

Development of

Client Components

Table of Contents

- Architecture..... 4
 - Client Side..... 4
 - Server Side..... 5
- Client Side Component Development..... 6
 - Page Element Level..... 6
 - Page Element Classes..... 6
 - Typical Cases..... 6
 - Naming Issues..... 7
 - PageElementColumn Usage..... 7
 - Component Life Cycle + Receiving of Data..... 8
 - Component sending Data back..... 9
 - General Component Properties..... 9
 - PageElement Usage..... 9
 - Adding your Component to the Client..... 10
- Server Side Component Development..... 12
 - Server Side Component Definition..... 12
 - Procedure to follow..... 12
 - Update of Server Side Data..... 13
 - These are just the Basics..... 13

This documentation explains the development of client components. This includes:

- Developing the client side component
- Making the component available on server side as part of a JSF library

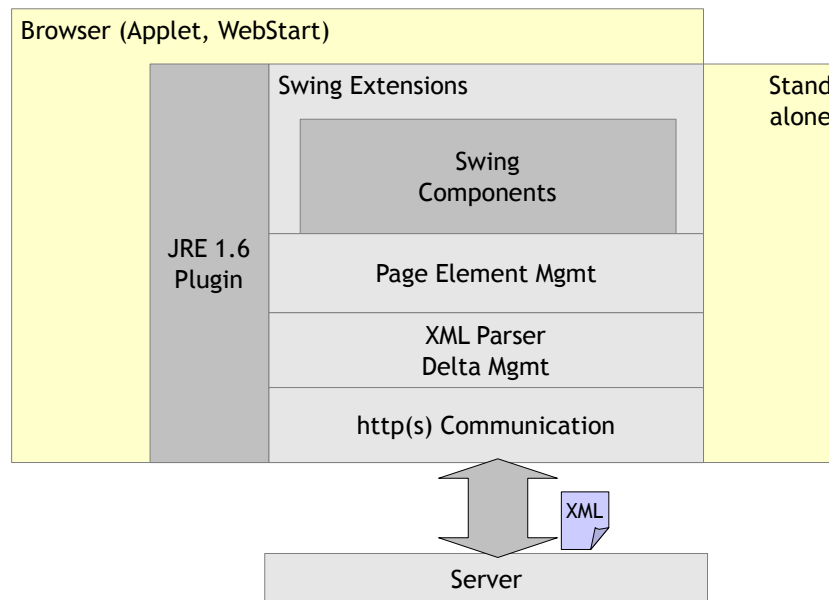
Readers are expected to have some knowledge in the following areas:

- Swing Component Development
- Java Server Faces

Architecture

Client Side

The client side architecture looks as follows:



The client side is a Java application either running in an applet container or started via Webstart or started as stand alone application.

The client consists of the following building blocks:

- **http(s) communication:** this block transfers http requests to the server and receives back an XML layout description. Along with the http request all data is passed that was changed on client side.
- **XML Parser/Delta Management:** this block parses the XML layout coming from the server and transfers changes into CaptainCasa page elements. A page element is a client side representation of a component coming as part of the XML layout definition. A page element encapsulates all graphical activities that a component provides - it is the bridge between XML processing and Swing component processing. Page elements - like the corresponding XML layout - are arrange in a tree, which is updated every time an updated layout is received.
- Each Page Element transfers its current status into corresponding Swing rendering. And the page element listens to events of the Swing components (e.g. value updates), which are then transferred back.

The most important part of the architecture is the Page Element level - this is the area where to add new components. Each page element has a well defined interface to integrate into the XML processing.

The Swing layer itself consist of the following items:

- The default Swing components
- Some enhanced components (e.g. Outlookbar) that are not available within the Swing component library.
- And, important: a layout manager that does arrange components in the way you are used to when using CaptainCasa Enterprise Client: there are containers, containers keep rows, rows keep components or containers themselves.

Server Side

On server side each component must be registered as part of a JSF component library. This means that you must provide the following items:

- The component must be registered inside the component libraries tag library definition (tld file) and inside tag libraries configuration (faces-config.xml)
- The component must provide a Component-class and a Tag-class.

Each tag library must be registered within the CaptainCasa environment so that its content can be reached during design time (your newly created component should be part of the select-able components within the Layout Editor) - and so that its content can be reached during runtime.

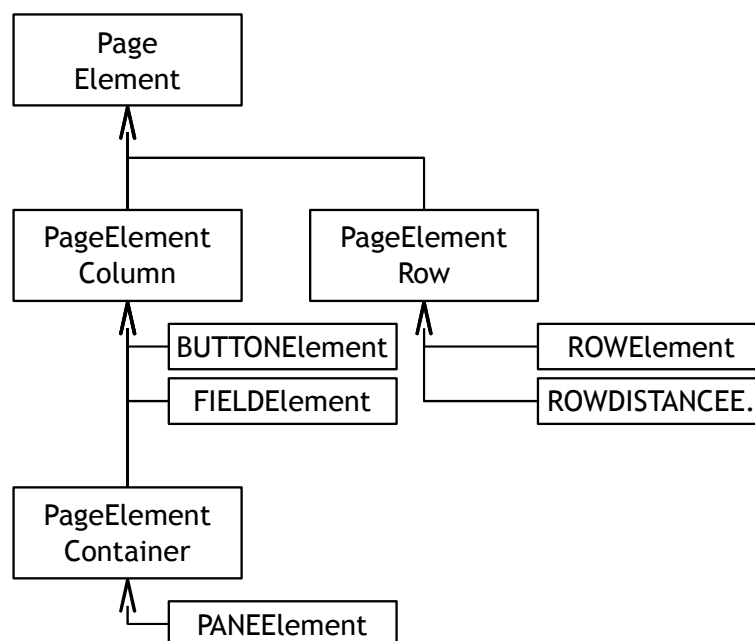
Client Side Component Development

Page Element Level

Page Elements are the counter parts to the XML elements that are transferred as layout definition from the server to the client. The client side framework maintains a page element tree which is 100% in synch with the XML layout definition that is managed on server side. This means: every time that new XML is received from the server, then corresponding updated are made inside the page element tree - page elements may be destroyed, if they are not contained in the layout anymore, they may be created if they are newly available and they are updated when there is a change of attribute values.

Page Element Classes

In order to simplify the creation of page elements and in order to properly arrange page elements within the layout management there are already some very useful page elements, that you should use for deriving your own components:



“PageElement” is the central class that all page elements derive from.

“PageElementColumn” is a page element that is associated with a Swing component that is rendered within one row.

“PageElementContainer” is a page element that is associated with a container component.

“PageElementRow” is a page element that is associated with a row component.

Please have a look into the JavaDoc documentation for detailed information about methods and properties of each class.

Typical Cases

The most typical cases are:

- You want to add a new component that you want to make available just as other components (fields, buttons). In this case you derive your page element implementation from “PageElementColumn”.
- You want to add a new component without any graphical representation. In this case you

derive your component from “PageElement”.

Naming Issues

The XML layout that is sent from the server contains XML tags and attribute values. E.g. a part of the XML may look like:

```

...
<myfield id="abcd" text="Hello" width="100"/>
...

```

There is a straight naming convention within the client processing: the class of the page element that serves the component must be named: “MYFIELDElement”, and it must be located either within the package “org.eclnt.client.elements.impl” or within the package “eclntextensions”.

PageElementColumn Usage

Let's have a look onto the implementation of the component “MYFIELD” to see the most important aspects. The MYFIELD component is a straight field - you can pass from server side a text which cab be edited by the user. Changes are sent back to the server processing.

```

package eclntextensions;
import java.awt.Component;
import javax.swing.JTextField;
import javax.swing.event.DocumentEvent;
import org.eclnt.client.controls.util.DefaultDocumentListener;
import org.eclnt.client.elements.PageElementColumn;

public class MYFIELDElement extends PageElementColumn
{
    // -----
    // inner classes
    // -----

    public class MyDocumentListener extends DefaultDocumentListener
    {
        public void changedUpdate(DocumentEvent e)
        {
            processUpdate();
        }
        public void insertUpdate(DocumentEvent e)
        {
            processUpdate();
        }
        public void removeUpdate(DocumentEvent e)
        {
            processUpdate();
        }
    }

    public class MyField extends JTextField
    {
        public MyField()
        {
            getDocument().addDocumentListener(new MyDocumentListener());
        }
    }

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

    MyField m_field;

    String m_text;
    boolean m_changeText = false;

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

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

```

```

public void setText(String value) { m_text = value; m_changeText = true; }
public String getText() { return m_text; }

public void createComponent()
{
    m_field = new MyField();
}

public Component getComponent()
{
    return m_field;
}

public void applyComponentData()
{
    if (m_changeText)
    {
        m_changeText = false;
        m_field.setText(m_text);
    }
    super.applyComponentData();
}

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

private void processUpdate()
{
    m_text = m_field.getText();
    registerDirtyInformation(getId(),m_text);
}
}

```

The important parts of the code are discussed in the following sub-chapters.

Component Life Cycle + Receiving of Data

The page element is created once a MYFIELD tag is contained within the XML layout definition that is sent from the server. The exact sequence of operations is:

- The page element instance is created. There is no specific constructor implemented in MYFIELDElement, so the default constructor is called.
- The method “**createComponent()**” is called. This method passes the signal to the page element, that it now should create the component. Pay attention: the component's attributes (i.e. all the set/get methods) are not yet called - you need to pass back a “naked” component. The component will be embedded into the current layout manager's row.
- Now all the **set/get methods** are called - passing properties that are part of the XML layout definition into the component. The set/get implementation must be of type “String” - the name of the properties must exactly match the name of the XML element attributes within the XML layout. - It is important to note that only these set/get methods are called, for which values actually have changed.
- Now the “**applyComponentData()**” method is called. This is the right point of time to pass element attributes into the component. All set/get methods have been processed at this point of time. As you see in the implementation data is only passed if there really was a change of data - managed by the different m_change* properties.

That's it!

The “set/get” phase and the “applyComponentData()” phase will now be repeated every time a layout will be received from server side.

What's about the end of the page element's life cycle? Well, the page element will be taken out of the page element tree. In general there is nothing to explicitly do within the page element itself. If you build up some dependencies which you want to clean up, there is a “**destroyElement()**” method, that you can override.

Component sending Data back

As you can see in the MYFIELD example there is some event listening to the Swing text field: when the user changes the conten of the field then a method “processUpdate()” is called. This method calls a method “registerDirtyInformation()” - which queues the changed value to be transferred to the server next time a server roundtrip is executed.

In general there are two ways to talk to the server:

- “registerDirtyInformation()” - this is the way you see in the example. Data is passed back to the server as part of the next request.
- “getPage().callServer()” - use this method to actually trigger a roundtrip to the server.

In both cases the http request that is sent to the server contains corresponding “id=value” pairs which transfer the data to the server. The id is formed out of the element’s client id (which you can get by calling “getId()” and - if required - some postfix. This sending of data needs to be in synch with the server side processing on JSF component layer: the JSF component layer is the one to decode the information from the http request.

General Component Properties

When you have a look into the page element inheritance hierarchy then you will see that a couple of properties are managed as part of “PageElementColumn” already:

- width/height: updates the width and height of your component. If not used, then the component will be sized dependent from its “getMinimumSize()” and “getPreferredSize()” response.
- flush: is set to true then data updates will immediately cause an update to the server side processing.
- background/bgpaint: these are managed automatically if your component supports the interface IBgpaint. In order to suppor this interface you just need to copy the methods that you see in the MyProgressbar-implemantaion of the example into your own component.
- rowalignmenty: aligns your component vertically within a row - you need to implement method “IAlignableInsideRow”.

PageElement Usage

In case your component does not have a visible representation then you choose the top class “PageElement” to derive your implementation from.

Have a look onto the following example: the component CLIENTCONFIG is transferring a server side definition of “country” and “language” and applies it to the client side Locale:

```
package ecIntextensions;

import java.awt.Component;
import java.util.Locale;

import org.ecInt.client.elements.PageElement;

public class CLIENTCONFIGElement
    extends PageElement
{
    // -----
    // inner classes
    // -----

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

    String m_language;
    String m_country;
    boolean m_changeLanguage = false;
    boolean m_changeCountry = false;

    // -----
}
```

```

// constructors
// -----

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

public void setLanguage(String value) { m_language = value; m_changeLanguage = true; }
public String getLanguage() { return m_language; }

public void setCountry(String value) { m_country = value; m_changeCountry = true; }
public String getCountry() { return m_country; }

public Component getComponent()
{
    return null;
}

public void invalidateLayoutSizeBuffer()
{
}

public void applyComponentData()
{
    super.applyComponentData();
    if (m_changeCountry ||
        m_changeLanguage)
    {
        m_changeCountry = false;
        m_changeLanguage = false;
        String country = m_country;
        if (country == null) country = Locale.getDefault().getCountry();
        String language = m_language;
        if (language == null) language = Locale.getDefault().getLanguage();
        Locale newLocale = new Locale(language, country);
        Locale.setDefault(newLocale);
    }
}

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

```

You see:

- There is no visible representation - as consequence the “createComponent()” and “getComponent()” implementation are kept to a minimum.
- Within the “applyComponentData()” method a new Locale is created and passed as default Locale of the client environment.

Adding your Component to the Client

Once having developed your own component you need to add it to the client processing.

The first thing to do is to package the component into an own .jar file. This .jar file needs to be added to the client's libraries.

When starting the client from a client side Java installation (i.e. no applet, no webstart, but start via “java” or “javaw”) then you just need to add the new .jar file to the classpath.

When starting the client via applet or webstart then you need to do some more steps: you need to sign the .jar file - and you also need to sign all .jar files that are coming from the default CaptainCasa delivery (eclnt.jar and - if required - swt.jar). Background: the applet or webstart processing will only accept signed .jar files, and it will only accept several .jar files if these are signed with the same signature.

Signing is done in two steps:

- You need to create one time a key using the keytool coming with your JDK:

```
%JAVA_HOME%\bin\keytool -genkey -keystore YOURKEYNAME -alias YOURKEYNAME
```

- You need to signe each .jar file with the key generated in the previous step:

```
call %JAVA_HOME%\bin\jarsigner -keystore YOURKEYNAME YOURJAR.jar YOURKEYNAME
```

Having done the signing you need to extend the library list within your applet pages (.html) or within your webstart definitions (.jnlp).

Please pay attention: signing has to be done every time when a .jar file is changed. This means: because CaptainCasa is delivering new .jar files with every release you need to re-sign your and CaptainCasa's client libraries correspondingly. - We recommend to create an ANT script to automatically execute the signing procedure.

Server Side Component Development

The server side component is the one that is responsible for...

- ...creating the XML that is passed to the client
- ...analysing a request if it contains data that comes from the client side component

This may sound complex - and indeed, there are a couple of definitions to get it working... - but it's the same as everywhere: it's simple if you the way it works. Most of the ugly tasks, such as generation of XML (note that delta-XML is sent to the frontend, which is more complicated than always sending "full XML"), are taken over by base classes you can derive from.

Server Side Component Definition

The typical situation is, that on server side you want to have a component definition which exactly matches the client side component.

This means: if the client side component is named "myfield" (=> client implementation in class MYFIELDElement), then you also have a "myfield" component on server side. On server side there is an explicit name space management by JSF, so your component will be prefixed correspondingly (e.g. "demo:myfield").

There are a couple of definitions you need to do in order to bring your component in:

- You need to define a tag library definition
- You need to register the new tag in your faces-config.xml
- You need to write your own component class and your own component tag class
- You need to register the component in the CaptainCasa editing and runtime environment.

All these steps are explained in the Developers' Guide already - please check the chapter "Adding own Components" - the only difference with new client side components is the implementation of the component class and component tag class.

Assuming that you create a server side component library in the way described in the Developers' Guide the following chapter only explains the specific thing which are relevant for a new client side component. The MYFIELD example is continued to be used.

Procedure to follow

Add the component to the control library - e.g. democontrols.tld:

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.0" xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-jsptaglibrary_2_0.xsd">
  <tlib-version>1.0</tlib-version>
  <short-name>democontrols</short-name>
  <uri>/WEB-INF/democontrols</uri>

  ...
  ...

  <tag>
    <name>myfield</name>
    <tag-class>democontrols.MYFIELDComponentTag</tag-class>
    <attribute><name>id</name></attribute>
    <attribute><isinput/><name>text</name></attribute>
    <attribute><name>width</name></attribute>
  </tag>

  ...
  ...
</taglib>
```

Register the component in the controlsarrangement.xml file:

```
<controlsarrangement>
  ...
  <tag name="t:row" below="demo:myfield"/>
  <tag name="demo:myfield" folder="Demo Extensions"/>
  ...
  ...
</controlsarrangement>
```

Register the component library in the “eclntjsfserver/config/controllibraries.xml” and within your project definition.

All this, up to now is exactly the same procedure as explained in the Developers Guide already. Now you need to implement the server side component and component tag classes: the implementation now looks much simpler:

```
package democontrols;

import org.eclnt.jsfserver.elements.BaseActionComponent;

public class MYFIELDComponent
    extends BaseActionComponent
{
}
}
```

...and...

```
package democontrols;

import org.eclnt.jsfserver.elements.BaseComponentTag;

public class MYFIELDComponentTag
    extends BaseComponentTag
{
}
}
```

This time (compared to the composite components explained in the Developers' Guide) there is no need for assembling other components within the Component-implementation. The server side components just pass their content into corresponding XML - and all this is managed by the base classes you derive your implementation from.

Update of Server Side Data

There is one default framework which is included in the base class's implementation as well: when the client sends a request that contains a name/value pair, in which the name is matching the component's id, then the value is transferred into the value binding of the attribute that is marked with “<isinput/>” within the tag library definition.

This sounds complex...: but, have a look onto the MYFIELD example:

- In the .tld the attribute “text” is marked with “<isinput/>”
- If the user of the component binds an expression “<myfield text="#{d.xyz.firstName}"/>” then the corresponding set-method will be called when the data is changed.

Have a look onto the client component's implementation (class MYFIELDElement): a dirty value is registered in the following way:

```
private void processUpdate()
{
    m_text = m_field.getText();
    registerDirtyInformation(getId(),m_text);
}
```

That's exactly the way the data transferred to the server: as name/value pair, with the component's id as value.

These are just the Basics...

This chapter only explained the straight forward use case of server side component development. In case you want to do more on server side: check the JSF documentation and apply it to your scenario.