How to build a bean for Servoy
– A step by step tutorial brought to you by Servoy Stuff

PART 7

GG. Room for improvements

In the last part of this tutorial, we created a simple Wicket input component based on a Panel and a TextField to emulate very basically our Swing Slider. This time we will integrate some JavaScript libraries and CSS/images resources to fully emulate a Slider in the client's browser.

But first, there are a few bug fixes/enhancements that I have found after last time that we will code into our current classes (I already said that there was always room for improvements, so each time you go back to your code, as I did here, you can find something to do better, and as you test your beans in different situations you can come across some annoying problem that needs to be fixed).

First in the ServoySlider class, I have made the following changes:

```
	private boolean snapToTicks = false;
```

Changed default value to false because our slider is more used as an input bean (with a wide range of values) than a navigation bean.

In the initComponent() method, added a few calls that were missing:

```
		setPaintLabels(paintLabels);
		setSnapToTicks(snapToTicks);
		setInverted(inverted);
		setOrientation(orientation);
```

And finally:
```
	public int isMajorTicks() {
```
Was wrongly named (it is not a boolean), so I changed it to:
```
	public int getMajorTicks() {
```

Second, in the ServoySwingSlider, I have made the following changes:

```
	private IClientPluginAccess app;
```

I have added a reference to the client (remember that it is initialized in the initComponent() method of the ServoySlider class with a call to component.initialize(app);), so that we will be able to set the status text when changing the value of the slider (see stateChanged() method enhancements below)

Then in the initialize() method of course, I set the variable:

```
	public void initialize(IClientPluginAccess app) {
		this.app = app;
	}
```

In the updateSliderMethod() I changed the call to the superclass to calls to the object's implementation:

```
      // the behaviour we had before:
      if (validationEnabled) {
            setMinimumValue(1);
            int max = currentFoundset.getSize();
            setMaximumValue(max);
            setMajorTicks(max-1);
            int tickSpacing = Math.max(Math.round(max/10),1);
            setMinorTicks(tickSpacing);
            setSliderValue(currentFoundset.getSelectedIndex()+1);
            updateLabels(); // added to update when used as navigation bean
      } else {
            setMinimumValue(0);
            setSliderValue(0);
      }
```

The biggest change is in the stateChanged() method. The fact is that in the previous implementation each time you moved the slider the values were updated (or the selectedRecord if you used it as a navigation bean), which was expensive in terms of call-back to the server and not really necessary. The normal behaviour should really be that while you move the slider the underlying model isn't really changed until you release it. To allow for this, there is a boolean in the Slider that can be tested to see if the user is fiddling with the slider's handle and when he has finished, that's the valueIsAdjusting property. You use it to test if it is worth it updating your model or if it's just a temporary value.
So our stateChanged() method has been added a few conditions and behaviour:

```
public void stateChanged(ChangeEvent e) {
      if (getValueIsAdjusting()) {
            if (app != null) {
                  if (ServoyUtils.isContainedInFoundset(
                        currentRecord, getDataProviderID())) {
                        if (isUsingFactor()) {
                              app.setStatusText(""+getNumberValue());
                        } else {
                              app.setStatusText(""+getValue());
                        }
                  } else {
                        app.setStatusText(""+(getValue()-1));
                  }
            }
      } else {
                  //… same code as before …

                  if (app != null) {
                        app.setStatusText("Done");
                  }
            }
      }
      ignoreUpdate = false;
}
```

So we first test our "valueIsAdjusting" property:
If it's true, we update the status bar (the one shown in the lower left corner of Servoy) with the temporary value coming from the correct method depending on the mode of our bean (as input field or navigator) and the factor if used. We update the status text by using the setStatutText() method of the IClientPluginAccess interface, that's why we needed a reference to it in the variable "app"...

By setting the status, we give feedback to the user to help him adjusts the value.

If the “valueIsAdjusting” is false (meaning the user has finished interacting with the slider and the value can be considered final), then we do the same thing as before, and we simple add a clue in the status bar indicating that the value has been taking into account.

The fact is that by coding this behaviour we will be closer to our web client implementation because it wouldn’t make sense to make Ajax calls from the browser to the server for each temporary value of the slider while the user is moving it, it could trigger hundreds of Ajax calls in a few seconds, would be very expensive in terms of resources and bandwidth and could make your application unresponsive in the end…

So by making the Swing bean behaving the same way we will reinforce the illusion that our slider is one and only component in the Smart client and the Web client.

Finally I have changed the updateLabels() method to take into account cases where the maximum value is not in the loop to add a label for it.

So this part:
```
for (int labelIndex = getMinimum();
      labelIndex <= getMaximum();
      labelIndex += getMajorTickSpacing() ) {
      int val = (isUsingFactor())
            ? (int)Math.round(labelIndex/getPrecisionFactor())
            : labelIndex;
      table.put( new Integer( labelIndex ), new JLabel( ""+val, JLabel.CENTER ) );
}
```

Has been changed to:
```
int labelIndex = getMinimum();
for ( ; labelIndex <= getMaximum(); labelIndex += getMajorTickSpacing() ) {
      int val = (isUsingFactor())
            ? (int)Math.round(labelIndex/getPrecisionFactor())
            : labelIndex;
      table.put( new Integer( labelIndex ), new JLabel( ""+val, JLabel.CENTER ) );
}
if (labelIndex != getMaximum()) {
      int val = (isUsingFactor())
            ? (int)Math.round(getMaximum()/getPrecisionFactor())
            : getMaximum();
      table.put( new Integer( getMaximum() ), new JLabel( ""+val, JLabel.CENTER ) )

}
```

HH. Slider JavaScript library

Now it's time to look at a Slider JavaScript library that could emulate our Swing Slider.

I found a few on the internet. All of them were lacking some important features, so I looked for one that was robust enough and the closest to what I needed to emulate the Swing component. In the end, I opted for a Slider from a library called WebFX that, although a little bit dated (last update was May 2006), offered the widest range of features already, had a clean Object Oriented code and was Open Source of course (Apache License v2.0).

You can find more info about this library here:
http://webfx.eae.net/
And you can download the original library here:
http://webfx.eae.net/download/slider102.zip

You will also find my enhanced version with the downloadable files of this tutorial (see at the end of this PDF), launch the contained well-named "test.html" and look at the html source of this file to get a glimpse of what we will be doing in Wicket later.

I contacted the author (Erik Arvidsson) to obtain the permission to use his library (with some hope that he will help me add some functionality, but I was basically told that I could do it myself :{). One good thing was that there was little dependencies to big JavaScript libraries (like YahooUI or MooTools or other Widget libraries), and that the project had already some CSS and images that were mimicking a Swing slider.

Still there was a lot missing, so I had to dig into the JavaScript of the "slider.js" file (which fortunately was very well written IMHO). Since I am not a JavaScript guru, it took some time to do it right, and maybe there are some places where the code I changed could be done better, but you are free to take a look and enhance/optimize it… If you do, please send it back so that I will make it available to all!

What was missing?
No invert support, no ticks support, nor labels support, no paintTrack flag, nor enable/disable, no precision support of course (it worked only with integer value, just like a regular Swing slider – adding it here was to allow for more concision in the ServoyWicketSlider class which was already quite crowded), no real support for Ajax partial updates, no images for disabled and hover for the 3 styles. Anyway, you will find in the header of the updated "slider.js" file a history of all the changes I made…

In the end, after scratching my head more than once, I had a very much usable JavaScript slider tested in Firefox and IE on Windows (I haven't had the occasion to test it on Mac OS X or Linux though, so you will have to tell me if it's OK, and implement your workarounds, if possible).

I have put all the dependencies (/js/ containing the 3 JavaScript files, and /css/ containing the CSS and images of the 3 provided styles) inside a "res" folder (and changed a bit the example files, so that I only had to copy this folder into my net/stuff/servoy/beans/slider/wicket/ folder and asks for a Refresh (F5 or right click on the wicket package in the Package Explorer view) to get all the files I needed for the Wicket bean properly enclosed into a neat "res" package…

II. Modifying the ServoyWicketSlider

Now that we have our JavaScript/CSS/images in our wicket/res package, let's see what we have to do on the ServoyWicketSlider to use them.

First we need to implement all the // TODO tasks and we will do that simply by adding the related variables, which are:

```
	private boolean focusable;
	private int orientation;
	private boolean inverted;
	private int minimumValue;
	private int maximumValue;
	private int majorTicks;
	private int minorTicks;
	private boolean paintTicks;
	private boolean paintLabels;
	private boolean paintTrack;
	private boolean snapToTicks;
	private int precision;
	private boolean usingDecimals;
```

Now all we have to do is to hook these newly added variables to the related getters and setters.

The getters are easy:

```
	public boolean getInverted() {
		return inverted;
	}
	public int getMajorTicks() {
		return majorTicks;
	}
	public int getMaximumValue() {
		return maximumValue;
	}
	public int getMinimumValue() {
		return minimumValue;
	}
	public int getMinorTicks() {
		return minorTicks;
	}
	public int getOrientation() {
		return orientation;
	}
	public boolean getPaintLabels() {
		return paintLabels;
	}
	public boolean getPaintTicks() {
		return paintTicks;
	}
	public boolean getPaintTrack() {
		return paintTrack;
	}
	public int getPrecision() {
		return precision;
	}
	public boolean getSnapToTicks() {
		return snapToTicks;
	}
	public boolean isFocusable() {
		return focusable;
	}
	public boolean isUsingDecimals() {
		return usingDecimals;
	}
```

The only special getter method that's a bit different is:

```
	public int getCurrentValue() {
		Number n = getNumberValue();
		return (n == null) ? 0 : n.intValue();
	}
```

Because we don't really have a currentValue of type int (there is little use for it really).

Then while we are at it we add 2 convenience methods:

```
	private boolean isUsingFactor() {
		return isUsingDecimals() && hasDataProvider();
	}

	private boolean hasDataProvider() {
		return ServoyUtils.isNotEmpty(getDataProviderID());
	}
```

that we will use later...

The setter methods are also straightforward:

```
	public void setFocusable(boolean focusable) {
		this.focusable = focusable;
	}
	public void setInverted(boolean inverted) {
		this.inverted = inverted;
	}
	public void setMajorTicks(int majorTicks) {
		this.majorTicks = majorTicks;
	}
	public void setMaximumValue(int maximumValue) {
		this.maximumValue = maximumValue;
	}
	public void setMinimumValue(int minimumValue) {
		this.minimumValue = minimumValue;
	}
	public void setMinorTicks(int minorTicks) {
		this.minorTicks = minorTicks;
	}
	public void setOrientation(int orientation) {
		this.orientation = orientation;
	}
	public void setPaintLabels(boolean paintLabels) {
		this.paintLabels = paintLabels;
	}
	public void setPaintTicks(boolean paintTicks) {
		this.paintTicks = paintTicks;
	}
	public void setPaintTrack(boolean paintTrack) {
		this.paintTrack = paintTrack;
	}
	public void setPrecision(int precision) {
		this.precision = precision;
	}
	public void setSnapToTicks(boolean snapToTicks) {
		this.snapToTicks = snapToTicks;
	}
	public void setUsingDecimals(boolean usingDecimals) {
		this.usingDecimals = usingDecimals;
	}
```

While the setCurrentValue() will forward to our setNumberValue() calculating the value if needed:

```
	public void setCurrentValue(int currentValue) {
		setNumberValue(currentValue*(Math.pow(10.0, getPrecision())));
	}
```

Now we will enhance our updateSlider() method to reflect the behaviour of the ServoySwingSlider, so this time our method will be changed to:

```
private void updateSlider() {
	if (ServoyUtils.isExisting(currentFoundset, currentRecord)) {
		if (ServoyUtils.isContainedInFoundset(currentRecord,
			getDataProviderID())) {

			Object o = currentRecord.getValue(getDataProviderID());
			this.numberValue = (Double)o;
			this.previousValue = getNumberValue();
			setChanged();
		} else {
			if (validationEnabled) {
				setMinimumValue(1);
				int max = currentFoundset.getSize();
				setMaximumValue(max);
				setMajorTicks(max-1);
				int tickSpacing = Math.max(Math.round(max/10),1);
				setMinorTicks(tickSpacing);
				this.numberValue = currentFoundset.getSelectedIndex()+1;
				this.previousValue = getNumberValue();
			} else {
				setMinimumValue(0);
				this.numberValue = 0;
				this.previousValue = getNumberValue();
			}
			setChanged();
		}
	}
}
```

The method has precisely the same kind of code as the ServoySwingSlider.
Note that we don't call setNumberValue() to avoid triggering a call-back to Servoy by setting the record value, we simply set the value of the local numberValue variable directly.

The setValidationEnabled should be also coded similarly as we did for the Swing component (you remember that Servoy set the validationEnabled property to false when in Find mode, so we need to check if we are using the slider as a navigation bean, because if so we need to disable it):

```
	public void setValidationEnabled(boolean validationEnabled) {
		this.validationEnabled = validationEnabled;
		if (!hasDataProvider()) {
			setEnabled(validationEnabled);
		}
	}
```

And then the setNumberValue() should also be changed to reflect the behaviour of the Swing bean:

```
public void setNumberValue(double numberValue) {
	this.numberValue = numberValue;
	if (previousValue != numberValue) {
		if (ServoyUtils.isExisting(currentFoundset, currentRecord)) {
			if (ServoyUtils.isContainedInFoundset(currentRecord,
				getDataProviderID())) {
				if (currentRecord.startEditing()) {
					if (isUsingFactor()) {
						currentRecord.setValue(getDataProviderID(),
							numberValue);
					} else {
						currentRecord.setValue(getDataProviderID(),
							getCurrentValue());
					}
					setChanged();
				}
			} else {
				currentFoundset.setSelectedIndex(getCurrentValue()-1);
			}
		}
	}
}
```

Again, it is similar to the Swing bean stateChanged() method except that we don't need to take into account a valueIsAdjusting property (the JavaScript does it for us), same thing for the ignoreUpdate flag that we don't need here since there is no listeners calling the method, only our code, so we can master when to update the record using this method or simply set the local variable value as we did in the updateSlider method()…

Until now we have tried to mimic as much as possible the Swing Slider implementation, using local variables, but there is nothing which links the JavaScript Slider to our Wicket component yet…

This is where we will need to add a Behavior to our component.

But what kind of behavior is it, really?
Well, you need to understand that a Wicket Behavior is a way to enhance a component, and one way to do that is by adding JavaScript to it.
What kind of JavaScript?
That's easy: all we have to do is to have a look at how we build a Slider in a simple html file.
So let's have a look at the Test.html file from the SliderJS folder containing the enhanced version of the Slider that you should have downloaded from the Servoy Stuff web site.

In the HTML header, we can see that we first insert the JavaScript files references:

```
<script type="text/javascript" src="res/js/range.js"></script>
<script type="text/javascript" src="res/js/timer.js"></script>
<script type="text/javascript" src="res/js/slider.js"></script>
```

Then we insert one of the CSS file (depending on the style we want to use) for example:

```
<link type="text/css" rel="StyleSheet" href="res/css/luna/luna.css" />
```

Then some static CSS:

```
<style type="text/css">
#slider-1 {
      vertical-align: top;
      font: normal 11px Tahoma, Verdana, Arial, Sans-Serif;
      overflow: visible;
}
</style>
```

Then in the body of the HTML we add the markup for our Slider, we are lucky it is very lean (and it looks very much like the Wicket markup we had before):

```
<div class="slider" id="slider-1" tabIndex="1">
      <input class="slider-input" id="slider-input-1"/>
</div>
```

In our example we also added an input field to update the value coming from the slider:

```
<div><input type="text" id="valueText" name="valueText" size="3"/></div>
```

Then finally we add some dynamic JavaScript:

```
<script type="text/javascript">

var textEl = document.getElementById ? document.getElementById("valueText") : null;
if (textEl) {
      textEl.onchange = function() {
            window.status = textEl.value;
      };
}

var sliderEl = document.getElementById ?
      document.getElementById("slider-1") : null;
var inputEl = document.getElementById ?
      document.getElementById("slider-input-1") : null;
var s = new Slider(sliderEl, inputEl, "horizontal");
s.setStyleType('luna'); // set to the name of the style sheet

s.setMinimum(0);
s.setMaximum(100);
s.setBlockIncrement(10); // usually equals to minorTicks
s.setMajorTickSpacing(50); // will drive the labels too
s.setMinorTickSpacing(10);
s.setInverted(false); // for natural use when vertical should be true
s.setPaintTicks(true);
s.setPaintLabels(true);
s.setPaintTrack(true);
s.setSnapToTicks(false);
s.setPrecision(3);
s.setUsingDecimals(true);
s.setEnabled(true);
s.onchange = function() {
      var textEl = document.getElementById ?
            document.getElementById("valueText") : null;
      if (textEl) {
            textEl.value = this.getExternalValue();
            textEl.onchange();
      }
};
```

```
s.onblur = function() {
      window.status = "done";
};
s.setExternalValue(0);
textEl.value = "0";
</script>
```

First, this script get a handle of the external text field we added to receive the value and add an onchange method hook to it.

Then it gets a handle of the div enclosing our slider, plus a handle of the input text field of our slider and it constructs a new Slider object from it, precising the orientation (either “horizontal” or “vertical”). Then it sets the styleType (which is the name of the CSS sheet we used), to allow for tweaking the ticks and labels placement based on the style.

And after that it calls the setters on the newly created JavaScript object with all the relevant values.

Then it sets an onchange event hook on the slider (this will update the value of the external text field) and it manually triggers the onchange event of the external text field. I made it like that because that’s exactly what we will have to do with our Wicket component which already has an onchange event attached that will not be triggered automatically (that’s the one coming from our SliderUpdatingBehavior, if you remember well).

Then we set value using the method setExternalValue() (if you look in the source of the slider.js file you will understand that the setExternalValue() method is the one using the precisionFactor()).

You can test it in a browser and see how it goes… Change some parameters to better understand it. But basically this code will have to be inserted in the html page generated by Servoy and this will be done by a Behavior that we will add to the ServoySwingSlider and that we will now create.

JJ. Coding a SliderScriptingBehavior

Our class will be a subclass or AbstractBehavior (org.apache.wicket.behavior.AbstractBehavior).
So let’s add it in our wicket package (create new class, choose the superclass, check “Constructors from superclass”, you know the drill☺)

Our class signature will be:

```
public class SliderScriptingBehavior extends AbstractBehavior {
```

It needs to be Serializable so add the required:

```
      private static final long serialVersionUID = 1L;
```

with a “quick fix”.

From our analysis of the test.html file we know that:
1/ we need to insert javascript and css references to the header
2/ we need the proper html markup to be inserted in the body
3/ we need a dynamic javascript to be inserted in the body as well

While step 2 is already addressed by the ServoyWicketSlider.html template file, here we need to take care of steps 1 and 2.

To be able to add something to the header, we simply need to override a method from the AbstractBehavior class:

```
      public void renderHead(IHeaderResponse response) {
```

To add some dynamic markup (whatever it is – JavaScript in our case), we need to override another method from the AbstractBehavior class:

```
      public void onRendered(Component component) {
```

Let's look at step 1. How do you insert javascript and css references to the header?
Easily enough Wicket is giving you an object as parameter which has all the useful methods we need. The first one is renderCSSReference(), you can pass it a number of different objects, but in our case we need to pass it a reference to one of our CSS file.
We will add some tests later to choose the proper CSS file to insert but the first thing we need to do is to create the references to the 3 CSS that might be added to our header.

We do this by adding some static variables to our class:

```
private static final CompressedResourceReference LUNA_CSS =
      new CompressedResourceReference(SliderScriptingBehavior.class,
            "res/css/luna/luna.css");
private static final CompressedResourceReference SWING_CSS =
      new CompressedResourceReference(SliderScriptingBehavior.class,
            "res/css/swing/swing.css");
private static final CompressedResourceReference BLUECURVE_CSS =
      new CompressedResourceReference(SliderScriptingBehavior.class,
            "res/css/bluecurve/bluecurve.css");
```

CompressedResourceReference is a Wicket class that takes a class and a path relative to that class. It will construct a reference that we will be able to use with the renderCSSReference() method of the IHeaderResponse object we receive in the renderHead() method.

One advantage of this class is that, as its name implies, it compresses the file before sending it to the client's browser (using gzip compression, if the browser accepts it).

We will use the same kind of reference for our JavaScript files, so let's add them as static variables to our class:

```
private static final JavascriptResourceReference RANGE_JS =
new JavascriptResourceReference(SliderScriptingBehavior.class, "res/js/range.js");
private static final JavascriptResourceReference TIMER_JS =
new JavascriptResourceReference(SliderScriptingBehavior.class, "res/js/timer.js");
private static final JavascriptResourceReference SLIDER_JS =
new JavascriptResourceReference(SliderScriptingBehavior.class, "res/js/slider.js");
```

This time we are using another very useful Wicket class: JavascriptResourceReference, which takes a class and a path relative to it to retrieve the javascript. This class has the advantage of first stripping all spaces from the JavaScript file, then compressing it using gzip before sending it to the client's browser, it is especially useful if your js files have not been compacted before (which is our case).

We will use these reference in our renderHead() method using another method of the IHeaderResponse object we receive: renderJavascriptReference() which takes a ResourceReference as parameter.

So right now our renderHead() method would look like this:

```
@Override
public void renderHead(IHeaderResponse response) {
      super.renderHead(response);

      response.renderCSSReference(LUNA_CSS);
      response.renderJavascriptReference(RANGE_JS);
      response.renderJavascriptReference(TIMER_JS);
      response.renderJavascriptReference(SLIDER_JS);
}
```

After a call to super, we set a CSS ad the 3 required JavaScript files.

KK. Doing it with style

We hard-coded the use of the “luna.css” style...
What about the other 2 possible styles? It would be nice to let the user choose which one to use in the web client. To do this, we will need to add a property to our ServoySlider class, along with a getter and a setter:

```
      private String webStyleType = "luna";

      public String getWebStyleType() {
            return this.webStyleType;
      }

      public void setWebStyleType(String webStyleType) {
            this.webStyleType = webStyleType;
            if (component != null) {
                  component.setWebStyleType(webStyleType);
            }
      }
```

But our component right now doesn’t have a setWebStyle() method, and in fact it is our interface IServoySliderBean that doesn’t declare this in its contract. So let’s add these 2 method signatures in the IServoySliderBean:

```
      public void setWebStyleType(String webStyleType);
      public String getWebStyleType();
```

Meaning that we will need to add a property and the related accessors into our ServoySwingSlider and our ServoyWicketSlider class:

```
      private String webStyleType;

      public String getWebStyleType() {
            return this.webStyleType;
      }
      public void setWebStyleType(String webStyleType) {
            this.webStyleType = webStyleType;
      }
```

And in the SliderScriptingBehavior class, to allow access to the parent’s properties, we will need to add the parent as a variable:

```
      private ServoyWicketSlider parent;
```

And a constructor with a parent reference:

```
      public SliderScriptingBehavior(ServoyWicketSlider parent) {
            super();
            this.parent = parent;
      }
```

This will be called in our ServoyWicketSlider in its own constructor method like this:

```
          add(new SliderScriptingBehavior(this));
```

Now we can code our renderHead() method this way:

```
ResourceReference sheet = LUNA_CSS;
if (Utils.stringSafeEquals(parent.getWebStyleType(), "swing")) {
      sheet = SWING_CSS;
} else if (Utils.stringSafeEquals(parent.getWebStyleType(),"bluecurve")) {
      sheet = BLUECURVE_CSS;
}
response.renderCSSReference(sheet);
```

But how can we constrain the user to type only the relevant style Strings and not something that we will not recognize as one of the 3 valid values? In other word, how do we do a “ValueList” property editor, where the user will only be allowed a certain set of values from a dropdown?

There is a way! And it’s kinda easy too! We need to create a class that implements the PropertyEditor interface (or even more easily one that extends the PropertyEditorSupport class, which is an adapter class). All we will need to override is the getTag() methods, so let’s do it.

Add a class in the net.stuff.servoy.beans.slider package, name it WebStyleTypePropertyEditor, it needs to extend (its superclass must be) “java.beans.PropertyEditorSupport”, so our signature will be:

```
public class WebStyleTypePropertyEditor extends PropertyEditorSupport {
```

Then in the source menu, choose “Override/Implement Methods...”, then choose the getTag() method. In this method you need to return a String[] array: a list of values!

```
@Override
public String[] getTags() {
      return new String[] { "luna", "swing", "bluecurve" };
}
```

Actually we will use constants and put them in a place accessible to our ServoyWicketSlider and any Behaviors too, in the IServoySliderBean interface for example:

```
      public static final String LUNA_STYLE = "luna";
      public static final String SWING_STYLE = "swing";
      public static final String BLUECURVE_STYLE = "bluecurve";
```

So that our WebStyleTypePropertyEditor class getTags() method will be:

```
      @Override
      public String[] getTags() {
            return new String[] {
                        IServoySliderBean.LUNA_STYLE,
                        IServoySliderBean.SWING_STYLE,
                        IServoySliderBean.BLUECURVE_STYLE
                  };
      }
```

And our webStyleType variable declaration in the ServoySlider class will be (setting the default):

```
      private String webStyleType = IServoySliderBean.LUNA_STYLE;
```

And the beginning of our renderHead() method in the SliderScriptingBehavior will be:

```
ResourceReference sheet = LUNA_CSS;
if (Utils.stringSafeEquals(parent.getWebStyleType(),
            IServoySliderBean.SWING_STYLE)) {
      sheet = SWING_CSS;
} else if (Utils.stringSafeEquals(parent.getWebStyleType(),
            IServoySliderBean.BLUECURVE_STYLE)) {
      sheet = BLUECURVE_CSS;
}
response.renderCSSReference(sheet);
```

Note the use of the Utils (com.servoy.j2db.util.Utils) stringSafeEquals method that we already saw before, to check for equality between 2 Strings with not risk of having NullPointerExceptions…

One thing remains to do though to get use of our new WebStyleTypePropertyEditor: we need to add it (along with the property) in the ServoySliderBeanInfo class, in the getPropertyDescriptors() method, so we add the following (in the try block before the creation of the PropertyDescriptor[] array from the List):

```
      pd = new PropertyDescriptor("webStyleType", ServoySlider.class);
      pd.setPropertyEditorClass(WebStyleTypePropertyEditor.class);
      liste.add(pd);
```

You could implement a property editor from JFrame or JDialog and create a whole custom interface to edit the property, except that I tried it and Servoy actually doesn't instantiate the Custom Component from the PropertyEditor properly if they are not of the basic type we made here…
I'm waiting to see what Tano has to say about this…

Anyway, back to our SliderScriptingBehavior. We just add a way to set the CSS and insert it in the header according to the user's choice, now let's look at all the other properties…

LL. Implementing the SliderScriptingBehavior

Next we need to add some inline css styles (they are related to the id of our component), so this time we will use the simple renderString() method, which will insert a String "as-is" in the header – there are warning about possible misuses of this method, so we need to take care that our String is properly constructed:

We can do this:

```
      final String ID = parent.getId();

      StringBuffer css = new StringBuffer();
      css.append(CssUtils.INLINE_OPEN_TAG);
      css.append("#"+ID +"{\n");
      css.append("\tvertical-align: top;\n");
      css.append ("\tfont: normal 11px Tahoma, Verdana, Arial, Sans-Serif;\n");
      css.append("\toverflow: visible;\n");
      css.append("}");
      css.append(CssUtils.INLINE_CLOSE_TAG);
      response.renderString(css.toString());
```

Note that we use the parent's ID as the selector of our CSS, since the container div will have its html markup id set by Wicket from the getMarkupId() method of our ServoyWicketSlider class.

Note also the use of the CssUtils (Wicket class) open and close static Strings.

Next: in the test.html file we saw that we made the creation of the Slider object after the insertion of the markup (div and input tags). To do that we could add the script to the onRendered() method of our SliderScriptingBehavior class, but the problem would be that each time the component is updated the Slider would be initialized again, which is not very efficient (especially with the methods that calculate the ticks and labels). So what?

It would be nice if we could add the Slider object creation to the header, while putting only the property setters relevant to the updates in the onRendered() method. But one problem with this is that the header method will be called before the insertion of the div and input tag that we need to pass as parameter to the JS Slider object constructor. Still, there is a way to do that, and it is by using the renderOnDomReadyJavascript() method of the IHeaderResponse object we got.

The call will only be done to our script once all the DOM of the html file has been properly constructed: meaning that the div and input we need will already be there!

So we could go ahead and start writing this:

```
final String markupID = parent.getMarkupId();
final String orientation = (parent.getOrientation() == 0) ? "horizontal" :
"vertical"; // the JavaScript object is using a String while the Swing bean uses
int

StringBuffer buff = new StringBuffer();
buff.append("var inputEl = document.getElementById ? document.getElementById('" +
markupID + "') : null;\n");
buff.append("var backingEl = document.getElementById ?
document.getElementById('slider-input" + ID + "') : null;\n");
buff.append("var s = new Slider(inputEl, backingEl, '" + orientation + "');\n");
```

And then:
```
response.renderOnDomReadyJavascript(buff.toString());
```

But the problem would now be with the way Wicket is inserting our code.
If you would look at the html produced by Servoy on a page from the use of the above code, you would see something like this:
```
Wicket.Event.add(window, "domready", function() { var sliderEl =
document.getElementById ? …etc…
```

As you can imagine, being wrapped inside a function mean that every variable created using the “var” keyword would create a local variable accessible only to that function. Now that would be a problem because later on we will need to get access to our “s” variable (the Slider).

To avoid this problem, we can actually setup a “global” variable inside our page, before we set our “domready” script. And while we are at it, we will need to use a unique name for our variable (because “s” is not very unique a name, is it? And if we want to insert more than one Slider on a form there are going to be conflicts)… So instead of s, we will construct a name based on something we know for sure is going to be unique: our Wicket ID.

So before our call to the domready method, let’s add this code to our method:
```
final String varID = "g_" + ID;
response.renderJavascript("var "+varID+";", varID);
```

We constructed a unique name from the wicket ID (we added a prefix to it to avoid conflict with the ID object itself), and added this “var” to our header. This variable will now be available from any script on the page!

So now, we can code our method the same way but assign the local objet to the “global” unique variable:

```
buff.append("var sliderEl = document.getElementById ? document.getElementById('" +
markupID + "') : null;\n");
buff.append("var inputEl = document.getElementById ?
document.getElementById('slider-input" + ID + "') : null;\n");
buff.append("var s = new Slider(sliderEl, inputEl, '" + orientation + "');\n");
buff.append(varID+" = s;\n");
```

The last line here is making the assignment of our local variable (“s” is OK here since it is enclosed in the function) to our “g_+ID” global variable.

Now we can code the rest of the JavaScript object property initialization:

```
buff.append(varID+".setOrientation('" + orientation + "');\n");
buff.append(varID+".setStyleType('" + parent.getWebStyleType() +
buff.append(varID+".setMinimum("+parent.getMinimumValue()+");\n");
buff.append(varID+".setMaximum("+parent.getMaximumValue()+");\n");
buff.append(varID+".setMajorTickSpacing("+parent.getMajorTicks()+");\n");
buff.append(varID+".setMinorTickSpacing("+parent.getMinorTicks()+");\n");
buff.append(varID+".setUnitIncrement(1);\n");
buff.append(varID+".setBlockIncrement("+parent.getMinorTicks()+");\n");
buff.append(varID+".setInverted("+parent.getInverted()+");\n");
buff.append(varID+".setPaintTicks("+parent.getPaintTicks()+");\n");
buff.append(varID+".setPaintLabels("+parent.getPaintLabels()+");\n");
buff.append(varID+".setPaintTrack("+parent.getPaintTrack()+");\n");
buff.append(varID+".setSnapToTicks("+parent.getSnapToTicks()+");\n");
buff.append(varID+".setPrecision("+parent.getPrecision()+");\n");
buff.append(varID+".setUsingDecimals("+parent.isUsingDecimals()+");\n");
buff.append(varID+".setExternalValue("+parent.getNumberValue()+");\n");
buff.append(varID+".setEnabled("+parent.isEnabled()+");\n");

response.renderOnDomReadyJavascript(buff.toString());
```

Actually, there are still a few problems here. Did you notice?
First, we make sure that our slider div has a unique ID, as well as the global Slider objet, but the “input-slider” element (the text field) still has a simple name so if we put more than one slider on a form this will create conflicts. We need to make it unique too!

Let’s do this in the ServoyWicketSlider’s constructor, where we had:

```
TextField field = new TextField("slider-input", new PropertyModel(this,
"numberValue"), Double.class);
add(field);
```

Let’s do it this way:

```
TextField field = new TextField("slider-input", new PropertyModel(this,
"numberValue"), Double.class);
field.setMarkupId("slider-input-"+getId());
field.setOutputMarkupId(true);
add(field);
```

Now that we have a unique name for our input field, we can change this line:

```
buff.append("var inputEl = document.getElementById ?
document.getElementById('slider-input" + ID + "') : null;\n");
```

from the SliderScriptingBehavior renderHead() method by this one:

```
buff.append("var backingEl = document.getElementById ?
document.getElementById('slider-input-" + ID + "') : null;\n");
```

If you were to do it this way in your Behavior, you would see that it works, only you would find another problem: the “onchange” event that we made sure to fire in our SliderUpdatingBehavior in the previous part of this tutorial will never fire because it will be replaced by the inner event method of our Slider object!

I have tried a few things to make it triggered from the slider.js file, but I failed, because basically the object is not Wicket Ajax ready!

So I have found a workaround, one that is based on the same trick that we used in our test.html file: remember that we had an external input field that also have an onchange method, and that we could update this field and trigger its “onchange” method from the “onchange” method of our Slider object? The trick is to do exactly the same thing, but using the current input TextField (which is bound to the Servoy’s record value in our code) as the external input field: meaning that we will need another input field that the Slider will use as its “backing” value holder.

Let’s do that… First, we change the ServoyWicketSlider.html markup to the following:

```
<wicket:panel>
      <input wicket:id="slider-input" type="text" class="slider-input"/>
      <input wicket:id="slider-backing" type="text" class="slider-input"
style="display:none" />
</wicket:panel>
```

Note that we got rid of the table layout since our Slider’s CSS will take care of laying out its elements.
Note that our backing input field is forced to be invisible (even when JavaScript is not used in the browser, in which case the input field would be seen, but not the backing field)

Now in the ServoyWicketSlider class we add the new “backing” input field to our Panel:

```
TextField backing = new TextField("slider-backing", new PropertyModel(this,
"numberValue"), Double.class
backing.setMarkupId("slider-backing-"+getId());
backing.setOutputMarkupId(true);
add(backing);
```

We make sure that our backing input field has a unique ID and that it is rendered in the final html. Then we can finally change our SliderScriptingBehavior class renderHead piece of code to:

```
buff.append("var backingEl = document.getElementById ?
document.getElementById('slider-backing-" + ID + "') : null;\n");
buff.append("var s = new Slider(sliderEl, backingEl, '" + orientation + "');\n");
```

And after all the properties have been set, we can now add an "onchange" method to our Slider object, like this:

```
buff.append(varID+".onchange = function() {\n");
buff.append("\tvar inputEl = document.getElementById ?
document.getElementById('slider-input-" + ID + "') : null;\n");
buff.append("\tif(inputEl) {\n");
buff.append("\t\tinputEl.value = this.getExternalValue();\n");
buff.append("\t\tinputEl.onchange();\n");
buff.append("\t};\n");
buff.append("};\n");
```

This will perform the trick, setting up the value of our "real" input text field to the value of the Slider object, triggering the Ajax call set on the "onchange" event of the "slider-input" text field.

You would think: OK, now we're done! But not quite, because there is still a problem here:
All the properties of our Slider object will be properly initialized, yes, but what happens if we script them (meaning if they need to change on Ajax updates?). You remember that we coded all the js_xxx methods in our ServoyWicketSlider, so we will need to update the values each time the component is redrawn (even from an Ajax update).

We said that we were going to use the onRendered() method that we override from AbstractBehavior. Basically in this method we will need to update all the properties that changed (not the other ones! If you look into the slider.js implementation you will see that it is not really made for Ajax updates, and if you reset the values each time you update the component, you will have the ticks, labels and position of the handle of the slider jump back an forth during the update, which is not very nice!).

So, because we want to update only the relevant values, we need a way to store the previous values for each of them. Are we going to add 15 "previous" variables? Seems like a lot of code, so let's do it differently, let's put the values into a Hashtable and check the new values against them.

We add a "map" variable to our class like this:

```
	private Hashtable map = new Hashtable();
```

Note that I am using a Hashtable here instead of a HashMap because it is ThreadSafe (look for this on the internet…)

Then we will need a few key constants so let's add all these too:

```
	private static final String STYLE_TYPE = "styleType";
	private static final String ORIENTATION = "orientation";
	private static final String ENABLED = "enabled";
	private static final String MAXIMUM = "maximumValue";
	private static final String MINIMUM = "minimumValue";
	private static final String MAJOR_TICKS = "majorTicks";
	private static final String MINOR_TICKS = "minorTicks";
	private static final String PRECISION = "precision";
	private static final String INVERTED = "inverted";
	private static final String PAINT_TICKS = "paintTicks";
	private static final String PAINT_LABELS = "paintLabels";
	private static final String PAINT_TRACK = "paintTrack";
	private static final String SNAP_TO_TICKS = "snapToTicks";
	private static final String USING_DECIMALS = "usingDecimals";
	private static final String NUMBER_VALUE = "numberValue";
```

So now, our code will insert the setters of our JavaScript Slider object conditionally, like this:

```
if (!Utils.stringSafeEquals(orientation, (String)map.get(ORIENTATION))) {
     buff.append(varID+".setOrientation('" + orientation + "');\n");
     map.put(ORIENTATION, orientation);
 }
if (!Utils.stringSafeEquals(parent.getWebStyleType(), (String)map.get(STYLE_TYPE)))
{
     buff.append(varID+".setStyleType('" + parent.getWebStyleType() + "');\n");
     map.put(STYLE_TYPE, parent.getWebStyleType());
}
```

Etc...

Here we are testing the equality of our saved value (if it exists) with the possible new value from a call to our parent variable’s getter methods.

But we will need some more utility methods for boolean, int and double, so we can do our own, and put it into our ServoyUtils class:

```
      public static boolean booleanEquals(boolean a, Boolean b) {
            return (b != null && a == b.booleanValue());
      }

      public static boolean intEquals(int a, Integer b) {
            return (b != null && a == b.intValue());
      }

      public static boolean doubleEquals(double a, Double b) {
            return (b != null && a == b.doubleValue());
      }
```

Note that these methods are using a primitive value (the parameter retrieved from the parent’s getter methods) and the related object (as retrieved from our map, which might return “null” if the previous value was not set).

But if we can now continue implementing these calls in our onRendered() method, we could actually use the same code for both the initialization and the update calls, it would be better than having code duplicate don’t you think? For example have a method which will return a StringBuffer object, and take a boolean parameter for “header” insert or update.

This is the method I propose in the end:

```
/**
 * @param header true if first insert
 * @return StringBuffer
 */
private StringBuffer insertPropertyScripts(boolean header) {
      final String ID = parent.getId();
      final String orientation = (parent.getOrientation() == 0) ? "horizontal" :
"vertical";
      final String varID = "g_" + ID;

      StringBuffer buff = new StringBuffer();

      if (header || !Utils.stringSafeEquals(orientation,
            (String)map.get(ORIENTATION))) {
            buff.append(varID+".setOrientation('" + orientation + "');\n");
            map.put(ORIENTATION, orientation);
      }
      if (header || !Utils.stringSafeEquals(parent.getWebStyleType(),
            (String)map.get(STYLE_TYPE))) {
            buff.append(varID+".setStyleType('" + parent.getWebStyleType() +
"');\n");
            map.put(STYLE_TYPE, parent.getWebStyleType());
      }

      if (header || !ServoyUtils.intEquals(parent.getMinimumValue(),
(Integer)map.get(MINIMUM))) {
            buff.append(varID+".setMinimum("+parent.getMinimumValue()+");\n");
            map.put(MINIMUM, parent.getMinimumValue());
      }
      if (header || !ServoyUtils.intEquals(parent.getMaximumValue(),
(Integer)map.get(MAXIMUM))) {
            buff.append(varID+".setMaximum("+parent.getMaximumValue()+");\n");
            map.put(MAXIMUM, parent.getMaximumValue());
      }

// ... etc. see the Eclipse Project code for the whole method

            return buff;
      }
```

Anyway, now we can now get rid of the call to the JavaScript setters in our renderHead() method and replace it with:

```
      buff.append(insertPropertyScripts(true));
```

While the onRendered() method will finally look like this:

```
@Override
public void onRendered(Component component) {
      Response response = component.getResponse();

      StringBuffer buff = insertPropertyScripts(false);
      if (buff.length() > 0) {
            response.write(JavascriptUtils.SCRIPT_OPEN_TAG);
            response.write(buff.toString());
            response.write(JavascriptUtils.SCRIPT_CLOSE_TAG);
      }
      super.onRendered(component);
}
```

Note the test of our StringBuffer length() method to see if we really need to insert something.
Note the use of JavascriptUtils (a Wicket class) static String for open and close <script> tags

That's pretty much it for this class which is at the heart of our external JavaScript integration.
It's almost time to wrap it all up!

You can deploy the bean and test it in Servoy, hopefully it is working properly, if not, check you code against the code provided in the updated Eclipse project download.

We will tackle one last problem that you probably didn't even notice before wrapping up.

MM. Why being non American can sometimes help...

This title is not intended to offense anyone but to reflect a sad truth about software in general, as non US users like me are usually plagued by far more bugs than they should...

As you might know, I'm French (yes, that also explains my terrible abuse of the English language in these tutorials, I know!), and I have sometimes noticed that using another language/locale by default instead of "en/US" helps quite often to trap some very well hidden bugs.

It turns out that when having my browser's language set to "French" by default, the values of the text input as set by Wicket for double number were using a comma (,) instead of a dot (.) as the decimal separator. But then when the Slider JavaScript object was setting up the value it was using a (.) while Wicket was still waiting for a (,), so it only kept the integer part of the number... What a mess!

To avoid this problem, since the JavaScript is using a dot anyway, and since the input text is hidden, I resolved to force the Locale of the Wicket converter to be Locale.US regardless of the user's locale. This is done in the ServoyWicketSlider constructor like this:

```
TextField field = new TextField("slider-input", new PropertyModel(this,
"numberValue"), Double.class) {
      private static final long serialVersionUID = 1L;

      @Override
      public IConverter getConverter(Class c) {
            return new SliderDoubleConverter();
      }
};
```

That's right! You can see here that I am creating a subclass of the TextField "on-the-fly", overriding one of its method to return my own Converter. That's a useful shortcut allowed since Java 1.5. Using this type of construct I don't need to create a new class just to override one method.

To know more about how Wicket is doing conversion, you can have a look at the IConverter interface; you can also peep into the code of the DoubleConverter class that I am going to subclass here in a new class that I created in the net.stuff.servoy.bean.slider.wicket package. Here is the code for the whole class:

```
public class SliderDoubleConverter extends DoubleConverter {

      private static final long serialVersionUID = 1L;

      @Override
      public NumberFormat getNumberFormat(Locale locale) {
            return NumberFormat.getInstance(Locale.US);
      }
}
```

We only need to override the getNumberFormat() method to force the use of a Locale.US based instance of a NumberFormat, which will be used by our text fields.

The astute reader could remark here that I could also have constructed this class "on-the-fly". The code would have been:

```
TextField field = new TextField("slider-input", new PropertyModel(this,
"numberValue"), Double.class) {
      private static final long serialVersionUID = 1L;

      @Override
      public IConverter getConverter(Class c) {
            return new DoubleConverter() {
                  private static final long serialVersionUID = 1L;

                  @Override
                  public NumberFormat getNumberFormat(Locale locale) {
                        return NumberFormat.getInstance(Locale.US);
                  }

            };
      }
};
```

Although the code above is perfectly legal Java 1.5 code, I felt that it was starting to be poorly readable, so I resolved to create my own subclass of DoubleConverter. Feel free to do it differently if you prefer, the compiler will accept it without problem anyway…
I always prefer to keep my code readable than to gain a few lines or a few CPU cycles. But that's my personal view, and not everyone is sharing it anyway ;-)

Back to our constructor method, we also need to add the same code to our "backing" TextField:

```
TextField field = new TextField("slider-input", new PropertyModel(this,
"numberValue"), Double.class) {
      private static final long serialVersionUID = 1L;

      @Override
      public IConverter getConverter(Class c) {
            return new SliderDoubleConverter();
      }
};
```

This is it! We really are done for this part!

You will be able to deploy the bean in Servoy, and test it in your preferred browser, try the different CSS Styles, try it on different fields, as a navigator bean, etc... The JavaScript implementation is not perfect, and it is not perfectly reflecting the Swing Slider but it is reasonably close IMHO.
Of course if you want, you can make some enhancements (there's always room for improvement, remember?) and if you find a bug, try to find a workaround and send it back so that others will benefit from your experience too and learn from it and have a nice reliable component to use in their solutions…

I know for example that in Firefox, the Slider is initializing itself after each Ajax updates, which is very annoying. I haven't found a workaround for that yet… It might involve a lot of refactoring of the slider.js file which is already very complicated for me with my limited confidence in browser's JavaScript coding. If you find a way to avoid this problem, please tell me, and submit the code so that we will all benefit from it!

As usual, you will find the complete Eclipse project on the Servoy Stuff web site, here:
http://www.servoy-stuff.net/tutorials/utils/t02/v5/ServoySlider_EclipseProject-v5.zip
(Get rid of the previous project of the same name and import in Eclipse)

The compiled bean (targeted for java 1.5) will be available here:
http://www.servoy-stuff.net/tutorials/utils/t02/v5/servoy_slider.jar
(Put in you /beans folder)

And the little “beans_tests” solution updated to use the new bean in situation will be available at:
http://www.servoy-stuff.net/tutorials/utils/t02/v5/beans_tests-v5.zip
(Unzip and import in Servoy 4.1.x)

And there is also the complete JavaScript project based on the WebFX code, but enhanced for our integration in Servoy, that you can download here:
http://www.servoy-stuff.net/tutorials/utils/t02/v5/SliderJS_EclipseProject.zip
It is also an Eclipse project, but you can use it without importing it if you prefer; simply unzip somewhere on your hard disk and peek into the JavaScript of the test.html file and of course the /res/js/slider.js file.

Hope you enjoyed this part!

I thought that this would be the end of this series but someone asked me for details on how to implement events in a Servoy bean, and hook these events with Servoy’s JavaScript method, and it is indeed an interesting topic (one that I have been discussing recently with Johan Compagner who gave me some pointers for a possibly better implementation – the discussion was about the busy plugin but it is relevant to any call-back/event implementation, so I will try to give you a resume of all this when appropriate).

In fact the event/call-back topic is big enough to justify another part so we will definitely have a (hopefully final) part 8 soon, dedicated to that!

And we will probably conclude with a look at what is cooking in the upcoming Servoy Tano, related to plugins and beans as the public API is finally benefiting from a massive update (lots of new and exciting classes and interfaces, long awaited “not so public” classes and interfaces now fully exposed, etc…), go and have a look for yourself:
http://www.servoy.com/docs/public-api/5xx/index.html
This is going to be big!

In the meantime, have fun with the now fully web compatible Slider in your preferred browser, show it around and explain how you did it ;-)

Patrick Talbot
Servoy Stuff
2009-08-03