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

**PART 2**

# E. What is wrong with regular Java Beans?

In the first part of this tutorial, we showed how to use a regular JSlider bean, and how to script it in Servoy to make it interact with the Servoy environment.

Although this method is working fine, and can definitely be used when you want to integrate a bean quickly in Servoy, it still has its drawbacks. I don't know how many of you guessed what they were already but the most obvious is the amount of scripting you need to add to your form to have a basic integration and make the regular bean aware of its environment.

No less than 5 methods and 2 variables were needed on the form to make the slider bean function properly.

And yet, it is not finished! I don't know if you tried it, but if you go into find mode, or do a search and thus change the number of records in the foundset, the slider bean will not be aware of that, and it will still display and try to operate on the total foundset.

To workaround that, we could extract the portion of code that was in the FORM_onShow() method and create a new function, like this:

```
function ACTION_setSliderValues()
{
	var max = foundset.getSize();
	elements.slider.maximum = max;
	// we want a maximum of 10 major ticks:
	var tickSpacing = Math.round(max/10);
	elements.slider.majorTickSpacing = tickSpacing;
	// and no minor ticks:
	elements.slider.minorTickSpacing = 0;
}
```

Then call it from the FORM_onShow() (instead of having the code duplicated – remember DRY?), and also call it from the REC_selectRecord() function like that:

```
function REC_selectRecord()
{
	index = controller.getSelectedIndex()
	ACTION_setSliderValues();
	elements.slider.setValue(index);
}
```

Now even if the foundset total number of record changes, the slider will follow.

But when you are in Find mode, it still believes that it can operate on a full number of records, which still feels a bit odd, so we would need to hook a method to the onFindCmd event of the form, like that for example:

```
function ACTION_sliderDisable()
{
	elements.slider.enabled = false;
	controller.find();
}
```

And also another one to the onSearchCmd and onShowAllRecordsCmd form events, like this:

```
function ACTION_sliderEnable()
{
	if (foundset.isInFind()) {
		controller.search();
	}
	elements.slider.enabled = true;
}
```

Now our bean is disabled when we are in find mode, and enabled again when we search and updated with the proper number of records.

That's a lot of scripting! We now have 8 methods in our simple form just to handle the bean/ And if you need to integrate in many forms, you will have to put this kind of code (or an adaptation of it to make it work in a more generic way, in a module for example).

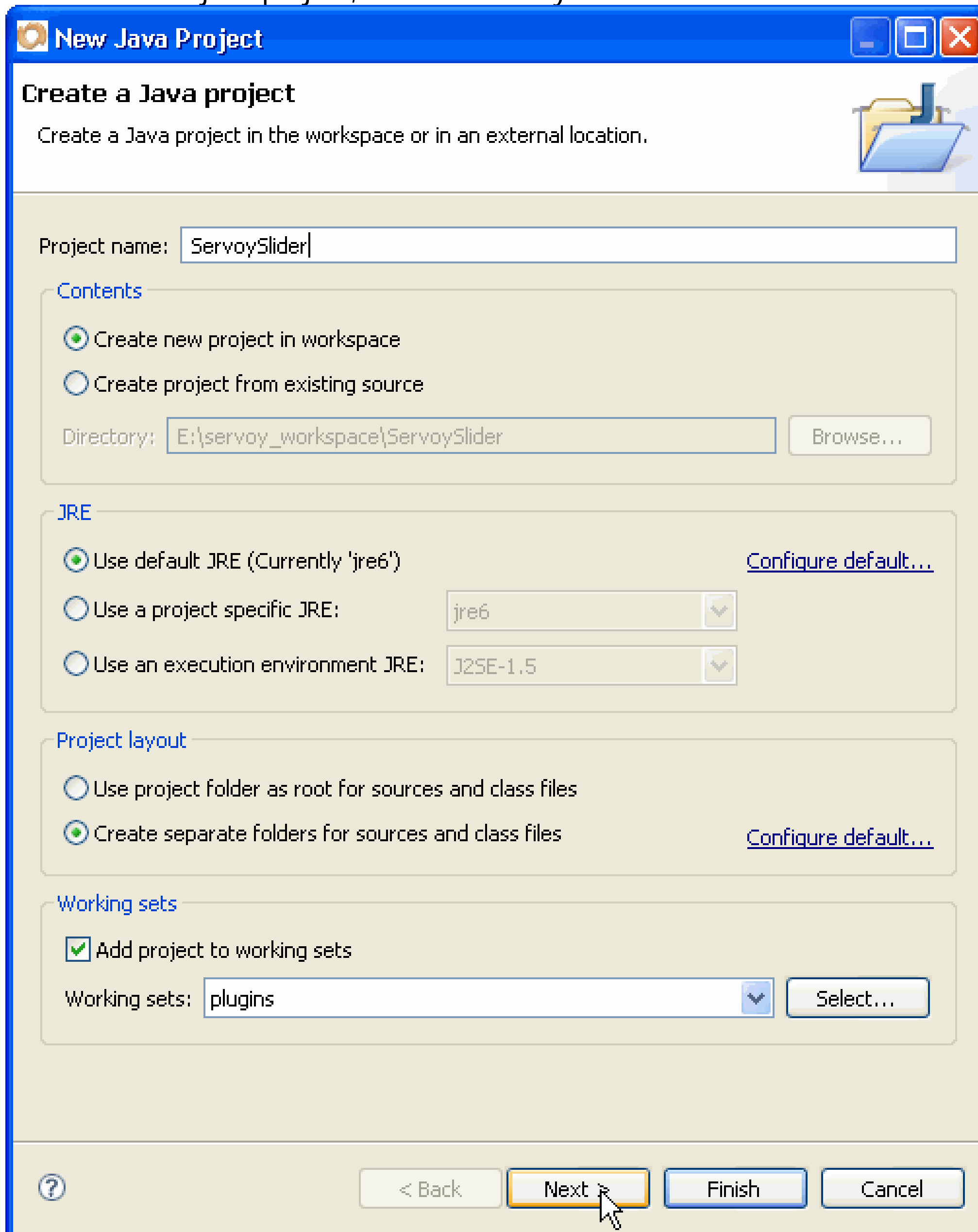
All this because our bean has no idea of the environment it is put in, in a word it is not Servoy-Aware!

This is how our goal now will be to build a JSlider based component, which will be Servoy-Aware, to help reducing the amount of scripting needed to integrate it in Servoy, thus making it more reusable.

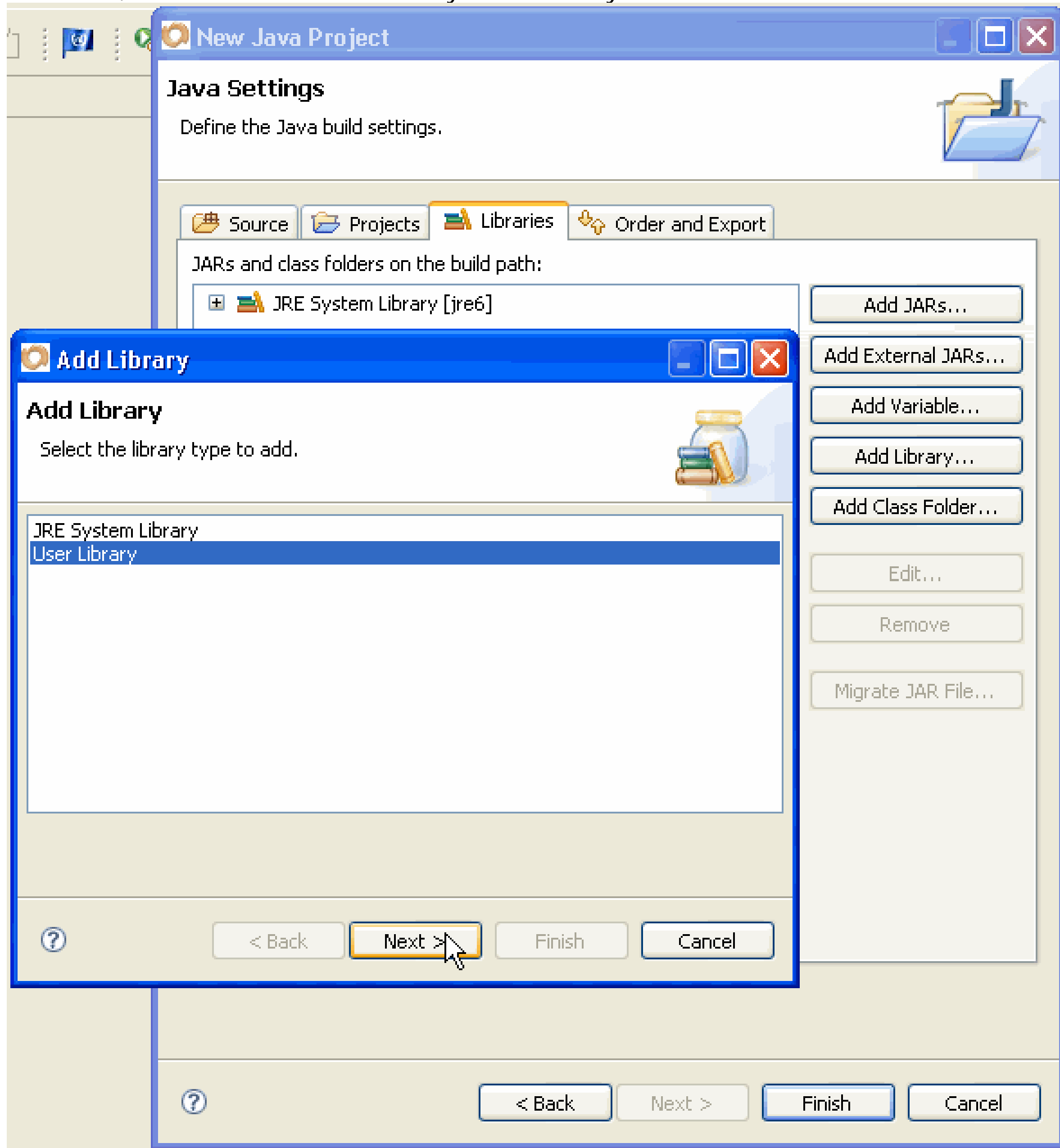
So let's launch our Servoy (any Eclipse distribution will do just fine too of course) in "Java" mode, switch to the "Java" perspective, and create a new Java project…

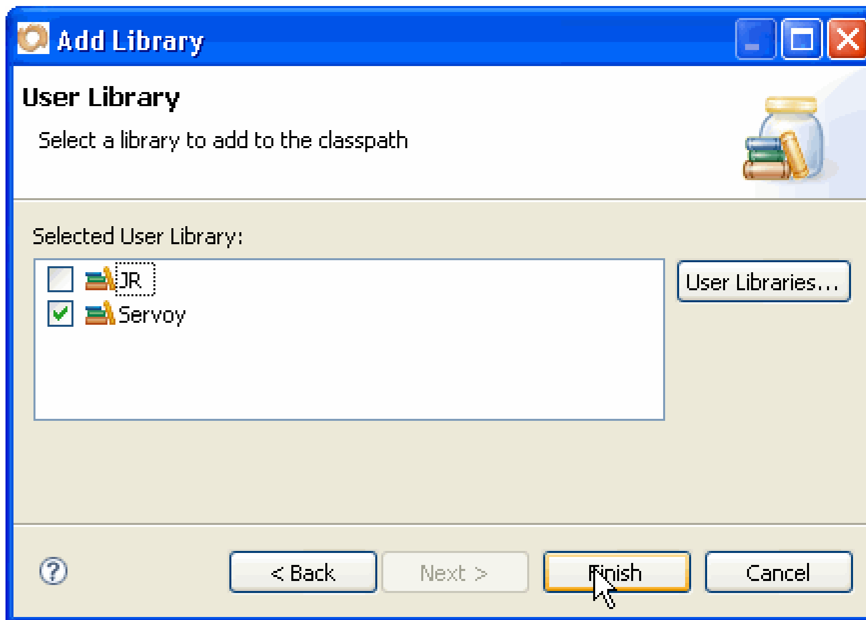
# F. The ServoySlider bean project

So we create a java project, name it "ServoySlider":



Of course, we need to add the "Servoy" User Library:






The "Servoy" User Library should contain these jars:

New Java Project

Java Settings

Define the Java build settings.

Source | Projects | Libraries | Order and Export

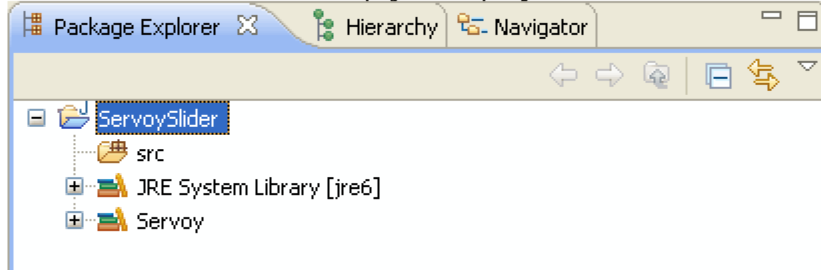
JARs and class folders on the build path:

- JRE System Library [J2SE-1.5]
- Servoy
  - Access rules: No rules defined
  - Native library location: (None)
  - j2db.jar - C:\Program Files\Servoy\application_server\lib
  - j2dbdev.jar - C:\Program Files\Servoy\application_server\lib
  - js.jar - C:\Program Files\Servoy\application_server\lib
  - wicket.jar - C:\Program Files\Servoy\application_server\lib
  - wicket-extentions.jar - C:\Program Files\Servoy\application_server\lib
  - wicket-calendar.jar - C:\Program Files\Servoy\application_server\lib
  - joda-time.jar - C:\Program Files\Servoy\application_server\lib

Add JARs...
Add External JARs...
Add Variable...
Add Library...
Add Class Folder...
Edit...
Remove
Migrate JAR File...

< Back | Next > | Finish | Cancel

So now, we have a nice empty Java project that shows in the Package Explorer like that:




Now, how are we going to take an innocent JSlider and make it "aware" of the Servoy environment?

A quick search in the Servoy public API for 4.1.x (and even in the 3.5.x API) doesn't give us too much hope. The only way right now, if we believe the API seems to build a plugin to get a hold of an **IClientPluginAccess** (the Servoy running client), which by way of its getFormManager() method could give us access to an **IFormManager** object, which in turn, by way of its getCurrentForm() could give us access to an **IForm** which in turn could give us access to an **IFoundSet** via its getFoundSet() method, except that this method is commented with the following disclaimer: CURRENTLY FOR INTERNAL USE ONLY, DO NOT CALL.

Now that's a bummer!

The catch is that a few weeks ago this disclaimer was not even here! And there's even more: there is another interesting interface that we will use later, the **IServoyAwareBean** interface, this one was there and I used it with Servoy 4.1.2 and 4.1.3 without any problems, but somehow, mysteriously, it just disappeared, and is now only available in the 4.2.x version of the API!

If you are like me and you think that this 4.2 version (aka Tano) is a long way from being released, then you want to try it in 4.1 anyway, I can guarantee you that it was there, and it works, as of 4.1.2+ anyway!

So let's get a look in the 4.2.x version of the API, and sure enough, if I look at the **IForm** getFoundSet(), the disclaimer is not here, and I can access the foundset with the related interface **IFoundSet** and in this interface, there is lots of very interesting methods, like getSize(), isInFindMode(),getSelectedIndex(), and even a way to add a listener addFoundSetEventListener(IFoundSetEventListener l) which will notify our component each time the foundset changes, with a **FoundSetEvent** event.

There's everything we need here!

All we need to decide is whether we are going to get a hold of the foundset via the **IClientPluginAcess** route or via the **IServoyAwareBean** route.

The only difference here is how we want to use the bean in the end: if we want to use it as a regular bean, putting it on a form set a few properties and voilà! Or if we want it to appear in the plugins node of the Servoy Solution Explorer, meaning that we will have to instantiate the bean via scripting.

I don't know about you, but I much prefer the second solution, so let's go ahead and implement our IServoyAwareBean.

# G. The IServoyAwareBean interface

First let's have a look at the javadocs to see what we need to implement:

com.servoy.j2db.dataui

## Interface IServoyAwareBean

**All Superinterfaces:**

IDisplay

---

```
public interface IServoyAwareBean
extends IDisplay
```

Interface to be used by beans to make them aware of Servoy.

---

| Method Summary | |
|---|---|
| void | `initialize(IClientPluginAccess access)`<br>Initializes the bean. |
| void | `setSelectedRecord(IRecord selectedRecord)`<br>Applies the record (currently selected) to the bean. |

| Methods inherited from interface com.servoy.j2db.dataprocessing.IDisplay |
|---|
| `isEnabled, isReadOnly, setValidationEnabled, stopUIEditing` |

| Method Detail |
|---|

There's not much here apparently, just 2 methods:

- *initialize*(), which will be passed an IClientPluginAccess object (our Servoy client), which we can use to get a hold of the foundset (see above)
- *setSelectedRecord*() which will pass an IRecord and is the java equivalent to the onRecordSelection event on a Servoy form

But wait, the interface also extends the IDisplay interface, so we need to look at this one as well:

com.servoy.j2db.dataprocessing

## Interface IDisplay

**All Known Subinterfaces:**
IServoyAwareBean

---

`public interface IDisplay`

Interface for components that have some additional functionality.
It is meant for components that are linked to dataProviders.

---

| Method Summary | |
|---|---|
| boolean | **isEnabled**()<br>Enabled displays are displays the user can interact with and are an active part of the UI. |
| boolean | **isReadOnly**()<br>Read-only displays are displays that do not let the user alter their displayed value. |
| void | **setValidationEnabled**(boolean mode)<br>Displays usually have special behavior in find mode - most of the times disabling validation & becoming editable.<br>This is called when entering/exiting find mode. |
| boolean | **stopUIEditing**(boolean looseFocus)<br>The method gets called when Servoy wants the contents of an display to be committed to it's model and make sure it is not editing content (can be record change, active form change, save, ...). |

Seems pretty straight-forward, and at least we have more then one line of comment this time:

- *isEnabled*() will need to return true if the component is enabled
- *isReadOnly*() will return true id the user can't modify the value
- *setValidationEnabled*() will be called by Servoy to ask our component to go into "find" mode
- *stopUIEditing()* will be called by Servoy each time it needs to read the value, we will use that later, since our first implementation of the slider will not be used to modify record values, but to navigate a foundset, there will be another version that will be used as an input component…

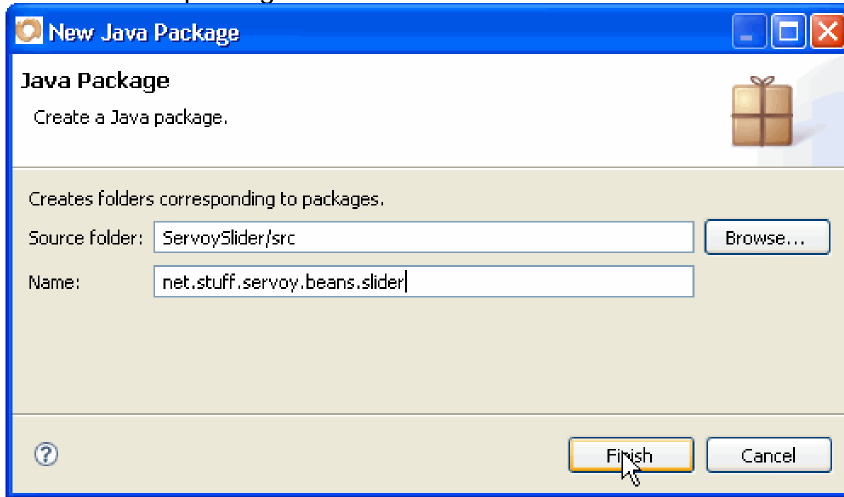
OK, we know what interface to implement, and we know that our component is basically a wrapper around a JSlider, so the easiest way to do that is to build a subclass of JSlider which will implement IServoyAwareBean, meaning our class signature will be:

```
public class ServoySlider extends JSlider implements IServoyAwareBean
```

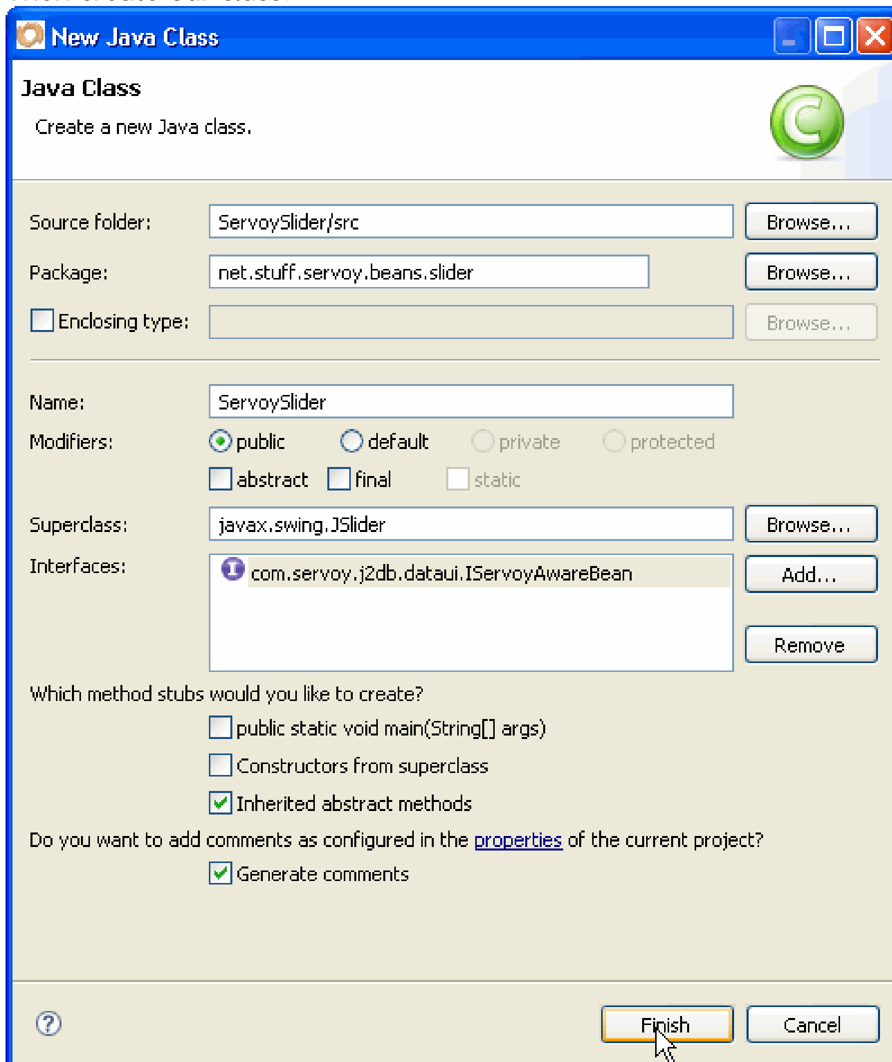
Let's create it!

# H. The ServoySlider class

First create a package to hold our class:



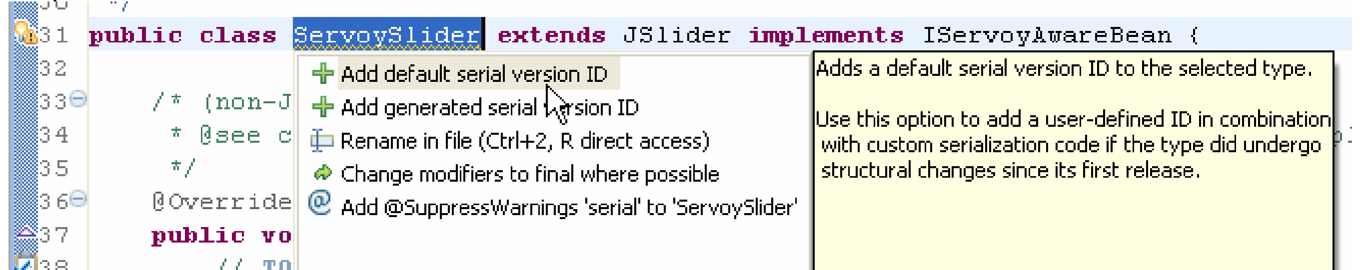

Then create our class:



We don't need to override all the constructors from the JSlider superclass, since a bean will always be instantiated by Servoy using the constructors with no parameters.

We will override the empty constructor later.

Eclipse shows a warning, because the JSlider being a JComponent is Serializable, which means that we also need to allow serialization of our subclass. An easy way to deal with that is to use the quick fix of Eclipse:




I will not go into the details of what's best between using a default private final static long serialVersionUID (1L) or to generate a serial version ID, just know that it is a deep field that is quite controversial in Java circles, you can google about it, know that here all we need to set is a simple default to allow serialization of our bean, so choose "Add default serial version ID" from the quick fix menu and save our class, the warning will disappear.

So, first let's get rid of the most obvious methods:

First the initialize is not what we are going to use, and we don't need access to the client for now:

```
@Override
public void initialize(IClientPluginAccess app) {
	// ignore for now
}
```

Our bean is not readOnly (it will allow users to change some value) so we return false here:

```
@Override
public boolean isReadOnly() {
	return false;
}
```

Here we store the value of the validationEnabled flag which will tell us if we are in find mode, we also set the slider to an enable state according to the mode:

```
@Override
public void setValidationEnabled(boolean paramBoolean) {
	this.validationEnabled = paramBoolean;
	setEnabled(paramBoolean);
}
```

And finally we return true to calls by Servoy to stopUIEditing, we don't need to implement anything here, just return that we are done (see the comment of this method in the API for more info):

```
@Override
public boolean stopUIEditing(boolean paramBoolean) {
	return true;
}
```

We already talked about the @Override annotation which basically tells the compiler that we know that we are overriding a method from a superclass or implementing an interface. This is Java 1.5+ but this

is fine since our bean will only be compatible with Servoy 4.1.x which has a requirement of Java 1.5 anyway.

Since we will be receiving a few values in parameters from the different methods here, we need to add variables to hold them, so add:

```
protected IFoundSet currentFoundset;
protected boolean validationEnabled;
```

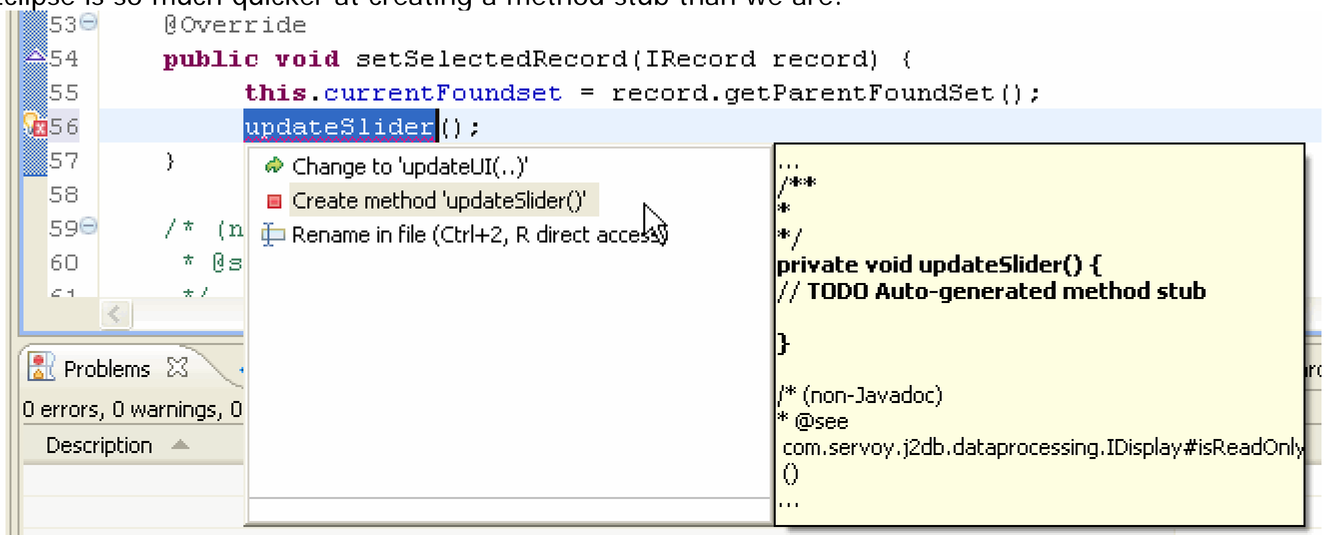
Why store a reference to an IFoundSet and not to an IRecord?
Well, right now in this version of the bean, we only want to create a navigation slider, not slider that will update values in records, in another tutorial, we will probably change that… This is called **refactoring** BTW, and we will see how good Eclipse is at that ;-)

We set these properties as protected because we don't want to have to create getters and setters for them but we want our component to allow subclasses access to these variables. If needed, we will add them later…

The real interesting method is the setSelectedRecord() and we are going to use it to update our "currentFoundset" property as well as updating the JSlider properties, so let's type the following:

```
public void setSelectedRecord(IRecord record) {
	if (record != null) {
		this.currentFoundset = record.getParentFoundSet();
		updateSlider();
	}
}
```

Easy enough, we get a hold of the parentFoundset of our record (if it's not null), but what about the updateSlider() method, we haven't created it yet so Eclipse is complaining that there is an error!
This is actually a trick to ask Eclipse to work for us: we'll use the quick fix to "fix that error", since Eclipse is so much quicker at creating a method stub than we are:




There we have our method stub typed for us:

```
private void updateSlider() {
	// TODO Auto-generated method stub
}
```

Remember what we did to set the slider values (see the ACTION_setSliderValues() in the script of our bean test)? We are going to implement roughly the same thing here, but this time in Java:

```
/**
 * Updates the slider from the values of the current Foundset
 */
private void updateSlider() {
      if (validationEnabled && currentFoundset != null) {
            setMinimum(1);
            int max = currentFoundset.getSize();
            setMaximum(max);
            int tickSpacing = Math.max(Math.round(max/10),1);
            setMajorTickSpacing(tickSpacing);
            setValue(currentFoundset.getSelectedIndex()+1);
      } else {
            setMinimum(0);
            setValue(0);
      }
}
```

Of course, we only update the slider if we are not in find mode and if it's not null; otherwise we set its value to 0;
**Note** that we set the value to the foundset's selectedIndex +1, because the foundset is 0 based.

This is all well and good, but what about changing the selected index of the foundset when the user interacts with the slider?

To do this, we will need to "listen" to the change of the value of the slider, and the easiest way to do that is to implement a listener into our own class… so let's add a ChangeListener interface to our class:

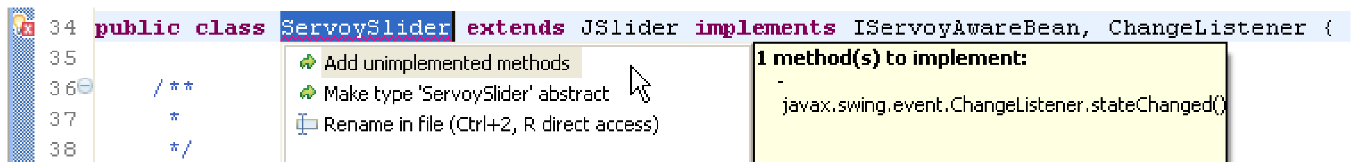
```
public class ServoySlider extends JSlider implements IServoyAwareBean,
ChangeListener{
```

You can see that Eclipse first complains that it doesn't know about the ChangeListener interface, so with our cursor at the end of the word, let's type ctrl + space (cmd + space for Mac addicts ☺).

Eclipse will add the following import to the top of our class:

```
import javax.swing.event.ChangeListener;
```

Fine! But Eclipse is still complaining, this time because we declare that our class implements the ChangeListener interface while it is not fulfilling its contract! Easy again, let's use "quick fix" to add the stub method:



Now Eclipse has added the stateChanged() method to our class, and all we need in that method is to update the index of our foundset like that:

```
@Override
public void stateChanged(ChangeEvent e) {
	if (currentFoundset != null) {
		currentFoundset.setSelectedIndex(getValue()-1);
	}
}
```

The ChangeEvent passed to us by this method have only a method to get the source of the event, but it happens that the source will always be our own class. So we can ignore the event altogether and set the foundset selectedIndex value to the value of our slider (-1 since the selectedIndex of the foundset is 0 based).

Once we got the int value, we update our currentFoundset's index to it (with a safety test checking that it is not null).

That's the main method that will take care of all the interactions our users will have with the bean. But we still need to set our class as a listener to the change in values. The easiest is to set it into a no parameter Constructor (the one that will be called by Servoy to instantiate our bean).

This is our constructor:

```
public ServoySlider() {
	super();
	addPropertyChangeListener(this);
}
```

Now it seems that we are all done with our implementation, isn't it?
Well almost... I don't know if some of you noticed but right now there is a problem to our implementation... For those of you who did, bravo! For those who didn't, don't worry, it will come with time (and a few Exceptions ;-)

Truly, I wouldn't recommend using the code as is...
Why?

Well, look closely at our updateSlider() method: This one will be fired by Servoy each time the record index changes, by a call to the setSelectedRecord() method.
But in the updateSlider() method we are calling setValue(), which in turns will fire the stateChanged() method, which will update the index of our foundset...
Meaning that Servoy might call again the setSelectedRecord() method and we will end up with an endless loop, not the kind of situation your user will enjoy!

So let's change our class of bit to handle this situation, in 3 simple steps:

First we will add a boolean to assert that we need to update our foundset index or not (depending whether the event comes form Servoy changing the value, or from the slider itself), so we add this boolean:

```
protected boolean ignoreUpdate = true;
```

We set it to true because the first time the stateChanged() method will be called, it will be done by Servoy by way of the bean properties, not by the setSelectedRecord() method.

Then we add a method to avoid using setValue() directly and trigger the stateChanged() ourselves, by setting our "ignoreUpdate" to true before setting the value:

```
	private void setSliderValue(int x) {
		ignoreUpdate = true;
		setValue(x);
	}
```

Then the last thing we need to do is to update our stateChanged() method to use this flag:

```
	@Override
	public void stateChanged(ChangeEvent e) {
		if (!ignoreUpdate) {
			if (currentFoundset != null) {
				currentFoundset.setSelectedIndex(getValue()-1);
			}
		}
		ignoreUpdate = false;
	}
```

I think we're done coding. All we need to do now is to wrap up our code in a nice little jar and use it in Servoy.

## I. Packaging and deployment of our Servoy-Aware bean

For Servoy to recognize our bean and put it in the "Place Bean…" dialog for us to choose, it needs a special file to be present in the jar we are going to create now.

This file is called a Manifest file. Basically, it holds information for the user of the jar, declaring what's in it. There is a lot of information on your library that can be useful for the software that is going to use it, like indicating that your jar contains a main class (the one that will be used when double-clicking the jar from your file system) in case your jar is a stand-alone program, the jarsigner utility will also hold hash information about the classes inside the jar, when you sign a jar.

In our case, we only need to put one piece of information: that the jar contains a Java-Bean and which class implements it.

So we will create a MANIFEST.MF file and put it into a META-INF folder of our project (standard place to put it), this MANIFEST.MF file is a simple text file and we will put the following content in it:

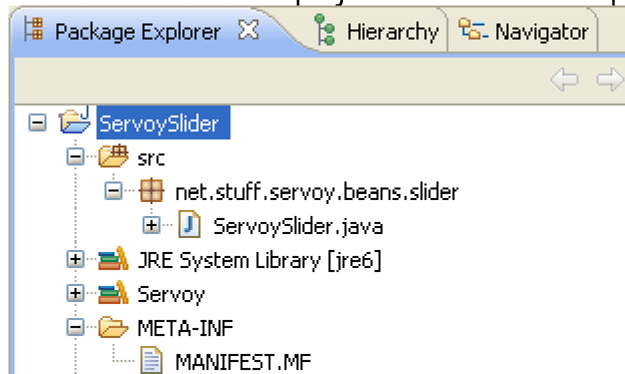
```
Manifest-Version: 1.0

Name: net/stuff/servoy/beans/slider/ServoySlider.class
Java-Bean: True
```

The first line is required, so leave it there.

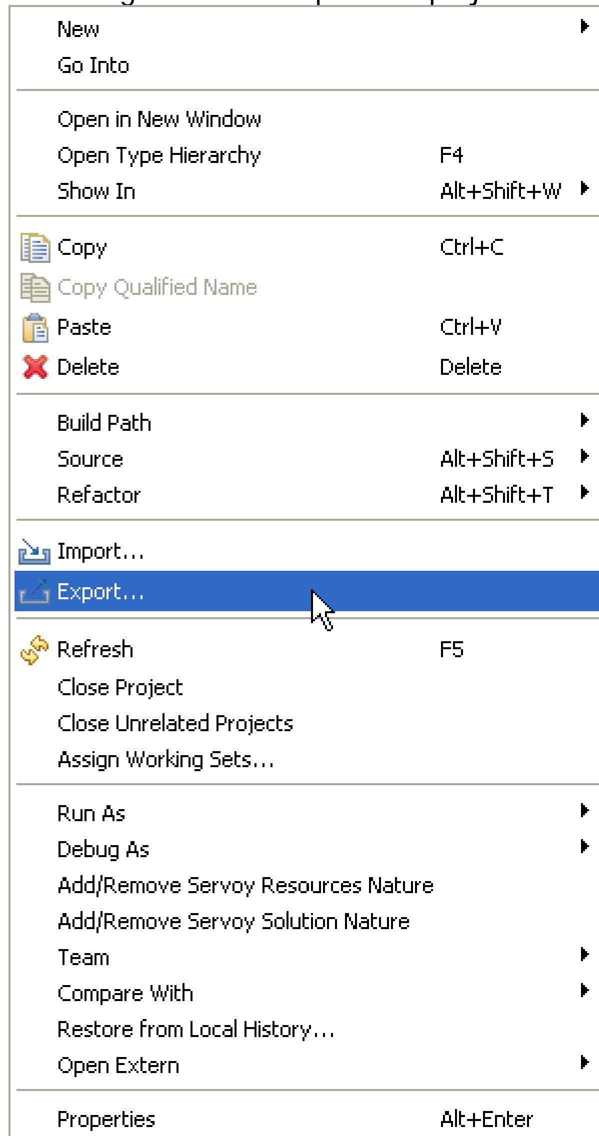
Then you have a line giving the fully qualified name of our class (with a "/" separator for the packages instead of a ".")

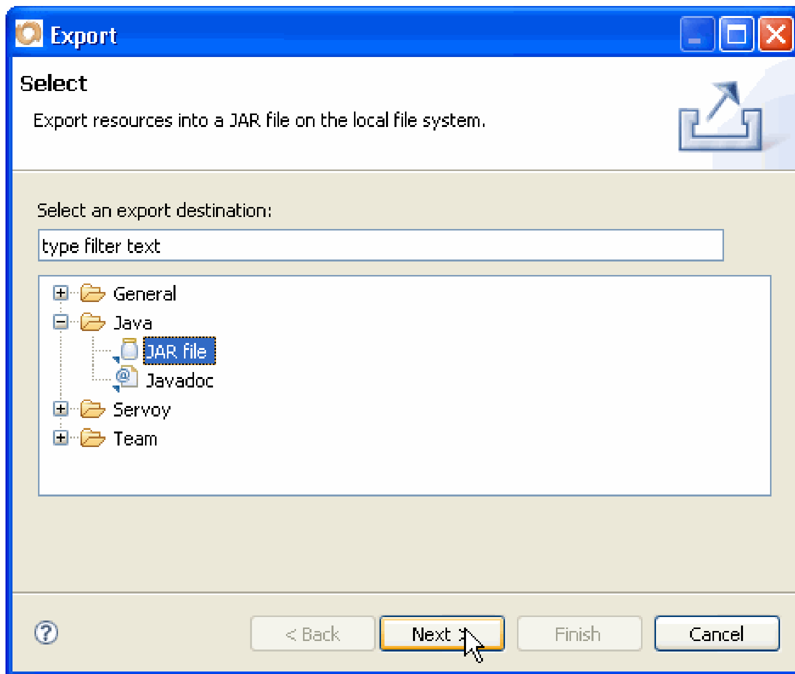
After that line we state that this class is indeed a Java-Bean.

Then you need to add a few line breaks. This is because Eclipse has a tendency to eat the mast few lines – this happened to me a few times, so now I make sure it has enough to eat ;-)
That's it for our MANIFEST.MF file.
Let's be sure that our project now has this simple structure in the Package Explorer:

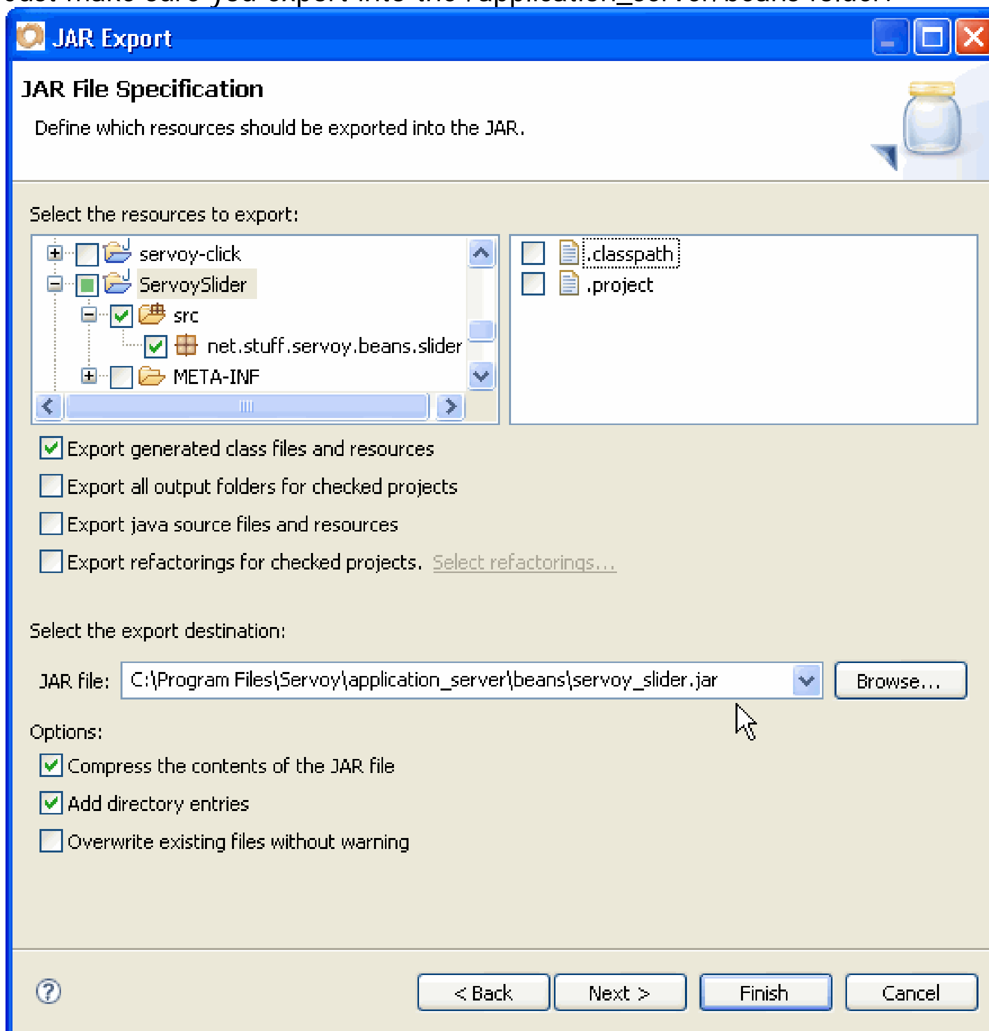



Last thing to do is to export our project as a JAR. You know the drill:

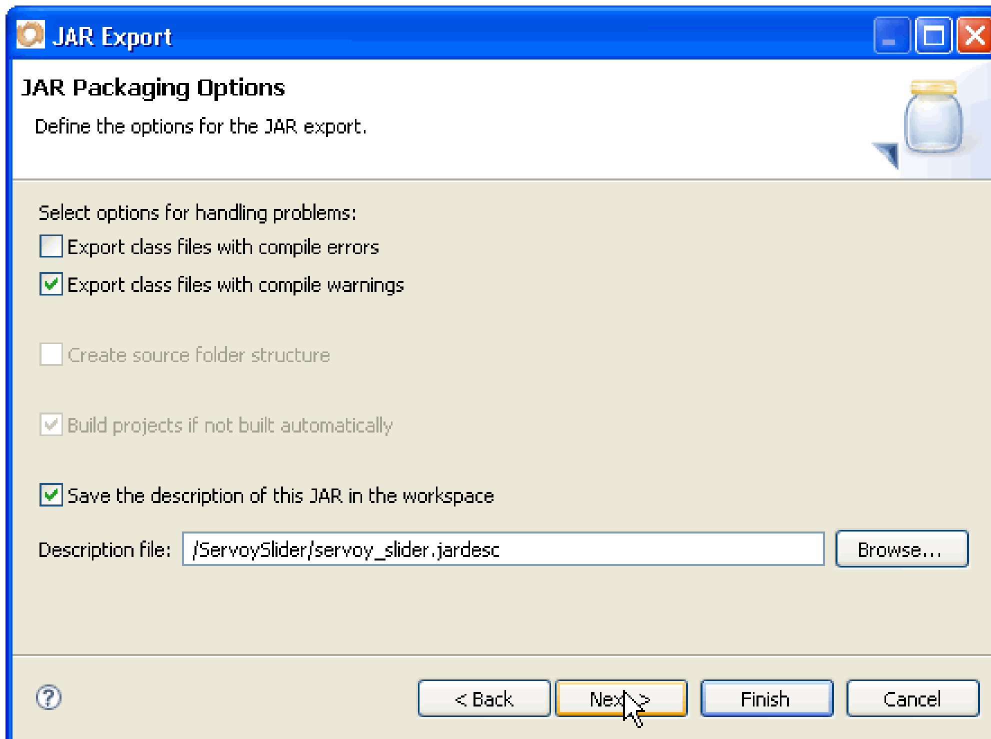




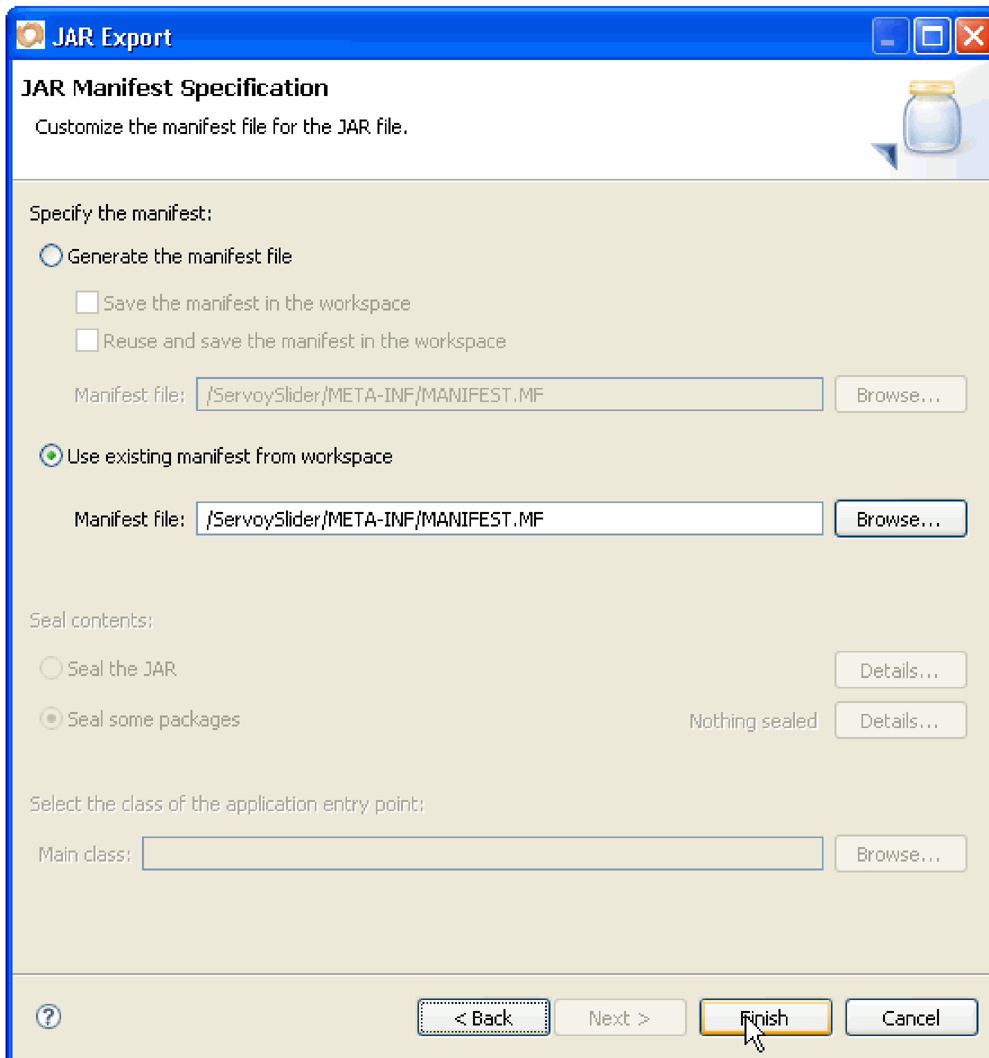

Just make sure you export into the /application_server/beans folder:



It's also a good idea to keep our jar export settings into a jardesc file, so make sure you do that:




And finally, don't forget to use check the "Use existing manifest from workspace" option, telling Eclipse to use the one from your project (normally located in /ServoySlider/META-INF/MANIFEST.MF):



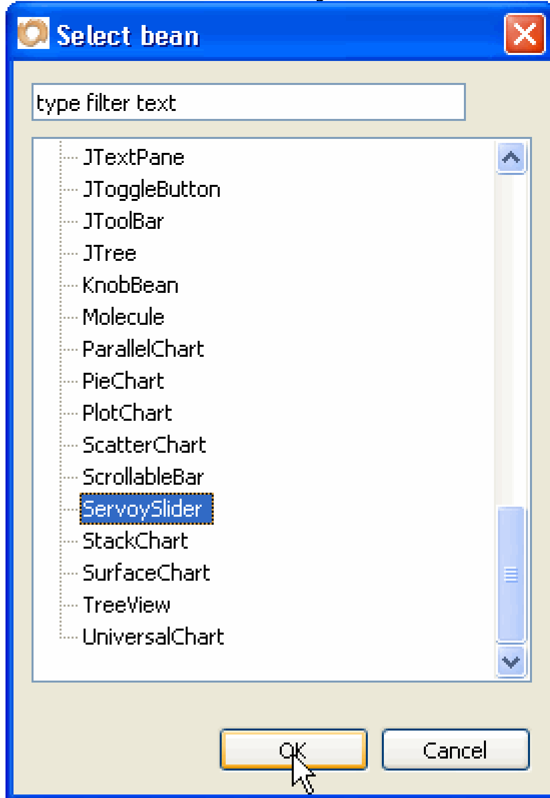

That's it. When clicking finish, you will have created and exported the bean inside your Servoy /beans folder and created a jardesc files for the next deployments.

It's time to restart Servoy to test our bean on a form

## J. Testing our Servoy-Aware bean

Back in Servoy, this time in the usual "Form Design" perspective.
Let's activate our "beans_tests" solution and open our "sliders" form in the form designer.

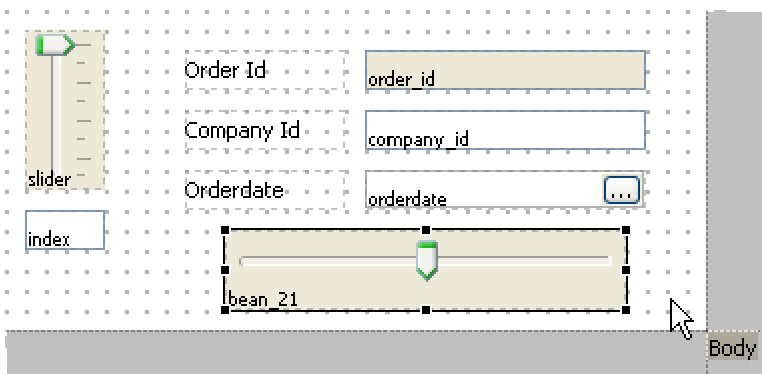
Choose “Place Bean…”, you should see our ServoySlider bean in the dialog, select it and click OK:



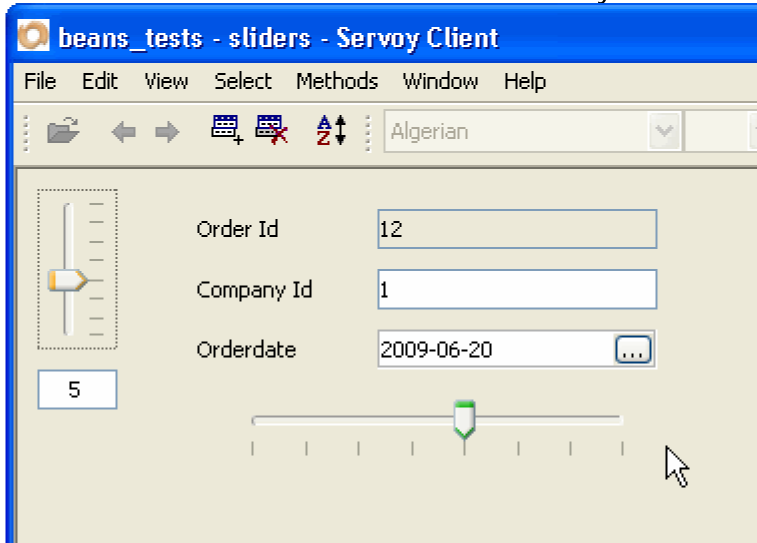

The bean is placed somewhere on the form with the usual 80x80 size :{

So let’s place it where we want it, and use the properties editor view on the bean to set its size and a few other properties:

- size: 200,40
- paintTicks: true



Now we can launch our smart client and finally see the end result of our efforts:




Total lines of code in the script: 0!
Total properties we had to set for the bean to appear like that on the form: 3 (location, size, paintTicks)!

Now that's what I call easy ;-)

And that's how our efforts in encapsulating a regular Java-Bean into a Servoy-Aware bean is starting to pay off, because from now on, each time you will want to use this bean on a form, you won't have to script a line!

As usual, you will find the complete Eclipse project on the Servoy Stuff web site, at this url:
http://www.servoy-stuff.net/tutorials/utils/t02/v1/ServoySlider_EclipseProject.zip
(Import in Eclipse)

The compiled bean (targeted for java 1.5) will be available here:
http://www.servoy-stuff.net/tutorials/utils/t02/v1/servoy_slider.jar
(Put in you /beans folder)

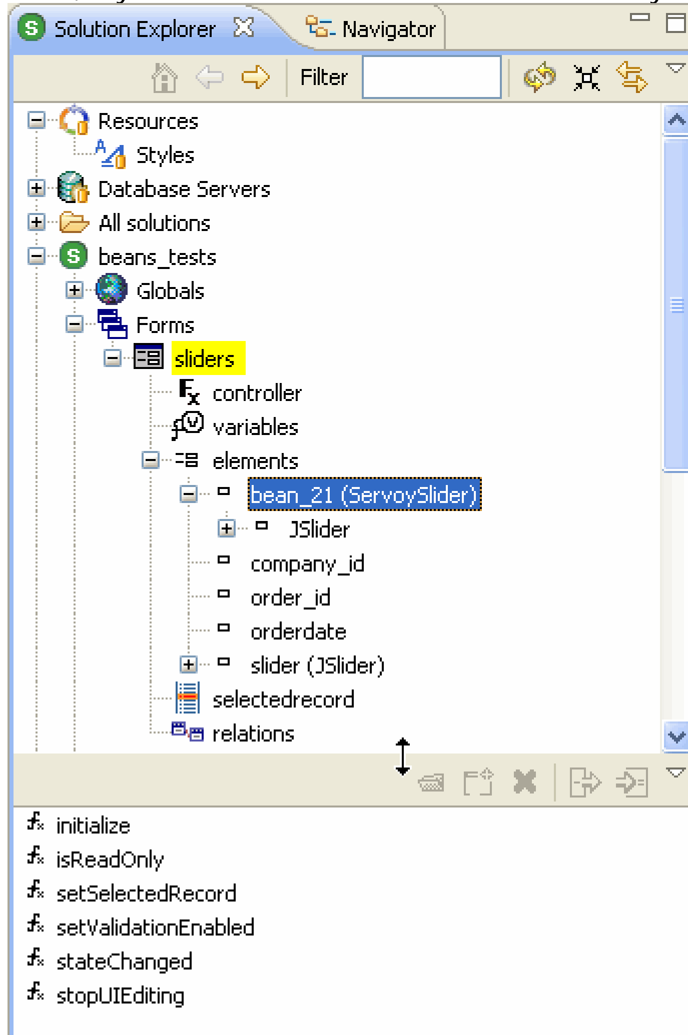
And the little "beans_tests" solution will be available at:
http://www.servoy-stuff.net/tutorials/utils/t02/v1/beans_tests-v1.zip
(Unzip and import in Servoy 4.1.x)

OK, so that's it? You really think we are going to keep our bean like that?
Nah! This is only version 1!

Of course, this bean is already quite usable as such, but it does need some polishing, don't you think?

First, as you can see from the properties editor, there are still a lot of properties that are not really useful (nor even usable), and we will first clean this up a bit…

Then, if you look at the “elements” node of Servoy’s Solution Explorer, you will see it appear like that:




This is far from being clean, since we don’t really want to show our implementation, but we would rather give some meaningful properties and methods access to our users via regular Servoy scripting. So we will also address that.

Then as I said earlier, there are other use cases for a slider: one of them would be to act as an input component. So that might well be another version of our slider.

And then of course, our bean right now is only usable in the smart client, when our goal would be to use it seamlessly on the web client as well… This is yet another big area that we will need to tackle in the next parts of this tutorial.

In the meantime, have fun with sliders all other the place ;-)
Feel free to comment, ask questions and bring on your suggestions, I always like to hear from you!

**Patrick Talbot**
Servoy Stuff
2009-07-02