

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

PART 5

S. Building a multi client input bean

In the previous parts of this tutorial, we build a Servoy Aware Swing Slider bean, we fine tuned it, and we added functionalities to make it useful, like a way to make it work on double values with a precision factor.

I wanted to go straight to how to transform our previous bean to make it Web compatible until I realized that there was too much things to see first (you can guess by the amount of pages here). I thought that being able to grasp on a simpler bean what is involved to make a multi client bean (Swing and Wicket) was a pre-requisite to transforming our bean which is already fairly complex and that (as you will see later) will need a good amount of refactoring again to make it work seamlessly with a Wicket part.

So I decided that this part will be focused on a multi client input bean and what it takes to build such a beast. We will do it on a much simpler component, for the Swing part we will subclass a JTextField and only add a minimum amount of code to it, while our Wicket implementation will be a subclass of Panel that will contain a TextField (both Wicket classes) and we will see how this goes. And we will also see how to link those two sides of the coin with an IServoyBeanFactory.

But first in Servoy we need to create a new Java project, call it "TestBean", add the required "Servoy" User library, create a package into the /src folder (I named mine "net.stuff.servoy.beans.test"). I leave you to do it yourself, you should know by now (if not, review the previous tutorials).

We also create two different packages: net.stuff.servoy.beans.test.swing for the Swing part of our bean and net.stuff.servoy.beans.test.wicket for the Wicket part of our bean.

We will start by implementing our Swing bean which is the easy part (we know exactly how to do it now, don't we?) but first we need to understand how Servoy is going to know which version of our bean to use depending on the client (smart or web).

The answer to that is inside the interface com.servoy.j2db.IServoyBeanFactory. Now if you try to look for it in the public API, you will not find it! Which I think is an error because without that interface there is no way you will be able to build a bean which will be used seamlessly in the smart client and the web client. Why isn't it here? I don't know. But I think it SHOULD be here! *

* **Update:** Johan Compagner agreed with me on the forum today (14th of July, Vive la France!) and the interface WILL be here soon! Hurray!

So the IServoyBeanFactory is the essential piece that will return Servoy the right component to use depending on the client. It does it with the use of one method:

```
public IComponent getBeanInstance(int appMode, IClientPluginAccess  
paramIClientPluginAccess, Object[] paramArrayOfObject)
```

This method is called by Servoy whenever it needs to retrieve an instance of a component for the developer environment as well as the smart client and the web client. It receives 3 parameters:

- an **int** which is the mode (see IClientPluginAccess constants to see that for the web it will be IClientPluginAccess.WEB_CLIENT = 5)
- an **IClientPluginAccess** object (a pointer to the Servoy application)
- an array of **Object** which are the parameters passed by Servoy

This method is called so that you can return an instance of the correct kind of bean (Swing or Wicket), but it is expecting an object that complies with the IComponent interface.

Now we are lucky because this *IComponent* interface is actually public, so we can have a look at the javadocs for it. It defines 24 methods that we will have to implements (most of them are getters and setters, though).

So the Swing bean and the Wicket bean to return will both have to comply with the IComponent interface. But we also know that to be able to use these as input bean they will need to implement the IServoyAwareBean as well. And we also know that to be complete we need to deal with at least one more property: the dataProviderID. So that's the ID of the interface we are going to add straight away in our main package, let's see the whole code of it:

```
public interface ITextComponent extends IComponent, IServoyAwareBean {  
  
    public void setText(String paramString);  
  
    public String getText();  
  
    public void setDataProviderID(String dataProviderID);  
  
    public String getDataProviderID();  
  
}
```

You see that I have build an interface, called it ITextComponent (we are building a text input bean this time), and I have made it an extension of both IComponent and IServoyAwareBean. Then I added accessor methods for the 2 essential properties of our beans: the text itself (a String) and the dataProviderID (another String which is the "key" of the value in our records).

Note that as usual, you will have to type ctrl+space (cmd+space on Mac) so that Eclipse will add the import in the header of your interface:

```
import com.servoy.j2db.dataui.IServoyAwareBean;  
import com.servoy.j2db.ui.IComponent;
```

Now that we have our interface ready, let's build our Swing implementation (it won't take too much since most of the properties are already part of the JTextField we are going to subclass).

T. Implementing the Swing part

Let's create a class, name it SwingTestBean and put it in the "net.stuff.servoy.beans.test.swing" package.

Its superclass will be JTextField (javax.swing.JTextField) and its interface will be the one we just created: ITextComponent, so the signature of our bean will be:

```
public class SwingTestBean extends JTextField implements ITextComponent {
```

Using quick fix will create all the relevant method stubs that need to be implemented, and as you see there isn't too much since most of them already exists in JTextField, so we are left with only:

From the IComponent interface we have:

```
public String getId() {
public void setComponentEnabled(boolean enabled) {
public void setComponentVisible(boolean visible) {
```

From the IServoyAwareBean interface we have:

```
public void initialize(IClientPluginAccess paramIClientPluginAccess) {
public boolean isReadOnly() {
public void setValidationEnabled(boolean validationEnabled) {
public boolean stopUIEditing(boolean paramBoolean) {
public void setSelectedRecord(IRecord record) {
```

From our ITextComponent interface we have:

```
public String getDataProviderID() {
public void setDataProviderID(String dataProviderID) {
```

To fulfill our implementation, we already know that we need to hold on to the current record and that we need to know the dataProviderID, so let's add these variables:

```
protected IRecord currentRecord;
private String dataProviderID;
```

Now on the code! If you look at the javadocs of the IComponent interface, there is actually one useful comment in it: it's about the getId() method, the comment says:

wicket id, normally the UUID prefixed with 'sv_'

Meaning that since this one is a Swing bean and not a Wicket bean, we can safely return null:

```
public String getId() {
    return null;
}
```

The following ones are pretty easy, they are just alias method names for 2 swing component properties, so let's implement them accordingly:

```
public void setComponentEnabled(boolean enabled) {
    setEnabled(enabled);
}
public void setComponentVisible(boolean visible) {
    setVisible(visible);
}
```

And we know about the initialize() method, but we don't really need to keep a pointer on the Servoy application (there will be no call back to JS methods in our simple bean), so we can ignore:

```
public void initialize(IClientPluginAccess paramIClientPluginAccess) {  
    // ignore  
}
```

For the isReadOnly() method we will also keep it simple and call the JTextField isEnabled() method:

```
public boolean isReadOnly() {  
    return !isEnabled();  
}
```

Then we can also ignore the setValidationEnabled() method call, since our bean is going to allow any kind of text – there will be no validation anyway:

```
public void setValidationEnabled(boolean validationEnabled) {  
    // ignore  
}
```

Finally the stopUIEditing() method is called each time Servoy wants to retrieve the data, and if it can, the method must return true (remember?), so let's do it simply like that:

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

And then we have our dataProviderID getter and setter, so let's implement them also:

```
public String getDataProviderID() {  
    return dataProviderID;  
}  
public void setDataProviderID(String dataProviderID) {  
    this.dataProviderID = dataProviderID;  
}
```

Now we are only left with the setSelectedRecord() method. You remember from our ServoySliderBean that it contained a few tests:

```
if (currentFoundset != null &&  
    currentFoundset.getSize() > 0 && currentRecord != null) {  
  
    if (hasDataProvider() && isContainedInFoundset(getDataProviderID())) {
```

Now we know that we are going to need the same kind of code for the Wicket bean, so it would be a good idea if we could put these tests into one single method that would be accessible from both class (the Swing bean and the Wicket bean).

That's the idea for a new "helper" class. So let's create a class that will have this method and add it to our package. I named mine "ServoyUtils" in the "net.stuff.servoy.beans.test" package. Now the best thing would be if that utility class could be called without the need to instantiate it (no "new" object created to access its methods), so we are going to put some static methods in it.

But wait! What about concurrency, what about threading? The thing is that you can use static methods safely in a multi-thread environment, but the only condition is that they will only use external objects (given as parameters) and not modify them! So that's fine in our case because we want a utility method which is going to read from the objects passed in parameters, let's code the following:

```

public static boolean isContainedInFoundset(IRecord record, String dataProviderID)
{
    if (record != null && isEmpty(dataProviderID)) {
        IFoundSet foundset = record.getParentFoundset();
        boolean isContained = false;
        String[] providers = foundset.getDataProviderNames(IFoundSet.COLUMNS);
        if (providers != null) {
            for (int i = 0; i < providers.length; i++) {
                if (dataProviderID.equals(providers[i])) {
                    isContained = true;
                    break;
                }
            }
        }
        if (isContained && foundset.getSelectedIndex() > -1) {
            return true;
        }
    }
    return false;
}

```

Note that our method is public, and note the use of the `static` keyword. Our method will be accessible from anywhere without the need to create a new `ServoyUtils()` object at all.

Note that the record and the dataProviderID are passed as parameters; these are the objects that will be used for the tests.

Note that I have used a non existing (yet) method “isEmpty” to test the dataProviderID. I just thought that this kind of test can be used in some other contexts too. We can use “quick fix” and ask Eclipse to create a stub for us and we will implement it like that:

```

public static boolean isEmpty(String s) {
    return (s != null && s.trim().length() > 0);
}

```

Now that we have our convenience method written in our `ServoyUtils` class, let’s get back to our `setSelectedRecord()` method that we can now write like this:

```

public void setSelectedRecord(IRecord record) {
    this.currentRecord = record;
    if (ServoyUtils.isContainedInFoundset(record, getDataProviderID())) {
        Object o = currentRecord.getValue(getDataProviderID());
        setText((o==null) ? null : o.toString());
    }
}

```

First, we hold our `currentRecord`. Then we use our newly created method from the `ServoyUtils` class to do all our tests in one go.

Note the direct use of the name of the class, that’s how you call static methods in Java. The method is accessible this way from anywhere in your code and we will use it of course in our Wicket bean too. Then we retrieve the object from our `currentRecord` with the `getValue()` method. Since it is an Object, we test if it is no null first, if it isn’t we simply call its `toString()` method to retrieve the String representation of it.

We now have our bean correctly updated by Servoy each time we change records. What about the other way round?

You probably guessed that we will need to implement a listener to listen to changes of our text value to update the currentRecord.

Now the JTextField class relies on ActionListener to propagate changes on the text (you could also implement a DocumentListener, but it will fire for each letter typed in the field, which is not very efficient), so let's use our class as an ActionListener to itself; we do this by changing its signature:

```
public class SwingTestBean extends JTextField implements ITextComponent,
ActionListener {
```

Use ctrl+space (cmd+space on Mac) to add the import:

```
import java.awt.event.ActionListener;
```

Then use "quick fix" to add the stub for the actionPerformed method, and implement it like so:

```
public void actionPerformed(ActionEvent e) {
    if (ServoyUtils.isContainedInFoundset(currentRecord, getDataProviderID())) {
        if (currentRecord.startEditing()) {
            currentRecord.setValue(getDataProviderID(), getText());
        }
    }
}
```

Note the use of our ServoyUtils static method again.

Note that we always need to ask Servoy the permission to update the record, the startEditing() call will return true if we can.

Then we simply set the value for our data provider in the current record to the value of the text of our JTextField.

Is that enough for the method to be called when the text changes? NO!

We need to add our class as a listener to the JTextField.

We do so by subclassing the no parameter constructor (the only one that will be used anyway):

```
public SwingTestBean() {
    super();
    addActionListener(this);
}
```

Note the call to the superclass constructor by the use of super(); You always need to do that if you want your superclass to be properly initialized.

Now the current record will be updated each time the user does an "Action" to our field (meaning each time he hits the return/enter key in it).

But there are still 2 little problems:

1/ the method will not be called if the user type something in our field and then exit using the mouse or type a tab to change focus for example.

2/ the current record will be updated when the user hits return/enter in our field EVEN if the value didn't change... This can be annoying because the user will see that the record has been edited when there are no real changes to it (there will be a little "e" in the status bar).

1/ To address the first problem, we will rely on the focus system. We will fire the actionPerformed method each time the field loses focus. Now we could add a FocusListener to our class signature and we would have to implement 2 methods (focusGained() and focusLost()) when the only one we are

interested in is really the `focusLost()` method. So instead of implementing the `FocusListener`, we will use a `FocusAdapter` and we will instantiate this `FocusAdapter` as an anonymous inner class. Let's see how that goes (it's easier than it sounds):

```
public SwingTestBean() {  
    super();  
    addActionListener(this);  
  
    addFocusListener(new FocusAdapter() {  
  
        public void focusLost(FocusEvent e) {  
            fireActionPerformed();  
        }  
  
    });  
}
```

After the `addActionListener()`, we added a new method "`addFocusListener()`" which waits a `FocusListener` object as parameter.

So we create it on the fly, by putting the new object directly inside the parenthesis of our `addFocusListener();` method...

Our new class will be a `FocusAdapter()` (look for it in the Java API javadocs).

And then inside this new `FocusAdapter` object, we subclass one of its method, the one we are interested in: the `focusLost()` method (you can use "Override/Implements methods..." from the Eclipse "Source" menu to get the stub for it).

In the `focusLost()` method, we call the method from the `JTextField` superclass which creates the `ActionEvent` and calls the `actionPerformed()` method on all the listeners (our class will be one of them).

This way of doing is a powerful shortcut that is possible since Java 1.5.

Look for **anonymous inner classes** on the internet to know more about it, they are especially useful when used with listeners because they allow you to code the minimum amount needed to implement a function. You don't need to create subclasses or listeners classes anymore or to implement empty methods for your listeners...

OK, so our `actionPerformed()` method will now be called each time a user exits our field or types return/enter in it. But we still need to address our second problem, which might happen even more now.

2/ We need to implement a way to test if our value really changed before updating the record. We can easily do so by keeping the previous value and comparing with the text value of our `JTextField`. So let's add a variable to hold the previous value:

```
protected String previousText;
```

Then we need to update it each time the record changes, in the setSelectedRecord() method, which will now be:

```
public void setSelectedRecord(IRecord record) {
    this.currentRecord = record;
    if (ServoyUtils.isContainedInFoundset(record, getDataProviderID())) {
        Object o = currentRecord.getValue(getDataProviderID());
        setText((o==null) ? null : o.toString());
        previousText = getText();
    }
}
```

And in our actionPerformed() method we will add a test before updating the current record:

```
public void actionPerformed(ActionEvent e) {
    if (!Utils.stringSafeEquals(previousText, getText())) {
        if (ServoyUtils.isContainedInFoundset(currentRecord,
            getDataProviderID())) {
            if (currentRecord.startEditing()) {
                currentRecord.setValue(getDataProviderID(), getText());
            }
        }
    }
}
```

Note that I used a call to Utils.stringSafeEquals(), now why do I need a special method and where does that Utils class come from?

First as the name of the method implies, it is a safe test on equality.

Why safe? Because it tests if the String are null before calling equals on the objects.

As you know calling myString.equals(anotherString) in Java will fire a NullPointerException (the dreaded NPE that is the plague of every Java programmer) if myString is null.

And the first time we call this test, we know that our previousText will be null, and the text of the slider might be null as well. This is where the Utils.stringSafeEquals helps.

The Utils class fully qualified name is actually com.servoy.j2db.util.Utils.

That's right it's coming from the Servoy public API! – And yes, it is public.

You can have a look at it in the public API and you will see that this class is full of convenience methods like this one that you can use freely inside your plugins and beans and which are meant to help you! All the methods in the Utils class are static so you can use them from anywhere in your code.

So, you know that if Eclipse complains that it can't find Utils, you can use ctrl+space (cmd+space on Mac) and choose com.servoy.j2db.util.Utils and it will happily add the import in the header of your class.

Now each time our actionPerformed() method is called it will first test if the record really needs to be updated and will do so only if needed.

That's it for our SwingTestBean!

Now we also need to add a BeanInfo as you remember from last time to get our properties right (and only the one we want) in the Servoy Properties editor, this is our SwingTestBeanInfo implementation:

```
public class SwingTestBeanInfo extends SimpleBeanInfo {

    @Override
    public PropertyDescriptor[] getPropertyDescriptors()
    {
        try {
            List<PropertyDescriptor> liste = new ArrayList<PropertyDescriptor>();

            liste.add(new PropertyDescriptor("dataProviderID", ServoyTestBean.class));
            liste.add(new PropertyDescriptor("name", SwingTestBean.class));
            liste.add(new PropertyDescriptor("background", SwingTestBean.class));
            liste.add(new PropertyDescriptor("foreground", SwingTestBean.class));
            liste.add(new PropertyDescriptor("font", SwingTestBean.class));
            liste.add(new PropertyDescriptor("toolTipText", SwingTestBean.class));
            liste.add(new PropertyDescriptor("transparent", SwingTestBean.class));
            liste.add(new PropertyDescriptor("border", SwingTestBean.class));
            liste.add(new PropertyDescriptor("size", SwingTestBean.class));
            liste.add(new PropertyDescriptor("location", SwingTestBean.class));

            PropertyDescriptor apropertydescriptor[] =
                liste.toArray(new PropertyDescriptor[0]);

            return apropertydescriptor;

        } catch (Exception e) {

            Debug.error((new StringBuilder()).append(
                "SwingTestBeanInfo: unexpected exception: "
            ).append(e).toString());

        }
        return null;
    }
}
```

If you don't remember anything about the **BeanInfo** interface, the **SimpleBeanInfo** class and what the hell they are suppose to do, I would suggest going back to part 3 of this tutorial.

Because now is the time to look into our Wicket bean...

U. Creating the Wicket part of our bean

If you have never heard about Wicket, now could be a good time to have a serious look at it.

Wicket is a Java web framework, it is an Apache project.

It is Open Source under ASL, the Apache Software License (meaning basically that you can download the source and use it in any projects including commercial ones, - read the license for the legal aspects), and it is one of the upcoming Java web frameworks.

One of the great advantages of Wicket upon other Java Web frameworks is that it is Component Oriented, not MVC – read about these 2 different paradigms on the web.

Wicket offers ways of creating your own components by subclassing or aggregating existing components. Another great feature of Wicket is that it is stateful, when the http protocol itself is stateless, - check for the significance of these terms on the internet.

And Wicket makes a great use of Ajax without the need of coding it yourself.

I leave you to discover these advantages as well as a few others on the official web site:

<http://wicket.apache.org/>

Go and see the examples online here:

<http://wicket.apache.org/examples.html>

And some advanced online examples there:

<http://www.wicketstuff.org/wicket13/>

One of the core contributors of Wicket is Johan Compagner, I'm sure the name rings a bell... That's right: he is also one of the core engineers of Servoy too! Talk of coincidence?

Anyway, to better understand what is going to follow, I strongly urge you to go and have a look at Wicket, maybe install it, or at least the examples (they are available as a .war file in the distribution, so easy to deploy on any Servlet container/java application server), you will also need to refer to the API, which is available online here:

<http://wicket.apache.org/docs/wicket-1.3.2/wicket/apidocs/index.html>

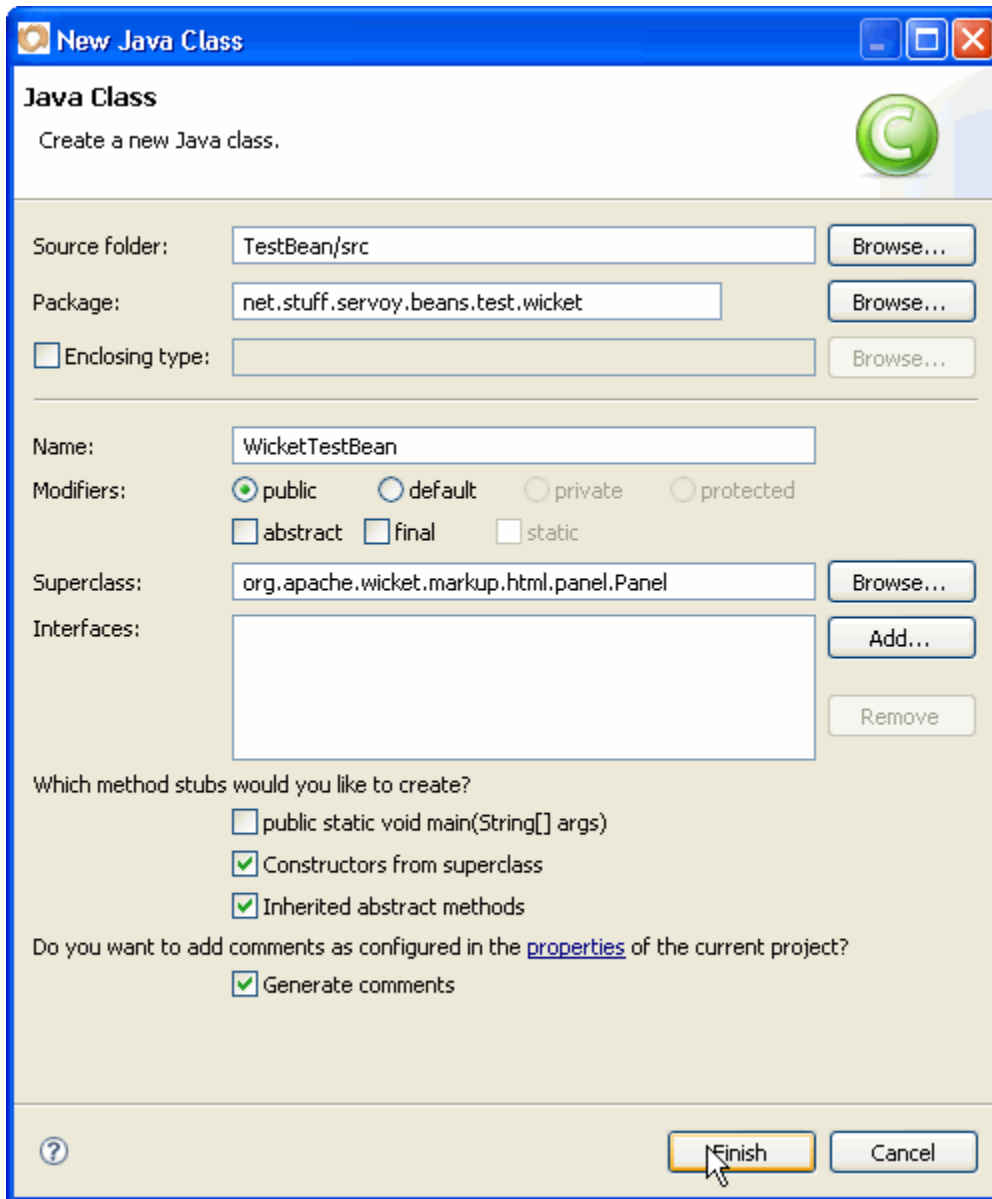
The version currently used by Servoy is 1.3.6 (slightly modified and compiled from the SVN sources according to Johan) so the online API version for 1.3 should do, you will also find the javadocs in the distribution available for download. I would suppose that the next version of Servoy (Tano?) will use Wicket 1.4 but for now there is no final version, only release candidates.

OK! Back to our bean: Inside our net.stuff.servoy.beans.test.wicket package, let's add a new class, call it WicketTestBean with a superclass of Panel (org.apache.wicket.markup.html.panel.Panel) – we will see why a Panel later...

For those who are following, I'm sure you noticed that we don't implement the ITextComponent straight away. We will do it differently this time: to make things a little bit easier for us later, we create a class that extends Panel but with no interface - we will add the interface later, so for now our class signature just looks like:

```
public class WicketTestBean extends Panel {
```

That's right! We just leave the interface blank. Why? Because we will add the properties we need first and to use the Generate Getters/Setters on our properties and then when we will add the interface, most of the implementation will already be there!



Note that we checked the "Constructors from superclass" because the Panel class doesn't have a default constructor, so we need to explicitly define at least one Constructor for our subclass. Once we click OK, the class has been constructed, a little quick fix will add the required

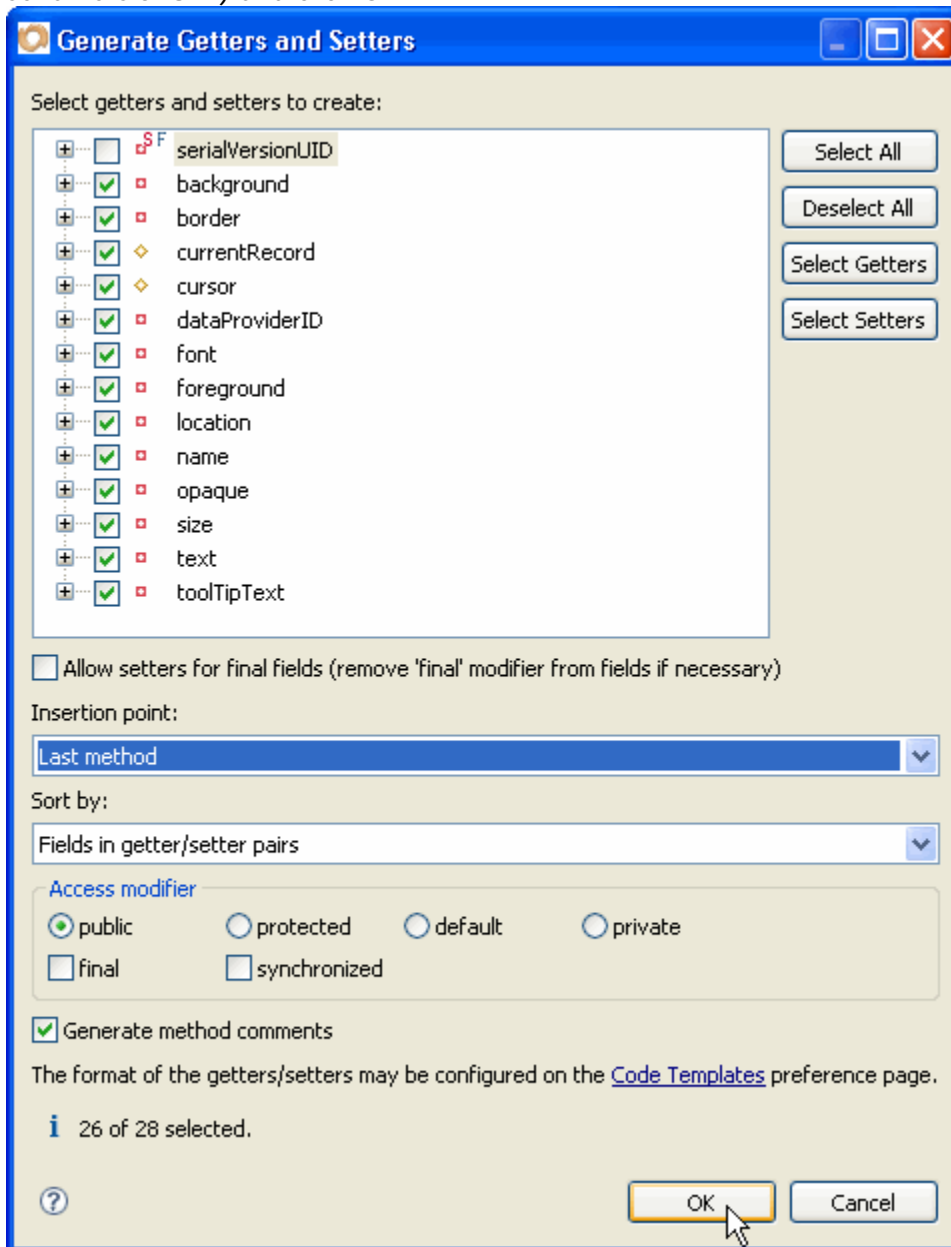
```
private static final long serialVersionUID = 1L;
```

And then, we add these properties straight away:

```
private Dimension size;
private Border border;
private Point location;
private Color background;
private Color foreground;
private Font font;
private String toolTipText;
private String name;
private boolean opaque;
private String text;
private String dataProviderID;
protected Cursor cursor;
```

All these have related methods in the ITextComponent interface that we will implement in a minute.

For now, we go to the menu "Sources > Generate Getters/Setters", select all properties (except the serialVersionUID) and click OK:



That's 26 methods generated for us in one go. That's what I call fast coding!

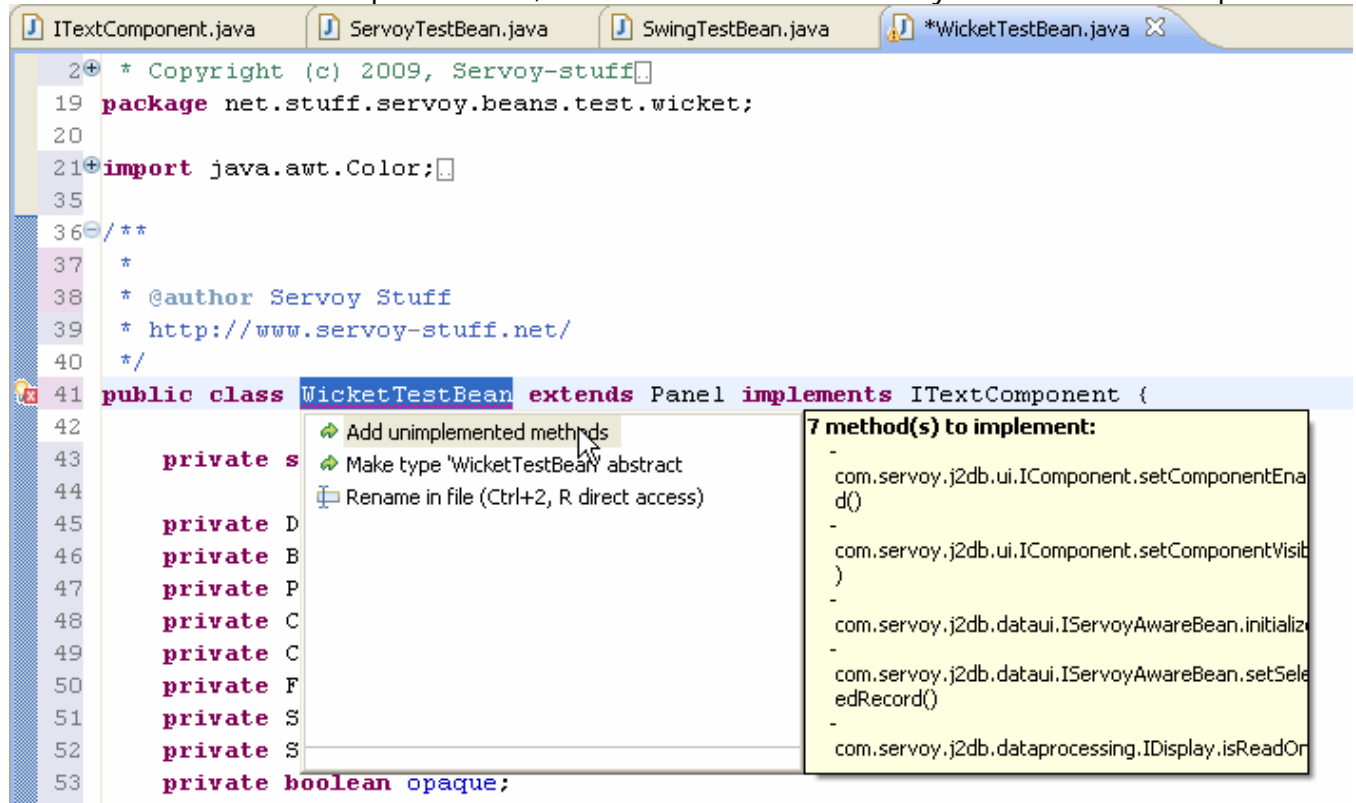
And finally we can add our interface in the class signature:

```
public class WicketTestBean extends Panel implements ITextComponent {
```

And we can also add the currentRecord IRecord variable that we know we will need later:

```
protected IRecord currentRecord;
```

And now when we use the quick fix trick, we can see that there are only 7 methods left to implement:



These 7 methods being mainly the one from the `IServoyAware` interface.

V. Building a Panel subclass

You can have a look for `Panel` in the Wicket API to see what it is about... Unlike the Servoy API, the Wicket API is full of useful comments. Oh yes please, Servoy team, let us poor developers have in Servoy the same kind of rich javadocs Wicket has!

Anyway, you will see from the Wicket API, in the header of the javadocs for `Panel` that:

“A panel is a reusable component that holds markup and other components.”

And that:

“a `Panel` has its own associated markup file and the container content is taken from that file, like:

```
<wicket:panel>
  <span wicket:id="mylabel">My label</span>
  ....
</wicket:panel> ”
```

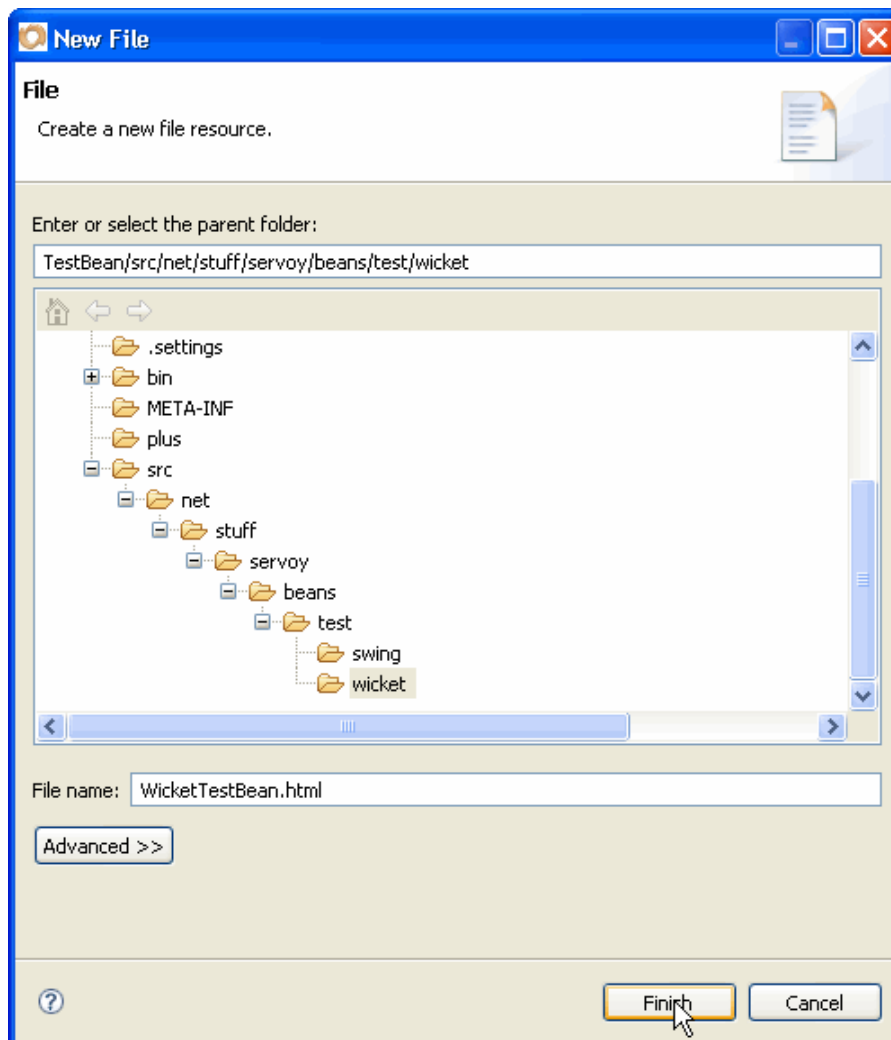
Meaning that it is a container (like a JPanel would be in Swing), that will hold any component we will put in and that it has an html file associated with it.

But why use a Panel instead of a TextField?

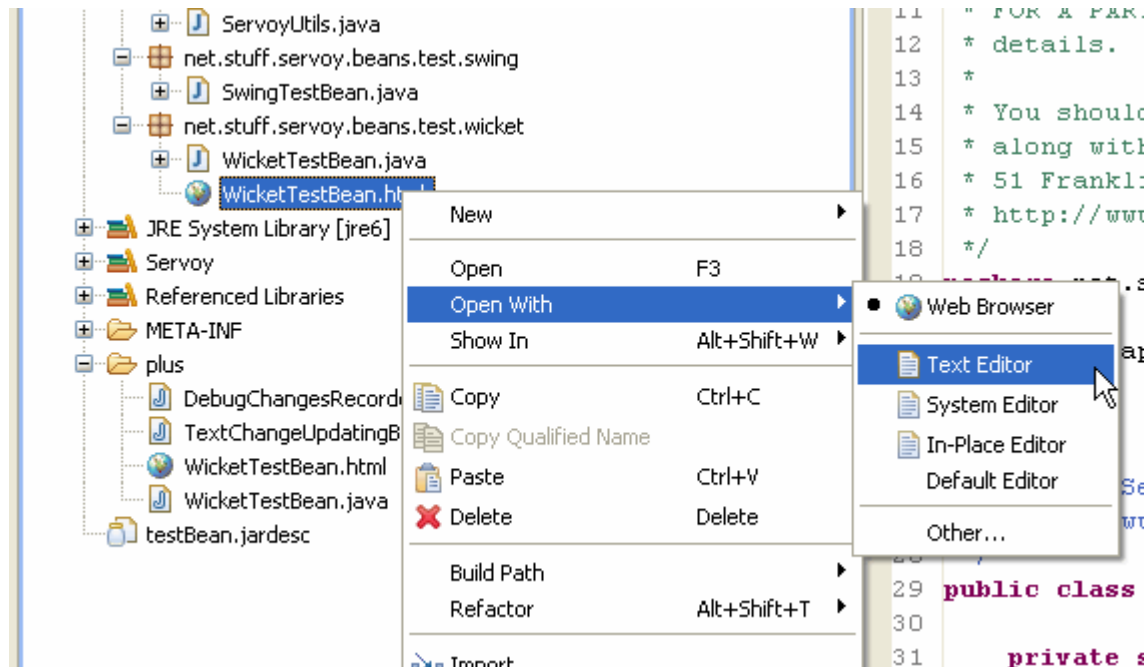
Well, I actually tried to use a TextField directly, but each time I did, Servoy complained that the markup was wrong, with an angry message and some nasty Exceptions: "input component must use an input tag but the markup contains <div>" or something like that (I actually don't remember the exact message, but you can try it yourself if you really want to see it, I prefer not to!).

It seems that Servoy, when it instantiates a Wicket bean, systematically places it inside a <div> html tag, but then Wicket complains because he is looking for an <input> html tag. So to work around this problem, the trick is to use a Panel, and then put our TextField in it, so Servoy will use the div for our Panel, and inside we will put our own markup, an <input> tag that will be the place where we will manage our TextField. Sounds complicated? A little bit, but it works fine once you do it right. Basically all we need is to add a new html file next to our java file (in the sources, inside the net.stuff.servoy.beans.test.wicket package).

Just use New File... and type its name, which should be the same as our Wicket class, with an "html" extension:



When you click "OK" Servoy has the bad idea (by default) to open the html file in the internal browser. You can close it and right click on the file to open it in a text editor:



Eclipse is clever enough to remember what was the last editor you used on a particular file so next time you will open this file it will open in the Text Editor automatically...

Put this html text inside your file:

<wicket:panel>

```
<table style='height: 100%; width: 100%; margin: 0px; padding: 0px; border-collapse: collapse;
table-layout: fixed; overflow: hidden;' >
```

```
<tr style='height: 100%; width: 100%; margin: 0px; padding: 0px' >
```

```
<td style='height: 100%; width: 100%; margin: 0px; padding: 0px' >
```

```
<input wicket:id="testField" type="text" style="border: none;
background-color: transparent; height: 100%; width: 100%; margin: 0px; padding: 0px;"/>
```

```
</td>
```

```
</tr>
```

```
</table>
```

</wicket:panel>

The spaces here are only for readability (in fact it's better if there is no space at all in all that markup, the html will be lighter and some browsers sometimes choke on carriage-return), so you can type it all on one line.

You can see that the html defines a panel inside a special tag `<wicket:panel>` and then there is a table (which is used for easier layout – I actually used the exact same html table used by the Wicket Panel used for the Calendar DateField), if I don't encapsulate my input field inside a table with these styles the layout has all sort of problems. So just trust me on this.

Or don't ;-). And if you want you can experiment with different styles and no table, but don't tell me then that I didn't warn you if your component shows a bit weird in you preferred browser...

What is really important here in any case (beside the table structure), is the fact that we define a panel `<wicket:panel>` and an input inside it with a wicket id `<input wicket:id="testField">`.

Note the id we chose (testField), because we will use it in our class when we add our TextField to the Panel programmatically.

OK Back to our bean. Let's start with the constructor; we need to add our TextField to the Panel, right? So we can do it like that:

```
public WicketTestBean(String id) {  
    super(id);  
  
    TextField field = new TextField("testField", new PropertyModel(this, "text"));  
    add(field);  
}
```

TextField is of course a Wicket class, its fully qualified name is really `org.apache.wicket.markup.html.form.TextField` (so when you use Quick Fix to add the import, choose this class).

The constructor of the TextField class we use takes 2 parameters: an id and an object of type `IModel` (`org.apache.wicket.model.IModel`), the first one is easy, we just give it the id we put in our html markup previously. The `IModel` is really important, this is how Wicket will feed the value of the input field with a real property – this is called “**binding**”.

Each input component in Wicket needs a Model to perform its binding with a **POJO** class (= Plain Old Java Object). And a `PropertyModel` is just one of the many Models you will find in the Wicket API.

You can have a look at the javadocs for the `IModel` and see all the “known implementing classes” to get a feel of how important this interface is in Wicket.

Alternatively you can find some information on the Wicket wiki, with examples (based on API 1.x but still valid), it will help you understand better than me what this is all about:

<http://cwiki.apache.org/WICKET/working-with-wicket-models.html>

And if that's not enough just Google for “Wicket model” and I'm sure you will find tons of tutorials, examples and explanations that will help you.

The `PropertyModel` we use here is particularly nice: all it needs is an object to work with and the name of one property of this object. Of course in our case the object is **this** (an instance of our `WicketTestBean` class) and the property we are interested about is the “**text**” property.

So our constructor now has declared our input component inside the Panel. But there are still a few more things to take care about, so we code this:

```
public WicketTestBean(String id) {
    super(id);

    setVersioned(false);
    setOutputMarkupId(true);

    add(StyleAttributeModifierModel.INSTANCE);

    add(TooltipAttributeModifier.INSTANCE);

    add(new SimpleAttributeModifier("class", getId()+" field"));

    TextField field = new TextField("testField", new PropertyModel(this, "text"));
    add(field);
}
```

The line `setVersioned(false)` just tells Wicket that he doesn't need to trace different version of our component (Wicket uses a mechanism to support the browser's back button). You will need to dig deep into Wicket to understand how it does it, but suffice to say here that we don't need back button support for our component and that's basically what we are saying here with this line.

The `setOutputMarkupId(true)` line that follows tells Wicket that it needs to have the id in the final html produced. This is important because some Ajax callbacks used by Servoy will use this id in the browser to update the component. – In short the html will have a tag `<div id="sv_XXXXXX">` created by Wicket (look at the source html of pages produced by Servoy and you will see lots of example of that).

Then we add 2 objects to our component: an instance of a **StyleAttributeModifierModel** (`com.servoy.j2db.server.headlessclient.dataui.StyleAttributeModifierModel`) and an instance of a **TooltipAttributeModifier** (`com.servoy.j2db.server.headlessclient.dataui.TooltipAttributeModifier`). Don't look in the public API for these because they aren't there! Which I think is a shame because they are needed to build a Wicket bean for Servoy.

I just wrote a post on Servoy's forum for some changes to the Public API, requesting that these classes be included. If you are reading this and like me would like to see (or make) more web client compatible beans for Servoy, please add your voice and support the request publicly on the forum too*:

<http://forum.servoy.com/viewtopic.php?f=15&t=12712>

* Johan Compagner replied today (July 14th) and said that he needed to check these 2 classes to see if they could be made public. I hope they will!

These 2 classes are here to keep trace of the modifications you do to the style and tooltip of your component, and they are especially needed if you use scripting to modify style or tooltip. Why aren't they in the Public API? Don't ask me, ask Servoy! We already saw that the `IServoyAwareBean` interface which is essential to building input beans is not public "yet" (it is part of the 4.2.x API, when it is already present in Servoy 4.1.x – it should be reinstated soon!) and some other interfaces and classes that we will soon encounter are not public at all, apparently... But they are used in some beans, like the `TreeViewBean` that I had the chance to get the source from Servoy.

I feel strongly that Servoy must commit to developers who want to build professional components, and for me a professional component must take into account the "wicket side" (the web client).

I hope that they will understand that and add a little bit more to the public API for people who want to build web client components and are serious about it. Without some of these interfaces and classes, it is virtually impossible to do so, when I was under the impression (drawn from the IComponent javadocs, for example) that the goal of the public API was to give developers the ability to develop third party components for the platform. – Apparently Johan Compagner is not totally against the idea, so that's good news!

After these two components added, we add a fixed attribute to our markup, using
`add(new SimpleAttributeModifier("class", getId()+" field"));`

This sets a "class" attribute to our enclosing <div> tag and set its value to the ID of our component + the key attribute "field" which is given to all input fields in Servoy (this allow for the CSS styling of all fields).

And then of course we have the code to create our TextField and add it to our component.
This is it for the code of our constructor (we will add one last thing later but for now that's enough).

The second constructor:

```
public WicketTestBean(String id, IModel model) {
```

which takes an ID and a Model is not really needed in our case since the model is taken from the "text" property of our class and not coming from another component, so we can get rid of it.

W. Implementing the Wicket bean

Now let's look at our implementation. For all the getters we are fine, because we have a variable that holds their values and that's all there is to it, but for the setters, we actually need to update our component's markup too!

To do this, we need to use yet another "not really public" class, which we will use in great length, this is the **ChangesRecorder** class (com.servoy.j2db.server.headlessclient.dataui.ChangesRecorder). The ChangesRecorder class is supposed to be modified in the next version of Servoy ("Tano", now officially called 5.0 by Johan Compagner in its answer to my request ☺), and should be public by then so we will likely have to adapt our beans but in the meantime, this class works as described here in 4.1+...

And to better use this class we will actually add it as a property to our class, like this:

```
public ChangesRecorder jsChangeRecorder =  
    new ChangesRecorder(new Insets(2,2,2,2), new Insets(0,0,0,0));
```

Now the ChangesRecorder class is the one that will hold all the style changes for our component and that will tell Wicket to update its markup accordingly. We pass it 2 Insets (java.awt.Insets) which defines the default border and padding for our component.

Once we have defined our ChangesRecorder inside a variable jsChangeRecorder, we will now use it in our setters, like so:

```
public void setSize(Dimension size) {  
    this.size = size;  
    if (this.size != null) {  
        jsChangeRecorder.setSize(size.width, size.height, this.border, null, 0);  
    }  
}
```

Note that in the absence of any javadocs and sources, you can still “see” what methods exist for a class by typing “.” after the jsChangeRecorder variable name: Eclipse will show you all the available methods and their parameters.

In our setSize method, we update our local variable first, then we test if it is not null. If it is not, we call our ChangesRecorder to set the new Size, this call will tell Wicket that the component is changed and will trigger an update of the html markup.

Now the other setters are obvious:

```
public void setBorder(Border border) {
    this.border = border;
    if (this.border != null) {
        ComponentFactoryHelper.createBorderCSSProperties(ComponentFactoryHelper.create
        BorderString(this.border), jsChangeRecorder.getChanges());
    }
}
```

You remember the ComponentFactoryHelper class (we encountered it in part 3 of this tutorial, another one that is not “officially public” yet damn useful for Wicket beans!), this class has some useful methods to get a Border from a String and in our case here to create the CSS for a border from a java.awt.Border, and it takes as parameter a ChangesRecorder object, surprise, surprise...

Next we have:

```
public void setLocation(Point location) {
    this.location = location;
    if (this.location != null) {
        jsChangeRecorder.setLocation(location.x, location.y);
    }
}
```

Using setLocation() on our ChangesRecorder.

Then:

```
public void setBackground(Color c) {
    this.background = c;
    if (c != null) {
        jsChangeRecorder.setBgcolor(PersistHelper.createColorString(c));
    }
}
```

Remember also the PersistHelper class? Another one we saw in part 3 of our tutorial, and still not “really public” but still very useful in case of Wicket beans.

The setForeground() method follows the same pattern:

```
public void setForeground(Color c) {
    this.foreground = c;
    if (c != null) {
        jsChangeRecorder.setFgcolor(PersistHelper.createColorString(c));
    }
}
```

As well as the setFont():

```
public void setFont(Font font) {
    this.font = font;
    if (font != null) {
        jsChangeRecorder.setFont(PersistHelper.createFontString(font));
    }
}
```

The setTooltip doesn't need more than the default:

```
public void setToolTipText(String tooltip) {
    this.toolTipText = tooltip;
}
```

This is because we already added a TooltipAttributeModifier instance in our constructor, remember?

```
add(TooltipAttributeModifier.INSTANCE);
```

The setName() doesn't need to trigger an update since this method will never be changed at runtime anyway, so leave it like that:

```
public void setName(String name) {
    this.name = name;
}
```

The setOpaque() method is straightforward (no need to test for null but the property is called "transparent" in the ChangesRecorder class – the opposite of opaque), so we do:

```
public void setOpaque(boolean b) {
    opaque = b;
    jsChangeRecorder.setTransparent(!b);
}
```

And the setComponentVisible() is also simple enough:

```
public void setComponentVisible(boolean visible) {
    setVisible(visible);
    jsChangeRecorder.setVisible(visible);
}
```

Now we have our setText() which is the heart of our bean, and we do it like that (you will recognize this code):

```
public void setText(String text) {
    this.text = text;
    if (ServoyUtils.isContainedInFoundset(currentRecord, getDataProviderID())) {
        if (currentRecord.startEditing()) {
            currentRecord.setValue(getDataProviderID(), text);
        }
    }
}
```

We can see the advantage of having our test method code encapsulated in a static method of a utility class, we can use it from our Wicket bean the same way we used it in our Swing bean.

But there is one thing that needs to be done: when the text is updated by the user (which is when the setText will be triggered, we need to tell Wicket to actually update the component. And to tell Servoy that it needs to "listen" to changes of our class, we can tell him to listen to "style" changes to our class, by implementing the interface **IStylePropertyChanges** (com.servoy.j2db.ui.IStylePropertyChanges). Yes, this is another interface that is not public yet is quite essential to building a Wicket component that will trigger updates back to Servoy.

So we add it to our class signature which will now be:

```
public class WicketTestBean extends Panel implements ITextComponent,
IStylePropertyChanges {
```

Eclipse will complain

1/ that it doesn't know the interface, so ctrl+space / cmd+space to add the import:

```
import com.servoy.j2db.ui.IStylePropertyChanges;
```

2/ that the methods are not implemented, so "Quick Fix" to add the 5 methods stubs needed.

Fortunately, these 5 new methods are really easy to implement, all we need is to forward the call to our jsChangeRecorder:

```
public Properties getChanges() {
    return jsChangeRecorder.getChanges();
}

public boolean isChanged() {
    return jsChangeRecorder.isChanged();
}

public void setChanged() {
    jsChangeRecorder.setChanged();
}

public void setChanges(Properties paramProperties) {
    jsChangeRecorder.setChanges(paramProperties);
}

public void setRendered() {
    jsChangeRecorder.setRendered();
}
```

OK, back to our setText() method, we need now to trigger an update in Servoy is to call our new method:

```
public void setText(String text) {
    this.text = text;
    if (ServoyUtils.isContainedInFoundset(currentRecord, getDataProviderID())) {
        if (currentRecord.startEditing()) {
            currentRecord.setValue(getDataProviderID(), text);
            setChanged();
        }
    }
}
```

And we will also use this trick to trigger updates from the setComponentEnabled() method:

```
public void setComponentEnabled(boolean enabled) {
    setEnabled(enabled);
    setChanged();
}
```

And we still have the methods from the IServoyAware bean to implement, but that is easy and we already know everything we need from previous tutorial:

```
public void initialize(IClientPluginAccess paramIClientPluginAccess) {
    // ignore
}

public void setSelectedRecord(IRecord record) {
    this.currentRecord = record;
    if (ServoyUtils.isContainedInFoundset(currentRecord, getDataProviderID())) {
        Object o = currentRecord.getValue(getDataProviderID());
    }
}
```

```

        this.text = (o==null) ? null : o.toString();
        setChanged();
    }
}

```

Note that we trigger a change here too (meaning the component needs updating).

```

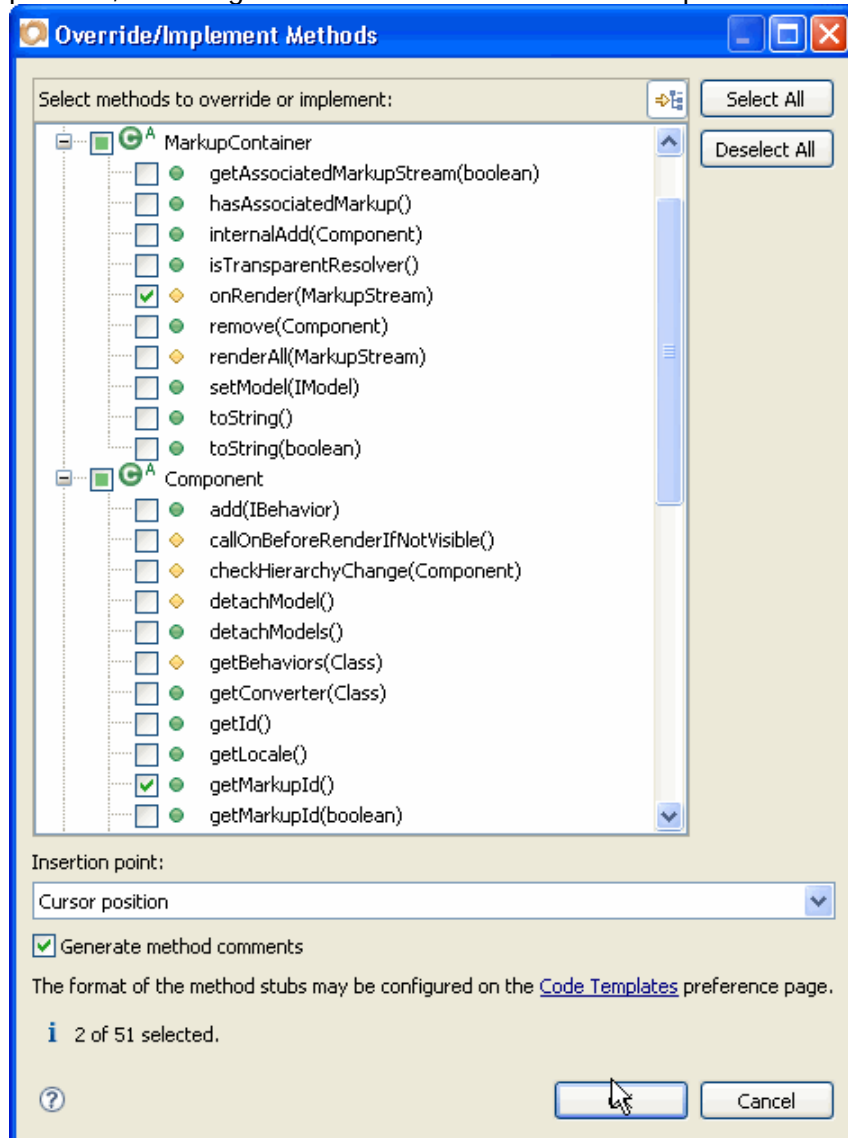
public boolean isReadOnly() {
    return !isEnabled();
}

public void setValidationEnabled(boolean validationEnabled) {
    // ignore
}

public boolean stopUIEditing(boolean paramBoolean) {
    return true;
}

```

Now look at the task panel in Eclipse, it should be empty! Is our implementation done? – Almost... We still need to override 2 more methods from Wicket superclasses to be sure that everything work as planned, so let's go to menu "Source > Override/Implement methods...", and choose:



Then implement these like so:

```
public String getMarkupId() {  
    return getId();  
}
```

This is to tell Wicket that the "markupId" that is going to be used for the <div> tag is going to be the id itself ("sv_xxxxx", where xxxxx is the UID of the component in Servoy)

The other one is:

```
protected void onRender(MarkupStream markupStream) {  
    super.onRender(markupStream);  
    setRendered();  
}
```

This is to update our local ChangesRecorder when the component is rendered and set its rendered flag to true when the component has finished rendering...

Look for more info about these two methods in the Wicket API and sources.

Yet another thing that is needed by Servoy to compute the correct placement of our bean on the html layout is for our component to implement the ISupportWebBounds (com.servoy.j2db.ui.ISupportWebBounds) - yes, I know, this one is not public either, so do like me and ask for it on the forum, we really need an updated Public API!

Our final (!) signature for our Wicket bean is going to be:

```
public class WicketTestBean extends Panel implements ITextComponent,  
IStylePropertyChanges, ISupportWebBounds {
```

You know the drill when adding a new interface: first ask Eclipse to add the correct import, then ask him to add the method stubs. There is only one method for ISupportWebBounds and we are going to code it like that:

```
public Rectangle getWebBounds() {  
  
    Dimension localDimension =  
        jsChangeRecorder.calculateWebSize(  
            size.width, size.height, border, null, 0, null  
        );  
  
    return new Rectangle(location, localDimension);  
}
```

Once again we use our faithful ChangesRecorder variable (just imagine what we would do without it, and remember that it is not public!) to calculate the size of our component in the web layout.

Is that it? I would like to say yes, but hum... we are very close.

You just remind me of these kids in a long journey in a car, asking every 5 minutes: "are we there yet?" ☺

X. Adding some Ajax behavior

One last thing we need to take care of in the Wicket bean is to make sure that our component triggers Ajax callback to Servoy whenever our value changes. Remember that we did it with an ActionListener and a FocusAdapter anonymous inner class in our Swing bean.

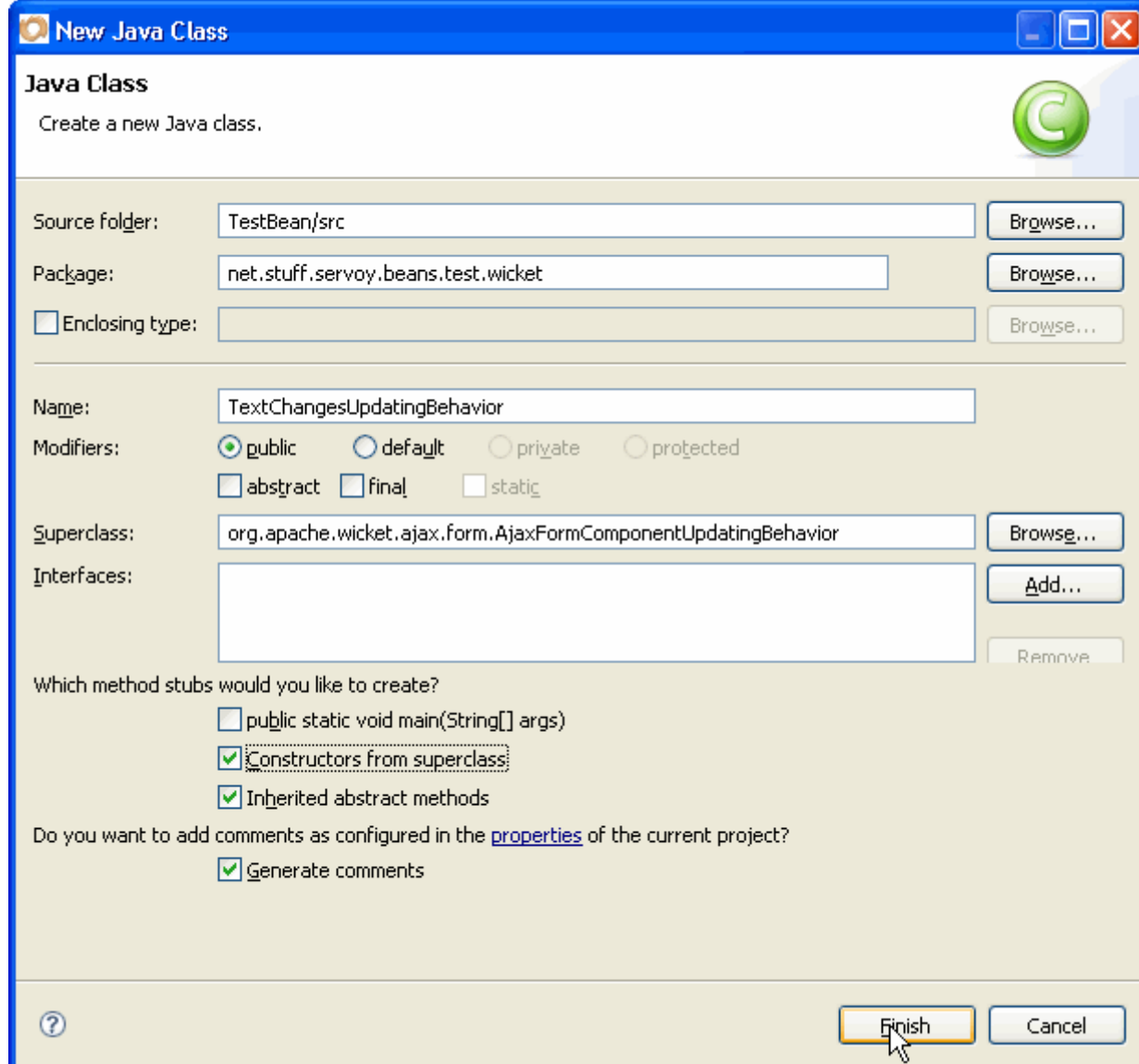
Well, here we are going to code the equivalent in Wicket.

Wicket doesn't know about java.awt listeners, but it relies heavily on "Behaviors", basically a Behavior is some code that will generate some JavaScript on the client side. There are regular behaviors but there are also AjaxBehaviors, and the one we will use is the well named

"AjaxFormComponentUpdatingBehavior"

(org.apache.wicket.ajax.form.AjaxFormComponentUpdatingBehavior to be precise).

We could create an anonymous inner class here, but since this is an important class and we might need to add some code later, we will create a proper java file for it. Let's add a new class "TextChangesUpdatingBehavior" to our "net.stuff.servoy.beans.test.wicket" package:



We checked “Constructors from superclass” because this class doesn’t have a default constructor so we will need to create one, and we will change it anyway.

It needs to be Serializable so we add the required (Quick Fix to the rescue!):

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

And we need to get access to our parent class (the WicketTestBean, so let’s add a variable to it):

```
protected WicketTestBean parent;
```

Now we can alter the constructor this way to store a reference to our parent:

```
public TextChangesUpdatingBehavior(String event, WicketTestBean parent) {  
    super(event);  
    this.parent = parent;  
}
```

Note the call to the constructor’s superclass to let it initialize correctly.

And then we need to define what event we are going to use. These are regular browser’s JavaScript event, so in our case that’s the “onchange” event (equivalent to the ActionEvent of our ActionListener in Swing) and the “onblur” event (equivalent to the “onFocusLost()” of our FocusListener in Swing).

To make things a little bit tidier, we can add 2 static values to hold our event Strings:

```
public final static String DATA_CHANGE = "onchange";  
public final static String LOST_FOCUS = "onblur";
```

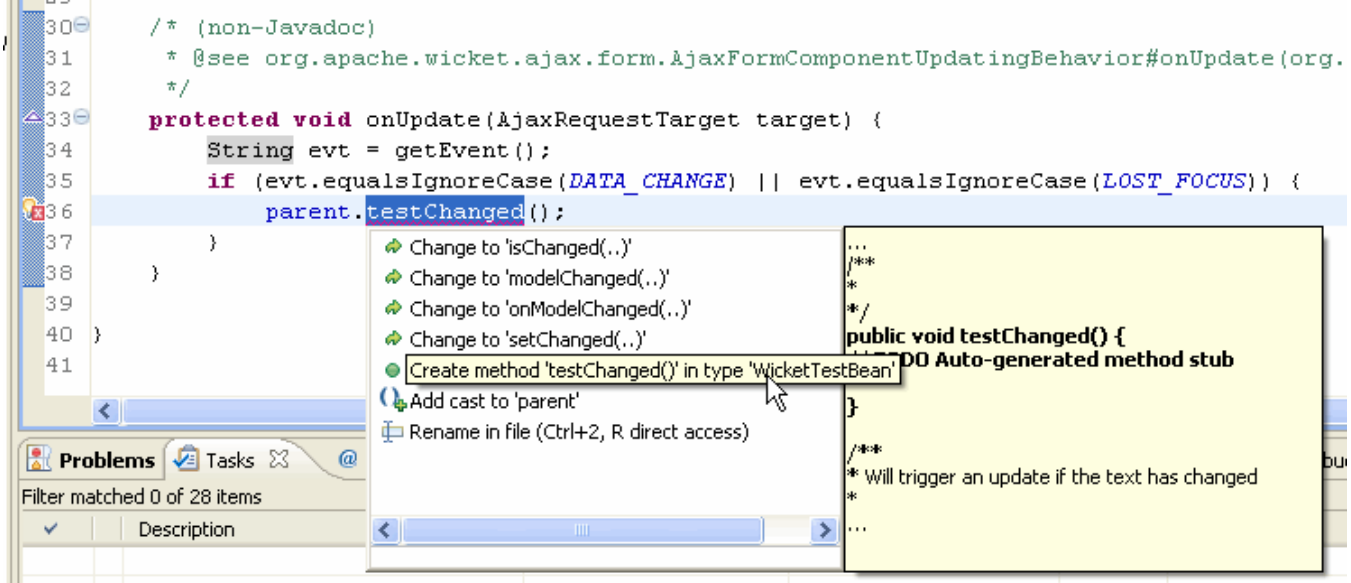
And now we can implement the only important method of our AjaxFormComponentUpdatingBehavior subclass the “onUpdate” method. This one will be called by Wicket to allow us to trigger some code on the event we want to “listen” to:

```
protected void onUpdate(AjaxRequestTarget target) {  
    String evt = getEvent();  
    if (evt.equalsIgnoreCase(DATA_CHANGE) || evt.equalsIgnoreCase(LOST_FOCUS)) {  
        parent.testChanged();  
    }  
}
```

Wicket will pass an AjaxRequestTarget to our method (we don’t really need it here anyway – but you can look for it in the Wicket API to know more...). What we do need is to check the kind of event, so we retrieve the current event from the superclass, it’s a String, and we check it against our static Strings.

If the event is one that we are interested in, we ask our parent to test if it’s changed...

The testChanged() method doesn't exist yet in our class, but there is little Eclipse won't do for you if you ask it nicely with a "Quick Fix" trick:



This will add the method stub directly in our WicketTestBean class (even if the java file is not opened)!

And now, do you remember what we did in our Swing bean before we updated our record? That's right, we checked if an update was needed by comparing the value of the new text to the previous version.

Which means we need to add a previousText variable to our class:

```
protected String previousText;
```

And then update its value when Servoy sets a selected record:

```
public void setSelectedRecord(IRecord record) {
    this.currentRecord = record;
    if (ServoyUtils.isContainedInFoundset(currentRecord, getDataProviderID())) {
        Object o = currentRecord.getValue(getDataProviderID());
        this.text = (o==null) ? null : o.toString();
        this.previousText = getText();
        setChanged();
    }
}
```

Now that our previousText value is coherent, we can use it to test if the new value is really « new » :

```
/**
 * Will trigger an update if the text has changed:
 */
public void testChanged() {
    if (!Utils.stringSafeEquals(previousText, getText())) {
        setChanged();
    }
}
```

You remember the Utils.stringSafeEquals we used in our Swing bean. Same thing here, and if it is needed we set our ChangesRecorder to "changed".

Now everything is finally in place for our AjaxBehavior, but we need to add it to our field, so we do it in our constructor which is (in its final incarnation):

```
public WicketTestBean(String id) {
    super(id);

    add(StyleAttributeModifierModel.INSTANCE);

    add(TooltipAttributeModifier.INSTANCE);

    setVersioned(false);
    setOutputMarkupId(true);

    TextField field = new TextField("testField", new PropertyModel(this, "text"));
    add(field);

    field.add(new
    TextChangesUpdatingBehavior(TextChangesUpdatingBehavior.DATA_CHANGE, this));

    field.add(new
    TextChangesUpdatingBehavior(TextChangesUpdatingBehavior.LOST_FOCUS, this));
}
```

Dora would say it too: we did it! (Sorry for the reference, my daughter is a big fan)
We've got our Wicket bean all set!

So you would think that all you need to do now is test this in your solutions...

But not so fast! Right now we have 2 different implementations of the same kind of code, one as a Swing component for the smart client (and the developer environment, actually), and one in Wicket for the web client. But yet nothing tells Servoy that they are the two sides of the same coin!

And that's where we will need the last piece of our puzzle, impersonated by yet another class, our ServoyTestBean class which will tie everything together nicely.

Y. Assembling all the pieces of our bean

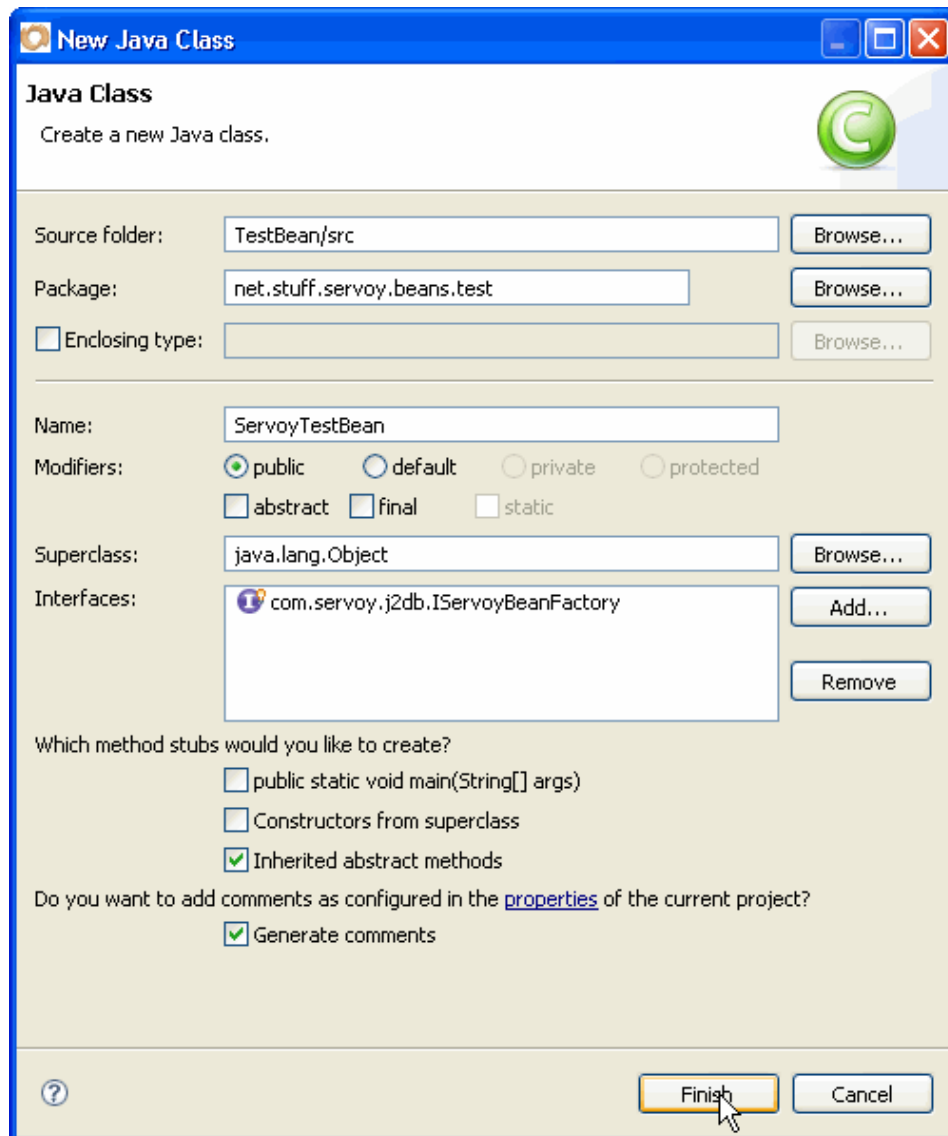
The key element is the use of yet another “not so public” interface, and one that you cannot do without to build smart/web client beans: that is the IServoyBeanFactory interface (com.servoy.j2db.IServoyBeanFactory).

Once again I urge you to petition with me to update the Public API javadocs to contain this interface. Without it, there is simply no Wicket beans, no web client beans possible!

So go to the forum, check this post: <http://forum.servoy.com/viewtopic.php?f=15&t=12712>

And please add your support to my request; I think it is more than reasonable, it is imperative! And an anonymous case in the support system, easily overlooked will not be enough. * Johan Compagner agreed to add it today (July 14th)

Back to our IServoyBeanInterface, this one only have two methods, so I guess we are lucky, not too much code to add here... We create a new simple class, call it “ServoyTestBean” inside our “net.stuff.servoy.beans.test” package, it must implement the IServoyBeanInterface, like so:



You will see the 2 methods stubs as defined by the IServoyBeanFactory interface and its ancestor.

The first method:

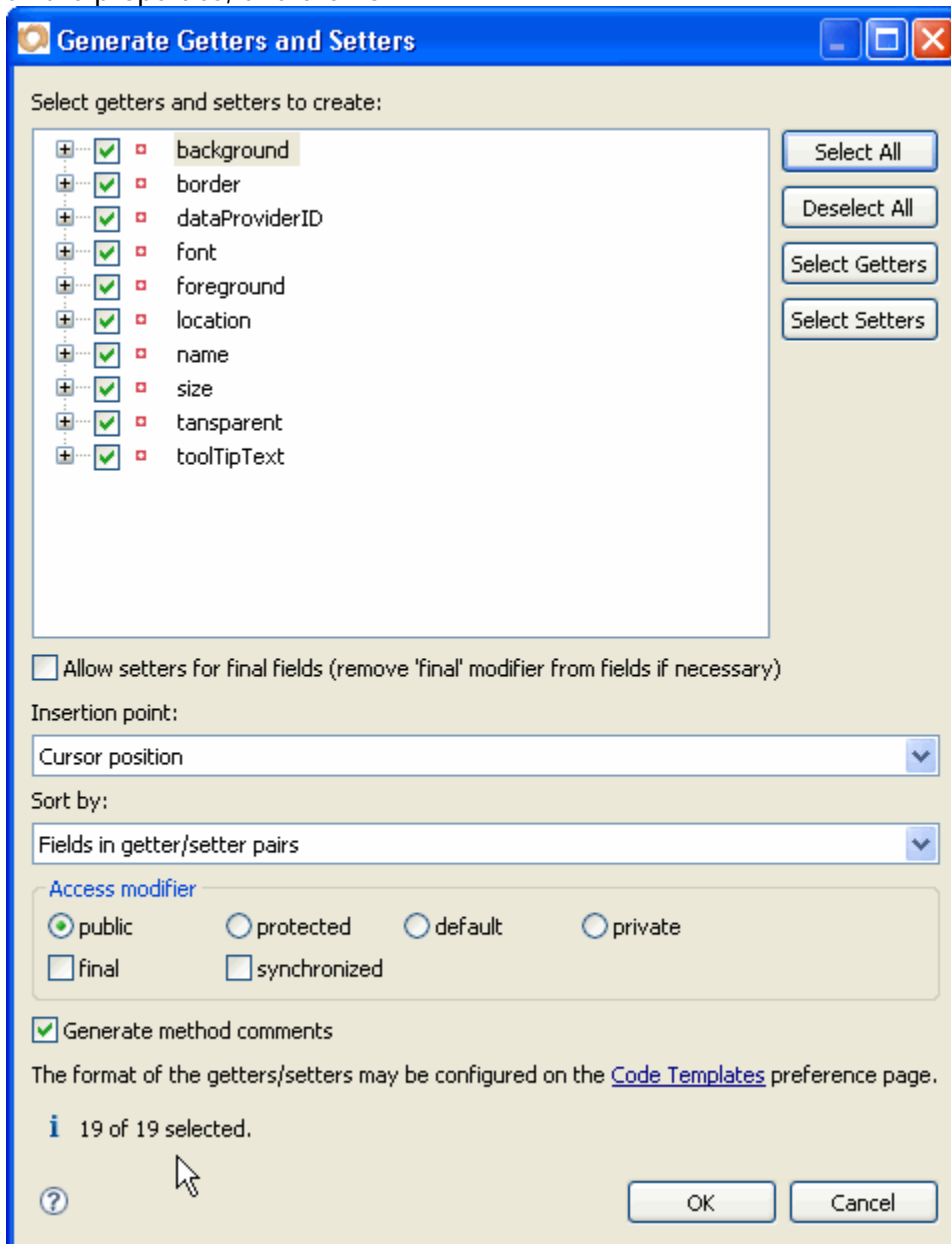
```
public IComponent getBeanInstance(int appMode, IClientPluginAccess  
paramIClientPluginAccess, Object[] paramArrayOfObject) {
```

This is the one that will do all the work, and we will see how to implement it in a minute...

But first, let's add all the properties our beans work with:

```
private Dimension size;  
private Point location;  
private String name;  
private Color background;  
private Color foreground;  
private Border border;  
private Font font;  
private String toolTipText;  
private boolean transparent;  
private String dataProviderID;
```

Then we generate our getters and setters. Menu "Source > Generate Getters and Setters...", we select all the properties, and click OK:



Note that of the 10 properties, Eclipse will generate 19 methods, that's because a getName() method has already been generated from our interface.

We also know that we are going to forward all the setXXX properties to the "real" component (the Swing bean or the Wicket bean), so to do it easily, we add a variable to hold it:

```
protected ITextComponent component = new SwingTestBean();
```

Note that:

- 1/ the variable is protected (we will not need to have a getter and a setter for it)
- 2/ the type of the variable is actually the interface ITextComponent, because it will then be capable of holding either a SwingTestBean or a WicketTestBean: again that's one great advantage of interfaces.
- 3/ we instantiated the variable with an instance of our SwingTestBean. Now why is that? Well, the thing is that when Servoy will first call the bean, it will be done from the "Place Bean..." command

(menu or button), meaning this will be in the developer environment... And as far as we know the developer environment is not a browser environment, it is a Swing environment (in fact it is a SWT environment, but I'm going too far here ☺).

So the first time Servoy calls the bean, it expects to find a Swing bean to place on the form, that's why we instantiate the SwingBean when we declare the variable (which is roughly equivalent to a call to a Constructor with no parameters).

OK, we will get back to our `getBeanInstanceMethod()`, but we already have everything we need to implement our setters.

We leave the getters just as Eclipse has done them; they are perfectly fine as-is.

The only one we need to change is the `getName()` which needs to be coded like that:

```
public String getName() {  
    return this.name;  
}
```

The setters will look strangely familiar to you, let's see:

```
public void setSize(Dimension size) {  
    this.size = size;  
    if (component != null) {  
        component.setSize(size);  
    }  
}  
  
public void setLocation(Point location) {  
    this.location = location;  
    if (component != null) {  
        component.setLocation(location);  
    }  
}  
  
public void setBackground(Color background) {  
    this.background = background;  
    if (component != null) {  
        component.setBackground(background);  
    }  
}  
  
public void setForeground(Color foreground) {  
    this.foreground = foreground;  
    if (component != null) {  
        component.setForeground(foreground);  
    }  
}  
  
public void setBorder(Border border) {  
    this.border = border;  
    if (component != null) {  
        component.setBorder(border);  
    }  
}  
  
public void setFont(Font font) {  
    this.font = font;  
    if (component != null) {  
        component.setFont(font);  
    }  
}
```

```

public void setToolTipText(String toolTipText) {
    this.toolTipText = toolTipText;
    if (component != null) {
        component.setToolTipText(toolTipText);
    }
}

public void setDataProviderID(String dataProviderID) {
    this.dataProviderID = dataProviderID;
    if (component != null) {
        component.setDataProviderID(dataProviderID);
    }
}

public void setTransparent(boolean transparent) {
    this.transparent = transparent;
    if (component != null) {
        component.setOpaque(!transparent);
    }
}

```

All these methods are build on the same principle, we store the value set by the user (from the Properties View Editor in Servoy Developer), then we check our component for null and forward the value to it...

The only one for which you actually need to think about a little is the setTransparent(), since we really only have an "opaque" property which is the exact contrary of "transparent" (isn't it?), we invert the boolean received before passing it to the setOpaque() method. No advanced Java topics here!

So it's time to get back to our getInstance() method, this method is called by Servoy when it wants to retrieve an instance of an object of type IComponent (that's why it is called this way: it comes from a well know Object Oriented Pattern called the "Factory" pattern, where an object is used to produce instances of objects from a set of parameters, see GOF pattern – "Gang Of Four" on the web).

So Servoy calls the method, and it sends 3 parameters to help us create the right kind of object:

- int: is the type of client actually running, and we only need to test if this is the Web client, and we have a constant for that in the IClientAccessPlugin interface: WEB_CLIENT
- IClientPluginAccess: the Servoy client (a reduced interface to it actually)
- Object[]: and array of objects, which are parameters. As far as I know there is only one object in this array, it is Servoy's UUID for the object as a String.

So we implement our method like this:

```

public IComponent getInstance(int appMode,
    IClientPluginAccess paramIClientPluginAccess,
    Object[] paramArrayOfObject) {

    String id = (String)paramArrayOfObject[0];
    if (appMode == IClientPluginAccess.WEB_CLIENT) {
        component = new WicketTestBean(id);
        initComponents();
    }
    return component;
}

```


- 1/ We get the ID as a String from the array of Object[] parameter passed
- 2/ we test if the call is from a web client, if so we
- 3/ set our component variable to be a WicketBean created from the constructor based on an ID (remember the WicketBean constructor?), and call for some component initializations (our wicket bean needs some default value, we will create the initComponents() in a minute: Quick Fix!).
- 4/ finally we return our component (a Swing bean or a Wicket bean, but still an IComponent) to Servoy

Our initComponents method only triggers the setters of our Wicket bean, with some default values if none have been provided ("Default" in the Properties View editor of Servoy).

```
/**
 * Initializes the wicket bean (sets all the default values).
 */
private void initComponents() {
    setName(name);
    setDataProviderID(dataProviderID);
    setBackground((background == null) ? Color.white : background);
    setForeground((foreground == null) ? Color.black : foreground);
    setBorder((border == null) ? new EtchedBorder(1) : border);
    setFont(font);
    setToolTipText(toolTipText);
    setTransparent(transparent);
}
```

That's it for our ServoyTestBean class, we can wrap it up.

Z. Wrapping it all up

Last but not least, if you have followed until here you will have noticed that right now our BeanInfo class is based on the SwingTestBean, which is not right because we want to use our IServoyBeanFactory ServoyTestBean.

It's time to change that so that it points to our ServoyTestBean instead.

A simple find/replace in Eclipse from "SwingTestBean" to "ServoyTestBean" should do the trick, and our only method should now look like that:

```
public PropertyDescriptor[] getPropertyDescriptors()
{
    try {
        List<PropertyDescriptor> liste = new ArrayList<PropertyDescriptor>();

        liste.add(new PropertyDescriptor("dataProviderID", ServoyTestBean.class));
        liste.add(new PropertyDescriptor("name", ServoyTestBean.class));
        liste.add(new PropertyDescriptor("background", ServoyTestBean.class));
        liste.add(new PropertyDescriptor("foreground", ServoyTestBean.class));
        liste.add(new PropertyDescriptor("font", ServoyTestBean.class));
        liste.add(new PropertyDescriptor("toolTipText", ServoyTestBean.class));
        liste.add(new PropertyDescriptor("transparent", ServoyTestBean.class));
        liste.add(new PropertyDescriptor("border", ServoyTestBean.class));
        liste.add(new PropertyDescriptor("size", ServoyTestBean.class));
        liste.add(new PropertyDescriptor("location", ServoyTestBean.class));

        PropertyDescriptor[] apropertydescriptor[] =
            liste.toArray(new PropertyDescriptor[0]);
    }
}
```

```
return apropertydescriptor;

} catch (Exception e) {

    Debug.error((new StringBuilder()).append(
        "SwingTestBeanInfo: unexpected exception: "
    ).append(e).toString());

}

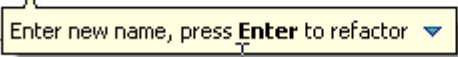
return null;
}
```

And the name of the class should also be changed, from SwingTestBeanInfo to SwingTestBeanInfo.

To do that, you can either select the name of the class in its signature, or select the file in the Package Explorer, and choose "Refactor > Rename...":

```
/**
 *
 * @author Servoy Stuff
 * http://www.servoy-stuff.net/
 */
public class SwingTestBeanInfo extends SimpleBeanInfo {

    /* (non-
    * @see java.beans.SimpleBeanInfo#getPropertyDescriptors()
    */
}
```



Either way, it will change the name of the class and the name of the file, and change every reference to it in our sources if any...

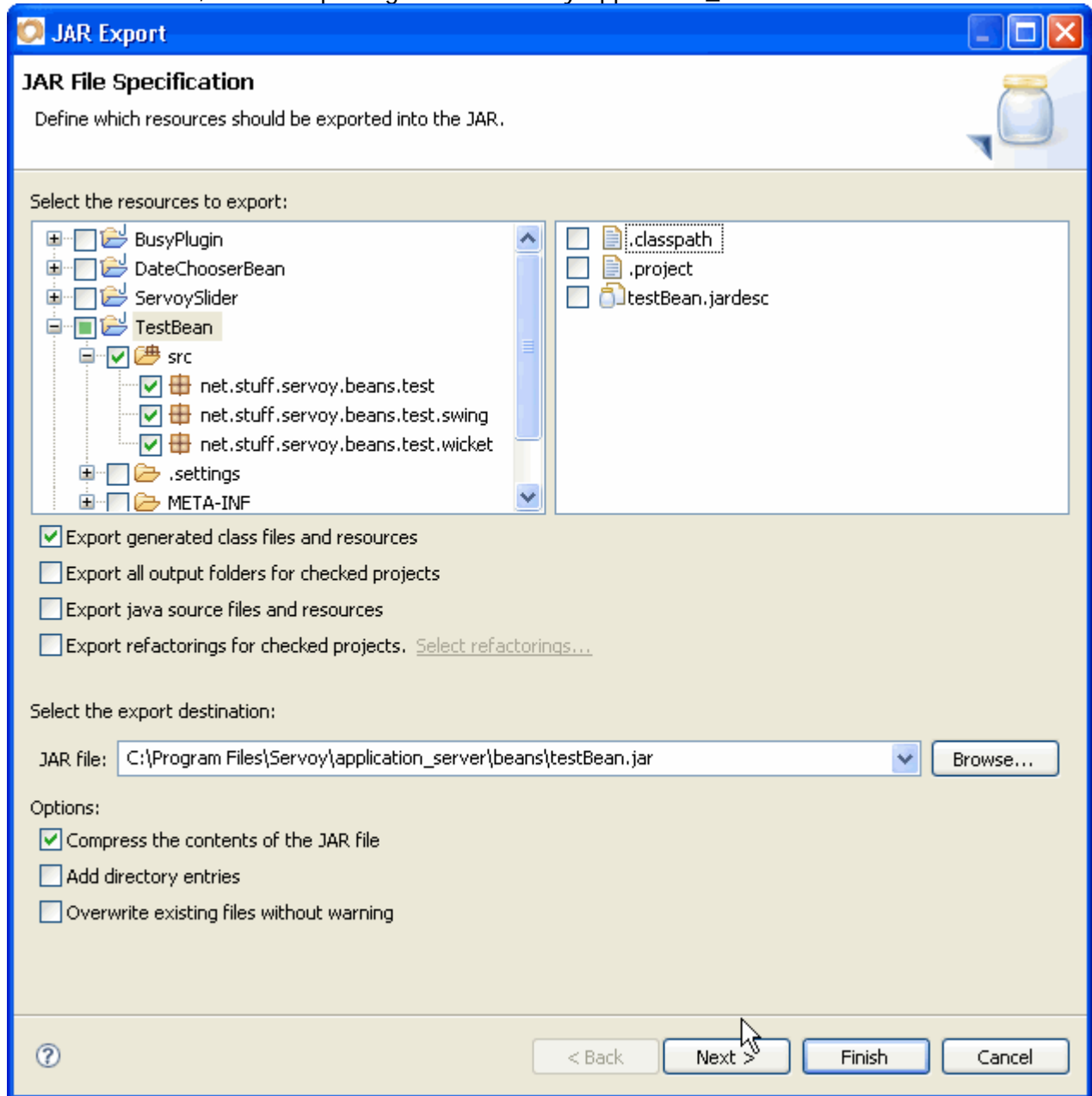
Of course, to wrap our bean we also need to create a MANIFEST.MF file with this content:

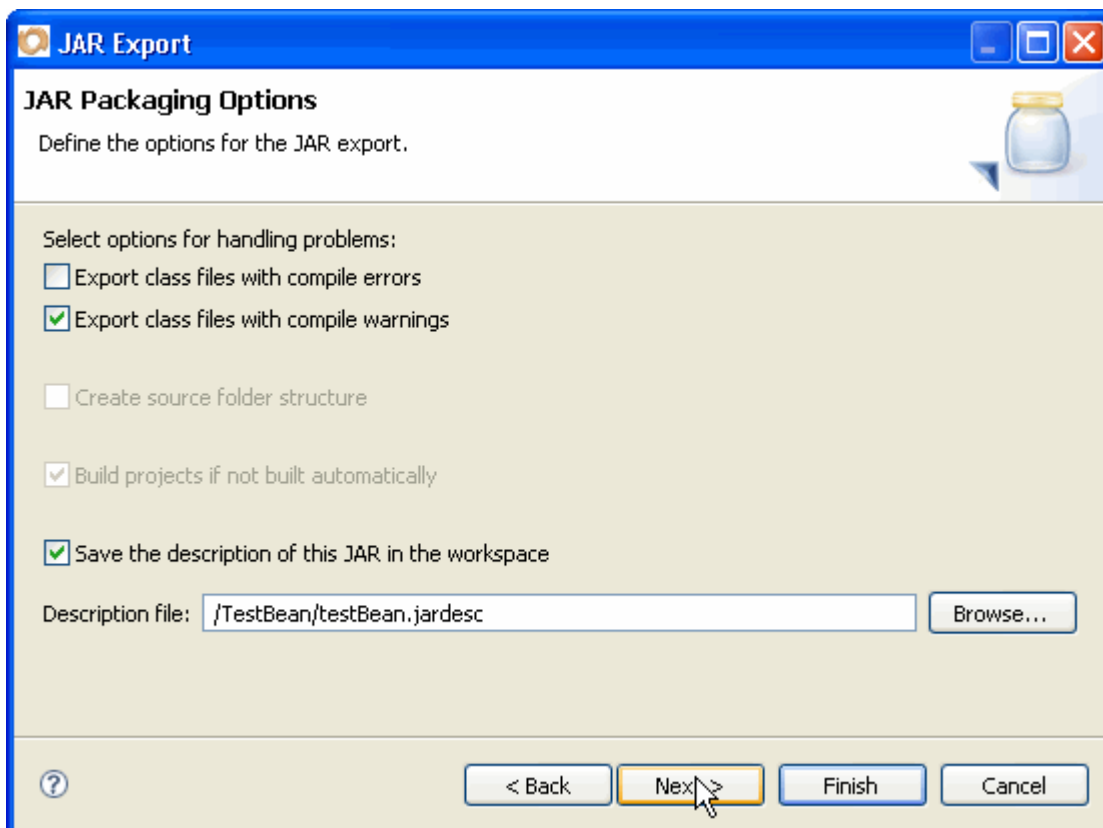
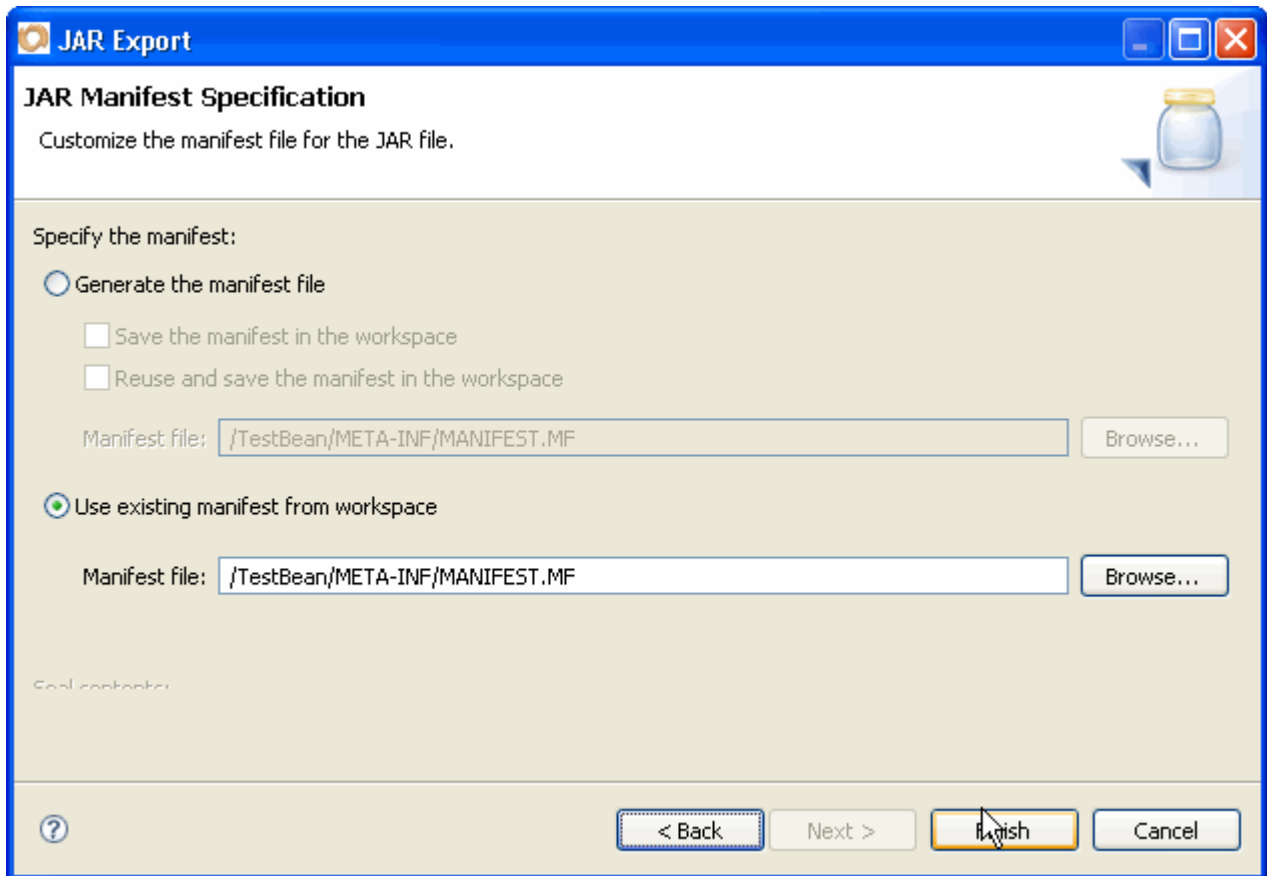
Manifest-Version: 1.0

Name: net/stuff/servoy/beans/test/ServoyTestBean.class

Java-Bean: True

And finally we need to create a JAR package, using the "Export..." function on our project, choosing "Java > JAR File", and then placing it in our Servoy/application_server/beans/ folder:





That's it! You can restart Servoy and place your bean on a form, give it a dataProvider, and see that we just made a text field that works exactly like Servoy's own text field. (Except for formatting, callbacks, etc. we did only the basics of an input component).

Try this bean in the smart client, and in the web client, and it should work just like a text field does (data-broadcasting included).

Of course this bean is not very useful you would say, but this part of the tutorial was not about "useful" but about addressing a lot of new concepts that you need to understand a little bit before we go back to our ServoySlider bean.

I have tried to make it as simple as possible so that you get a good grasp of all the main concepts.

Now it's up to you to play with it and try to add a few more properties for example, and go to the Wicket API and examples to learn more tips and tricks that are beyond the scope of this little tutorial but that might prove useful later when you will want to build Wicket beans...

Here you should have learned the basics of what Wicket beans are about in Servoy.

As always, you will find the complete Eclipse project on the Servoy Stuff web site, here:

http://www.servoy-stuff.net/tutorials/utis/t02/TestBean_EclipseProject.zip

(Import in any Eclipse distribution)

The compiled bean (targeted for java 1.5) will be available here:

<http://www.servoy-stuff.net/tutorials/utis/t02/testBean.jar>

(Put in you /beans folder)

And there is no solution attached because you can use this bean just like you would use a simple Text field, and I'm not going to try to teach you that ;-)

Hope you liked this tutorial, and that you did learn a few things from it, and that it will help you build some great beans for Servoy that will be web compatible.

Feel free to comment, question and suggest on the Servoy Stuff web site, I always like to hear from you!

And get ready for the next part where we will go back to our ServoySliderBean and prepare the ground for implementing the Wicket side of it. This will involved some refactoring and hopefully will teach you a few more useful tips and tricks in Java for Serclipse!

See you then,

Patrick Talbot

Servoy Stuff

2009-07-14