CaptainCasa Enterprise Client

CaptainCasa Enterprise Client

Developers' Guide
“Beta Parts”

Table of Contents

“Beta Parts”

There are certain parts of CaptainCasa Enterprise Client that are part of the delivery but that are not yet published as official part of the product. This documentation tells how to use these parts.

Please note:

- There is no guarantee that these parts will be official parts of CaptainCasa Enterprise Client at a certain point of time.
- Programming Interfaces may change.
- There is no warranty.

You want to use these parts in order to get them to know? Fine! - Please contact us in case of problems and/or questions.

You want to use this part seriously in future time? - Definitely contact us, so that we know about!

“Embedded Mini Server” - Run CaptainCasa as Fat Client

Overview

Yes, CaptainCasa Enterprise Client is a thin client framework from topology point of view. The client is an XML processor, rendering the XML layout content either using Java Swing or JavaFX. And the server is some processing that runs within the servlet/JSF management.

Running Client and Server in one Process

You can of course run both the client and the server within one Java-Process and by this run the whole UI and application as self-containing “fat client”.

The one way to do this is to use Tomcat and start Tomcat in the same process as the UI client. We provide a scenario in which Tomcat runs as embedded server within the client process – the communication between client and server is not going through http but through an internal Tomcat API.

But...:

- Even though Tomcat is a nice server with quite small footprint the whole solution still has some strong smell of “client-server”: you have to deploy into the embedded Tomcat etc. And you internally see Tomcat opening up several threads, which is not really usual for fat client operations...
- In addition: the embedded mode is not supported anymore with Tomcat >= 7.

Embedded Mini Server

As consequence we implemented a very restricted in-process JSF-processor which has the character of a library, which does not come with some own thread management and which does not require some deployment infrastructure. We called this environment “Embedded Mini Server”.

How does the client know that it should contact the Embedded Mini Server processing and not use the normal http communication?

- If the URL that is passed to the client starts with “mini://mini” then the communication part of the client will contact the mini server processing – and not use http.

Of course there are some restrictions:

- As mentioned the Embedded Mini Server does not come with an own thread management. So the threading is driven by the client: the client calls the server API, which is then directly executed within the thread of the client.
- Only the JSF processing and servlet processing is made available that is used in the context of CaptainCasa Enterprise Client! The Embedded Mini Server is not a fully featured servlet/JSF engine!

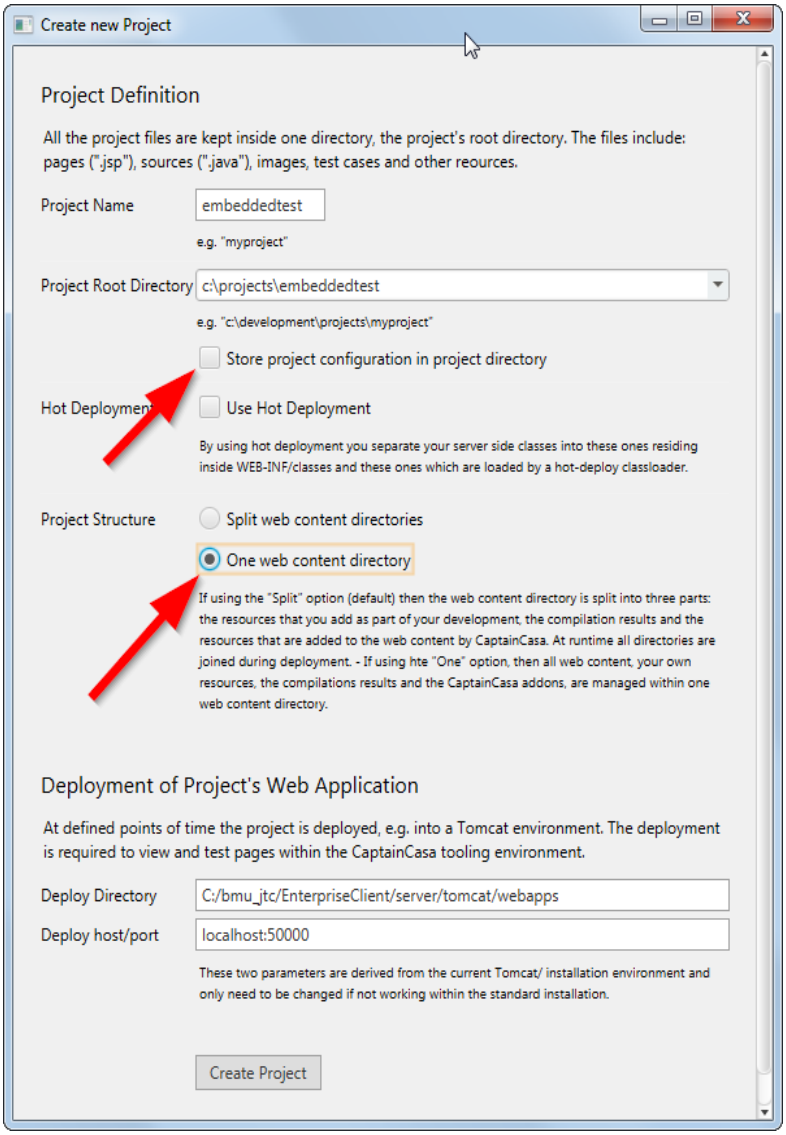
Making it work

The development of dialogs is still done the usual CaptainCasa way: the processing is running within the server, the development within the CaptainCasa toolset and Eclipse (or

any other IDE). So during development you are still working within a distributed environment. - But it's really simple to start your development result in “fat mode” any time in parallel.

Create the Project

Start the CaptainCasa Toolset and create a new project. In the project definition dialog define the following:

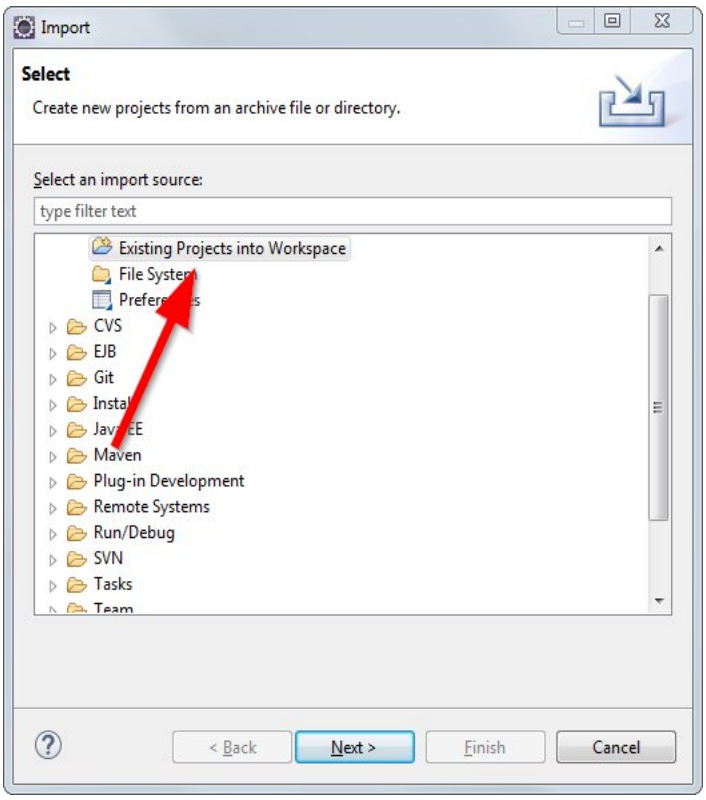


- The name of the project (here “embeddedtest”)
- The root directory of the project (here “c:\project\embeddedtest”)
- Uncheck the option “Store project configuration in project directory”
- Define the usage of “One web content directory”

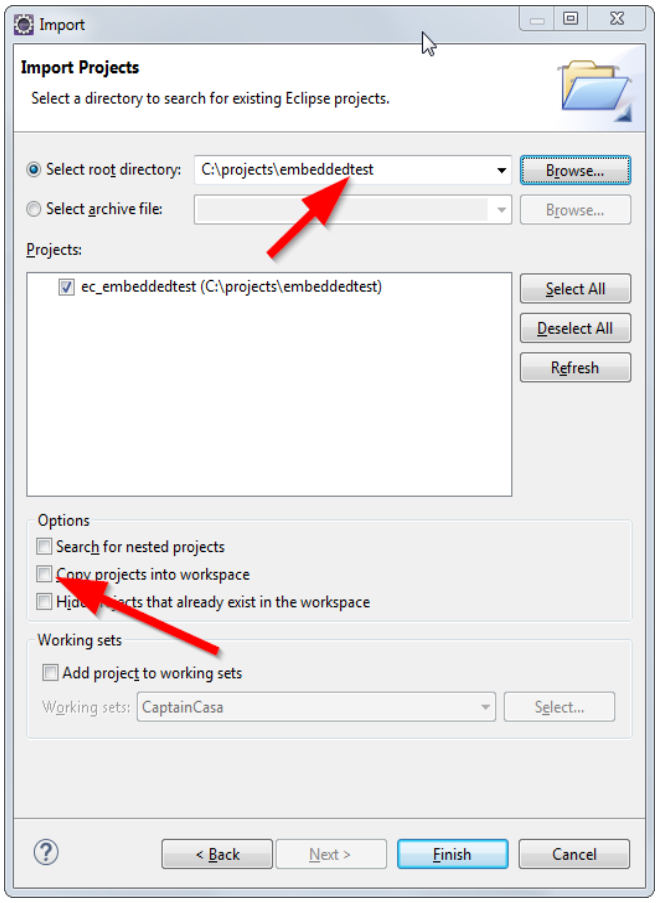
Then press “Create Project”.

Import Project into Eclipse

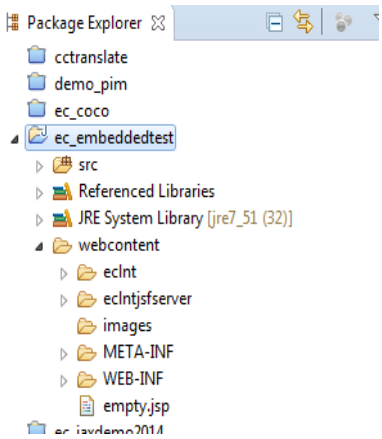
Now import the project into Eclipse. Select “File => Import...”:



Then select the root directory of the project and make sure that the project content is NOT copied into your workspace:



After pressing “Finish” the project will be imported and you see it in the project list:



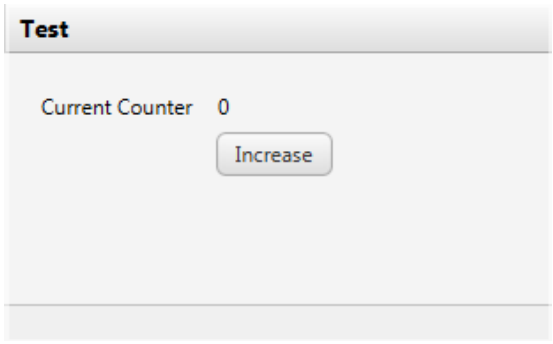
So you see: this is the normal creation of a project with some special configuration parameters. The only one that differs from the default creation of a project is the definition, that you want to keep all web content within one directory.

Develop a Dialog

The following is just normal working with CaptainCasa Enterprise Client. Please find details in the Tutorial “Creating the first Page”.

- Define a page (.jsp) and define the layout (XML) of the page.
- Develop the code for the page.

In our case a very simple page “test.jsp” is created as example.



The XML layout is:

```
<t:rowtitlebar id="g_1" text="Test" />
<t:rowbodypane id="g_2" rowdistance="5">
  <t:row id="g_3">
       <t:label id="g_4" text="Current Counter" width="100" />
       <t:label id="g_5" text="#{d.TestUI.counter}" />
  </t:row>
  <t:row id="g_6">
       <t:coldistance id="g_7" width="100" />
       <t:button id="g_8" actionListener="#{d.TestUI.onIncreaseAction}"
          text="Increase" />
  </t:row>
</t:rowbodypane>
<t:rowstatusbar id="g_9" />
```

The code is:

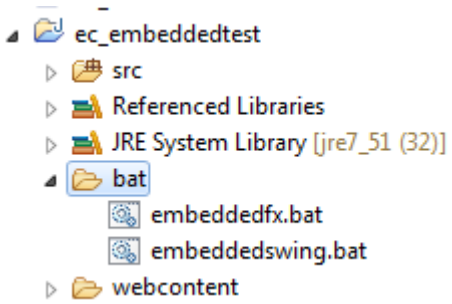
```
package managedbeans;

import java.io.Serializable;
import org.eclnt.editor.annotations.CCGenClass;
import org.eclnt.jsfserver.pagebean.PageBean;

import javax.faces.event.ActionEvent;

@CCGenClass (expressionBase="#{d.TestUI}")

public class TestUI
    extends PageBean
    implements Serializable
{
    // ------------------------------------------------------------------------
    // members
    // ------------------------------------------------------------------------

    int m_counter = 0;

    // ------------------------------------------------------------------------
    // constructors & initialization
    // ------------------------------------------------------------------------

    public TestUI()
    {
    }

    public String getPageName() { return "/test.jsp"; }
    public String getRootExpressionUsedInPage() { return "#{d.TestUI}"; }

    // ------------------------------------------------------------------------
    // public usage
    // ------------------------------------------------------------------------

    public int getCounter() { return m_counter; }
    public void setCounter(int value) { this.m_counter = value; }

    public void onIncreaseAction(ActionEvent event)
    {
        m_counter++;
    }
}
```

You may test the page in the preview of the Layout Editor.

Running it as Fat Client

For running this page “test.jsp” in a Fat Client mode, you need to simply start a Java Program and pass certain parameters to it.

For this example we create a sub directory “bat” within the project:

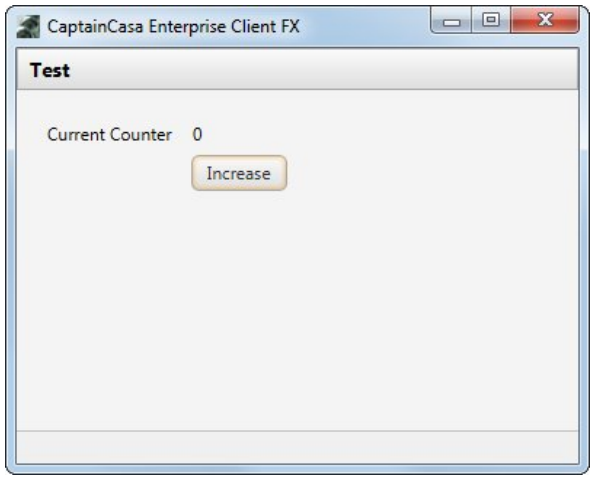


The directory contains two batch scripts, one for the JavaFX client and one for the Java Swing client. You can find the templates for the .bat files within the directory “<ccinnstalldir>/resources/embeddedminiserver”.

The script for the FX client is:

```
set CCINSTALLDIR=C:\temp\cc20140813_I2
set CCPROJECTDIR=C:\projects\embeddedtest

REM ...the jar eclntembeddminiserver (in your CaptainCasa installation directory)
set TCP=%CCINSTALLDIR%\resources\embeddedminiserver\eclntembeddedminiserver.jar

REM ...the client jars
set TCP=%TCP%;%CCPROJECTDIR%\webcontent\eclnt\libfx\eclntfx.jar
set TCP=%TCP%;%CCPROJECTDIR%\webcontent\eclnt\libfx\jpedal.jar

REM ...the CaptainCasa Server library
set TCP=%TCP%;%CCPROJECTDIR%\webcontent\WEB-INF\lib\eclntjsfserver.jar

REM ...project classes
set TCP=%TCP%;%CCPROJECTDIR%\webcontent\WEB-INF\classes

"%CCINSTALLDIR%\server\jre\bin\java.exe" "-cp" "%TCP%"
"org.eclnt.miniserver.MiniServerStarterFX" "page=/test.jsp" "webcontentdirectory=
%CCPROJECTDIR%\webcontent" "tempdirectory=c:\temp\embeddedtest"
```

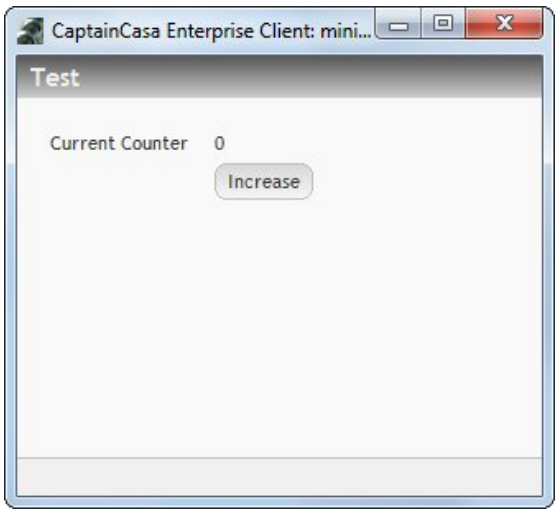
After starting you will see the following:



The script for the Swing client is:

```
set CCINSTALLDIR=C:\temp\cc20140813_I2
set CCPROJECTDIR=C:\projects\embeddedtest

REM ...the jar eclntembeddminiserver (in your CaptainCasa installation directory)
set TCP=%CCINSTALLDIR%\resources\embeddedminiserver\eclntembeddedminiserver.jar

REM ...the client jars (you may additional client jars if required)
set TCP=%TCP%;%CCPROJECTDIR%\webcontent\eclnt\lib\eclnt.jar

REM ...the CaptainCasa Server library
set TCP=%TCP%;%CCPROJECTDIR%\webcontent\WEB-INF\lib\eclntjsfserver.jar

REM ...project classes
set TCP=%TCP%;%CCPROJECTDIR%\webcontent\WEB-INF\classes

"%CCINSTALLDIR%\server\jre\bin\java.exe" "-cp" "%TCP%"
"org.eclnt.miniserver.MiniServerStarterSwing" "page=/test.jsp"
"webcontentdirectory=%CCPROJECTDIR%\webcontent"
"tempdirectory=c:\temp\embeddedtest"
```

After starting you will see the following:



Restrictions

The embedded mini server is NOT a full servlet/JSF container (otherwise it would not be “mini” anymore). It includes these functions that are required by CaptainCasa server side processing. The following list summarizes the restrictions that you should be pay attention to:

- You may not use expressions containing logic. E.g. “#{d.Xxxx.count == 9}” is not allowed. You may only use straight expressions like “#{d.Xxxxx.count}”.

...but what you still can do!

All the functions that have to do with HttpSession management are still available. The same is true for functions around the FacesContext. This means:

- You still can store “global variables” in the session context
- You still can access the FacesContext e.g. in order to get the view root.
- etc. etc.

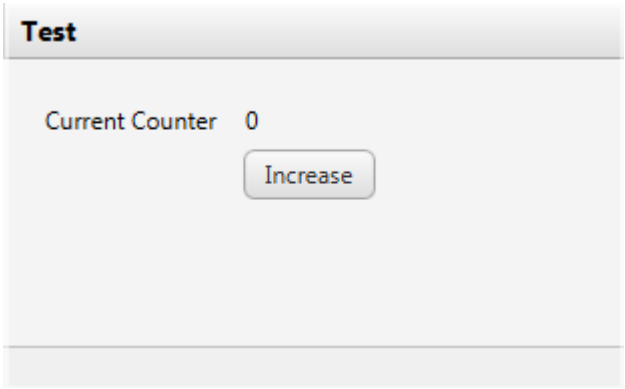
We clearly encourage you to keep the “server thinking” (multiple users are running in one Java process) as base of your development. Then you can really run one and the same software in “fat mode” and in “server mode”.

Self-containing Components

Overview

Let's introduce the self-containing component concept by example:

The following page is implemented first the “traditional” way – and then the self-containing way.



The “traditional” Way

The layout of the page is kept in a XML definition, that is part of a .jsp file:

```
<t:rowtitlebar id="g_1" text="Test" />
<t:rowbodypane id="g_2" rowdistance="5">
  <t:row id="g_3">
       <t:label id="g_4" text="Current Counter" width="100" />
       <t:label id="g_5" text="#{d.TestUI.counter}" />
  </t:row>
  <t:row id="g_6">
       <t:coldistance id="g_7" width="100" />
       <t:button id="g_8" actionListener="#{d.TestUI.onIncreaseAction}"
          text="Increase" />
  </t:row>
</t:rowbodypane>
<t:rowstatusbar id="g_9" />
```

In the layout there are components that are bound to some Java processing using expressions:

- The text of the label “g_5” is bound to “#{d.TestUI.counter}”
- The processing behind the button “g_8” is bound to “#{d.TestUI.onIncreaseAction}”

The code of the corresponding class is:

```
package managedbeans;

import java.io.Serializable;
import org.eclnt.editor.annotations.CCGenClass;
import org.eclnt.jsfserver.pagebean.PageBean;

import javax.faces.event.ActionEvent;

@CCGenClass (expressionBase="#{d.TestUI}")

public class TestUI
    extends PageBean
    implements Serializable
{
```

```
    // ------------------------------------------------------------------------
    // members
    // ------------------------------------------------------------------------

    int m_counter = 0;

    // ------------------------------------------------------------------------
    // constructors & initialization
    // ------------------------------------------------------------------------

    public TestUI()
    {
    }

    public String getPageName() { return "/test.jsp"; }
    public String getRootExpressionUsedInPage() { return "#{d.TestUI}"; }

    // ------------------------------------------------------------------------
    // public usage
    // ------------------------------------------------------------------------

    public int getCounter() { return m_counter; }
    public void setCounter(int value) { this.m_counter = value; }

    public void onIncreaseAction(ActionEvent event)
    {
        m_counter++;
    }
}
```

So there is a layout associated with some object at runtime. By using a PageBean-object (as in the example), you may now use this dialog as reusable module. The layout (XML) and the processing (Java) is separated – all state of a component (e.g. the text of the label) is either directly defined (text=”Current conter”) or is read from some processing object via expression.

The “self-containing” Way

In the self-continaing way the same dialog is defined in the following way:

```
<t:row id="g_1" componentbinding="#{d.TestScUI.anchor}" />
```

The jsp-page only contains one row with a COMPONENTBINDING definition. (Please note: the attribute “componentbinding” is not avaiable through the Layout Editor, please add it manually within the XML part of the .jsp file).

And the program behind is:

```
package managedbeans;

import java.io.Serializable;

import javax.faces.component.UIComponent;

import org.eclnt.editor.annotations.CCGenClass;
import org.eclnt.jsfserver.elements.BaseActionEvent;
import
org.eclnt.jsfserver.elements.impl.selfcontaining.ISCComponentActionListener;
import org.eclnt.jsfserver.elements.impl.selfcontaining.SC_BUTTONComponent;
import org.eclnt.jsfserver.elements.impl.selfcontaining.SC_COLDISTANCEComponent;
import org.eclnt.jsfserver.elements.impl.selfcontaining.SC_LABELComponent;
import org.eclnt.jsfserver.elements.impl.selfcontaining.SC_PANEComponent;
import org.eclnt.jsfserver.elements.impl.selfcontaining.SC_ROWBODYPANEComponent;
import org.eclnt.jsfserver.elements.impl.selfcontaining.SC_ROWComponent;
import org.eclnt.jsfserver.elements.impl.selfcontaining.SC_ROWSTATUSBARComponent;
import org.eclnt.jsfserver.elements.impl.selfcontaining.SC_ROWTITLEBARComponent;
import org.eclnt.util.valuemgmt.ValueManager;

@CCGenClass (expressionBase="#{d.TestScUI}")

public class TestScUI implements Serializable
{
```

```
    UIComponent m_anchor;
    SC_PANEComponent m_pane;

    SC_LABELComponent m_counterLabel = new SC_LABELComponent();

    public TestScUI()
    {
        renderPane();
    }

    public void setAnchor(UIComponent value)
    {
        if (m_anchor == value) return;
        m_anchor = value;
        m_anchor.getChildren().add(m_pane);
    }

    private void renderPane()
    {
        m_pane = new SC_PANEComponent();
        m_pane.setWidth("100%");
        m_pane.setHeight("100%");
        {
            // titlebar
            SC_ROWTITLEBARComponent titlebar = new SC_ROWTITLEBARComponent();
            m_pane.getChildren().add(titlebar);
            titlebar.setText("Test");
        }
        {
            SC_ROWBODYPANEComponent bodyPane = new SC_ROWBODYPANEComponent();
            m_pane.getChildren().add(bodyPane);
            bodyPane.setRowdistance("5");
            {
                SC_ROWComponent row = new SC_ROWComponent();
                bodyPane.getChildren().add(row);
                SC_LABELComponent l1 = new SC_LABELComponent();
                row.getChildren().add(l1);
                l1.setText("Current counter");
                l1.setWidth("100");
                row.getChildren().add(m_counterLabel);
                m_counterLabel.setText("0");
            }
            {
                SC_ROWComponent row = new SC_ROWComponent();
                bodyPane.getChildren().add(row);
                SC_COLDISTANCEComponent d = new SC_COLDISTANCEComponent();
                row.getChildren().add(d);
                d.setWidth("100");
                SC_BUTTONComponent b = new SC_BUTTONComponent();
                row.getChildren().add(b);
                b.setText("Increase");
                b.addSCActionListener(new ISCComponentActionListener()
                {
                    @Override
                    public void processSCActionEvent(BaseActionEvent arg0)
                    {
                        String s = (String)m_counterLabel.getText();
                        int currentCounter = ValueManager.decodeInt(s,-1);
                        currentCounter++;
                        m_counterLabel.setText(""+currentCounter);
                    }
                });
            }
        }
        {
            // statusbar
            SC_ROWSTATUSBARComponent statusbar = new SC_ROWSTATUSBARComponent();
            m_pane.getChildren().add(statusbar);
        }
    }


}
```

What you see is...:

- The dialog is built up using SC_* components (SC for “self containing”). The components

are created and are hierarchically arranged. So what you did within the XML in the "traditional" example, you now do via code – you assemble the component tree.

- The attributes of the components are filled via corresponding set/get methods.
- There is no expression binding anymore – you just access the data of a component through the component itself.
- There is an actionListener with the button: if the button is pressed then the actionListener is called.

The component tree that you build up is stored in the member "m_pane". The binding to the JSF processing is done through the anchor-definition ("setAnchor(...)"), in which the component node of the page is passed.

Event-Processing

Each component has two types of listeners:

- ISCComponentActionListener that you add by calling "addSCActionListener"
- ISCComponentAttributeListener that you add by calling "addSCAttributeListener"

Action Events

The action event processing you already got to know in the example. In simple cases there is only one event-type that you need to take care of (as in the example). But it is possible as well, that there are more than one event type per component: a button for example may react on clicks – but also on drop operations.

Each event type is represented by a certain class, all event classes are named following the pattern "BaseAtionEvent*", e.g. there is an event "BaseActionEventInvoke" and "BaseActionEventDrop". In the actionListener processing you have to check the event type and react correspondingly.

Data-Processing

The value of an attribute is available through a correpsonding get-ter method. For example the SC_FIELD component provides a getText()-method.

Example:

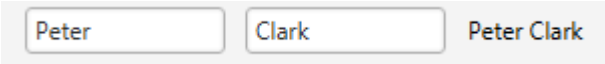
When the user presses the "Concatenate button" then the content of both fields is concatenated, the result is shown following the button. The corresponding code (that is plugged into the example above) is:

```
{
    SC_ROWComponent row = new SC_ROWComponent();
    bodyPane.getChildren().add(row);
    row.setColdistance("10");
    final SC_FIELDComponent firstName = new SC_FIELDComponent();
    row.getChildren().add(firstName);
    firstName.setText("");
    firstName.setWidth("100");
    firstName.setWidth("100");
    final SC_FIELDComponent lastName = new SC_FIELDComponent();
    row.getChildren().add(lastName);
    lastName.setText("");
    lastName.setWidth("100");
```

```
                lastName.setWidth("100");
                final SC_BUTTONComponent button = new SC_BUTTONComponent();
                row.getChildren().add(button);
                button.setText("Concatenate");
                final SC_LABELComponent result = new SC_LABELComponent();
                row.getChildren().add(result);
                button.addSCActionListener(new ISCComponentActionListener()
                {
                    @Override
                    public void processSCActionEvent(BaseActionEvent arg0)
                    {
                        result.setText(firstName.getText() + " " +
lastName.getText());
                    }
                });
            }
```

You see: the current value of the fields is just picked by calling “getText()”.

If you want to update the result field immediately when the user does update the fields, then you need to define some flushing with the fields:

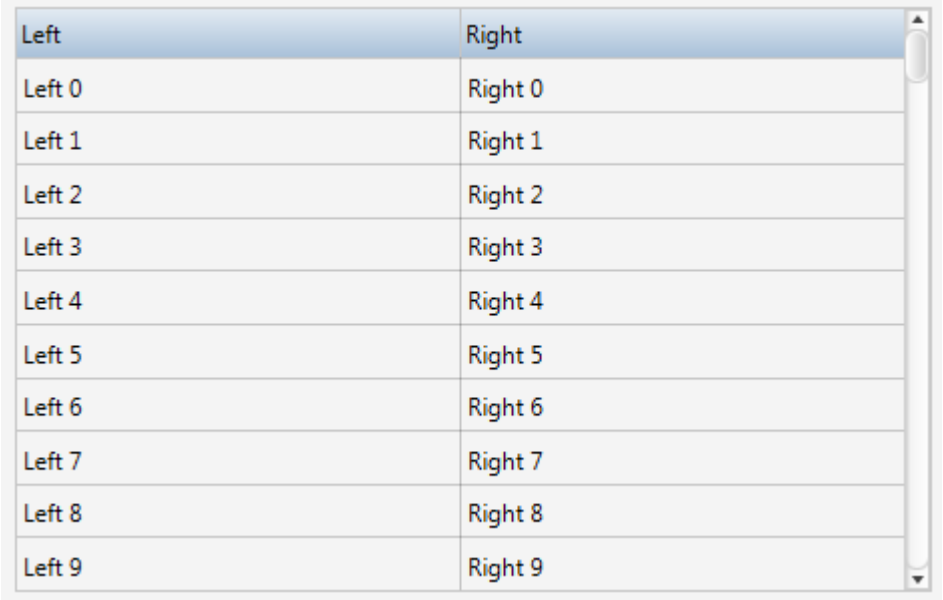



```
            {
                SC_ROWComponent row = new SC_ROWComponent();
                bodyPane.getChildren().add(row);
                row.setColdistance("10");
                final SC_FIELDComponent firstName = new SC_FIELDComponent();
                row.getChildren().add(firstName);
                firstName.setText("");
                firstName.setWidth("100");
                firstName.setWidth("100");
                firstName.setFlush("true");
                final SC_FIELDComponent lastName = new SC_FIELDComponent();
                row.getChildren().add(lastName);
                lastName.setText("");
                lastName.setWidth("100");
                lastName.setWidth("100");
                lastName.setFlush("true");
                final SC_LABELComponent result = new SC_LABELComponent();
                row.getChildren().add(result);
                firstName.addSCActionListener(new ISCComponentActionListener()
                {
                    @Override
                    public void processSCActionEvent(BaseActionEvent event)
                    {
                        if (event instanceof BaseActionEventFlush)
                            result.setText(firstName.getText() + " " +
lastName.getText());
                    }
                });
                lastName.addSCActionListener(new ISCComponentActionListener()
                {
                    @Override
                    public void processSCActionEvent(BaseActionEvent event)
                    {
                        if (event instanceof BaseActionEventFlush)
                            result.setText(firstName.getText() + " " +
lastName.getText());
                    }
                });
            }
```

The flushing is triggered once the user leaves the field. If you want to trigger it during data input, then you may use the setFlushtimer() method with the SC_FIELD component. In this method you pass a duration as number of milliseconds (e.g. 1000). If the user changed the field content and does not do any further input anymore then the flushing is triggered after the duration specified.

Grid processing

Let's take a look onto the following example:



Left	Right
Left 0	Right 0
Left 1	Right 1
Left 2	Right 2
Left 3	Right 3
Left 4	Right 4
Left 5	Right 5
Left 6	Right 6
Left 7	Right 7
Left 8	Right 8
Left 9	Right 9

The corresponding code is:

```
public class TestScUI implements Serializable
{
    public class GridItem extends SC_FIXGRIDItem
    {
        List<ISCComponent> i_cellComponents = new ArrayList<ISCComponent>();
        SC_LABELComponent i_left = new SC_LABELComponent();
        SC_LABELComponent i_right = new SC_LABELComponent();
        public GridItem()
        {
            i_cellComponents.add(i_left);
            i_cellComponents.add(i_right);
        }
        @Override
        public List<ISCComponent> getCellComponents() { return
i_cellComponents; }
        @Override
        public void onRowSelect() {} // click processing...
        @Override
        public void onRowExecute() {} // double click processing...
    }

    UIComponent m_anchor;
    SC_PANEComponent m_pane;

    SC_FIXGRIDListBinding<GridItem> m_grid = new
SC_FIXGRIDListBinding<GridItem>();

    public TestScUI()
    {
        renderPane();
        for (int i=0; i<100; i++)
        {
            GridItem gi = new GridItem();
            gi.i_left.setText("Left " + i);
            gi.i_right.setText("Right " + i);
            m_grid.getItems().add(gi);
        }
    }

    public void setAnchor(UIComponent value)
    {
        if (m_anchor == value) return;
        m_anchor = value;
```

```
        m_anchor.getChildren().add(m_pane);
    }

    private void renderPane()
    {
        m_pane = new SC_PANEComponent();
        m_pane.setWidth("100%");
        m_pane.setHeight("100%");
        {
            SC_ROWBODYPANEComponent bodyPane = new SC_ROWBODYPANEComponent();
            m_pane.getChildren().add(bodyPane);
            bodyPane.setRowdistance("5");
            {
                SC_ROWComponent row = new SC_ROWComponent();
                bodyPane.getChildren().add(row);
                SC_FIXGRIDComponent grid = new SC_FIXGRIDComponent();
                row.getChildren().add(grid);
                grid.setSbvisibleamount(10);
                grid.setWidth("100%");
                grid.setBorder("left:1;top:1;right:1;bottom:1;color:#00000030");
                grid.setBordercolor("#00000030");
                grid.setBorderheight("1");
                grid.setBorderwidth("1");
                grid.setObjectbinding(m_grid);
                {
                    SC_GRIDCOLComponent col = new SC_GRIDCOLComponent();
                    grid.getChildren().add(col);
                    col.setText("Left");
                    col.setWidth("50%");
                    SC_LABELComponent l = new SC_LABELComponent();
                    col.getChildren().add(l);
                }
                {
                    SC_GRIDCOLComponent col = new SC_GRIDCOLComponent();
                    grid.getChildren().add(col);
                    col.setText("Right");
                    col.setWidth("50%");
                    SC_LABELComponent l = new SC_LABELComponent();
                    col.getChildren().add(l);
                }
            }
        }
    }


}
```

Each grid is represented by the component SC_FIXGRID. Inside the SC_FIXGRID component you define the columns (SC_GRIDCOL), inside the columns you define the cell component type (SC_GRIDCOL => SC_LABEL). This is the render-part.

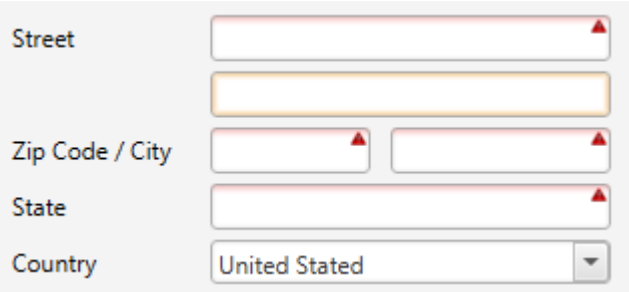
On data side you have to define a special list object (m_grid) of type “SC_FIXGRIDListBinding”. This list provides a getItems() method in which you add/remove/update the items of the list – each item being optically represented by a correpsonding row within the grid. The item must be an object of class SC_FIXGRIDItem. Each item must provide the component instanced for the corresponding row.

Please pay attention: The components that you define within the render() method must exactly match the components that you define within the item!

We now built up the grid using SC_LABEL components as cell components. You may use any other component of course as well.

Creating Application Components

The creation of application components is very simple. Let's assume you want to create a component for editing an address:



The address component already comes with features like:

- observation if mandatory input was done
- if “United States” is selected then a state needs to be defined, otherwise no state needs to be defined

All you need is to derive a class from e.g. component SC_PANE:

```
package managedbeans;

import
org.eclnt.jsfserver.elements.impl.selfcontaining.ISCComponentAttributeListener;
import
org.eclnt.jsfserver.elements.impl.selfcontaining.SCComponentAdapterBinding;
import org.eclnt.jsfserver.elements.impl.selfcontaining.SC_COLDISTANCEComponent;
import org.eclnt.jsfserver.elements.impl.selfcontaining.SC_COMBOBOXComponent;
import org.eclnt.jsfserver.elements.impl.selfcontaining.SC_FIELDComponent;
import org.eclnt.jsfserver.elements.impl.selfcontaining.SC_LABELComponent;
import org.eclnt.jsfserver.elements.impl.selfcontaining.SC_PANEComponent;
import org.eclnt.jsfserver.elements.impl.selfcontaining.SC_ROWComponent;
import org.eclnt.jsfserver.elements.util.ValidValuesBinding;

public class SC_ADDRESSComponent extends SC_PANEComponent
{
    // ------------------------------------------------------------------------
    // inner classes
    // ------------------------------------------------------------------------

    public static class AddressData
    {
        public String i_street1;
        public String i_street2;
        public String i_zipCode;
        public String i_city;
        public String i_state;
        public String i_country = "US";
    }

    public class AttributeUpdateListener implements ISCComponentAttributeListener
    {
        @Override
        public void processSCAttributeValueChange(String arg0, Object arg1)
        {
            transferComponentsToData();
        }
    }

    // ------------------------------------------------------------------------
    // members
    // ------------------------------------------------------------------------

    AddressData m_data;
    SC_FIELDComponent m_fieldStreet1;
    SC_FIELDComponent m_fieldStreet2;
    SC_FIELDComponent m_fieldZipCode;
    SC_FIELDComponent m_fieldCity;
    SC_ROWComponent m_rowState;
    SC_FIELDComponent m_fieldState;
    SC_COMBOBOXComponent m_comboCountry;

    AttributeUpdateListener m_attributeUpdateListener = new
AttributeUpdateListener();
```

```
// ------------------------------------------------------------------------
// constructors
// ------------------------------------------------------------------------

public SC_ADDRESSComponent()
{
    this(new AddressData());
}

public SC_ADDRESSComponent(AddressData data)
{
    render();
    m_data = data;
    transferDataToComponents();
}
// ------------------------------------------------------------------------
// public usage
// ------------------------------------------------------------------------

// ------------------------------------------------------------------------
// private usage
// ------------------------------------------------------------------------

private void render()
{
    setRowdistance("5");
    {
        SC_ROWComponent r = new SC_ROWComponent();
        addSCChild(r);
        SC_LABELComponent l = new SC_LABELComponent();
        r.addSCChild(l);
        l.setText("Street");
        l.setWidth(100);
        m_fieldStreet1 = new SC_FIELDComponent();
        r.addSCChild(m_fieldStreet1);
        m_fieldStreet1.setWidth("200");
        m_fieldStreet1.setFlush(true);
        m_fieldStreet1.addSCAttributeListener(m_attributeUpdateListener);
    }
    {
        SC_ROWComponent r = new SC_ROWComponent();
        addSCChild(r);
        SC_LABELComponent l = new SC_LABELComponent();
        r.addSCChild(l);
        l.setText("");
        l.setWidth(100);
        m_fieldStreet2 = new SC_FIELDComponent();
        r.addSCChild(m_fieldStreet2);
        m_fieldStreet2.setWidth("200");
        m_fieldStreet2.setFlush(true);
        m_fieldStreet2.addSCAttributeListener(m_attributeUpdateListener);
    }
    {
        SC_ROWComponent r = new SC_ROWComponent();
        addSCChild(r);
        SC_LABELComponent l = new SC_LABELComponent();
        r.addSCChild(l);
        l.setText("Zip Code / City");
        l.setWidth("100");
        m_fieldZipCode = new SC_FIELDComponent();
        r.addSCChild(m_fieldZipCode);
        m_fieldZipCode.setWidth(80);
        m_fieldZipCode.setFlush(true);
        m_fieldZipCode.addSCAttributeListener(m_attributeUpdateListener);
        SC_COLDISTANCEComponent d = new SC_COLDISTANCEComponent();
        r.addSCChild(d);
        d.setWidth(10);
        m_fieldCity = new SC_FIELDComponent();
        r.addSCChild(m_fieldCity);
        m_fieldCity.setWidth(110);
        m_fieldCity.setFlush(true);
        m_fieldCity.addSCAttributeListener(m_attributeUpdateListener);
    }
    {
        m_rowState = new SC_ROWComponent();
        addSCChild(m_rowState);
        SC_LABELComponent l = new SC_LABELComponent();
```

```
            m_rowState.addSCChild(l);
            l.setText("State");
            l.setWidth(100);
            m_fieldState = new SC_FIELDComponent();
            m_rowState.addSCChild(m_fieldState);
            m_fieldState.setWidth("200");
            m_fieldState.setFlush(true);
            m_fieldState.addSCAttributeListener(m_attributeUpdateListener);
        }
        {
            SC_ROWComponent r = new SC_ROWComponent();
            addSCChild(r);
            SC_LABELComponent l = new SC_LABELComponent();
            r.addSCChild(l);
            l.setText("Country");
            l.setWidth(100);
            m_comboCountry = new SC_COMBOBOXComponent();
            r.addSCChild(m_comboCountry);
            m_comboCountry.setWidth("200");
            m_comboCountry.setValidvaluesbinding(createCountryValidValues());
            m_comboCountry.setFlush(true);
            m_comboCountry.addSCAttributeListener(m_attributeUpdateListener);
        }
    }

    private void transferComponentsToData()
    {
        m_data.i_street1 = (String)m_fieldStreet1.getText();
        m_data.i_street2 = (String)m_fieldStreet2.getText();
        m_data.i_zipCode = (String)m_fieldZipCode.getText();
        m_data.i_city = (String)m_fieldCity.getText();
        m_data.i_state = (String)m_fieldState.getText();
        m_data.i_country = (String)m_comboCountry.getValue();
        processUILogic();
    }

    private void transferDataToComponents()
    {
        m_fieldStreet1.setText(m_data.i_street1);
        m_fieldStreet2.setText(m_data.i_street2);
        m_fieldZipCode.setText(m_data.i_zipCode);
        m_fieldCity.setText(m_data.i_city);
        m_fieldState.setText(m_data.i_state);
        m_comboCountry.setValue(m_data.i_country);
        processUILogic();
    }

    private void processUILogic()
    {
        m_fieldStreet1.setBgpaint(null);
        m_fieldZipCode.setBgpaint(null);
        m_fieldCity.setBgpaint(null);
        m_fieldState.setBgpaint(null);
        m_comboCountry.setBgpaint(null);
        if ("US".equals(m_data.i_country))
        {
            m_rowState.setRendered(true);
            if (m_data.i_state == null || m_data.i_state.trim().length() == 0)
{ m_fieldState.setBgpaint("error()"); }
        }
        else
        {
            m_rowState.setRendered(false);
        }
        if (m_data.i_street1 == null || m_data.i_street1.trim().length() == 0)
{ m_fieldStreet1.setBgpaint("error()"); }
        if (m_data.i_zipCode == null || m_data.i_zipCode.trim().length() == 0)
{ m_fieldZipCode.setBgpaint("error()"); }
        if (m_data.i_city == null || m_data.i_city.trim().length() == 0)
{ m_fieldCity.setBgpaint("error()"); }
        if (m_data.i_country == null || m_data.i_country.trim().length() == 0)
{ m_comboCountry.setBgpaint("error()"); }
    }

    private ValidValuesBinding createCountryValidValues()
    {
        ValidValuesBinding result = new ValidValuesBinding();
        result.addValidValue("CH","Sitzerland");
```

```
        result.addValidValue("DE","Germany");
        result.addValidValue("FR","France");
        result.addValidValue("US","United Stated");
        return result;
    }
}
```

You see the coding principle is exactly the same as with a normal dialog, but now the code is encapsulated within an SC_ADDRESS component.

In the SC_ADDRESS component the data transfer from/into the component is done using the data structure “AddressData” which is transferred into the individual field/combobox components.

The component itself might be used by other classes in the same was e.g. SC_PANE is used:

```
    SC_ADDRESSComponent m_address = new SC_ADDRESSComponent();
...
...
...
    renderin of dialog:
            {
                SC_ROWComponent row = new SC_ROWComponent();
                bodyPane.getChildren().add(row);
                row.getChildren().add(m_address);
            }
```

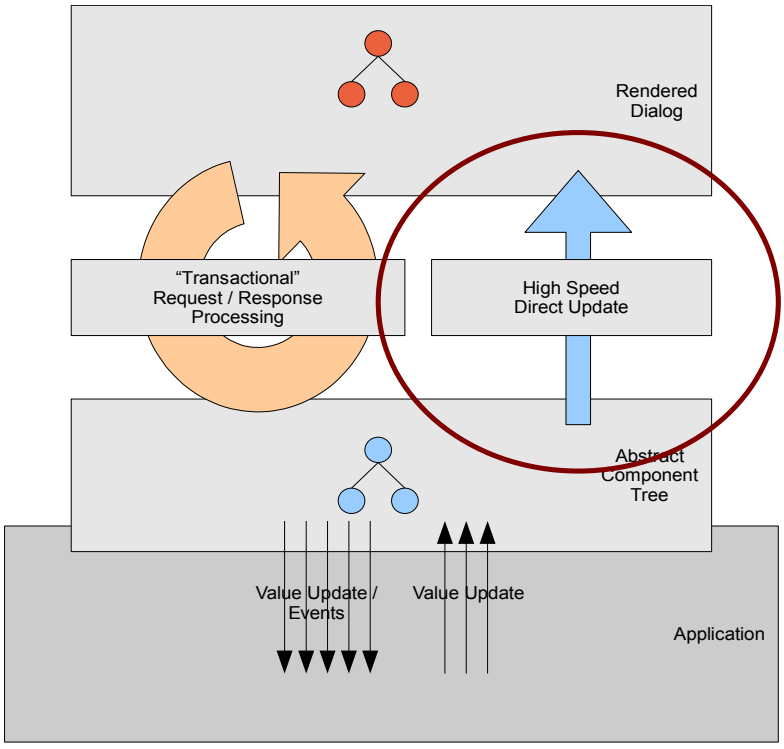
Direct Update of Client

By default there is a request-response based communication between client and server. This type of communication forms the “consistency backbone” of the application: the user updates data in the frontend, the updated data is synchronized with the server processing, the server side data is passed back to the client. During communication the user input is blocked.

While this communication is “perfect” for ensuring consistency it is not perfect for very fast updates of the page:

- There is a certain overhead involved with each request-response-cycle, which is not relevant for normal situations, but which is relevant if e.g. updating the page via request-response 10 times per second.
- The input of data is not really comfortable, if the page is blocked due to communication very frequently.

Consequence: we introduced some parallel communication line to the client – the direct update. Via direct update you may change the content of read-only data components very fast, without going through the request-response cycle.



When the client runs decoupled from the server side processing, then the direct update uses a parallel http connection that is opened up by the client. The communication principle is similar to long polling scenarios: the client sends a URL request to the server, the server is not responding immediately but passes the information back when available. You may compare it with a slow page loading in other scenarios. - If the communication breaks for any reason (time out, ...) then the client is notified and immediately opens up the next http connection to gather for data on server side.

If the client is running as fat client together with the server, then there is no explicit communication at all, but there is a direct API update from server to client side.

What to do

Tell the JSP Page to open up a Direct Update Communication

A direct update will only opened up, if the JSP page contains a CLIENTDIRECTUPDATE component.

```
<t:beanprocessing id="g_1" >
    <t:clientdirectupdate id="g_2" />
</t:beanprocessing>
...
...
<t:pageaddons id="g_pa"/>
```

You do not have to define any further parameters. In nested dialog scenarios, it is enough to define the CLIENTDIRECTUPDATE component within the out most dialog.

Invoke the Update from your Code

The interface is quite simple:

```
IDirectUpdate du = DirectUpdate.getInstance();

...
...
du.update(componentId,attribute,value);
```

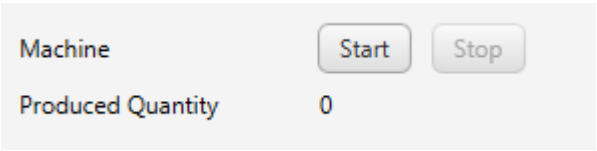
There is one hidden complexity: you need to get the instance of IDirectUpdate within the request-response processing thread. You may use it afterwards in some different thread.

...implicit Direct Update for self-containing Components

If using the self-contiaining components then you do not have to worry about calling the DirectUpdate-interface at all – it is implicitly called by the component. You still have to define the CLIENTIDERCTUPDATE within your page! - but nothing more.

Example – Controlling a Machine

The following dialog shows the monitoring of a machine:



“Start” and “Stop” are normal buttons – invoking the normal request-response processing. The produced quantity is updated by direct update.

The XML of the layout is:

```
<t:beanprocessing id="g_1">
  <t:clientdirectupdate id="g_2" />
</t:beanprocessing>
<t:rowbodypane id="g_3" rowdistance="5">
  <t:row id="g_4">
       <t:label id="g_5" text="Machine" width="150" />
       <t:button id="g_6"
          actionListener="#{d.TestDirectUpdate.onStartAction}"
          enabled="#{d.TestDirectUpdate.startEnabled}" text="Start" />
       <t:coldistance id="g_7" />
       <t:button id="g_8"
```

```
          actionListener="#{d.TestDirectUpdate.onStopAction}"
          enabled="#{d.TestDirectUpdate.stopEnabled}" text="Stop" />
  </t:row>
  <t:row id="g_9">
       <t:label id="g_10" text="Produced Quantity" width="150" />
       <t:label id="PRODUCEDQTY" text="#{d.TestDirectUpdate.producedQty}" />
  </t:row>
</t:rowbodypane>
```

You see:

- There is the CLIENTDIRECTUPDATE component in be BEANPROCESSING section.
- The label for the produced quantity has some defined id “PRODUCEDQTY”

The code looks as follows:

```
package managedbeans;

import java.io.Serializable;

import javax.faces.event.ActionEvent;

import org.eclnt.editor.annotations.CCGenClass;
import org.eclnt.jsfserver.directupdate.DirectUpdate;
import org.eclnt.jsfserver.directupdate.IDirectUpdate;
import org.eclnt.jsfserver.pagebean.PageBean;

@CCGenClass (expressionBase="#{d.TestDirectUpdate}")

public class TestDirectUpdate
    extends PageBean
    implements Serializable
{
    // ------------------------------------------------------------------------
    // inner classes
    // ------------------------------------------------------------------------

    class ProductionSimulator extends Thread
    {
        @Override
        public void run()
        {
            while (m_machineRunning)
            {
                try { Thread.sleep(100); } catch (Throwable t) {}
                increaseProducedQty();
                System.out.println("qty = " + m_producedQty);
            }
        }
    }

    // ------------------------------------------------------------------------
    // members
    // ------------------------------------------------------------------------

    IDirectUpdate m_directUpdate;

    long m_producedQty = 0;
    boolean m_machineRunning = false;

    // ------------------------------------------------------------------------
    // constructors & initialization
    // ------------------------------------------------------------------------

    public TestDirectUpdate()
    {
        m_directUpdate = DirectUpdate.getInstance();
    }

    public String getPageName() { return "/testdirectupdate.jsp"; }
    public String getRootExpressionUsedInPage() { return "#{d.TestDirectUpdate}";
}

    // ------------------------------------------------------------------------
    // public usage
    // ------------------------------------------------------------------------
```

```
    public long getProducedQty() { return m_producedQty; }
    public void increaseProducedQty()
    {
        this.m_producedQty++;
        m_directUpdate.update("PRODUCEDQTY","text",""+m_producedQty);
    }

    public boolean getStopEnabled() { return m_machineRunning; }
    public boolean getStartEnabled() { return !m_machineRunning; }

    public void onStopAction(ActionEvent event)
    {
        m_machineRunning = false;
    }

    public void onStartAction(ActionEvent event)
    {
        m_machineRunning = true;
        (new ProductionSimulator()).start();
    }

    // ------------------------------------------------------------------------
    // private usage
    // ------------------------------------------------------------------------
}
```

You see:

- A thread is simulating the machine – the thread is started and stopped in the action listeners of the buttons.
- At creation point of time of the object the IDirectUpdate instance is picked and stored.
- When the production simulation thread updates the produced quantity then the update is both done in the member “m_producedQty” and is sent via direct update to the client as well. For directly updating the client the id “PRODUCEDQTY” is used.

Same Example – now with self-containing Components

The layout definition is:

```
<t:beanprocessing id="g_1">
  <t:clientdirectupdate id="g_2" />
</t:beanprocessing>
<t:row id="g_3" componentbinding="#{d.TestDirectUpdateSC.anchor}" />
<t:pageaddons id="g_pa"/>
```

Nothing special here:

- You need to define the CLIENTDIRECTUPDATE component.
- You need to define the typical anchor component.

The code is:

```
package managedbeans;

import java.io.Serializable;

import org.eclnt.editor.annotations.CCGenClass;
import org.eclnt.jsfserver.elements.BaseActionEvent;
import
org.eclnt.jsfserver.elements.impl.selfcontaining.ISCComponentActionListener;
import org.eclnt.jsfserver.elements.impl.selfcontaining.SC_BUTTONComponent;
import org.eclnt.jsfserver.elements.impl.selfcontaining.SC_COLDISTANCEComponent;
import org.eclnt.jsfserver.elements.impl.selfcontaining.SC_LABELComponent;
import org.eclnt.jsfserver.elements.impl.selfcontaining.SC_PANEComponent;
import org.eclnt.jsfserver.elements.impl.selfcontaining.SC_ROWComponent;

import javax.faces.component.UIComponent;
import javax.faces.event.ActionEvent;
```

```
@CCGenClass (expressionBase="#{d.TestDirectUpdateSC}")

public class TestDirectUpdateSC implements Serializable
{
    // ------------------------------------------------------------------------
    // inner classes
    // ------------------------------------------------------------------------

    class ProductionSimulator extends Thread
    {
        @Override
        public void run()
        {
            while (m_machineRunning)
            {
                try { Thread.sleep(100); } catch (Throwable t) {}
                increaseProducedQty();
                System.out.println("qty = " + m_producedQty);
            }
        }
    }

    // ------------------------------------------------------------------------
    // members
    // ------------------------------------------------------------------------

    boolean m_machineRunning = false;
    long m_producedQty = 0;

    UIComponent m_anchor;
    SC_PANEComponent m_pane;
    SC_BUTTONComponent m_buttonStart;
    SC_BUTTONComponent m_buttonStop;
    SC_LABELComponent m_labelProducedQty;


    // ------------------------------------------------------------------------
    // constructors
    // ------------------------------------------------------------------------

    public TestDirectUpdateSC()
    {
        render();
    }

    // ------------------------------------------------------------------------
    // public usage
    // ------------------------------------------------------------------------

    public void setAnchor(UIComponent value)
    {
        if (m_anchor == value)
            return;
        m_anchor = value;
        m_anchor.getChildren().add(m_pane);
    }

    // ------------------------------------------------------------------------
    // private usage
    // ------------------------------------------------------------------------

    private void render()
    {
        m_pane = new SC_PANEComponent();
        m_pane.setRowdistance(5);
        {
            SC_ROWComponent r = new SC_ROWComponent();
            m_pane.addSCChild(r);
            {
                SC_LABELComponent l = new SC_LABELComponent();
                r.addSCChild(l);
                l.setText("Machine");
                l.setWidth(150);
            }
            {
                m_buttonStart = new SC_BUTTONComponent();
                r.addSCChild(m_buttonStart);
```

```
                m_buttonStart.setEnabled(true);
                m_buttonStart.setText("Start");
                m_buttonStart.addSCActionListener(new
ISCComponentActionListener()
                {
                    @Override
                    public void processSCActionEvent(BaseActionEvent arg0)
                    {
                        m_buttonStart.setEnabled(false);
                        m_buttonStop.setEnabled(true);
                        m_machineRunning = true;
                        (new ProductionSimulator()).start();
                    }
                });
            }
            {
                SC_COLDISTANCEComponent d = new SC_COLDISTANCEComponent();
                r.addSCChild(d);
                d.setWidth(10);
            }
            {
                m_buttonStop = new SC_BUTTONComponent();
                r.addSCChild(m_buttonStop);
                m_buttonStop.setEnabled(false);
                m_buttonStop.setText("Stop");
                m_buttonStop.addSCActionListener(new ISCComponentActionListener()
                {
                    @Override
                    public void processSCActionEvent(BaseActionEvent arg0)
                    {
                        m_buttonStart.setEnabled(true);
                        m_buttonStop.setEnabled(false);
                        m_machineRunning = false;
                    }
                });
            }
        }
        {
            SC_ROWComponent r = new SC_ROWComponent();
            m_pane.addSCChild(r);
            {
                SC_LABELComponent l = new SC_LABELComponent();
                r.addSCChild(l);
                l.setText("Produced Quantity");
                l.setWidth(150);
            }
            {
                m_labelProducedQty = new SC_LABELComponent();
                r.addSCChild(m_labelProducedQty);
                m_labelProducedQty.setText("0");
            }
        }
    }

    private void increaseProducedQty()
    {
        m_producedQty++;
        m_labelProducedQty.setText(""+m_producedQty);
    }
}
```

Also nothing special here... but: you do not see any calling of IDirectUpdate! - THe reason for this is that the SC_*-components automatically access the direct update interface if they are not updated within the context of a request-response processing,

CaptainCasa GmbH
Hindemithweg 13
D – 69245 Bammental
+49 6223 484147

www.CaptainCasa.com
info@CaptainCasa.com