top level class does, above), you can associate your custom ValueEncoder with your entity class so that Tapestry will automatically use it every time a ValueEncoder is needed for items of that type (such as with the Select, RadioGroup, Grid, Hidden and AjaxFormLoop components). Just add lines like the following to your module class (usually AppModule.java):

AppModule.java (partial)

```
...
    public static void contributeValueEncoderSource(MappedConfiguration<Class<Color>,
                        ValueEncoderFactory<Color>> configuration) {
        configuration.addInstance(Color.class, ColorEncoder.class);
    }
```

If you are contributing more than one ValueEncoder, you'll have to use raw types, like this:

AppModule.java (partial)

```
...
    public static void contributeValueEncoderSource(MappedConfiguration<Class,
                        ValueEncoderFactory> configuration)
    {
        configuration.addInstance(Color.class, ColorEncoder.class);
        configuration.addInstance(SomeOtherType.class, SomeOtherTypeEncoder.class);
    }
```

What if I omit the ValueEncoder?

The Select component's "encoder" parameter is optional, but if the "value" parameter is bound to a complex object (not a simple String, Integer, etc.) and you don't provide a ValueEncoder with the "encoder" parameter (and one isn't provided automatically by, for example, the Tapestry Hibernate integration), you'll receive a "Could not find a coercion" exception (when you submit the form) as Tapestry tries to convert the selected option's encoded value back to the *object* in your Select's "value" parameter. To fix this, you'll either have to 1) provide a ValueEncoder, 2) provide a Coercion, or 3) use a simple value (String, Integer, etc.) for your Select's "value" parameter, and then you'll have to add logic in the corresponding onSuccess event listener method:

SelectWithListDemo.tml (partial)

```
<t:select t:id="colorMenu" value="selectedColorId" model="ColorSelectModel" />
```

SelectWithListDemo.java (partial)

```
...
    public void onSuccessFromMyForm() {
        // look up the color object from the ID selected
            selectedColor = colorService.findById(selectedColorId);
            ...
    }
```

But then again, you may as well create a ValueEncoder instead.

Why is this so hard?

Actually, it's really pretty easy if you follow the examples above. But why is Tapestry designed to use SelectModels and ValueEncoders anyway? Well, in short, this design allows you to avoid storing (via @Persist, @SessionAttribute or @SessionState) the entire (potentially large) list of objects in the session or rebuilding the whole list of objects again (though only one is needed) when the form is submitted. The chief benefits are reduced memory use and more scalable clustering due to having far less HTTP session data to replicate across the nodes of a cluster.