

CaptainCasa Enterprise Client

# Developers' Guide



# Table of Contents

<b>About this Guide</b> .....	<b>12</b>
Copyright Notice.....	12
<b>Project Setup</b> .....	<b>14</b>
Basics.....	14
Project Structure.....	15
Default Project (no “compact”, no “hot deployment”).....	15
Compact Project (no “hot deployment”).....	17
Hot Deployment Project.....	18
Other Project Structure.....	19
Location for Storing the Project Configuration.....	19
Project file of the Enterprise Client Toolset.....	19
Basics.....	19
Most significant Attributes.....	20
Other Attributes.....	21
Deployment Configuration.....	21
Referencing special Values.....	21
Integrating an existing Project into the CaptainCasa Toolset.....	21
Template Management.....	22
<b>Principles of Development</b> .....	<b>24</b>
The Layout.....	24
The Managed Bean.....	25
How Objects are created on Server Side.....	27
Resolving of Expressions.....	27
The “#{d.” Fragment.....	27
The “.AddressDetailUI.” Fragment.....	27
The “.firstName}” Fragment.....	28
Resolution is done with every Request!.....	28
Viewing the Page outside the Toolset.....	28
Java Application Access.....	29
Dispatcher Concept.....	29
Some Details about the default Tomcat Configuration.....	31
<b>Working with Components</b> .....	<b>32</b>
Types of Components.....	32
Container - Row - Column.....	32
Conventions.....	32
Non Graphical Components.....	32
Rules which apply to all Components.....	33
Component Sizing Aspects.....	33
Action Listener.....	33
Flush Management (Component Level).....	34
Flush Management (Container/Row Level).....	34
Background Painting (BACKGROUND and BGPAINT).....	34
Focus Management - Setting Focus to dedicated Component.....	35
Focus Management - Defining the Focus-/Tab-Sequence.....	37
Attribute TRIGGER.....	39
Special Container / Row Components.....	40
COLSYNCHEDPANE, COLSYNCHEDPANEROW - Connected Columns.....	40
ROWFLEXLINECONTAINER - A Row with Line Breaks.....	41
<b>Grid Components</b> .....	<b>43</b>
FIXGRID Basics.....	43
FIXGRID, GRIDCOL, GRIDHEADER, GRIDFOOTER.....	43
Data Binding.....	44
Event Binding.....	46
Rendering Aspects.....	46
Columns Sizing Aspects.....	46
...some more info about “Server side number of Items” versus “Client Side number of Items”.....	47
Tree Processing.....	48

Column Sequence and Column Sizing.....	51
Column Sequence.....	51
Column Widths.....	52
Persistence Management with FIXGRIDs.....	52
Column Sorting.....	52
The straight forward Way.....	52
If SORTREFERENCE is undefined.....	55
Directly setting the Sort Status.....	55
Influencing the default Sorting.....	56
Implementing an own Sort Algorithm.....	56
Multiple Column Sorting.....	57
Techincal Info behind.....	57
Usage from Server Side Java.....	57
Switching Off Multiple Column Sorting.....	58
Focus Issues.....	58
Setting the Focus into Grid Line.....	58
Avoiding Line-Selection on Focus.....	59
Extended Grid Functions.....	60
Column Sequence.....	60
Using the Columns Sequence Popup.....	60
Search & Export.....	61
Using the Default Popup.....	61
Using the APIs.....	62
“Internal” Details.....	62
Using own Page Definitions.....	63
Using own Page Beans.....	63
Row Detail Popup.....	63
Grid Popup Menu.....	64
Saving and Restoring a Grid's Runtime State.....	64
Implicit or explicit Storing.....	65
Interface IFIXGRIDPersistence2.....	65
Performance Optimization of Grids.....	66
Grid Performance.....	66
“Change Index” Optimization.....	66
“Change Index” Optimization Scenarios.....	67
“Data Bag” Optimization.....	67
<b>REPEAT Component.....</b>	<b>69</b>
Usage of the REPEAT Component.....	69
Nesting REPEAT Components.....	71
Performance Considerations.....	72
In general.....	72
Server Side Scrolling.....	73
<b>Page Navigation.....</b>	<b>76</b>
Basics.....	76
JSF Navigation Concepts.....	76
Enterprise Client Concepts.....	76
Page Inclusion vs. Page Bean Inclusion.....	76
Consequences.....	78
Page Bean Modularization - Example.....	78
Page.....	78
Java Code.....	79
JSP Page.....	81
Page Bean Details.....	81
IPageBean / PageBean Instances.....	82
The ROWPAGEBEANINCLUDE Component.....	82
What happens internally.....	83
Page Bean Navigation.....	83
Page Navigation.....	83
Popup Management.....	85
Page Bean Patterns.....	87
Performance Optimization.....	87
Preconfigured Popups.....	88
OK-Popup.....	88

Yes/No Popup.....	89
Value Selection Popup with COMBOFIELD.....	89
Overview.....	89
Preconfigured Value Help Popups.....	90
Type Ahead Management.....	90
<b>Working with Menus.....</b>	<b>91</b>
Example Reference.....	91
MENUBAR.....	91
POPUPMENU.....	91
Usage of POPUPMENU.....	91
Event Reaction.....	92
Event Reaction on MENUITEM Level.....	93
Event Reaction on Component Level.....	94
Finding the right POPUPMENU.....	94
Hotkey Definitions.....	94
Addendum - Hotkey in Buttons, Icons, etc.....	95
Dynamic Menu Definitions.....	95
Dynamic POPUPMENU - attribute POPUPMENULOADROUNDTRIP.....	95
<b>Working with Drag &amp; Drop.....</b>	<b>98</b>
Example Reference.....	98
Details.....	98
Attribute DRAGSEND.....	98
Attribute DROPRECEIVE.....	98
Event Processing in the Server Side Java Code.....	98
Information associated with Drop Event.....	99
Controlling the Component Highlighting during Drag & Drop.....	99
<b>Working with Shapes.....</b>	<b>101</b>
Components.....	101
Example.....	102
Getting into more complex Scenarios.....	102
<b>Working with Managed Beans.....</b>	<b>104</b>
Basics.....	104
Dynamic Properties.....	104
Usage of Maps.....	104
Usage of Collections / Arrays.....	105
Dynamic Properties - Dynamic Bean Browser.....	105
Data Type Considerations.....	105
...without doing anything special.....	105
...with explicitly passing back type information.....	106
Property Binding within Grid Processing.....	106
Accelerated Property Access.....	106
Method Binding.....	107
Basics.....	107
One method, various events.....	108
Tool support.....	109
Adapter Binding.....	109
Purpose.....	109
Basic Idea.....	109
Definition of Adapter Objects.....	110
What happens at Runtime.....	110
Example.....	110
Overriding Adapter Binding.....	112
Related Information.....	113
Exception Management.....	113
Default Handling of Exception.....	113
Additional Functions on top of JSF.....	113
Consequences for Implementation.....	114
Customizing the Server Side Error Screen.....	115
Best Practice: Checking if your whole Application is available.....	116
Property Change Notification.....	116
Interface IPropertyResolverAware.....	116
Interface IPropertyResolverAware2.....	116

Interface IPropertyValueConverter.....	117
Interface IValueBindingListener.....	117
Class CascadingValueBindingListener.....	118
JSF Phase Management.....	119
Overview.....	119
Starting Runnables to be executed in a certain Phase.....	120
Adding some BEANPROCESSING Statements into the Page.....	121
Http Session Management.....	123
Management of HTTP Sessions.....	123
Accessing Http Session.....	123
Reacting on Closing of Http Session.....	124
Closing the Session when Client is closed.....	124
Define Name of Session Id encoded into URLs.....	125
Configuring the Session Timeout.....	125
Complex Expressions.....	125
Expression Replacements.....	126
Consequence.....	127
<b>Dynamic Page Layout.....</b>	<b>128</b>
Overview.....	128
“By Page”.....	128
“Within a Page”.....	128
“By Page”.....	129
Building your Dynamic Page Provider.....	129
Registering your Page Provider.....	130
Pay Attention - Buffering!.....	130
“Within a Page” - Integrating XML Layout Definition dynamically.....	130
ROWDYNAMICCONTENT Component.....	130
Details on ROWDYNAMICCONTENTBinding.....	133
Using “concrete Classes” for assembling ComponentNode Instances.....	133
Details on managing Ids.....	133
Which way to go: the XML-way - or the tree-of-nodes-way?.....	134
Automated Update of Expressions.....	134
ROWDYNAMICCONTENT Component <==> DYNAMICCONTENT Component.....	135
“Within a Page” - Working with the Component Tree.....	135
Basics.....	135
Attribute COMPONENTBINDING and how to use.....	136
Points of Time, when the setter for COMPONENTBINDING is called.....	138
Pay Attention: setId(...)......	139
Pay Attention: Workplace Management and Expressions.....	139
One Comment for Eclipse Users.....	139
<b>Style Management.....</b>	<b>140</b>
Overview.....	140
JavaFX CSS Definitions.....	140
Style Definition Files.....	140
Style Variants.....	141
Defining Styles that extend other Styles.....	142
Defining Tag Variants referring to other Tag Definitions.....	143
Selecting a Style.....	143
Expressions as Style Attribute Values.....	144
Default Style.....	144
Selecting the Style by URL Parameter.....	144
Style Manager API.....	144
Creating Style Definition Files out of Page Layout.....	145
Usage Detail: Extending the generated Style from another Style than “default”.....	147
JavaFX Client: Link to CSS Management.....	147
JavaFX Styling.....	147
JavaFX-CSS Style Management within CaptainCasa Enterprise Client.....	148
Cascading Style Sheet Assignment.....	148
Style Class Assignment.....	149
Cascading Style Class Assignment.....	150
Style Class Assignment via Style Variant.....	150
Rounding.....	150
Additional FX Client Information in fx.xml.....	151
Defining own Styles.....	151
Style “default” <=> Style “defaultfx”.....	151

Swing Client: Setting Client Font Family.....	152
Working with transparent Colors.....	154
Concept.....	154
Definition and Usage of transparent Colors.....	154
Central Background Definition.....	155
Central Background Definition for Popup Dialogs.....	155
Using the BGPAINTE Attribute.....	155
“mandatory” and “error”.....	156
“rectangle”.....	156
“roundedrectangle”.....	156
“image”.....	156
“heximage”.....	157
“scaledimage” and “scaledheximage”.....	157
“write”.....	157
“writeifempty”.....	158
“oval”.....	158
“border” and “roundedborder”.....	158
“line”.....	158
“nodisabled”.....	158
“Flat” and “Undecorated” Style when using Webstart.....	158
Client Parameters.....	158
SYSTEMICON Component.....	159
ROWTITLEBAR and PANE Component: attribute ISWINDOWMOVER.....	159
Detail Configuration: some more Client Parameters.....	159
<b>Internationalization Issues.....</b>	<b>160</b>
Client Locale <=> Server Locale Settings.....	160
Internationalization Parameters for the Client.....	160
Updating the Parameters at Runtime.....	161
Http-request Parameters.....	161
Orientation and Images.....	161
Orientation and Alignment Definitions.....	162
Scaling of Images.....	162
Update the Client Settings from Server Side.....	162
Language Management within your Application.....	163
Setting Server Side Locale.....	163
Accessing Literal Translations - Basics.....	163
The default way: using built in Resource Management.....	164
Doing it on your own...!.....	165
Synchronizing Client Side Locale and Server Side Locale Settings.....	168
Scenario: Client Settings dominate Server Settings.....	168
Scenario: Server Settings dominate Client Settings.....	168
Management of Dates (and Times).....	168
Updating Client side Literals.....	169
Client side Interface.....	169
Server side Interface.....	170
<b>Online Help (“F1 Help”).....</b>	<b>172</b>
Assignment of HELPIDs.....	172
The Default Framework.....	172
Plugging in an own Framework.....	173
Interface IOnlineHelpProcessor.....	173
Default Implementation “OnlineHelpProcessorJBrowser”.....	173
Old Default Implementation “OnlineHelpProcessor”.....	174
<b>Workplace Framework.....</b>	<b>176</b>
Basics.....	176
Purpose.....	176
Example - The Demo Workplace.....	176
Terms.....	177
Development Effort.....	178
Creating your first Workplace.....	178
Dispatcher Concepts.....	178
The Dispatcher so far.....	178
The Dispatcher in the Workplace Context.....	178
Addressing the correct Dispatcher.....	180

Accessing the Workplace Environment from your Bean.....	180
<b>The Workplace API.....</b>	<b>181</b>
IWorkpageContainer.....	181
WorkpageStartInfo.....	181
Core Functions.....	182
Special Usage - Starting Functions.....	182
IWorkpage / Workpage.....	183
Start Parameters.....	183
Workpage Life Cycle Aspects - Closing of a Workpage.....	184
Workpage Lifecycle Aspects - Additional Information.....	185
Inter Workpage Eventing.....	185
Addon Components in the Workplace Management.....	187
<b>Workplace Perspective Management.....</b>	<b>188</b>
What the Workplace Perspective Management does.....	188
The WORKPLACE Page.....	189
Configuring a Perspective.....	191
Configuring multiple Perspectives for multiple Users.....	192
Switching between different Perspectives.....	192
Loading the Workplace Perspective when changing the User.....	193
Workplace Perspective - Low Level API.....	194
<b>Workplace Functions Management - ...by declaration.....</b>	<b>194</b>
Creation of Page to hold Function Tree(s).....	195
Set up Function Trees.....	196
Assign Function Tree to User.....	196
<b>Working with Macros.....</b>	<b>197</b>
Example Use Case.....	197
Macro Definition.....	197
Macro Definition by XML.....	198
Macro Definition by Class Implementation.....	198
Pay Attention when processing Grid Cells.....	199
Macro Usage.....	200
Some more Details.....	200
Overriding a Macro Value.....	200
Tolerant Attribute References.....	200
Macros in the Layout Editor.....	201
The REFERENCE Attribute.....	201
Macros within the Grid Processing (FIXGRID).....	202
Related Topics.....	203
<b>Adding own Components.....</b>	<b>204</b>
Adding Composite Components.....	204
Component <=> Component Library.....	204
Composite Component "Artifacts".....	204
Create the "tld file".....	205
Update the "faces-config.xml".....	206
Implement the component.....	206
Update the "controllibraries.xml".....	207
Add "controlsarrangement.xml".....	208
Result.....	208
Issues when creating new Components.....	209
Using Hot Deployment? - Pay Attention!.....	209
Add the Component Source into the right Source Directory.....	209
Check Project Definition.....	209
Commonly used Attributes.....	209
Some useful methods of Components.....	210
Expressions - and how to update in the correct Way.....	210
Component Attributes - "Mandatory, Preferred - All, Prproposed Values".....	211
Flexible Composite Components and the Grid.....	212
Restarting/Reloading the Toolkit.....	212
Example: "FLEXFIELD" Component.....	212
Exmample Screen.....	212
Component Implementation.....	215
<b>Page Bean Components.....</b>	<b>218</b>
Basic Idea - One self-containing JAR File.....	218
Example.....	218

Using a Page Bean Component.....	219
Component PAGEBEANCOMPONENT.....	219
Developing a Page Bean Component - By Example.....	220
The basic Files.....	220
The XML Layout.....	220
The Java Code.....	220
The Property Files.....	223
The Configuration File.....	223
Packaging and Delivering a Page Bean Component.....	223
<b>File Download &amp; File Upload.....</b>	<b>224</b>
File Download.....	224
Static URLs - Static Content.....	224
Dynamic URLs - Dynamic Content.....	225
Big Dynamic Content Scenarios.....	226
Synchronous - Asynchronous.....	226
Opening the File at Client Side.....	226
File Upload.....	227
Synchronous Upload.....	227
Asynchronous Upload.....	228
Asynchronous Upload - Notification of "Finished"!	230
Garbage Collection Issues.....	230
Special Directory "\${local}/" or "{local}/" - Automated Download.....	231
Special Directory "\${temp}/" or "{temp}/".....	232
Other Components around Client Side File Management.....	232
FILECHOOSER - Selecting + passing the Name of a Client Side File.....	232
FILECREATEDIRECTORY - Creating a directory on Client Side.....	232
Configuration Issues.....	232
Tomcat.....	232
<b>Touch Input.....</b>	<b>234</b>
Components.....	234
Definition of own Keyboard Layouts.....	235
Definition of keyboard layouts.....	235
Navigation between keyboard layouts.....	236
<b>Server-side Events, long Server-side Operations.....</b>	<b>237</b>
Introduction.....	237
Default: Request-Response driven Communication.....	237
Requirement for enhanced Communication.....	237
Pay Attention - Thread-Complexity, J2EE Rules.....	238
Constant Polling - The TIMER way.....	238
Long Polling - True, synchronous Event Coupling.....	239
Long Polling - The "Comet-Way".....	243
The normal, "traditional" way - One waiting thread per Long Polling.....	243
The Comet Way - No waiting thread per Long Polling.....	243
Using the Comet Way.....	243
Comet Disadvantage - Only Tomcat...!.....	244
The up to date "Servlet 3.0" way - No waiting thread per Long Polling.....	244
Long Operations with Messages.....	245
<b>Integration Scenarios.....</b>	<b>247</b>
HTML Page Integration (static).....	247
Libraries to be included.....	247
Passing Parameters into the Server Side Application.....	247
"Special Parameters" since Java 1.6 Update 10 - Important!.....	249
Accessing the Applet Context.....	250
HTML Page Integration (dynamic).....	250
Defining the Client Libraries to be loaded via URL Parameter.....	250
Centrally defining the "cclibs"-Parameter.....	251
Centrally defining the Client-Parameters.....	251
Specifying a Template File by URL Parameter.....	251
Specifying the default Template File by Configuration.....	252
Passing additional URL Parameters.....	252
Passing additional URL Parameters that are always Part of the Request.....	253
Webstart Integration.....	253

Static JNLP Definition.....	253
Dynamic Definition.....	253
Webstart / JNLP: the “codebase” Parameter.....	254
Passing Cookies.....	255
Closing the Client.....	255
<b>Embedding Applet into existing HTML Application.....</b>	<b>256</b>
Embedding the Applet + Session Considerations.....	256
JavaScript Integration.....	259
Security Issues.....	262
Opening HTML Pages from the CaptainCasa Enterprise Client Applet.....	263
<b>Embedding the CaptainCasa Client into Portals.....</b>	<b>263</b>
Overview.....	263
Generic Portlet “PortletIntegrator”.....	264
Configuration via portlet.xml.....	266
Accessing Portal Information from CaptainCasa JSF Processing.....	267
Pay Attention...!.....	267
<b>Starting Browser Window out of CaptainCasa Client.....</b>	<b>267</b>
<b>Swing Integration.....</b>	<b>268</b>
Page not coming up properly?.....	270
Passing Parameters to the Server side Application at Startup.....	270
Exchanging Data with the Server side Application at Runtime.....	270
CLIENTMETHODRECEIVER Component.....	271
CLIENTMETHODCALLER Component.....	271
<b>ACTIVEX Integration.....</b>	<b>271</b>
<b>Integrating Client Devices by Usage of Client Side http.....</b>	<b>273</b>
<b>Integration of Client Devices by Usage of Serial Interface.....</b>	<b>273</b>
<b>Integration of Client Devices by Usage of generic Interface “ISubDevice”.....</b>	<b>274</b>
Component CLIENTSUBDEVICE, Interface ISubDevice.....	274
Example.....	275
Deployment.....	278
<b>Integration of HTML Pages with embedded JavaScript.....</b>	<b>278</b>
<b>PDF Document Integration.....</b>	<b>280</b>
<b>Client-Id Management.....</b>	<b>281</b>
Defining and Storing the Client Id in the temporary Directory of the Client.....	281
Usage of Cookies.....	281
Accessing the Client-Id on Server Side.....	282
Storing dependent Data.....	283
<b>Client-InstanceId Management.....</b>	<b>283</b>
<b>Using “Hot UI Deployment”.....</b>	<b>284</b>
Overview + When to Use.....	284
Hot UI Deployment Framework Details.....	285
Resolution of Managed Beans.....	285
Configuration.....	286
(Eclipse) Project Configuration.....	286
Design Time <=> Runtime.....	287
<b>Security Issues.....</b>	<b>288</b>
The JSESSIONID appendix.....	288
Basics on session-id management and risks involved.....	288
Solution.....	288
Avoiding the Replaying of http - CLIENTSECID.....	289
<b>Developing plain HTML Pages.....</b>	<b>290</b>
Positioning.....	290
HT* Components.....	290
Writing HTML Pages.....	291
Writing the Code behind the Page.....	292
Styling.....	293
Default Style.....	293
Statically exchanging the default Style.....	293
Dynamically assigning the Style.....	293
Individual styling.....	293
Special HT* Components.....	293

Component HTJSCALL.....	294
HTJSCALL - Only use once!.....	296
HTJSCALL - Darkening the Page from your Code.....	296
<b>Selected Topics.....</b>	<b>297</b>
Configuration.....	297
Definition of Error Screen for Server-side Errors.....	297
Definition of Error Screen for Client-side Errors.....	298
<b>Appendix - Stream Store Persistence.....</b>	<b>300</b>
Stream Store API.....	300
Where is the Data stored?.....	301
File based Persistence.....	301
JDBC based Persistence.....	301
Design Time vs. Run Time Data.....	302
<b>Appendix - Component Overview.....</b>	<b>303</b>
<b>Appendix - Component Attribute Reference.....</b>	<b>310</b>
Attribute Value Formats.....	310
Semicolon separated List.....	310
Method Value.....	311
Value List.....	311
Color Values.....	312
Some Information about some Attributes.....	312
action.....	312
actionListener.....	312
background.....	312
bgpaint.....	312
border.....	313
contenttype.....	313
enabled.....	313
image.....	313
padding.....	314
page.....	314
stylevariant.....	314
tooltip.....	314
<b>Appendix - Dynamic Introspection within the Bean Browser.....</b>	<b>315</b>
Method “introspectDynamically”.....	315
Parameters of “introspectDynamically”.....	316
“references”.....	316
“pathList”.....	316
<b>Appendix - Non Page Bean based Navigation.....</b>	<b>318</b>
The ROWINCLUDE Component.....	318
Navigation Example.....	318
The most outside Page.....	319
The logon Page.....	320
The Dispatcher Bean.....	321
The Workplace Page.....	322
Nesting Pages - Conclusion.....	323
ROWINCLUDE - Extended Usage.....	323
ROWINCLUDE - Important Note!.....	325
Popups.....	325
Modal Popups.....	325
Modeless Popups.....	326
Popup Closing.....	326
Popup Decoration.....	326
Popup Data Synchronization.....	327
Popup Background Style Settings.....	327
Writing generically usable Popups.....	327
Sizing Popups.....	329
Positioning Popups.....	329
<b>Appendix - Starting Enterprise Client Pages.....</b>	<b>330</b>
Three Possibilities.....	330
Applet.....	330
Web Start.....	331

Command Line.....	331
Usage with .ccapplet and .ccwebstart.....	332
Passing Client Parameters.....	332
List of Parameters.....	332
Defining Client Parameters to be copied into http-Header.....	339
Defining Client Parameters to be transferred into cookies.....	340

# About this Guide

This guide tells you how to develop own screens with CaptainCasa Enterprise Client.

Please note that there are a couple of examples that are delivered with the Enterprise Client Installation. Open the “Demo Workplace” to view the examples. For every control there is at least one example showing how to use it. The demo web application contains all the Java source files (“workplacesrc” directory) and all the JSP files (“workplace” directory)

For this reason this guide will only summarize the principles of working with certain components - the details are contained in the examples.

Before reading this guide it is useful to read the “Tutorial - First Development Steps with CaptainCasa Enterprise Client”. The tutorial explains step by step how to create the first project and the first screens.

---

## Copyright Notice

All intellectual property that is contained in CaptainCasa Enterprise Client belongs to CaptainCasa GmbH. You must not copy or decompile any part of CaptainCasa Enterprise Client without the explicit agreement of CaptainCasa GmbH. Certain CaptainCasa license types grant the “full access” to the sources - please contact [info@CaptainCasa.com](mailto:info@CaptainCasa.com) for more information.

A usage of CaptainCasa Enterprise Client in a productive environment and a redistribution is only allowed with obtaining an official license from CaptainCasa GmbH. Various types of licenses are available, including the so called “binary license” which allows the free usage and free redistribution as part of your application. Contact [info@CaptainCasa.com](mailto:info@CaptainCasa.com) for obtaining detailed license information.

The “binary license” of CaptainCasa Enterprise Client is provided WITHOUT WARRANTIES.

CaptainCasa Enterprise Client makes use of the following contained libraries / software products:

- Java Runtime Environment JRE 1.6 from Sun Microsystems. Copyright and license information is available in the directory <installdir>/server/jre.
- Tomcat Engine by Apache Group. Copyright and license information is available in the directory <installdir>/server/tomcat.
- Some icons in the tools and the demos were taken from <http://www.famfamfam.com/lab/icons/silk/>.
- JasperReports is included as an optional client component - which is available via the GNU Library or Lesser General Public License (LGPL), see <http://sourceforge.net/projects/jasperreports/> for more details
- SimplyHTML is included as an optional client components - which is available via the GNU General Public License. License information is available via <http://simplyhtml.sourceforge.net/> and is available in the about-box of the corresponding editor component as well. - Minimal code changes were done on base of the simplyhtml-package having to do with tailoring the editor to be used as component outside the scope of file processing. The modified source can be obtained any time by sending a mail to [info@CaptainCasa.com](mailto:info@CaptainCasa.com).
- The demos are using the library JFreeChart, details on licensing are available here: <http://www.jfree.org/jfreechart/>

- The client side PDF renderer is using the JPedal library for converting PDF documents into corresponding preview images. The library is using the LGPL license, please find details here: <http://sourceforge.net/projects/jpedal/>

# Project Setup

Please read the documentation “Tutorial - First Development Steps with CaptainCasa Enterprise Client” in order to step by step create a CaptainCasa project and to learn how to execute first development steps.

Developing with CaptainCasa means that you set up projects in which you manage layout definitions, resources (images) and your server side code (Java). When creating a project within the CaptainCasa toolset there are various options - this chapter explains the structural aspects behind.

---

## Basics

The server side of CaptainCasa Enterprise Client is based on the default J2EE web application concept:

Somewhere you have a directory which represents the web content. The directory structure of this web content looks like this:

```
</web application>
  /...
  /images
    icon1.png
  /META-INF
  /WEB-INF
  /lib
    *.jar
  /classes
    Page1UI.class
    Page2UI.class
  web.xml
  page1.jsp
  page2.jsp
```

The structure is defined by the J2EE-Servlet specification. It is used for building so called .war files, which are the ones to be deployed into various application server environments.

Inside the web application you provide the following artifacts:

- Page definitions: these are .jsp files which themselves hold the JSF components. Please do not get confused by the name “.jsp”, which sounds a bit old-fashioned! The content of the .jsp files is pure XML. No scripting is allowed!
- Compiled Java code: pages access corresponding code, e.g. a button in a page is bound to some method (“action listener”) in a corresponding Java program. The compiled code is kept as .class or .jar file, typically within the WEB-INF-directory.
- Java libraries: your code accesses Java libraries. For example the JSF libraries are required with CaptainCasa. Or: CaptainCasa itself provides a jar-library for the server side, e.g. containing the component implementation for all the graphical components that are provided.
- Resources like images, texts, etc. which are contained in any directory folders

You see: the directory structure which is used at runtime by the servlet container (e.g. Tomcat) contains various files: some authored by you (your jsp-pages, your code), some coming from CaptainCasa and some coming from other parties.

The CaptainCasa toolset together with any development environment takes care of authoring your artifacts and deploying them into a directory structure that is the one shown above.

## Project Structure

The project directory structure on the one hand is somehow related to the deployment directory structure - but of course on the other hand differs. While the purpose of the deployment directory structure is to “melt everything together” in order to efficiently deploy, the purpose of the project directory structure is to clearly separate your own artifacts (the ones to check-in/out to/from a version control system), the compiled results of your artifacts and the artifacts from other parties.

When creating a project by using the CaptainCasa toolset there is a configuration dialog at the very beginning, in which you define the project layout:

**Project Definition**

All the project files are kept inside one directory, the project's root directory. The files include: pages (".jsp"), sources (".java"), images, test cases and other resources.

Project Name   
e.g. "myproject"

Project Root Directory   
e.g. "c:\development\projects\myproject"

Store project configuration in project directory

**Hot Deployment**  Use Hot Deployment

By using hot deployment you separate your server side classes into these ones residing inside WEB-INF/classes and these ones which are loaded by a hot-deploy classloader.

**Project Structure**  Release 5  Compact  
 Release 4

The default layout of the project directory structure was updated from release 4.0 to release 5.0 of CaptainCasa Enterprise Client. With release 5.0 there is a much clearer separations between source files (the ones to add to version control) and "other" files. - In addition you may define within the 5.0 structure, not to take over the CaptainCasa addons for a web application into your project. In this case define "Compact" correspondingly.

**Deployment of Project's Web Application**

At defined points of time the project is deployed, e.g. into a Tomcat environment. The deployment is required to view and test pages within the CaptainCasa tooling environment.

Deploy Directory   
Deploy host/port

These two parameters are derived from the current Tomcat/ installation environment and only need to be changed if not working within the standard installation.

The important parameters are:

- Name of the project and its root directory
- The information if to use “Hot Deployment” or not
- The information how to store the project configuration
- The information if to define a “compact” project

### Default Project (no “compact”, no “hot deployment”)

When creating a project within the CaptainCasa toolset and doing nothing else than specifying the project's name and the project's directory then the project structure looks as follows:

```
<project directory>
```





What is the advantage?

- The project is much smaller now. CaptainCasa includes all the “heavy libraries” (jsf- and jsf-related jar-files) and quite a lot of resources (like default pages and images) that are now not copied anymore into each project.

What is the disadvantage?

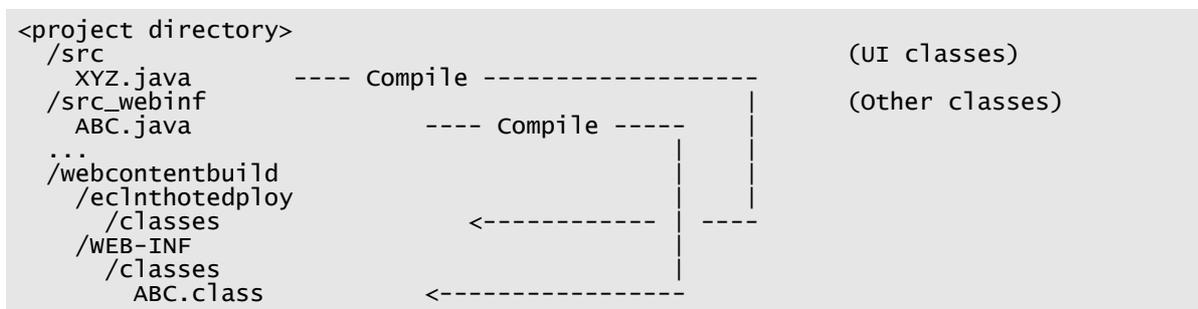
- The project is not self containing anymore.
- Example: with explicitly keeping the CaptainCasa addons within “/webcontentcc” it is possible to use different versions of CaptainCasa within your projects. Now, with using the central CaptainCasa addons, every project is using the installed CaptainCasa version.

When should you use “compact” projects?

- You should think about using compact projects when due to any reason you work with a lot of projects simultaneously. “A lot of” is somewhere in the area of “>20”.
- We know about situations in which projects are defined on a very fine granular level (some pages only) and in which later on applications are bundled out of the projects. Of course this requires some extra management in the area of deployment and of course this only makes sense for dedicated scenarios - but if it makes sense, then the usage of “compact” projects makes a lot of sense.

## Hot Deployment Project

Now, let's briefly talk about the “hot deployment” option that you have when creating projects. When using “hot deployment” then the Java-sources of your project are managed within two directories: “/src” and “/src\_webinf”. The one directory compiles the sources to “/webcontentbuild/eclnhotdeploy/classes”, the other directory compiles the classes to “/webcontentbuild/WEB-INF/classes”:



At runtime the classes in “/WEB-INF/classes” are loaded once a web application is loaded. When changing the classes (due to re-deployment) then the whole web application is required to be shut down in order to re-load them.

The classes in “/eclnhotdeploy/classes” can be reloaded by the CaptainCasa toolset within a running web application. There is no need to re-load the web application.

Hot deployment significantly speeds up the development in many scenarios. Typically, when (re-)loading a web application there is quite a lot of initialization done within the application - e.g. an application might use Hibernate as persistence framework, so Hibernate checks if a database is still in sync with the application when being started. As consequence a re-load of a web application is a quite heavy operation, which you do not want to execute just because of changing some UI classes.

In the hot deployment project these classes that are UI related are managed in the “/src” directory. And these classes that are not UI related, or that require to be run within the class loader of the web application are managed in the “/src\_webinf” directory.

There is an own chapter “Hot Deployment” in which you can find more detailed information on hot deployment.

## Other Project Structure

All the project structures that you have read about so far, are not implemented in a hardwired way within the CaptainCasa toolset. Per project there is an xml configuration file that explicitly defines where the toolset can find what information - in order to properly access files (e.g. JSP-pages) and in order to correctly deploy.

As consequence you can setup any project structure on your own and guide the CaptainCasa toolset what to do using the project configuration file.

The structure of this file is explained in the following text.

## Location for Storing the Project Configuration

Per project the CaptainCasa toolset stores the project configuration within an XML file. This file can be either stored in the root directory of the project that you create. Or it can be stored within the directory structure of the CaptainCasa toolset.

We in general recommend to use the default option (“Store project configuration in project directory”): the project file in most cases is some information to be checked in/checked out into some source management system (Subversion, CVS, ...) to be shared between different developers.

---

## Project file of the Enterprise Client Toolset

### Basics

The project file is generated automatically when you create the project using the Enterprise Client toolset.

There are two locations in the file system where the project file is stored:

If having selected option “Store project configuration in project directory” when having created the project, then the project file is located directly within the project directory. Within the toolset there is an XML file that points to the project directory, so that the toolset knows where to look for the project file:

```
Project file in project folder:
<projectdirectory>
  .ccproject ==> contains project configuration

Link to project file in CaptainCasa toolset:
<CaptainCasa Install Directory>
  /tools
    /embeddedservlet
      /webapps
        /editor
          /config
            ...
            <project>.xml ==> contains link to <projectdirectory>
            ...
```

If having deselected option “Store project configuration in project directory” when having created the project, then the project file is directly stored within the toolset:

```
<CaptainCasa Install Directory>
  /tools
    /embeddedservlet
      /webapps
        /editor
```

```

/config
...
<project>.xml ==> contains project configuration
...

```

Inside the project file, there are a couple of parameters for defining where the project files are located.

## Most significant Attributes

The “ccfirst” project that was created within the tutorial has the following content:

```

<project
  projectdirectory    ="c:\projects\ccfirst"
  webcontentdirectory=" ${project}/webcontent"
  javasourcedirectory=" ${project}/src"
  javasourcewebinfdirectory=" ${project}/src_webinf"

  javaclassdirectory=" ${project}/webcontentbuild/ec\l\hotdeploy/classes"
  javaclasswebinfdirectory=" ${project}/webcontentbuild/WEB-INF/classes"

  webappaddondirectory=" ${project}/webcontentcc"

  webcontentdeploydirectory="C:/temp/cc20130909/server/tomcat/webapps/ccfirst"
  webcontentroot          ="ccfirst"
  webhostport              ="localhost:50000"

  ...
  ...
>

<deploycopyinfo fromdir=" ${project}/webcontentbuild"
  todir=" ${projectdeploy}"/>
<deploycopyinfo fromdir=" ${project}/webcontentcc"
  todir=" ${projectdeploy}"/>

</project>

```

You already see the most significant attributes:

- “projectdirectory” - This is the main directory of your development project. (If using the option “Store project configuration in project directory” when creating the project, then the projectdirectory is not set - it automatically is the directory in which the project file “.ccproject” is stored).
- “webcontentdirectory” - This is the directory in which you typically keep the “Web Content” of your development (typically a sub directory of “projectdirectory”). It is the directory in which the layout editor stores its layouts.
- “javasourcedirectory” - This is the directory in which the sources of the UI related code are kept. It is referenced by the code generation tool, that is part of the Enterprise Client toolset.
- “jasourcewebinfdirectory” - This is the directory where non-UI related sources are kept. It is only used when using hot deployment.
- “javaclassdirectory” - This is the directory in which your UI-related code is compiled
- “javaclasswebinfdirectory” - This is the directory in which your non-UI-related code is compiled
- “webappaddondirectory” - This is the directory in which all that, what CaptainCasa adds to a web application, is kept. In the example this directory is a sub-directory of the project. If using “compact” projects this is some directory within your CaptainCasa installation (outside your project).

- “webcontentdeploydirectory” - This is the directory that is used by the Enterprise Client toolset to deploy to. By default this is a sub-directory of “<installdir>/server/tomcat/webapps”.
- “webcontentdeployfromdirectory” - This is an optional parameter which allows to support scenarios, in which the project's webcontent directory (parameter “webcontentdirectory”) is the one to store .jsp files, but is NOT the one to use as directory to be used as copy-source for deployment.
- “webcontextroot” and “webhostport” - These are the URL details that are required to access the deployed application within your runtime environment.

## Other Attributes

There are many other attributes that you may use within the project file. A documentation of these attributes is automatically copied into the comment section of the XML file that is created for your project.

## Deployment Configuration

When using the deployment of the Enterprise Client toolset (“Reload server” or “Hot Deploy”) then all the relevant files are transferred from the project directory structure into the deployment directory:

- The “webcontentdirectory” is always copied.
- Additional directories are copied following the “deploycopyinfo” definitions. In the example you see that the compiled classes (“/webcontentbuild”) and the CaptainCasa addons (“/webcontentcc”) are copied as well.

After the file transfer has finished, the toolset triggers the servlet engine (default: Tomcat) to load the new information. In case of a “reload” this means that the web-application is re-started, in case of “hot deploy” this means that parts of the classes are reloaded without re-starting the whole web application.

## Referencing special Values

Within the project configuration you may reference “special values”:

- “\${project}” is the value of attribute “projectdirectory”
- “\${projectdeploy}” is the value of attribute “webcontentdeploydirectory”
- “\${env.xxx}” is the value of environment variable “xxx” (internally resolved by calling Java API System.getenv())
- “\${sys.xxx}” is the value of system property “xxx” (internally resolved by calling System.getProperty())

---

## Integrating an existing Project into the CaptainCasa Toolset

When using the option “Store project configuration in project directory” the project file is part of the project folder - and is not kept within the directory structure of the CaptainCasa toolset.

To integrate such project into the project list of CaptainCasa you need to define a small XML file with the name...

```
<installation>/tools/embeddedserver/webapps/editor/config/projectcs/<projectName>.
```

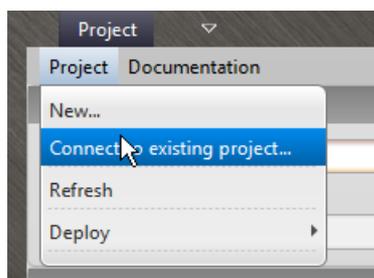
```
xml
```

...with the following content:

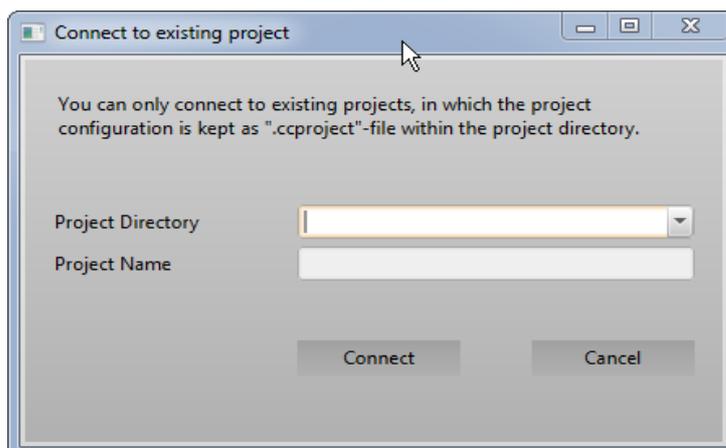
```
<project projectdirectory="<nameOfProjectDirectory">  
  projectfilestoredinprojectdirectory="true">  
</project>
```

The file serves as a pointer to the project directory and its contained “.ccproject” configuration file.

You may create the file by using a certain function of the CaptainCasa toolset:



Define project directory and name in the dialog that shows up:



After pressing “Connect” the corresponding XML file will be generated.

## Template Management

When creating a new layout you pick a certain template as base for the .jsp file that is opened within the Layout Editor. By default there are a couple of templates predefined by CaptainCasa:



You can extend the list of templates by adding own ones through the project definition:

```
<project ... >  
  ...  
  <template resource="/abc/def.jsp" image="/ghi/jkl.jpg"/>  
  ...
```

```
<template ... />  
<template ... />  
</project>
```

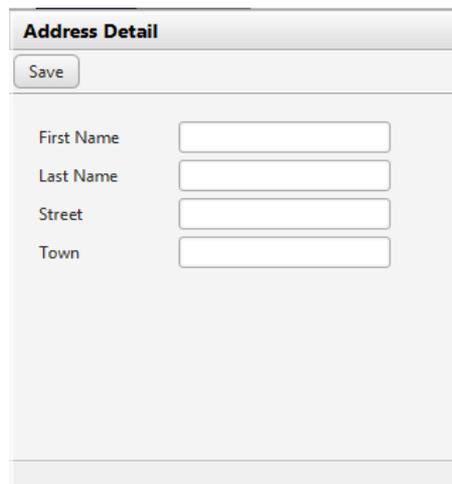
Each template must provide a JSP file which is copied into the newly created page. And it must provide an image which is displayed within the template selection. Both files need to be located within your web application, so that they are accessible as resource.

# Principles of Development

In the tutorial “First Development Steps” you already got to know the basic aspects and artifacts when developing with Enterprise Client:

- You define a page definition which assembles graphical components to form some nice layout. The page definition is kept as .jsp file (Java Server Page).
- Within the page definition, you define the attributes of components. An attribute value may be defined “hard wired” or may be defined “dynamically”. The dynamic definition is pointing to a bean using expressions.
- A bean is an object that resides on server side and that is the data and method counterpart of the page that is running inside the client. The bean is sometimes referred to as “managed bean”.
- The managed bean provides properties and methods which are referenced from the page components.

Let's take a closer look into the internal structures. As example we continue to use the address-page that is created within the tutorial:



---

## The Layout

The layout of the page above is:

```
<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>

<%@taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@taglib prefix="h" uri="http://java.sun.com/jsf/html"%>

<%@taglib prefix="t" uri="/WEB-INF/ectnt"%>

<!-- ===== CONTENT BEGIN ===== -->
<f:view>
  <h:form>
    <f:subview id="addressdetailg_sv">
      <t:rowtitlebar id="g_1" text="Address Detail" />
      <t:rowheader id="g_2">
        <t:button id="g_3"
          actionListener="#{d.AddressDetailUI.onSaveAction}" text="Save" />
      </t:rowheader>
      <t:rowbodypane id="g_4" rowdistance="5">
```

```

        <t:row id="g_5">
            <t:label id="g_6" text="First Name" width="100" />
            <t:field id="g_7" text="#{d.AddressDetailUI.firstName}"
width="150" />
        </t:row>
        <t:row id="g_8">
            <t:label id="g_9" text="Last Name" width="100" />
            <t:field id="g_10" text="#{d.AddressDetailUI.lastName}"
width="150" />
        </t:row>
        <t:row id="g_11">
            <t:label id="g_12" text="Street" width="100" />
            <t:field id="g_13" text="#{d.AddressDetailUI.street}"
width="150" />
        </t:row>
        <t:row id="g_14">
            <t:label id="g_15" text="Town" width="100" />
            <t:field id="g_16" text="#{d.AddressDetailUI.town}" width="150" />
        </t:row>
    </t:rowbodypane>
    <t:rowstatusbar id="g_17" />
    <t:pageaddons id="g_pa" />
</f:subview>
</h:form>
</f:view>
<!-- ===== CONTENT END ===== -->

```

Let's pick some elements to demonstrate what's going on:

```
<t:field id="g_10" text="#{d.AddressDetailUI.lastName}" width="150" />
```

This is the first field definition. The width of the field is “hard-wired”. The text is referencing to a managed bean, meaning: the value is taken from the bean and it is written back into the bean when the user makes changes.

```
<t:button id="g_3"
    actionListener="#{d.AddressDetailUI.onSaveAction}" text="Save" />
```

This is the button calling an “onSaveAction” method within the bean.

---

## The Managed Bean

The managed bean is a normal bean definition that is the logical counterpart of the page. In this example the code is:

```

package managedbeans;

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

import javax.faces.event.ActionEvent;

@CCGenClass (expressionBase="#{d.AddressDetailUI}")
public class AddressDetailUI
    extends PageBean
    implements Serializable
{
    // -----
    // members
    // -----

    String m_town;
    String m_street;
    String m_lastName;
    String m_firstName;

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

```

```

// -----
public AddressDetailUI()
{
}

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

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

public String getTown() { return m_town; }
public void setTown(String value) { this.m_town = value; }

public String getStreet() { return m_street; }
public void setStreet(String value) { this.m_street = value; }

public String getLastName() { return m_lastName; }
public void setLastName(String value) { this.m_lastName = value; }

public String getFirstName() { return m_firstName; }
public void setFirstName(String value) { this.m_firstName = value; }

public void onSaveAction(ActionEvent event)
{
    if (m_firstName == null || m_lastName == null)
    {
        Statusbar.outputError("Please define all name fields.");
        return;
    }
    m_town = m_firstName + "/" + m_lastName;
}
}

```

You see:

- The class provides the properties that are referenced.
- The class provides the method that is referenced. The method needs to provide one parameter to pass on - the “ActionEvent”. Pay close attention during development: it's the “faces-ActionEvent” that you need to use as parameter, NOT the “AWT-Event”. Do not use code-complete functions too fast...!
- The class is derived from the base class “PageBean”. This is not obligatory! You also may directly define a bean class for your page processing. But: using the PageBean-framework significantly simplifies the modularization of your user interface: page beans are designed to be re-usable - for example as sub-part of another page, or as dialog that you want to pop up. Please read further information within the chapter “Page Bean Modularization”.

By default there is one instance of the bean available at runtime - per session! So you do not have to pay attention to the fact that several users are accessing your application simultaneously. (Of course you may want to check if there is some conflict if users work in parallel on the same data - but this is some different issue, where you have to think optimistic or pessimistic locking etc.)

You will later on see, that if the page is re-used, there might be several instances according to your coding: e.g. you might render within one screen a “supplier address” on the left and a “vendor address” on the right. both using the “AddressDetailUI”. In this case you have two separate instances - which are fully under your control.

---

## How Objects are created on Server Side

### Resolving of Expressions

Let's get into a bit more detail about how an instance of “AddressDetailUI” is created on server side.

When the page is requested by the client then a request is sent to the server, the XML of the page (.jsp file) is read and interpreted. During interpretation the expressions within the page are resolved in order to pick the dynamic data which is bound to certain attributes.

An expression contains several fragments - separated by a dot “.”. During resolution the fragments are processed from the left to the right.

### The “#{d.” Fragment

Let's take as example the expression “#{d.AddressDetailUI.firstName}”.

The first part of the expression is the “d”. And this part is resolved by the just normal JSF mechanism: JSF takes a look into the faces-config.xml file and finds out what class representation is behind the “d”. JSF creates and registers a new instance of this class - typically using a constructor without any parameters.

The faces-config.xml of your project was created during the project creation and contains the following content:

```
<managed-bean>
  <managed-bean-name>d</managed-bean-name>
  <managed-bean-class>managedbeans.Dispatcher</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

JSF sees that the scope of the bean is a “session”-scope. This means that the bean will be kept in the session - and will not be removed once the current request is processed.

The Dispatcher class that is referenced also was added to your project when creating the project. Of course you can update the faces-config.xml to your needs, e.g. by moving the Dispatcher-class into a different package than the one proposed!

### The “.AddressDetailUI.” Fragment

The Dispatcher is a simple object factory - providing the interface “java.util.Map”.

Whereas the normal resolution of a “.” within an expression follows the corresponding “set/get”-methods of a bean, the resolution of a “.” with a map is calling the “Map.get(.”) and “Map.put(.”) methods.

The dispatcher as consequence is called with “get(“AddressDetailUI”)”. It first checks if an object for this name is already available in its map. If not it creates an instance of AddressDetailUI and add the instance to its map.

The following default assumptions are made:

- The dispatcher assumes the “AddressDetailUI” class to reside in the same package as itself.
- The dispatcher assumes the “AddressDetailUI” class to have one of the following constructors:

```
either:   public AddressDetail() { ... }
or       :   public AddressDetailUI(IDispatcher dispatcher) { ... }
```

## The “.firstName}” Fragment

Well, now it's easy: the instance of the class AddressDetailUI is checked for the “getFirstName()” method - and finally the value returned by this object is the one that is passed into the corresponding attribute.

## Resolution is done with every Request!

What was described before now is executed every time the client sends a request to the server.

- Within the request processing changes of the data on client side (due to user input) are transferred into corresponding expressions (thus: object-properties) on server side.
- Then actions are invoked.
- Then the collection of data is started as described and data changes are sent back to the client side.

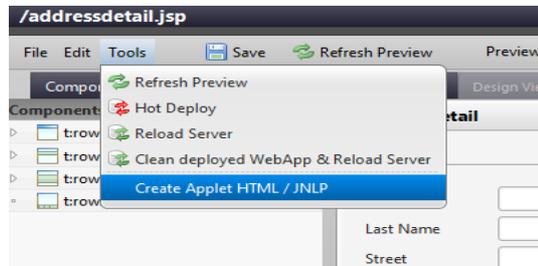
Pay attention not to add any intensive operations into the get-methods that are referenced during expression resolution as consequence. The expression resolution must be something very fast because it is repeated with every request!

---

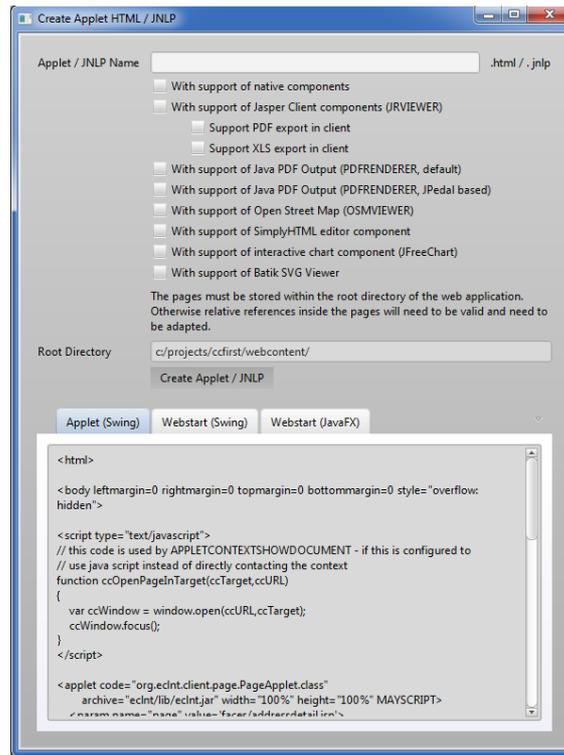
## Viewing the Page outside the Toolset

In order to see the client outside the WYSIWYG environment of the layout editor, you need to either define a so called “.jnlp” file for web-start access, or you need to define an HTML page containing an applet definition - or you can directly start the client as Java application.

You can directly create the JNLP and HTML files from the CaptainCasa toolset:



Call the menu item “Create Applet HTML/JNLP” and the following dialog will appear:



Specify the name of the files to be created. For the first examples you do not have to pay attention to the check boxes that fill up the screen. The files that are created are directly placed within the “/webcontent” directory of your project.

When do you have to pay attention to the check boxes? - Later on, you will get to know certain components that do require special libraries to be available on client side. E.g. there is a component PDFRENDERER which directly renders PDF using a Java library that is provided by JPedal (see copyright notices at the beginning of this documentation). This library must be registered within the HTML/JNLP file, so that the client side runtime environment loads the corresponding library.

## Java Application Access

You can of course start the client of the CaptainCasa Enterprise Client framework as an application as well.

The following batch file describes how to start a page using the Swing client:

```
set TCP=..\resources\webappaddons\ecInt\lib\ecInt.jar
start jre\bin\javaw.exe -cp %TCP% org.ecInt.client.page.PageBrowser
"http://localhost:50000" "/ccfirst/addressdetail.jsp"
```

The following batch file describes how to start a page using the FX client:

```
set TCP=%TCP%;..\resources\clientfx\ecIntfx.jar
start jre\bin\javaw.exe -cp %TCP% org.ecInt.fxclient.elements.PageBrowserStarter
"http://localhost:50000" "/demos/faces/workplace/workplaceFX.jsp"
```

## Dispatcher Concept

You already got to know the “#{d.}”-dispatcher in the text before. The dispatcher serves as factory to find beans, typically using their class name. By default the dispatcher tries to find the bean class in the same package that the dispatcher itself is positioned.

When creating a project then a default dispatcher is automatically generated into the project, being located in package “managedbeans”. Of course you can change this any time - please do not forget to update the faces-config.xml accordingly.

If an expression “#{d.AddressDetailUI. ...}” is to be resolved then the default class arrangement is:

```
managedbeans          <== Package
  Dispatcher
  AddressDetailUI
```

Of course there are situations in which your project grows and you do not want to arrange all UI classes within one package. In this case you may use a “dispatcherinfo.xml” file and tell the dispatcher where to search for classes. The dispatcherinfo.xml must be positioned in the dispatcher’s package.

```
managedbeans          <== Package
  Dispatcher
  AddressDetailUI
  dispatcherinfo.xml
```

The content of the file is some XML configuration:

```
<dispatcherinfo>
  ...
  <managedpackage name="other.package1"/>
  <managedpackage name="other.package2"/>
  ...
  <managedbean name="ArticleMaster" class="com.appl.log.ArticleMasterBean"/>
  <managedbean name="CustomerMaster"
class="com.appl.sales.CustomerMasterBean"/>
  ...
</dispatcherinfo>
```

You can configure...

- “managedpackages” definitions: these are packages in which the dispatcher looks for class names to fit to the class name it currently has to resolve
- “managedbean” definitions: these are single naming hints for explicitly telling the dispatcher how to resolve a certain name

The sequence in which the dispatcher operates when resolving a name is:

1. The dispatcher looks for “managedbean” definitions.
2. The dispatcher looks into its own package
3. The dispatcher processes the “managedpackage” packages

In some cases, having one central dispatcherinfo.xml file is not adequate. Example: if you have multiple project aspects being resided within one web application then you may want that each project is managing its managed beans on its own.

The dispatcherinfo.xml file allows to point to other dispatcherinfo.xml files that are residing in a different package. In this case the content of dispatcherinfo.xml may look as follows:

```
<dispatcherinfo>
  <!-- normal content -->
  ...
  <!-- pointer to other dispatcherinfo.xml -->
  <dispatcherinfoextension resource="com/appl/finance/dispatcherinfo.xml"/>
</dispatcherinfo>
```

---

## Some Details about the default Tomcat Configuration

This chapter is only relevant for readers who are experienced with Tomcat configuration issues.

The installation of Enterprise Client comes with a Tomcat server on its own. The purpose is:

- Having a Tomcat platform for running the tools (Layout Editor)
- Having a Tomcat platform for “ad hoc” starting with development projects

In order to “nicely” support the continuous development as described in the previous chapter (“refresh” button in Layout Editor reloads web application) the following configuration items automatically come with the Tomcat installation:

- The “manager” web application (tomcat/webapps/manager) is configured to run without security - corresponding information is commented out in its web.xml file. - Background: when refreshing your project by using the “refresh” button then the toolset will, after having finished copying, send a message to the manager application, which tells the Tomcat manager to reload the project's web application. Normally this message requires an administrator logon to the manager application - by having switched off the security this is not required anymore.
- The tomcat/conf/context.xml file was updated so that Tomcat will NOT reload a web application when the web.xml file is changed. Background: the reloading of the web application is explicitly done through the “refresh” button of the toolkit. There is no need to observe further files.

```
<!-- Default set of monitored resources -->
<!--
<watchedResource>WEBINF/web.xml</watchedResource>
-->
```

The Tomcat that is part of the Enterprise Client installation is not usable for productive usage - it is optimized for pure development usage!

# Working with Components

Components are the essential graphical elements that you use in order to define layouts. A whole page definition consists of an assembly of components, starting with some basic ones (e.g. “t:rowbodypane”) and ending with some very specific ones (“t:field”).

---

## Types of Components

### Container - Row - Column

Within a layout, components are assembled using a container-row-column layout management.

- Container components are areas in which you can place row components.
- A row component can hold column components.
- All typical “end controls” like field, check box, button, etc. are column components. You put these components inside a row. Each column component has attributes like width and height, and as a result occupies a certain amount of space in the row.
- There are column components as well that themselves are container components. This means: inside a row you put a container as control, and inside the container you start again putting rows.

You see: containers contain rows, rows contain columns, columns may be containers as well.

All this, together with the possibility to define widths and heights either in a “pixel mode” or in a “percentage mode”, allows you to build very flexible layouts, that correspond to the size of the screen they are called in.

### Conventions

For better knowing the role of a component there are some naming conventions:

- Row components have the prefix “row”. Example: “t:row”.
- Container components have the suffix “pane”. Example: “t:pane”.
- Column components do not have a specific prefix or suffix. Example: “t:button”

Let’s look at the name “t:rowbodypane”: it is a row component, i.e. you may put it into an container. On the other side it is a container component as well, this means it itself can contain row components.

It may sometimes help to compare container components with HTML tables (“table”), row components with HTML rows (“tr”) and column components with HTML columns (“td”). The same way you can nest HTML tables into HTML tables you can nest containers into containers with Enterprise Client. ...but, be careful: this is a comparison only!

### Non Graphical Components

There are some components which do not have any meaning from a layout point of view, but which are used in the layout to specify some server processing aspect.

Example: there is a component which allows to set a managed bean property.

---

## Rules which apply to all Components

### Component Sizing Aspects

The sizing of a component is determined by its width and its height.

There are certain sizing modes which can be used simultaneously:

- No sizing at all: if you do not pass a specific size, the component will try to size itself. E.g. a button will occupy the width it requires to place the text inside.
- Pixel sizing: you may define width/height as “pixel values”. Well, pixel is not completely true because there is a multiplication factor in the client, by which pixel values are multiplied. This factor is “1” by default, but can be increased in order to change this sizing, e.g. in order to make everything a bit bigger. This is a useful function in the area of accessibility support.
- Percentage sizing: you may define sizes as percentage value. You need to be aware of the paradigm that the percentage sizing is based on the space, that a component receives from its components above. This is a kind of outside in sizing approach - the top layer tells what space it grants, and the inside layers tell (by percentage definition) what they take from the granted space.

### Action Listener

Most components provide an “actionListener” attribute. This refers to a method implementation within a managed bean. All the events that are associated with one component are sent to the server through this one action listener method.

Each event for a component is associated with a certain command, which is a string value. The command is the name of the specific event, sometimes the command also comes with parameters. The format of the type string is comparable with a method call, examples are: “flush()”, “drop(value)”.

On server side the action listener is represented by a method, providing an “ActionEvent” as parameter. All events that are triggered by Enterprise Client passing a sub-class of ActionEvent - with the name “BaseActionEvent”. This class has a “getCommand()” method that allows you to get the command string value, and it has a “getParams()” method which allows you to get the parameters, if any passed.

All default commands that are used by Enterprise Client are collected in the interface “IBaseActionEvent”:

Example: a button has an event “invoke()” when the user presses the button and has an event “drop(value)” when the user drags and drops information onto the button. A proper implementation of the server-side action listener looks like this:

```
public class XYZ
    implements IBaseActionEvent
{
    public void onButtonEvent(ActionEvent ae)
    {
        BaseActionEvent bae = (BaseActionEvent)ae;
        if (bae.getCommand().
            equals(EVTYPE_INVOKE))
        {
            ...
            ...
        }
        else if (bae.getCommand().
            equals(EVTYPE_DROP))
        {
            ...
            ...
        }
    }
}
```

```
}  
  }  
}
```

Please note: the command and parameter values are string values. You need to compare using “.equals”, you may not use “==”!

## Flush Management (Component Level)

All input components (e.g. field, check box, radio button, etc.) manipulate a piece of data which is typically referenced to a managed bean property.

The default behavior of the client processing is that data changes are kept in the client and wait for the next significant event to be transferred to the server. A significant event may for example be a button event.

All input components provide for the property “flush”. If this is set to “true” then the data change is treated as a significant event and causes a roundtrip in which all data changed is transferred to the server. In addition, each component may provide an action listener which receives a “flush()”-event.

By using the flush mechanism you can build layouts in which parts of the layout directly respond to the user changing data.

With the FIELD component there is an attribute FLUSHTIMER in addition. By default, when setting the FLUSH attribute of a field to “true”, data changes are flushed to the server when the user leaves the field (“focusLost” event). By setting the FLUSHTIMER you can define that after a certain duration on inactivity changes are flushed to the server automatically.

In general: pay attention to correctly setting the FLUSH attribute! Be aware of cost of round trips in your and your customers' environment.

## Flush Management (Container/Row Level)

In addition to the FLUSH definition on component level there is a flush management on container level. On any container/row level you can define the attribute “FLUSHAREA” to true or false. If setting the attribute to “true” then a roundtrip is triggered to the server side if ...

- ... data was changed within this area by the user
- ... **and** the user leaves this area (e.g. by putting the focus out of the area)

## Background Painting (BACKGROUND and BGPAIN)

All of the container components and some of the column components provide the possibility to define the background coloring of the component in a quite sophisticated way.

The most plain way is to use the BACKGROUND color attribute. You can specify the background color in one of two ways:

- #RRGGBB - this is the direct way, defining the color with red, green, blue values
- #RRGGBBTT - with “TT” you assign a transparency value. This means you can define the transparency factor, with “FF” being the highest value, and “00” being the lowest value. Example: if you want the background of a component to be a “light shading on top of the existing background” then use the color “#00000020” - a black with little transparency.

Then there is the BGPAIN attribute: via BGPAIN you can execute a series of paint commands, each one being a command like “rectangle(0,0,100,100,#FF0000)”, concatenated by semicolons.

The commands available area:

- rectangle(x,y,width,top,color) ==> plain rectangle
- rectangle(x,y,width,top,color1,color2,direction) ==> a rectangle in which the color starts with “color1” and ends with “color2”, dependent on the direction, “vertical” or “horizontal”.
- roundedrectangle(x,y,width,height,radient1,radient2,color) ==> rounded rectangle, the rounding defined by the radient values
- roundedrectangle(x,y,width,height,radient1,radient2,color1,color2,direction) ==> rounded rectangle with color effect, same as with rectangle coloring
- image(x,y,address,position) ==> output of an image. The address is a URL, relative to the page (“images/xyz”) or absolute within the web application (“/dirxyz/images/...”). The position defines how the image is positioned relative to the x,y point you define. The possible values are “lefttop”, “leftmiddle”, “leftbottom”, “centertop”, “centermiddle”, “centerbottom”, “righttop”, “rightmiddle”, “rightbottom”. Example: if an image should be “on the very left top” then the definition is “image(0,0,xyz.jpg,lefttop)”. If the image should be on the right bottom then use “image(100%,100%,xyz.jpg,rightbottom)”. If it should be centered then use “image(50%,50%,xyz.jpg,centermiddle)”.
- image(x,y,width,height,address,position) ==> the same as the direct image command, but now with the definition of the width and height.

The x,y,width and height values can be defined in severals ways:

- Pixels, “100”
- Percentages, “50%”
- Combination “Percentage-Pixels”, e.g. “100%-5”. Only “-” is allowed as operand, the first value must be a percentage value, the second one an absolute value. Example: you may want to draw some “padded rectangle” by defining “rectangle(5,5,100%-5,100%-5,#FF0000)”.

Color values in the BGPAIN command can be either defined using #RRGGBB or #RRGGBBTT, as described above.

When using a sequence of commands with BGPAIN then the paint commands are executed in exactly the sequence that you define.

## Focus Management - Setting Focus to dedicated Component

CaptainCasa Enterprise Client provides an explicit control over setting the keyboard focus to a dedicated component.

All input elements provide the attribute REQUESTFOCUS. You need to set this attribute for those components that you want to explicitly assign the keyboard focus to.

The most simple way of usage is to define a component's REQUESTFOCUS attribute to be the fix value “creation”:

```
<field ... requestfocs="creation" ...>
```

In this case the keyboard focus of a page that is rendered within the client is automatically moved to this component. After being rendered the attribute will not have

any further consequences - from now on the focus is following the user's navigation.

You also can control the focus from server side within a "live page". For this reason you need to bind the REQUESTFOCUS attribute to a managed bean's property. The property must return a long-value. Every time you want a component to gain the focus, you need to update the value. For updating you use a utility Class "RequestFocusManager".

Have a look onto the following scenario:

First Name	<input type="text" value="AAA"/>
Last Name	<input type="text" value="BBB"/>
Street	<input type="text" value="CCC"/>
ZipCode	<input type="text" value="DDD"/>
City	<input type="text" value="EEE"/>

By using the combo box, a field is selected that then gains the focus.

In the layout definition each FIELD component is bound to a different property for controlling the requesting of the focus:

```
<t:rowdemobodypane id="g_3" objectbinding="demoRequestfocus" rowdistance="2" >
  <t:row id="g_4" >
    <t:label id="g_5" text="First Name" width="120" />
    <t:field id="g_6" requestfocus="#{d.demoRequestfocus.firstNameRF}"
text="#{d.demoRequestfocus.firstName}" width="200" />
  </t:row>
  <t:row id="g_7" >
    <t:label id="g_8" text="Last Name" width="120" />
    <t:field id="g_9" requestfocus="#{d.demoRequestfocus.lastNameRF}"
text="#{d.demoRequestfocus.lastName}" width="200" />
  </t:row>
  <t:row id="g_10" >
    <t:label id="g_11" text="Street" width="120" />
    <t:field id="g_12" requestfocus="#{d.demoRequestfocus.streetRF}"
text="#{d.demoRequestfocus.street}" width="200" />
  </t:row>
  <t:row id="g_13" >
    <t:label id="g_14" text="ZipCode" width="120" />
    <t:field id="g_15" requestfocus="#{d.demoRequestfocus.zipCodeRF}"
text="#{d.demoRequestfocus.zipCode}" width="200" />
  </t:row>
  <t:row id="g_16" >
    <t:label id="g_17" text="City" width="120" />
    <t:field id="g_18" requestfocus="#{d.demoRequestfocus.cityRF}"
text="#{d.demoRequestfocus.city}" width="200" />
  </t:row>
  <t:rowdistance id="g_19" height="20" />
  <t:row id="g_20" >
    <t:coldistance id="g_21" width="120" />
    <t:combobox id="g_22"
actionListener="#{d.demoRequestfocus.onFocussedFieldAction}" flush="true"
value="#{d.demoRequestfocus.focussedField}" width="120" >
      <t:comboboxitem id="g_23" text="firstName" value="firstName" />
      <t:comboboxitem id="g_24" text="lastName" value="lastName" />
      <t:comboboxitem id="g_25" text="street" value="street" />
      <t:comboboxitem id="g_26" text="zipCode" value="zipCode" />
      <t:comboboxitem id="g_27" text="city" value="city" />
    </t:combobox>
  </t:row>
</t:rowdemobodypane>
```

When the user selects the combo box then a corresponding method is invoked on server side:

```
package workplace;

import org.eclnt.jsfserver.session.RequestFocusManager;

public class DemoRequestfocus extends DemoBase
```

```

{
    protected long m_firstNameRF;
    protected long m_lastNameRF =
RequestFocusManager.getCreationRequestFocusCounter();
    protected long m_streetRF;
    protected long m_cityRF;
    protected long m_zipCodeRF;
    public long getFirstNameRF() { return m_firstNameRF; }
    public long getLastNameRF() { return m_lastNameRF; }
    public long getStreetRF() { return m_streetRF; }
    public long getCityRF() { return m_cityRF; }
    public long getZipCodeRF() { return m_zipCodeRF; }

    protected String m_firstName;
    protected String m_lastName;
    protected String m_street;
    protected String m_zipCode;
    protected String m_city;
    public void setFirstName(String value) { m_firstName = value; }
    public String getFirstName() { return m_firstName; }
    public void setLastName(String value) { m_lastName = value; }
    public String getLastName() { return m_lastName; }
    public String getStreet() { return m_street; }
    public void setStreet(String value) { m_street = value; }
    public String getZipCode() { return m_zipCode; }
    public void setZipCode(String value) { m_zipCode = value; }
    public String getCity() { return m_city; }
    public void setCity(String value) { m_city = value; }

    protected String m_focussedField;
    public String getFocussedField() { return m_focussedField; }
    public void setFocussedField(String value) { m_focussedField = value; }

    public void onFocussedFieldAction(ActionEvent event)
    {
        if (m_focussedField == null)
            return;
        else if (m_focussedField.equals("firstName"))
            m_firstNameRF = RequestFocusManager.getNewRequestFocusCounter();
        else if (m_focussedField.equals("lastName"))
            m_lastNameRF = RequestFocusManager.getNewRequestFocusCounter();
        else if (m_focussedField.equals("street"))
            m_streetRF = RequestFocusManager.getNewRequestFocusCounter();
        else if (m_focussedField.equals("zipCode"))
            m_zipCodeRF = RequestFocusManager.getNewRequestFocusCounter();
        else if (m_focussedField.equals("city"))
            m_cityRF = RequestFocusManager.getNewRequestFocusCounter();
    }
}
}

```

The value for the properties that control the focus management is taken from the class RequestFocusManager. This class provides two methods:

- getNewRequestFocusCounter() - you receive a counter that indicates that the corresponding component receives the focus one time - when the response is processed on client side
- getCreationRequestFocusCounter() - this is the same as setting the attribute's value to "creation"

Please note: when working with grids there is an additional function. You may set the focus to a certain grid row, by using the function FIXGRIDBinding.selectAndFocus(item).

## Focus Management - Defining the Focus-/Tab-Sequence

While the REQUESTFOCUS attribute is responsible for setting the focus to a dedicated component from server side, there is a parallel mechanism that allows you to explicitly define the tab-sequence within your page.

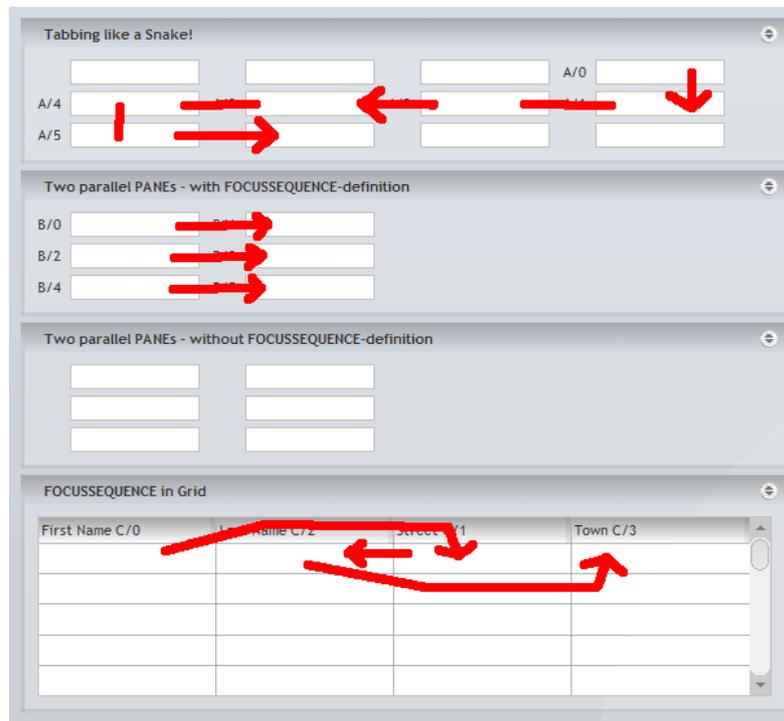
By default the tab-sequence is defined by the layout definition: the sequence of the components within the layout definition is exactly the sequence which is used by the

focus management. This default results are adequate for most scenarios, but you sometimes may want to override this default sequence:

By using the attribute FOCUSSEQUENCE you can define an own tab-sequence within your page. The definition is as follows:

- The value that you define for the attribute FOCUSSEQUENCE consists out of an area and an index definition. Both are concatenated, with a “/”-slash in between. Example. the area has the name “A”, the index is “1”, so the full value is “A/1”. (Please. as usual: pay attention to no add blank spaces...!)
- If a component's FOCUSSEQUENCE attribute is defined, e.g. the value being “A/1” then the focus management tries to find the next component to tab to in the following way:
  - If checks if there is a component defined, that is defined within the same area “A”, and which holds the next highest index-value. If there is such a component, then this is the one to be focused next.
  - If there is no component left within the area “A” then just the normal, next component is chosen, i.e. then the default sequence is used.

Example, taken from demo workplace:



The labels in front of the FIELDS represent the FIELDS' definitions for the attribute FOCUSSEQUENCE.

Why is there an explicit area-definition, and not just a focus index? - The answer is: now you can define areas in the page with a certain tab-sequence definition. When restructuring the page, e.g. by moving components from the top to the bottom, then the focus definitions in many cases are still valid.

If including pages (using ROWINCLUDE or ROWPAGEBEANINCLUDE), then each included page is kept independent from the other pages. I.e. you may use the area “A” on two different pages - both will not know from one another at runtime.

## Attribute TRIGGER

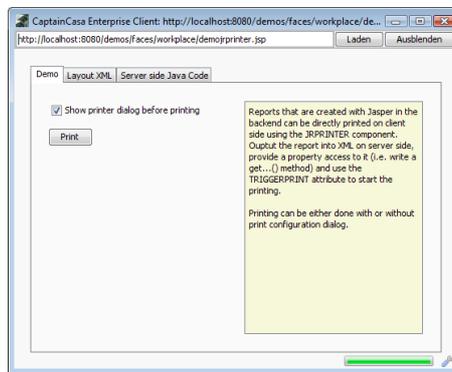
Some components provide a property containing the name "trigger". These components provide some client side function that you need to invoke from server side.

Example: the PAINTAREA component provides an attribute TRIGGER: if the trigger is invoked then a certain animation is done on client side. Or: the JRPRINTER (Jasper Reports printer) component can be invoked using its attribute TRIGGERPRINT in order to print out some reports on client side.

Triggers in general are treated in the following way:

- The client side functions is changed every time the trigger is updated, i.e. when the trigger changes its value.
- If the trigger's value is null, then no client side function is executed.

There is a class "org.eclnt.jsfserver.elements.util.Trigger" which you should directly use as property value, that you bind via expression. Have a look onto the following example, printing out a Jasper report on request:



Layout:

```
<f:view>
<h:form>
<f:subview id="workplace_demojrprinter_21">
<t:beanprocessing id="g_1" >
<t:jrprinter id="g_2" jasperxml="#{d.DemoJrprinter.jasperXml}"
triggerprint="#{d.DemoJrprinter.triggerPrint}"
withprintdialog="#{d.DemoJrprinter.withDialog}" />
</t:beanprocessing>
<t:rowdemobodypane id="g_3" objectbinding="#{d.DemoJrprinter}">
<t:row id="g_4" >
<t:checkbox id="g_5" selected="#{d.DemoJrprinter.withDialog}" text="Show
printer dialog before printing" />
</t:row>
<t:rowdistance id="g_6" height="10" />
<t:row id="g_7" >
<t:button id="g_8" actionListener="#{d.DemoJrprinter.onPrint}"
text="Print" />
</t:row>
</t:rowdemobodypane>
<t:pageaddons id="g_pa"/>
</f:subview>
</h:form>
</f:view>
```

Code:

```
package workplace;

import java.io.Serializable;

import javax.faces.event.ActionEvent;

import org.eclnt.editor.annotations.CCGenClass;
import org.eclnt.jsfserver.elements.util.Trigger;
```

```

import org.eclnt.jsfserver.managedbean.IDispatcher;
import org.eclnt.util.file.ClassloaderReader;
import org.eclnt.util.file.FileManager;

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

public class DemoJrprinter
    extends DemoBase
    implements Serializable
{
    public DemoJrprinter(IDispatcher dispatcher)
    {
        super(dispatcher);
    }

    protected boolean m_withDialog = true;
    public boolean getWithDialog() { return m_withDialog; }
    public void setWithDialog(boolean value) { m_withDialog = value; }

    protected Trigger m_triggerPrint = new Trigger();
    public Trigger getTriggerPrint() { return m_triggerPrint; }
    public void setTriggerPrint(Trigger value) { m_triggerPrint = value; }

    protected String m_jasperXml;
    public String getJasperXml() { return m_jasperXml; }
    public void setJasperXml(String value) { m_jasperXml = value; }

    public void onPrint(ActionEvent event)
    {
        String s = (new
ClassLoaderReader()).readUTF8File("workplace/resources/jasperxmlexport.xml", true)
;
        m_jasperXml = s;
        m_triggerPrint.trigger();
    }
}

```

## Special Container / Row Components

The component structure “container - row - component” that was introduced at the beginning of this chapter is the general structure that is used for defining the layout of pages.

In addition there are some special components which are related to this structure but behave a bit differently:

### COLSYNCHEDPANE, COLSYNCHEDPANEROW - Connected Columns

The COLSYNCHEDPANE component is a container, in which its contained rows (of type COLSYNCHEDPANEROW) are not rendered independent from another, but are rendered in a “connected” way.

Take a look onto the following screen shot:

The layout definition is:

```

<t:colsynchedpane id="g_5" background="#00000030"
border="#00000030" coldistance="5" padding="10" rowdistance="5">
  <t:colsynchedrow id="g_6">
    <t:label id="g_7" text="First Name" width="100" />
    <t:field id="g_8" width="100" />
    <t:button id="g_9" text="Clear" />
  </t:colsynchedrow>
  <t:colsynchedrow id="g_10">
    <t:label id="g_11" text="Last Name" width="100" />
    <t:field id="g_12" width="100" />
    <t:button id="g_13" text="Clear" />
  </t:colsynchedrow>
  <t:colsynchedrowdistance id="g_14" height="20" />
  <t:colsynchedrow id="g_15">
    <t:label id="g_16" text="Marital Status" width="100" />
    <t:checkbox id="g_17" text="Married" />
    <t:button id="g_18" text="Clear" />
  </t:colsynchedrow>
</t:colsynchedpane>

```

Each row consists out of three components. You see that the components are arranged in a common column structured: the button of the third row is not directly positioned behind the check box (as it would have happened in a normal ROW component) - but is aligned to the column structure of the columns above.

You may use a COLSPAN attribute that is available in each component to identify that one component overlaps several other components of other rows.

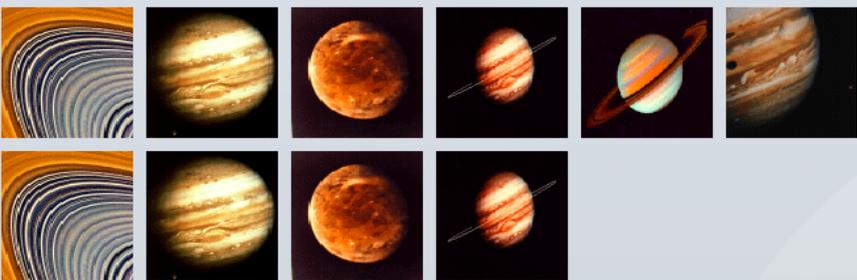
This COLSYNCHEDPANE reminds a bit to the HTML-Table (<table>...</table>) definition - with all its positive and negative consequences: when adding a control into one row of the table then you have to update all other rows and e.g. adjust the COLSPAN definitions.

## ROWFLEXLINECONTAINER - A Row with Line Breaks

The special row component ROWFLEXLINECONTAINER allows to arrange a sequence of contained controls. The controls are horizontally listed - one after the other. If there is no sufficient horizontal space, then a “line break” is done, and the controls are arranged in a second (or third or fourth...) line.

In the following example the images are arranged in two lines...

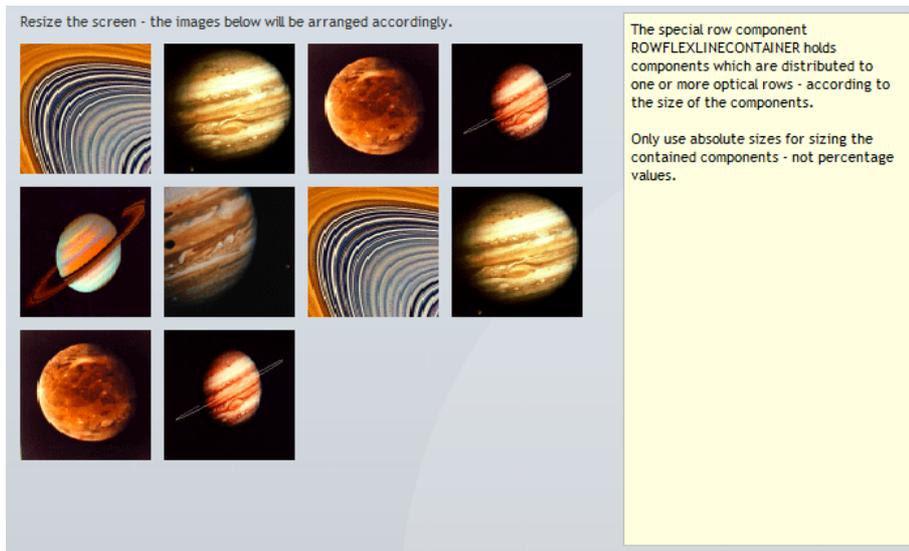
Resize the screen - the images below will be arranged accordingly.



The special row component ROWFLEXLINECONTAINER holds components which are distributed to one or more optical rows - according to the size of the components.

Only use absolute sizes for sizing the contained components - not percentage values.

...or in three lines...



...dependent on the size of the screen.

The corresponding XML is:

```
<t:rowflexlinecontainer id="g_4" coldistance="10" rowdistance="10">0
  <t:image id="g_5" height="100" image="/images/pictures/photo0.gif"
    kepratio="false" width="100" />
  <t:image id="g_6" height="100" image="/images/pictures/photo1.gif"
    kepratio="false" width="100" />
  <t:image id="g_7" height="100" image="/images/pictures/photo2.gif"
    kepratio="false" width="100" />
  <t:image id="g_8" height="100" image="/images/pictures/photo4.gif"
    kepratio="false" width="100" />
  <t:image id="g_9" height="100" image="/images/pictures/photo5.gif"
    kepratio="false" width="100" />
  <t:image id="g_10" height="100" image="/images/pictures/photo6.gif"
    kepratio="false" width="100" />
  <t:image id="g_11" height="100" image="/images/pictures/photo0.gif"
    kepratio="false" width="100" />
  <t:image id="g_12" height="100" image="/images/pictures/photo1.gif"
    kepratio="false" width="100" />
  <t:image id="g_13" height="100" image="/images/pictures/photo2.gif"
    kepratio="false" width="100" />
  <t:image id="g_14" height="100" image="/images/pictures/photo4.gif"
    kepratio="false" width="100" />
</t:rowflexlinecontainer>
```

Please note: all components that are defined in the row should not be defined with a percentage width or height definition!

# Grid Components

The FIXGRID component is the component for building grids. Grids include:

- Lists
- Table with various input cells
- Trees

The FIXGRID arranges any type of component in a grid of fixed columns, that's where the name comes from. The type of components is up to you, i.e. you can arrange labels, fields, combo boxes, buttons, etc.

The FIXGRID can hold any number of rows - on the server side. On client side only those rows are loaded and presented that the user currently sees. When scrolling through the grid the grid automatically updates its items from the server. We call this "server side scrolling".

---

## FIXGRID Basics

### FIXGRID, GRIDCOL, GRIDHEADER, GRIDFOOTER

A FIXGRID component is structured in the following way:

```
FIXGRID
* GRIDCOL
  1 <column component>
* GRIDHEADER
  * <column component>
* GRIDFOOTER
  * <column component>
```

Each grid has a series of columns, per column you define one GRIDCOL component defines the initial width and the text of the column.

Inside the GRIDCOL component you define exactly one content component, i.e. you define if the column's cells should be a FIELD or a LABEL or whatever control. It is possible to define a PANE as content component - and inside the PANE to open up any content. So any content can be placed inside the GRIDCOL components, but there must only exist one component within the GRIDCOL definition itself.

```
Examples:
GRIDCOL          <== valid
  LABEL
GRIDCOL          <== valid
  FIELD
GRIDCOL          <== valid
  PANE
  ROW
  LABEL
  LABEL
GRIDCOL          <== invalid, only ONE content component inside GRIDCOL!
  LABEL
  LABEL
```

The grid may have header rows and footer rows. Per row you define a corresponding GRIDHEADER or GRIDFOOTER component.

Have a look at the example "demominispread.jsp":



(for lists) and FIXGRIDTreeBinding (for trees).

This FIXGRID-object on server side is referenced by the OBJECTBINDING attribute of the FIXGRID.

Let's focus on list processing first, trees will follow later on:

The FIXGRIDListBinding supports a method `getItems()` which passes back a `java.util.List` interface in which you now can add your data rows.

Note: adding data rows on server side does not mean that all the data is transferred to the client. If adding 1000 rows on server side, and only 10 rows are displayed on client side, then as consequence also only 10 rows will be loaded into the client. The FIXGRIDListBinding unburdens you from the task of scrolling through the rows.

Each data object that you now put into a FIXGRIDListBinding collection needs to provide the properties that are referenced by the controls inside the GRIDCOL definitions.

You may already have noted that inside the property definitions of FIELD and FORMATTEDFIELD components that are located below GRIDCOL components, the expression “`{<property>}`” is used. This expression indicates that the property value is taken from the object that represents the row.

Have a look at the implementation which belongs to the “`demominisread.jsp`” demo mentioned above:

```
public class MyRow extends FIXGRIDItem
{
    String m_region;
    double m_revenue;
    double m_cost;
    public void setRegion(String value) { m_region = value; }
    public String getRegion() { return m_region; }
    public double getRevenue() { return m_revenue; }
    public void setRevenue(double value) { m_revenue = value; }
    public double getCost() { return m_cost; }
    public void setCost(double value) { m_cost = value; }
    public double getProfit() { return m_revenue - m_cost; }
    public String getProfitColor() { if (getProfit()<0) return "#FFE0E0"; else
return "#E0FFE0"; }
}

FIXGRIDListBinding<MyRow> m_sheet = new FIXGRIDListBinding<MyRow>();

public DemoMinisread()
{
    MyRow r;
    r = new MyRow(); r.m_region = "North"; m_sheet.getItems().add(r);
    r = new MyRow(); r.m_region = "East"; m_sheet.getItems().add(r);
    r = new MyRow(); r.m_region = "South"; m_sheet.getItems().add(r);
    r = new MyRow(); r.m_region = "West"; m_sheet.getItems().add(r);
}

public FIXGRIDListBinding<MyRow> getSheet() { return m_sheet; }

public void onCreateRow(ActionEvent event)
{
    MyRow r = new MyRow();
    r.m_region = "new";
    m_sheet.getItems().add(r);
}
```

There is the the “sheet” property of type FIXGRIDListBinding. It is filled with objects of type “MyRow”, that internally provides the properties “region”, “revenue”, etc. These are exactly the properties that are referenced by the FIELD and FORMATTEDFIELD definitions of the layout.

You can also see how objects are added into the grid: MyRow objects are created and simply added to the “sheet” property.

Consequence: on server side, the grid is managed as just normal collection.

## Event Binding

There are two special events that you get notified on server side:

- “Selection of row” - this is either invoked when the user clicks onto a row or when the user navigates via keyboard into a row
- “Execution of row” - this is either invoked when the user double clicks onto a row or when the user presses the “enter key” inside a row.

The events are represented by the protected methods:

- “onRowSelect”
- “onRowExecute”

Both methods are part of the FIXGRIDItem class which is used as base class for the row object.

## Rendering Aspects

The FIXGRID component provides a number of attributes to control the rendering. There are some aspects to be aware of:

- HEIGHT/ SBVISIBLEAMOUNT: the SBVISIBLEAMOUNT defines the number of lines that are shown in the client. This is NOT the maximum number of grid lines, but just the number of lines presented to the user! - If defining SBVISIBLEAMOUNT without defining a HEIGHT, then the visible grid will contain the number of lines as defined by SBVISIBLEAMOUNT. If defining SBVISIBLEAMOUNT and also defining a HEIGHT (e.g. “100%”), then the client may optically reduce the number of visible items, if they do not fit into the area defined by the height. In this case the SBVISIBLEAMOUNT value represents the maximum number of rows transferred to the client.
- BORDERHEIGHT, BORDERWIDTH, BORDERCOLOR: this is the definition of the grid lines that are painted between the grid components.
- BORDER: this is the normal border definition that is taken for the whole grid. If you do not want the grid to be surrounded by a one pixel border, define the value “left:0;right:0;bottom:0;top:0”.
- ROWHEIGHT: this is the minimum height (pixel value) of a grid line. Grid lines may be rendered with a greater height (e.g. is there is too much space, in this case the row height is equally stretched to fill the grid), but never with a lower height.

## Columns Sizing Aspects

The WIDTH attribute of the GRIDCOL by default is responsible for sizing the corresponding column. You can define the following types of values:

- “100” - in this case the column is sized with a width of 100 pixels (in case the client zoom factor is 100%).
- “50%” - in this case the columns receives 50% of the un-occupied width of the whole grid.
- “50%;200” - in this case the column receives 50% of the un-occupied width, but 200 pixels in any case. This definition is very useful: the un-occupied width is the width of the whole grid minus the width that is taken by columns with pixel sizing.

By default the sizing of the grid columns only depends on the WIDTH definition, as shown. If the components within the grid column cells do not fit in, then they are cut accordingly.

But, there is also the possibility to derive the width of a column from its content cells. This is the purpose of the attribute GRIDCOL-DYNAMICWIDTHSIZING. While rendering the grid on client side, the client will measure the size of each cell within a certain column - and the whole column will be sized according to the widest cell.

When using GRIDCOL-DYNAMICWIDTHSIZING you can in addition still maintain the WIDTH attribute. The interpretation is as follows:

- `dynamicwidthsizing="true" width="50"`: the column will be sized according to its content, but will receive 50 pixels as minimum in any case
- `dynamicwidthsizing="true" width="100%"`: the column will be sized according to its content, but will receive 100% of the width in any case.

Please pay attention when using dynamic width sizing;

- The client only knows these data items that are currently loaded on client side. As result it will execute the sizing based on its current content. In case the user scrolls through the items it may happened that the columns, that are defined to support dynamic width sizing, will change their widths.

### **...some more info about “Server side number of Items” versus “Client Side number of Items”**

You can skip this chapter when first time using grids ...or you can immediately read and get some deeper understanding of how the grid internally operates:

The FIXGRID component is able to handle grids with a large amount of items. The simple reason for this: only these items are sent to the client and as consequence are potentially rendered in the client, that are currently relevant in the client.

This means: the grid processing on server side may handle a grid with 10.000 items - but actually only a few of them are sent to the client. This ensures a scalability both from network volume and from rendering performance perspective. - Of course there is some “side-effect”: when the client only knows few of the items, then it has to talk to the server during scroll operations in order to update.

There is one important attribute, ruling all this - which is the attribute FIXGRID-SBVISIBLEAMOUNT. The server will always send the number of items to the client that is defined with SBVISIBLEAMOUNT. So, if SBVISIBLEAMOUNT is defined as “10” and the grid is newly rendered then by default the items with the index 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 are transferred to the client side - only if they are available at all.

The property FIXGRIDBinding-sbvalue is relevant for telling the grid, where the current “area of interest” is. So, at the beginning it is by default “0”, when the user scrolls to a certain position it is updated accordingly. You may also set sbvalue from your server side program in case you want to scroll to a certain item of the grid.

- “Sbvalue” is representing the index of the first top content item of the grid. You may set the property during runtime - if the value is a valid index of you item list.

Now let's take a look onto the rendering. There are two cases:

- If there is NO height defined with the grid, then the grid's size is automatically driven by SBVISIBLEAMOUNT. If SBVISIBLEAMOUNT is defined to be “10” then the grid will have 10 content lines.
- If there IS a height defined with the grid (typically some percentage height, e.g. “100%”) then there is a bit more logic involved on client side. Remember: the grid always receives from the server side a number of items defined by SBVISIBLEAMOUNT. So there are three situations:

- 1. The number of items sent from the server exactly fit into the grid's content area... bingo! No problem at all...
- 2. The content area of the grid is too small to fit the items from the server side. In this case only these items are rendered in the grid, that really fit. So the server may send 10 items, but the client decides to only present 5 of them due to missing optical space on the screen. - There's nothing to worry about, because the user just can scroll to the next items. This scrolling is a “normal”, server side scrolling again.
- 3. The content area of the grid is too big. So the grid offers optical space for a lot of items, but the server only comes with 10 items. In this case the grid distributed the 10 items into its available content area, this means that 10 rows are rendered, each row getting its corresponding height, so that all 10 fit the content area. In other words: the row height of each line is “stretched” to fit.

You see: when explicitly defining a HEIGHT with a grid, you need to think a bit about how to set SBVISIBLEAMOUNT:

- If your grid typically receives a small screen area to fit in, then SBVISIBLEAMOUNT should be defined low as well (e.g. “10”). Imagine your grid receiving a height of e.g. 150 pixels only (due to other controls occupying space) and you having defined SBVISIBLEAMOUNT as 100: in this case always 100 items are sent from the server to the client, just to let the client find out that e.g. 94 of them never fit into the grid... So there is a waste of data transferred and there is some useless processing on client side, which you can avoid by properly setting SBVISIBLEAMOUNT.
- A typical maximum number for SBVISIBLEAMOUNT is in the area of “50”. Which grid will really have more than 50 optical lines, even if the screen is maximized? Of course you might have special situations to also go beyond “50”, but then you should do so with some good reason. ;-)

By the way: the SBVISIBLEAMOUNT of the grid can not be changed once a grid is rendered! You must set it correctly from the beginning on.

---

## Tree Processing

A tree is a normal FIXGRID implementation with three special things:

- Inside the FIXGRIDCOL components you specify a TREENODE component.
- On server side you use FIXGRIDTreeBinding as collection class (instead of FIXGRIDListBinding), and the row class is derived from FIXGRIDTreeItem (instead of FIXGRIDItem).
- Inside the row object you have an additional event “onToggle” that indicated that the user opens/closes a corresponding tree row.

All the other things stay the same: you build the tree on server side, the transfer of the visible rows to the client is automatically done.

Have a look onto the following example:

Country / Town	Inhabitants
Germany	80.000.000
Banmental	6.800
Berlin	3.000.000
Munich	1.000.000
Hamburg	1.200.000
Liechtenstein	20.000
Vaduz	20.000
United Kingdom	60.000.000
London	6.000.000
Manchester	500.000
United States	280.000.000
Los Angeles	5.000.000
New York	9.000.000
San Francisco	1.000.000

The tree is a FIXGRID component containing two GRIDCOL columns: one with a TREENODE component inside, the other one with a FORMATTEDFIELD component:

```
<t:rowdemobodypane id="g_3" objectbinding="demoTree" >
  <t:row id="g_4" >
    <t:fixgrid id="g_5" avoidroundtrips="true" bordercolor="#D0D0D0"
borderheight="0" borderwidth="1" height="100%" objectbinding="#{d.demoTree.tree}"
rowheight="16" sbvisibleamount="25" width="100%" >
      <t:gridcol id="g_6" text="Country / Town" width="100%" >
        <t:treenode id="g_7" />
      </t:gridcol>
      <t:gridcol id="g_8" text="Inhabitants" width="100" >
        <t:formattedfield id="g_9" align="right" format="int" value="{.
{inhabitants}" />
      </t:gridcol>
    </t:fixgrid>
  </t:row>
  <t:row id="g_10" >
    <t:button id="g_11" actionListener="#{d.demoTree.onOpenAllNodes}" text="Open
all" />
    <t:button id="g_12" actionListener="#{d.demoTree.onCloseAllNodes}"
text="Close all" />
  </t:row>
</t:rowdemobodypane>
```

The server side program shows that the grid processing is very similar to the normal grid processing:

```
package workplace;

import java.util.List;

import javax.faces.event.ActionEvent;

import org.eclnt.jsfserver.defaultscreens.Statusbar;
import org.eclnt.jsfserver.elements.impl.FIXGRIDTreeBinding;
import org.eclnt.jsfserver.elements.impl.FIXGRIDTreeItem;

public class DemoTree extends DemoBase
{
  // -----
  // inner classes
  // -----

  public class MyRow extends FIXGRIDTreeItem
  {
    int i_inhabitants;
    public MyRow(FIXGRIDTreeItem parent, String text, int inhabitants,
boolean
isEndNode)
    {
      super(parent);
      setText(text);
      i_inhabitants = inhabitants;
      if (isEndNode)
        setStatus(STATUS_ENDNODE);
    }
    public void setInhabitants(int value) { i_inhabitants = value; }
  }
}
```

```

    public int getInhabitants() { return i_inhabitants; }
    public void onToggle()
    {
        StatusBar.outputMessage("TOGGLE on " + getText());
    }
    public void onRowExecute()
    {
        if (getStatusInt() == STATUS_CLOSED) setStatus(STATUS_OPENED);
        else if (getStatusInt() == STATUS_OPENED) setStatus(STATUS_CLOSED);
    }
    public void onRowSelect()
    {
        StatusBar.outputMessage("SELECT on " + getText());
    }
}

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

FIXGRIDTreeBinding<MyRow> m_tree = new FIXGRIDTreeBinding<MyRow>();

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

public DemoTree()
{
    // fill the tree
    MyRow country;
    MyRow town;
    country = new MyRow(m_tree.getRootNode(), "Germany", 80000000, false);
    town = new MyRow(country, "Bammental", 6800, true);
    town = new MyRow(country, "Berlin", 3000000, true);
    town = new MyRow(country, "Munich", 1000000, true);
    town = new MyRow(country, "Hamburg", 1200000, true);
    country = new MyRow(m_tree.getRootNode(), "Liechtenstein", 20000, false);
    town = new MyRow(country, "Vaduz", 20000, true);
    country = new MyRow(m_tree.getRootNode(), "United
Kingdom", 60000000, false);
    town = new MyRow(country, "London", 6000000, true);
    town = new MyRow(country, "Manchester", 500000, true);
    country = new MyRow(m_tree.getRootNode(), "United
States", 280000000, false);
    town = new MyRow(country, "Los Angeles", 5000000, true);
    town = new MyRow(country, "New York", 9000000, true);
    town = new MyRow(country, "San Francisco", 1000000, true);
}

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

public FIXGRIDTreeBinding<MyRow> getTree() { return m_tree; }
public void onOpenAllNodes(ActionEvent ae)
{
setStatusInFolderNodes(m_tree.getRootNode(), FIXGRIDTreeItem.STATUS_OPENED);
}

public void onCloseAllNodes(ActionEvent ae)
{
setStatusInFolderNodes(m_tree.getRootNode(), FIXGRIDTreeItem.STATUS_CLOSED);
}

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

private void setStatusInFolderNodes(FIXGRIDTreeItem node, int status)
{
    List<FIXGRIDTreeItem> nodes = node.getChildNodes();
    for (int i=0; i<nodes.size(); i++)
    {
        FIXGRIDTreeItem childNode = nodes.get(i);
        if (childNode.getStatusInt() != childNode.STATUS_ENDNODE)

```

```

        {
            childNode.setStatus(status);
            setStatusInFolderNodes(childNode, status);
        }
    }
}

```

Some aspects to put focus on:

- While in the normal grid implementation items are filled into a list by using a corresponding “getItems().add(…)” method, grid items are created by specifying the parent node as parameter of the constructor: “new FIXGRIDItem(parentNode)”.
- Each tree has a root node that is not visible. It is the “0-level” root of the tree. The “FIXGRIDTreeBinding”-object already provides a root node, but you can also pass an own instance through the constructor of “FIXGRDTreeBinding”. The first visible level of nodes is the level below the root node.
- Each node item has two special properties:
  - The “set/getText()” sets the node's text.
  - The “set/getStatus()” tells about the type of node:
    - STATUS\_OPENED is a folder node that is currently opened
    - STATUS\_CLOSED is a folder ndoe that is currently closed
    - STATUS\_ENDNODE is a leaf node of the tree
- Each item has three important methods that you can override for reacting on tree actions:
  - “onToggle()” - this method is called when the user opens/closes a folder node. The status is set correspondingly. You can use this event in order to read a tree step by step: when the user opens a certain area you read the data below the area.
  - “onRowSelect()” and “onRowExecute()” - this is the same as with normal grid processing. The functions are called when the user clicks or double-clicks a certain tree node.

Please have a look into the JavaDoc documentation and into the examples of the demo workplace in order to find more information about implementing trees.

---

## Column Sequence and Column Sizing

This is something for a bit more advanced programmers, but not too complex as well: influencing the sequence of columns in a grid, and influencing the widths of the columns.

### Column Sequence

By default a user can rearrange the columns of a grid on his/her own by dragging and dropping the headers of a column onto one another.

If the user does so, a property will automatically be populated inside the server side FIXGRIDBinding object. The property “columnsequence” then contains a semicolon-separated value which indicates the new sequence of columns.

```

FIXGRIDBinding
  getColumnsequence
  setColumnsequence

```

Example: if a grid has 4 columns, and the user arranges the last column to be the first one, then the value of the property will be:

```
3;0;1;2
```

This means: the first visible column is the original column 3; the second column is the original column 0, etc.

The same property can also be used for setting the column sequence. Thus, you can override the column sequence that is given by the layout definition.

Consequence: if you want to provide personalizable grids to your user, in which the sequence of columns is kept with the user profile, you can do so!

## Column Widths

The original columns widths are taken out of the GRIDCOL definitions that are part of the layout definition.

Something similar, compared to the column sequence management, happens within the column width management: when the user starts to change the columns widths, a property will be populated on server side. The property's name is "modcolumnwidths":

```
FIXGRIDBinding  
getModcolumnwidths  
setModcolumnwidths
```

Now the property contains the widths of the columns, as modified by the user. The property might look like:

```
132;50%;100;50%
```

Note: the sequence of widths is representing the original sequence of columns as defined within the layout definition. The sequence is not dependent on the visible sequence of columns.

When setting the property from your server side program the widths will be applied to the current grid.

## Persistence Management with FIXGRIDs

There is a pre-thought persistence management available - in which the column sequence and column sizes are persisted. The persistence management is invoked when defining a PERSISTID with your FIXGRID. The PERSISTID is the unique used for identifying the corresponding grid and the column data that is stored for this grid.

Please check the interface IFIXGRIDPersistence2 for more information.

---

## Column Sorting

The FIXGRID provides a nice way of sorting its items. There is an attribute SORTREFERENCE that is defined with the GRIDCOL component. When defined, the grid will automatically sort its content once the user clicks with the mouse onto the title of the column.

### The straight forward Way

The SORTREFERENCE is an expression that points to the property that is used for sorting. The expression is built in exactly the same way as expressions which are used within the grid's cell components: ".{xyz}".

Have a look at the following example:

First Name	Last Name	Married	
Alain	Prost	<input checked="" type="checkbox"/>	Remove
Michael	Schumacher	<input checked="" type="checkbox"/>	Remove
Nik	Heidfeld	<input type="checkbox"/>	Remove
Niki	Lauda	<input checked="" type="checkbox"/>	Remove
<input type="button" value="Add row"/>			

The first column indicates that it is sorted in an ascending way. Once the user clicks the column header the sorting will change accordingly.

The JSP layout definition is:

```
<t:fixgrid id="g_5" avoidroundtrips="true" background="#F0F0F0"
bordercolor="#C0C0C0" borderheight="1" borderwidth="1" height="100%"
objectbinding="#{d.demoGrid.rows}" sbvisibleamount="25" width="100%" >
  <t:gridcol id="g_6" sortreference=".{firstName}" text="First Name"
width="50%" >
    <t:field id="g_7" text=".{firstName}" />
  </t:gridcol>
  <t:gridcol id="g_8" sortreference=".{lastName}" text="Last Name"
width="50%" >
    <t:field id="g_9" text=".{lastName}" />
  </t:gridcol>
  <t:gridcol id="g_10" sortreference=".{married}" text="Married" width="50" >
    <t:checkbox id="g_11" align="center" background="#FFFFFF" selected="
{married}" />
  </t:gridcol>
  <t:gridcol id="g_12" width="80" >
    <t:button id="g_13" actionListener=".{onRemove}"
contentareafilled="false" text="Remove" />
  </t:gridcol>
</t:fixgrid>
```

You see:

- Only the first three columns are sortable - only for those columns a SORTREFERENCE attribute is defined.
- The SORTREFERENCE attribute points to the same property that is referenced within the cell components.

When looking at the code you will see that all sort operations by default are provided without affecting your code:

```
package workplace;

import javax.faces.event.ActionEvent;

import org.eclnt.jsfserver.elements.impl.FIXGRIDItem;
import org.eclnt.jsfserver.elements.impl.FIXGRIDListBinding;

public class DemoGrid extends DemoBase
{
    // -----
    // inner classes
    // -----

    public class MyRow extends FIXGRIDItem
    {
        String i_firstName;
        String i_lastName;
        boolean i_married;
        // -----
        public void setFirstName(String value) { i_firstName = value; }
        public String getFirstName() { return i_firstName; }
        public void setLastName(String value) { i_lastName = value; }
        public String getLastName() { return i_lastName; }
        public void setMarried(boolean value) { i_married = value; }
        public boolean getMarried() { return i_married; }
    }
}
```

```

// -----
public void onRemove(ActionEvent ae)
{
    m_rows.getItems().remove(this);
}
}

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

FIXGRIDListBinding<MyRow> m_rows = new FIXGRIDListBinding<MyRow>();

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

public DemoGrid()
{
    // create some dummy items
    MyRow row;
    row = new MyRow();
    row.setFirstName("Michael");
    row.setLastName("Schumacher");
    row.setMarried(true);
    m_rows.getItems().add(row);
    row = new MyRow();
    row.setFirstName("Alain");
    row.setLastName("Prost");
    row.setMarried(true);
    m_rows.getItems().add(row);
    row = new MyRow();
    row.setFirstName("Niki");
    row.setLastName("Lauda");
    row.setMarried(true);
    m_rows.getItems().add(row);
    row = new MyRow();
    row.setFirstName("Nik");
    row.setLastName("Heidfeld");
    row.setMarried(false);
    m_rows.getItems().add(row);
}

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

public FIXGRIDListBinding<MyRow> getRows() { return m_rows; }

public void onAddRow(ActionEvent ae)
{
    MyRow row = new MyRow();
    row.setFirstName("new");
    row.setLastName("new");
    m_rows.getItems().add(row);
}
}

```

The default sorting checks that data type of a property during sorting. The following data types are supported:

- String
- boolean, int, long, float, double
- Data
- BigInteger, BigDecimal

All other types are converted into strings (by using `.toString()` method), the sorting then will be done using those strings.

## If SORTREFERENCE is undefined...

If you do not explicitly define the SORTREFERENCE as shown in the previous text then the system tries to find it on its own. It takes a look into the component that is directly located within the GRIDCOL element - which is the cell component. Then it checks for values available within the TEXT, VALUE or other attributes and uses the corresponding expression as SORTREFERENCE.

This is the reason why you typically do not have to define a SORTREFERENCE when just defining a LABEL or a FIELD as cell component.

But, please note: this automatic “guessing” is only executed on the top most component within a GRIDCOL.

## Directly setting the Sort Status

For each grid columns that explicitly or implicitly holds a SORTREFERENCE you can access the sort reference object in the following way:

```
public interface IFIXGRIDBinding<itemClass extends FIXGRIDItem>
{
    /**
     * Sort information that is kept per sortable column.
     */
    public static class FIXGRIDSortInfo
    {
        int i_sortStatus = SORT_UNSORTED;
        public String getSortStatus()
        {
            switch (i_sortStatus)
            {
                case SORT_UNSORTED:
                    return null;
                case SORT_ASCENDING:
                    return "ascending";
                case SORT_DESCENDING:
                    return "descending";
            }
            throw new Error("Should never happen: " + i_sortStatus);
        }
        public void setSortStatus(int sortStatus)
        {
            i_sortStatus = sortStatus;
        }
        ...
    }

    static final int SORT_UNSORTED = 0;
    static final int SORT_ASCENDING = 1;
    static final int SORT_DESCENDING = 2;

    ...
    ...
    ...

    public FIXGRIDSortInfo getSortInfoForReference(String sortReference);
}
```

The server side grid collections (FIXGRIDListBinding or FIXGRIDTreeBinding) support the interface IFIXGRIDBinding. IFIXGRIDBinding includes the definition of a FIXGRIDSortInfo class, and it allows to access the sort info by using the method “getSortInfoForReference”.

By using the method “setSortStatus()” you may initialize the sort status of a column on your own. Maybe you already read the data from the database in a sorted way and you want to pass this information on to the user.

Please pay attention: “getSortInfoForReference(…)” is the only correct way to access the FIXGRIDSortInfo object - do not directly access the map of FIXGRIDSortInfo objects that is

available as well!

And: the value that you pass as “sortReference” parameter is the explicit or implicit value of the attribute SORTREFERENCE, without expression-prefix and postfix. This means: if the SORTREFERENCE is defined as “.{lastName}” then the value you pass as parameter is “lastName”.

## Influencing the default Sorting

The default sorting scans the values of certain grid column and then compares them using the Java sorting that is part of “java.util.Collections”. For comparing the values it checks the data type of the values and chooses a corresponding comparator. Example: if the values are of type “Integer”, then a numeric comparator is chosen - if the values are of type “String” then a lexical Comparator is used.

You can explicitly define the comparator to be used during sorting by overriding the grid's method “FIXGRIDBinding.findSortComparatorForColumnName(...)”:

```
protected Comparator findSortComparatorForColumnName(String sortReference)
{
    ...
    // select your own Comparator for the corresponding sortReference
    // or passe super.findSortComparatorForColumnName(sortReference) for
    // using the default comparator
    ...
}
```

Example for using this method: the data type of column values for a certain reason (e.g. accessing the database in a simple way) is “String”, but the contained data actually is of type integer. The default sorting would use a lexical comparator, while you tell to use a numeric one.

## Implementing an own Sort Algorithm

In some cases you may want to implement your own algorithm for sorting. Maybe you do not want to sort the grid within memory but want to re-read the grid from the database, passing new sort commands with the SQL.

In this case you need to be aware of what internally happens when the user clicked with the mouse onto a sortable column:

- The client sends a request to the server processing.
- Within the server an ActionEvent of type BaseActionEventGridSort is created.
- The event is passed on to the action listener of the grid collection. This is the method FIXGRIDBinding.onGridAction() (FIXGRIDBinding is the parent of FIXGRIDListBinding and FIXGRIDTreeBinding).
- The method sortGrid() is called.

```
protected abstract void sortGrid(String sortReference,
                                String objectBindingString,
                                boolean ascending);
```

- The method receives the following information:
  - The sortReference itself.
  - The OBJECTBINDING string that is defined in the FIXGRID component's OBJECTBINDING attribute.
  - The information if the sorting needs to be done in an ascending or descending order.

You can override the `sortGrid()` method within an own sub-class of `FIXGRIDListBinding` (or `FIXGRIDTreeBinding`) and implement your own sort algorithm. Within this sort algorithm you can perform any grid operation you like: you can remove grid items, add new ones, etc.

## Multiple Column Sorting

In addition to the sorting by one column there is the so called multiple column sorting. The user invokes this sorting by holding down the `ctrl`-key when clicking onto the column headers of the grid. The sequence in which the user clicks the header defines the priority of sorting.

The number of sort icons that is shown in the column header gives a visual feed back of the sort sequence:

First Name <sup>^^^</sup>	Last Name <sup>^^</sup>	Birthday <sup>^</sup>	Mar.	
Niki	Lauda	▼	<input checked="" type="checkbox"/>	Remove
Nik	Heidfeld	▼	<input type="checkbox"/>	Remove
▶ Michael	Schumacher	23.07.2012 ▼	<input checked="" type="checkbox"/>	Remove
Alain	Prost	23.07.2012 ▼	<input checked="" type="checkbox"/>	Remove

In the example the first sort priority is the birthday, the second priority is the last name - followed by the first name.

### Technical Info behind

The multiple column sorting is managed on top of the normal column sorting.

The class `FIXGRIDSortInfo` that was introduced in the previous chapter holds a property "sortSequence" - with "0" being the first priority, "1" the seconds priority, etc.

The actual sorting is done by the method `FIXGRIDBinding.processMultipleSort()`. So this is the one to override when implementing own algorithms for multiple sorting.

The default implementation of the `processMultipleSort()` method internally calls the single column sorting (i.e. method `FIXGRIDBinding.sortGrid(...)`) multiple times - starting with the lowest sort priority first and then going back to the highest priority. The expectation behind is that the single column sorting algorithm is a stable sort. As consequence the result of processing one sort after the next is the correct multi-column sorting that you expect.

(Of course the default column sorting provided by CaptainCasa is a stable one, so there is no need to adapt anything!)

### Usage from Server Side Java

You can directly sort the content of the grid by calling the following methods of `FIXGRID(List/Tree)Binding`:

```
getSortInfo().clear();
FIXGRIDSortInfo fsi;
fsi = getSortInfoForReference("lastName");
fsi.setSortSequence(0);
fsi.setSortStatus(SORT_ASCENDING);
fsi = getSortInfoForReference("firstName");
fsi.setSortSequence(1);
fsi.setSortStatus(SORT_ASCENDING);
```

```
resort();
```

The steps are:

- You first clear all sort status information that is currently available. (If there is a fresh grid you do not need to do this step.)
- Then you define the sort information per column. For each definition you have to define the access string (expression without any leading or closing brackets), the sort status and the sort sequence (0 for first rank, 1 for seconds, ...).
- By calling resort() the sorting is applied to the current grid.

When initially sorting a grid then you need to pay attention to, that the grid must be bound to its component (FIXGRID) - otherwise the sorting will not work. The binding to the component with a fresh grid is done when the grid is processed in the JSF render phase the first time (this is when the page's XML is processed).

As consequence you need to shift the sort code into the initialize() method which is called when the component binds to the FIXGRIDBinding object. You need to override your FIXGRIDList/TreeBinding, e.g. in the following way:

```
public class MyFIXGRIDListBinding extends FIXGRIDListBinding<MyRow>
{
    public void initialize()
    {
        super.initialize();
        // this is how to sort by one column
        {
            sort("firstName",true);
        }
        // this shows how to sort by multiple columns
        {
            getSortInfo().clear();
            FIXGRIDSortInfo fsi;
            fsi = getSortInfoForReference("lastName");
            fsi.setSortSequence(0);
            fsi.setSortStatus(SORT_ASCENDING);
            fsi = getSortInfoForReference("firstName");
            fsi.setSortSequence(1);
            fsi.setSortStatus(SORT_ASCENDING);
            resort();
        }
    }
}
```

## Switching Off Multiple Column Sorting

In cases in which you want to switch off multi column sorting, just set the attribute FIXGRID-MULTICOLUMNSORT to "false" - then only single column sorting will be active.

---

## Focus Issues

### Setting the Focus into Grid Line

The typical way of setting the client keyboard focus to a certain component is by using the component attribute REQUESTFOCUS. The mechanism described in chapter "Rules that apply to all Components - Focus Management" is of course applicable for all components that you insert into grid rows as well.

There is an additional simplification function available in order to move the client focus into the first cell of a grid row. You simply need to call the function "FIXGRDBinding.selectAndFocus(FIXGRIDItem)".

Please take a look at the following example:

```
public class DemoGrid extends DemoBase
{
    // -----
    // inner classes
    // -----

    public class MyRow extends FIXGRIDItem
    {
        String i_firstName;
        String i_lastName;
        boolean i_married;
        // -----
        public void setFirstName(String value) { i_firstName = value; }
        public String getFirstName() { return i_firstName; }
        public void setLastName(String value) { i_lastName = value; }
        public String getLastName() { return i_lastName; }
        public void setMarried(boolean value) { i_married = value; }
        public boolean getMarried() { return i_married; }
        // -----
        public void onRemove(ActionEvent ae)
        {
            m_rows.getItems().remove(this);
        }
    }

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

    FIXGRIDListBinding<MyRow> m_rows = new FIXGRIDListBinding<MyRow>(true);

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

    public DemoGrid()
    {
    }

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

    public FIXGRIDListBinding<MyRow> getRows() { return m_rows; }

    public void onAddRow(ActionEvent ae)
    {
        // add the item
        MyRow row = new MyRow();
        row.setFirstName("new");
        row.setLastName("new");
        m_rows.getItems().add(row);
        // select the item
        m_rows.deselectCurrentSelection();
        m_rows.selectAndFocusItem(row);
    }
}
}
```

You see that in the method “onAddRow” a new item is added to the grid. After adding the item first the current selection of the grid is removed. Then the method “selectAndFocusItem” is called. This method performs 3 steps:

- It selects the item so that it is marked with a certain background.
- It ensures that the item is visible within the grid - this means, that the grid e.g. is scrolled if the item is currently within a grid area that is not visible on client side.
- And it moves the focus into the grid row.

## Avoiding Line-Selection on Focus

By default lines of a grid are selected by the user in the following ways:

- A. The user clicks with the mouse into a line.
- B. The user presses a key within a line.

- C. The user somehow focuses a component within the line.

In certain constellations option C. is not adequate for line selection. E.g. when opening a popup dialog from a grid line and when changing the selection of the grid in the dialog, then the default behavior of the popup dialog when being close is to focus the component it was started from. If this is the grid - with some changed line selection - then the focus change will automatically select the line which was focused before opening the popup dialog.

You can switch off the option C. as consequence. Use attribute `FIXGRID-SELECTONROWFOCUS` and set value "false".

## Extended Grid Functions

### Column Sequence

You may open a default dialog in which the user can explicitly define the sequence of columns and the visibility of columns.

#### *Using the Columns Sequence Popup*

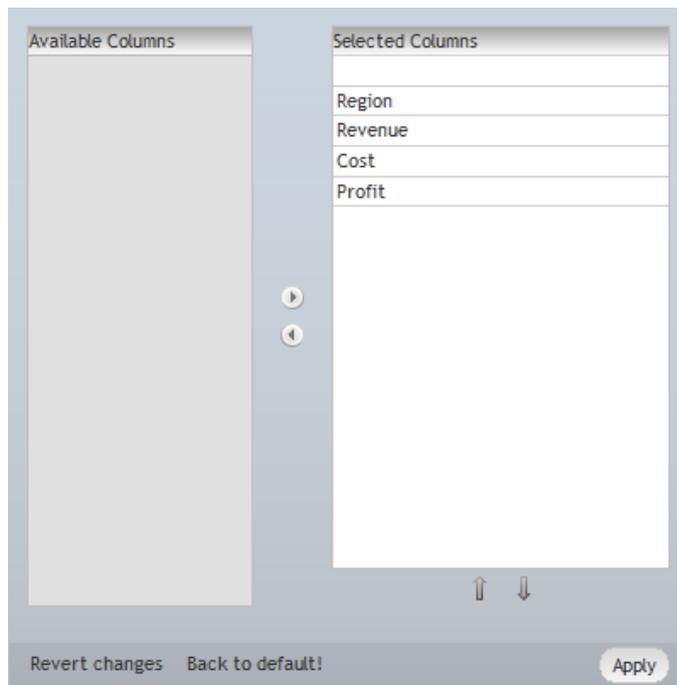
The easiest way to include the columns sequence popup into your screen is by simply adding an invoker component (e.g. button, link) and referencing via expression to the "onOpenGridDetails"-method, that is directly available with all `FIXGRIDBinding` implementations on server side.

Example:

```
<t:row id="g_2">
  ...
  ...
  <t:link id="g_8"
    actionListener="#{d.DemoMinisread.sheet.onEditColumnDetails}"
    text="Columns..." />
  ...
</t:row>
<t:row id="g_9">
  <t:fixgrid id="g_10" avoidroundtrips="true" background="#FFFFFF80"
    bordercolor="#D0D0D0" borderheight="1" borderwidth="1"
    cellselection="true" drawoddevenrows="true" emptycolor="#F0F0F0"
    height="100%" multiselect="true"
    objectbinding="#{d.demoMinisread.sheet}" sbvisibleamount="20"
    width="100%">
  ...
  ...
```

You see: the grid (in `t:fixgrid`) is bound by using expression "`#{d.demoMinisread.sheet}`" - referencing on server side to `FIXGRIDListBinding` object. The method "onEditColumnsDetails" is directly available within this object.

As result the following popup dialog will be opened:



## Search & Export

The server side grid management provides a couple of useful functions:

- Search of text within grid data
- Export of grid data into various formats

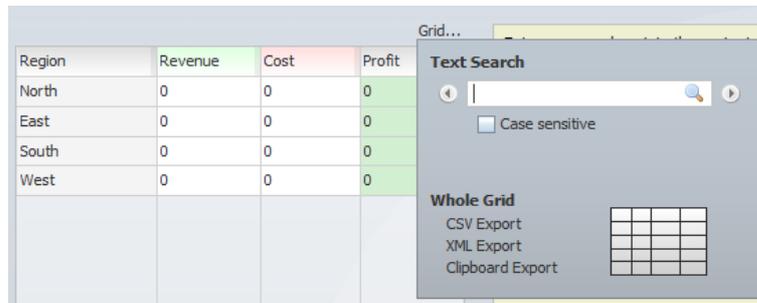
### *Using the Default Popup*

The easiest way to include these functions is to embed the method “FIXGRIDBinding.onOpenGridFunctions(ActionEvent event)” into your screen, e.g. by providing a corresponding LINK or BUTTON component.

Example: in the mini-budget demo of the demo workplace, the function is added as LINK component:

```
<t:row id="g_2">
  <t:col distance id="g_3" width="100%" />
  <t:link id="g_4"
    <u>actionListener="#{d.DemoMinispread.sheet.onOpenGridFunctions}"
    </u>
    text="Grid..." />
</t:row>
<t:row id="g_5">
  <t:fixgrid id="g_6" avoidroundtrips="true" background="#FFFFFF80"
    bordercolor="#D0D0D0" borderheight="1" borderwidth="1"
    cellselection="true" emptycolor="#F0F0F0" height="100%"
    multiselect="true" objectbinding="#{d.demoMinispread.sheet}"
    sbvisibleamount="20" width="100%">
    ...
    ...
    ...
  </t:fixgrid>
</t:row>
```

Result: a popup dialog will be opened once the user presses the LINK component:



The extended functions are provided within the popup.

### Using the APIs

There are a couple of server side Java APIs that are provided by the FIXGRIDBinding class, which is the base for all server side grid implementations. Please check the JavaDoc documentation in the following areas:

- FIXGRIDBinding.getTextSearcher()
- FIXGRIDBinding.getExporter()

All the functions that are provided in the default popup are available by an internal API as well.

### “Internal” Details...

In case you do not like the default dialogs that are opened for defining the grid column sequence and/or the search & export dialog: here are some “internals”:

The code in FIXGRIDBinding for open the popup dialogs is:

```
public void onEditColumnDetails(ActionEvent ae)
{
    final DefaultScreens ds = DefaultScreens.getSessionAccess().getOwner();
    final GridDetails gd = DefaultScreens.getSessionAccess().getGridDetails();
    IGridDetailsListener gl = new IGridDetailsListener()
    {
        public void reactOnApplied()
        {
            ds.closePopup(gd);
        }
        public void reactOnClose()
        {
            ds.closePopup(gd);
        }
    };
    gd.prepare(this,gl,true,GridDetails.PAGENAME_COLUMNSDETAILS);
    ModelessPopup popup = ds.openModelessPopup(gd,"",400,400,new
    IModelessPopupListener()
    {
        public void reactOnPopupClosedByUser()
        {
            ds.closePopup(gd);
        }
    });
    popup.setUndecorated(true);
    popup.setCloseOnClickoutside(true);
    popup.setStartfromrootwindow(false);
}
```

A page bean “DefaultScreens” is picked and opened as modeless dialog. The page bean is initialized using its “prepare(...)” method. The last parameter that is passed into the prepare method is the name of the .jsp-page that is to be used.

There are two static variables that are used:

- GridDetails.PAGENAME\_COLUMNDETAILS (default “”) (default “/ecIntjsfserver/popups/griddetails.jsp”)
- GridDetails.PAGENAME\_FUNCTIONS (default “/ecIntjsfserver/popups/gridfunctions.jsp”)

### Using own Page Definitions

In case you want to use other, own pages there are two ways:

- You may simply set the two static members of GridDetails, so that they contain new default values.
- You may override the FIXGRIDBinding class that you are using and implement own “onXYZ”-methods that you call.

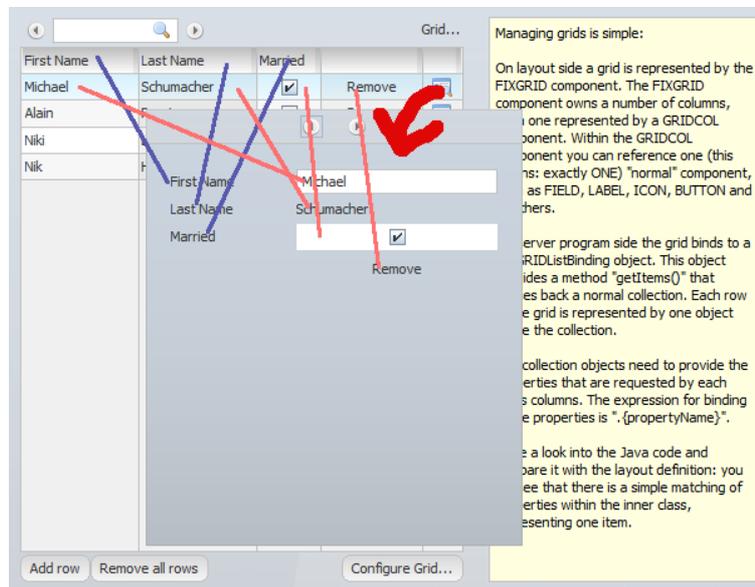
### Using own Page Beans

In case you may use own dialogs and not only applying optical changes to the default dialogs, but also adding new functions - you also need to somehow influence the way that the page bean for the dialogs is picked.

You do so by calling the static method “DefaultScreens.Factory.initClassGridDetails(..)”. Include the calling of this method into the start up process of you application.

## Row Detail Popup

The grid management provides a nice function: it is able to render the currently selected item into a popup window. All the item data is available as vertical list within this window - using exactly the same components as are used inside the grid row:



Opening this popup is very simple: you just need to call...

- FIXGRIDBinding.getRowDataUI().onOpenRowDataPopup() - in order to list all the columns that are currently visible. Columns that the user does not want to see are not taken over.
- or FIXGRIDBinding.getRowDataUI().onOpenRowDataPopupAllColumns() - in order to list all columns that are defined in the layout definition

Again you can call the popup via an API and add own functions: `FIXGRIDBinding.getRowDataUI().openPopup(...)` will pass you back the popup window, so that you e.g. can update its size or position according to your requirements.

---

## Grid Popup Menu

The grid's popup menu processing ("right mouse click menu") is based on the default popup menu processing of CaptainCasa Enterprise Client: you need to define a `POPUPMENU` with a certain id within your page. This `POPUPMENU` then is referenced by the grid components.

There are three possibilities to define a popup menu within a grid definition:

- You of course can assign a `POPUPMENU` component to just normal grid cell components - just using the normal attribute `POPUPMENU`. In this case the `actionListener` of the corresponding component will receive a corresponding event of type `BaseActionEventPopupMenuItem` when the user selected a popup menu item. Example:

```
FIXGRID ...
  GRIDCOL text="..." ...
    FIELD text="..." popupmenu="..." actionListener="{...}" ...
```

- If you want to have on `POPUPMENU` processing for a whole line then you can define this in the following way:

```
FIXGRID ... rowpopupmenu="..."
```

The corresponding popup menu will be shown when the user presses the right mouse button on any cell of a line. As reaction on server side you need to override the method `FIXGRIDItem.onPopupMenu(...)` within your item-class.

- ...and finally you may define the popup menu for the whole grid by specifying:

```
FIXGRID ... popupmenu="..."
```

A typical usage scenario for a grid is to use `FIXGRID-ROWPOPUPMENU` and `FIXGRID-POPUPMENU` in parallel:

- If the user clicks on an item then the popup menu assigned via `ROWPOPUPMENU` will be shown.
- If the user clicks on some empty space within the grid then the popup menu assigned via `POPUPMENU` will be shown.

Typically the `ROWPOPUPMENU` definition (i.e. contained menu items) includes the `POPUPMENU` definition: if the grid is fully occupied with line items, there is no empty space left, and as consequence there is no way for the user to reach the `POPUPMENU` items if not defined in the `ROWPOPUPMENU` definition as well.

---

## Saving and Restoring a Grid's Runtime State

During runtime a user may update the grid in various ways:

- he/she may re-arrange the columns
- he/she may resize columns
- he/she may sort the grid

A common function that is required is to save this information and to restore it if the user revisits the grid.

Of course you can implement such function straight on your own by just using the grid's API. But there is a shortcut to simplify the default scenarios - which is activated by assigning a PERSISTID to the corresponding grid:

## Implicit or explicit Storing

In the FIXGRIDBinding there are two protected methods:

```
protected void storePersistentDataImplicitly()
{
    storePersistentData();
}

protected void storePersistentData()
{
    ...
}
```

Every time a grid's structure is updated, the method “storePersistentDataImplicitly” is called. This method by default directly calls the method “storePersistentData” - which takes the current grid configuration and stores it. This means every time the user e.g. re-arranges the columns of a grid the grid structure is immediately stored.

If you do not want the storing to be implicitly done, then just override “storePersistentDataImplicitly” and do ...nothing! (Maybe you want to maintain a change indicator or do something else...).

## Interface IFIXGRIDPersistence2

The method “storePersistentData()” internally finds an instance of an implementation of the interface IFIXGRIDPersistence2.

```
public interface IFIXGRIDPersistence2
{
    public PersistentInfo readPersistentInfo(FacesContext context,
                                           FIXGRIDBinding gridBinding,
                                           String pageName,
                                           String persistId);

    public void updatePersistentInfo(FacesContext context,
                                    FIXGRIDBinding gridBinding,
                                    String pageName,
                                    String persistId,
                                    PersistentInfo persistentInfo);
}
```

The finding is done through the system.xml configuration file:

```
<fixgrid
  persistence=
    "org.eclnt.jsfserver.util.fixgridpersistence.DefaultFIXGRIDPersistence"
  ...
/>
```

The default class to handle the persistence id “DefaultFIXGRIDPersistence”. This class transforms the current grid configuration into an XML string and stores it within the streamstore.

In many scenarios you just want to use the default persistence and manage the actual persisting of the XML on your own. In this case the only thing you need to do it to override two methods:

```
protected String readSerializedPersistentInfo(FacesContext context,
                                             FIXGRIDBinding gridBinding,
                                             String pageName,
                                             String persistId)
```

```

    {
        String xml = ....read the XML...
        return xml;
    }

    protected void saveSerializedPersistentInfo(FacesContext context,
                                                FIXGRIDBinding gridBinding,
                                                String pageName,
                                                String persistId,
                                                PersistentInfo persistentInfo,
                                                String xml)
    {
        ...store XML...
    }

```

## Performance Optimization of Grids

In general you do not have to worry about grid performance - both client and server-side. But, there may come up situations in which you should, such as:

- Grids with many columns (many meaning > 30)
- Several grids with many columns displayed at the same point of time

### Grid Performance

Before getting into details about how to optimize, first have a look at the performance aspects of grid processing.

The FIXGRID - GRIDCOL - etc. component combination, that forms a grid actually is multiplied out at runtime: for each row of the grid there is a corresponding row, each row containing exactly these components that are defined below the GRIDCOL definition.

While multiplying out the expressions of all components are concatenated correspondingly.

Example: in the FIXGRID definition there is an expression defined in the attribute OBJECTBINDING, e.g. “#{abc.grid}”. This expression points to the FIXGRIDBinding object on server side. Below the GRIDCOL definition there might be a FIELD definition, the TEXT attribute of FIELD pointing to “.{firstName}”. For each row, a FIELD component will be multiplied out, the expressions of the components will be:

```

<fixgrid objectbinding="#{abc.grid}">
</fixgrid>

Row 1:    #{abc.grid.rows[0].firstName}
Row 2:    #{abc.grid.rows[1].firstName}
Row 3:    #{abc.grid.rows[2].firstName}
etc.

```

During JSF request processing all expressions that are referenced within the current component tree are processed. Now imagine a grid with a SBVISIBLEVALUE of “30” and a column count of “50”: in this case 1500 expressions are evaluated on server side with every request.

Remember: only those rows are processed which are actually visible on client side, i.e. not all rows of the grid are processed, but still this is quite a lot to do.

### “Change Index” Optimization

The optimization is quite simple: each grid row explicitly tells if it has changed during a request processing or not. In case it has not changed, the components of the grid row are not processed at all. This means also the corresponding resolution of expressions is not

done.

You see: the application on server side now has to “help” the surrounding component framework, by telling that certain components do not need to be processed.

There are two things you need to do in order to optimize the grid processing:

- (1) You need to declare the fact that a grid is optimized
- (2) You need to tell about changes in grid items.

The first part is simple. Just use the constructor of `FIXGRIDListBinding` or `FIXGRIDTreeBinding`, that allow to pass along the boolean flag “changeIndexIsSupported”. E.g. for `FIXGRIDListBinding` call the constructor in the following way:

```
FIXGRIDListBinding<MyRow> m_grid = new FIXGRIDListBinding(true);
```

The second aspect is something you really need to pay attention to: every time a grid row's content is updated you need to tell the grid line about it. If you do not tell, the component processing will assume that nothing has changed within the data of the grid line. You tell the change of data in the following way:

```
FIXGRIDItem row = ...;
row.getChangeIndex().indicateChange();
```

Please note: you just have to call the `indicateChange()` method in case that an existing item was updated. You do not need to notify about:

- Moving one item within the grid to a new position
- Adding items, removing items
- Clearing the grid and filling it with new data
- Selecting / de-selecting grid lines

All this is recognized as change internally. (BTW: Calling the “`indicateChange()`” method too often will never cause an error on server side...)

## “Change Index” Optimization Scenarios

Optimization is never a problem in some simple scenarios:

- Read-only grids - if no data is changeable in the grid, then you just can define the grid to be optimized, that's it.
- Grids in which the only change of data is caused by the user interface. This means: no data inside the grid is calculated out of user input data. In this case no active changing of data is done. Again, you just need to define the grid to be optimized, that's it.

## “Data Bag” Optimization

If your grid has a lot of columns (e.g. > 100 columns) then the following optimization makes sense for all read-only values:

- On Grid Item level you provide a so called “data bag” - which is a String-serialized map, containing all the read only values of one item in the format “name1;value1;name2:value”;...”
- The grid cell controls refer to the “data bag” by referencing the corresponding name in the following way “@cc\_db:<name>@”

Consequence: all the data is picked from the item in one chunk - and transferred to the client in one chunk. The distribution of the data from “data bag” into the corresponding

cell components is done on client side.

Example + How to implement:

- In the grid item class of your grid you need to implement the interface “IDatabagProvider”:

```
...
public class GridItem extends FIXGRIDItem implements IDatabagProvider
{
    String i_firstName;
    String i_lastName;
    String i_databag;
    public String getDatabag()
    {
        if (i_databag == null)
        {
            Map<String,String> m = new HashMap<String,String>();
            m.put("firstName",i_firstName);
            m.put("lastName",i_lastName);
            i_databag = ValueManager.encodeComplexValue(m);
        }
        return i_databag;
    }
    public String getFirstName() { ... } // required for sorting!
    public String getLastName() { ... } // required for sorting!
}
}
```

You see: the databag collects all the information in one string which is normally picked from individual properties.

- In the FIXGRID definition you define the grid in the following way:

```
...
FIXGRID ...
GRIDCOL SORTREFERENCE="{firstName}" ...
LABEL TEXT="@cc_db:firstName@" ...
GRIDCOL SORTREFERENCE="{lastName}" ...
LABEL TEXT="@cc_db:lastName@" ...
...
```

So, instead of binding attributes to expressions you need to bind them to names of the “data bag”.

That's it!

Typically the “data bag” optimization is used in grids which are generated using (ROW)DYNAMICCONTENTBINDING. There is a corresponding example in the demo workplace (search for text “Grid Performance”) in which the “data bag” generation is demonstrated.

Please pay attention:

- The “data bag” is usable for read-only attributes only. E.g. you must not bind a FIELD's TEXT-attribute to a “data bag” name!
- If not showing some columns of a grid (e.g. due to user selection of visible columns) then the corresponding data of the column is not transferred to the client at all. With the “data bag” you need to take special care about this! By default the whole “data bag” is transferred to the client side - regardless if the contained pieces of information are referenced by a component or not.
- When using “data bag” then you need to explicitly define the GRIDCOL-SORTREFERENCE so that sorting still works on server side.

# REPEAT Component

In the previous chapter you got to know the FIXGRID component as one way of rendering information, in which you have a collection of items to be presented to the user.

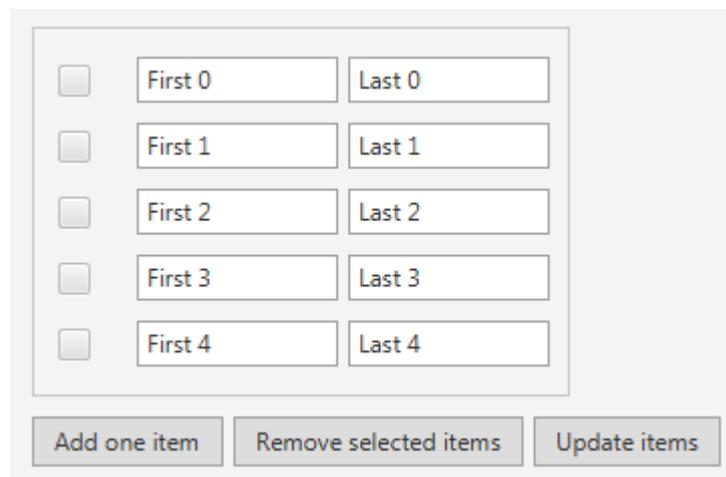
The advantage of the FIXGRID component: it allows a powerful and flexible handling of cell-oriented grids. The disadvantage: if you want to present the item information to the user in some more heterogeneous way, then things are a bit more difficult. - In the FIXGRID component cells are primarily sized by the grid processing (“outside-in sizing”). And in the FIXGRID you think in homogeneous cells.

As consequence there is an alternative: the REPEAT component. This component allows a flexible way of presenting collection items to the user, by just repeating a defined sequence of components for each item of the collection.

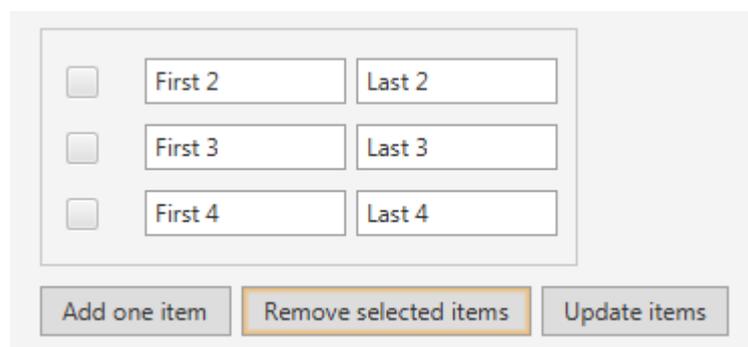
---

## Usage of the REPEAT Component

The usage of the REPEAT component is very simple. Take a look onto the following page:



When selecting the first 2 items and pressing “Remove selected items” the screen looks like:



The layout definition is:

```
<t:row id="g_2">  
  <t:pane id="g_3" border="#00000030" padding="10">  
    <t:repeat id="g_4" listbinding="{#d.DemoRepeat.items}">  
      <t:rowdistance id="g_5" height="5" />
```

```

        <t:row id="g_6" coldistance="5">
            <t:checkbox id="g_7" selected="{selected}" />
            <t:coldistance id="g_8" width="5" />
            <t:field id="g_9" text="{firstName}" width="100" />
            <t:field id="g_10" text="{lastName}" width="100" />
        </t:row>
        <t:rowdistance id="g_11" height="5" />
    </t:repeat>
</t:pane>
</t:row>
<t:rowdistance id="g_12" height="10" />
<t:row id="g_13" coldistance="5">
    <t:button id="g_14"
        actionListener="#{d.DemoRepeat.onAddItemAction}"
        text="Add one item" />
    <t:button id="g_15"
        actionListener="#{d.DemoRepeat.onRemoveItemsAction}"
        text="Remove selected items" />
    <t:button id="g_16"
        actionListener="#{d.DemoRepeat.onUpdateItemsAction}"
        text="Update items" />
</t:row>

```

You see:

- In the layout definition there is a REPEAT component that binds to a “#{d.DemoRepeat.items}” expression.
- Within the REPEAT component's content there are just normal controls (e.g. FIELD). The controls bind to server side properties using the “{...}” expression notation.

This means the content of the REPEAT component is repeated for each single item of the list that is bound via the attribute REPEAT-LISTBINDING. Inside the REPEAT the properties of the items are reference by “{...}” expressions.

The server-side Java code is:

```

package workplace;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

import org.eclnt.editor.annotations.CCGenClass;
import org.eclnt.jsfserver.pagebean.PageBean;
import org.eclnt.workplace.IWorkpageDispatcher;

import javax.faces.event.ActionEvent;

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

public class DemoRepeat
    extends DemoBasePageBean
    implements Serializable
{
    // -----
    // inner classes
    // -----

    public class Item
    {
        boolean m_selected = false;
        String i_firstName;
        String i_lastName;
        public String getFirstName() { return i_firstName; }
        public void setFirstName(String firstName) { i_firstName = firstName; }
        public String getLastName() { return i_lastName; }
        public void setLastName(String lastName) { i_lastName = lastName; }
        public boolean getSelected() { return m_selected; }
        public void setSelected(boolean value) { this.m_selected = value; }
    }

    // -----
    // members

```

```

// -----
List<Item> m_items = new ArrayList<Item>();

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

public DemoRepeat(IWorkpageDispatcher dispatcher)
{
    super(dispatcher);
    for (int i=0; i<5; i++)
    {
        Item it = new Item();
        it.i_firstName = "First " + i;
        it.i_lastName = "Last " + i;
        m_items.add(it);
    }
}

public String getPageName() { return "/workplace/demorepeat.jsp"; }
public String getRootExpressionUsedInPage() { return "#{d.DemoRepeat}"; }

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

public List<Item> getItems() { return m_items; }

public void onAddItemAction(ActionEvent event)
{
    Item it = new Item();
    it.i_firstName = "New item";
    it.i_lastName = "New item";
    m_items.add(it);
}

public void onRemoveItemsAction(ActionEvent event)
{
    for (int i=m_items.size()-1; i>=0; i--)
    {
        Item item = m_items.get(i);
        if (item.getSelected() == true)
        {
            m_items.remove(i);
        }
    }
}

public void onUpdateItemsAction(ActionEvent event)
{
    for (Item item: m_items)
    {
        item.i_firstName += "#";
        item.i_lastName += "#";
    }
}

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

```

The list of items (m\_items) that is used for the REPEAT processing is just a normal instance of java.util.List - here it is an ArrayList. The items' class is an inner class in the example - but may be a “normal” class as well, of course.

By updating the items - either the number of items or the content of the items - the REPEAT component automatically follows and adapts the rendering accordingly.

---

## Nesting REPEAT Components

It is possible to nest one REPEAT component inside an other. Please check the corresponding example within the Demo Workplace:

Department	001
Name	Human Resources
<b>Employee Id</b>	<b>Employee Name</b>
0001	John Wayne
0002	Mark Twain
Department	002
Name	Sales
<b>Employee Id</b>	<b>Employee Name</b>
0003	Hugo Egon Balder
0004	Wolfgang Ambross
0005	Loriot
Department	003
Name	Production
<b>Employee Id</b>	<b>Employee Name</b>
0006	Henry Ford
0007	Robert Bosch

The principle behind is simple:

- The outer REPEAT binds to a certain list (in this case the list of departments).
- Each department item itself provides a list of employees - which is bound by an inner REPEAT component.

In principal there is no limit of nesting.

---

## Performance Considerations

### In general

The REPEAT component binds to a list and renders the content of the list by repeating its content correspondingly. Of course this means that the component has a certain performance impact if it comes to lists with a lot of items:

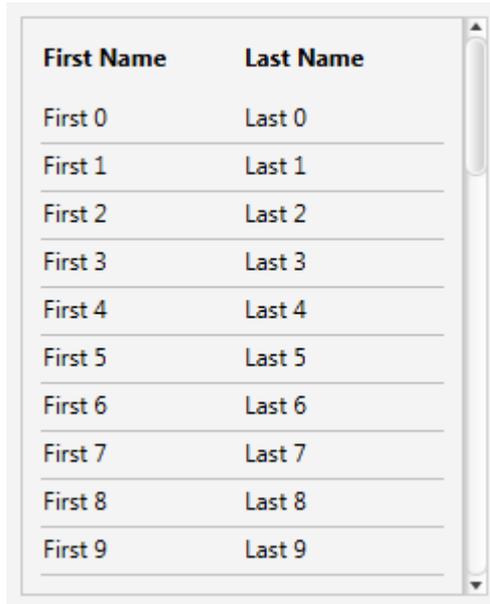
- The amount of data that is transferred from the server to the client increases.
- The effort for building up the user interface within the client increases.

As consequence you should take care to not send too big lists into the REPEAT component processing. It's difficult to exactly define some limit, but in general you should think about performance aspects if the list has more than 50 items.

## Server Side Scrolling

The solution what to do with long lists is that you need to split the whole list into partitions and allow the user to navigate between. - You need some indirection between the original, long list and the list that is presented to the user.

This is exactly what the class “ServerSideScrollList” does. Please take a look at the example that is part of the Demo Workplace:



First Name	Last Name
First 0	Last 0
First 1	Last 1
First 2	Last 2
First 3	Last 3
First 4	Last 4
First 5	Last 5
First 6	Last 6
First 7	Last 7
First 8	Last 8
First 9	Last 9

From Layout Definition point of view the REPEAT part looks “just normal”:

```
<t:pane id="g_3" border="#00000030" padding="10">
  <t:row id="g_4">
    <t:label id="g_5" font="weight:bold" text="First Name"
      width="100" />
    <t:label id="g_6" font="weight:bold" text="Last Name" width="100" />
  </t:row>
  <t:rowdistance id="g_7" height="10" />
  <t:repeat id="g_8"
    listbinding="#{d.DemoRepeatServerSideScrolling.items}">
    <t:row id="g_9">
      <t:label id="g_10" text="{firstName}" width="100" />
      <t:label id="g_11" text="{lastName}" width="100" />
    </t:row>
    <t:rowdistance id="g_12" height="3" />
    <t:rowline id="g_13" />
  </t:repeat>
</t:pane>
```

The REPEAT binds to some “items” list on the server side.

But looking into the code you will see that instead of using a plain List-implementation a special class “ServerSideScrollList” is used:

```
public class Item
{
  String m_firstName;
  String m_lastName;
  public String getFirstName() { return m_firstName; }
  public void setFirstName(String firstName) { m_firstName = firstName; }
```

```

        public String getLastName() { return m_lastName; }
        public void setLastName(String lastName) { m_lastName = lastName; }
    }

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

    serversideScrollList<Item> m_items = new ServersideScrollList<Item>();

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

    public DemoRepeatServersideScrolling(IWorkpageDispatcher dispatcher)
    {
        super(dispatcher);
        for (int i=0; i<50; i++)
        {
            Item it = new Item();
            it.setFirstName("First " + i);
            it.setLastName("Last " + i);
            m_items.getItems().add(it);
        }
    }

    public String getPageName() { return
"/workplace/demorepeatserversidescrolling.jsp"; }
    public String getRootExpressionUsedInPage() { return
"#{d.DemoRepeatServersideScrolling}"; }

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

    public ServersideScrollList<Item> getItems() { return m_items; }

```

The items are not directly accessed in “m\_items”, but are accessed by calling the method “getItems()”.

From the screenshot above you see that not 50 items are rendered within the client - but only 10. This is due to the default behavior of the ServerSideScrollList - separating all the items in partitions of 10 items.

The class ServerSideScrollList allows you to:

- pass an own list in the constructor
- define the number of items that are sent to the client side
- define the top index of the item that is shown
- navigate within all the items by providing corresponding methods

Please check the JavaDoc API documentation of this class.

In the example, next to the PANE holding the REPEAT processing there is the definition of a SCROLLBAR component:

```

<t:row id="g_2">
  <t:pane id="g_3" border="#00000030" padding="10">
    ...
    <t:repeat id="g_8"
      listbinding="#{d.DemoRepeatServersideScrolling.items}">
      ...
    </t:repeat>
    ...
  </t:pane>
  <t:scrollbar id="g_14" flush="true" height="-100"
    orientation="vertical"
    sbmaxvalue="#{d.DemoRepeatServersideScrolling.items.maxIndex}"
    sbminvalue="0"
    sbsize="#{d.DemoRepeatServersideScrolling.items.numberOfItems}"
    value="#{d.DemoRepeatServersideScrolling.items.topIndex}" />

```

```
</t:row>
```

This SCROLLBAR component binds to the properties that are provided by the ServerSideScrollList. When moving the scroll bar then the corresponding top index of the ServerSideScrollList instance is set - and the REPEAT content is scrolled correspondingly.

# Page Navigation

## Basics

CaptainCasa Enterprise Client provides the following possibilities of navigation:

- Nesting of Pages
- Popup Dialogs
  - modal
  - modeless

Of course all possibilities can be mixes with one another.

This chapter covers the basic aspects of page navigation and page modularization. Please also check the chapter “Page Bean Modularization” - which is a very useful concept built on top.

## JSF Navigation Concepts

You may already know something about JSF navigation concepts: these are based on so called “action” definitions, which then refer to definitions inside faces-config.xml. Being a quite nice way to define page sequences, the JSF navigation concepts are not able to manage complex (but typical!) scenarios of typical rich client user interfaces, including:

For these reasons we do not internally use and support the JSF navigation concepts.

## Enterprise Client Concepts

CaptainCasa Enterprise Client offers two ways of managing navigation:

- Nesting of pages: one page contains one or several other pages
- Dialogs: one page opens one or more modal or modeless dialogs

...what's about page sequences? Well, page sequences are just a special usage type for nesting pages - in which the outer page updates the inner page following certain application rules.

## Page Inclusion vs. Page Bean Inclusion

We need to take a look back into the chronology of Enterprise Client. ...Sorry!

The first possibility of including one page within another page was the “page inclusion” (component ROWINCLUDE). The outer page (e.g. “outer.jsp”) directly defines an area, in which an inner page (e.g. “inner.jsp”) is running.

```
+++++
+ outer.jsp                               +
+                                         +
+ ...#{d.outerUI.xxx}                     +
+                                         +
+                                         +
+ +++++                                   +
+ + inner.jsp                             +
+ +                                         +
+ + ...#{d.InnerUI.xxx}...               +
+ +++++                                   +
+++++
```

Well, this concept is very simple on the one hand - because the result is something like a (virtual) big XML-file - merged out of the outer and the inner page's xml. So, where are the difficulties?

The difficulties are that within the page definition there are expressions. E.g. the inner page contains an expression “#{d.InnerUI.firstName}”. The expressions are taken over when the inner page is nested by the outer page. So the virtual big page consists out of expressions addressing the outer page (e.g. #{d.OuterUI.user}”) and addressing the inner page (“#{d.InnerUI.firstName}”). On server side this means that two instances of beans or addressed through the “d”-Dispatcher, which are not linked in any way - other than sharing the same dispatcher instance.

Let's add one more level of complexity: now the outer page not only wants to show one inner page, but it wants to show the inner page twice - e.g. on the left and on the right side. Of course filled with different data!

```

+++++
+ outer.jsp
+
+ ...#{d.OuterUI.xxx}
+
+
+ ++++++ ++++++
+ + inner.jsp + inner.jsp
+ + + + +
+ + ...#{d.InnerUI.xxx}... + ...#{d.InnerUI.xxx}...
+ + ++++++ ++++++
+++++

```

And now, the static taking over of expressions while building the virtual big page does not work anymore. The left inner page and the right inner page are referencing the same expression - and this means they are referencing the same object instance on server side!

Consequence: you can by using “page inclusion” embed one page twice (or more), but the data shown in the page is always the same.

This is the reason why an alternative to page inclusion was added to the Enterprise Client: the “Page Bean Inclusion”.

When thinking in Page Beans then basically you think in UI objects - and not in jsp-layouts anymore. The UI objects are called “page beans”. Page beans are just normal server side objects, which are derived from the abstract class “PageBean”. When following the tutorial you already created page bean classes - so there is nothing special about.

Let's immediately check the complex example from above and show how it is built using page beans:

```

+++++
+ OuterUI instance
+ ...using: outer.jsp
+
+
+ OuterUI.getLeft() OuterUI.getRight()
+ ++++++ ++++++
+ + InnerUI instance + InnerUI instance
+ + ...using: inner.jsp + ...using: inner.jsp
+ + ...#{d.InnerUI.xxx}... + ...#{d.InnerUI.xxx}...
+ + ++++++ ++++++
+++++

```

Now there is an outer instance of “OuterUI” which internally holds two instances of “InnerUI”. The instances are available through the properties “left” and “right”. The pseudo code of the OuterUI class is:

```
public class OuterUI
{
    InnerUI m_left = new InnerUI();
    InnerUI m_right = new InnerUI();

    public InnerUI getLeft() { return m_left; }
    public InnerUI getRight() { return m_right; }
}
```

You already see:

- The class OuterUI directly holds and influences the contained instances “right” and “left”. It is responsible for providing the inner objects to be shown on the left and on the right side.
- And: basically you do not have to care about expressions at all anymore. The component that is used for including inner page beans into outer page beans (ROWPAGEBEANINCLUDE) is automatically handling all expression issues.

## Consequences

In the meantime we definitely recommend to only use the page bean way for any type of navigation. It's just much simpler to use and it's covering any complexity when it comes to nesting pages into one another.

Of course the page inclusion still is available, but: only use it if you have a real reason. The default strategy of your navigation implementation should always be page bean based.

This means:

- Use ROWPAGEBEANINCLUDE for nesting pages (and not ROWINCLUDE)
- Use the page bean's inherited methods (openModalPopup()...) for creating dialogs (and not static methods on ModalPopup)

---

## Page Bean Modularization - Example

### Page

This is the screenshot of the example:

Open Vacation Address

First Name

Last Name

**Business Address**

Street (1)

Street (2)

**Home Address**

Street (1)

Street (2)

There is an “outside” page containing two “inside” pages, both sharing the same jsp-page, but containing different instance data. When pressing the “Open Vacation Address” button then a third instance of the address is shown in a popup window.

## Java Code

Let's take a look onto the code of the outside page first:

```

package workplace;
...
public class DemoPageBeanUI
    extends workpageDispatchedPageBean
    implements Serializable
{
    // -----
    // members
    // -----

    protected String m_lastName;
    protected String m_firstName;

    DemoPageBeanAddressUI m_homeAddress = new DemoPageBeanAddressUI();
    DemoPageBeanAddressUI m_businessAddress = new DemoPageBeanAddressUI();
    DemoPageBeanAddressUI m_vacationAddress = new DemoPageBeanAddressUI();

    ModalPopup m_popup;

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

    public DemoPageBeanUI(IWorkpageDispatcher dispatcher)
    {
        super(dispatcher);
        initPageBeans();
    }

    public String getPageName() { return "/workplace/demopagebean.jsp"; }
    public String getRootExpressionUsedInPage() { return "#{d.DemoPageBeanUI}"; }

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

    public DemoPageBeanAddressUI getHomeAddress() { return m_homeAddress; }
    public DemoPageBeanAddressUI getBusinessAddress() { return m_businessAddress; }

```

```

}

public String getLastName() { return m_lastName; }
public void setLastName(String value) { m_lastName = value; }

public String getFirstName() { return m_firstName; }
public void setFirstName(String value) { m_firstName = value; }

public void onOpenVacationAddress(ActionEvent event)
{
    openModelessPopup(m_vacationAddress,"Vacation Address",0,0,new
ModelessPopup.IModelessPopupListener()
    {
        public void reactOnPopupClosedByUser()
        {
            closePopup(m_vacationAddress);
        }
    });
}

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

private void initPageBeans()
{
    m_homeAddress.setTitle("Home Address");
    m_homeAddress.setStreet1("Home Street 1");
    m_homeAddress.setStreet2("Home Street 2");
    m_businessAddress.setTitle("Business Address");
    m_businessAddress.setStreet1("Business Street 1");
    m_businessAddress.setStreet2("Business Street 2");
    m_vacationAddress.setTitle("Vacation Address");
    m_vacationAddress.setStreet1("Vacation Street 1");
    m_vacationAddress.setStreet2("Vacation Street 2");
}
}
}

```

Each contained page is represented by a corresponding page instance (m\_homeAddress, m\_businessAddress and m\_vacationAddress), with corresponding get-methods for accessing the instances. The sub-pages are members / properties of the outside page.

In addition you see that there are two special methods “getPageName()” and “getRootExpressionUsedInPage()” that are implemented.

The code of the contained instances is:

```

package workplace;

...

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

    protected String m_title = "<Title>";
    protected String m_street1 = "<Street1>";
    protected String m_street2 = "<Street2>";

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

    public String getPageName() { return "/workplace/demopagebeanaddress.jsp"; }
    public String getRootExpressionUsedInPage() { return
"#{d.DemoPageBeanAddressUI}"; }

    public String getStreet1() { return m_street1; }
    public void setStreet1(String value) { m_street1 = value; }

    public String getStreet2() { return m_street2; }
}

```

```

public void setStreet2(String value) { m_street2 = value; }

public String getTitle() { return m_title; }
public void setTitle(String value) { m_title = value; }

}

```

Again the code contains an implementation of the “getPageName()” and “getRootExpressionUsedInPage()”.

## JSP Page

Now, let's take a look onto the outside jsp page definition:

```

/workplace/demopagebean.jsp:
<t:rowheader id="g_1">
  <t:button id="g_2"
    actionListener="#{d.DemoPageBeanUI.onOpenVacationAddress}"
    text="Open Vacation Address" />
</t:rowheader>
<t:rowbodypane id="g_3" rowdistance="5">
  <t:row id="g_4">
    <t:label id="g_5" text="First Name" width="100" />
    <t:field id="g_6" text="#{d.DemoPageBeanUI.firstName}" width="100" />
  </t:row>
  <t:row id="g_7">
    <t:label id="g_8" text="Last Name" width="100" />
    <t:field id="g_9" text="#{d.DemoPageBeanUI.lastName}" width="100" />
  </t:row>
  <t:rowdistance id="g_10" height="20" />
  <t:rowpagebeaninclude id="g_11"
    pagebeanbinding="#{d.DemoPageBeanUI.businessAddress}" />
  <t:rowpagebeaninclude id="g_12"
    pagebeanbinding="#{d.DemoPageBeanUI.homeAddress}" />
</t:rowbodypane>
<t:rowstatusbar id="g_13" />

```

The integration of the two address pages into the outside page is not done by using the ROWINCLUDE - but by using the ROWPAGEBEANINCLUDE component.

The jsp page definition of the address page is a “very normal” one:

```

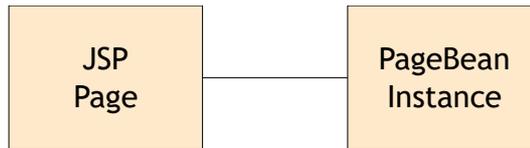
/workplace/demopagebeanaddress.jsp:
<t:row id="g_1">
  <t:pane id="g_2" border="#00000030" padding="0" rowdistance="0">
    <t:rowtitlebar id="g_3" text="#{d.DemoPageBeanAddressUI.title}" />
  </t:pane>
  <t:row id="g_4">
    <t:pane id="g_5" padding="10" rowdistance="5">
      <t:row id="g_6">
        <t:label id="g_7" text="Street (1)" width="100" />
        <t:field id="g_8" text="#{d.DemoPageBeanAddressUI.street1}"
          width="100" />
      </t:row>
      <t:row id="g_9">
        <t:label id="g_10" text="Street (2)" width="100" />
        <t:field id="g_11" text="#{d.DemoPageBeanAddressUI.street2}"
          width="100" />
      </t:row>
    </t:pane>
  </t:row>
</t:row>

```

---

## Page Bean Details

The concepts are based on the paradigm that for one page there is one bean serving the page:



Example: the “demopagebeanaddress.jsp” is related to the “DemoPageBeanAddressUI” class - all its content is managed by the class.

### IPageBean / PageBean Instances

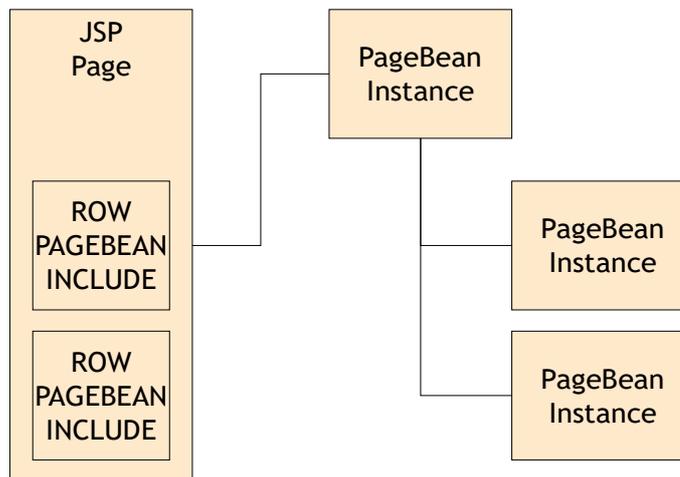
A bean covering all the aspects of one page is called “PageBean” - and implements the interface “IPageBean”, typically by deriving from the “PageBean” class.

A page bean has to implement an interface that allows the modularization framework to integrate it into other JSP pages - executed by the ROWPAGEBEANINCLUDE component. The two methods that need to be provided are:

- `getPageName()` - this method needs to return the page name of the page that is to be used for the bean. Typically this is one page, but it's also possible to return a page name that fits best to the application's situation - if you have multiple JSP-pages that use the same page bean.
- `getRootExpressionUsedInPage()` - this method needs to tell the expression that is used inside the .jsp definition for accessing the page bean object. This sounds complex but it isn't: in the example above, within the page “demopagebeanaddress.jsp” the page bean is addressed by the expression “#{d.DemoPageBeanAddressUI}” - so that's the root expression that is used within the page definition!

### The ROWPAGEBEANINCLUDE Component

This is the component to embed a page bean into a page.



Embedding covers two aspects:

- 1. - The jsp page of the page bean is included.
- 2. - The expressions in the page that is included are updated so that they fit to the current scenario and so that the page's content is really pointing to the correct page bean instance.

Result: there is no additional effort for integrating page bean instances within a page. The page beans instances are normal members / properties of other page beans

instances. While the normal ROWINCLUDE component just optically includes a page into another pages, the ROWPAGEBEANINCLUDE component

## What happens internally...

Inside the management of the ROWPAGEBEANINCLUDE component there is an automated update of the expressions of an included page.

Let's assume the following scenario:

- The outside page is accessed through expression: “#{d.d\_1.DemoPageBeanUI}”.
- It includes an inside page bean (as in the example above) that is made available through the property “businessAddress”. Consequence: the expression for accessing the inside page's root object is “#{d.d\_1.DemoPageBeanUI.businessAddress}”.
- The inside page is defined within a JSP page that point to its root object via “#{d.DemoPageBeanAddressUI}”. Result: the expression replacement is defined, so that all references within the inside page are updated from “#{d.DemoPageBeanAddressUI.\*}” to “#{d.d:1.DemoPageBeanUI.businessAddress.\*}”.

---

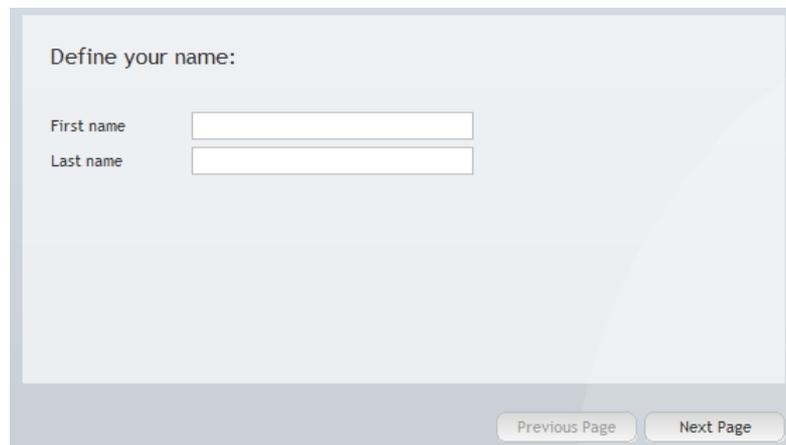
## Page Bean Navigation

### Page Navigation

...this is the typical question: “How do I switch from one page bean to the next?”.

The answer is: pages and page beans are independent objects and can not themselves switch to the next page. As consequence, navigation needs to be done through an outside page, that holds an inside page - and that exchanged the inside page.

This may sound complex, but indeed is not. Let's take a look onto the following example:



The screenshot shows a web form with the following elements:

- Title: Define your name:
- Input field: First name
- Input field: Last name
- Buttons: Previous Page and Next Page

When pressing the “Next Page” button then the screen changes to:

The outside page is defined in the following way:

```

...
...
<t:row id="g_2">
  <t:pane id="g_3" height="100%" width="100%">
    <t:rowpagebeaninclude id="g_4"
      pagebeanbinding="#{d.DemoPageBeanNavOutside.currentStep}" />
    </t:pane>
  </t:row>
<t:row id="g_5">
  <t:coldistance id="g_6" width="100%" />
  <t:button id="g_7"
    actionListener="#{d.DemoPageBeanNavOutside.onPrevious}"
    enabled="#{d.DemoPageBeanNavOutside.previousEnabled}"
    text="Previous Page" width="100%" />
  <t:coldistance id="g_8" width="5%" />
  <t:button id="g_9"
    actionListener="#{d.DemoPageBeanNavOutside.onNext}"
    enabled="#{d.DemoPageBeanNavOutside.nextEnabled}"
    text="Next Page" width="100%" />
</t:row>
...

```

There is a ROWPAGEBEANINCLUDE definition, point via its expression to the “currentStep” property of the Java program:

```

public class DemoPageBeanNavOutside
  extends DemoBase
  implements Serializable
{
  int m_currentStepIndex = 0;
  List<IPageBean> m_steps = new ArrayList<IPageBean>();

  public DemoPageBeanNavOutside(IWorkpageDispatcher workpageDispatcher)
  {
    super(workpageDispatcher);
    m_steps.add(new DemoPageBeanNavStep1());
    m_steps.add(new DemoPageBeanNavStep2());
  }

  public String getPageName() { return "/workplace/demopagebeannavoutside.jsp"; }
  public String getRootExpressionUsedInPage() { return
    "#{d.DemoPageBeanNavOutside}"; }

  public IPageBean getCurrentStep() { return m_steps.get(m_currentStepIndex); }
  public boolean isPreviousEnabled() { return m_currentStepIndex > 0 ? true:
  false; }
  public boolean isNextEnabled() { return m_currentStepIndex < (m_steps.size()-
  1) ? true: false; }

  public void onPrevious(ActionEvent event)
  {
    if (m_currentStepIndex > 0)

```

```

        m_currentStepIndex--;
    }

    public void onNext(ActionEvent event)
    {
        if (m_currentStepIndex < (m_steps.size()-1))
            m_currentStepIndex++;
    }
}

```

The “currentStep” property passes back a page bean instance, that is selected dependent from an index, that is updated when the user presses the “Next” or the “Previous” button.

The contained page beans are straight and simple page bean implementations, e.g. the first step’s bean and layout is:

```

<t:rowbodypane id="g_1" rowdistance="5">
  <t:row id="g_2">
    <t:label id="g_3" font="size:16" text="Define your name:" />
  </t:row>
  <t:rowdistance id="g_4" height="20" />
  <t:row id="g_5">
    <t:label id="g_6" text="First name" width="100" />
    <t:field id="g_7" text="#{d.DemoPageBeanNavStep1.firstName}"
      width="200" />
  </t:row>
  <t:row id="g_8">
    <t:label id="g_9" text="Last name" width="100" />
    <t:field id="g_10" text="#{d.DemoPageBeanNavStep1.lastName}"
      width="200" />
  </t:row>
</t:rowbodypane>

public class DemoPageBeanNavStep1
  extends PageBean
  implements Serializable
{
  protected String m_firstName;
  protected String m_lastName;

  public DemoPageBeanNavStep1()
  {
  }

  public String getPageName() { return "/workplace/demopagebeannavstep1.jsp"; }
  public String getRootExpressionUsedInPage() { return
    "#{d.DemoPageBeanNavStep1}"; }

  public String getFirstName() { return m_firstName; }
  public void setFirstName(String value) { m_firstName = value; }

  public String getLastName() { return m_lastName; }
  public void setLastName(String value) { m_lastName = value; }
}

```

In the example the navigation is done through the outside page - by pressing corresponding buttons. Of course the navigation could also be triggered from the inside page - in this case you can just set up any interface relation (in means of simple Java-API) between the outside page and the inside page(s).

## Popup Management

Each page bean instance inherits the following methods for opening and closing popup windows:

- openModalPopup(IPageBean pageBean, ...)

- openModelessPopup(IPageBean pageBean, ...)
- closePopup(IPageBean pageBean)

The open-methods return back an instance of ModalPopup / ModelessPopup.

The following example shows a page bean, opening up an other page bean:



... opening:

The page bean that is called as popup is exactly the same one as used in the previous example, explaining navigation concepts. So, let's focus onto the page that opens up the popup.

The layout and code are:

```
<t:row id="g_2">
  <t:button id="g_3"
    actionListener="#{d.DemoPageBeanPopupCaller.onOpenPageBeanInPopup}"
    text="Open Page Bean in Popup" />
</t:row>

public class DemoPageBeanPopupCaller
  extends workpageDispatchedPageBean
  implements Serializable
{
  public DemoPageBeanPopupCaller(IWorkpageDispatcher workpageDispatcher)
  {
    super(workpageDispatcher);
  }

  public String getPageName() { return
"/workplace/demopagebeanpopupcaller.jsp"; }
  public String getRootExpressionUsedInPage() { return
"#{d.DemoPageBeanPopupCaller}"; }

  public void onOpenPageBeanInPopup(ActionEvent event)
  {
    // create page to be opened in popup
    final DemoPageBeanAddress address = new DemoPageBeanAddress();
    address.setTitle("nice title");
    address.setStreet1("nice street 1");
    address.setStreet2("nice street 2");
    // open popup (width & height are set to 0, so that popup is sized
    // by its content
    final ModalPopup mp = openModalPopup(address,"Title of popup",0,0,null);
    mp.setPopupListener(new ModalPopup.IModalPopupListener()
    {
      // This method is called when the user explicitly
      // closes the popup by alt-F4 or by clicking
      // onto the right-top close-icon
      public void reactOnPopupClosedByUser()
      {
        closePopup(address);
        Statusbar.outputMessage("Modal popup was closed!");
      }
    });
  }
}
```



an optimized rendering management is processed:

- UPDATEONINNEREVENTONLY: if you set this to “true” then the corresponding area will not be processed if an event is thrown outside the area. E.g. if the user navigates within the function tree on the left, then the content area will not be processed, assuming that there was no change.
- UPDATEISOLATION: if you set this to “true” then events within the corresponding area will not cause a processing of the whole screen - but only this isolated area will be rendered/processed.

So the one attribute (UPDATEONINNEREVENTONLY) protects the area from being rendered on outside events - the other attribute (UPDATEISOLATION) protects outside areas from being rendered on inner events.

You may set both attributes to “true” in parallel - and then fully encapsulate your area.

But... - pay attention: By defining one of the attributes you step back from the principle that the whole screen always is automatically updated during a request-response processing!

By calling function “ThreadData.getInstance().registerChangeUpdatingAllAreas()” you can on server side override all optimization and isolation. You need to do this call if you know that a certain updated within your isolated area will also affect data that is rendered in other parts of the screen.

Please note: the two attributes are not really required in typical scenarios. They should be applied in “bigger” scenarios only - and should not be used “just because they exist”...

---

## Preconfigured Popups

The layout and processing of a popup is completely up to you - any page can be called as popup.

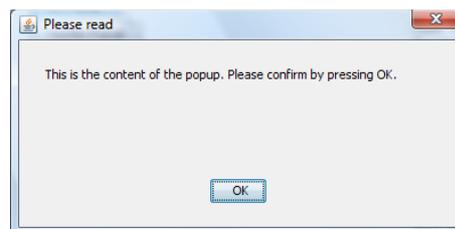
CaptainCasa Enterprise Client comes with some default popups in order to support the most common popup scenarios:

- OK-Popup
- Yes/No-Popup
- Value Selection Popup

The opening and closing of the popups is wrapped by a corresponding Java interface.

### OK-Popup

The “OK-Popup” allows to output a text message to the user. The user can confirm the message by clicking an OK button:



The Java code to open the popup on server side is:

```
public void onOpenOKPopup(ActionEvent ae)
{
```

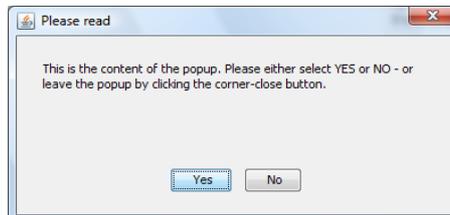
```

OKPopup.createInstance
("Please read",
 "This is the content of the popup.
 Please confirm by pressing OK.");
}

```

## Yes/No Popup

The Yes/No popup is similar to the OK popup - but allows the user to decide “Yes” or “No”.



The popup is called by passing an implementation of interface “IYesNoListener”:

```

public void onOpenYESNOPopup(ActionEvent ae)
{
    YESNOPopup.createInstance
    (
        "Please read",
        "This is the content ... ",
        new IYesNoListener()
        {
            public void reactOnNo()
            {
                StatusBar.outputMessage("No");
            }
            public void reactOnYes()
            {
                StatusBar.outputMessage("Yes");
            }
        }
    );
}

```

## Value Selection Popup with COMBOFIELD

There is a special field component that is designed to be linked with value help popups: the COMBOFIELD component.

### Overview

The component supports all properties of the normal FIELD component. It is rendered in a similar way to a combo box - so that the user immediately sees that value help is provided for this field:



There are three ways to activate value help within the component:

- The user clicks onto the arrow-icon.
- The user presses “F4” inside the component.
- The user presses “ctrl-Space” inside the component.

In all cases a value help request is sent to the action listener of the component. The event that is passed into the server side action listener method is an event of data type “BaseActionEventValueHelp”.

It's now the server side processing that decides how to support the user. The typical option is to open up a popup window and present the user a list of valid values, or to present a popup allowing the user to search for values.

When opening a popup then there are two types of popups that make sense:

- You may open a modal popup - in this case the user must explicitly close the popup before continuing with the field input.
- You may open a modeless popup that is closed automatically if the user clicks outside.

## Preconfigured Value Help Popups

For presenting values within a list of valid values there is a preconfigured management including preconfigured popups. Please check the demo workplace for examples and coding details.

## Type Ahead Management

A special type of usage is the "type ahead" usage. In this case the user input some text into the field. After the field recognizing that no further input was done for a certain while a value help request is sent to the server. The server side logic now provides a list of reasonable values, the list being already filtered against the user's current input.

The type ahead management is based on the FLUSH, FLUSHTIMER and KEEPFOCUS attribute of the COMBOFIELD component.

- By setting FLUSH to "true" the component will send data changes to the server. Normally these data changes are sent at this point of time when the user steps out of the field into another component.
- By in addition setting the FLUSHTIMER to a certain millisecond-value the FLUSH will be executed when the user stopped with typing in new data.
- By setting the KEEPFOCUS to true the keyboard focus will be kept within the field and will not move into the popup that is opened.

A flush event is processed within the action listener. The event is of type "BaseActionEventFlush". The event provides a method by that you can determine if the flush was caused by a timer: "getFlushWasTriggeredByTimer".

Result: in your implementation you get notified in two ways when the user wants to get some value help:

- You receive a normal value-help-event because the user did explicitly press "F4" or "ctrl-space".
- Your receive a flush-event because the user stopped with data input. In this case you need to double check that the flush event was triggered by the timer, and not by stepping out of the component.

Please check the demo workplace for coding details.

# Working with Menus

CaptainCasa Enterprise Client supports two different types of menus:

- menu bar: typically located in the header area of a screen
- popup menus: opening up when pressing with the right mouse button on a certain control

Both types of menus work with the same type of components: MENU and MENUITEM, but used in a slightly different way.

---

## Example Reference

Please check the “demopopupmenu.jsp” example for details on JSP layout definition and code.

---

## MENUBAR

The menubar is quite simple: it may contain MENU components, and the MENU components themselves contain MENUITEM components. MENUITEM components are the ones which the user selects in order to invoke a certain server side function.

You can nest multiple MENU items below one MENUBAR so that the menu itself has several hierarchical layers.

Each MENU component has a text (obligatory) and an image (optional).

Each MENUITEM in addition has an ACTIONLISTENER attribute - which is the one called when the user invokes the MENUITEM.

The menu bar can either be statically built inside the layout, i.e. you arrange all the subcomponents as part of the layout definition. Or you dynamically add components using the COMPONENTBINDING attribute.

---

## POPUPMENU

### Usage of POPUPMENU

The principles of POPUPMENU management are simple as well:

- POPUPMENU definitions include the POPUPMENU instance itself and a list of MENUITEM instances that form the popup menu. Each POPUPMENU needs to get assigned a unique id. - This something “unusual”: normally you do not care about the id of components, they are generated by the Layout Editor automatically. Now, with POPUPMENU instances, you do - pay attention to define the ID values properly!
  - Within the MENUITEM you can define an ACTIONLISTENER that is executed (event type: BaseActionEventSelect) when the user selects a menu item.
  - And there is a COMMAND attribute in which you can specify a string value that identifies the MENUITEM instance. Background: the ACTIONLISTENER of the component, which references the POPUPMENU is called as well (event type: BaseActionEventPopupMenuItem). Part of the event information is the command string.
- From components that use the POPUPMENU instance, it is referenced by specifying the ID within the attribute POPUPMENU of the component.

## Example:

```
<t:rowdemobodypane id="g_3" actionListener="#{d.demoPopupMenu.onBodyAction}"
objectbinding="demoPopupMenu" popupmenu="POPUPMENU3" >
  <t:row id="g_4" >
    <t:foldablepane id="g_5"
actionListener="#{d.demoPopupMenu.onFoldablePaneAction}" popupmenu="POPUPMENU4"
text="Foldable Pane" width="100%" >
      <t:row id="g_6" >
        <t:label id="g_7" actionListener="#{d.demoPopupMenu.onLabel1Action}"
popupmenu="POPUPMENU1" text="Label 1" width="120" />
        <t:field id="g_8" actionListener="#{d.demoPopupMenu.onField1Action}"
popupmenu="POPUPMENU1" width="200" />
      </t:row>
      <t:rowdistance id="g_9" />
      <t:row id="g_10" >
        <t:label id="g_11" actionListener="#{d.demoPopupMenu.onLabel2Action}"
popupmenu="POPUPMENU2" text="Label 2" width="120" />
        <t:field id="g_12" actionListener="#{d.demoPopupMenu.onField2Action}"
popupmenu="POPUPMENU2" width="200" />
      </t:row>
    </t:foldablepane>
  </t:row>
  <t:rowdistance id="g_13" height="20" />
  <t:row id="g_14" actionListener="#{d.demoPopupMenu.onRowAction}"
popupmenu="POPUPMENU3" >
    <t:textpane id="g_15" text="Some content in this row. The popup menu is
defined on row level." width="100%" />
  </t:row>
</t:rowdemobodypane>

<t:popupmenu id="POPUPMENU1" >
  <t:menuitem id="g_16" command="OPTION1" text="Option 1" />
  <t:menuitem id="g_17" command="OPTION2" text="Option 2" />
  <t:menuitem id="g_18" command="OPTION3" text="Option 3" />
</t:popupmenu>

<t:popupmenu id="POPUPMENU2" >
  <t:menuitem id="g_19" command="OPTION4" text="Option 4" />
  <t:menuitem id="g_20" command="OPTION5" text="Option 5" />
  <t:menuitem id="g_21" command="OPTION6" text="Option 6" />
</t:popupmenu>

<t:popupmenu id="POPUPMENU3" >
  <t:menuitem id="g_22" command="OPTION7" text="Option 7" />
  <t:menuitem id="g_23" command="OPTION8" text="Option 8" />
  <t:menuitem id="g_24" command="OPTION9" text="Option 9" />
</t:popupmenu>

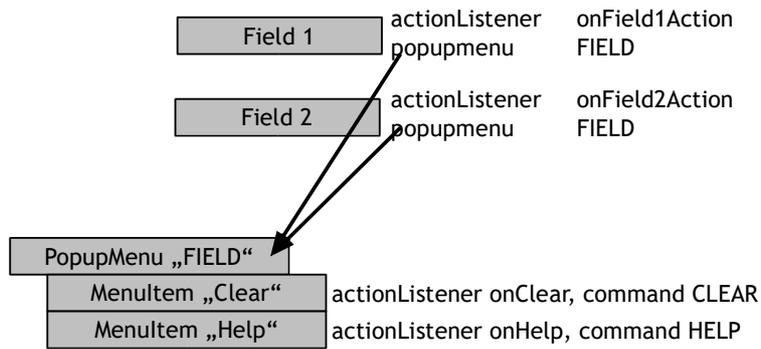
<t:popupmenu id="POPUPMENU4" >
  <t:menuitem id="g_25" command="OPTION10" text="Option 10" />
  <t:menuitem id="g_26" command="OPTION11" text="Option 11" />
  <t:menuitem id="g_27" command="OPTION12" text="Option 12" />
</t:popupmenu>
```

In the example you see 4 popup menus, that are referenced from diverse components. You can reference the popup menus from various levels of components. In the example they are referenced from “small components” like LABEL and FIELD, as wells as from “big componetns” such as the ROWBODYPANE.

## Event Reaction

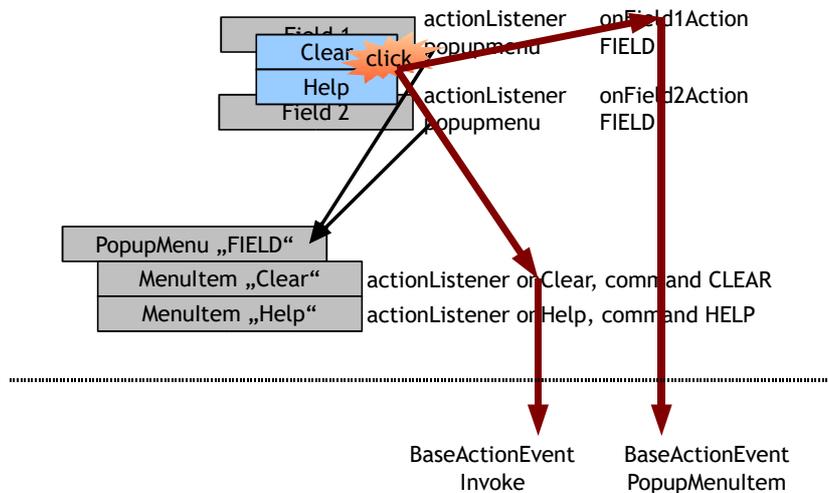
The event reaction of the user selecting a popup menu item either is done in the action listener of the MENUITEM or it is done in the action listener of the component referencing the POPUPMENU.

Have a look onto the following page definition:



There are two fields defined, each field being assigned to the same popup menu “FIELD”, that provides two menu items.

The user now can open the same popup menu on both fields. When selecting a menu item on one of the fields the following happens:



On server side two actionListeners are triggered that allow to react on the event:

- “MENUITEM level”: the actionListener that is defined with the menu item is called
- “Component level”: the actionListener that is defined with the FIELD component is called

You may implement and assign both actionListeners or you may implement one of the actionListeners.

### Event Reaction on MENUITEM Level

The first option is “just normal”:

```
<t:popupmenu id="FIELD" >
  <t:menuitem id="g_25" text="clear" actionListener="#{xxx.onClear}"
  command="CLEAR"/>
  <t:menuitem id="g_26" text="Help" actionListener="#{xxx.onHelp}"
  command="HELP"/>
</t:popupmenu>
```

The action listener “#{xxx.onClear}” is called, the event that is passed is of type “BaseActionEventInvoke”.

In this scenario you do not get any information about the component on which the popup menu was opened. Remember: there may be several components referencing the same POPUPMENU instance.

Use this type of event reaction if the reaction of the user selecting the menu item is

always the same for all referencing elements and if the reaction does not depend on the component on which the menu item was selected.

## Event Reaction on Component Level

The action listener of the component that references the POPUPMENU is called as well. The event type is “BaseActionEventPopupMenuItem”. Part of the information that is available with the event is the COMMAND string of the MENUITEM instance that was selected by the user.

As consequence you can have multiple components using the same POPUPMENU instance but having a different processing of the menu selection.

On server side you typically have code that looks as follows:

```
...
...
public void onField1Action(ActionEvent event)
{
    if (event instanceof BaseActionEventPopupMenuItem)
    {
        BaseActionEventPopupMenuItem e = (BaseActionEventPopupMenuItem)event;
        String command = e.getCommand();
        if (“CLEAR”.equals(command))
        {
            ...
        }
        else if (“HELP”.equals(command))
        {
            ...
        }
    }
}
```

## Finding the right POPUPMENU...

Inside a page definition you reference POPUPMENU components by using the attribute POPUPMENU within a component (e.g. a field or a pane). You now can define different popup menus for different areas of the page - e.g. you may want to have a general popup menu that is valid for the whole page (e.g. defined on ROWBODYPANE level) and you may want to have a special popup menu for a certain area (e.g. defined on PANE level).

The client will always select the popup menu, that is adequate. Starting with the component in which the right mouse button was clicked, it steps up the component hierarchy until it finds a POPUPMENU attribute definition. The first definition it finds is the one which is used for building the popup menu.

---

## Hotkey Definitions

The POPUPMENU component offers a second very important feature. It allows to bind a menu item to a key. Pressing the key has the same meaning than opening the popup menu and selecting the menu item with the mouse button.

It may first sound strange that hotkey definitions are made as part of the popup menu definition, but makes sense... for two reasons:

- Every time you define hotkeys for a screen, you need to make sure that these hotkeys are reachable without keyboard as well. Now, this is exactly true with popup menu definitions: commands are reachable by right mouse button, and - optionally - by keyboard.
- The assignment of POPUPMENU instances to components rules that one POPUPMENU instance is only valid within a certain area of the screen. E.g. a POPUPMENU defined for a certain PANE is only valid for the PANE and its included components. - This is the

same for the hot keys as well: they are only valid when the focus is within the area referencing the POPUPMENU. As a result, an “Enter-Hotkey” may have a different reaction if the user is in the one field than if the user is in an other field.

The hotkey definition is done by assigning the so called event keycode to the MENUITEM component's attribute HOTKEY. Open the value help within the layout editor in order to get a list of common keycodes. - In front of the keycode you can write the prefixes “ctrl-”, “alt-”, “shift-” in order to combine the keycode with one (or more) of these special keys.

### Addendum - Hotkey in Buttons, Icons, etc.

You define HOTKEY definitions as mentioned if you do not have any “invoker component” that you can attach the hot key to.

You can also directly define hotkey definitions in the HOTKEY attribute of BUTTON, ICON, LINK and other components.

---

## Dynamic Menu Definitions

The MENU/POPUPMENU definitions, that you saw in this chapter so far, are all statically defined popup menus: the popup menu is defined as part of the layout definition.

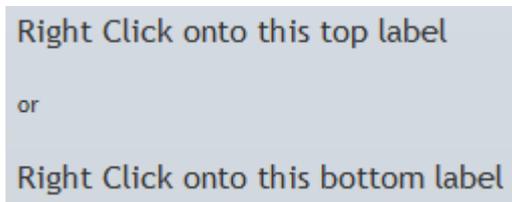
Of course you can build dynamic popup menus by using the just normal DYNAMICCONTENT component at any position within the MENU/POPUPMENU definition.

### Dynamic POPUPMENU - attribute POPUPMENULOADROUNTRIP

There are some cases in which you want to build the POPUPMENU at this point of time when the user presses with the right mouse button into a certain component.

There is the attribute POPUPMENULOADROUNTRIP which you can define within the component that binds to the POPUPMENU definition. When setting the attribute to “true”, then a roundtrip to the server is triggered when the user presses the right mouse button. The ActionListener of the component is called, receiving the event type “BaseActionEventPopupMenuLoad”. Together with the usage of a DYNAMICCONTENT you can now build up the popup menu.

Example: the following screen consists of two “big” labels, both of them with right mouse button support:



The layout definition is:

```
<t:row id="g_2">
  <t:label id="g_3"
    actionListener="#{d.DemoPopupMenuLoadRoundtrip.onLabelTopAction}"
    font="size:16" popupmenu="CENTRALPOPUP"
    popupmenuloadroundtrip="true"
    text="Right Click onto this top label" />
</t:row>
<t:row id="g_4">
  <t:label id="g_5" text="or" />
</t:row>
```

```

<t:row id="g_6">
  <t:label id="g_7"
    actionListener="#{d.DemoPopupMenuLoadRoundtrip.onLabelBottomAction}"
    font="size:16" popupmenu="CENTRALPOPOP"
    popupmenuloadroundtrip="true"
    text="Right click onto this bottom label" />
</t:row>
<t:popupmenu id="CENTRALPOPOP">
  <t:dynamiccontent id="g_8"
    contentbinding="#{d.DemoPopupMenuLoadRoundtrip.dynMenuContent}" />
</t:popupmenu>

```

In the layout there is only one popup menu definition “CENTRALPOPOP”, internally only holding a DYNAMICCONTENT component. The popup menu is references from both “big” of the big labels.

Because with both labels the attribute POPUPMENULOADROUNDTRIP is defined as “true” the corresponding ActionListener of each label is called, which the builds up the dynamic menu content:

```

public void onLabelTopAction(ActionEvent event)
{
  if (event instanceof BaseActionEventPopupMenuLoad)
  {
    List<ComponentNode> nodes = new ArrayList<ComponentNode>();
    {
      ComponentNode node = new ComponentNode("t:menuitem");
      node.addAttribute("text","First of top");
      node.addAttribute("command","TOP1");
      nodes.add(node);
    }
    {
      ComponentNode node = new ComponentNode("t:menuitem");
      node.addAttribute("text","Second of top");
      node.addAttribute("command","TOP2");
      nodes.add(node);
    }
    m_dynMenuContent.setContentNodes(nodes);
  }
  else if (event instanceof BaseActionEventPopupMenuItem)
  {
    BaseActionEventPopupMenuItem e = (BaseActionEventPopupMenuItem)event;
    StatusBar.outputSuccess("Command of popup menu item: " + e.getCommand());
  }
}

public void onLabelBottomAction(ActionEvent event)
{
  if (event instanceof BaseActionEventPopupMenuLoad)
  {
    List<ComponentNode> nodes = new ArrayList<ComponentNode>();
    {
      ComponentNode node = new ComponentNode("t:menuitem");
      node.addAttribute("text","First of bottom");
      node.addAttribute("command","BOT1");
      nodes.add(node);
    }
    {
      ComponentNode node = new ComponentNode("t:menuitem");
      node.addAttribute("text","Second of bottom");
      node.addAttribute("command","BOT2");
      nodes.add(node);
    }
    {
      ComponentNode node = new ComponentNode("t:menuitem");
      node.addAttribute("text","Third of bottom");
      node.addAttribute("command","BOT3");
      nodes.add(node);
    }
    m_dynMenuContent.setContentNodes(nodes);
  }
  else if (event instanceof BaseActionEventPopupMenuItem)
  {
    BaseActionEventPopupMenuItem e = (BaseActionEventPopupMenuItem)event;
    StatusBar.outputSuccess("Command of popup menu item: " + e.getCommand());
  }
}

```

```
}  
}
```

# Working with Drag & Drop

Implementing Drag&Drop is easy! ;-)

Drag&Drop is supported with nearly all components. Currently “internal drag & drop is supported” i.e. You can drag & drop data from one component to the other - you currently cannot drag & drop information from outside (e.g. from Microsoft Excel).

Drag & drop covers two aspects:

- Each component can define which types of drag information it supports.
- Each component can define what information can be dragged from it. The definition contains both the definition of a drag-type and the definition of a drag-content.
- When dragging and dropping the cursor changes automatically from “can drop” into “cannot drop” corresponding to if the drag information of the drag sender fits to the drop type definition of the drop receiver.
- Dependent from if the user presses the ctrl-key on the keyboard either a “drop” operation or a “copy” operation is executed.

---

## Example Reference

Please check details on JSP layout and code within the example “demodragdrop.jsp”.

---

## Details

### Attribute DRAGSEND

Each component that supports drag&drop supports the attribute DRAGSEND. This attribute contains a string that represents the content of the component.

The string is built in the following way: <dragtype>:<draginfo>, e.g. a string might be “article:4711”.

The DRAGSEND information is carried from one component to the next during a drag & drop operation.

### Attribute DROPRECEIVE

Each component that supports drag & drop supports the attribute DROPRECEIVE. This attribute contains a semicolon separated list of all drag-types that can be dropped onto the component.

E.g. a component defines DROPRECEIVE to be “article;customer;file”.

By defining the DROPRECEIVE attribute a component specifies which types of drag data can be dropped onto the component.

### Event Processing in the Server Side Java Code

At the point of time when the user drops information onto a component, the component that receives the drop event calls its ACTIONLISTENER.

In the action listener there are two event types that represent the drag & drop operation:

- “BaseActionEventDrop” - this is the event that is passed when the user does a normal drag & drop

- “BaseActionEventDropCopy” - this is the event that is passed when the user holds down the ctrl-key while performing the drag & drop

The “BaseActionEventDropCopy” inherits from “BaseActionEventDrop”. The main method both events provides is the `getDragInfo()` method. This returns the information behind the DRAGSEND attribute of the component where the drag & drop was started from.

## Information associated with Drop Event

Please take a look into the Java documentation of the drop event (class `BaseActionEventDrop`). There is quite important information associated with the event:

- of course the information that comes with the drag&drop operation (i.e. the string “<dragtype>:<draginfo>”), so that you know what type of information was dropped
- the x,y position of the drop operation - here you can use multiple ways for receiving the information about the drop position
  - ...as pixel value
  - ...as percentage value

## Controlling the Component Highlighting during Drag & Drop

By default, when the user drags a certain information from one component to others, the user interface behaved in the following way:

- A component, that is able to process the drag & drop (i.e. DRAGSEND information of the sender matches DROPRECEIVE information of the receiver) will be highlighted when the user moves with the mouse over the component.
- The highlighting by default is some yellow shading that covers the whole component.

You can choose other types of highlighting as well, by building up the DROPRECEIVE information in the following way: “<dragtype>:<dropZoneInfo>”. The following “drop zone info” definitions are available:

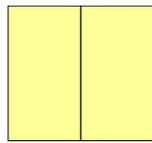
- “horizontalsplit”
- “verticalsplit”
- “edges”
- “sides”
- “sidesandcenter”
- “grid(<raster>)” e.g. “grid(25)”

Example: instead of defining DROPRECEIVE as “article” you can define DROPRECEIVE as “article:horizontalsplit”.

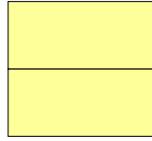
As result of explicitly assigning a drop zone information, the highlighting when moving the mouse over the component will be updated according to the definition, e.g. in case of using “horizontalsplit” only the left or the right area of the component will be highlighted during a drag & drop operation.

Please note: the definition of a “<dropZoneInfo>” is a pure optical advice, that the client uses for highlighting purposes. In case of dropping information you still need to figure out the drop zone on your own, by accessing the drop position that is associated with the drop event.

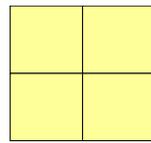
The following graphics illustrates the “sensitive” areas when using diverse dropzones:



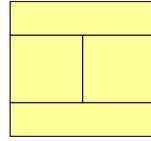
horizontalsplit



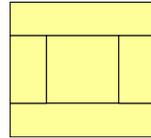
verticalsplit



edges



sides

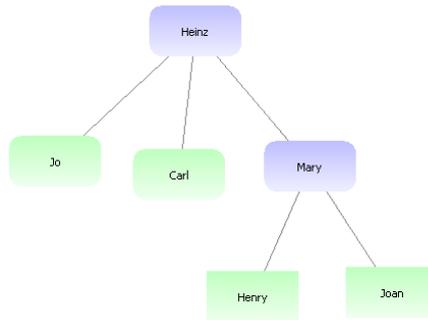


sidesandcenter

The lines in the graphics are at the position 0, 25%, 50%, 75%.

# Working with Shapes

CaptainCasa Enterprise Client provides a simple but efficient way to create shape graphics:



The features that are provided are:

- Flexible definition of shapes
  - Diverse painting commands
  - Shapes can be selected (mouse click / double click)
  - Shapes can be associated with a popup menu (right mouse click)
  - Shapes can be dragged and dropped
  - Shapes can consist out of complete panes, that themselves may include just normal components, such as FIELD and BUTTON components
- Definition of lines between shapes
  - Lines may have multiple begin / end arrow styles
  - Lines can be selected (mouse click / double click)
  - Lines can be associated with a popup menu (right mouse click)
- Animated movement of shapes
  - When changing the position of a shape then the shape is moved to its new position with a certain animation.

---

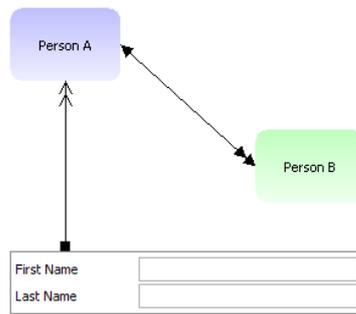
## Components

There are only a few components which you need to know about:

- PAINTAREA - this is the painting area on which shapes are placed. A painting area holds a certain size (width/height) and has a certain background coloring. If the shapes that are added to the PAINTAREA exceed the size of the PAINTAREA definition then scrollbars are shown automatically.
- PAINTAREAITEM - this is the typical shape component. It is an area in which you place drawing commands using the BGPAINT command set.
- PAINTAREAPANEITEM - this is a shape component that internally opens up a normal PANE components, so that you can place normal CaptainCasa components inside.
- PAINTAREALINEITEM - this is a connection between two points.

## Example

Have a look onto the following page:



The corresponding layout definition is rather simple:

```
<t:rowdemobodypane id="g_1" objectbinding="#{d.DemoPaintAreaSimple}">
  <t:row id="g_2">
    <t:paintarea id="g_3" background="#FFFFFF" height="100%" width="100%">
      <t:paintareaitem id="g_4"
        bgpaint="roundedrectangle(0,0,100%,100%,20,20,#C0C0FF,#F0F0FF,vertical);write(50%
        ,50%,Person A,centermiddle)" bounds="50;50;90;60" />
      <t:paintareaitem id="g_5"
        bgpaint="roundedrectangle(0,0,100%,100%,20,20,#C0FFC0,#F0FFF0,vertical);write(50%
        ,50%,Person B,centermiddle)" bounds="250;150;90;60" />
      <t:paintarealineitem id="g_6" arrowfrom="1" arrowto="2"
        bounds="140;80;110;100" />
      <t:paintareapaneitem id="g_7" bgpaint="#C0C0C0" border="#808080"
        bounds="50;250;290;52" padding="5" rowdistance="2">
        <t:row id="g_8">
          <t:label id="g_9" text="First Name" width="100" />
          <t:field id="g_10" width="100%" />
        </t:row>
        <t:row id="g_11">
          <t:label id="g_12" text="Last Name" width="100" />
          <t:field id="g_13" width="100%" />
        </t:row>
      </t:paintareapaneitem>
      <t:paintarealineitem id="g_14" arrowfrom="4" arrowto="6"
        bounds="95;110;0;140" />
    </t:paintarea>
  </t:row>
</t:rowdemobodypane>
```

You see:

- The position of shapes and lines is done using a BOUNDS property. This property is a semicolon separated definition of “left;top;width;height”.
- In the PAINTAREAITEM component the rounded rectangle and text is drawn using a corresponding BGPAIN attribute definition.
- The PAINTAREAPANEITEM component is defined very similarly to the PAINAREAITEM component but allows to put “normal” components inside.

---

## Getting into more complex Scenarios

The step from “simple” to “complex” is not too far:

- You may define the shape components with expression bindings instead of using static attributes. E.g. the BOUNDS property may be derived from a server side bean using just normal “#{...}” syntax.

- You may define the shape components dynamically using the COMPONENTBINDING attribute of PAINTAREA. In this case the jsp layout definition is not statically defined but is defined dynamically by your program. Read the information contained in the chapter “Dynamic Page Layout” for more details.
- You may use drag & drop features: PAINRAREAITEM and PAINTAREAPANEITEM provide a DRAGSEND property, PAINTAREA provides a DROPRECEIVE property. The drop event processing passes the exact pixel drop position to the server side processing so that you can update your shape arrangement correspondingly.

Check the examples within the demo workplace for more information.

# Working with Managed Beans

This chapter tells you about referencing properties of managed beans from user interface components. And it tells you about useful ways to add certain application functions into the server side request processing.

---

## Basics

Attributes of a user interface component (e.g. the TEXT attribute of a LABEL component) can be either defined in a static way (“First Name”) or in a dynamic way (“#{xxx.yyy}”). - Well, there are two exceptions of this rule - the attribute STYLEVARIANT and ATTRIBUTEMACRO can only be set in a static way, but this is the only exception.

A managed bean is a Java Bean object (at runtime) that is created and managed by the Java Server Faces environment.

A managed bean is declared in the faces-config.xml configuration file, telling the JSF environment about...:

- the class name
- the life cycle of an instance
- the logical name that is used inside referencing expressions

A valid faces-config.xml definition may look like:

```
<?xml version="1.0" encoding="UTF-8"?>
<faces-config
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
  version="1.2">
  <managed-bean>
    <managed-bean-name>demoHelloWorld
    </managed-bean-name>
    <managed-bean-class>test.DemoHelloWorld
    </managed-bean-class>
    <managed-bean-scope>session
    </managed-bean-scope>
  </managed-bean>
</faces-config>
```

A managed bean provides:

- properties and
- methods (action listeners)

that can be referenced from page definitions.

---

## Dynamic Properties

### Usage of Maps

The expression language that is used for referencing managed bean properties allows to gently work with Maps (java.util.Map-instances).

Assume the bean “aaaaa” of the previous example is implemented in the following way:

```

public class AaaaaBean ...
{
    HashMap m_values = new HashMap();
    public Map getValues()
    {
        return m_values;
    }

    ...somewhere...
    {
        m_values.put("firstName", "...");
        m_values.put("lastName", "...");
    }
}

```

In this case you can reference the map's items in the following straight way:

- `#{d.Aaaaa.values.firstName}`
- `#{d.aaaaa.values.lastName}`

You see: properties can be either hard-coded by providing a corresponding set/get implementation or can be soft-coded by using maps.

## Usage of Collections / Arrays

In the same way as described with maps you may define lists (implementations of `java.util.List`) or object arrays and reference them with expressions:

```

public class AaaaaBean ...
{
    List m_persons = new ArrayList<Person>();
    public List getPersons()
    {
        return m_persons;
    }
}

```

The expression to reference a certain item of the list look like:

- `#{d.aaaaa.persons[0].firstName}`
- `#{d.aaaaa.persons[3].lastName}`

## Dynamic Properties - Dynamic Bean Browser

When using dynamic properties then you may also want to reflect dynamic properties within the Bean Browser - the tool on the right of the Layout Editor, from which you can drag & drop expressions from the bean hierarchy into the component attributes.

There is a certain mechanism that allows to dynamically provide the property information for the bean browser. Please check the corresponding appendix of this documentation.

---

## Data Type Considerations

### ...without doing anything special...

With “fix-coded” properties, that provide a set/get method, things are simple: the data type that is used within the set/get-method definition is the one which is transferred into the object once the user updates the value on client side.

With dynamic attributes things are a bit more complex - but still simple:

- If there is already a property value available within the map or list, then a value that

was input on client side, is transferred with the same data type as the one of the already existing object.

Example: imagine your map already contains a value of type “Integer” which is accessed by key “weight”. If now the user updates the value on client side, then the server side processing tries to create a value of type “Integer” as well. It follows, what was already defined within the map or list.

- If there is no property value available within the map or list, e.g. the map does not contain a value yet, then the data type for passing changed properties depends on the component that manages the property. If the component is a CALENDAR component then a Date-value is passed. If the component is a FORMATTEDFIELD component with FORMAT “int” then an Integer-value is passed. If the component does not imply any specific data type then by default “String” is used.

In short words: for dynamic content (map, list, ...) first the server side processing tries to stick to the data type of the value which is already available. If none is available then a data type is chosen that fits to the component - if there is no specific one, “String” is chosen.

### ...with explicitly passing back type information...

In addition to the default mechanism described in the previous section you can explicitly implement an interface in order to pass back data type information:

```
package org.eclnt.jsfserver.util;  
  
public interface IPropertyTypeResolver  
{  
    public Class resolveType(String propertyName);  
}
```

You may implement this interface on any level that is accessed by expressions.

---

## Property Binding within Grid Processing

Within the grid processing (i.e. list and tree processing using FIXGRID component), there are the following binding rules:

- The grid itself binds to a property of type “FIXGRIDBinding” - either by using “FIXGRIDListBinding” or by using “FIXGRIDTreeBinding”. This binding definition is just a normal expression of type “#{aaa.bbb.grid}”.
- All elements inside grid rows (i.e. components that are located below the GRIDCOL control) are referenced with the special expression “.{xxx}”. When using the “.{ }” expression then the root for accessing the property is the corresponding row object of the grid.
- All rules that apply for normal expressions (e.g. dynamic properties), apply to the “.{ }” as well.

---

## Accelerated Property Access

When not using dynamic property binding, i.e. when not using maps or arrays, then the resolution of properties is done via normal Java introspection. For each property that is referred to within an expression there is a corresponding getter-/setter-method that is accessed correspondingly.

Example: if the expression is “#{d.TestUI.firstName}”, then the bean behind “TestUI” has a corresponding pair of methods:

```
public class TestUI
{
    public void setFirstName(String value) { ... }
    public String getFirstName() { ... }
}
```

The introspection of properties and their setter-/getter-methods is powerful on the one hand, but from performance of view is much slower than a direct access to the set/get method.

That's the reason why a special interface can accelerate the resolution of data:

```
package org.eclnt.jsfserver.util;

public interface IAcceleratedPropertyAccess
{
    public final static Object NOT_AVAILABLE = new Object();
    public Object getPropertyValue(String property);
}
```

If a managed bean implements this interface then the property resolution will first try to load the property via the interface. If the interface passes back a value that is not the NOT\_AVAILABLE object, then this value will be used - otherwise the normal introspection will be done.

The class "TestUI" could as consequence be accelerated in the following way:

```
public class TestUI implements IAcceleratedPropertyAccess
{
    public Object getPropertyValue(String property)
    {
        if ("firstName".equals(property))
            return getFirstName();
        return NOT_AVAILABLE;
    }

    public void setFirstName(String value) { ... }
    public String getFirstName() { ... }
}
```

The interface is only available for the getting of values - because this is executed much more often than the setting of values.

You typically should think about using the interface every time you have one and the same type of object being accessed a lot of times (e.g. accessed in grids).

---

## Method Binding

### Basics

The same rules that are used for referencing properties are used for referencing methods. A method expression (“#{d.aaaa.onXyz}”) must have a counter part method within the referenced bean. The method needs to have the following signature:

```
public class AaaaBean ...
{
    ...
    ...
    public void onXyz(ActionEvent event)
    {
        ...
        ...
    }
    ...
    ...
}
```

```
}
```

## One method, various events

All components of CaptainCasa Enterprise Client pass an event of type “BaseActionEvent”, which is a subclass of ActionEvent:

```
public void onXYZ(ActionEvent event)
{
    BaseActionEvent bae = (BaseActionEvent)event;
    ...
}
```

The base action event refers to the actual client event that triggered the request processing. It provides the functions:

- getCommand() ==> Client command
- getParams() ==> parameters, that are associated with the command

Please note: one component can be associated with different commands. Example: a FIELD component has one ActionListener-method in which the following events can be passed:

- flush ==> user input of data, only comes when FIELD-FLUSH=”true”
- drop, dropcopy ==> drop event, only comes when FIELD-DROPRECEIVE is configured
- popupmenu-command ==> only comes when POPUPMENU management is used

All events for these three commands are passed to one and the same method, that is the action listener method for this component. In order to better separate events from one another there is a sub-class of BaseActionEvent for each command:

```
BaseActionEvent
...
BaseActionEventDrop
BaseActionEventDropCopy
BaseActionEventFlush
...
```

Then name of the sub-class always is “BaseActionEvent+Command”. Each subclass provides - if required - straight access methods to the parameters that are associated with an event.

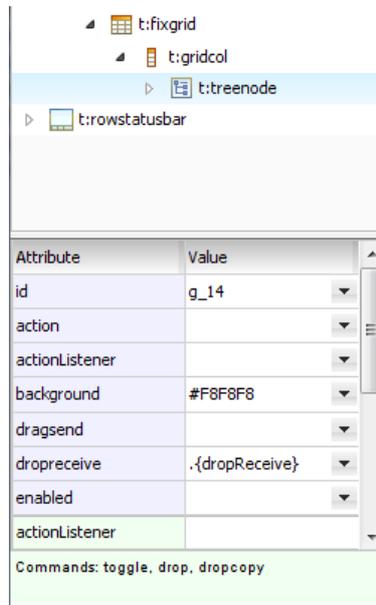
An event may be associated with additional data that is sent as event parameters. You can either access the parameters by the getParams() method of BaseActionEvent or you may use concrete accessor methods of the corresponding subclass of BaseActionEvent.

The typical structure of the method that acts as action listener looks like:

```
public void onXYZ(ActionEvent event)
{
    if (ae instanceof ActionEventFlush)
    {
        ...
    }
    else if (ae instanceof ActionEventDrop)
    {
        ...
    }
    else if (ae instanceof ActionEventDropCopy)
    {
        ...
    }
}
```

## Tool support

When maintaining component attributes within the Layout Editor then you will see a list of all commands that are available with a component:



The list of commands will look different, depending from the value of attributes.

---

## Adapter Binding

Since CaptainCasa Enterprise Client Release 3.0 there is a special type of binding available, the so called “adapter binding”.

### Purpose

A component, e.g. a field, provides a quite high number of attributes to define the way it looks and the way it behaves. In many cases the attribute values are defined by expressions, so that at runtime the attribute value is taken from server side bean property values.

Example: in case of a field the definition may look like:

```
<t:field ...
  text="#{d.xyz.name}"
  enabled="#{d.xyz.nameEnabled}"
  rendered="#{d.xyz.nameRendered}"
  bgpaint="#{d.xyz.nameBgpaint}"
.../>
```

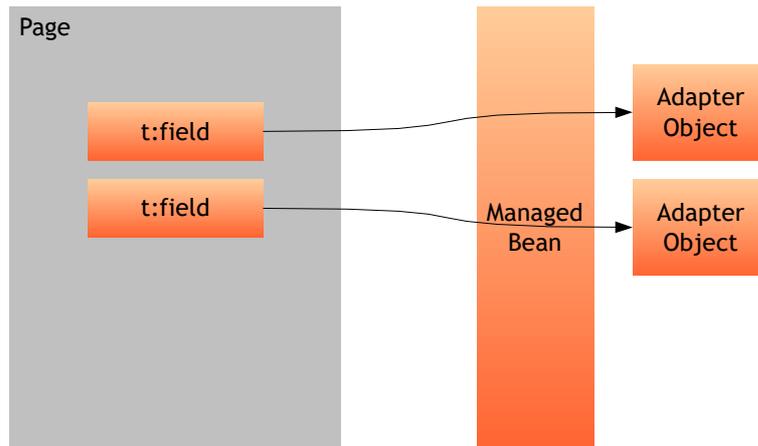
There is a quite high effort for defining the server side structures and for defining the expressions for each attribute. To reduce this effort CaptainCasa provides two possibilities that you can select from:

- You can use Macros in order to automatically generate the expressions (see “Working with Macros”)
- or...: you can use the adapter binding defined in this chapter.

### Basic Idea

The basic idea is very simple: a component (e.g. the field) is bound to a so called adapter object. The adapter object is a “mini object” directly providing certain attributes for the

component - so that the component delegates value requests directly to the adapter object. The adapter object is some kind of “counter part” for the component.



The components that support the usage of adapter objects provide an attribute ADAPTERBINDING, that needs to point to a stable instance of adapter object.

## Definition of Adapter Objects

Adapter objects implement the following interface:

```
public interface IComponentAdapterBinding
{
    public Set<String> getFixAttributeNames();
    public Set<String> getDynamicAttributeNames();
    public Class getAttributeType(String attributeName);

    public void setAttributeValue(String attributeName, Object value);
    public Object getAttributeValue(String attributeName);

    public void onAction(ActionEvent event);
}
```

The adapter tells the component which attributes it takes responsibility for, and then provides the corresponding set/get-access methods.

In case the component supports an action listener, the adapter's “onAction” method is called.

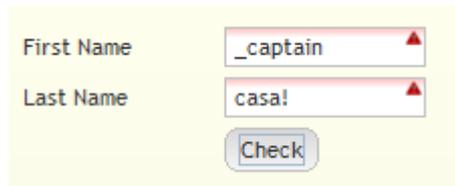
## What happens at Runtime

At runtime the component (e.g. within the render phase of the request processing) gathers all its attribute values in order to check if updated information needs to be sent to the client side. The corresponding attribute values are requested from the adapter object - if the component is created new then all (both the fix and dynamic attributes) are requested, if the component is already existing then only the dynamic attributes are requested.

If the component is able to updated a certain attribute (e.g. a FIELD component updates its TEXT attribute), then the update is done through the corresponding “setAttributeValue” method - in the case of update objects you also need to implement the “getAttributeType” method for the update-attribute.

## Example

The following example shows how to use the adapter binding for simple scenarios - in order to then cover more and more complex scenarios.



There are two fields, depending on the input a certain background painting is shown.

The jsp layout definition is:

```
<t:row id="g_2">
  <t:label id="g_3" text="First Name" width="100" />
  <t:field id="g_4"
    adapterbinding="#{d.DemoAdapterBindingSimple.firstName}"
    width="100" />
</t:row>
<t:row id="g_5">
  <t:label id="g_6" text="Last Name" width="100" />
  <t:field id="g_7"
    adapterbinding="#{d.DemoAdapterBindingSimple.lastName}"
    width="100" />
</t:row>
<t:row id="g_8">
  <t:col distance id="g_9" width="100" />
  <t:button id="g_10"
    actionListener="#{d.DemoAdapterBindingSimple.onCheck}"
    text="Check" />
</t:row>
```

You see that both field components are bound via adapter binding.

The code is:

```
public class DemoAdapterBindingSimple
  implements Serializable
{
  // -----
  // inner classes
  // -----

  static Set<String> MYFIELDADAPTER_FIXATTRIBUTES;
  static Set<String> MYFIELDADAPTER_DYNATTRIBUTES;

  static
  {
    MYFIELDADAPTER_FIXATTRIBUTES = new HashSet<String>();
    MYFIELDADAPTER_DYNATTRIBUTES = new HashSet<String>();
    MYFIELDADAPTER_DYNATTRIBUTES.add("text");
    MYFIELDADAPTER_DYNATTRIBUTES.add("enabled");
    MYFIELDADAPTER_DYNATTRIBUTES.add("bgpaint");
  }

  public class MyFieldAdapter implements IComponentAdapterBinding
  {
    boolean i_containsError = false;
    String i_text = "";
    boolean i_enabled = true;
    public Set<String> getFixAttributeNames()
    { return MYFIELDADAPTER_FIXATTRIBUTES; }
    public Set<String> getDynamicAttributeNames()
    { return MYFIELDADAPTER_DYNATTRIBUTES; }
    public Class getAttributeType(String attributeName)
    {
      if ("text".equals(attributeName))
        return String.class;
      else
        throw new Error("The attribute " + attributeName +
          " is not supported!");
    }
    public Object getAttributevalue(String attributeName)
    {
      if ("text".equals(attributeName))
        return i_text;
      else if ("enabled".equals(attributeName))
```

```

        return i_enabled;
    else if ("bgpaint".equals(attributeName))
    {
        if (i_containsError == true)
            return "error()";
        else
            return null;
    }
    else
        throw new Error("The attribute " + attributeName +
            " is not supported!");
}
public void setAttributevalue(String attributeName, Object value)
{
    if ("text".equals(attributeName))
        i_text = (String)value;
    else
        throw new Error("The attribute " + attributeName +
            " is not supported!");
}
public void onAction(ActionEvent event) {}
}

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

MyFieldAdapter m_firstName = new MyFieldAdapter();
MyFieldAdapter m_lastName = new MyFieldAdapter();

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

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

public MyFieldAdapter getFirstName() { return m_firstName; }
public MyFieldAdapter getLastName() { return m_lastName; }

public void onCheck(ActionEvent event)
{
    m_firstName.i_containsError = false;
    m_lastName.i_containsError = false;
    // some "checks"
    if (m_firstName.i_text.startsWith("_"))
        m_firstName.i_containsError = true;
    if (m_lastName.i_text.endsWith("!"))
        m_lastName.i_containsError = true;
}

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

```

The class “MyFieldAdapter” is defined as inner class. It takes over responsibility for the attributes “text”, “bgpaint”, “enabled”. Please not that all properties are defined within the set of dynamic properteis. And, please note: even though there is no fix attribute that is provided by “MyFieldAdapter” you need to return an empty set.

The value of “bgpaint” is derived from a flag indicating if the adapter binding contains an error or not (“i\_containsError”).

## Overriding Adapter Binding

If an attribute that is provided by an adapter binding is explicitly defined within the .jsp layout definition, then (of course...) the explicitly defined value overrides the value coming from the adapter binding.

## Related Information

There are some helper classes implementing `IComponentAdapterBinding`: “`ComponentAdapterBindingBase`” and “`ComponerAdapterBindingMap`”. Please check the JavaDoc if these helper classes are useful for you.

Please check the chapter “Working with Macros” as well, which provides a - in some parts similar way - of simplifying the working with “rich sets of dynamic attributes per component”.

---

## Exception Management

The normal exception management within the JSF processing is not specified too much - but leaves the exception processing to the servlet container. In order to provide some more “nicer” management if exceptions the CaptainCasa sever side framework provides some additional functions on top of JSF.

During a server side request life cycle there are the three important phases of an application processing:

- Data Transfer Phase - this is the phase when values from the user interface client are passed into the server side managed beans (“set-ter methods are called”)
- Invoke Phase - this is the phase when action listeners are called
- Render Phase - this is the phase when the page is rendered, i.e. when expressions against managed beans are resolved (“get-ter methods are called”).

(Yes, there are some more phases defined in JSF but these are left out here for simplification purpose.)

From consistency point of view the data transfer phase and the invoke phase are most critical:

- An exception in the set-phase may result in inconsistency of data because a value that was transferred from the client side could not be transferred into the application.
- An exception in the invoke phase may result in inconsistency of data because while updating information within your application processing the processing might have been interrupted by the exception.

The render phase is not as critical because no data is changed within this phase - but data is just being picked.

## Default Handling of Exception

By default an exception that is thrown by your application processing during the data transfer and the invoke phase leads to an abort of the current processing. The user received an error screen in which the exception information is shown. This error screen can be customized - you find more information about how to define the error screen in one of the following chapters.

By default an exception that is thrown by your application processing during the render phase is ignored - there are some information messages that are registered within the log, that's it.

## Additional Functions on top of JSF

During the data transfer (“set”) and the invoke (“actionListener”) phase there are the following additional functions:

- If an exception occurs within the “set”-method or the “actionListener”-method then the method “onApplicationErrorDuringSet(...)” or “onApplicationError(...)” are called exactly on this object level, on which the “set”-method or “actionListener”-method was called.
- If the corresponding method exists and if this method itself does not throw an exception, then the request processing continues. If not then the next upper level of the expression is called with exactly the same methods.

Uuuh, this sounds complex, but indeed isn't. Example:

A button is bound to “#{d.aaa.bbb.onXyz}”. Let's check what happens...:

```
#{d.aaa.bbb.onXyz} ==> onXyz is called!
!!! onXyz processing throws exception/error

#{d.aaa.bbb.onApplicationError} is called, if available
#{d.aaa.onApplicationError} is called, if available
ã{d.onApplicationError} is called, if available
```

Please note: the “Walking up the expression stack” is interrupted after the first “onApplicationError” method was processed without throwing an exception/error itself.

The same happens when during the data transfer phase a property is set:

```
#{d.aaa.bbb.firstName} ==> setFirstName(...) is called
!!!setFirstName processing throws exception/error

#{d.aaa.bbb.onApplicationErrorDuringSet} is called, if available
#{d.aaa.onApplicationErrorDuringSet} is called, if available
ã{d.onApplicationErrorDuringSet} is called, if available
```

## Consequences for Implementation

A normal impementation, without using the additional error management may look like:

```
public void onXyz(ActionEvent event)
{
    try
    {
        ...
        ...
    }
    catch (RuntimeException r)
    {
        StatusBar.outputAlert(...);
    }
    catch (Error e)
    {
        StatusBar.outputAlert(...);
    }
}
```

With using the optional exception management your code now looks the following way:

```
public void onXyz(ActionEvent event)
{
    ...
    ...
}

public void onApplicationError(ApplicationErrorInfo aei)
{
    StatusBar.outputAlert(...);
}
```

This means: in case a method (“onXyz”) causes an error, automatically a method “onApplicationError()” is called. This method then can then handle the error management, i.e. It may do some output to the status bar.

All information about the original error is passed via the “ApplicationErrorInfo” object.

You may use the interface “IErrorAware” in order to implement the “onApplicationError\*” methods on the corresponding object level. (The calling of the methods does not depend on the extension of this interface, so the interface is “just” a helper for implementation purposes.)

```
package org.eclnt.jsfserver.util;

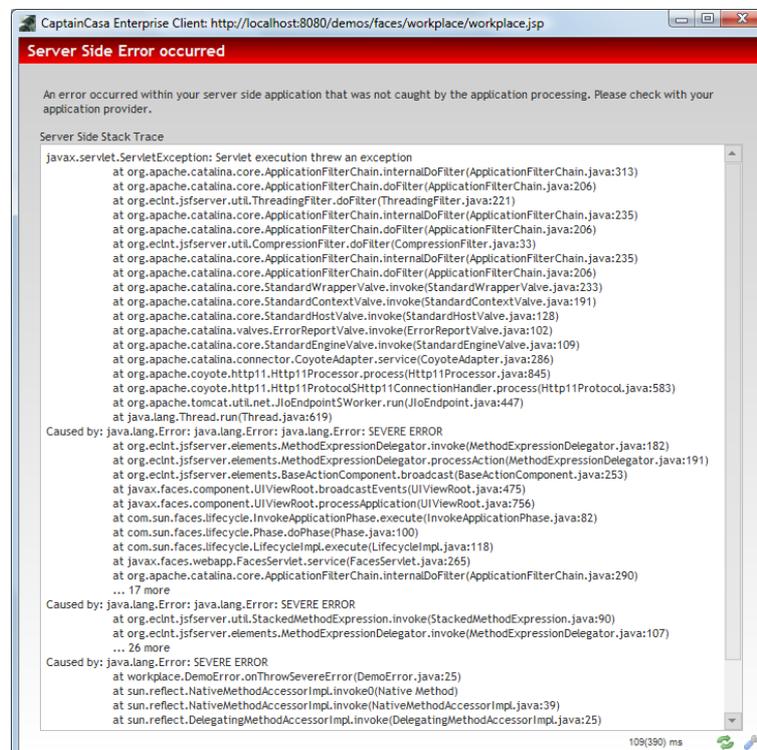
import org.eclnt.jsfserver.elements.ApplicationErrorInfo;
import org.eclnt.jsfserver.elements.ApplicationErrorInfoDuringSet;

public interface IErrorAware
{
    /**
     * This method is called in case of an error when a method expression
     * is executed during the INVOKE-phase of the JSF processing.
     * <br><br>
     * In case this method throws an error/ runtime exception itself
     * then this error is delegated correspondingly.
     */
    public void onApplicationError(ApplicationErrorInfo aei);

    /**
     * This method is called in case of an error when a set-value-expression
     * is executed during the SET-phase of the JSF processing.
     * <br><br>
     * In case this method throws an error/ runtime exception itself
     * then this error is delegated correspondingly.
     */
    public void onApplicationErrorDuringSet(ApplicationErrorInfoDuringSet aeids);
}
```

## Customizing the Server Side Error Screen

The default screen that comes up if an error was not caught by the application processing is:



The name of the corresponding page is “/eclntjsfserver/includes/showservererror.jsp”.

You may define your own error screen layout (e.g. copy the original page and modify it). The name of the updated page needs to be passed as client parameter “errorscreen”.

## Best Practice: Checking if your whole Application is available

There are certain checks that you may want to do in general, with every request. E.g. check if your application is connected to the database, if a user is logged on, etc. In this case you may define you “out-most” page in the following way:

```
<t:rowinclude page="/rescuepage.jsp" rendered="#{d.appNotOK}"/>
<t:rowinclude page="/normalpage.jsp" rendered="#{d.appOK}"/>
```

“appOK/appNotOK” are booleans that check the health-status of the application. If the application is not healthy anymore then the rescuepage is shown, otherwise the normal application is shown.

---

## Property Change Notification

There are a couple of interfaces that are considered when a property value is set into a managed bean. The interfaces are executed (if available) during the JSF phase, in that all client side value changes are transferred into the corresponding managed beans (“update phase”).

Who is calling the interfaces? It's the CaptainCasa/JSF environment. JSF allows to bring in own processing in various parts of the normal request processing. E.g. for transferring a value that was changed within the user interface client, there is a so called “property resolver”. CaptainCasa provides for own resolvers that provide the additional functions to check for the interfaces that are described within the following text.

Please also check the JavaDoc documentation that is available for the interfaces.

### Interface IPropertyResolverAware

A bean that is updated by the user interface request processing may optionally implement the interface “IPropertyResolverAware”:

```
package org.eclnt.jsfserver.util;

public interface IPropertyResolverAware
{
    public void reactOnSetValue(Object property, Object value);
    public void reactOnSetValue(int index, Object value);
}
```

The interface is used by the so called property resolver: the property resolver is an object within the CaptainCasa JSF processing that is responsible for passing values coming from the user interface client into corresponding beans that are bound by expressions (e.g. “#{d.address.street}”). The interface is called after the corresponding value was set.

In case the property resolver updates the property of a bean, it will check if the bean implements the IPropertyResolverAware-interface. If yes, then the corresponding method of the interface is called.

Example: if the value “#{d.address.street}” is set, then the bean behind “#{d.address}” will be checked for the implementation of IPropertyResolverAware.

### Interface IPropertyResolverAware2

This interface is similar to IPropertyResolverAware, but not only tells about direct value

changes to the current object, but also tells about value changes within a sub-object of the current object.

```
public interface IPropertyResolverAware2
{
    public void reactOnSetValue(String completeExpression,
                               String propertyName,
                               Object value);

    public void reactOnSetValue(String completeExpression,
                               int index,
                               Object value);

    public void reactOnSetValueInSubObject(String completeExpression,
                                           Object value);
}
```

## Interface IPropertyValueConverter

In special situations it is required to change objects coming from the user interface before transferring them into the corresponding object-property. This is when you may use the interface IPropertyValueConverter:

```
public interface IPropertyValueConverter
{
    public Object convertObject(Object property, Object value);
    public Object convertObject(int index, Object value);
}
```

The interface is called before the value is set into the corresponding property. You may update the value within your interface implementation.

## Interface IValueBindingListener

While the interfaces IPropertyResolverAware and IPropertyValueConverter operate on the last bean level, the one where the property is set, the interface IValueBindingListener is a more general one. It gets notified whenever a value binding is executed within the JSF phase when values are transferred into the managed beans.

```
package org.eclnt.jsfserver.util;

public interface IValueBindingListener
{
    public void reactOnValueBindingSet(ValueBinding valueBinding,
                                       Object value);
}
```

For using the interface, you need to...:

- (1) implement a class, that implements the interface
- (2) register an instance of the class within the session context

An example is:

```
// implementation if IValueBindingListener
public class MyValueBindingListener implements IValueBindingListener
{
    String m_log = "";
    public void reactOnValueBindingSet(ValueBinding valueBinding,
                                       Object value)
    {
        m_log += "IValueBindingListener, Property change: " +
            ValueBindingUtil.getOriginalExpressionString
                (valueBinding) +
            " / " + value + "\n";
    }
}
```

```
// somewhere in the initialization part of your code:
HttpSessionAccess.setValueBindingListener
    (new MyValueBindingListener());
```

## Class CascadingValueBindingListener

There is an implementation of `IValueBindingListener` that comes with the CaptainCasa standard delivery: `CascadingValueBindingListener`. This implementation notifies each object level of an expression that is just set, so that the corresponding level can react.

Example: if a value is passed into “`#{d.d_1.PersonUI.firstName}`”, then all object levels of the expression are notified:

```
#{d.d_1.PersonUI}
#{d.d_1}
#{d}
```

As consequence you can e.g. set flags, in which you keep the information that something has changed within the corresponding object. The notification is done by checking if each object supports interface `ICascadingValueBindingListener` - the interface is an inner class definition within `CascadingValueBindingListener`.

In the following example on grid item level this change information is managed by using `CascadingValueBindingListener`:

```
package workplace;

import java.io.Serializable;

import javax.faces.el.ValueBinding;
import javax.faces.event.ActionEvent;

import org.eclnt.editor.annotations.CCGenClass;
import org.eclnt.jsfserver.elements.impl.FIXGRIDItem;
import org.eclnt.jsfserver.elements.impl.FIXGRIDListBinding;
import org.eclnt.jsfserver.util.CascadingValueBindingListener;
import org.eclnt.jsfserver.util.HttpSessionAccess;
import
org.eclnt.jsfserver.util.CascadingValueBindingListener.ICascadingValueBindingList
ener;
import org.eclnt.workplace.IworkpageDispatcher;

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

public class DemoCascadingValueBinding
    extends DemoBase
    implements Serializable
{
    // -----
    // inner classes
    // -----

    public class GridItem
        extends FIXGRIDItem
        implements java.io.Serializable, ICascadingValueBindingListener
    {
        boolean m_changed = false;

        protected String m_carId;
        public String getCarId() { return m_carId; }
        public void setCarId(String value) { m_carId = value; }

        protected String m_carType;
        public String getCarType() { return m_carType; }
        public void setCarType(String value) { m_carType = value; }

        public boolean getChanged() { return m_changed; }

        public void reactOnSetValue(ValueBinding arg0,
```

```

ValueBinding arg1,
Object arg2)
    {
        m_changed = true;
    }
}

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

protected FIXGRIDListBinding<GridItem> m_grid = new
FIXGRIDListBinding<GridItem>();
protected String m_street;
protected String m_lastName;
protected String m_firstName;

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

public DemoCascadingValueBinding(IWorkpageDispatcher dispatcher)
{
    super(dispatcher);
    HttpSessionAccess.setValueBindingListener
    (new CascadingValueBindingListener());
    for (int i=0; i<10; i++)
    {
        GridItem gi = new GridItem();
        gi.setCarType("Volkswagen");
        gi.setCarId("HD - BM 666"+i);
        m_grid.getItems().add(gi);
    }
}

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

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

public FIXGRIDListBinding<GridItem> getGrid() { return m_grid; }
}

```

---

## JSF Phase Management

### Overview

During a server side request processing the following activities are executed:

- Data is decoded from the client request - this is something which is transparent for the business application, i.e. this is done by the JSF components implicitly-
- Data is transferred into managed beans (update phase)
  - “setXyz(…)”
- Methods are invoked within beans (invoke phase)
  - “onXyz(ActionEvent event)”
- The page response is rendered (render phase) and sent back to the client. Within this process the components ask the managed beans for their current values.
  - “getXyz()”

JSF knows some additional phases which are currently not relevant when using CaptainCasa Enterprise Client.

## Starting Runnables to be executed in a certain Phase

In some cases it makes sense that managed beans are aware of certain JSF phases, that are processed during a request processing. The most important JSF phases that are relevant for CaptainCasa Enterprise Client processing are:

- decode phase (request values are applied JSF component tree)
- update phase (data changes are transferred to the managed beans)
- invoke phase (actionListeners are invoked)
- render phase (the XML for the client is rendered, the JSF component tree is encoded)

Example: values are passed into a status bar in order to be output on client side. But: the values should be automatically set back to initial values, in order to not show the same message again and again.

The base for listening to JSF phases is provided by JSF itself: there is the possibility to register phase listeners within the faces-config.xml file. These phase listeners are called during request processing and can communicate to the application by using the JSF context (FacesContext). Please read more details about this within the JSF documentation.

CaptainCasa Enterprise Client provides a simple mechanism which is based on the JSF phase listener concept that allows to simply attach “Runnables” to the beginning or the ending of certain phases. The interface is pretty simple:

```
public class PhaseManager
{
    ...
    public static void runAfterRenderResponsePhase(Runnable run) {...}
    public static void runBeforeRenderResponsePhase(Runnable run) {...}
    ...
}
```

The runnable object is executed at the point of time represented by the method name. Currently the PhaseManager supports “Runnables” that are executed before or after the “render response phase”.

Let's assume you want to output an error information only one time to the client. The coding might look the following way:

```
public class XYZBean implements Serializable
{
    public static class OutputTextClearer implements Runnable, Serializable
    {
        public void run()
        {
            m_outputText = "";
        }
    }
    ...
    public void onSave(ActionEvent event)
    {
        ...
        m_outputText = "Error occurred when saving";
        PhaseManager.runAfterRenderResponsePhase(new OutputTextClearer());
        ...
    }
    ...
}
```

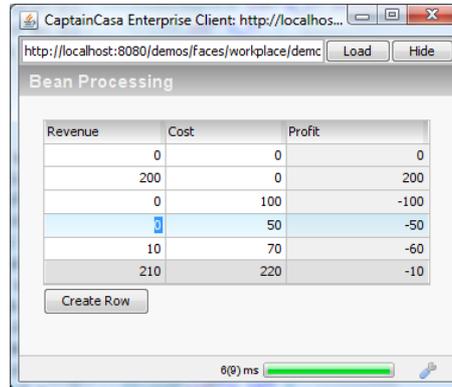
The method “onSave” is called in the invoke phase. After the invoke phase the render phase is processed, in which the XML is built that is sent to the client. At the end of the phase the runnable is called, setting back the outputText-value.

After executing the runnables of one phase, the runnables are removed. Runnables are processed only one time.

## Adding some BEANPROCESSING Statements into the Page

In addition to adding “Runnables” by program (previous chapter) you also can add “Runnables” by declaration. There is a component BEANPROCESSING and a component BEANMETHODINVOKER to do so.

Have a look onto the following example:



The user can input numbers into the grid. Every time a request is processed on server side, the profit and the aggregated number in the footer line will be updated.

Have a look into the layout definition:

```

<t:beanprocessing id="g_2" >
  <t:beanmethodinvoker id="g_3"
    actionListener="#{d.demoBeanprocessing.onUpdateGrid}"
    jsfphase="invokeEnd" />
</t:beanprocessing>
<t:rowtitlebar id="g_4" text="Bean Processing" /><t:rowbodypane id="g_5"
rowdistance="2" >
  <t:row id="g_6" >
    <t:fixgrid id="g_7" bordercolor="#C0C0C0" borderheight="1" borderwidth="1"
      objectbinding="#{d.demoBeanprocessing.rows}"
      rowheight="16" sbvisibleamount="5" >
      <t:gridcol id="g_8" text="Revenue" width="100" >
        <t:formattedfield id="g_9" align="right" format="int" value="
{revenue}" />
      </t:gridcol>
      <t:gridcol id="g_10" text="Cost" width="100" >
        <t:formattedfield id="g_11" align="right" format="int" value=".{cost}" />
      </t:gridcol>
      <t:gridcol id="g_12" text="Profit" width="100" >
        <t:formattedfield id="g_13" align="right" background="#F0F0F0"
enabled="false"
          format="int" value=".{profit}" />
      </t:gridcol>
      <t:gridfooter id="g_14" >
        <t:formattedfield id="g_15" align="right" background="#E0E0E0"
enabled="false"
          format="int" value="#{d.demoBeanprocessing.totalRevenue}" />
        <t:formattedfield id="g_16" align="right" background="#E0E0E0"
enabled="false"
          format="int" value="#{d.demoBeanprocessing.totalCost}" />
        <t:formattedfield id="g_17" align="right" background="#E0E0E0"
enabled="false"
          format="int" value="#{d.demoBeanprocessing.totalProfit}" />
      </t:gridfooter>
    </t:fixgrid>
  </t:row>
  <t:row id="g_18" >
    <t:button id="g_19" actionListener="#{d.demoBeanprocessing.onCreateRow}"
      text="Create Row" />
  </t:row>
</t:rowbodypane>

```

The interesting part is the highlighted one: there is the definition that the method “#{d.demoBeanprocessing.onUpdateGrid}” is executed at the end of the invoke phase. This means: whatever happens in the update or in the invoke phase, this method is processed at the end of the invoke phase.

Have a look into the code of the managed bean:

```
package workplace;

import javax.faces.event.ActionEvent;

import org.eclnt.editor.annotations.CCGenClass;
import org.eclnt.jsfserver.elements.impl.FIXGRIDItem;
import org.eclnt.jsfserver.elements.impl.FIXGRIDListBinding;

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

public class DemoBeanProcessing extends DemoBase
{
    static int MAX_ROWS = 5;

    public DemoBeanProcessing()
    {
        for (int i=0; i<MAX_ROWS; i++)
        {
            MyRow r = new MyRow();
            m_rows.getItems().add(r);
        }
    }

    public class MyRow extends FIXGRIDItem implements java.io.Serializable
    {
        protected int m_profit;
        public int getProfit() { return m_profit; }
        public void setProfit(int value) { m_profit = value; }

        protected int m_cost;
        public int getCost() { return m_cost; }
        public void setCost(int value) { m_cost = value; }

        protected int m_revenue;
        public int getRevenue() { return m_revenue; }
        public void setRevenue(int value) { m_revenue = value; }
    }

    protected FIXGRIDListBinding<MyRow> m_rows = new FIXGRIDListBinding<MyRow>();
    public FIXGRIDListBinding<MyRow> getRows() { return m_rows; }

    protected int m_totalProfit;
    public int getTotalProfit() { return m_totalProfit; }

    protected int m_totalCost;
    public int getTotalCost() { return m_totalCost; }

    protected int m_totalRevenue;
    public int getTotalRevenue() { return m_totalRevenue; }

    public void onUpdateGrid(ActionEvent event)
    {
        m_totalRevenue = 0;
        m_totalCost = 0;
        m_totalProfit = 0;
        for (int i=0; i<MAX_ROWS; i++)
        {
            MyRow r = m_rows.getItems().get(i);
            r.m_profit = r.m_revenue - r.m_cost;
            m_totalRevenue += r.m_revenue;
            m_totalCost += r.m_cost;
            m_totalProfit += r.m_profit;
        }
    }

    public void onCreateRow(ActionEvent event)
    {
        MyRow r = new MyRow();
    }
}
```

```
        m_rows.getItems().add(r);  
    }  
}
```

In the method all the calculated numbers are re-calculated.

You see: by using the BEANMETHODINVOKER component you can make sure that a certain processing that is relevant for the user interface processing is always called with a certain request processing phase.

This makes programming in many cases much easier and structured. In the example the calling of the “onUpdateGrid()” method makes sure that the calculated numbers are correct - there is no necessity to make sure in a fine-granular way, that every update of numbers needs to also update the calculated numbers.

---

## Http Session Management

### Management of HTTP Sessions

The server part of CaptainCasa Enterprise Client is based on JSF processing - and as consequence is based on the normal HTTP session management provided by the servlet container. All application session state is kept within the HTTP session. This is done automatically for managed beans - remember that in faces-config.xml you define the “managed-bean-scope” for your managed beans.

Because the front end part of CaptainCasa Enterprise Client can be started in three different modes there are three different cases to look at when explaining the creation of http sessions on server side:

- Start as Web Start: each front end, that is started by webstart receives its own http session on server side.
- Start as Applet: by default each front end that is started receives its own http session on server side. This may first sound confusing - because the browsers typically manages one session per browser instance or per several browser instances, using a cookie that transfers the session id from client to server. But: CaptainCasa Enterprise client uses URL encoding for passing the session id and explicitly does NOT use cookies. As consequence the session of the application running inside an applet by default does not share the session with its embedding application around. This default behavior has a lot of advantages: you may open an applet twice or more times within your browser environment - each starting of the applet on server side is transferred into an own http session. There will never be the case that two applet instances are accessing the same http session.

For special situations there is a special configuration of the client, that tells the client to take over the session id of the embedding html page. In this case you need to embed the applet by using “.ccapplet” and by appending the URL parameter “?ccsamesession=true”. But: now you need to be aware of that your server side session might be used by several applets in parallel. It's the outside application, embedding the applet into HTML pages, that must take care of the proper assignment of server side sessions.

- Start as Client Application: each front end that is started receives its own http session on server side.

### Accessing Http Session

Sometimes you may want to access the http session in which your managed beans are

kept. CaptainCasa provides some convenience functions that unburden you from working with JSF interfaces directly. Internally, these functions do nothing else than wrapping default JSF APIs.

The class “HttpSessionAccess” provides a collection of static functions:

```
public static HttpSession getCurrentHttpSession()
public static HttpSession getCurrentHttpSession(FacesContext context)
public static HttpServletRequest getCurrentRequest()
public static ServletContext getServletContext()
public static String getServletTempDirectory()
public static HttpServletResponse getCurrentResponse()
```

There is one important issue to pay attention to: you may only call these functions in the context of processing a certain request from the client.

Well, of course this is the default case: your code is triggered by a client side event that sends an http request to the server side processing. But: there may be situations when you start a second thread on server side for long running tasks - in this case the thread is not bound to http processing but to your own processing. As consequence you must not use the HttpSessionAccess functions!

## Reacting on Closing of Http Session

The class HttpSessionAccess provides a nice way of getting notified when an http session is closed:

```
public static HttpSessionListenerDelegator
getCurrentHttpSessionListenerDelegator()
```

By accessing the “getCurrentHttpSessionListenerDelegator()” method you receive an object, in which you can add implementations of interface “IHttpSessionClosedReactor”.

The interface looks as follows:

```
public interface IHttpSessionClosedReactor
{
    public void reactOnClosed();
}
```

When using the Workplace Management then this interface is already implemented by certain classes:

- The “DefaultDispatcher” (and its sub-classes, e.g. “WorkpageDispatcher”) registers on top level to be notified when a session is closed. In this case the closing is delegated to the “destroy()” method of the dispatcher.
- The dispatcher’s destroy method itself delegates the closing to all instances that have registered as life cycle listeners.

Please find more information about the workplace management within the corresponding chapter of this documentation.

## Closing the Session when Client is closed

The default way a session is closed on server side is by timeout. Even if the user closes his/her UI client (or in case of applet: if the user leaves the applet page) - even then the server does by default not receive a signal to invalidate the server side session.

But: there's a way to do so. Place the component SESSIONCLOSER into the out-most page of your application. The component will send a session-invalidate-signal to the server side at this point of time when it is removed, i.e. when the out-most page is closed.

Example: the demo workplace's out-most page is “/demos/workplace/workplace.jsp”:

```
<t:beanprocessing id="g_1" >
  <t:sessioncloser id="g_2" />
</t:beanprocessing>
<t:row id="g_3" >
  <t:pane id="g_4" bgpaint="#{d.workplace.bgpaint}" height="100%"
    popupmenu="WORKPLACE" requestfocus="creation" width="100%" >
    <t:rowbodypane id="g_5" background="null!"
      bgpaint="rectangle(0,20,100%,1,#FFFFFFF30)"
      padding="top:4;bottom:10;left:20;right:20" >
      <t:row id="g_6" >
        ...
        ...
        ...
      </t:row>
    </t:rowbodypane>
  </t:pane>
</t:row>
```

You see that - positioned below the component BEANPROCESSING - there is the component SESSIONCLOSER.

## Define Name of Session Id encoded into URLs

All communication between the front end of CaptainCasa Enterprise Client and the server part are not relying on passing cookies between client and server but are relying on proper URL encoding of session ids. By default URL encoding is inserting a certain string into a URL - “;jsessionid=xxxxxx”.

In some situations you may configure your web/ application server so that it uses a different name for encoding the session id - e.g. “;myjsessionid=xxxxxx”. In this case you need to pass this information into the front end processing of CaptainCasa Enterprise Client: use client parameter “jsessionidname” to tell the client about the changed name of the session id.

A full list of client parameters is provided in the appendix of this documentation.

## Configuring the Session Timeout

The timeout of a server side session is defined within the web.xml deployment descriptor of your application:

```
...
...
</servlet-mapping>

<session-config>
  <session-timeout>30</session-timeout>
</session-config>

<mime-mapping>
...
...
```

The value is interpreted as “duration of inactivity in minutes”. When passing “-1” then a session will never be timed out. Please use “-1” only when using CaptainCasa Enterprise Client in embedded mode.

---

## Complex Expressions

The CaptainCasa server runtime is based on Java Server Faces (JSF). The expression management (e.g. “#{d.xyz.abc}”) that is part of CaptainCasa is delegated to the expression management of JSF.

In this documentation we only refer to simple expressions:

- “#{d.abc.def}” for direct property binding (the property might be a static one with set/get - or might be a map-based one...)

- “#{d.abc.list[4]}” for binding to list/array structures
- “#{d.abc.map['harry]}” for binding to map
- “#{d.abc.map['harry'].list[3].firstName}” - ...all combinations of simple expressions

In general we (strongly) recommend to keep with simple expressions - and at least to be very careful when using more complex expressions, that are provided by the JSF expression management as well.

Examples for complex expressions are:

- “#{d.abc.def] is nice.”
- “#{d.abc.def == true}”
- “#{d.abc.def + 5}”
- “#{d.ab.def + d.abc.ghi}”

There are three reasons:

- Structural: first it is nice to keep any type of logic in one place (in the code) and not to have logic distributed over several instances... If a LABEL shows some wrong value, you have to check or debug the code - and do not have to analyze complex expressions that are executed in front of the code. Of course this may have the consequence that you have to create some more properties within your code - but again: you know where to go to if things are not working...
- Performance: CaptainCasa comes with an own expression resolver which is optimized for simple expressions. This resolver is much faster than the default expression resolver. If you only have “some” properties to be resolved, than you will not run into trouble. But if you e.g. have a grid with 20 rows and 50 columns, then the number of resolutions per roundtrip is high - and then the difference will become feel-able.
- Problems with expression replacements: CaptainCasa uses expression replacement in certain areas:
  - in the grid/repeat processing
  - in the workplace processing
  - in the managed bean processing.

Expression replacements update the original expression that is defined in the JSP-XML definition and adapt the original expression to the concrete situation. If complex expressions are not defined in an adequate way, the expression replacement might fail and the binding between page components and server side objects will fail.

## Expression Replacements

Let's take a look into expression replacements in order to check, what type of complex expressions are possible (though not recommended...!).

- In the grid area, a grid binds to some FIXGRIDBinding instance via its attribute OBJECTBINDING (e.g. “#{d.abc.grid}”). Inside the grid columns the expression typically starts with a dot-notation (e.g. “.{firstName}”). At runtime the grid iterates through the visible grid items and replaces the leading “.{“ of the column expression with the corresponding row of the grid-expression. The result for example is: “#{d.abd.grid.rows[0].firstName}”.

If the expression of the grid column is “.{firstName} xxx” then the result will be “#{d.abd.grid.rows[0].firstName} xxx”, which is fine.

If the grid column expression does NOT start with “.{“ (e.g. “xxx .{firstName}”) then the replacement will not succeed and the concatenated expression is not correct. The same happens if having multiple expressions inside the grid column expression (e.g. “. {revenue - cost}”)

- In the workplace area the leading “#{d.” of the expressions is replaced by the corresponding sub-dispatcher that a workpage belongs to. Example: “#{d.abc.def}” is changed to “#{d.d\_1.abc.def}”. The effect here: all the beans running in the scope of a dispatcher are automatically running in the scope of an isolated sub-dispatcher. - Similarly to the grid management, only expressions that really start with “#{d.” are updated - expressions consisting out of several property references are not correctly updated (e.g. “#{d.abc.revenue - d.abc.cost}”).
- In the managed bean processing the root expression of a page is the one that is replaced by the current bean's expression. Example: in a managed bean there is an expression “#{d.person.firstName}”, the bean's root expression is “#{d.person}”. Now this bean is used in the department bean, e.g. to include the person-screen of the department's leader. The expression for including the page bean is “#{d.department.leader}”.

In the fully scenario “#{d.person}” of the included screen is replaced with “#{d.department.leader}” so that the full expression is “#{d.department.leader.firstName}”.

Again: expressions must start with “#{d.person}” in order to be replaced. Expressions with two property references will fail (“#{d.person.salary + d.person.bonus}”) because only the first property reference starts with a “#{d.person}”.

## Consequence

The first consequence is: keep yourself out of the game of complex expressions!!! You see: this is the only chapter of the documentation mentioning them at all...

If you want to use complex expressions only use these ones...

- ...starting with “#{“ or with “.{“
- ...not holding additional references two, where the second one does not hold a leading “#{“ or “.{“

If you want to use complex expressions: do not use them in the grid column processing for server performance reason.

# Dynamic Page Layout

When working with the CaptainCasa Enterprise Client toolset then the normal sequence of operations is:

- You define the layout (.jsp) file, using the Layout Editor. In the layout file you reference to managed beans using JSF expressions.
- You implement the managed beans.

The advantage of working this way is: the layout definition is done in a declarative way and it is kept outside the bean processing.

There are situations as well, in which it is difficult to completely specify a layout in a static way, because the layout depends on the logic.

---

## Overview

There are two general ways to provide dynamic page layouts:

- (A) - “by page”  
When one page is included into another page (using ROWINCLUDE component) then the page name of the included page (“/xyz/abc.jsp”) may either be a page name pointing to a static file that is part of the web content directory - or it may point to a program that dynamically generated the page’s XML content.
- (B) - “within a page”  
Inside a static page definition you can define areas in which you want to dynamically create the layout.

...and (A) and (B) of course can be used together, if it makes sense.

### “By Page”

The “by page” generation is quite powerful when it comes to structuring your system, because the result is a page that you embed via normal page name, but which internally is created dynamically. E.g. the page “/dyn/article” may render a page that is built up dynamically according to the article’s meta data - and the page “/dyn/customer” may render a page that is built up using the customer’s meta data.

### “Within a Page”

The “within a page” dynamic layout is also very powerful, because you can change “on the fly” parts of your page. The base of still is a static JSP page. But: in this page you define “exit points” which allow you to add the components dynamically.

There are two exit point types:

- (1) A special component ROWDYNAMICCONTENT provides the possibility that you dynamically add some XML layout definition at the given point. The XML is defined by a server side managed bean property.
- (2) Most components support an attribute COMPONENTBINDING. This attribute allows to you to directly get notified when the page is rendered during the JSF request processing and allows you to directly manipulate the JSF component tree.

Type (1) is very simple to program - you just need to build up an XML string which is then parsed and transferred into components by the ROWDYNAMICCONTENT component. - Type (2) give you direct access to the JSF component model and requires some more attention

during programming.

We clearly recommend to start with type (1) - but also read how to use type (2) so that you know about it.

---

## “By Page”

### Building your Dynamic Page Provider

The creation of dynamic layouts is quite simple:

You define a class that implements interface “IDynamicPageProvider”:

```
package org.eclnt.jsfserver.dynamicpages;

public interface IDynamicPageProvider
{
    String getPath();
    String readDynamicPage(String pageName);
}
```

Your implementation needs to implement two methods:

- “getPath()”: this method returns the page-name-path that is managed by this dynamic page provider. The path must be returned with leading slash, e.g. “/dynpath”. As result, all pages that are included with a page name starting with “/dynpath” will be managed by your implementation.
- “readDynamicPage(..)”: this method is called when a page actually is to be loaded at runtime. As parameter you receive the full page’s name - so you can deduct any type of information out of the page name, that you require for your dynamic layout creation.

Example:

```
package dynamicpages;

import org.eclnt.jsfserver.dynamicpages.IDynamicPageProvider;
import org.eclnt.util.dynaccess.IDynamicIntrospectionSupported;

public class DemoDynamicPageProvider
    implements IDynamicPageProvider
{
    public String getPath()
    {
        return "/demodyn";
    }

    public String readDynamicPage(String pageName)
    {
        return
            "<t:row>"+
            "  <t:label text='Hello world! Page name: '"+pageName+"' />"+
            "</t:row>"+
            """;
    }
}
```

This page provider is rather simple, but it shows the most important issues:

- The layout that you create does not need to be a full JSP page, but it's just the XML part.
- You do not have to care about the id-attributes of elements - if you do not explicitly specify then they are created dynamically.
- The layout must be “one row’ with contained content”. In case of returning several

rows you must build up a container that contains the rows:

```
NOT:
row
  label
  field
row
  label
  field

BUT:
row
  pane
  row
    label
    field
  row
    label
    field
```

## Registering your Page Provider

The registration of your page provider is done within the file “/eclntjsfserver/config/system.xml”:

```
<system>
  ...
  <dynamicpages>
    <dynamicpageprovider
      name="dynamicpages.DemoDynamicPageProvider"/>
  </dynamicpages>
  ...
</system>
```

## Pay Attention - Buffering!

Pay attention - the page provider is called in the same way as normally static “.jsp” pages are read from the file system. Once read, they are buffered.

The dynamic page provider is NOT called every time a dynamic page is resolved within a ROWINCLUDE, but is typically called one time per page name only!

---

## “Within a Page” - Integrating XML Layout Definition dynamically

### ROWDYNAMICCONTENT Component

The easiest way to embed dynamic layout into a page is the ROWDYNAMICCONTENT component. The component provides an attribute CONTENTBINDING which point to a server side property.

Within the server side property there are two ways of passing the dynamic layout:

- as XML string or
- as tree of component nodes

Example:

```
<t:row id="g_2">
  <t:label id="g_3" width="120" />
```

```

<t:label id="g_4" text="Change Content" width="120" />
<t:button id="g_5"
    actionListener="#{d.DemoRowDynamicContent.onFlip}" text="Flip" />
</t:row>
<t:rowdistance id="g_6" height="20" />
<t:rowline id="g_7" />
<t:rowdistance id="g_8" height="20" />

<t:rowdynamiccontent id="g_9"
contentbinding="#{d.DemoRowDynamicContent.content}" />

<t:row id="g_10">
<t:label id="g_11" width="120" />
<t:label id="g_12" text="Change Content" width="120" />
<t:button id="g_13"
    actionListener="#{d.DemoRowDynamicContent.onFlip}" text="Flip" />
</t:row>

```

In the layout there is a “dynamic row”, the CONTENTBINDING attribute points to a certain managed bean property.

The server side implementation is:

```

public class DemoRowDynamicContent
{
    ...

    protected ROWDYNAMICCONTENTBinding m_content = new
ROWDYNAMICCONTENTBinding();

    String m_firstName = "First";
    String m_lastName = "Last";

    boolean m_toggle = false;

    public DemoRowDynamicContent(IWorkpageDispatcher dispatcher)
    {
        super(dispatcher);
        renderDynamically();
    }

    public ROWDYNAMICCONTENTBinding getContent() { return m_content; }

    public void setFirstName(String value) { m_firstName = value; }
    public String getFirstName() { return m_firstName; }

    public void setLastName(String value) { m_lastName = value; }
    public String getLastName() { return m_lastName; }

    public void onOK(ActionEvent event)
    {
        m_firstName = m_firstName + " OK";
        m_lastName = m_lastName + " OK";
    }

    public void onFlip(ActionEvent event)
    {
        m_toggle = !m_toggle;
        renderDynamically();
    }

    private void renderDynamically()
    {
        if (m_toggle)
        {
            // dynamic content via object tree
            ComponentNode pane = new ComponentNode("t:pane");
            pane.addAttribute("rowdistance", "5");
            {
                ComponentNode row = new ComponentNode("t:row");
                pane.addSubNode(row);
                {
                    ComponentNode label = new ComponentNode("t:label");
                    row.addSubNode(label);
                    label.addAttribute("text", "First Name");
                    label.addAttribute("width", "120");
                }
            }
        }
    }
}

```



## Details on ROWDYNAMICCONTENTBinding

By the way: why is the layout definition encapsulated into a ROWDYNAMICCONTENTBinding object, and not directly passed as string? The reason is: with every JSF rendering the ROWDYNAMICCONTENT component checks if there has happened an update to the dynamic layout. If there is an update then the page content will be updated correspondingly.

For fast checking for updated the ROWDYNAMICCONTENTBinding object internally provides a counter that is updated every time new content is assigned into the component. As result only this counter is compared to check for layout updates, it's not the whole layout string that is checked for updates.

## Using “concrete Classes” for assembling ComponentNode Instances

In the example you saw that you can build up a tree structure, consisting of ComponentNode instances. Each instance holds its name (e.g. “t:label”), its attributes and its sub-instances.

Instead of using the generic ComponentNode class you may also use concrete classes: per component there is one class, e.g. there is a class “FIELDNode” for “t:field”. All attributes of the component are available through corresponding set-methods.

```
...
...
PANENode pane = new PANENode();
pane.setRowdistance("5");
{
    ROWNode row = new ROWNode();
    pane.addSubNode(row);
    {
        LABELNode label = new LABELNode();
        row.addSubNode(label);
        label.setText("First Name");
        label.setWidth("120");
        FIELDNode field = new FIELDNode();
        row.addSubNode(field);
        field.setText("#{d.DemoRowDynamicContent.firstName}");
        field.setWidth("120");
    }
}
...
...
```

The “concrete” classes are directly inheriting from class “ComponentNode” - so you can use the “concrete way” and the “generic way” in parallel.

## Details on managing Ids

You may have recognized that in the layout XML that was created dynamically in the example above there was no assignment of id-attributes. When defining a static layout then ids are mandatory attributes - though you will seldom recognized this, because they are generated internally.

The simple rule is: if you do not define ids on your own within your layout XML then the ROWDYNAMICCONTENT component will do this job for you. - But: if you define ids, then you need to pay attention at your own to not assigning the same id twice. Double assignment of ids will result in an ugly error “deep in the JSF functions”... Only assign ids, if you really have a good reason to.

What could be a good reason?

You need to be aware of that an id is the core parameter to indicate at client side that there is no change of the control structure. This means: the client, when executing its

delta management when receiving an XML layout from the server, checks components that already drawn against components coming through the new XML layout by their id. If the id matches, then the client component is continued to be used on client side, if the id does not match then the client component is removed and the new client component is drawn.

This means:

- Only call the `setContentXML()` method when needed, i.e. when you really build up a new XML layout. Do not call it e.g. with every request processing.
- When you create quite big layouts in a dynamic way, and when at same point of time only minimal parts of this layout really change then it makes sense to assign the ids by your own, so that the stable parts of what you generate do not always change their id.

## Which way to go: the XML-way - or the tree-of-nodes-way?

As you can see in the example above there are two ways of defining the dynamic layout:

- You either define a tree of component nodes, and pass them via the method `setContentNode(...)` or the method `setContentNodes(...)`.
- Or you define an XML String and pass it into the method `setContentXML(...)`.

In general the “objectified way” (tree of nodes) is the easier and faster one:

- You do not have to assemble proper XML at all. All the rules for formatting XML do not have to be applied.
- The dynamic layout is NOT packaged into a String, that is immediately parsed by a SAX parser so that an object view is re-built - but the layout is passed as object tree and directly transferred into a corresponding tree. There is NO String processing in the mid.

Sometimes, of course, the XML way is the better choice. Maybe you have some templates that you just update and directly place as dynamic layout into the page.

In this case you have to make sure that your XML is properly built, otherwise you will get some runtime exceptions. The assembling of an XML string by using a `StringBuffer` is a bit dangerous, sometimes. Imaging the following code:

```
StringBuffer sb = new StringBuffer();
...
...
sb.append("<t:label text=\"" + m_labelText + "\"/>");
...
...
```

In this case the label's text is dynamically set by a corresponding variable. But: the variable may hold “strange values”, at least from XML point of view. Imagine the value of the variable being 'abc"de', then the resulting XML will be...

```
<t:label text="abc"de"/>
```

...which is not a valid XML string anymore. The same happens when adding characters like “>”, “<”, “&”, etc.

Result: for properly assembling XML strings, the `StringBuffer` may not be the best way. Think about using `XMLWriter`-classes. Or, of course: build your layout using the tree-of-nodes-way...

## Automated Update of Expressions

When working with `ROWDYNAMICCONTENT` components then there are some aspects that

are automatically covered - the most important aspect: the execution of the CONTENTREPLACE-mechanism within the ROWINCLUDE or ROWPAGEBEANINCLUDE component.

Within CaptainCasa it is possible to arrange one page within another page - and by specifying a so called CONTENTREPLACE attribute to virtually update the expression of the included page. When in the page there is an expression like “#{d.PageUI.firstName}” then this expression may be updated, e.g. in the workplace management, to “#{d.d\_1.PageUI.firstName}”. This update of expressions is automatically done within the processing auf ROWDYNAMICCONTENT - you do not have to take care, and can just name your expressions in the same way as you name them within a static page.

This is a great advantage about ROWDYNAMICCONTENT - and is the reason why we generally strongly recommend the usage of ROWDYNAMICCONTENT for creating dynamic page layouts.

## ROWDYNAMICCONTENT Component <==> DYNAMICCONTENT Component

There is a counter part to the ROWDYNAMICCONTENT component: the DYNAMICCONTENT component.

The simple reason for having two components:

- The ROWDYNAMICCONTENT component always is embedded into a layout as a row. It automatically opens up a row so that its content is places into this row.
- The DYNAMICCONTENT component is working in the exact same way as the ROWDYNAMICCONTENT, but can embedded into a layout “everywhere”. Example: it can be embedded within a MENUBAR component (where the embedding of a row does not make sense).

The counter part bean property for the DYNAMICCONTENT component is the class DYNAMICCONTENTBinding, offering the exact same interface as ROWDYNAMICCONTENTBinding (actually DYNAMICCONTENTBinding it is an “empty extension” of ROWDYNAMICCONTENTBinding).

---

## “Within a Page” - Working with the Component Tree

The direct manipulation of the component tree should only be used when being familiar with JSF on the one hand and with being familiar with what's going on in the area of expression replacements (workplace Management, PageBean management).

You should definitely use by default the more abstracted way of using the ROWDYNAMICCONTENT component for all normal cases - and should have some special reason why you want to use the direct component tree update as described in this part of the documentation!

### Basics

On server side a page is kept as tree of components. This is JSF. The tree is built up from the JSP layout definition at the beginning, but allows further manipulation afterwards. During the process of transferring JSP into the component tree, so called tag handler classes are used. In the CaptainCasa control library there is one tag handler class for each tag, the name of the class is “<TAGNAME>ComponentTag”, e.g. “FIELDComponentTag” for the “t:field” tag.

Within the tree there are component instances. With CaptainCasa components these instances are of class “<TAGNAME>Component”, e.g. “FIELDComponent”. All component classes inherit from the class “BaseComponent”, which itself inherits from the JSF class

“UIComponentBase”.

What internally happens during “JSP to tree conversion”: a tag handler instance is created, this object creates/updates the component instance.

Why all these details? Because you will now see how you can do “on your own”, within the server side program what normally is done when transferring JSP into the component tree. You can - by using component tag instances - create component instances, assemble them to build a tree and plug them into an existing component tree. - It's like “automating the layout editor”.

## Attribute COMPONENTBINDING and how to use

Not all, but many components support the attribute COMPONENTBINDING. This attribute refers to a managed bean property. Normally properties are used to take data and transfer it to the client. With the property defined in COMPONENTBINDING it's vice versa: the framework pushes an object into the setter of the property when the page is created or processed.

All this sounds quite complex, but it isn't...! Let's take a look onto an example. In the following example the page is dynamically rendered dependent from the object structure that is currently selected:



The layout definition is quite short:

```
<t:rowbodypane id="g_3" >
  <t:row id="g_4" >
    <t:button id="g_5" actionListener="#{d.demoDynamic2.onEditPerson}" text="Edit
Person" />
    <t:button id="g_6" actionListener="#{d.demoDynamic2.onEditArticle}"
text="Edit Article" />
  </t:row>
  <t:rowdistance id="g_7" height="20" />
  <t:row id="g_8" >
    <t:pane id="g_9" componentbinding="#{d.demoDynamic2.pane}" rowdistance="2" />
  </t:row>
</t:rowbodypane>
```

Why is it short? Because the main part of the rendering of the page is not contained in the layout definition but is dynamically added. Have a look into the COMPONENTBINDING attribute within the PANE definition: this points to a managed bean property “#{d.demoDynamic2.pane}”.

During runtime this attribute is filled with the PANE component instance that is managed inside the server side component tree. The bean now can react and can dynamically add content.

```
public class DemoDynamic2 extends DemoBase
{
  // -----
  // inner classes
  // -----

  public class Person
  {
    String m_firstName;
    String m_lastName;
    String m_street;
  }
}
```

```

String m_town;
String m_country;
public void setFirstName(String value) { m_firstName = value; }
public String getFirstName() { return m_firstName; }
public void setLastName(String value) { m_lastName = value; }
public String getLastName() { return m_lastName; }
public void setStreet(String value) { m_street = value; }
public String getStreet() { return m_street; }
public void setTown(String value) { m_town = value; }
public String getTown() { return m_town; }
public void setCountry(String value) { m_country = value; }
public String getCountry() { return m_country; }
}

public class Article
{
String m_articleNo;
String m_text;
String m_description;
String m_unitOfMeasure;
public void setArticleNo(String value) { m_articleNo = value; }
public String getArticleNo() { return m_articleNo; }
public void setText(String value) { m_text = value; }
public String getText() { return m_text; }
public void setDescription(String value) { m_description = value; }
public String getDescription() { return m_description; }
public void setUnitOfMeasure(String value) { m_unitOfMeasure = value; }
public String getUnitOfMeasure() { return m_unitOfMeasure; }
}

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

Person m_person = new Person();
Article m_article = new Article();
Object m_object = m_person;

PANComponent m_pane; // component binding

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

public Object getO()
{
return m_object;
}

public void onEditPerson(ActionEvent ae)
{
m_object = m_person;
buildPage();
}

public void onEditArticle(ActionEvent ae)
{
m_object = m_article;
buildPage();
}

public void setPane(PANComponent pane)
{
if (m_pane == pane)
return;
m_pane = pane;
buildPage();
}

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

private void buildPage()
{
m_pane.getChildren().clear();
List<String> properties = readProperties();
for (int i=0; i<properties.size(); i++)

```

```

    {
        String property = properties.get(i);
        // row
        ROWComponentTag rt = new ROWComponentTag();
        BaseComponent r = rt.createBaseComponent();
        m_pane.getChildren().add(r);
        // label
        LABELComponentTag lt = new LABELComponentTag();
        lt.setText(property);
        lt.setWidth("120");
        r.getChildren().add(lt.createBaseComponent());
        // field
        FIELDComponentTag ft = new FIELDComponentTag();
        ft.setText("#{d.demoDynamic2.o."+property+"}");
        ft.setWidth("200");
        r.getChildren().add(ft.createBaseComponent());
    }
}

private List<String> readProperties()
{
    List<String> result = new ArrayList<String>();
    Class objectClass = m_object.getClass();
    Method[] methods = objectClass.getMethods();
    for (int i=0; i<methods.length; i++)
    {
        String methodName = methods[i].getName();
        if (methodName.equals("getClass"))
            continue;
        if (methodName.startsWith("get") &&
            methodName.length() > 3 &&
            methods[i].getParameterTypes().length == 0)
        {
            String propertyName = methodName.substring(3,4).toLowerCase() +
                methodName.substring(4);
            result.add(propertyName);
        }
    }
    return result;
}
}

```

What are the important parts of the code?

- The method “buildPage()”, that is called both in the “setPane()” method (that’s the first time) and then every time the object changed, is responsible for the server side rendering of components. In the “buildPage()” method for each property of the currently selected object a row, label and field is added to the page.
- Adding components to the page is done in the following way: a component tag is created and filled with all the attributes you require. From the tag, using the method “createBaseComponent()”, a component instance is created. The component instance is put into the component tree using the “getChildren()” method.

You see: via the COMPONENTBINDING the setPane() method is called, passing the PANE component from the original JSF component tree. Then you are responsible for managing what happens below the component.

## Points of Time, when the setter for COMPONENTBINDING is called

The setter for the component binding is set with **every** request, at the point of time, when the page is in its “encoding” phase. This is the last JSF phase during a JSF request cycle on server side.

This is important to know: you may have seen the following statement in the code above, which avoids a constant rebuilding of the tree:

```

public void setPane(PANEComponent pane)
{
    if (m_pane == pane)

```

```
        return;  
        m_pane = pane;  
        buildPage();  
    }
```

If there is no change to the component binding then there is no necessity to re-build the own components.

### **Pay Attention: setId(...)**

A component tag object provides the method setId() - that you can use for passing an identifier to the tag. The id will be taken over into the component later on.

The general advice is: do not use this setId() method unless there is a very specific reason to do so. The runtime system will create identifiers automatically. By letting the system create the identifiers you automatically avoid getting back errors from the Java Server Face level, telling you that duplicate ids were found within the component tree reflecting your page.

In some early examples, that were part of the demo workplace environment, the setId() method was explicitly called... - now you do not need to call it anymore!

### **Pay Attention: Workplace Management and Expressions**

When using the Workplace Management then you need to pay attention to properly defining expressions. Background: there is a runtime expression update with pages being called as workpage content. This expression update need to be taken into consideration when passing expressions as attribute values.

Please read the corresponding chapter within the part “Workplace Management” of this documentation.

### **One Comment for Eclipse Users**

When working with the Tag-Classes e.g. ROWComponentTag, as described above, then you may realize that the code completion does not properly work for all the methods that are provided.

In this case make sure that the following libraries are part of your project definition:

- (a) eclntjsfserver.jar
- (b) jsf-api.jar, servlet-api.jar and jsp.api.jar

All the libraries of (b) are available via the Tomcat installation that comes with CaptainCasa Enterprise Client - take a look into the Tomcat's “/lib” directory.

# Style Management

## Overview

CaptainCasa Enterprise Client comes with a style concept that provides the following functions:

- Separation of typical look and feel aspects from the individual component definition: if you for example want a background of a page to be shaded in a certain way then you do not have to repeat the corresponding BGPAINTE definition with each page, but you can define the shading centrally.
- Definition of several styles with different look and feel in order to support multiple styles for one and the same page. The style is later on assigned to the session of a user. As consequence you may define a “default style”, a “marketing style” and a “high contrast style” to serve different types of users.

In addition and in parallel to the management of styles there is the concept of using transparent color definitions, which is described in this chapter as well.

## JavaFX CSS Definitions

For JavaFX there are some extensions to the style management - so that it is easily possible to use JavaFX-CSS definitions in order to set up the base look and feel of components.

Whereas the base look and feel of components is “hardwired” in Swing (e.g. if a field has a certain rounding) it is driven by CSS-definitions in JavaFX. Consequence: the adaptation of the base look and feel of your components is much simpler.

## Style Definition Files

First talk about the so called “style definition files” - which are valid both for the Swing client and for the JavaFX client. Actually they are nothing else than some pre-setting of attributes, based on the server side attribute definitions of a component.

Style definition files are XML files that are kept in the /eclntjfsfserver/styles directory of your web application.

```
<web application>
  /...
  /eclntjfsfserver
    /styles
      /...
      /default
        style.xml
      /ccblue
        style.xml
      /...
  /...
```

For each style there is one subdirectory. Within the subdirectory there is a file with the fix name “style.xml”.

Open the style definition file of the “default” directory. It contains statements like:

```
<style>
  <tag name="checkbox" variant="default">
    <set attribute="foreground" value="#FFFFFF"/>
  </tag>
```

```

<tag name="combofield" variant="default">
  <set attribute="background" value="#FFFFFF"/>
</tag>

<tag name="field" variant="default">
  <set attribute="background" value="#FFFFFF"/>
</tag>

<tag name="foldablepane" variant="default">
  <set attribute="bgpaint"
value="rectangle(0,0,100%,30,#374966,#89a2bd,vertical)"/>
  <set attribute="foreground" value="#FFFFFF"/>
  <set attribute="innerbgpaint"
value="rectangle(0,0,100%,100%,#6c7f97,#7c8fa7,vertical)"/>
  <set attribute="innerpadding" value="5"/>
  <set attribute="padding" value="0"/>
  <set attribute="font" value="size:11;weight:bold"/>
  <set attribute="image"
value="/eclntjsfserver/images/foldablepanedefaulticon.png"/>
</tag>
  ...
</style>

```

What you see:

- For each (or at least certain) components style definitions are done.
- Each style definition consists out of a list of attributes and values. The names of the attributes correspond to the attributes that are provided by the component itself.
- Style definitions for a component are made together with a reference to a “variant”, which is “default” in all the definitions above.

The information is used for “pre-setting” the attribute values of a component. Imagine a component being first pre-set with the values above, before it gets applied the values you define inside your layout definition.

In your layout definition (.jsp file) in which you define the attributes of a component the following things happen:

- When you do not define an attribute, then automatically the pre-set attribute of the style definition is chosen.
- When you define an attribute, then your attribute definition overrides the one coming from the style definition.
- When you want to override a definition coming from the style to be “not set”, then you use the defined word “null!” as attribute value.

## Style Variants

You already saw the word “variant” in the file definition file above. In this file all variants were defined to be “default”.

With style variants you can setup different styles for one component, a “default” one and several other ones:

```

<style>
  ...
  ...
  <tag name="field" variant="default">
    <set attribute="background" value="#FFFFFF"/>
  </tag>

  <tag name="field" variant="highlighted">
    <set attribute="background" value="#FFC0C0"/>
    <set attribute="font" value="weight:bold"/>
  </tag>

```

```
</tag>
...
</style>
```

From a layout definition (.jsp file) a component references a variant using the attribute STYLEVARIANT:

```
...
<t:field ... /> <!-- no variant, default one is chosen -->
<t:field ... stylevariant="highlighted"/>
...
```

In the layout editor you directly see all existing style variants for a component:



In the “Style Variant” field you can directly assign a style for a selected component. The default style is “” (null).

We definitely encourage the usage of style variants! Style variants are a very strong mechanism to organize the look and feel of your application. With outsourcing most of your “special style settings” into style variants you are able to provide different look and feels for your application with just selecting a corresponding style definition, in which each variant is defined.

There's one important thing to note: style variant cannot be assigned dynamically (i.e. via expression). They only can be set statically.

## Defining Styles that extend other Styles

You may define a style definition file completely on your own, in which you list all the tags with all their variants - this is what is described above.

Or you may define a style definition file that extends an other style definition file. As consequence you take over all the information of the other file and only need to define the differences that you want to apply.

The format of the definition file is an extension of what you already know:

```
<style extends="default">
...
...
<tag name="label" extendsparenttag="true">
  <set attribute="font" value="weight:bold"/>
</tag>
<tag name="button">
  <set attribute="contentareafilled" value="false"/>
  <set attribute="font" value="weight:bold;size:14"/>
</tag>
...
...
</style>
```

By specifying “extends” on style-level you can define the parent style of your style definition. As consequence all tag definitions of the parent style are implicitly taken over.

You now define your own tag definitions just as normal. By default all your tag-attribute definitions are, if defined, overriding the complete tag definition from the parent style.

But you can also define “extendsparenttag” to be “true”: in this case all the definition within the corresponding tag of the parent style are mixed into what you define within your style.

The example above extends the default style in the following way:

- For the LABEL component in addition to the parent style definition the attribute FONT is set to the value defined.
- For the BUTTON component the parent style definition is completely overridden with what is defined within this style.

## Defining Tag Variants referring to other Tag Definitions

In a similar way you can define tag variants that take over the style attributes of other tag definitions, and themselves just add some extra definitions:

```
<style ...>
    ...
    ...
    <tag name="button">
        <set attribute="font" value="weight:bold;size:14"/>
    </tag>
    <tag name="button" variant="HIGHLIGHTED"
        extendstag="button" extendsvariant="default">
        <set attribute="contentareafilled" value="false"/>
        <set attribute="bgpaint" value="background(#FF000030)"/>
    </tag>
    ...
    ...
</style>
```

The “HIGHLIGHTED” variant of the example extends the normal button definition. (Remember: if not explicitly defining a variant then the variant “default” is assumed - so the first button definition is the “default” one).

Of course you can mix both extension variants: the style extension described in the previous chapter and the tag extension that you see here. When mixing, the tag extension will be executed first, the style extension is applied afterwards.

## Selecting a Style

The style is part of the session information that is managed with each http session at server side. You can access or set the style information in the following way:

```
public void onStyleCCBLUE(ActionEvent ae)
{
    SessionInfo.getSessionInstance().setStyle("ccblue");
}

public void onStyleDEFAULT(ActionEvent ae)
{
    SessionInfo.getSessionInstance().setStyle("default");
}
```

As any operation accessing the “SessionInfo” class, you need to be in the processing of a request on a server side in order to obtain a valid result.

Important: Note that the style definition is applied afterwards for all components which are built up. It is NOT changed for components that are already rendered. So, typically you should place the selection of a style for a user into a logon page, which comes as first page and then switches to the content pages, so that the whole page receives an update.

You may call the function...

```
HttpSessionAccess.reloadClient()
```

...in order to reload all the components that are currently displayed. In this case all components are rendered from the scratch and the style values are taken over as consequence.

## Expressions as Style Attribute Values

In principal you may also use expressions as attribute value definition. Example:

```
<style ...>
  ...
  <tag name="pane">
    <set attribute="background" value="#{xxx.yyy}"/>
  </tag>
  ...
</style>
```

Please note: like with the normal style management style values are applied when the page components are rendered on server side the first time. This means: if the value behind the expression changes, then this is only reflected by components which are new - not for existing ones.

Also use “HttpSessionAccess.reloadClient()” in order to re-create all components and as consequence to re-process all expressions.

## Default Style

You can set the default style for all sessions by maintaining the configuration file /eclntjfsfserver/config/sessiondefaults.xml.

```
<sessiondefaults
  ...
  style="default"
  ...
/>
```

## Selecting the Style by URL Parameter

You can select the style by adding URL parameter “ccstyle=<styleName>” to the .ccapplet or .ccwebstart-URL for starting a CaptainCasa Enterprise Client page.

```
http://localhost:50000/demos/workplace.workplace.ccapplet?ccstyle=abcstyle
```

The same can be done at any place where you reference the .jsp page via URL.

## Style Manager API

Please check the Java API documentation (Java Doc) within the style area. The central class “StyleManager” provides functions for accessing style information at runtime.



```

<t:row id="g_10">
  <t:textpane id="g_11" font="size:14" stylevariant="def_default"
    text="default textpane" />
</t:row>
<t:row id="g_12">
  <t:textarea id="g_13" background="#FFFFFF" font="size:14"
    stylevariant="def_default" text="default textarea" />
</t:row>
<t:row id="g_14">
  <t:button id="g_15"

bgpaint="roundedrectangle(0,0,100%,100%,40,40,#00000040,#00000010,vertical);roundedborder(0
,0,100%,100%,40,40,#C0C0C0,1)"
border="top:4;bottom:4;left:15;right:15" contentareafilled="false"
font="size:14" stylevariant="def_default" text="default button" />
  <t:button id="g_16"

bgpaint="roundedrectangle(0,0,100%,100%,40,40,#00000040,#00000010,vertical);roundedborder(0
,0,100%,100%,40,40,#C0C0C0,1)"
border="top:4;bottom:4;left:15;right:15" contentareafilled="false"
font="size:14;weight:bold" foreground="#400000"
stylevariant="def_highlight" text="highlight button" />
</t:row>
<t:row id="g_17">
  <t:tabbedpane id="g_18" background="#40000030" font="size:14"
    foreground="#600000" height="100" stylevariant="def_default"
    width="200">
    <t:tabbedpanetab id="g_19"
      bgpaint="rectangle(0,0,100%,100%,#00000000,#00000030,vertical)"
      stylevariant="def_default" text="tab" />
    <t:tabbedpanetab id="g_20" text="tab" />
  </t:tabbedpane>
</t:row>
</t:rowbodypane>
<t:rowstatusbar id="g_21" />
...
...

```

Every time a component is defined with a style variant “def\_\*” the attribute information behind the component are extracted and transferred into a style variant definition. The name of the variant in the style definition is derived from the “def\_\*” name. - Please note that the default variant explicitly needs to be defined (“def\_default”).

All components that do not point to a “def\_\*” style definition are treated just as normal.

The style definition that is generated out of the jsp file is:

```

<style>
  <tag name="t:rowtitlebar" variant="default">
    <set attribute="bgpaint"

value="rectangle(0,0,100%,100%,#40000080,#40000040,vertical);rectangle(0,0,100%,100%,#00000
020,#00000050,horizontal)" />
    <set attribute="font" value="size:16;weight:bold" />
    <set attribute="foreground" value="#FFFFFFC0" />
    <set attribute="padding" value="3" />
  </tag>
  <tag name="t:label" variant="default">
    <set attribute="font" value="size:14" />
  </tag>
  <tag name="t:label" variant="highlight">
    <set attribute="font" value="size:14;weight:bold" />
    <set attribute="foreground" value="#400000" />
  </tag>
  <tag name="t:field" variant="default">
    <set attribute="background" value="#FFFFFF" />
    <set attribute="font" value="size:14" />
  </tag>
  <tag name="t:field" variant="default">
    <set attribute="background" value="#FFFFFF" />
    <set attribute="font" value="size:14;weight:bold" />
    <set attribute="foreground" value="#400000" />
  </tag>
  <tag name="t:textpane" variant="default">
    <set attribute="font" value="size:14" />
  </tag>
  <tag name="t:textarea" variant="default">
    <set attribute="background" value="#FFFFFF" />
    <set attribute="font" value="size:14" />
  </tag>
  <tag name="t:button" variant="default">

```

```

        <set attribute="bgpaint"

value="roundedrectangle(0,0,100%,100%,40,40,#00000040,#00000010,vertical);roundedborder(0,0
,100%,100%,40,40,#C0C0C0,1)" />
        <set attribute="border" value="top:4;bottom:4;left:15;right:15" />
        <set attribute="contentareafilled" value="false" />
        <set attribute="font" value="size:14" />
    </tag>
    <tag name="t:button" variant="highlight">
        <set attribute="bgpaint"

value="roundedrectangle(0,0,100%,100%,40,40,#00000040,#00000010,vertical);roundedborder(0,0
,100%,100%,40,40,#C0C0C0,1)" />
        <set attribute="border" value="top:4;bottom:4;left:15;right:15" />
        <set attribute="contentareafilled" value="false" />
        <set attribute="font" value="size:14;weight:bold" />
        <set attribute="foreground" value="#400000" />
    </tag>
    <tag name="t:tabbedpane" variant="default">
        <set attribute="background" value="#40000030" />
        <set attribute="font" value="size:14" />
        <set attribute="foreground" value="#600000" />
    </tag>
    <tag name="t:tabbedpanetab" variant="default">
        <set attribute="bgpaint"
            value="rectangle(0,0,100%,100%,#00000000,#00000030,vertical)" />
    </tag>
</style>

```

By default not all attributes of a component definition within the jsp page are taken over: the attributes “id”, “text”, “width”, “height”, “value” are not take over.

You can explicitly define to take over all attribute values into the style definition (i.e. including e.g. width and height), by naming the style “def\_keeppall\_<variantName>”.

## Usage Detail: Extending the generated Style from another Style than “default”

By default, when saving a layout with the name “styledef\_XXX.jsp” a “XXX”-style is generated. The generated style is extended from the “default” style which is part of the CaptainCasa runtime.

If you want to extend the style from a dedicated other style then you need to name the layout definition in the following way “styledef\_XXX\_extends\_YYY” where “XXX” is the generated style and “YYY” is the style that you want to extend.

---

## JavaFX Client: Link to CSS Management

The XML base server-side presetting of attributes via style definitions as explained in the previous chapter is both valid for the Swing and the FX version of Enterprise Client.

### JavaFX Styling

JavaFX comes with some sophisticated styling concept that is far more flexible than the Swing Look&Feel approach: in JavaFX there are style sheet definitions, by which you can define the visual representation of a control on a fine granular level in a declarative way. The definition of style sheets is very similar to the definition of style sheets within the HTML environment.

There are two aspects of “.css” (cascading style sheets):

- “Styling”: you define style classes that hold the definition of how you want a certain component to look like. Each style class has a name - later on you assign the name to a concrete usage of the component in a page.
- “Cascading”: there are two areas of cascading:

- “file level”: you may define a sequence of style sheet definition files as environment to style your components. JavaFX will check for style definitions from the top most file down to the bottom file and assemble all style definitions accordingly: if it does not find a certain style in the top most style definition it continues with the next one etc.
- “class level”: when assigning a certain style class to a component you may do this by assigning a style class sequence. Example: you may define a style “mybutton” that holds the look and feel of your buttons. And you may define a style “myspecialbutton” in which you define just some extra definitions. In a certain page you define that the sequence “mybutton” and “myspecialbutton” should be applied to a certain button. JavaFX will automatically mix both styles accordingly.

There is a lot of documentation of JavaFX styles publically available, so we do not want to explain too much here - but better concentrate on how the JavaFX style management is applied within the CaptainCasa Enterprise Client.

## JavaFX-CSS Style Management within CaptainCasa Enterprise Client

The JavaFX style management is embedded into the normal CaptainCasa style management: when taking a look into the style directories within /eclntjfsfserver/styles then there is one “fx.css” file per style directory:

```

.../eclntjfsfserver
  /styles
    /default
      style.xml <== the CaptainCasa “server-side” style
      fx.css <== the JavaFX-style that is used for the components
      fx.xml <== additional config info for FX client
    ...
  /defaultfx
    style.xml <== the CaptainCasa “server-side” style
    fx.css <== the JavaFX-style that is used for the components
    fx.xml <== additional config info for FX client
    ...

```

...and there is one “fx.xml” file per style directory, holding additional information to control the visual appearance of the FX client. The “fx.xml” file is a collection of simple name-value definitions.

### Cascading Style Sheet Assignment

When the FX client is started with a certain style (e.g. “mystylefx”) then by default the corresponding “fx.css” is selected to be used by the client for styling the controls.

You may define a style steel sheet sequence by adding the following to fx.xml:

```

File: eclntjfsfserver/styles/mystylefx/fx.xml:

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ccStyleExtension>

  ...

  <styleValue>
    <name>styleSequence</name>
    <value>defaultfx</value>
  </styleValue>

  ...

</ccStyleExtension>

```

Now the definition tells the FX client to first load the style

“elcntjsfserver/styles/defaultfx/fx.css” before loading the current style sheet “eclntjsfserver/styles/mystylefx/fx.css”.

The value behind “styleSequence” contains a semicolon-separated sequence of style names that are loaded in the corresponding order before the own style s loaded.

If defining own styles (like “mystylefx”) then typically you define the style “defaultfx” to be the first style to be loaded within your style sequence. As result all the basic style definitions of the default CatpainCasa FX-styling are taken over - you only have to define additional style classes or modifications to the style within your style class.

## Style Class Assignment

If looking into a style sheet file (e.g. “eclntjsfserver/styles/defaultfx/fx.css”) then you see that there are various style classes that are applied to corresponding components.

Example:

```
/* -----  
   LABEL  
-----  
*/  
  
.cc_label_sp  
{  
    -fx-cursor: default;  

```

Typically you find for a certain component (here: LABEL) two style classes: one style class for the “surrounding pane” which forms the background for the component and one for the component itself. This is a reflection of the internal component structure that is internally used for all CaptainCasa components within the FX client:

- The surrounding pane (“xxx\_sp” stlye) represents the background. All BGPAIN drawing operations that you can apply to a component are executed on this background.

- The component itself is arranged on top of the background. That's the reason why all the components have a transparent background themselves - to let the background component shine through.

You also find an explicit “-disabled” style class definition for these components which can be enabled/disabled. The reason for this is, that when disabling a component you may not only want the component to change its background color (“from white to grey”) but you may in some cases change much more. Example: for a field component you may remove the border when being disabled - so that the field look like a label when being disabled.

Within the JSF page you assign a certain style class by using the attribute “fxstyleseq”:

```
<t:label ... fxstyleseq="cc_label" .../>
```

The style sequence definition both sets the corresponding style class for the foreground component (“cc\_label”) and it sets the corresponding style class for the background component (“cc\_label\_sp”).

### ***Cascading Style Class Assignment***

You may also pass a style class sequence as value:

```
<t:label ... fxstyleseq="cc_label;cc_labelspecial" .../>
```

In this case first the style classes “cc\_label/cc\_label\_sp” are assigned to the component, and then the style classes “cc\_labelspecial/cc\_labelspecial\_sp” are assigned on top.

### **Style Class Assignment via Style Variant**

Of course it is not the right strategy to individually assign the attribute “fxstyleseq” for every individual component tag definition within a JSP page. So you just use the normal style management for CaptainCasa serer side components in order to simplify the assignment.

Example: the style.xml for the defaultfx-style looks as follows:

```
<style ...>
  ...
  <tag name="autocomplete" variant="default">
    <set attribute="fxstyleseq" value="cc_autocomplete"/>
    <set attribute="rounding" value="3"/>
  </tag>
  <tag name="button" variant="default">
    <set attribute="fxstyleseq" value="cc_button"/>
    <set attribute="rounding" value="5"/>
  </tag>
  ...
  ...
</style>
```

### **Rounding**

For the styling of components for the FX Client the definition of the “rounding” attribute plays an important role as well. The value that you define is applied both to the background of the component (“background”-attribute) and to the border definition (“border”-attribute) and to the so called clipping of the component. (Clipping is the area which actually is occupied and drawn by the component.)

By defining the “rounding” you do not have to explicitly repeat the definition within the fx.css style definition.

## Additional FX Client Information in fx.xml

The “fx.xml” file within a style directory that was so far mentioned in the context of defining style sequences holds quite a lot of other definitions that are used by the FX client in order to control individual rendering operations.

(For Swing Client users: many of these definitions that are now inside the fx.xml file were passed into the Swing client by using client parameters.)

In the fx.xml definition for the “defaultfx” style (eclntjsfserver/styles/defaultfx/fx.xml) you find a list of all the parameters and some documentation about their usage.

## Defining own Styles

When defining a new, own style (e.g. “mystylefx”) then the typical way to go is:

- Create an own style directory “/eclntjsfserver/styles/mystylefx”.
- Create an own “style.xml” file that extends from “defaultfx” - but does not contain any own definitions (at the beginning):

```
<style extends="defaultfx" >
</style>
```

- Create an own “fx.css” which is empty at the beginning:

```
/* fx.css without content */
```

- Create an own “fx.xml” which only contains the definition of the style sheet sequence at the beginning:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ccStyleExtension>
  <stylevalue>
    <name>styleSequence</name>
    <value>defaultfx</value>
  </stylevalue>
</ccStyleExtension>
```

Now you have defined all the files - you will at the beginning not see any visual difference to the “defaultfx” style, because everything is inherited automatically.

When now starting to develop own definitions, please go on as careful as possible:

- Check within “defaultfx/fx.css” what definitions are available, which ones you want to change for what reason.
- Do not copy e.g. whole areas of “defaultfx/fx.css” into your own “mystylefx/fx.css” - but only use your own style sheet file for things that you really put on top of the defaultfx-style. Otherwise you will end up in some mess of definitions, where you do not know anymore if you just copied things for no reason or for good reason.
- Inside fx.css use your own prefix for own style definitions. Do not define new style classes starting with “cc\_”. (Of course you may override “cc\_”-style classes!)

## Style “default” <=> Style “defaultfx”

You may wonder, why there are two style definitions coming with CaptainCasa: “default” and “defaultfx”.

The reason is:

- Using the Swing client, there is no CSS-management. As consequence a lot of look and

feel definitions are part of the CaptainCasa “style.xml” files, e.g. containing quite complex BGPAINT operations to draw the component’s background.

- Using the FX client, there a CSS-management - containing the look and feel definitions for a certain component. As consequence the “style.xml” within the “defaultfx” style is quite thin.

When no explicit style is defined then the server will use “defaultfx” for FX clients and “default” for Swing clients.

---

## Swing Client: Setting Client Font Family

For setting the font that is used within the client it is possible to pass the name of the font family as client parameter. The parameter’s name is “fontfamily” - setting client parameters is described in Appendix “Starting Enterprise Client Pages - Passing Client Parameters”.

The valid names can be listed with the following Java program.

```
package org.ecInt.client.ztest;

import java.awt.Font;
import java.awt.GraphicsEnvironment;

public class TestFonts
{
    public static void main(String[] args)
    {
        String[] families = GraphicsEnvironment
            .getLocalGraphicsEnvironment()
            .getAvailableFontFamilyNames();
        for (String family: families)
            System.out.println(family);
    }
}
```

The result output in case of Windows OS is:

18thCentury  
Aharoni  
Alien Encounters  
Almonte Snow  
Andalus  
Angsana New  
AngsanaUPC  
Arabic Typesetting  
Arial  
Arial Black  
Arial monospaced for SAP  
Arial Narrow  
Arial Unicode MS  
Asimov  
Baby Kruffy  
Batang  
BatangChe  
Bitstream Vera Sans  
Bitstream Vera Sans Mono  
Bitstream Vera Serif

Blackadder ITC  
BN Jinx  
BN Machine  
Bobcat  
Book Antiqua  
Bookman Old Style  
Bradley Hand ITC  
Browallia New  
BrowalliaUPC  
Calibri  
Cambria  
Cambria Math  
Candara  
Candles  
Century  
Century Gothic  
Chinyen  
Comic Sans MS  
Consolas  
Constantia  
Copperplate Gothic Bold  
Copperplate Gothic

Light  
Corbel  
Cordia New  
CordiaUPC  
Courier New  
Cracked Johnnie  
Creepygirl  
Curlz MT  
DaunPenh  
David  
DejaVu Sans  
DejaVu Sans Condensed  
DejaVu Sans Light  
DejaVu Sans Mono  
DejaVu Serif  
DejaVu Serif Condensed  
DFKai-SB  
Dialog  
DialogInput  
Digifit  
DilleniaUPC  
Distant Galaxy

DokChampa  
 Dotum  
 DotumChe  
 Edwardian Script ITC  
 Engravers MT  
 Eras Demi ITC  
 Eras Light ITC  
 Estrangelo Edessa  
 Ethnocentric  
 EucrosiaUPC  
 Euphemia  
 Eurostile  
 FangSong  
 Felix Titling  
 Fingerpop  
 Flubber  
 Franklin Gothic Book  
 Franklin Gothic Demi  
 Franklin Gothic Demi Cond  
 Franklin Gothic Heavy  
 Franklin Gothic Medium  
 Franklin Gothic Medium Cond  
 FrankRuehl  
 FreesiaUPC  
 Freestyle Script  
 French Script MT  
 Garamond  
 Gautami  
 Gazzarelli  
 Georgia  
 Gisha  
 Good Times  
 Gulim  
 GulimChe  
 Gungsuh  
 GungsuhChe  
 Hand Me Down S (BRK)  
 Heavy Heap  
 Hollywood Hills  
 Impact  
 Induction  
 IrisUPC  
 Iskoola Pota  
 JasmineUPC  
 Jokerman  
 Juice ITC  
 KaiTi  
 Kalinga  
 Kartika  
 KodchiangUPC

Kristen ITC  
 Latha  
 Leelawadee  
 LetterOMatic!  
 Levenim MT  
 LilyUPC  
 LittleLordFontleroy  
 Lucida Bright  
 Lucida Console  
 Lucida Handwriting  
 Lucida Sans  
 Lucida Sans Typewriter  
 Lucida Sans Unicode  
 Mael  
 Maiandra GD  
 Malgun Gothic  
 Mangal  
 Marlett  
 Matisse ITC  
 Meiryo  
 Microsoft Himalaya  
 Microsoft JhengHei  
 Microsoft Sans Serif  
 Microsoft Uighur  
 Microsoft YaHei  
 Microsoft Yi Baiti  
 MingLiU  
 MingLiU-ExtB  
 MingLiU\_HKSCS  
 MingLiU\_HKSCS-ExtB  
 Miriam  
 Miriam Fixed  
 Mistral  
 Mongolian Baiti  
 Monospaced  
 Monotype Corsiva  
 MoolBoran  
 MS Gothic  
 MS Mincho  
 MS PGothic  
 MS PMincho  
 MS Reference 1  
 MS Reference 2  
 MS Reference Sans Serif  
 MS UI Gothic  
 MT Extra  
 MV Boli  
 Narkisim  
 Nasalization  
 Neon Lights  
 NSimSun

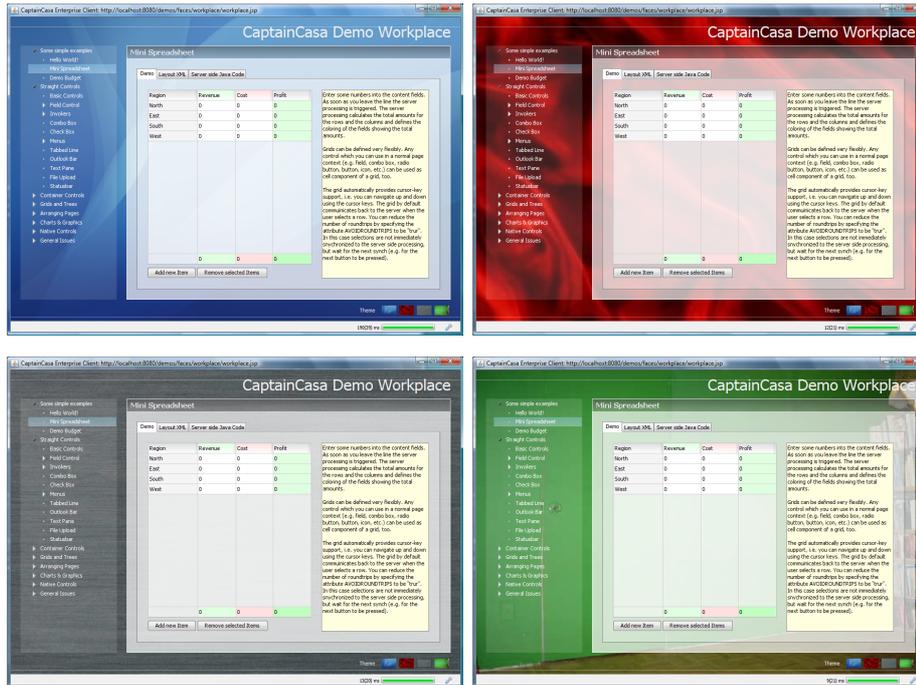
Nyala  
 OCR A Extended  
 OpenSymbol  
 Palatino Linotype  
 Papyrus  
 Parry Hotter  
 Perpetua  
 Plantagenet Cherokee  
 PMingLiU  
 PMingLiU-ExtB  
 PR Celtic Narrow  
 Pristina  
 Raavi  
 Rockwell  
 Rockwell Extra Bold  
 Rod  
 SansSerif  
 SAPDings  
 SAPIcons  
 Segoe Print  
 Segoe Script  
 Segoe UI  
 Serif  
 SF Movie Poster  
 Shruti  
 SimHei  
 Simplified Arabic  
 Simplified Arabic Fixed  
 SimSun  
 SimSun-ExtB  
 Snowdrift  
 Sylfaen  
 Symbol  
 Tahoma  
 Tempus Sans ITC  
 Terminator Two  
 Times New Roman  
 Traditional Arabic  
 Trebuchet MS  
 Tunga  
 Verdana  
 Vivaldi  
 Vrinda  
 webdings  
 wingdings  
 wingdings 2  
 wingdings 3  
 woodcut  
 X-Files  
 Year supply of fairy cakes

# Working with transparent Colors

The previous chapter explained that there is the possibility to define different style definitions in order to provide some default coloring for the components. This chapter tells you how to use transparent color definitions in order to effectively defining styles.

## Concept

The concept is simple: instead of defining concrete component colors you only define transparent component colors which put some shading on top of a central color definition. The concept is used within the demo workplace: by exchanging the background image the whole coloring of all components is updated:



Compare the coloring of the title bar component: because the title bar's color is just defined as some shading on top of the background, its look and feel automatically adapts to the background settings.

## Definition and Usage of transparent Colors

Transparent colors are defined using a color value which is formed in the following way:

```
#RRGGBBTT  
  
RR ==> red value  
GG ==> green value  
BB ==> blue value  
TT ==> transparency value, 00 == transparent, FF == covering
```

Transparent colors can be used at any place where colors are defined for a certain component:

- BACKGROUND attribute
- BGPAIN attribute

Of course you can use transparent colors in the style definition files as well. Example: the color definition of the default title bar looks the following way:

```
<tag name="rowtitlebar" variant="default">
```

```

<set attribute="border" value="top:0;left:0;right:0;bottom:0;color:#909090"/>
<set attribute="bgpaint"
value="rectangle(0,0,100%,100%,#00000060,#00000020,vertical)"/>
<set attribute="font" value="size:15;weight:bold"/>
<set attribute="foreground" value="#F0F0F0"/>
<set attribute="padding" value="3"/>
</tag>

```

The title bar's background painting is defined as a coloring from “black with transparency 60” to a “black with transparency 20” in vertical direction.

## Central Background Definition

Using transparent colors has the consequence that you need to define a central background definition for your pages. I.e. within your “out most” page definition, the one which includes all your other pages, you need to specify the background definition - typically using a BGPAIN or a BACKGROUND attribute definition.

Example: in the demo workplace the out most page is the “workplace.jsp” page:

```

<t:rowbodypane id="g_2" bgpaint="#{d.bgpaint}"
padding="top:10;bottom:10;left:20;right:20" >
    ...
    ...
</t:rowbodypane>
<t:rowstatusbar id="g_24" />

```

The BGPAIN definition is binding to a property “wp.bgpaint” - this means the background definition is not fix coded but comes from a property.

## Central Background Definition for Popup Dialogs

Popup dialogs are windows on their own, i.e. they do not automatically take over the main screen's background coloring. For this reason there is a Java API that you can call for defining the background painting of all popup dialogs:

```
BasePopup.initialize("rectangle(0,0,100%,100%,#F0F0F0,#D0D0D0,vertical)");
```

Using the API you can pass any BGPAIN parameter.

---

## Using the BGPAIN Attribute

The attribute BGPAIN is a very powerful one. It allows to define a sequence of drawing operations for a given component. There is a set of operations available that cover a lot of nice background features that you may apply.

The attribute value that you assign is a concatenation of operations, separated by semicolon “;”. The sequence of operations is later at runtime the sequence of execution. If the first operation draws a rectangle, the second writes a text, then the text will be output on top of the rectangle.

There's also a nice way of defining dimensions.

- Pixel definition. Example: “100”. As usual pixel definitions are scaled in the frontend; only with 100% scale factor one pixel really represents a pixel on the screen.
- Percentage definitions. Example: “50%”.
- “Percentage - Pixel” definitions: “100%-50”. Typical usage example: you may want to define a rectangle that covers the whole area, but with a padding of 10 pixels: “rectangle(10,10,100%-20,100%-20)”. - Please note: the minus operator “-” is the only

one that is supported.

Pay attention when defining and concatenating drawing operations. There are no blanks allowed between the operations and between the operations' attributes - everything needs to be concatenated in an absolutely condensed way!

### “mandatory” and “error”

- **mandatory()**: the component is drawn with an indicator that tells the user that data needs to be input. The indicator immediately disappears when the user starts with data input.
- **error()**: the component is drawn with an indicator that tells the user that the data contained in the component (e.g. field) does not contain valid data.

### “rectangle”

- **rectangle(x,y,width,height,color)**: plain rectangle
- **rectangle(x,y,width,height,color1,color2,horizontal/vertical)**: rectangle with color being transferred from color1 to color2 either horizontally or vertically.

User “rectangle” with transparent colors in order to lighten or darken certain areas. Example: “rectangle(0,0,100%,100%,#FFFFFF20)” will lighten up the whole component area, i.e. the background will shine through but will be covered with a light white coloring.

### “roundedrectangle”

- **roundedrectangle(x,y,width,height,radiusx,radiusy,color)**: rounded rectangle with plain color
- **roundedrectangle(x,y,width,height,radiusx,radiusy,color1,color2,horizontal/vertical)**: rounded rectangle with color changing from color1 to color2.

### “image”

- **image(x,y,imagePath,imageAnchor)**: the image is output at the x,y position. The imagePath is a definition pointing to an image within the web application. We recommend to always define images from the root of your web application, e.g. “/images/image1.png”.
- **image(x,y,width,height,imagePath,imageAnchor)**: the image is now scaled to a certain size.

The “imageAnchor” attribute defined what position of the image is used as anchor. The anchor positions is the one that exactly is used as x,y-position. Possible values are:

- “lefttop”, “leftmiddle”, “leftbottom”
- “centertop”, “centermiddle”, “centerbottom”
- “righttop”, “rightmiddle”, “rightbottom”

Examples:

- **image(0,0,100%,100%,/images/abc.png,lefttop)** - the image fills the whole area
- **image(50%,50%,/images/abc.png,centermiddle)** - the image is exactly position in the center of the area
- **image(100%,100%,100,100,/images/abc.png,rightbottom)** - the image is put into the

right bottom corner and is sized to 100\*100 pixels.

## “heximage”

This is the same as “image” but not the image is not transferred as imagePath but is directly transferred as hex-string definition:

- `heximage(x,y,imageHexString,imageAnchor)`
- `heximage(x,y,width,height,imageHexString,imageAnchor)`

The heximage-operation typically is used when the image is not stored in the file system but is derived e.g. from a database.

For generating an hexString within your server side coding use class “ValueManager”, method “encodeHexString(byte[])”.

## “scaledimage” and “scaledheximage”

When explicitly sizing an image/heximage then by default the sizing may change the width/height ration of the image. In case you want to avoid this you must use “scaledimage”/ “scaledheximage”:

- `scaledimage(x,y,width,height,imagePath,imageAnchor)`
- `scaledheximage(x,y,width,height,imageHexString,imageAnchor)`

## “write”

- `write(x,y,text,textAnchor)`
- `write(x,y,text,fontsize,fontcolor,textAnchor)`
- `write(x,y,text,fontsize,fontcolor,textAnchor)`
- `write(x,y,text,fontsize,fontcolor,fontweight,textAnchor)`
- `write(x,y,text,fontsize,fontcolor,fontweight,textAnchor,rotationInDegrees)`
- `write(x,y,text,fontsize,fontcolor,fontweight,textAnchor,rotationInDegrees,scaleX,scaleY)`

The operation outputs some text. The “textAnchor” definition is the same as with the image-operation.

The “fontWeight” either is “default” or “bold” or “italic”.

In case of specifying a rotation then the rotation needs to be defined in degrees, e.g. “180”. The rotation is done with the textAnchor-point as center.

If defining scaling then scaling is done as last operation, i.e. the rotation is executed first, then the scaling is done. The default scaling is “...,1,1)” this means all x and y definitions are output exactly as defined. In case you define “...,1,-1)” then the y coordinates are multiplied by -1, i.e. the text will be mirrored. In case you define “...,1,-0.5)” then the text will be mirrored and the output height of the text will be reduced to 50%.

You achieve nice effects when drawing one text in normal and in mirrored mode, with different coloring:



To do so you need to define two write-operations:

```
write(140,20,Some text,20,default,#FFFFFF80,lefttop);write(140,60,Some text,20,#FFFFFF20,default,lefttop,0,1,-1)
```

### “writeifempty”

This is a variant of the “write” command - the write is only executed if the value within the corresponding component is not set.

Example: you may want to write some text in the background of a FIELD component, that only appears if the user has not specified value yet. As soon as the user starts entering data, the text will disappear automatically.

### “oval”

- **oval(x,y,width,height,color)** - an oval is painted into the coordinates, plain coloring
- **oval(x,y,width,height,color1,color2,vertical/horizontal)** - oval with color changing from color1 to color2.

Example: “oval(50%-20,50%-20,40,40,#FF0000)” will draw a circle into the middle of the component’s area.

### “border” and “roundedborder”

- **border(x,y,width,height,color,thickness)** - rectangular border
- **roundedborder(x,y,width,height,radiusx,radiusy,color,thickness)** - rounded rectangular border

### “line”

- **line (x1,y1,x2,y2,thickness,color)** - a line is painted from the first coordinate to the second, with a certain color and a certain thickness (in pixels)

### “nodisabled”

- **nodisabled()**

Uuuuh, this is a special one. This command switches off the automatic darkening that is done when a component’s ENABLED attribute is defined to be false.

---

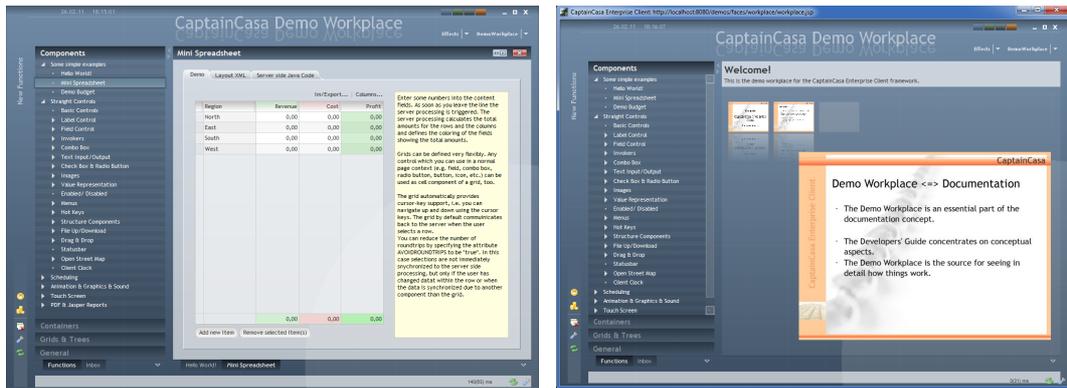
## “Flat” and “Undecorated” Style when using Webstart

By default an application client that is started via Webstart (“.jnlp”, “.ccwebstart”) is started as window on its own. The window itself is a normal window of the underlying operating system, i.e. its border and title bar is styled in the normal way.

### Client Parameters

By setting some client parameters you can start the client in a way, that suppressed the typical “heavy” window styling of the operating system:

- By setting parameter “flatmode” to “true” the window title bar and the window border will be replaced by a CaptainCasa variant, which looks much “thinner”.
- By setting parameter “undecorated” no titlebar and no border will be drawn at all.



Demo Workplace: started “undecorated=true, flatmode=true” vs. default

## SYSTEMICON Component

When starting the client in “undecorated” mode, then there are no icons drawn anymore, that the user can use in order to close the client - these are icons that are normally part of the title bar...

In order to allow the user to close the application, and to maximize/minimize the window, you can arrange SYSTEMICON components onto the top page. The SYSTEMICON has a SYSTEMFUNCTION attribute, by which you can select one of the functions:

- ...close the client
- ...minimize / maximize the client
- ...open the configuration dialog
- ...reload the current page

The SYSTEMICON instances with the functions “close”, “minimize” or “maximize” only show up, when the application is started by using Webstart. They are not shown when the applications is started as applet.

## ROWTITLEBAR and PANE Component: attribute ISWINDOWMOVER

Either with a ROWTITLEBAR or with a PANE component you can set the attribute ISWINDOWMOVER to “true”. As result any dragging&dropping on this component will be used for moving the corresponding window.

## Detail Configuration: some more Client Parameters

In order to “fine-tune” the look and feel of the window broder and the window title bar when using “flatmode=true”, there are some more client parameters:

- flatmodeborderframe - border definition for frames
- flatmodeborderdialogmodal / flatmodeborderdialogmodeless - border definitkions for dialogs
- flatmodetitlebgpaintframe - bgpaint definition for the title bar of frames
- flatmodetitlebgpaintdialogmodal / flatmodetitlebgpaintdialogmodeless - bgpaine definition for the title bar of dialogs.

# Internationalization Issues

The Enterprise Client allows to define user interfaces which are “fully internationalized”. This includes:

- Date format, Decimal format etc. are formatted and checked against country / languages specific rules and formats
- Literals can be translated in multiple languages. Multiple users can be logged on, each user seeing literals in his/her own language.
- Right to Left is supported.

---

## Client Locale <==> Server Locale Settings

Before getting into details first have a look onto which pieces of software are concerned when talking about internationalization:

- The client side is a Java program that by default takes over the language & country settings from the Java virtual machine on client side.
- The server side is Java program that by default takes over the language & country settings of the server side.

So, by default, both programs are decoupled from internationalization point of view. This makes sense in some scenarios and may be a bit confusing in other scenarios. Basically we took over this concept from normal browsers: also your normal HTML browser has a certain language (typically defined with installation) that has nothing to do with the language of the HTML content that is shown inside: you may start an English browser and view German pages. Print & configuration dialogs & default dialogs (e.g. calendar) will come up in the language of the browser, content dialogs of the application will come up in the language of the server side application.

This is important to always in keep in mind. - In the next chapters we will first introduce how to configure the client locale settings, then we will talk about the server side settings - and then talk about strategies in order to combine both.

---

## Internationalization Parameters for the Client

There are a couple of start parameters which are passed by the applet or web start processing, that define the client startup settings:

- orientation ==> left to right “ltr” or right to left “rtl”
- country ==> ISO code for country, e.g. “DE” for Germany, “US” for United States
- language ==> ISO code for language, e.g. “de” for German, “en” for English

The parameters are used to initialize the client's Java virtual machine. All system messages, log messages and default popups (such as file selection) depend on the locale definition of the client's virtual machine and are output correspondingly. When using the FORMATTEDFIELD input field then e.g. date and decimal values are formatted according to the locale definition.

When not defining any startup parameters then the default locale of the installed Java runtime environment is used.

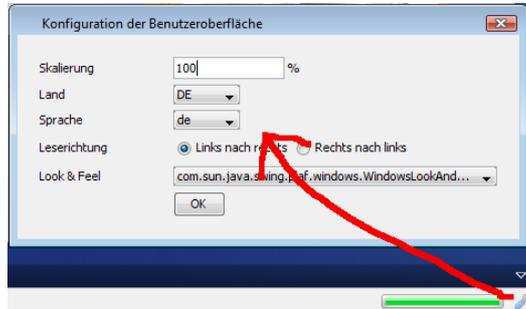
Please note:

- Do not mix the language definition of the virtual machine with the language definition

in which literals are brought to the user. Compare to the normal Internet browser: the browser has a language definition, the language of the pages is (or: may be) different.

## Updating the Parameters at Runtime

The user can open a client configuration dialog in which all of the parameters can be updated. The popup opens when clicking onto the tool icon on the bottom right:



## Http-request Parameters

The locale settings of the client are transferred to the server for information purpose with every http request. The http-header parameters are:

- “eclnt-country” for the country
- “eclnt-language” for the language
- “eclnt-orientation” for the orientation (value is either “ltr” for left to right or “rtl” for right to left)
- “eclnt-client” for the type of the client (value is either “applet”, “webstart” or “application”, dependent from how you started the client)

You can access the client parameters by using standard JSF APIs on server side, e.g. for pre-selecting a default language. There is a short-cut to the JSF APIs available: use the static methods of class “HttpSessionAccess”.

```
HttpSessionAccess.getCurrentRequest().getHeader(...);
```

## Orientation and Images

When using the orientation “right to left” then the whole layout flow is changed: while rows normally are layouted from the left to the right, they are now layouted from the right to the left.

When using images then you have to decide if a certain image should be “mirrored” or if the image is the same for both “left to right” and “right to left” orientation. The definition is done in the following way:

- Images are references through attributes of components. E.g. there is a `BUTTON-IMAGE` attribute or an `ICON-IMAGE` attribute. In this attribute you pass a URL of the image, the URL is relative to the current web application's location: e.g. “../images/xyz.png” or “/images/xyz.png”.
- If you append the String “&mirrorrtl” to the image's URL (e.g. “../images/xyz.png&mirrorrtl”) then the client will not load the image “../images/xyz.png” but it will load the image “../images/xyz\_rtl.png”.
- Result: if you want to mirror a certain image when the user selects “right to left” orientation then you have to indicate the mirroring by appending “&mirrorrtl” to the

image name, and you have to provide for a second image with the image name extension “\_rtl”.

## Orientation and Alignment Definitions

All alignment definitions that you do within your layout will be “mirrored” when starting the client in “right to left mode”. This means, “left” is automatically transferred into “right”, and vice versa.

You can also define alignments by using “left!” and “right!” (yes, with exclamation mark at the end!). In this case you define that the alignment is kept also when the layout is rendered in right to left direction.

## Scaling of Images

Well, scaling of images has nothing to do with internationalization, but the topic is mentioned in this chapter as well, because it also has to do with adding extra information to an image definition.

The user can set a so called size factor as start parameter or can set the scale factor by opening the client configuration popup.

The default behaviour is:

- When the user selects a size factor then all images are up-scaled accordingly. This ensures that the size relations between text and images are kept in synch.
- If you want to explicitly avoid the automated image scaling then you need to append “&noscale” to the file name of the image. E.g. by defining BUTTON-IMAGE to be “./xyz.png&noscale” the image “xyz.png” will always be drawn with the same size.

## Update the Client Settings from Server Side

There is a component CLIENTCONFIG that allows to update the client's language and country definition from server side at runtime. The CLIENTCONFIG component provides a couple of client parameters that are normally passed statically when starting the client - and allows to set these parameters at runtime by you server side processing.

Please note: the CLIENTCONFIG component should be placed at the beginning of the out-most page of your scenario. This means: when building up pages out of other pages using ROWINCLUDE or ROWPAGEBEANINCLUDE then the CLIENTCONFIG typically should be defined within the “outest page”.

And, please also note: when changing language and country settings via CLIENTCONFIG, then these changes only apply for components that are newly rendered. This typically makes sense for scenarios, in which there is a logon screen in front of the real application - during logon the CLIENTCONFIG is configured, all subsequent application pages are built up as new pages.

If you want to change the language/country settings of the client within the application already being active, then use the server side method “HttpSessionAccess.reloadClient()” in order to make the client refresh its whole content. As consequence all components will be built up once again on client side - as result, e.g. all calendar fields will get assigned the correct localization.

When setting language and country on client side through CLIENTCONFIG then automatically the user will not be able anymore to him/herself override these setting within the client configuration dialog.

---

# Language Management within your Application

## Setting Server Side Locale

The language settings of your pages are managed on top root level of your JSF component tree. You set the language definition in the following way:

```
public void onGerman(ActionEvent ae)
{
    FacesContext.getCurrentInstance().getViewRoot().setLocale(Locale.GERMANY);
}

public void onEnglish(ActionEvent ae)
{
    FacesContext.getCurrentInstance().getViewRoot().setLocale(Locale.ENGLISH);
}
```

Typically you set the Locale at the very beginning of your user's session, e.g. at logon point of time. You may use any Locale definition that is available in Java.

Different users may be logged on with different languages.

## Accessing Literal Translations - Basics

Within your server side application you may use various ways to support multiple languages for your literals.

For all the ways you need to define an expression for accessing the literal instead of hard-wiring the literal. Example: instead of defining the attribute LABEL-TEXT in the way “First Name”, you need to define an expression like “#{xyz.firstName}”.

- (1) You may use a default resource binding that is provided by the CaptainCasa Enterprise Client Framework. Using this resource you can reference different resource bundles that hold the literal translations.
- (2) You may use the “f:loadbundle” tag which comes with the default JSF tag library in order to access a resource bundle's property definitions. Please check the normal JSF documentation how to use resource bundles together with the “loadbundle” tag. When using “loadbundle” then you must add the loadbundle-tag within the outmost page of your application. I.e.: you cannot use it within a page that you include via the ROWINCLUDE tag.
- (3) You may use normal managed bean binding in order to access you literals. Note that there is a “session” binding of managed beans and a “request” binding of managed beans. In many cases the binding to language information can be set to “request” based, while the application itself may be defined as “session” based. Within the managed bean binding you may use the possibilities that are offered as part of the server side interpretation of expressions: you may use the binding Maps in order to build up a flexible way to address you literals. Find an example on this in the next chapter.

In general we recommend to use option (1): in this case the management of resources is directly integrated into the tool environment (Layout Editor) and information is kept in a standard way: as resource bundle.

Option (2) makes sense if you are used to native JSF usage, option (3) is the one to do everything on your own, e.g. you in case you keep your translation information in an text database.

## The default way: using built in Resource Management

CaptainCasa Enterprise Client provides a default way of managing multi language resources: literal information is outsourced in form of resource bundles. Resource bundles are text files that keep the information in the following way:

```
firstName=First Name
lastName=Last Name
```

Resource bundles are kept in several files, each one containing the “property=value” pairs for a certain language and/or country.

Example: there may be the files:

```
literals.properties
literals_en.properties
literals_de.properties
```

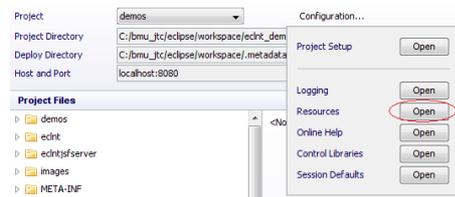
The file name is built out of the “resource bundle”, the language (optionally: the country) and the extension “.properties”. The files are located in a package of your compiled program. During runtime the multi language management will select the right file, which is closest to the session's internationalization settings.

Property files have some strange management of non-ASCII characters. All characters above ASCII code 127 need to be encoded. The German translation of the literal “street” is “Straße” - in the resource bundle file it is written in the following way:

```
street=Stra\u00DFe
```

Now, this is the technology behind, you will see that the default way of managing resources is quite simple:

Within the project's web application, inside the directory /eclntjfsserver/config there is the file “resources.xml”. Either edit the file from the file system or use the project configuration:

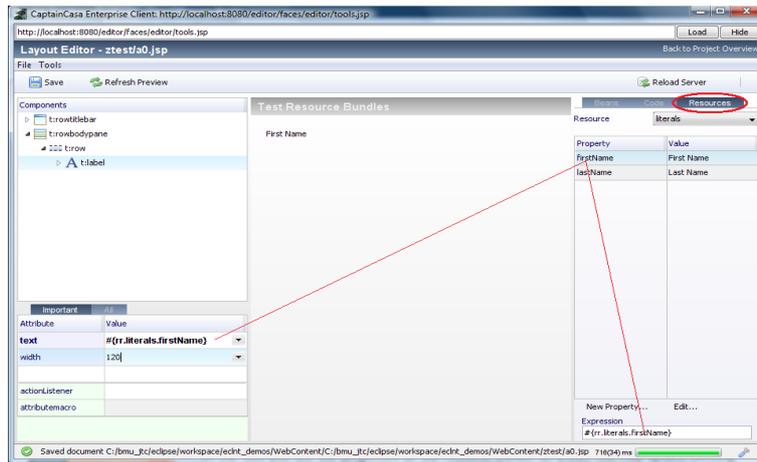


In the file you specify the resource bundles that you want to maintain within your project. Typically you have exactly one resource bundle for literals:

```
<resources>
  <resource name="literals" package="workplace.resources"/>
</resources>
```

For each resource bundle you define the “base name” and you define the package, in which it is stored.

From now on you can use the “Resource Editor” tool, which is part of the Layout Editor:



In the tool you see the properties of the resource bundle as a list. You may update existing properties or create new ones. Each property is associated with an expression, that can be used within component definitions. The expression is formed in the following way: “#{rr.resourceBundleName.propertyName}”.

The tool maintains the default property file. E.g. if the resource name is “literals”, then the tool will maintain the “literals.properties” file. This should be the one that you constantly use during development, and that you later on use as base for translations into other languages.

Translations are quite easily done: just copy the base file (e.g. “literals.properties”) into another file (e.g. “literals.properties\_de”) and then replace the property values. Use an editor which automatically translates non-ASCII characters into corresponding escape sequences, there are e.g. a plenty available as plugins for the Eclipse environment.

If at runtime a certain property name is resolved and no information is found within the corresponding resource bundle then a default value is returned. The default for this default value is a string that contains the name of the resource bundle and the name of the property that is resolved - so that you can directly see which value is missing in which resource bundle.

You can override this default behavior by assigning an explicit return value within the resource definition:

```
<resources>
  <resource name="literals" package="workplace.resources"
    defaultvalue="null"/>
</resources>
```

You can define either “null” or any other text value (e.g. “Literal missing”).

When using the resource management as described, then you can access literals within your server side Java code in the following way:

```
ResourceManagement.findLiteral("literals","firstName");
...accesses the same literals as accessed via:
#{rr.literals.firstName}
```

### Doing it on your own...!

This chapter shows how you may on your own use managed beans to access multi language information. You should read this chapter in case you want to implement an own resource management. This chapter again uses resource bundles to keep the literal translations, but the main focus is on showing how to arrange managed bean codings in order to access the literal translations.

First, there's the definition of resource files. Within the resource files there is the translation from a key into a string value. Resource files have a “base name” and have variation names depending on the language information they are defined for. (All this is default Java resource management, so please also have a look into the JavaDoc of the class “ResourceBundle”).

Example: there are two files, which are part of the classes / library-jar file:

*This is the content of the file “Literals.properties” within the package “org.ec1nt.jsfserver.i18n.resources”:*

```
OKPopup_ok = OK
YESNOPopup_yes = Yes
YESNOPopup_no = No
```

*And this is the content of “Literals\_de.properties” within the same package:*

```
OKPopup_ok = OK
YESNOPopup_yes = Ja
YESNOPopup_no = Nein
```

You see: two files with two different translations for the same keys.

The next step is to provide a managed bean that accesses the resource bundle. The managed bean claims to be a map, so that an expression is translated into a corresponding “get()” call against the map:

```
public class I18N
    implements Map<String,String>, Serializable
{
    protected static String s_bundle =
"org.ec1nt.jsfserver.i18n.resources.Literals";

    public void clear() {}
    public boolean containsValue(Object value) { return false; }
    public Set<Entry<String, String>> entrySet() { return null; }
    public boolean isEmpty() { return false; }
    public Set<String> keySet() { return null; }
    public String put(String key, String value) { return null; }
    public void putAll(Map<? extends String, ? extends String> m) { }
    public String remove(Object key) { return null; }
    public int size() { return 0; }
    public Collection<String> values() { return null; }

    public boolean containsKey(Object key)
    {
        String result = get(key);
        if (result == null)
            return false;
        else
            return true;
    }

    public String get(Object key)
    {
        ResourceBundle rb =
ResourceBundle.getBundle(s_bundle, FacesContext.getCurrentInstance().getViewRoot()
.getLocale());
        return rb.getString(key.toString());
    }
}
```

Only two methods are implemented of the map: the get() method and the containsKey() method. You see that the get() method is delegating the call to the resource bundle. The locale used for accessing the resource bundle is taken from the root node element (getViewRoot()) - this is “normal JSF”.

All the manipulation part of the map is not required, because the map serves only the

purpose of accessing literals.

The last step is to declare the class above as managed bean under a certain name. This is done in “faces-config.xml”:

```
<faces-config>
  ...
  ...
  <managed-bean>
    <managed-bean-name>ecInti18n</managed-bean-name>
    <managed-bean-class>org.ecInt.jsfserver.i18n.I18N</managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
  </managed-bean>
  ...
</faces-config>
```

That's it: you now can refer to a literal within a JSP definition in the following way:

```
<f:view>
<h:form>
<f:subview id="ecIntjsfserver_popups_yesnog_8">
<t:rowbodypane id="g_2" >
  <t:row id="g_3" >
    <t:textpane id="g_4" height="100%" text="#{ecIntdefscr.yesNoPopup.text}"
width="100%" />
  </t:row>
  <t:rowdistance id="g_5" height="10" />
  <t:row id="g_6" >
    <t:coldistance id="g_7" width="50%" />
    <t:button id="g_8" actionListener="#{ecIntdefscr.yesNoPopup.onYes}"
image="../images/yesnopopup_yes.png" text="#{ecInti18n.YESNOPopup_yes}" />
    <t:coldistance id="g_9" />
    <t:button id="g_10" actionListener="#{ecIntdefscr.yesNoPopup.onNo}"
image="../images/yesnopopup_no.png" text="#{ecInti18n.YESNOPopup_no}" />
    <t:coldistance id="g_11" width="50%" />
  </t:row>
</t:rowbodypane>
</f:subview>
</h:form>
</f:view>
```

Please note when transferring this example to your application:

- You need to write an own class as described before. Your own class may be a subclass of the I18N class that sets the s\_bundle member in the constructor to fit to your resource file names.
- You need to bind the bean under a different way than “ecInti18n”.

Maybe you want to add a method to also access the resource bundle from outside, so that you can access your literals from application processing, then you may add a method like:

```
/**
 * This method may only be called within a request processing.
 */
public static ResourceBundle getBundle()
{
    return ResourceBundle.
        getBundle(s_bundle,
FacesContext.getCurrentInstance().getViewRoot().getLocale());
}
```

Now you can also add your literals in order to for example output dynamic messages.

One last comment: pay attention that property resource files are NOT UTF-8 based. Unfortunately. You need to escape all characters with an internal integer value > 127. Best, you download a property file editor into your IDE for maintaining property files. There are plenty available for the Eclipse toolset, e.g. via [http://propedit.sourceforge.jp/index\\_en.html](http://propedit.sourceforge.jp/index_en.html).

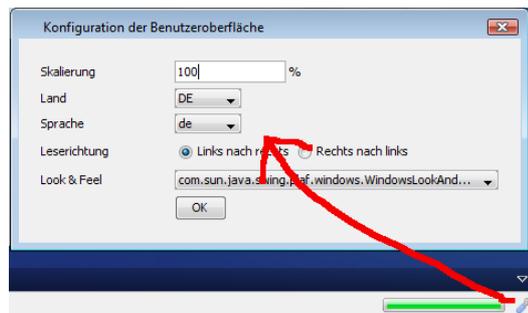
---

## Synchronizing Client Side Locale and Server Side Locale Settings

As introduced at the beginning of the chapter client locale settings and server locale settings in principal are independent from one another. But, there are ways to combine both.

### Scenario: Client Settings dominate Server Settings

In the client you can update the locale settings:



You already got to know that the current country/language settings are passed to the server side via http-header parameters (eclnt-language, eclnt-country).

Based on this, there is a special configuration on server side that will update (if required) the server side settings with every roundtrip. Edit the configuration file “webcontent/eclntjsfserver/config/sessiondefaults.xml” so that the attribute “takeoverclientlocalesettings” is set to “true”:

```
<sessiondefaults
  ...
  takeoverclientlocalesettings="true"
  ...
/>
```

### Scenario: Server Settings dominate Client Settings

Use the component CLIENTCONFIG in order to control the client locale settings from server side. If setting country & language through this component then the user will not be able anymore to update the client settings through the client configuration dialog - the combo boxes for selecting the language and the country will be read-only.

Typically the CLIENTCONFIG component is placed into the out-most page of your application so that it is contained only one time - in order to avoid confusion.

---

## Management of Dates (and Times)

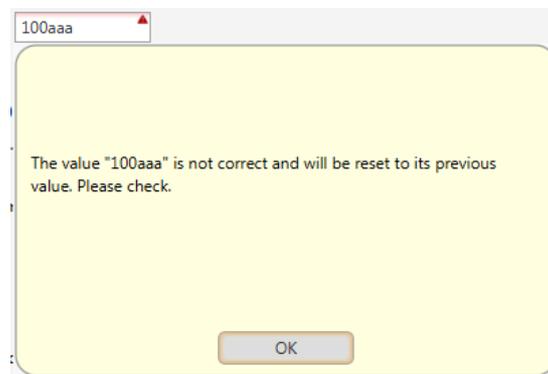
When working with the class “java.util.Date” in Java then you always need to pay attention to correctly managing the time zone aspect. Please read the Javadoc for details if being unsure about dealing with the classes “Date”, “Calendar”, “TimeZone”.

Every time a date is managed in a component (e.g. Calendar) then you need to also add a time zone reference. The date value that is passed “via the line” between client and server is the long-value of the date. A reliable date interpretation is only possible when ensuring that the time zone that is used in the client is exactly the time zone used on server side.

## Updating Client side Literals

...this chapter is a quite special one. And by default you can skip it - the client side literals are part of the CaptainCasa client and we hope that we are not too bad when finding literal texts for the few situations when the client talks to the user.

Example: when using the FORMATTEDFIELD component and when the user keys in some wrong value then the following message is shown:



This message is part of the client side FORMATTEDFIELD component - you do not have to explicitly specify the text it by your application.

The text is kept in property files on client side which are part of the eclnt.jar/ecIntfx.jar file that is loaded to the client side. Inside the jar file the corresponding files are kept in package org/ecInt/client/resources:



So, this chapter tells about what you have to do, if you explicitly want to update these messages. There are two ways to influence the client side literals - one by implementing a client side interface, one by providing a server side interface.

### Client side Interface

You may update the literals and replace them by own text definitions by implementing the client side interface "IClientLiteralsExtension":

```
public interface IClientLiteralsExtension
{
    public String getLit(String literal);
}
```

The class that you add to the client processing needs to be passed as client parameter "clientliteralsextensionclassname".

Every time the client needs some text for a certain literal id, it will first check the interface - and only will then use the property definitions if the interface returns null.

...of course implementing the client side interface means some effort on your side:

- You need to add your jar-file to the list of jar-files that is referenced in the HTML/JNLP definitions.
- You need to sign your jar-file and all existing client jar files with the same certificate.

This is the reason we in addition added a server side interface as well.

## Server side Interface

If the client is started with client parameter “loadclientliteralsfromserver=true” then the client will connect the server and request the literal texts from the server.

The things you have to do:

- Set client parameter “loadclientliteralsfromserver” to “true” when starting the client.
- Activate servlet “ClientLiteralServlet” in the web.xml:

```

    <servlet id="ClientLiteralServlet">
      <servlet-name>ClientLiteralServlet</servlet-name>
      <servlet-
class>org.eclnt.jsfserver.clientliteralloader.ClientLiteralServlet</servlet-
class>
      <load-on-startup>1</load-on-startup>
    </servlet>

...
...

    <servlet-mapping>
      <servlet-name>ClientLiteralServlet</servlet-name>
      <url-pattern>*.ccclientliterals</url-pattern>
    </servlet-mapping>

```

(The web.xml definition is part of web.xml\_template - you can copy from there.)

- Implement the server side interface “IClientLiteralServerLoader”:

```

package org.eclnt.jsfserver.clientliteralloader;

public interface IClientLiteralServerLoader
{
    List<LiteralInfo> loadLiteralInfos(String language, String country);
}

LiteralInfo is:
package org.eclnt.jsfserver.clientliteralloader;

public class LiteralInfo
{
    private String m_key;
    private String m_text;

    public LiteralInfo(String key, String text)
    {
        m_key = key;
        m_text = text;
    }

    public String getKey() { return m_key; }
    public String getText() { return m_text; }
}

```

- Set the class name of your implementation inside the eclntjsfserver/config/system.xml configuration file:

```

<system>
    ...
    ...

```

```
<clientliteralserverloader name="xyz.YourImplementation"/>
  ...
</system>
```

Every time the client needs some text for a certain literal id, it will load (and then buffer) the server side literals. If the server side definitions contain the corresponding literal id then this one is used - otherwise the default CaptainCasa texts are used.

# Online Help (“F1 Help”)

CaptainCasa contains a simple but flexible online help management. It provides the following functions:

- Context sensitive online help - when pressing F1 on one component a different help will show up than pressing F1 on an other component.
- Default management of online help texts - online help texts are stored by default within a defined directory of the web application.
- Internationalization - the selection of online help depends from the language a user is currently working in

In addition to the default online help management there is an interface to plug in any other online help management frameworks. As consequence you can store the online help information in a completely different way than the default, and you can visualize the online help to the user in a different way.

---

## Assignment of HELPIDs

The base of all is the assignment of so called “helpid” values to components. Most input components offer an HELPID attribute, into which you can pass a corresponding value:

```
<t:field id="g_6" helpid="firstName" width="200" .../>
```

The value that you assign is completely up to you. When using the default online help framework then the value must only contain characters that you can use for building proper file names.

Components with defined HELPID attribute automatically support the following behaviour:

- When the user presses F1 onto the component then the online help framework is invoked.
- Using the default framework a modal window will pop up showing a text which corresponds to the language and the helpid.

---

## The Default Framework

By default all online help texts are HTML texts that are stored within a certain directory. The name of the directory is specified in the configuration file “eclntjsfserver/config/onlinehelp.xml”:

```
<onlinehelp contentdirectory="/onlinehelp/">
</onlinehelp>
```

Within the directory that you define the following file structure needs to be defined:

```
webcontent/
  onlinehelp/
    de/
      firstName.html
      ...
    en/
      firstName.html
      ...
```

You see:

- For each language there is an own directory.

- Each helpid corresponds to an HTML text - which is the one to be shown in the window popping up when the user presses F1.

## Plugging in an own Framework

### Interface IOnlineHelpProcessor

This is the central interface that is called on server side when the user presses F1 on a component:

```
package org.eclnt.jsfserver.onlinehelp;

public interface IOnlineHelpProcessor
{
    public void processOnlineHelp(String helpId, String language);
}
```

The help-id that fits to the component and the language are passed as parameters - now it's the processor's task to present to the user an adequate online help.

The name of the class that is implementing the interface needs to be defined in the configuration file /eclntjsfserver/config/onlinehelp.xml:

```
<onlinehelp
    contentdirectory="/onlinehelp/"
    onlinehelpprocessor=
        "org.eclnt.jsfserver.onlinehelp.defaultimpl.OnlineHelpProcessorJBrowser">
</onlinehelp>
```

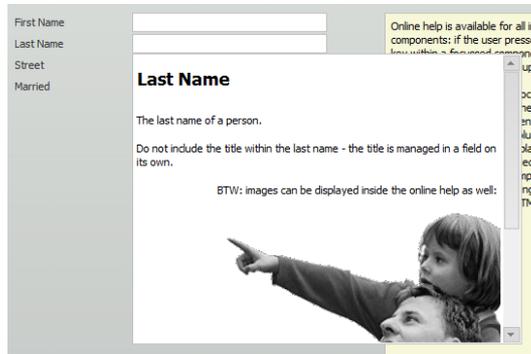
If not explicitly defined, then by default the implementation "OnlineHelpProcessorJBrowser" is selected.

### Default Implementation "OnlineHelpProcessorJBrowser"

The default implementation brings up a URL that is composed out of the directory for the online help ("contentdirectory"), the language and the help-id.

Example: if the "contentarea" is defined as "/onlinehelp/", and if the language is "en" and if the help-id is "firstName", then the URL would be: "/onlinehelp/en/firstName.html".

On client side a JBROWSER component is popped up in a modeless dialog that shows the URL directly aside the component that requested the online help:



Because things are quite simple, have a look into the implementation:

```
package org.eclnt.jsfserver.onlinehelp.defaultimpl;
```

```

...
...
/**
 * Online help processor that opens up a modal dialog in which a JBrowser
 * Component is loaded with a URL of the online help page.
 */
public class OnlineHelpProcessorJBrowser implements IOnlineHelpProcessor
{
    ModelessPopup m_popup;

    public void processOnlineHelp(String helpId, String language)
    {
        // calculate URL
        String url = OnlineHelpConfiguration.getContentDirectory();
        if (url == null)
        {
            throw new Error("Online help is not configured yet:
/ecIntjssfserver/onlinehelp.xml, content directory not defined");
        }
        if (!url.startsWith("/")) url = "/" + url;
        if (!url.endsWith("/")) url = url + "/";
        url = url + language + "/" + helpId + ".html";
        OnlineHelp.setOnlineHelpJBrowser(url);
        m_popup = ModelessPopup.createInstance();
        IModelessPopupListener mpl = new ModelessPopup.
            IModelessPopupListener()
        {
            public void reactOnPopupClosedByUser()
            {
                m_popup.close();
            }
        };
        m_popup.open("/ecIntjssfserver/popups/onlinehelpjbrowser.jsp",
            "Online Help",400,300, mpl);
        m_popup.setUndecorated(true);
        m_popup.setCloseonclickoutside(true);
        m_popup.setStartfromrootwindow(false);
        CLog.L.log(CLog.LL_INF,"Showing online help: " + url);
    }
}

```

The URL is assembled, and a modeless popup is called. The modeless popup itself is the one to hold the JBROWSER component.

Because the JBROWSER components is used for rendering the HTML text on client side, pay attention to the fact that quite complex HTML may not be rendered correctly. The JBROWSER component is sufficient for simple, static HTML (no JavaScript...) only.

## Old Default Implementation “OnlineHelpProcessor”

There is an old implementation as well, which used the class “OnlineHelpProcessor”, that itself implements “IOnlineHelpProcessor”. This implementation does not send a URL to the client but the full online text itself. The online text then is passed into a TEXTPANE component - the same one that is used inside JBROWSER component.

This implementation is even more restrictive when it comes to what you can do with HTML. E.g. it fails to interpret and META-header parameter within an HTML document. And the component has problems with resolving references (e.g. to images).

We recommend the usage of the “OnlineHelpProcessor” in cases, in which you want to send plain text or RTF-formatted text to the frontend. For cases, that are HTML-related we clearly recommend to use the default “OnlineHelpProcessorJBrowser”.

Inside the OnlineHelpProcessor implementation there is an additional interface in order to resolve the text for the online help:

- IOnlineHelpReader: in case you only want to influence the way, the online help HTML texts of the default framework are read, you can implement this interface. As

consequence online help texts will pop up in the normal way, but the text sources are not read from the file system but are read by your implementation. For example you may want to read the texts from an existing text database.

The interface is defined as follows:

```
package org.eclnt.jsfserver.onlinehelp;

public interface IOnlineHelpReader
{
    public class OnlineHelpText
    {
        String i_content;
        String i_type;
        public OnlineHelpText(String type, String content)
        {
            super();
            i_content = content;
            i_type = type;
        }
        public String getType() { return i_type; }
        public String getContent() { return i_content; }
    }

    public OnlineHelpText readOnlineHelpText(String helpId, String language);
}
```

The implementation has to return an OnlineHelpText-object which contains the following information:

- The content type of the text (“text/html”, “text/rtf” or “text/plain”)
- The text itself.

The text will be displayed within a popup dialog.

# Workplace Framework

## Basics

### Purpose

The user interface of typical applications consists out of a collection of individual functions. Each function is represented by some screen definition, the screen may of course itself subdivide into several other screens and may open various dialogs.

...so far you looked on Enterprise Client as technology to build such functions.

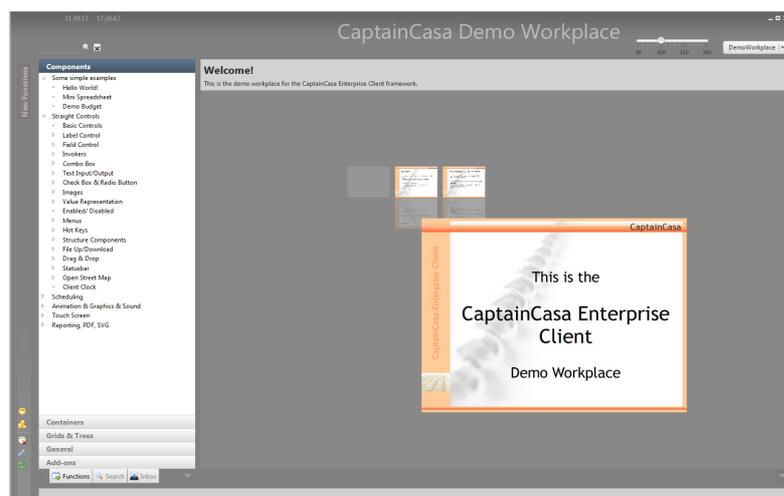
What you now need is some framework, to arrange these functions in order to form a workplace for an individual user. This includes the following aspects:

- The user must see the set of functions that he/she can start, e.g. represented by one or more function tree.
- Functions must be started in some kind of isolated container - both optically and from processing point of view. It must be possible to start the same function twice (e.g. to process two orders in parallel)
- Functions must have a defined a life-cycle: being started, being processed - and being closed.
- You may run multiple functions in parallel - e.g. you process an order, in parallel you create a customer record, in parallel you observe a stock list of a warehouse.
- You may want to arrange these started functions flexibly on your screen.

The workplace framework of Enterprise Client is exactly serving these requirements. It is an optional framework, so you do not have to use it! But it definitely makes sense to take a deep look into it before coding your own workplace framework.

### Example - The Demo Workplace

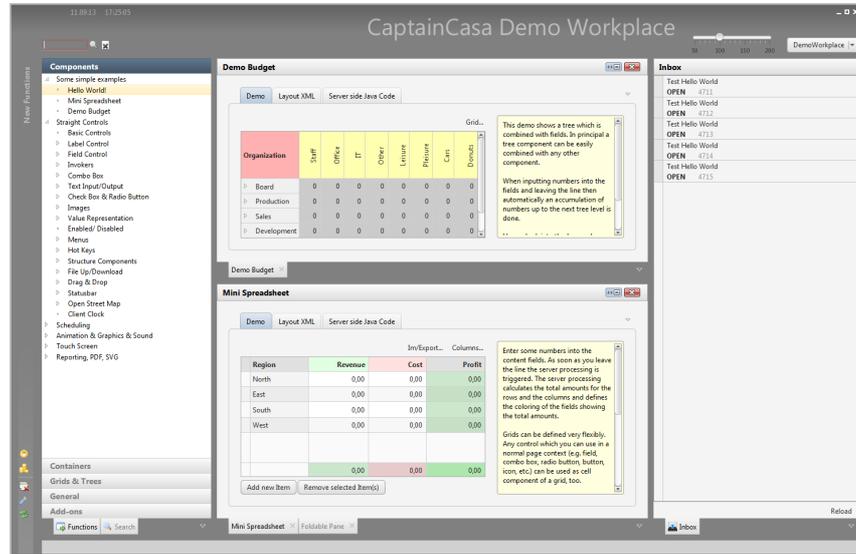
The demo workplace is started with two container areas, one on the left one on the right.



The user can start functions from several function trees, the functions are by default started within the right container area. For each function as “tab” is shown at the bottom

of the container area.

By dragging and dropping the user can re-arrange the started functions and open up new container areas:



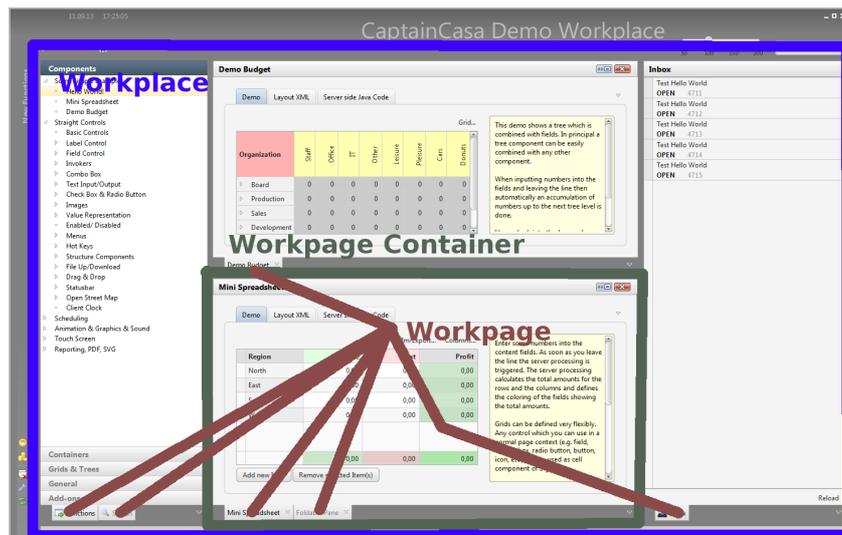
The user can isolate functions into own dialogs, so that they are running in a decoupled window (and e.g. can be moved on a different physical screen).

## Terms

When talking about the workplace, then certain terms will be frequently used:

- “Workplace” - this is the whole thing
- “Workpage” - this is the dialog of a started function
- “Workpage Container” - this is an area in which work pages are kept

CaptainCasa Enterprise Client



A workplace holds one or more workpage containers.

A workpage holds one or more workpages.

A workpage is a normal Enterprise Client page started within the context of the workplace.

## Development Effort

Basically there is no effort at all to start all pages that you have developed within the context of a workplace. It just works...

But: you typically want to make use of the workplace APIs. Examples:

- You may want to get notified when the user wants to close a workpage. You may want to not allow the user to close a page (e.g. because unsaved data is contained).
- You may want to start workpages from your current one: e.g. from a list of orders you want to start the order detail page as a new workpage, so that it runs decoupled from the list.

In these situations you want to talk to the workplace directly.

---

## Creating your first Workplace

...all the information that you need in order to build your first workplace was outsourced into a separate tutorial “Building a workplace”.

We strongly recommend to process this tutorial so that you have some first, working workplace in which you can embed functions and in which you can test if you access the API in the right way.

---

## Dispatcher Concepts

This chapter is the toughest one. It explains how the workplace internally works. On the one hand you do not need too much about these internal things, because at the end you do not see them during development.

On the other hand they are essential for some deep understanding, especially when it comes to understanding how the objects are separated from one another on server side.

### The Dispatcher so far...

You got to know the Dispatcher class/object so far by being a factory object that is “always” written in front of the bean class that you address from a page.

Example: your page “order.jsp” binds a certain field via expression “#{d.OrderUI.orderNumner}” to the server side processing.

We already explained that the dispatcher is some kind of Map-instance, having a certain strategy for resolving names - if they are not contained in the map already. So if the map is asked for “OrderUI” and does not know a corresponding object yet, then the map is creating an OrderUI-object and adds it to its data.

### The Dispatcher in the Workplace Context

Now comes the workplace...:

The workplace uses a dispatcher “WorkpageDispatcher” which is an enriched version of the normal dispatcher. It not only can resolve class names into objects, but it also can manage sub-dispatchers.

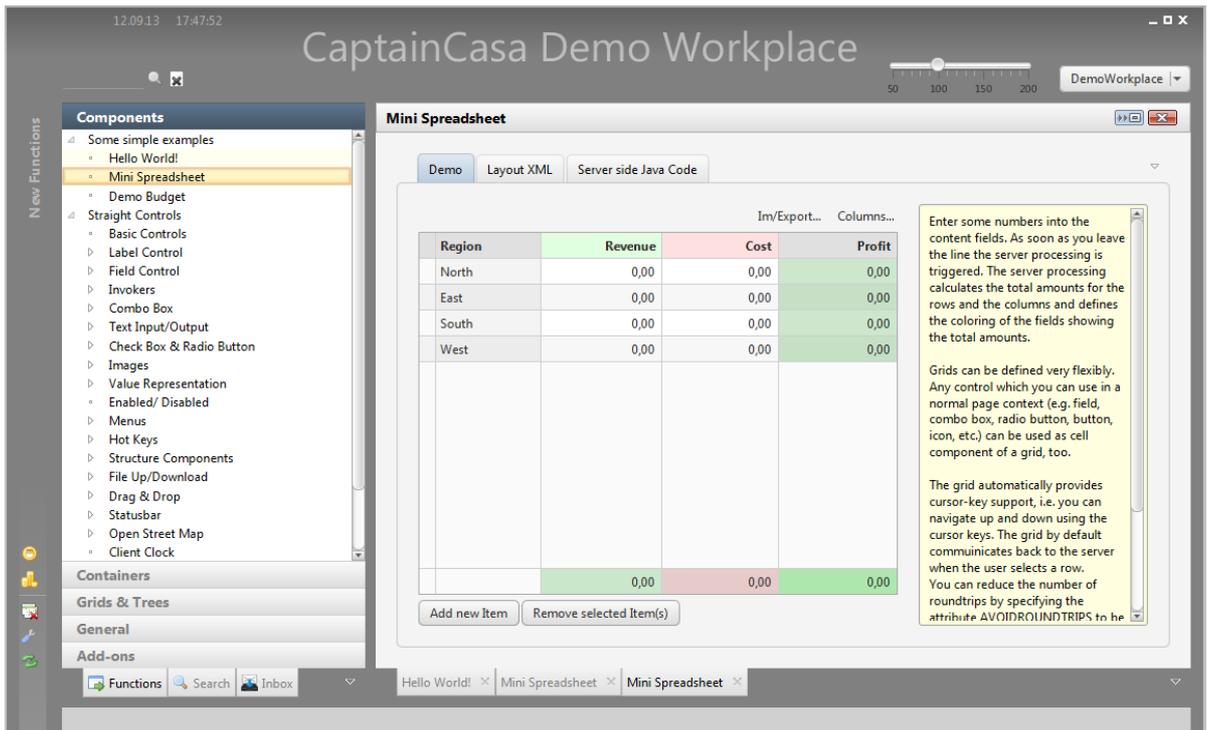
This sounds complex but indeed isn't. If a name to be resolved starts with “d\_” then the dispatcher does not resolve this name as usual, but creates a new “WorkpageDispatcher”

instance and manages this instance within its map. This instance is referred to as sub dispatcher in the following text.

When creating the instance of the sub dispatcher, then the same dispatcher-class is used than the one creating the dispatcher. E.g. if you derive your own dispatcher from “WorkpageDispatcher”, then the same class will be used for building the sub-dispatchers addresses via “d\_”-names.

You may already see: the dispatcher - which actually is a factory - is now able to manage sub dispatchers, each one being a factory on its own. The workplace now is responsible for building one sub-dispatcher for each workpage.

Example: in the following workplace 6 workpages are started:



The workpages are: “Functions”, “Search”, “Inbox”, “Hello World”, “Mini Spreadsheet”, “Mini Spreadsheet”.

How is this represented within the objects on the server side? - The object are structured the following way:

```

d                                     <== the root dispatcher
.d_1                                 <== subdispatcher instance
  .FunctionsUI                       <== functions bean
.d_2                                 <== subdispatcher instance
  .SearchUI                          <== search bean
.d_3                                 <== subdispatcher instance
  .InboxUI                            <== inbox bean
.d_4                                 <== subdispatcher instance
  .HelloWorldUI                      <== hello world bean
.d_5                                 <== subdispatcher instance
  .SpreadSheetUI                     <== spreadsheet bean
.d_6                                 <== subdispatcher instance
  .SpreadSheetUI                     <== spreadsheet bean
  
```

The root dispatcher is holding 6 subdispatchers. In each subdispatcher the corresponding page bean is referenced.

Each workpage is internally associated with one subdispatcher, so that the objects between workpages are never shared. Especially this is important when looking into the

“Mini Spreadsheet”, which is started twice in the workplace: there is one instance per workpage. As consequence there never is some conflict between the user processing the first and the second instance.

## Addressing the correct Dispatcher

The workplace is showing workpages in dedicated areas - called workpage containers. Each workpage is bound to a screen definition - the screen that is started with the workpage. Basically the workplace management does nothing else then managing which workpage is currently shown in which workpage container.

In the example above one of the “Mini Spreadsheet” instances is just shown to the user on the right side of the workplace. The workplace loads the corresponding page and while loading tells the page to update all contained expressions from “#{d.\*}” to “#{d.d\_5.\*}”. As consequence the expressions in the page that originally are directly pointing to the root dispatcher and then to the bean are updated “on the fly” - and now are pointing to the correct subdispatcher.

This is done completely automatically.

## Accessing the Workplace Environment from your Bean

In case that you are interested to get to know the dispatcher that is responsible for managing your bean, you need to inherit your bean class from the following base classes:

- WorkpageDispatchedPageBean
- (or WorkpageDispatchedBean when not using page beans)

Both provide constructors into which the “owning” dispatcher of the bean is passed as parameter.

```
public MyXXXXBean extends workpageDispatchedPageBean
{
    public MyXXXXBean(IWorkpageDispatcher dispatcher)
    {
        super(dispatcher);
        ...
    }

    private void yyyy()
    {
        ...
        getOwningDispatcher();
        ...
        getWorkpage();
        ...
    }
}
```

As shown in the code above, you at any point of the class can access the core objects of the workplace environment:

- getOwningDispatcher() - the “owning” dispatcher, returning a class of interface “IWorkpageDispatcher”
- getWorkpage() - the workpage that the bean is started in, returning a class of “IWorkpage”
- getWorkpageContainer() - the workpge container the page currently is kept inside, returning a class of “IWorkpageContainer”

---

## The Workplace API

The three core interfaces of the workplace management are accessible from the page bean that is started within the workplace context as shown in the previous chapter. For detailed information of each method of the interfaces please check the JavaDoc documentation.

### IWorkpageContainer

```
public interface IWorkpageContainer
{
    public void addworkpage(IWorkpage workpage);
    public void switchToWorkpage(IWorkpage wp);
    public IWorkpage getWorkpageForId(String workpageId);
    public void closeworkpage(IWorkpage workpage);
    ...
    ...
}
```

This interface gives access to the workpage container, managing workpages.

When starting a workpage then part of the workpage information (IWorkpage) is always an id. The id is a string, that is built up by yourself - so you can use any semantics that is suitable to you. You use the id to identify a certain page in the workpage container.

Example: when you start a workpage showing an article 4711, then your id might be "ARTICLE\_4711". Maybe the article 4711 already is displayed as workpage within your workplace? So you do not want to show the page twice, but better want to switch to the workpage already displaying the article?

In this case you first check with "getWorkpageForId("ARTICLE\_4711")" if there is already a workpage. If so you use "switchToWorkpage(...)" to bring it to the front, if not you create a new workpage. The creation is done on lower level by passing an instance of IWorkpage, using the default implementation "Workpage".

Or you use a helper interface "IWorkpageStarter":

```
public interface IWorkpageStarter
    extends Serializable
{
    public IWorkpage startworkpage(IWorkpageDispatcher workpageDispatcher,
                                   IWorkpageContainer workpageContainer,
                                   workpageStartInfo startInfo);
}
```

You obtain an instance of IWorkpageStarter by calling WorkpageStarterFactory.getInstance().

With IWorkpageStarter you pass an instance of "WorkpageStartInfo", which is a class that is used everywhere in the workplace context, where you specify information about a page to be started. You may remember the tutorial, when defining e.g. the function tree or the default perspective: there you defined WorkpageStartInfo instances using the JAX-B-XML representation, now you do it in the interface "by bean".

### WorkpageStartInfo

The definition that is required for at runtime starting a page is kept in an object of type "WorkpageStartInfo". Once started, an instance of "IWorkpage" is built and registered within the workplace. So "WorkpageStartInfo" holds the information about how to start a page, whereas "IWorkpage" holds the information about an actual instance of a started page.

## Core Functions

WorkpageStartInfo contains the following information:

- Which is the page to be opened for the function? - The workplace may switch from one function to another and as consequence has to update a certain graphical ROWINCLUDE area...  
Please note: there are two ways of defining the page to be opened: the “page-way” and the “page bean way”: you either directly pass the name of the page that you want to open via the method “setJspPage(…)” - or you pass the name of the page bean instance via the method “setPageBeanName(…)”.  
Please note that the name of the page bean instance is NOT its class name, but is the name under which the page beans is managed within the dispatcher. E.g. if the page beans is managed within the dispatcher via “#{d.AbcUI}” then the name is “AbcUI”.
- Which is the title of the function? - The workplace shows all opened functions in a task bar - and it renders a title bar on top of the content area. Both require a textual information.
- Which is the (language independent id) of the function? - Maybe you want to ask the workplace if a function is already started in order to avoid a double-starting. Each function that is started is associated with an id, so that you can check at runtime.
- Are there certain start parameters to pass to the function, so that it receives some additional information?

There is a class “WorkpageStartInfo” and an interface “IWorkpageStartInfo” which collects all this information:

```
public interface IWorkpageStartInfo
{
    public void setJspPage(String value);
    public void setPageBeanName(String value);
    public void setImage(String value);
    public void setText(String value);
    public void setDecorated(boolean decorated);
    public void setOpenMultipleInstances(boolean openMultipleInstances);
    public void setId(String value);
    public void setParam(String paramName, String paramValue);
    public void removeParam(String paramName);
    ...
    ...
}
```

## Special Usage - Starting Functions

Normally you define within the WorkPageStartInfo-instances which page to start.

You in addition can define that instead of starting a page a certain internal function is invoked. In this function you can “do what you want”.

You do so by passing a class name via the “setFunctionClassName(…)” method of WorkpageStartInfo. A new instance of the class will be created when the user starts the corresponding item. The instance is expected to support interface “IWorkpageFunctionExecute”. Example:

```
public class DemoworkpageFunction implements IWorkpageFunctionExecute
{
    public IWorkpage executeFunction(IWorkpageDispatcher rootDispatcher,
                                    IworkpageStartInfo workpageStartInfo)
    {
        ...
        ...
        return null;
    }
}
```

```
}
```

## IWorkpage / Workpage

When starting a workpage (e.g. by double clicking into the function tree), then a corresponding object is created within the workplace framework. The object supports interface “IWorkpage” - the default implementation is class “Workpage”.

The interface IWorkpage is:

```
public interface Iworkpage
{
    public String getJspPage();
    public String getTitle();
    public String getIconURL();
    public String getId();
    public boolean isDecorated();

    public void setWorkpageContainer(IworkpageContainer container);
    public IworkpageContainer getWorkpageContainer();

    public IworkpageDispatcher getDispatcher();
    public String getParameter(String name);

    ...
}
```

The work page is associated with certain information that the workplace needs in order to draw a titlebar for the work page and in order to list the page in the task bar:

- the JSP page (“/order.jsp”)
- the title (“Order 4711”)
- an icon url (“/images/order.png”)
- and id that differentiates the content of this page from other pages (“order/4711”)

### Start Parameters

There is also the possibility to define start parameters when defining a workpage: the start parameters are simple String-objects (for good reason we do not allow complex objects to be passed...).

The typical procedure of passing start parameters is:

- The start parameters are defined in the WorkpageStartInfo using the setParam(..) method. You may pass any number of parameters.
- At runtime when the page really is started (e.g. user double clicks function tree node) then a IWorkpage instance is created, taking over all start parameters from WorkpageStartInfo.

The application can take over the parameters by accessing the workpage using method “getWorkpage()”, reading the parameters and preparing its data content accordingly.

Example: in the function tree you add the function to show article with id “4711”. As consequence you create a WorkpageStartInfo instance (by bean, by XML) in which you define parameter “ARTICLEID” to be “4711”. The code to open the page in the workplace may look as follows:

```
...UI bean of the starting bean, e.g. list of articles...:

    ...
    workpageStartInfo wpsi = new workpageStartInfo();
```

```

wpsi.setId("ARTICEL_4711");
wpsi.setPageBeanName("ArticleDisplayUI");
wpsi.setText("Article Display 4711");
wpsi.setParam("ARTICLEID", "4711");
...
IWorkpageStarter wps = WorkpageStarterFactory.getWorkpageStarter();
wps.startWorkpage(getOwningDispatcher(),
                 getWorkpageContainer(),
                 wpsi);
...

```

The workpage is started, the ArticleDisplayUI instance is created - so in the constructor you read the start parameters in the following way:

```

public class AritcleDisplayUI extends workpageDispatchedPageBean
{
    public ArticleDisplayUI(IWorkpageDispatcher dispatcher)
    {
        super(dispatcher);
        String articleId = getWorkpage().getParam("ARTICLE");
        initialize(articleId);
    }

    private void initialize(String articleId)
    {
        ...
        ...
    }
}

```

## Workpage Life Cycle Aspects - Closing of a Workpage

When a workpage is started (e.g. double click in the function tree) then a corresponding "IWorkpage" object is created, itself being linked to a corresponding dispatcher object that manages the UI beans for the workpage's context.

This is the starting procedure. But: the workplace management also manages the ending of a workpage: e.g. the user presses the "top right close button" of the workpage. The sequence of operations is as follows:

- The workplace management tells the request to close to the workpage, by calling its "close" method.
- The workpage delegates this close request to all objects which have registered. Objects (e.g. UI beans) can register themselves by implementing the interface "IWorkpageLifeCycleListener" and by adding themselves to the workpage using the method "IWorkpage.addLifeCycleListener(...)".

The interface contains two important methods:

```

package org.ecInt.workplace;

public interface IWorkpageLifeCycleListener
{
    ...
    public boolean close();
    public void closeForced();
    ...
}

```

- The closing of the workpage is delegated to all listeners, i.e. The "close()" method of the listeners is called.
- Each listener now can decide if it agrees with closing or not - by either returning "true" or "false". If one listener returns "false" then the whole closing process is interrupted.

There is a second method "closeForced()". In this method the listeners just have to close - there is no chance to escape.

## Workpage Lifecycle Aspects - Additional Information

Please take a detailed look into the `IWorkpageLifecycleListener` interface, it also contains very useful other methods:

```
package org.eclnt.workplace;

public interface IWorkpageLifecycleListener
{
    // ...the ones from the previous chapter...
    public boolean close();
    public void closeForced();

    public void reactOnDestroyed();

    public void reactOnShownInContentArea();
    public void reactOnHiddenInContentArea();
    public void reactOnShownInPopup();
}
```

The most important method is the method “`reactOnDestroyed()`”. This is called in two situations:

- The workpage is closed by the user. After the “`close()`” and “`closeForced()`” method call, the “`reactOnDestroyed()`” method is called.
- Closing of the sessions, e.g. due to session timeouts: in this case the “`reactOnDestroyed()`” method is called without any other methods being called before.

You should use the `reactOnDestroyed()` method for tidying up resources,

## Inter Workpage Eventing

The workplace framework allows to start functions in so called workpages. Each function's managed beans are isolated from other functions' managed beans so that there is a maximum level of isolation between: each workpage is running “on its own” without getting disturbed by other workpages running in parallel.

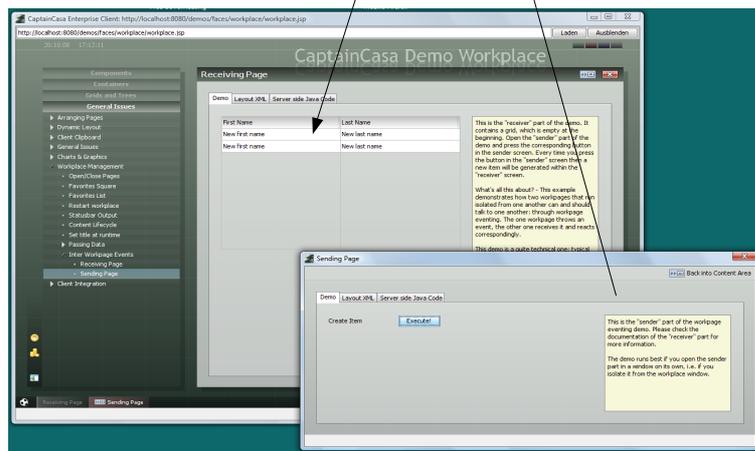
Now, there are certain situations in which you explicitly want one function of the workplace to talk to another function. For example there is a list of objects started as one function and a detail processing of an object started as a second function. As soon as the user updates the detail, the list should be updated as well.

Of course there is always the possibility to use some “model type of eventing” below the managed beans: i.e. if list and detail are talking to the same model then both can be updated by corresponding model events. But this in reality is not always available...

So what we explain in this chapter, is the possibility to let one function (workpage) talk to other functions (workpages) - but still following the concept of isolation between functions.

The solution is “by eventing”: there is a event processing that is added to the workpage management. You can throw events from one workpage and process these events on other workpages. In the demo workplace there is an example showing this:

## Workpage Event Processing



Two workpages are opened in parallel, one isolated as popup. When the user presses the button “Execute” on the foreground workpage then a grid item is added inside the background workpage.

Take a look onto the event sender:

```
public void onCreateItem(ActionEvent event)
{
    DemowpAddItemEvent workpageEvent = new DemowpAddItemEvent();
    getWorkpage().throwWorkpageProcessingEvent(workpageEvent);
}
```

The method onCreateItem() is called every time the user presses the button. It creates an instance of an event class and passes it to the workpage's method “throwWorkpageProcessingEvent()”. The event class itself looks the following way:

```
public class DemowpAddItemEvent
    extends WorkpageProcessingEvent
{
}
```

It just derives from “WorkpageProcessingEvent” which comes with the CaptainCasa server processing. Of course your implementation may contain additional members and methods that you want to pass with a certain event.

The receiver screens has the following code:

```
public class DemowpReceive
    extends WorkpageDispatchedBean
    implements Serializable
{
    public class MyWorkpageProcessingEventListener
        implements IWorkpageProcessingEventListener
    {
        public void processEvent(WorkpageProcessingEvent event)
        {
            if (event instanceof DemowpAddItemEvent)
                addNewItem();
        }
    }

    public class MyRow extends FIXGRIDItem implements java.io.Serializable
    {
        protected String m_lastName;
        public String getLastName() { return m_lastName; }
        public void setLastName(String value) { m_lastName = value; }

        protected String m_firstName;
    }
}
```

```

    public String getFirstName() { return m_firstName; }
    public void setFirstName(String value) { m_firstName = value; }
}

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

protected FIXGRIDListBinding<MyRow> m_rows = new FIXGRIDListBinding<MyRow>();

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

public DemowpReceive(IDispatcher dispatcher)
{
    super(dispatcher);
    getWorkpage().addWorkpageProcessingEventListener
        (new MyWorkpageProcessingEventListener());
}

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

public FIXGRIDListBinding<MyRow> getRows() { return m_rows; }

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

private void addNewItem()
{
    MyRow row = new MyRow();
    row.setFirstName("New first name");
    row.setLastName("New last name");
    m_rows.getItems().add(row);
}

}

```

It contains an event listener class/object that implements the interface `IWorkpageProcessingEventListener`. The listener gets called by the workplace processing every time a workplace event is thrown somewhere on another workpage. The listener checks if the event is relevant for the current page and executes a corresponding method.

Result: a coupling between two workpages is defined - without hard-wiring the processing of both pages.

## Addon Components in the Workplace Management

There are some additional add on components that are available within the workplace framework.

- `ROWWORKPAGECONTAINER` - this is one concrete content area. The component is internally created when using the “big” `ROWWORKPLACE` component.
- `ROWWORKPAGESELECTOR` - this is the task bar of one workpage container. The component is internally created when using the “big” `ROWWORKPLACE` component.
- `ROWWORKPLACEFUNCTIONTREE` - this is the function tree. You already got to know the component within the tutorial “Creating a Workplace”.
- `ROWWORKPLACEFAVORITES` - this is a list of favorites. You may drop information from the function tree into this area. As consequence a corresponding icon will be added.

# Workplace Perspective Management

So far you have seen:

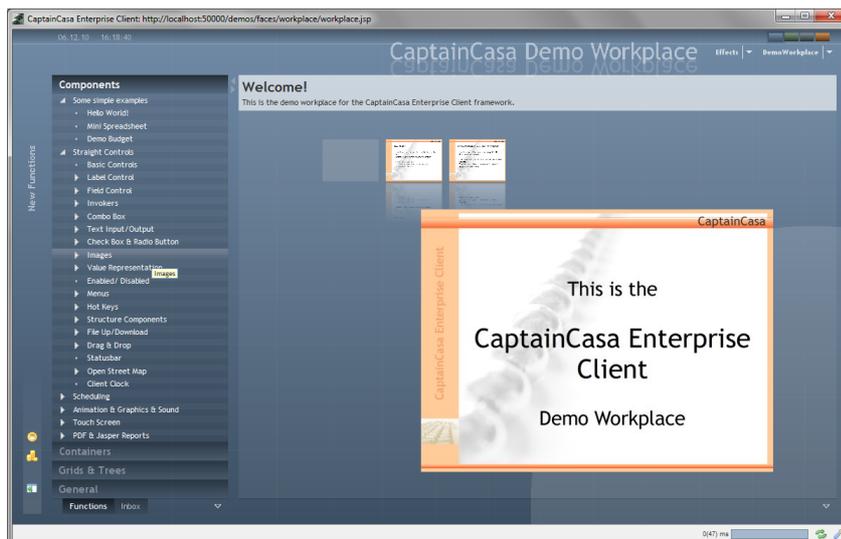
- There is a dispatcher concept for separating objects between different user activities.
- There is a workpage container concept, a workpage container being a container area in which workpages can be started.
- There may be multiple workpage containers, separated by a workpage-container-id - allowing to distribute several workpages onto several workpage containers.

Now, all this is brought together into a quite powerful framework on top of this: the workplace perspective management.

## What the Workplace Perspective Management does

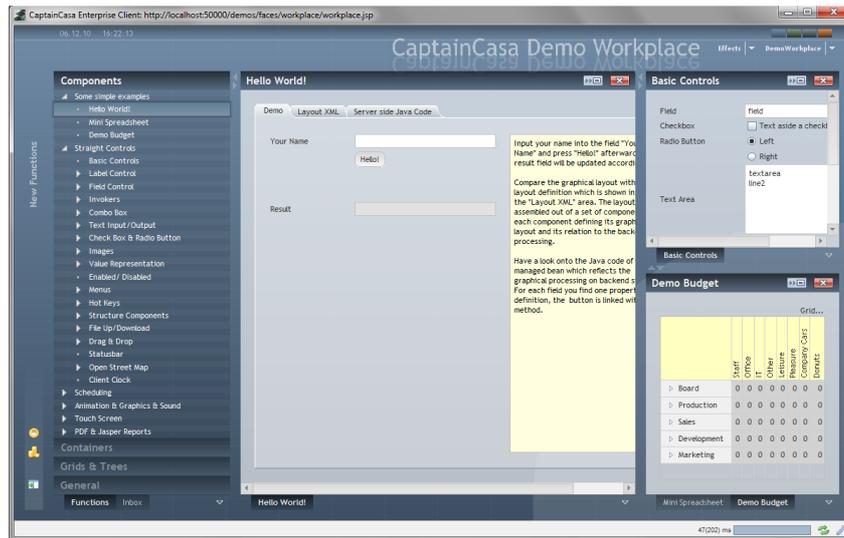
The basis of the workplace perspective management is the capability to run multiple workpage containers in parallel. The user can - by simply dragging and dropping move functions from the one container into the next. The user can split up workpage containers into new ones and as result can flexibly arrange a workpage container environment that fits to his/her needs.

Example: the demo workplace is started with the following default perspective:



There are two areas: one on the left, holding the function tree and some inbox management - and one on the right, so far not holding any page (the background page is shown).

After starting some functions, the user can drag&drop the corresponding workpages by picking them within their title area - or by picking the corresponding selector item. The result may look like:



The “perspective” now contains four areas, one of them (the central) one, being the “root” area. A “perspective” is the definition of a certain arrangement of workpage containers, together with the information what page is started within a workpage container.

The perspective management can either be used by API or it can be used by configuration. Within the configuration it is possible to...:

- ...assign multiple pre-configured perspectives to one user
- ...define one default perspective for the user
- ...save the personal perspectives that a user creates on his/her own

The following text will guide you through the steps to create such a “perspective-driven” workplace environment.

## The WORKPLACE Page

The first step is to create a page that actually holds the workplace. This is quite simple - just define the following page:

```
<t:rowbodypane id="g_1" background="#808080">
  <t:rowworkplace id="g_2" objectbinding="#{d.workpageContainer}"
    wpselectorposition="bottom" />
</t:rowbodypane>
<t:rowstatusbar id="g_3" />
```

The central component is the ROWWORKPLACE component. It points via its OBJECTBINDING attribute to the workpage dispatcher's property “workpageContainer”.

When previewing the page the result looks like:



You see two workpage container areas, one on the left holding “Page 1”, one on the right holding no page. ...well yes, if we say “You see...” then actually you see some info about a missing page on the left, and some empty area on the right - but this is OK (up to now).

The information comes out of a default configuration that splits the workplace into two parts, and that opens “Page 1” on the left side.

Please note: you can create a workplace page by using a corresponding template when creating the page in the Layout Editor:



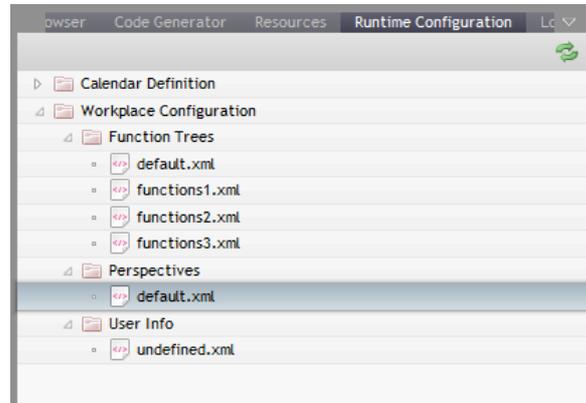
In this case the page will look like:



Some background coloring and some additional useful components are arranged in addition to the central ROWORKPLACE component.

## Configuring a Perspective

Now, let's update this perspective. To do so, open the “Runtime Configuration” tool within the layout editor's tool area (on the right):



There are three sections of configuration:

- function trees
- perspectives
- user infos

Inside the “Perspectives” section you see a file “default.xml”. By double clicking you can open and edit the file content. The default content is:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<workplaceTileInfo>
  <text>Demo Workplace</text>
  <split>
    <orientation>horizontal</orientation>
    <dividerLocation>250</dividerLocation>
    <subContainer1>
      <id>FUNCTIONS</id>
    </subContainer1>
    <subContainer2/>
  </split>
  <startViews>
    <jspPage>/page1.jsp</jspPage>
    <id>page1</id>
    <text>Page 1</text>
    <decorated>true</decorated>
    <closeSupported>true</closeSupported>
    <popupSupported>true</popupSupported>
    <openMultipleInstances>false</openMultipleInstances>
    <startSubworkpageContainerId>FUNCTIONS</startSubworkpageContainerId>
  </startViews>
</workplaceTileInfo>
```

A perspective configuration always consists of two parts:

- The definition of the screen split up - in this case the layout contains a split pane, being divided into two workpage containers, one with id “FUNCTIONS”, the other one without id (the default one).
- The definition of what page to start where by default - in this case the page “/page1.jsp” is started within the workpage container with id “FUNCTIONS”.

You may update the file to your needs. Internally the XML definition that you edit is a JAXB binding to class “WorkplaceTileInfo”. Please consult the JavaDoc for details what configuration is possible to be done within the XML file.

Please note: the runtime configuration that you edit, is internally stored within the

/webcontent/eclntjsfserver/config/ccworkplace directory of your project. You can also edit the information directly within your development environment (Eclipse, ...).

You need to reload your application in order to bring changes from you project into the runtime.

## Configuring multiple Perspectives for multiple Users

After now having maintained the first perspective you can set up multiple perspectives, either to be used by one user (user may switch between different perspectives) or to be used by several users.

The way to do is: Just create new perspective definitions within the “Perspectives” section of the runtime configuration. Then edit the files within the “User Info” section. The default configuration file that is chosen is “undefined.xml”:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<workplaceUserInfo>
  <perspectives>default</perspectives>
  <defaultPerspective>default</defaultPerspective>
  <functionTree>default</functionTree>
  <backgroundPage>/empty.jsp</backgroundPage>
</workplaceUserInfo>
```

Within the file you may define multiple “perspectives” to be used. These are the “potencial” perspectives for one user. And there is one “defaultPerspective” which is the one that is opened by default.

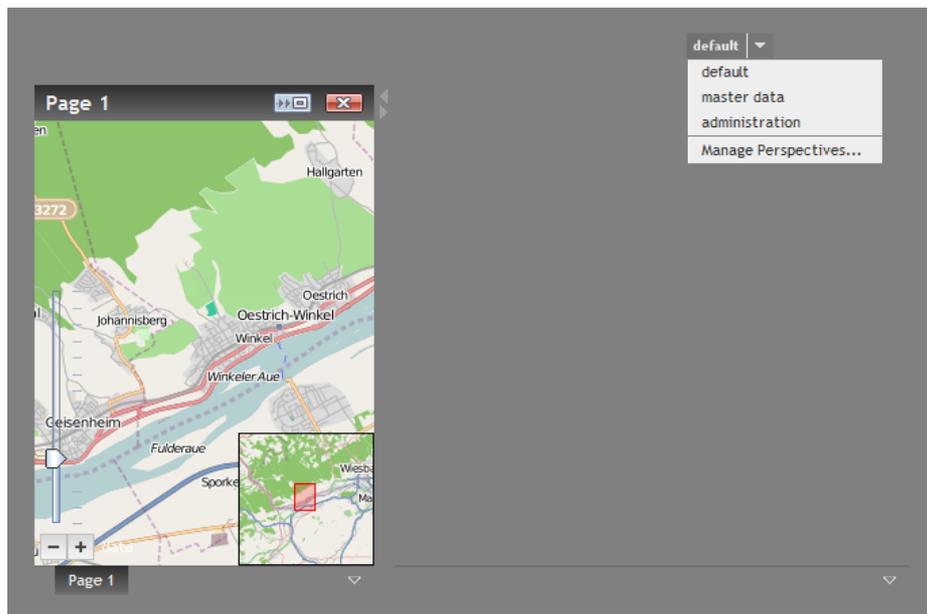
How is this user info read at runtime?

The system calls interface “IUserAccess” in order to ask you application what the id of the currently logged on user is. With the result it tries to find a corresponding definition “<userId>.xml” within the “User Info” section.

When not explicitly defining an “IUserAccess” interface then a default/dummy one is used, returning user “undefined” by default - this is the reason, why “undefined.xml” is selected in the default environment.

## Switching between different Perspectives

When defining multiple perspectives to be available for a certain user, then you may use a nice component WORKPLAGEPERSPECTIVESELECTOR to switch between the perspectives.



The layout definition is:

```
<t:rowbodypane id="g_1" background="#808080">
  <t:row id="g_2">
    <t:coldistance id="g_3" width="100%" />
    <t:workplaceperspectiveselector id="g_4"
      objectbinding="#{d.workpageContainer}" />
    <t:coldistance id="g_5" width="100%" />
  </t:row>
  <t:rowdistance id="g_6" height="20" />
  <t:rowworkplace id="g_7" objectbinding="#{d.workpageContainer}"
    wpselectorposition="bottom" />
</t:rowbodypane>
<t:rowstatusbar id="g_8" />
<t:pageaddons id="g_pa"/>
```

The component is bound via its OBJECTBINDING attribute to the “workpageContainer” of your workpage dispatcher.

As you may have seen already, the menu that allows to switch between different perspectives also provides a section “Manage Perspectives...”. The purpose behind is:

- The user may create new perspectives by starting new functions (within your application) and by dragging and dropping workpages into different workpage containers.
- The function “Manage Perspectives...” opens up a dialog, in which the user can store his/her personal perspective.

## Loading the Workplace Perspective when changing the User

When changing the user then you have to tell the workplace perspective management about. This is done by calling the following function:

```
IworkpageDispatcher
  .getworkpageContainer()           // passes IworkpageContainer
  .prepareworkplaceForCurrentUser()
```

So this is the function to be called after e.g. an explicit log on of a user.

## Workplace Perspective - Low Level API

So far you saw in this chapter how to set up a workplace using a configuration via XML definitions. This is the “automated and most comfortable” way of using the workplace perspective management.

Of course you can directly access the workplace perspective management via interfaces. The most important issues are:

In your dispatcher implementation you can switch off the automatic functions by creating a `WorkpageContainer`-instance through the following constructor:

```
/**
 * This constructor may be used if you want to switch off the usage
 * of the default workplace management. The default workplace management
 * reads the runtime configuration for the current user and creates
 * a corresponding workplace perspective.
 */
public workpageContainer(IWorkpageDispatcher dispatcher,
                        boolean withPreparingWorkplaceForCurrentUser)
{
    ...
}
```

For creating an own instance you need to override the dispatcher's `createWorkpageContainer()` method, e.g. in the following way:

```
protected IWorkpageContainer createworkpageContainer()
{
    return new workpageContainer(this, false);
}
```

Now there is “no automated magic” anymore, but you can directly access the interfaces. The `workpage container (IWorkpageContainer)` provides access to the perspective management by its `tile manager`:

```
public interface IWorkpageContainer
{
    ...
    public workplaceTileManager getTileManager();
    ...
}
```

The `WorkplaceTileManager` e.g. allows to render a certain layout that you pass inside via the method:

```
public class workplaceTileManager
{
    ...
    public void importworkplaceTileInfo(workplaceTileInfo tileInfo)
    {
        ...
    }
    ...
}
```

Please use these hints and dive into the `JavaDoc` documentation for the corresponding classes.

---

## Workplace Functions Management - ...by declaration

Function trees are the typical tree structure of a workplace, providing all call-able functions for a certain user. You can either define function trees by program or you can define function trees by XML definition - this is what this chapter is about.

There are certain steps to set up a function tree, that is managed “by declaration”:

- Create a page to hold the function tree(s)
- Set up the function tree's XML definition
- Assign the function tree XML definition to the user definition

## Creation of Page to hold Function Tree(s)

The easiest way to create a page to hold the function trees is to use a certain template when creating the page in the Layout Editor:



After creating the page you will see the following preview in the Layout Editor:



Well, this looks not too nice - but remember: some more colorful background typically is provided “behind” the page when being used, so things will look much nicer then.

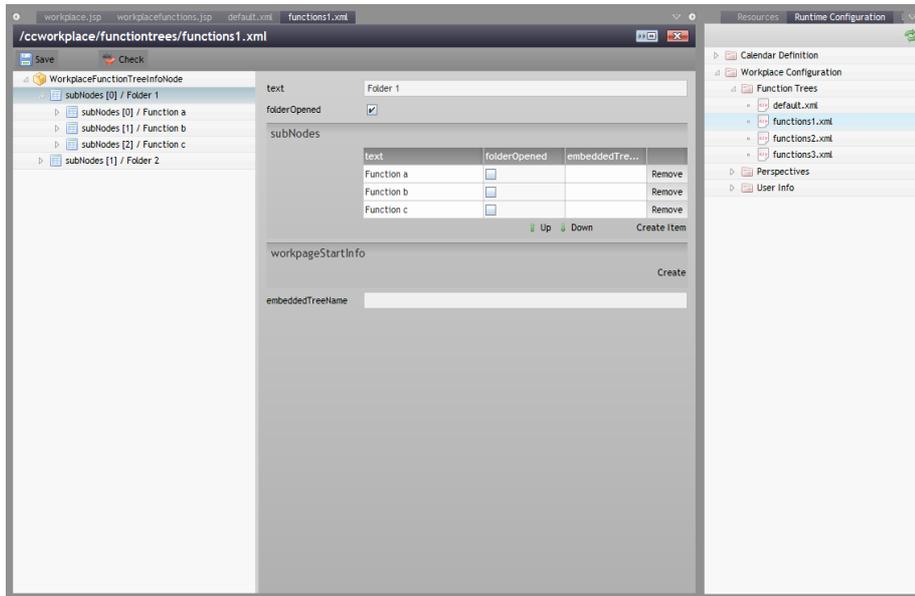
```
<t:row id="g_1">
  <t:pane id="g_2" background="#00000010" height="100%" width="100%">
    <t:rowworkplacefunctions id="g_3"
      objectbinding="#{d.workpageContainer}" />
  </t:pane>
</t:row>
```

The central component is the ROWWORKPLACEFUNCTIONS component - pointing to the dispatcher's workpage container via its OBJECTBINDING attribute.

The component checks the current user (IUserAccess interface), and loads the function tree that is defined for this user. By default there are some dummy function tree definitions already available - that's where the content you see is from.

## Set up Function Trees

Function trees are set up within the runtime configuration - very similar to the definitions in the perspective management:



A function tree is a hierarchical arrangement of node definitions. Each node may either be a “folder node” - i.e. it contains a number of sub-nodes. Or it may be a “page node”, i.e. it is associated with some information about which page to start.

There is a special type of node definition: nodes, that point to other function tree definition (in the default project this is the “default.xml” definition). By using these nodes you can build up function trees out of several other function trees - you do not have to embed all functions for a certain scenario into one tree definition, but can split the information into several re-usable tree definitions.

When defining function trees, please note: the first level of node definitions is always used to form the “outlook bar”.

## Assign Function Tree to User

The assignment is done within the user info section - each user points to one function tree.

# Working with Macros

CaptainCasa Enterprise Client provides a so called Macro Management that can simplify the server side definition of JSP pages significantly. It is typically applied, when you have a quite generic way of passing data from and to your UI components

In short: a macro is a definition which allows to automatically set component attributes based on the macro's input parameters. The macro is executed at runtime on server side.

---

## Example Use Case

A FIELD component provides a high number of attributes that you may use:

- The TEXT attribute is the field's value
- The ENABLED attribute tells if the user may input data or not
- The BGPAIN/ BACKGROUND and FOREGROUND attribute determine the field's coloring

In an application you typically want to control all of these attributes dynamically:

Well, with the TEXT, it's obvious - this is the core data passed in and out. But the other attributes are important as well: dependent on the user's rights or dependent on a logical decision within your business logic you want to enable or disable the field. If the field's content is not correct you want to highlight the field. Etc. etc. ...

As consequence your server side managed bean provides all this information for the FIELD component. The field's attribute definition may look like:

```
<field id="g_57"
      text="#{bean.zipCode}"
      bgpaint="#{bean.zipCodeBGPAIN}"
      enabled="#{bean.zipCodeENABLED}"
      width="200"/>
```

This is just an example! You may also shift the "paint" and "enabled" information into parallel objects!

What you see from the FIELD component definition: it binds to the property "zipCode" - and all the information around is arranged in properties that are linked with "zipCode".

It's now a hell of work to manually maintain all this information - expecting that there are many fields following the same naming pattern. And: in case you extend the naming and logic pattern (e.g. introducing a new property "#{bean.zipCodeFOREGROUND}" to rule the foreground color) you need to update all the FIELD definitions within your JSP pages.

That's where macros come in! A macro is a simple and consistent way of defining how to apply attribute values following a naming pattern. Sounds difficult, but is not at all!

---

## Macro Definition

A macro is a Java object supporting interface "IMacro". There are currently two way to create macro instances:

- By defining an XML definition - in this case a default macro object will be created automatically that is following the XML definition
- By defining a class implementing "IMacro" and by registering this class in a certain XML file

## Macro Definition by XML

A macro is an XML definition that is stored in the directory “/eclntjsfserver/config/macros”. The file name that you assign is the macro's name.

Let's look onto the macro definition - using the name “prop.xml” - that simplifies the creation of FIELD components within the example use case scenario:

```
<macro>
  <applysto>
    <tag name="t:field"/>
    <tag name="t:combofield"/>
  </applysto>
  <parameters>
    <parameter name="property"/>
  </parameters>
  <attributes>
    <attribute name="text" value="#{bean.${property}}"/>
    <attribute name="bgpaint" value="#{bean.${property}BGPAIN}"/>
    <attribute name="enabled" value="#{bean.${property}ENABLED}"/>
  </attributes>
</macro>
```

The definition includes:

- You define for which components the macro is usable.
- You define the input parameters of the macros. In the example use case there is one parameter (“property”).
- You define how attributes are automatically set - referencing the input parameter's value (“\${property}”).

## Macro Definition by Class Implementation

You can implement an own class supporting interface “IMacro”.

Example:

```
package demomacros;

import org.eclnt.jsfserver.elements.BaseComponentTag;
import org.eclnt.jsfserver.elements.macros.IMacro;

public class DemoMacro implements IMacro
{
    public boolean checkIfApplicable(String tagName)
    {
        if (tagName.equals("t:field")) return true;
        if (tagName.equals("t:combofield")) return true;
        return false;
    }

    public boolean checkIfAttributeIsAffected(String attribute)
    {
        if (attribute.equals("text")) return true;
        if (attribute.equals("bgpaint")) return true;
        if (attribute.equals("enabled")) return true;
        return false;
    }

    public void executeMacro(BaseComponentTag tag, String[] macroParams)
    {
        String property = macroParams[0];
        String property = macroParams[0];
        if (tag.getAttributeMap().get("text") == null)
            tag.setText("#{bean."+property+"}");
        if (tag.getAttributeMap().get("bgpaint") == null)
            tag.setBgpaint("#{bean."+property+"BGPAIN}");
        if (tag.getAttributeMap().get("enabled") == null)
            tag.setEnabled("#{bean."+property+"ENABLED}");
    }
}
```

```

public String getName()
{
    return "propbyclass";
}

public String[] getMacroParamNames()
{
    return new String[] {"property"};
}
}

```

The macro currently does exactly the same as the XML macro definition that was shown within the previous chapter. But, of course: now you could extend the macro's logic by any kind of Java programming.

From the example code you see that the macro only transfers values into the component's attributes if they are null. This means: in case the user explicitly defines component attributes then the macro will not override these definitions.

- This is to be pointed out: if a macro does not do this check for null values then quite confusing situations will occur. The user e.g. specifies a value for a component that will be overwritten by your macro - as consequence the user will see his/her definition within the .jsp page, but this will not be the definition actually applied at runtime. - Consequence: you may only overwrite attribute values if they are null! (By the way: style attribute values are applied after macro values are applied!)

The macro is used both at runtime (this is when the executeMacro() is called) - and at design time: the Layout Editor needs to know which components and attributes of a component are affected. As consequence: keep the macro's internal Java coding as simple as possible, so that it can be instantiated without problems by the Layout Editor environment.

The class needs to be registered in the configuration file “/eclntjsfserver/config/macros/javamacros.xml”:

```

<javamacros>
  <javamacro classname="demomacros.DemoMacro"/>
</javamacros>

```

## Pay Attention when processing Grid Cells

When implementing your own macro by writing your own class for interface “IMacro”, then there is one special aspect that you have to pay attention to.

A grid definition is typically done into the following way:

```

FIXGRID objectbinding="#{d.TestUI.grid}"
  GRIDCOL text="..."
    FIELD text=".{firstName}"

```

The property within the grid cell definition is typically not written with an absolute expression but with a relative expression (here: `.{firstName}`). At runtime the grid processing creates several instances of the cell component (here: FIELD) and assigns a correct, full expression to each cell instance.

Example: if the grid above creates its first line of controls the FIELD's text will get assigned the following expression: “`#{d.TestUI.grid.rows[0].firstName}`”.

Please be aware of the fact that the macro is called first for the original component (FIELD text=`“.{firstName}”`) and then for the generated components (FIELD text=`“#{d.TestUI.grid.rows[0].firstName}”`, ...). Changes that you might apply via macro when processing the “`“.{...}”`” expression will automatically be applied to the per-line-components.

In some special situations (e.g. when modifying a component's attribute) you need to pay attention to this, in order to avoid a double-processing of the same rules you apply via Macro. In this case just check if an expression starts with “.{“ and only apply your rules when an expression starts with “#{“.

---

## Macro Usage

A macro is used in a component by defining the attribute `ATTRIBUTEMACRO`. Let's use the use the macro “prop.xml” of the previous chapter and apply it to the example use case:

```
<field id="g_57"
      attributemacro="prop(zipCode)"
      width="200"/>
```

That's it! The field will look and behave exactly the same as in the “long definition” in the chapter above.

And: you may update the macro any time - e.g. you may add a new attribute that is generated (e.g. “zipCodeFOREGROUND”). You do not need to re-define the JSP pages using the macro - it is automatically applied. Of course you must not change the macro's structure in an incompatible way - e.g. removing a macro parameter or changing the order of parameters.

---

## Some more Details

### Overriding a Macro Value

You may any time override a macro by an explicit attribute definition. Example:

```
<field id="g_57"
      attributemacro="prop(zipCode)"
      enabled="false"
      width="200"/>
```

The ENABLED-definition of the component will always be stronger than the macro. A macro will only set a component's attribute value if it is not explicitly defined.

(Please note: in case of Java class based macros the implementor of the macro has to guarantee that no already defined values are overwritten!)

### Tolerant Attribute References

When applying a macro at runtime the macro management will only match these macro attribute definitions that can be applied to the UI component.

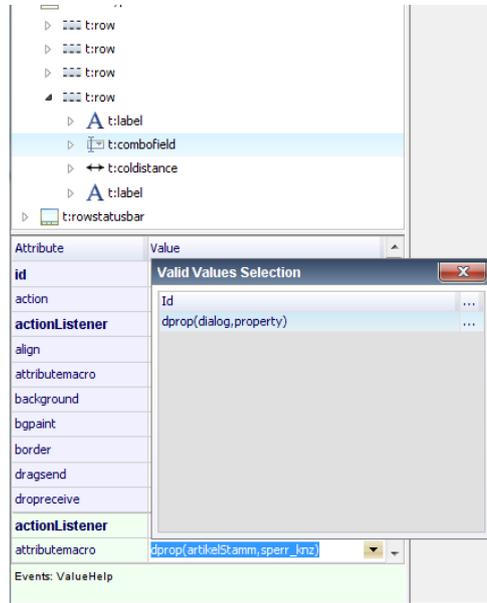
Example: the macro “prop.xml” from above should also be used for `FORMATTEDFIELD` components. These components do not hold their content in the `TEXT` attribute but in the `VALUE` attribute. As consequence the macro is extended in the following way:

```
<macro>
  <applysto>
    <tag name="t:field"/>
    <tag name="t:combofield"/>
    <tag name="t:formattedfield"/>
  </applysto>
  <parameters>
    <parameter name="property"/>
  </parameters>
  <attributes>
    <attribute name="text" value="#{bean.${property}}"/>
    <attribute name="value" value="#{bean.${property}}"/>
    <attribute name="bgpaint" value="#{bean.${property}BGPAINT}"/>
  </attributes>
</macro>
```

```
<attribute name="enabled" value="#{bean.${property}ENABLED}"/>
</attributes>
</macro>
```

## Macros in the Layout Editor

Within the Layout Editor there is a “attributemacro” field in which you can define the macro:



Only these macros will show up in the value help that can be applied to the selected component.

Once defining a macro then all attributes that are set by the macro will be marked:

Attribute	Value
id	g_20
action	
actionListener	(macro)
align	
attributemacro	dprop(artikelStamm,sperr_knz)
background	
bgpaint	(macro)
border	
dragsend	
dropreceive	
actionListener	(macro)
attributemacro	dprop(artikelStamm,sperr_knz)

## The REFERENCE Attribute

The one way of passing parameters into a macro definition is to list the parameters in the macro call. Example: “dprop(artikelStamm,sperr\_knz)”.

The second way is to use the REFERENCE attribute that is available with each component. Within the reference attribute you can pass a list of “name:value” pairs, separated by semicolon. This list can be accessed both from the XML Macro definition and from the implemented Macro definition.

Example: in a page there is the following definition:

```
<t:field id="g_48" attributemacro="crud.DetailField()"
reference="b:address;p:town" />
```

The reference contains a “b”-value and a “p”-value (“b” for bean, “p” for property). Of course, the naming is completely up to you, “b” and “p” are just examples.

Now you can access these values inside your XML macro definition:

```
<macro>
  <applysto>
    <tag name="t:field"/>
    <tag name="t:formattedfield"/>
    <tag name="t:calendarfield"/>
    <tag name="t:textarea"/>
    <tag name="t:combobox"/>
    <tag name="t:radiobutton"/>
  </applysto>
  <parameters>
  </parameters>
  <attributes>
    <attribute name="bgpaint" value="#{d.${ref.b}.selBgpaints.${ref.p}}"/>
    <attribute name="text" value="#{d.${ref.b}.sel.${ref.p}}"/>
    <attribute name="value" value="#{d.${ref.b}.sel.${ref.p}}"/>
    <attribute name="width" value="100%"/>
  </attributes>
</macro>
```

The value of “b” is accessed with “\${ref.b}” and the value of “p” with “\${ref.p}”. The macro itself does not provide any parameters - all macro input information is taken from the REFERENCE attribute.

The implementation version of the macro would contain the following code:

```
public void executeMacro(BaseComponentTag tag, String[] macroParams)
{
    String reference = tag.getAttributeMap().get("reference");
    if (reference == null)
        return;
    Map<String,String> m = ValueManager.decodeComplexValue(reference);
    String bValue = m.get("b");
    String pValue = m.get("p");
    ...
    ...
}
```

## Macros within the Grid Processing (FIXGRID)

When writing Java-base macros then you need to be aware of how the grid processing (FIXGRID component) internally works.

A grid definition contains column definitions:

```
FIXGRID sbvisibleamount="3"
GRIDCOL
  FIELD/... <== cell component, FIELD as example
GRIDCOL
  FIELD/... <== cell component, FIELD as example
```

The grid interprets this structure when it is accessed first time and internally multiplies out this structure:

```
FIXGRID
GRIDROW
  GRIDHEADERLABEL <== derived from first GRIDLCOL
  GRIDHEADERLABEL <== derived from second GRIDCOL
GRIDROW
  FIELD
  FIELD
GRIDROW
  FIELD
  FIELD
GRIDROW
  FIELD
  FIELD
```

From macro processing point of view there are two situations when macros are applied:

- Processing of JSP page: this is what happens normally - the page's component tags are

read and transformed into component instances.

- Processing of FIXGRID component: this is the multiplication of grid cell items per row

It's now your choice when to apply your macro's functions: either with the JSP processing or with the FIXGRID processing or with both.

To find out BaseComponentTag provides a special method:

```
BaseComponentTag.isGridCellComponent()
```

This method returns “false” in the JSP processing phase, and returns “true” if the current component tag is a “multiplied-out-one”.

By the way: what's the main difference, when the components below GRIDCOL are multiplied out? - Answer: the expressions... - An expression “.{xxx}” is transferred into “#{yyy.rows[index].xxx}” by the grid management.

## Related Topics

- See “Dynamic Page Management” to dynamically define components and attributes within a page.
- See “Adding own Components” to define components that themselves consist out of other components.

# Adding own Components

There are certain situations in which you want to add own components to the CaptainCasa Enterprise Client processing:

- (A) You may want to extend the client by own swing components. In this case these components must on the one hand adhere to the client's component framework and on the other hand must have a server side JSF representation.
- (B) You may want to add “composite components”, i.e. components that compose existing components in a smart and reusable way.

This part of the documentation does not cover the (A) aspect yet - but concentrates on the (B) aspect. Documentation on (A) is included in an own documentation covering the client part (Swing part) of writing new components.

Component Management in principal is covered by JSF itself. But...: to go through all the JSF aspects about writing own components without guidance is quite difficult. Typically you want to re-use existing CaptainCasa components, either by grouping them in a certain way or by using them in a certain usage-context - so all you need is some administration on top of existing components. This is what this chapter is about.

---

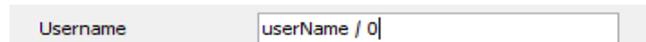
## Adding Composite Components

The example that is used in this chapter is a quite simple one: a combination out of a label and a field - a “LABELFIELD” component, as it will be called later.

By specifying the following layout definition...

```
<demo:labelfield id="g_4" labeltext="Username" text="#{loginBean.userName}" width="200" />
```

...a LABEL and a FIELD component will be created, looking like:



The LABELFIELD component will be a component that manages (“composes”) its content by arranging a LABEL and a FIELD component.

### Component <=> Component Library

In the following text there will be the differentiation between the term “component” and “component library”. Well, the difference is quite easy: a component library is a collection of components.

Typically you define one component library holding several components.

### Composite Component “Artifacts”

The following server side development objects will play a role in the further explanations:

- “tld file” - the tag library definition that is expected by JSF for component tag definitions. There is one tld-file per component library, in which meta data for several components is listed.
- “facesconfig.xml” - components need to be registered in this JSF file as well. There is one facesconfig file that holds all the components that you want to use in addition to the normal CaptainCasa components.
- “Component tag class, component class” - the Java classes which are the server side

implementation of the component. Both classes are standard JSF classes, but based on tag library base classes coming from CaptainCasa. Per component there is a “component tag class” and “component class”.

- “controllibraries.xml” - a file expected by the CaptainCasa server side runtime in which all additional component libraries are listed.
- “controlsarrangement.xml” - a file expected by the layout editor of CaptainCasa that tells how components can be assembled. Per component library there is one controlsarrangement.xml definition.

Quite a lot of things to be aware of... But: at the end it's quite easy!

One advice before starting: carefully follow all naming conventions that you will get to know! And...: carefully type names (e.g. uri-Strings): they are case sensitive!

## Create the “tld file”

Create a tag library definition file, directly located within the WEB-INF directory. Select a nice name, that does not clash with other file names.

Example: the following file has the name “democontrols.tld”, the content is:

```
<?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>labelfield</name>
    <tag-class>democontrols.LABELFIELDComponentTag</tag-class>
    <body-content>scriptless</body-content>
    <attribute><name>id</name></attribute>
    <attribute><name>width</name></attribute>
    <attribute><name>text</name></attribute>
    <attribute><name>labeltext</name></attribute>
  </tag>
</taglib>
```

The important parts are:

- URI: for each component library you need to define an URI, that will later on uniquely identify the component library. While controls will later on be defined together with a prefix (e.g. “demo:labelfield”), that might be different from installation to installation, the URI really is the unique identification of a control library.
- Tag definition, name, tag-class: all component implementation must reside in ONE package. The name of the class must be “package.<NAMEOFCOMPONENT>ComponentTag”. Do only use lower case letters for the name of the component (“labelfield”) and translate all the characters into upper case when naming the tag class (“LABELFIELDComponentTag”).
- Always add the tag “body-content” with value “scriptless”.
- Tag definition, attributes: each component must have a tag “id”. List all the attributes that your component has. Use lowercase names for your attributes. Try to re-use these names, which are already used by CaptainCasa components (e.g. “width” and “height”) - of course you can add own attribute names, but try to be as consistent as possible to CaptainCasa components.

## Update the “faces-config.xml”

Components need to be registered inside the faces-config.xml in the following way:

```
<faces-config
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
  version="1.2">

  <application>
  ...
  ...
</application>

  <component>
    <component-type>democontrols.LABELFIELDComponent</component-type>
    <component-class>democontrols.LABELFIELDComponent</component-class>
  </component>

  <managed-bean>
  ...
  ...
```

Each component must be listed in exactly the way shown in the example file.

## Implement the component

You need to do two class implementations, both being kept in a class file that is reachable by the WEB-application loader (i.e. Class file should be below WEB-INF/classes or WEB-INF/lib). Both classes need to be in the package that you used within the tld-file and the faces-config.xml.

The first class is an “easy one”:

```
package democontrols;

import org.eclnt.jsfserver.elements.BaseComponentTag;

public class LABELFIELDComponentTag
{
  extends BaseComponentTag
  {
    public void setLabelText(String value)
    {
      m_attributes.put("LabelText",value);
    }
  }
}
```

This is the tag implementation. For each attribute of your component that does not have a corresponding set-method in the BaseComponentTag class you need to implement the set-method in the way shown:

- “id”, “text” and “width” already have an implementation in the BaseComponentTag implementation. Please either use the JavaDoc of CaptainCasa Enterprise Client or use your IDE (e.g. by showing all methods inherited from BaseComponentTag) in order to find out, if a certain attribute is already implemented on BaseComponentTag-level or not.
- “LabelText” does not have an implementation and thus needs to be added

The second class does the aggregation of other components. It is the class which is the component class.

```
package democontrols;

import java.io.IOException;
```

```

import javax.faces.context.FacesContext;

import org.eclnt.jsfserver.elements.BaseActionComponent;
import org.eclnt.jsfserver.elements.BaseComponent;
import org.eclnt.jsfserver.elements.impl.FIELDComponent;
import org.eclnt.jsfserver.elements.impl.FIELDComponentTag;
import org.eclnt.jsfserver.elements.impl.LABELComponentTag;

public class LABELFIELDComponent
    extends BaseActionComponent
{
    /*
     * Rendering phase - begin of rendering of component.
     */
    public void encodeBegin(FacesContext context) throws IOException
    {
        if (getChildren().size() == 0)
        {
            // pick component's own attributes
            String width = getAttributeString("width");
            String text = getAttributeString("text");
            String labeltext = getAttributeString("labeltext");
            // create label component
            LABELComponentTag lct = new LABELComponentTag();
            lct.setId(createSubId());
            lct.setText(labeltext);
            lct.setWidth("120");
            BaseComponent lc = lct.createBaseComponent();
            // create field component
            FIELDComponentTag fct = new FIELDComponentTag();
            fct.setId(createSubId());
            fct.setText(text);
            fct.setWidth(width);
            BaseComponent fc = fct.createBaseComponent();
            // add components to component hierarchy
            getChildren().add(lc);
            getChildren().add(fc);
        }
    }
}

```

Components are kept as tree in JSF - so components that you add as part of your own component need to be added via corresponding methods in the list provided by the “getChildren()” method.

Components are NOT created directly (e.g. by “new FIELDComponent()”), but always through their component tag class. By doing so, all default issues that are part of a component creation are automatically done, including:

- If an attribute points to an expression then a corresponding JSF binding object is created.
- If attributes are pre-set by a style definition then this is done as part of the component creation.
- If there are content replacements to be applied for expressions, then they can be easily applied: there are some comments on this later on, which you carefully have to read when directly creating expressions in your component.
- ...and many more things, that you have to pay attention to, if directly creating component instances.

## Update the “controllibraries.xml”

The CaptainCasa server runtime needs to know certain information about each component library that is used. For this reason there is a file “controllibraries.xml” which is located in the directory “eclntjsfserver/config” of your web application.

The file should look like:

```
<!--
Registration of control libraries. Registration is required in case
of own controls that are based on CaptainCasa controls. Information
is both required at runtime and at design time.
-->

<controllibraries>
  <controllibrary prefix="demo"
    tldfilename="democontrols.tld"
    packagename="democontrols"
    uri="/WEB-INF/democontrols"/>
</controllibraries>
```

Pay attention: all attributes like “tldfilename”, “packagename” and “uri” must be 100% in synch with the definitions you did in the previous steps.

The prefix is the short name of the control library, that will from now on be used as prefix for all component references. E.g. the component “labelfield” will be named “demo:labelfield” in all layout definitions.

## Add “controlsarrangement.xml”

Each component library has an XML definition file that tells the layout editor of CaptainCasa Enterprise Client, in which way components can be arranged.

The file looks like:

```
<controlsarrangement>
  <tag name="t:row" below="demo:labelfield"/>
</controlsarrangement>
```

The file contains the information, under which control your new control can be placed.

The file needs to be placed into the following file: “<componentpackagename>/resources/controlsarrangement.xml”.

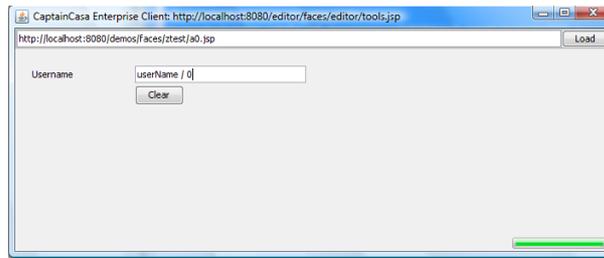
## Result

Now, having done all these steps, you can add the component into your layout. The component will appear inside the layout editor, just as normal base components of the CaptainCasa Enterprise Client framework.

Example: the following layout is defined as:

```
<f:view>
<h:form>
<f:subview id="ztest_a0g_29">
<t:rowbodypane id="g_2" popupmenu="GENERAL" rowdistance="3" >
  <t:row id="g_3" >
    <demo:labelfield id="g_4" labeltext="Username" text="#{loginBean.userName}"
width="200" />
  </t:row>
  <t:row id="g_5" >
    <t:coldistance id="g_6" width="120" />
    <t:button id="g_7" actionListener="#{loginBean.onClear}" text="Clear" />
  </t:row>
</t:rowbodypane>
</f:subview>
</h:form>
</f:view>
```

The corresponding screen output is:



---

## Issues when creating new Components

### Using Hot Deployment? - Pay Attention!

#### ***Add the Component Source into the right Source Directory***

If having set up your CaptainCasa project with using hot deployment, then be careful:

- The code (component tag and component class) and the controlsarrangement.xml definition must not be defined within the “/src” part of your project, but within the “/src\_webinf” part of your project!

Background: JSF is directly managed within the servlet environment, i.e. it is managed within the web application class loader. This is the one reading classes and libraries from the WEB-INF/classes and WEB-INF/lib directory of your web application.

#### ***Check Project Definition***

Within the Layout Editor the resource “controlsarrangement.xml” is read in order to show the user which component can be arranged below which other component. The “controlsarrangement.xml” file is located within the source code, so the Layout Editor looks for the source code. There are two directories that point to the source directories within the project definition file:

- “javasourcedirectory” - in a hot deployment scenario this directory is the one holding the code that is compiled into the hot deployment folders (typically: “eclnthotdeploy/classes”)
- “javasourcwebinfdirectory” - in a hot deployment scenario this directory is the one holding the code that is compiled into the WEB-INF/classes directory

Make sure that both project definitions are correctly defined - “javasourcwebinfdirectory” needs to point to the source directory in which your control implementation is kept.

#### **Commonly used Attributes**

There is a set of attributes that you “should” provide in order to keep consistency with the existing components:

- ID: ...this was already mentioned, but we re-emphasize here again: the component must have an attribute “id”!
- ATTRIBUTEMACRO and REFERENCE: these are the attributes that are required in order to use macros for your component
- COMMENT: the comment...

## Some useful methods of Components

As shown in the demo above there are some methods that are quite useful for implementing your component class:

```
public void encodeBegin(FacesContext context) throws IOException
{
    if (getChildren().size() == 0)
    {
```

This is what typically is done within a composite component: when the component is rendered it checks if its composed content is available or not. This may happen in the following situations:

- the component is created as new component (page rendered the first time)
- the component is rendered as different object, e.g. when the whole component tree is rebuilt

```
// pick component's own attributes
String width = getAttributeString("width");
String text = getAttributeString("text");
```

With “getAttributeString(…)” you request the String value that was maintained with the component. If your component is part of a layout (.jsp file) then the components holds certain attributes (e.g. “width='100' text='#{d.xxx.yyy}”).

The method returns the String value that is maintained with the corresponding attribute. Pay attention: there is a second method (“getAttributesGet(…)”) that returns the actual value of the attribute. In case you defined an expression behind a certain attribute, “getAttributesGet(…)” will return the actual object, that is reached via the expression. This means, the expression is resolved and the result of the resolution is passed back.

```
getChildren().add(m_field);
```

This is the method that is used for building up a component hierarchy. Each component may hold a list of sub-components.

## Expressions - and how to update in the correct Way

In quite a lot of cases (e.g. workplace management, page bean management) expressions are updated through a content replace management. E.g. in case of the workplace an expression may be updated from “#{d.DemoUI.name}” to “#{d.d\_1.DemoUI.name}”.

If you now create a component that internally creates own components as well - and if you create new expressions in the component then you have to make sure that the content replacement is passed correctly.

Example:

```
FIELDComponentTag fct = new FIELDComponentTag();
fct.setText("#{d.DemoUI.name}");
```

In the example the expression is directly passed into the component. So the result is that the component receives the expression “as is”.

If the component is used in a page that is started in the workplace environment, then as result the normal expression update will not be executed, because you defined the expression in a “hard way”.

So, you have to make sure that the newly created component tag receives some information, so that it knows about the current content replace situation:

```
FIELDComponentTag fct = new FIELDComponentTag();
fct.defineContentReplacements(findContentReplaceListDuringRendering());
fct.setText("#{d.DemoUI.name}");
```

By calling “defineContentReplacement” as shown in the code you make sure that now all current content replacements are executed within the new component tag as well.

## Component Attributes - “Mandatory, Preferred - All, Prproposed Values”

Inside the Layout Editor there is a certain arrangement of component attributes.

- “Important” attributes are shown in a list “on the left” - all other attributes are shown in a list “on the right”.
- And: certain attributes can be marked as mandatory.

You can configure this by adding two more configuration files - into the “resources” directory - i.e. the same directory you used for storing your controlsarrangement.xml file.

The configuration files are:

### (A) controlattributeusage.xml

```
<controlattributeusage>
  <tag name="demo:labelfield">
    <attribute name="text"/>
  </tag>

  ...additional tags...

</controlattributeusage>
```

For each component all mandatory attributes are listed.

### (B) controlfavoriteattributes.xml

```
<controlfavoriteattributes>
  <tag name="demo:labelfield">
    <attribute name="text"/>
    <attribute name="width"/>
  </tag>

  ...additional tags...

</controlfavoriteattributes>
```

For each component all the important attributes are listed.

Finally, there is one configuration file that controls the list of proposed values for each attribute (independent from the components that uses the attribute). This list is shown in the layout editor, when the user presses the value help icon of the corresponding attribute:

### (C) controlattributeinfo.xml

```
<controlattributeinfo>
  <attribute name="labeltext" text="Text of label.">
    <attributevalue value="text1" text="Text 1"/>
    <attributevalue value="text2" text="Text 2"/>
    <attributevalue value="text3" text="Text 3"/>
    <attributevalue value="text4" text="Text 4"/>
    <attributevalue value="text5" text="Text 5"/>
  </attribute>
</controlattributeinfo>
```

## Flexible Composite Components and the Grid...

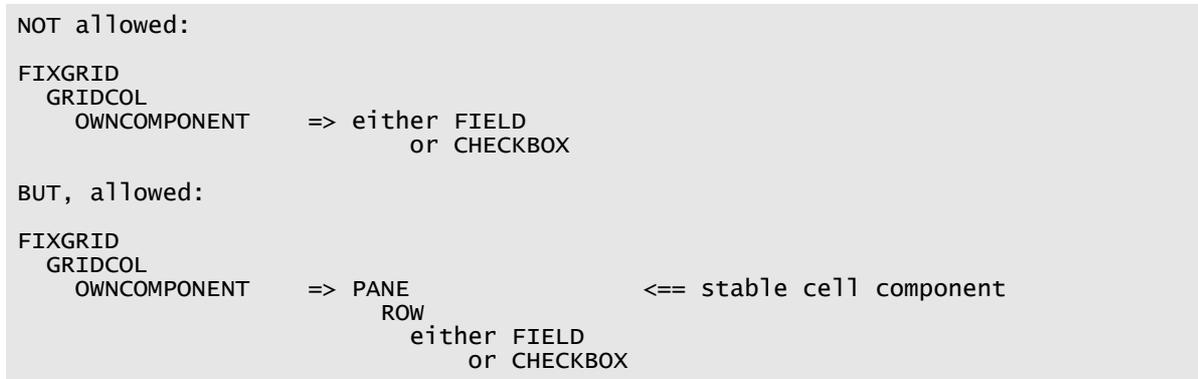
There is one rule within the grid management (FIXGRID component), that is very important: a cell's component needs to be stable throughout the life-cycle of the grid.

Imagine a flexible component that renders its content dependent on certain data. E.g. if the data is a string value then a FIELD component is rendered, if the data is a boolean value then a CHECKBOX component is rendered. (There is an example later on, that exactly demonstrates such a component.)

If this component is used as grid cell (i.e. below GRIDCOL component), then the component must make sure, that its outest content is stable:

- It is not allowed that directly within the grid cell you one time render a FIELD and another time you render a CHECKBOX.
- But: it is allowed to render a PANE into the cell and then, within a ROW, you may either render the FIELD or the CHECKBOX. In this case the outest cell component is the PANE - inside the PANE components may change - but the PANE itself is stable.

To better illustrate:



At runtime you find out if a component is used in a grid, by checking if the parent's parent of the component is of type FIXGRIDComponent:

```
if (getParent().getParent() instanceof FIXGRIDComponent)
{
    // running in grid!
}
```

Before writing you first own component, in which the content is depending from certain data aspects, please check the FLEXFIELD-example before starting your implementation.

### Restarting/Reloading the Toolkit

When defining new component libraries to be part of your project, then you need to restart the CaptainCasa toolkit one time.

---

## Example: “FLEXFIELD” Component

This is an example how to write components, for which the composed content depends on the current data situation.

### Example Screen

Let's start with a demo screen:

Name

Weight

Married

Car	Production Day	Production Day
Car 0	01.01.1990	<input checked="" type="checkbox"/>
Car 1	01.01.1990	100,00
Car 2	01.01.1990	<input checked="" type="checkbox"/>

The layout contains a component FLEXFIELD - that is a demo composite component:

```
<t:rowbodypane id="g_5" rowdistance="5">
  <t:row id="g_6">
    <t:label id="g_7" text="Name" width="100" />
    <demo:flexfield id="g_8" value="#{d.DemoFlexField.name}"
      width="100" />
  </t:row>
  <t:row id="g_9">
    <t:label id="g_10" text="weight" width="100" />
    <demo:flexfield id="g_11" value="#{d.DemoFlexField.weight}"
      width="100" />
  </t:row>
  <t:row id="g_12">
    <t:label id="g_13" text="Married" width="100" />
    <demo:flexfield id="g_14" value="#{d.DemoFlexField.married}"
      width="100" />
  </t:row>
  <t:row id="g_15">
    <t:fixgrid id="g_16" background="#FFFFFF" bordercolor="#00000030"
      borderheight="1" borderwidth="1" multiselect="true"
      objectbinding="#{d.DemoFlexField.grid}" sbvisibleamount="5"
      selectorcolumn="1" width="100%">
      <t:gridcol id="g_17" text="Car" width="50%">
        <demo:flexfield id="g_18" value="{car}" width="100" />
      </t:gridcol>
      <t:gridcol id="g_19" text="Production Day" width="50%">
        <demo:flexfield id="g_20" value="{productionDay}" width="100" />
      </t:gridcol>
      <t:gridcol id="g_21" text="Production Day" width="100%">
        <demo:flexfield id="g_22" value="{comment}" width="100" />
      </t:gridcol>
    </t:fixgrid>
  </t:row>
  <t:row id="g_23">
    <t:button id="g_24" actionListener="#{d.DemoFlexField.onAddItem}"
      text="Add Item" />
    <t:col distance id="g_25" width="5" />
    <t:button id="g_26"
      actionListener="#{d.DemoFlexField.onRemoveItems}"
      text="Remove Item" />
  </t:row>
</t:rowbodypane>
```

All the field/ check box/ calendar field components that you see on the screen are generated by the FLEXFIELD component. The component checks the data type of the corresponding property value and renders a corresponding component.

You see, that the component is used both directly within the page - and is used within a grid, as grid cell.

The code of the backing bean is quite straight forward:

```
package workplace;

import java.io.Serializable;
import java.math.BigDecimal;
import java.util.Calendar;
import java.util.Date;
import java.util.TimeZone;

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

```

import org.eclnt.editor.annotations.CCGenClass;
import org.eclnt.jsfserver.elements.impl.FIXGRIDItem;
import org.eclnt.jsfserver.elements.impl.FIXGRIDListBinding;
import org.eclnt.workplace.IworkpageDispatcher;
import org.eclnt.workplace.workpageDispatchedBean;

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

public class DemoFlexField
    extends workpageDispatchedBean
    implements Serializable
{
    public class GridItem extends FIXGRIDItem implements java.io.Serializable
    {
        protected Date m_productionDay;
        public Date getProductionDay() { return m_productionDay; }
        public void setProductionDay(Date value) { m_productionDay = value; }

        protected String m_car;
        public String getCar() { return m_car; }
        public void setCar(String value) { m_car = value; }

        Object m_comment;
        public void setComment(Object value) { m_comment = value; }
        public Object getComment() { return m_comment; }
    }

    protected Boolean m_married = false;
    protected BigDecimal m_weight = new BigDecimal("10");
    protected String m_name = "Name";
    protected FIXGRIDListBinding<GridItem> m_grid = new
FIXGRIDListBinding<GridItem>();

    public DemoFlexField(IworkpageDispatcher workpageDispatcher)
    {
        super(workpageDispatcher);
        for (int i=0; i<3; i++)
        {
            GridItem gi = new GridItem();
            gi.setCar("Car " + i);
            try
            {
                Calendar c = Calendar.getInstance(TimeZone.getTimeZone("UTC"));
                c.set(Calendar.YEAR,1990);
                c.set(Calendar.MONTH,Calendar.JANUARY);
                c.set(Calendar.DAY_OF_MONTH,1);
                gi.setProductionDay(c.getTime());
            }
            catch (Throwable t) {}
            if (i%2 == 0)
                gi.setComment(new Boolean(true));
            else
                gi.setComment(new BigDecimal("100"));
            m_grid.getItems().add(gi);
        }
    }

    public FIXGRIDListBinding<GridItem> getGrid() { return m_grid; }

    public Boolean getMarried() { return m_married; }
    public void setMarried(Boolean value) { m_married = value; }

    public BigDecimal getWeight() { return m_weight; }
    public void setWeight(BigDecimal value) { m_weight = value; }

    public String getName() { return m_name; }
    public void setName(String value) { m_name = value; }

    public void onRemoveItems(ActionEvent event)
    {
        for (GridItem gi: m_grid.getSelectedItems())
            m_grid.getItems().remove(gi);
    }

    public void onAddItem(ActionEvent event)
    {
        GridItem gi = new GridItem();

```

```

        gi.setCar("<new>");
        gi.setProductionDay(new Date());
        gi.setComment("Comment");
        m_grid.getItems().add(gi);
    }
}

```

## Component Implementation

The FLEXFIELD component is defined in its tag library definition in the following way:

```

...
<tag>
  <name>flexfield</name>
  <tag-class>democontrols.FLEXFIELDComponentTag</tag-class>
  <attribute><name>id</name></attribute>
  <attribute><name>value</name></attribute>
  <attribute><name>width</name></attribute>
</tag>
...

```

The implementation of the Tag-class is quite simple, because all the attributes are contained in the parent implementation already:

```

package democontrols;

import org.eclnt.jsfserver.elements.BaseComponentTag;

public class FLEXFIELDComponentTag extends BaseComponentTag
{
}

```

The most interesting part is the implementation of the component itself:

```

package democontrols;

import java.io.IOException;
import java.math.BigDecimal;
import java.util.Date;

import javax.faces.context.FacesContext;
import javax.faces.context.ResponseWriter;
import javax.servlet.http.HttpSession;

import org.eclnt.jsfserver.elements.BaseActionComponent;
import org.eclnt.jsfserver.elements.BaseComponent;
import org.eclnt.jsfserver.elements.ComponentDump;
import org.eclnt.jsfserver.elements.impl.CALENDARFIELDComponentTag;
import org.eclnt.jsfserver.elements.impl.CHECKBOXComponentTag;
import org.eclnt.jsfserver.elements.impl.FIELDComponent;
import org.eclnt.jsfserver.elements.impl.FIELDComponentTag;
import org.eclnt.jsfserver.elements.impl.FIXGRIDComponent;
import org.eclnt.jsfserver.elements.impl.FORMATTEDFIELDComponentTag;
import org.eclnt.jsfserver.elements.impl.LABELComponentTag;
import org.eclnt.jsfserver.elements.impl.PANECComponentTag;
import org.eclnt.jsfserver.elements.impl.ROWComponent;
import org.eclnt.jsfserver.elements.impl.ROWComponentTag;
import org.eclnt.jsfserver.util.ExpressionManagerV;
import org.eclnt.jsfserver.util.HttpSessionAccess;

public class FLEXFIELDComponent
  extends BaseActionComponent
{
    // -----
    // members
    // -----

    transient Class m_lastValueClass = null;
    transient Boolean m_isInGrid = null;

    // -----

```

```

// public usage
// -----

public void encodeBegin(FacesContext context) throws IOException
{
    ensureSessionIsValid(HttpSessionAccess.getCurrentHttpSession(context));
    if (m_isInGrid == null)
    {
        if (getParent().getParent() instanceof FIXGRIDComponent)
            m_isInGrid = true;
        else
            m_isInGrid = false;
    }
    // pick data from object
    String valueExpression = getAttributeString("value");
    Object valueObject =
ExpressionManagerV.getValueForExpressionString(context,valueExpression);
    String width = getAttributeString("width");
    // in case of grid usage => create stable outest component
    BaseComponent contentCell = null;
    if (getChildren().size() == 0)
    {
        if (m_isInGrid == true)
        {
            PANESComponentTag pt = new PANESComponentTag();
            BaseComponent p = pt.createBaseComponent();
            ROWComponentTag rt = new ROWComponentTag();
            BaseComponent r = rt.createBaseComponent();
            getChildren().add(p);
            p.getChildren().add(r);
            contentCell = r;
        }
        else
        {
            contentCell = this;
        }
    }
    else
    {
        if (m_isInGrid == true)
        {
            contentCell =
(BaseComponent)getChildren().get(0).getChildren().get(0);
        }
        else
        {
            contentCell = this;
        }
    }
    // check if control update is required
    Class valueClass = null;
    if (valueObject != null) valueClass = valueObject.getClass();
    if (m_lastValueClass != valueClass) contentCell.getChildren().clear();
    m_lastValueClass = valueClass;
    // render control
    if (contentCell.getChildren().size() == 0 && valueObject != null)
    {
        if (m_isInGrid) width = "100%";
        BaseComponent fieldComponent =
buildFieldContent(context,valueExpression,valueObject,width);
        contentCell.getChildren().add(fieldComponent);
    }
}

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

private BaseComponent buildFieldContent(FacesContext context,
String valueExpression,
Object valueObject,
String width)
{
    if (valueObject instanceof String)
    {
        FIELDComponentTag tag = new FIELDComponentTag();
        tag.setText(valueExpression);
        if (width != null) tag.setWidth(width);
    }
}

```



# Page Bean Components

Page Beans are a comfortable and simple way of encapsulating a screen as object and re-use it throughout various parts of your application. (Please read details in the chapter “Page Navigation”.)

A page bean consists out of...:

- The Java-logic of one page
- The .jsp-page containing the XML layout definition

Through being an excellent modularization technology within one project, page beans are not really distributable throughout various projects in a simple way. Imagine you want to use one page bean implementation, that you made in one project, in another project as well. In this case you have to copy various issues from one project to the other:

- the .class/.jar file containing the logic of the page bean-level
- the .jsp file containing the layout
- the .properties files that may contain text literals
- various .png/.jpg/... files that are referenced within the page bean

---

## Basic Idea - One self-containing JAR File

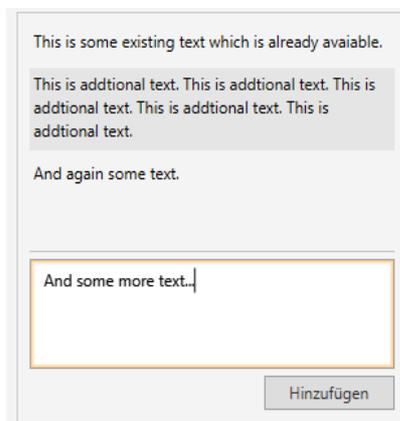
So the concept behind Page Bean Components is:

- Keep the normal, approved Page Bean processing,...
- ...but allow “everything” to be arranged inside a .jar file and thus make it simple to distribute the page bean component across projects
- ...and allow a nicer integration into the CaptainCasa toolset (e.g. by letting the developer configure the page bean component from the Layout Editor)

By adding the JAR file to your project (e.g. as JAR within WEB-INF/lib) you can use the page bean component - and do not have to think about cross-copying additional files in addition.

### Example

In the following text we will explain the Page Bean Component concept by using an example - the “multi text” component.



The screenshot shows a web form with the following content:

- Text: "This is some existing text which is already available."
- Text area (shaded): "This is additional text. This is additional text. This is additional text. This is additional text. This is additional text."
- Text: "And again some text."
- Text area (bordered): "And some more text..."
- Button: "Hinzufügen"

The component displays/edits a list of text-strings. Each time the user adds some new text and presses “Add”, the new text is added to the list of texts on top of the input text area.

The component is part of the demo workplace (demos-project). It is contained in package “democontrols”. The class name of the component is “DemoPBCMultiText”.

---

## Using a Page Bean Component

### Component PAGEBEANCOMPONENT

There is a component PAGEBEANCOMPONENT that allows to use a page bean component:

```
<t:row id="g_2">
  <t:pagebeancomponent id="g_3"
    pagebeanbinding="#{d.DemoPageBeanComponent.multiText}"
    pagebeanclass="democontrols.DemoPBCMultiText"
    pagebeaninitdata="height:300;width:300" />
</t:row>
```

There are three important attributes:

- The PAGEBEANBINDING is the expression that points to an instance of the page bean component inside your server side class.
- The PAGEBEANCLASS is a hint for the Layout Editor toolset - By knowing the class, the tool environment can access some meta data about the page bean component.
- the PAGEBEANINITDATA is a complex value string, that is passed into the page bean component when it is first time accessed. Each page bean component may specify a set of initialization parameters - typically configuring optical attributes of the page bean component.

The server side code for integrating the page bean component inside your class is:

```
package workplace;

...

public class DemoPageBeanComponent
  extends PageBean
{
    DemoPBCMultiText m_multiText = new DemoPBCMultiText();

    public DemoPageBeanComponent(IWorkpageDispatcher workpageDispatcher)
    {
        super(workpageDispatcher);

        // filling page bean with data + passing listener
        List<String> texts = new ArrayList<String>();
        texts.add("This is some existing text which is already available.");
        m_multiText.prepare(texts, new DemoPBCMultiText.IListener()
        {
            public void reactOnUpdate()
            {
                // ...
            }
        });
    }

    ...

    public DemoPBCMultiText getMultiText() { return m_multiText; }

    ...
}
```

You see: page bean components are embedded into you page and page processing in the

same way as normal page beans.

---

## Developing a Page Bean Component - By Example

Let's implement the "multi text" component as component "DemoPBCMultiText" in package "democontrols".

### The basic Files

The following files are contained in the package "democontrols":

```
democontrols
  DemoPBCMultiText.xml           => The layout
  DemoPBCMultiText.java         => The code
  DemoPBCMultiText.properties   => Literals
  DemoPBCMultiText_de.properties => Literals in German
  DemoPBCMultiText.config       => XML configuration
```

### The XML Layout

The "DemoPBCMultiText.xml" contains the layout definition. This is the same as a normal .jsp file - but now you only have to refer to the XML content of the .jsp file - not the JSF prologue.

In the example the following XML is used:

```
<t:pane rowdistance="5" border="#00000030" padding="10" width="{pb.width}"
height="{pb.height}">
  <t:row>
    <t:scrollpane width="100%" height="100%">
      <t:rowdynamiccontent contentbinding="{pb.dynContent}" />
    </t:scrollpane>
  </t:row>
  <t:rowline />
  <t:row>
    <t:textarea width="100%;100" height="80" text="{pb.newText}"
      bgpaint="{pb.newTextBgpaint}" requestfocus="{pb.newTextRequestFocus}"
    />
  </t:row>
  <t:row>
    <t:coldistance width="100%" />
    <t:button text="{pb.lit.btnAdd}" width="1%;100"
      actionListener="{pb.onAddAction}" />
  </t:row>
</t:pane>
```

You see: just normal "layout XML"... Please note:

- You may use any expression for referencing the page bean inside the XML. In the example the root expression is "{pb}". But you may also use "{d.Whatever}" when creating the page bean by copying it from an existing page bean. - There are no restrictions!
- In the example there are no ids assigned to the components - you may use ids, of course. You may use "g\_"-ids if you copied the XML from an existing .jsp file. Or you may not defined any ids at all (as in the example).
- You may already see in the example that literals are referenced via "{pb.lit.\*}" - directly being resolved using the property files that are part of the package.

### The Java Code

The "DemuPBCMultitext.java" file contains the code of the page bean component:

```

package democontrols;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;

import javax.faces.event.ActionEvent;

import org.eclnt.jsfserver.elements.componentnodes.PANENode;
import org.eclnt.jsfserver.elements.componentnodes.ROWNode;
import org.eclnt.jsfserver.elements.componentnodes.TEXTPANENode;
import org.eclnt.jsfserver.elements.impl.ROWDYNAMICCONTENTBinding;
import org.eclnt.jsfserver.pagebean.component.PageBeanComponent;
import org.eclnt.jsfserver.session.RequestFocusManager;

public class DemoPBCMultiText extends PageBeanComponent
{
    // -----
    // inner classes
    // -----

    public interface IListener
    {
        public void reactOnUpdate();
    }

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

    String m_width = "200";
    String m_height = null;

    String m_newText = null;
    List<String> m_texts = new ArrayList<String>();

    IListener m_listener;
    ROWDYNAMICCONTENTBinding m_dynContent = new ROWDYNAMICCONTENTBinding();
    boolean m_error = false;
    long m_newTextRequestFocus;

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

    @Override
    public String getRootExpressionUsedInPage() { return "#{pb}"; }

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

    public String getNewText() { return m_newText; }
    public void setNewText(String newText) { m_newText = newText; }

    public List<String> getTexts() { return m_texts; }

    public void setWidth(String width) { m_width = width; }
    public String getWidth() { return m_width; }
    public void setHeight(String height) { m_height = height; }
    public String getHeight() { return m_height; }

    public ROWDYNAMICCONTENTBinding getDynContent() { return m_dynContent; }

    @Override
    public void initializePageBean(Map<String, String> initData)
    {
        super.initializePageBean(initData);
        {
            String s = initData.get("width");
            if (s != null) setWidth(s);
        }
        {
            String s = initData.get("height");
            if (s != null) setHeight(s);
        }
        render();
    }
}

```

```

public void prepare(List<String> texts, IListener listener)
{
    m_texts = texts;
    m_newText = null;
    m_listener = listener;
    render();
}

public String getNewTextBgpaint()
{
    String result = "write_empty(10,50%,"+getLit().get("phText")
+",10,#c0c0c0,leftmiddle)";
    if (m_error) result += ";error()";
    return result;
}

public long getNewTextRequestFocus() { return m_newTextRequestFocus; }

public void onAddAction(ActionEvent event)
{
    m_error = false;
    if (m_newText == null || m_newText.trim().length() == 0)
    {
        m_error = true;
        m_newTextRequestFocus =
RequestFocusManager.getNewRequestFocusCounter();
        return;
    }
    m_texts.add(m_newText);
    m_newText = null;
    m_newTextRequestFocus = RequestFocusManager.getNewRequestFocusCounter();
    render();
    if (m_listener != null)
        m_listener.reactOnUpdate();
}

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

private void render()
{
    PANENode p = new
PANENode().setWidth("100%").setHeight("100%").setRowdistance(5);
    int counter = -1;
    for (String text: m_texts)
    {
        counter++;
        ROWNode r = new ROWNode();
        p.addSubNode(r);
        TEXTPANENode t = new TEXTPANENode().setWidth("100%").setText(text);
        if (counter%2 == 1)
            t.setBackground("#00000010");
        r.addSubNode(t);
    }
    m_dynContent.setContentNode(p);
}
}

```

Again you see: a just normal page bean implementation... - with some special remarks:

- The method “getPageName()” which you normally have to implement, does not need to be implemented for page bean components. The page bean processing just knows that the page bean component’s XML resided in a .xml file with the same name as the class name.
- There is a method “initializePageBean()” passing a String-Map of data. This method allows to pass initialization information that can be defined when embedding a page bean component in a page. It allows the user of a page bean to configure visual aspects within the layout editor.

## The Property Files

Literals are kept in property files. The page bean component author decides which languages to support. In the example there are two files:

```
File: DemoPBCMultiText.properties
btnAdd = Add
phText = Please enter some text...

File: DemoPBCMultiText_de.properties
btnAdd = Hinzufügen
phText = Bitte geben Sie einen Text ein...
```

## The Configuration File

The configuration file “DemoPBCMultiText.config” passes information into the Layout Editor environment to support the user of a page bean component when embedding the component into a page. The file is NOT used operationally at runtime at the moment.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<pageBeanConfig>
  <columnComponent>true</columnComponent>
  <param>
    <name>width</name>
  </param>
  <param>
    <name>height</name>
  </param>
</pageBeanConfig>
```

There are the following definitions:

- The page bean component declares via tag “columnComponent” if it is used as column component (“true”) or as row component (“false”)
- All parameters that are interpreted in the “initializePageBean()” method are listed.

---

## Packaging and Delivering a Page Bean Component

Packaging and delivering a page bean component means that you just have to package all artifacts into one JAR file. This JAR file then contains the following files:

```
democontrols
  DemoPBCMultiText.xml           => The layout
  DemoPBCMultiText.class         => The byte code
  DemoPBCMultiText.properties   => Literals
  DemoPBCMultiText_de.properties => Literals in German
  DemoPBCMultiText.config       => XML configuration
```

Of course you are “allowed” to add further class files in order to split the page bean component’s functions into several classes. Of course you may bundle several page bean components into one JAR file.

The user of a page bean just has to add the JAR file to his/her server side processing - typically by adding it to WEB-INF/lib.

# File Download & File Upload

There are a couple of components that provide the possibility to upload/download files from/to the user's client. The demo workplace contains a number of examples that demonstrate what you can do - please take implementation details from there.

---

## File Download

The following components all trigger the download of a certain file to the client side.

- FILEDOWNLOADBUTTON  
FILEDOWNLOADLINK
- FILEDOWNLOAD

While FILEDOWNLOADBUTTON and FILEDOWNLOADLINK are visible components, that trigger the download when the user presses the button/link, FILEDOWNLOAD is an invisible component that start the download via a trigger-attribute.

Once triggered a corresponding popup will be opened for the user, so that the user can select/change the location of the file to be downloaded.

The download itself is done through a URL-attribute that is defined as attribute of the component. The URL should point into your web application. The client opens up the URL, reads the bytes by a corresponding http-request and stores the bytes within the defined file on client side.

### Static URLs - Static Content

Well, what we tell now sounds very easy (and it is), but it will not meet your real life requirements... Never the less it maybe useful for certain situations.

```
...  
<t:filedownloadbutton url="/images/car.png" filename="car.png"/>  
...
```

The URL `"/images/car.png"` point to a static file `"car.png"` within the `"images"` directory of your web application. The file name on client side, that will be proposed to the user is `"car.png"` as well.

You see, it is very easy to download static content that is part of your web application. BUT: you should never use this way of accessing information in order to download dynamic content!

Example: you could define a directory `"download"` within your web application, and then write some temporary files into the download directory, that you then dynamically pass as URL into the download-component.

Why you should never do this: first of all some applications server (different to Tomcat), do not open up a web-application-directory at all - but store the web application somehow in internal files. As consequence there is no directory to write to.

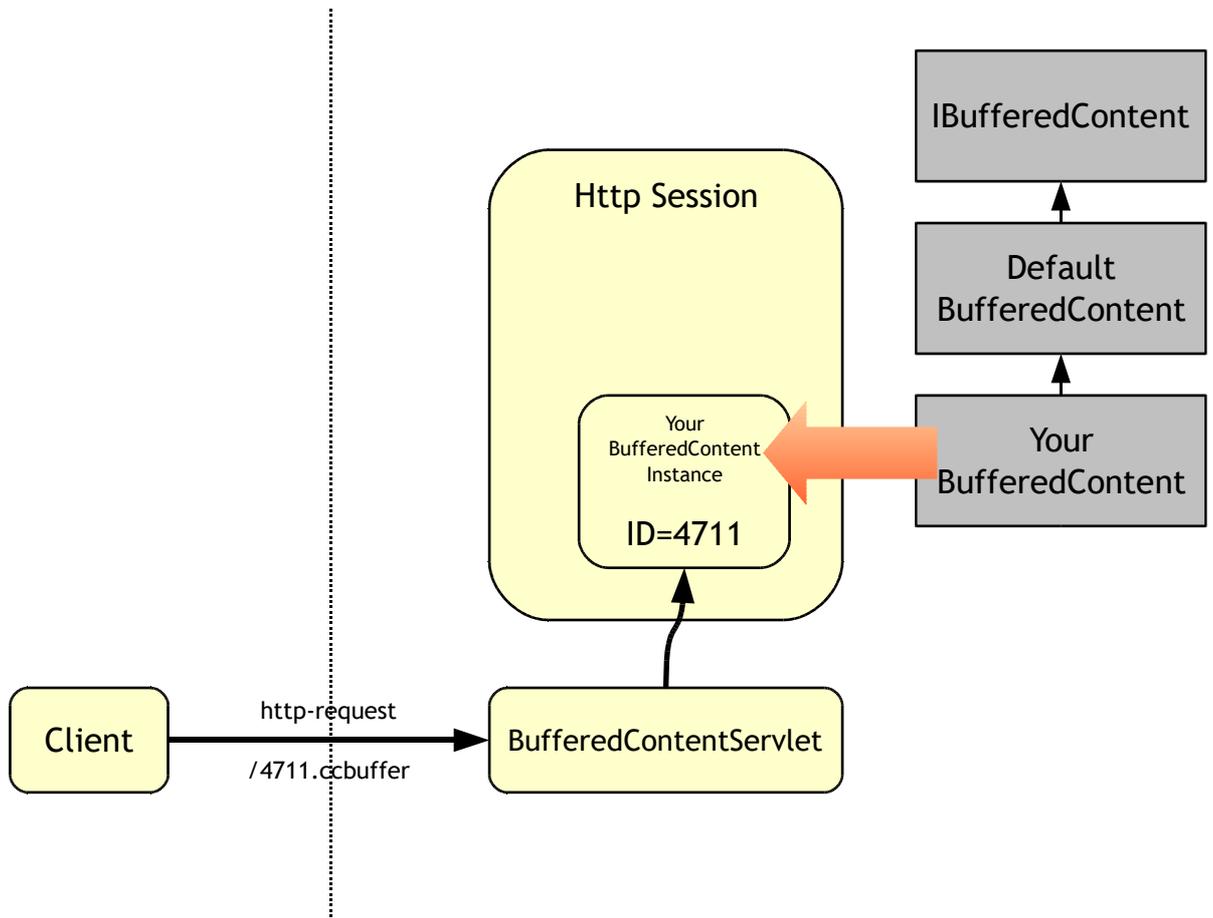
And: in clustered scenarios requests potentially may hit different machines. So the client, resolving the URL `"/download/temp1234"` may hit a different machine than the one were you wrote the file to.

In short: writing information into the web application directory is `"dirty programming"` and that's what you should not do. There are much nicer ways, in addition - please have a look into the next chapter.

## Dynamic URLs - Dynamic Content

The solution to provide dynamic content via URL is quite simple: instead of passing back a static URL, a URL is passed back that invokes a certain servlet, that then itself triggers the passing of the dynamic content. A consequence the request that is sent from client side when the download is triggered hits a dynamic program structure rather than a static file system.

In order to simplify this dynamic, servlet based loading of data there are already some structures defined within the server side processing of CaptainCasa Enterprise Client.



There is an interface “IBufferedContent” and a default implementation “DefaultBufferedContent”. The method that you need to define in your implementation is the following:

```
class YourBufferedContent extends DefaultBufferedContent
{
    public byte[] getContent()
    {
        ...
        // load the content here
        ...
    }
}
```

When creating an instance of this class you need to register this instance within the so called buffered-content-manager:

```
YourBufferedContent bc = new YourBufferedContent();
BufferedContentMgr.add(bc);
```

As part of the creating of the instance and of registering, the instance has received a unique id and is registered under this id within the session context.

From the instance you can now load a URL - the URL contains all the parameters that are required later on to find the instance from the BufferedContentServlet, shown in the graph above. The instance's URL may for example be:

```
BUFFERED_4711.ccbuffer;jsessionid=dfk34823544539
```

This URL, that you receive from the instance is “perfectly made” for being passed as URL into the download components.

When the user triggers the download, then the URL is requested from client side - the instance is caught up by accessing the session context and the instance's method “getContent()” is called. The result of “getContent()” is passed back as response to the client side - and stored as local file over there.

## Big Dynamic Content Scenarios

When applying the mechanism explained in the previous chapter to scenarios, in which you may want to download megabytes of data, then there is one problem: you have to pass back the content as byte-array. This means, that the whole megabyte-content is kept in memory on server side for a certain duration of time.

As consequence there is a second interface mechanism, that exactly operates the same way as already described with IBufferedContent in the previous chapter - but now not loading the content as byte-array, but now by letting you pass the content into an output stream.

```
class YourBufferedContent extends DefaultBufferedStreamContent
{
    public void writeStream(OutputStream stream)
    {
        ...
        ...
        // provide content and write into stream
        ...
        ...
    }
}
```

## Synchronous - Asynchronous

Each component for downloading information from the server to the client provides an attribute ASYNCHRONOUS. By default the download is done in a synchronous way - i.e. the user has to wait with further processing on client side until the download is finished. By setting ASYNCHRONOUS to “true” you can decouple the downloading from the normal request/response processing of the Enterprise Client.

## Opening the File at Client Side

Typically a file is downloaded on client side, the corresponding download popup show the status of the download (bytes received etc.) and then after successful download automatically is closed.

By defining the attribute OPENSUPPORTED to be true, you can change this behavior:

- The download popup will be kept open after successful download.
- A button “Open” will be enabled.

Once pressing the “open” button the client will use the Java Desktop functions in order to start the right program to open the file.

By using this feature you e.g. can dynamically create office documents (spreadsheets etc.) on server side, download them to the client side and immediately open them after download.

---

## File Upload

For uploading files there are a couple of components as well:

- FILEUPLOAD - synchronous upload of files, component looks like a combo box
- FILEUPLOADLINK, FILEUPLOADLINKASYNCHRONOUS - synchronous/asynchronous upload of files, component is rendered as link
- FILEUPLAODBUTTON, FILEUPLOADBUTTONASYNCHRONOUS - synchronous/ asynchronous upload of files, component looks like a button
- FILEUPLOADASYNCHRONOUS - asynchronous upload of files, component is not visible to the user - it can be triggered by server side processing.

The components provide a different way of rendering, but all operate the same way, when it comes to uploading client side file information.

There are two different usage modes that can be applied:

- Synchronous Upload
- Asynchronous Upload

### Synchronous Upload

This is the default mode. When the user triggers the upload (e.g. by pressing the button of FILEUPLAODBUTTON), then a file selection is popped up. After selecting the file the content will be uploaded as just normal request that is coming from the client.

Consequently the action listener that is associated with the component is called on server side, and the file information (client file name and file content) is passed through the “BaseActionEventUpload”.

Advantage: the processing of the server side is “in sync” with the upload processing: e.g. you might directly process the uploaded content within the follow on processing.

There are two disadvantages of synchronous upload components that you need to pay attention to, especially when dealing with bigger file sizes (starting at “some megabytes”):

- Because the content of the file is part of the normal data request processing, there is no feedback to the user, that the request processing will now take long(er). The user just sees some hour glass and e.g. does not get any feedback about how many bytes of the file are currently uploaded. The user also cannot interrupt the upload processing - all the client operation is blocked until the file is completely sent to the server side.
- The content of the file needs to be converted into some string (here we use a simple hex-byte representation) in order to be embedded into the normal http request protocol (“name1=value1&name2=value2&name3=value3...”). This means that the whole file is loaded within the client processing (e.g. 1MB of byte[] size), is converted into a hex-string (2 Mill. characters, 4 MB of char[] size) and in some situations kept twice

temporarily (e.g. when appending it to a string buffer).

Consequence: use the synchronous upload for “not-too-big” file sizes and if there is a certain sequence of processing on server side that you want to enforce. For files which are bigger than 5MBs please use asynchronous upload components, especially because of memory considerations.

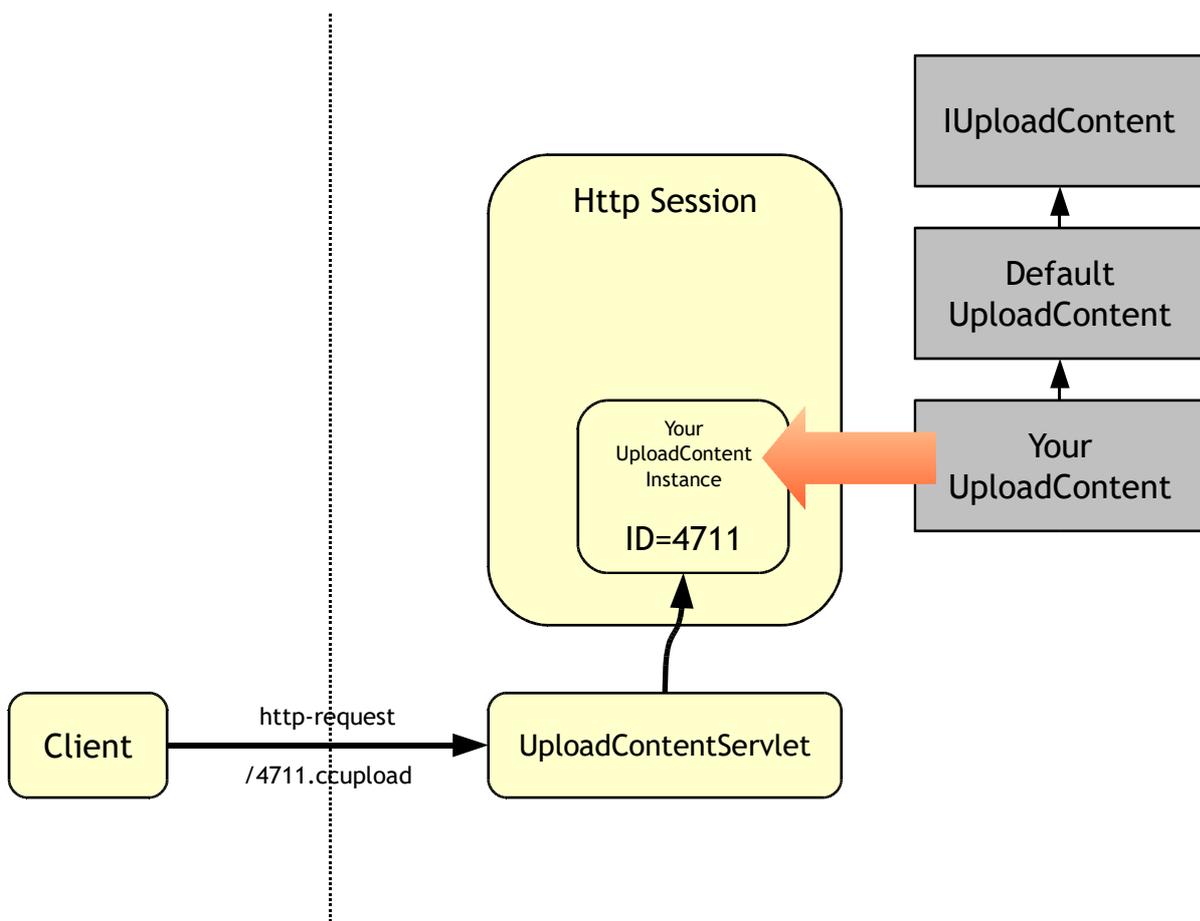
## Asynchronous Upload

This mode is invoked when specifying the attribute ASYNCHRONOUSUPLOADURL within the components.

Now the file is uploaded to a URL in parallel to the normal request/response processing of the Enterprise Client. The URL needs to point to a servlet processing that processes the file content on server side.

Very similar to the servlet management that is used for downloading dynamic content (please take a look into this chapter), there is an interface/object structure available that simplifies the server side processing when receiving an upload request:

You either implement the interface IUploadContent (by sub-classing from DefaultUploadContent) or you implement the interface IUploadStreamContent (by sub-classing from DefaultUplaodStreamContent).



The same structure is applied when using interface “IUploadStreamContent”.

```
public class MyUploadContent extends DefaultUploadContent
{
    public void beginPassing()
    {
```

```

        m_content = "Upload started.\n";
    }
    public void passClientFile(String fileName, byte[] bytes)
    {
        m_content += "File was passed: client name " +
            fileName + ", length " + bytes.length + "\n";
    }
    public void endPassing()
    {
        m_content += "Upload ended.";
    }
}

public class MyuploadStreamContent extends DefaultUploadStreamContent
{
    public void beginPassing()
    {
        m_content = "Stream upload started.\n";
    }
    public void passClientFilesAsStream(InputStream stream)
    {
        try
        {
            StringBuffer sb = new StringBuffer();
            while (true)
            {
                int b1 = stream.read();
                if (b1 < 0) break;
                char c = (char)b1;
                sb.append(c);
            }
            m_content += sb.toString()+ "\n";
        }
        catch (Throwable t)
        {
            CLog.L.log(CLog.LL_ERR,
                "error when receiving file from client",t);
        }
    }
    public void endPassing()
    {
        m_content += "Stream upload ended.";
    }
}

```

When implementing your own classes to manage upload requests, you need to implement these methods that actually transfer the data from the upload-request into the application:

- beginPassing()
- passClientFile() for IUploadContent, passClientFilesAsStream() for IUploadStreamContent
- endPassing()

When creating instances you need to register these instances, so that they are parked within the http session context on server side. The registration is done in the following way:

```

// creating instances + register
MyUploadContent muc = new MyUploadContent();
uploadContentMgr.add(muc);
MyUploadStreamContent musc = new MyUploadStreamContent();
uploadContentMgr.add(musc);
// assign URL of instances to property-members that are
// references by the upload components
m_uploadURL = muc.getURL();
m_uploadStreamURL = musc.getURL();

```

The URL that you provide to be used as ASYNCHRONOUSUPLOADURL is taken out of the “getURL()” method of your instances.

Please read the JavaDoc in order to find more details about the usage of the interfaces, e.g. about the format of the stream when using IUploadStreamContent.

## Asynchronous Upload - Notification of “Finished”!

After having successfully uploaded a file/ some files the client side upload processing will trigger the action listener that is bound within the corresponding upload component. The event that is passed into the action listener is of class “BaseActionEventUploadAsynchronousFinished”.

## Garbage Collection Issues

The registration of “IUploadContent” and “IUploadStreamContent” objects into the “UploadContentMgr” means that a central session instance now keeps pointers to the instances that you pass.

You need to careful think about when to un-register your instances - otherwise your server side memory will not garbage collect your objects. Un-registering is done by calling the “UploadContentMgr.remove(...)”-method.

There are two techniques how to un-register:

- The first one is “obvious”: if you have a certain point of time within your application prcessing, at which you know that you can release the instances, then you need to call the remove(...)-method. - When using CaptainCasa's workplace management, then pages are running in the context of so called workpages. Workpages provide a lifecycle listener interface, so your code might look like:

```
m_uploadContent = new MyUploadContent();
UploadContentMgr.add(m_uploadContent);
m_uploadStreamContent = new MyUploadStreamContent();
UploadContentMgr.add(m_uploadStreamContent);
getworkpage().addLifecycleListener(new
workpageDefaultLifecycleListener()
{
    public void reactOnDestroyed()
    {
        super.reactOnDestroyed();
        UploadContentMgr.remove(m_uploadContent);
        UploadContentMgr.remove(m_uploadStreamContent);
    }
});
```

- Another technique is to always re-new the objects with each request. For removing existing instances you may use the PhaseManager (a utility class of CaptainCasa):

```
IUploadContent m_uploadContent;

public class RemoveUploadContent implements Runnable
{
    public void run()
    {
        if (m_uploadContent != null)
        {
            UploadContentMgr.remove(m_uploadContent);
            m_uploadContent = null;
        }
    }
}

public String getUploadContentURL()
{
    if (m_uploadContent == null)
    {
        m_uploadContent = new ...;
        UploadContentMgr.add(m_uploadContent);
        PhaseManager.runBeforeUpdatePhase(new RemoveUploadContent());
    }
}
```

```
} return m_uploadContent.getURL();
```

Result: the registration of an object is only kept for a certain point of time (until next request is processed).

## Special Directory “\${local}/” or “{local}/” - Automated Download

Some times you may want to download and upload files without the user explicitly having to confirm the file name and without being explicitly interrupted. Example: your application does want to store certain client side files in order to process these files by other client side applications.

In this case you can define the file name of the file to be down/uploaded in the following way: “\${local}/xxx/abc.txt”, or “{local}/xxx/abc.txt”.

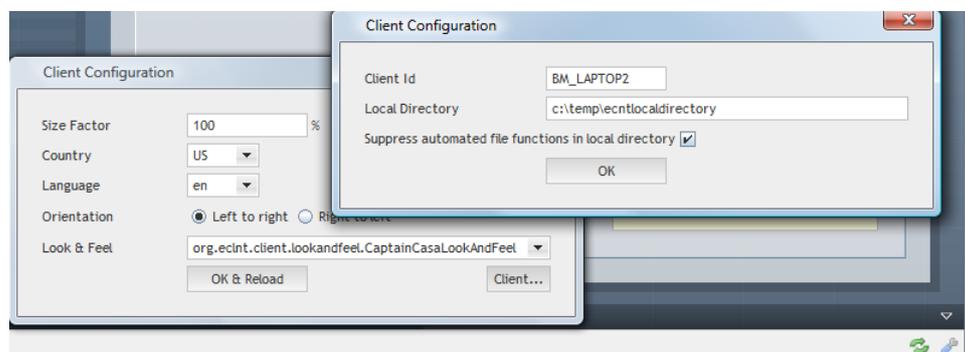
Of course directly download and uploading files from the client is something that does not sound too healthy from security point of view: the user needs to trust you application that it does not grab some client side private files and upload them to the server side.

As consequence there is a certain concept that meets the interests of automation on the one side and the interests of security and privacy on the other side. By default automated file transfer can only happen within a certain so called “local directory” on the user’s client.

The strategy to find the local directory is:

- The user may explicitly define the local directory within his/her client side environment. To do so the user has to open the client configuration dialog and define the name of the directory (see screen-shot below).
- If the user does not define an explicit local directory, then the system find it on its own, using the temporary directory, that is assigned by the operating system to the current user.
- The user can in addition define that he/she does not want to take part in the upload/download automation at all.

The configuration of the client side “local directory” is done within the configuration area of the client:



The configuration is stored on client side within the temporary directory that is assigned by the operating system. The information about “local directory” is not passed to the server side.

In case the application want to use automated functions for up and download it in

addition hat to tell so by defining the attribute LOCALFILEMODE, that is available on all FILEDOWNLOAD\* (and all FILEUPLOAD\*) components. There are the values

- “dark” and
- “confirm” (only download)

that can be passed. In case of “dark” the user just sees a notification about the download - in case of “confirm” the user needs to confirm the download into the local directory, but missing directories will be automatically created.

---

## Special Directory “\${temp}/” or “{temp}/”

As part of the FILEDOWNLOAD\* and FILEUPLOAD\* components you may pre-define a file name that is used on client side. You can use the prefix “\${temp}/” or “{temp}/” to indicate that you want to download the file in the temporary directory of you client. The location of the temporary directory depends on the client's operating system.

In this case you can define the file name of the file to be down/uploaded in the following way: “\${local}/xxx/abc.txt”, or “{local}/xxx/abc.txt”.

Please pay attention: the slash of “\${temp}/” or “{temp}/” must always be a forward slash (“/”), also in cases where you know that there is a Windows operating system on client side!

---

## Other Components around Client Side File Management

### FILECHOOSE - Selecting + passing the Name of a Client Side File

By using the component FILECHOOSE you can let the user input/select the name of a client side file or directory. This component just transfers the name, not the content.

### FILECREATEDIRECTORY - Creating a directory on Client Side

With this component you can make sure that a certain file exists on client side. If it does not then the user is asked if the client should create it.

Of course this component is quite useful, e.g. if you want the user to have certain directory structures in which you later on want to download further file information. But: you of course need to be aware of that the server side processing in principal does not know anything about the client side file structure.

In other words: either you have very homogeneous file structures and client devices (e.g. you expect a “C:\”-drive to be available on any machine...) or you have a quite sophisticated server side logic, that knows details about the client side. - Just to mention: in inhomogeneous scenarios (some Windows-clients, some Linux-clients) it may be quite difficult to assume, a “C:\” drive is available on any machine...

---

## Configuration Issues

When upload client side information, then you will typically exceed limits that are defined for “normal request processing”. Most typical, there is a “max request size” definition somewhere in your application server configuration settings...

### Tomcat

You need to configure the connector in tomcat/conf/server.xml to allow post-request-

sizes greater than 1 MB.

```
<Connector port="50000" protocol="HTTP/1.1"  
connectionTimeout="20000"  
redirectPort="50001"  
maxPostSize="-1">
```

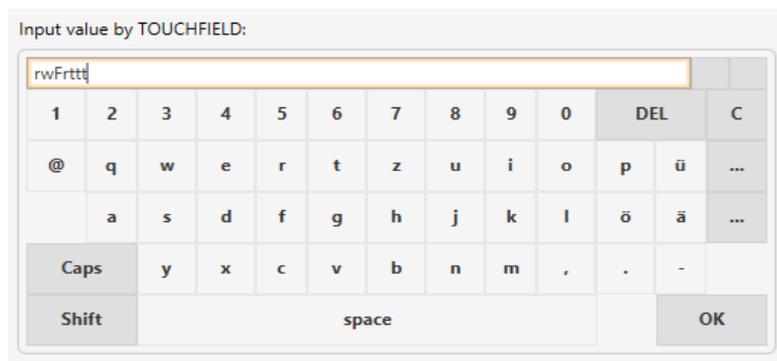
By defining a maxPostSize of “-1” any request sizes are allowed to be uploaded.  
Please read further details within the documentation available with Tomcat.

# Touch Input

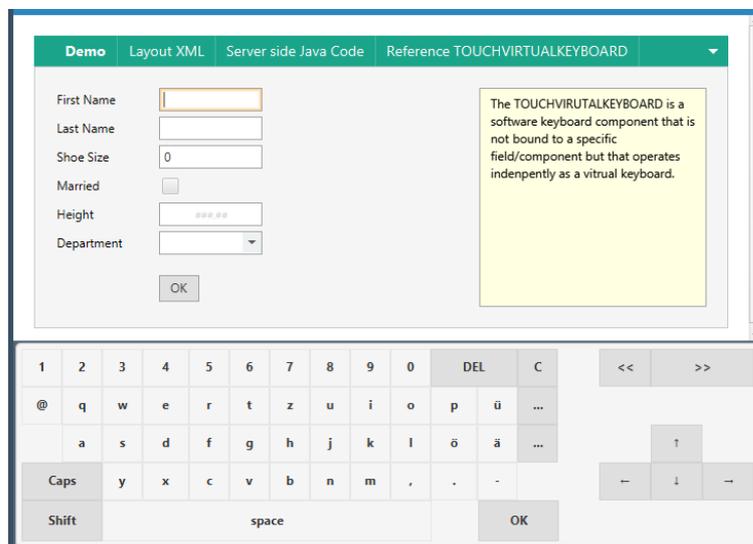
## Components

CaptainCasa provides a couple of components that allow to key in text using virtual keyboards:

- The TOUCHFIELD component is a keyboard that is directly rendered into the page:



- If allows the user to specify exactly one text value, which is bound to a server side property.
- All \*FIELD components provide two attributes:
  - TOUCHSUPPORT - is set to true then a virtual keyboard opens up below or on top of the component. By default the keyboard-type is derived from the field - e.g. a numeric keyboard is opened for FORMATTEDFIELD of type "integer".
  - TOUCHLAYOUT - is a link to some own keyboard definition. You may define own keyboard layouts and reference them by setting this attribute.
- The TOUCHVIRTUALKEYBOARD component is an independent keyboard that you can place somewhere into your layout. While the TOUCHFIELD only takes responsibility for one value, the TOUCHVIRTUALKEYBOARD component serves as independent keyboard for all fields / input element of one page. It always updates the component, that currently owns the focus.



## Definition of own Keyboard Layouts

You may define own keyboard layouts that are referenced by the corresponding component definition.

Example:



The field definition is:

```
<t:field id="g_11" touchlayout="owndef" touchsupport="true" width="100" />
```

The attribute TOUCHLAYOUT is defined as “owndef”, which is the pointer to the server side definition of a keyboard layout.

### Definition of keyboard layouts

On server side the definition is kept in a configuration file “touchlayouts.xml”, which itself is part of the directory “/eclntjsfserver/config”. The file's content is defined in the following way:

```
<touchlayouts>
...
...
<layout name="owndef"
  line0="t;h;i;s"
  line1="i;s;@null@a"
  line2="s;t;r;a;n;g;e"
  line3="l;a;y;o;u;t;DEL/1/@clearlast@;C/1/@clearall@" />
<layout name="helloworld"
  line0="H;e;l;l;o"
  line1="w;o;r;l;d"
  line2="DEL/1/@clearlast@;C/1/@clearall@" />
...
...
</touchlayouts>
```

For each layout there is one corresponding “<layout ... />” section. In the section there are several lines being numbered line0, line1, line2 etc.

Each line itself contains the key-definition for one row of the layout. The format is: “<def1>;<def2>;<def3>;...” - each definition representing one key.

A key definition may be defined in two ways:

- “Simple way”: you just define the character that is the one to be represented by the key. The key itself has a default size, the text of the key is the character and the key that is added to the text when pressing the key also is the character. Example: by defining “t” you define a “t”-key.

- “Complex way”: you may define the key by three individual definitions, separated by slash “/”. The definitions are “<displayText>/<width>/<char>”:
  - <displayText>: this is the text that is rendered onto the key
  - <width>: this is the width of the key button. “1” means the normal key width, “2” means 2 times the normal key width, etc.
  - <char>; the character that is added into the text when the user presses the key button

There are a couple of special characters that you may use as <char>-definition:

- “\x47”: this is the representation of a slash (“/”)
- “&#xx;” - you may define any unicode character by defining its decimal representation
- @null@: this is a null key. It will not be rendered as button but as gap within the keyboard layout.
- @clearall@: when pressed then the current content will be cleared
- @clearlast@: when pressed then the last character/ character where the current focus is located will be removed
- @shift@: shift-key for keying in uppercase characters
- @capslock@: caps-key for a sequence of uppercase characters
- @tab@/@backtab@: tab-navigation, only useful for TOUCHVIRTUALKEYBOARD component
- @cursorleft@,@cursorrigh@,@cursorup@,@cursordown@: cursor navigation in current input
- @ok@: the enter key (character “\n”)

## Navigation between keyboard layouts

There is the possibility to navigate between different keyboard layouts. Just define the key “@layout:<layoutname>@” as character.

# Server-side Events, long Server-side Operations

Within the Demo Workplace (“General Issues” > “Server Event Processing”) there are couple of examples for each aspect documents within this chapter. Please look there for details for implementation.

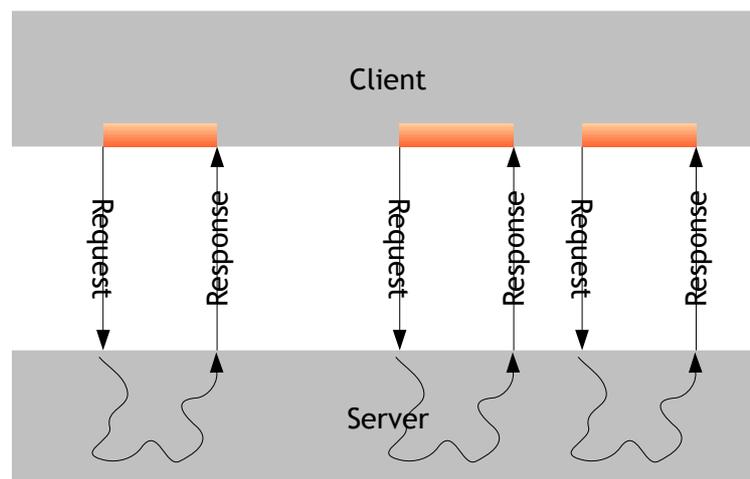
---

## Introduction

### Default: Request-Response driven Communication

The default way the CaptainCasa client talks to the server side processing is:

- The client communicates a request to the server - e.g. after the user having pressed a button.
- The request is assigned to a thread on server side, within this thread the request is processed, i.e. its values are set into the application, the action listeners are invoked and the components are rendered on server side.
- The response is sent back to the client.



During the phase when the client talks to the server the client itself is blocked for input, in order to avoid inconsistency of data.

### Requirement for enhanced Communication

The strict request-response driven way of communication is the best type of communication for the default input/output while the user is working with a screen:

- It only builds up connections for a limited point of time.
- The server side processing is within one thread, there are no multi threading issues. The thread either can be destroyed or cleaned up for further usage after processing.

But of course there are certain limitations as well:

- Long lasting server side operations mean a long blocking of the user interface. During blocking time the user just has to wait and to hope that he/she will receive back a response... there is no e.g. messaging telling the user what he long lasting operation currently is doing,

- Events that are triggered on server side cannot be sent to the client side. The client can pick event data with the next request it sends, but this may depend on the user's activity.

Result: there is a need for enhanced, “event-driven” communication and processing for certain, dedicated scenarios - in parallel to the normal request-response driven way of processing.

## Pay Attention - Thread-Complexity, J2EE Rules...

In this chapter you will see, how certain processing on server side will be shifted into threads on their own, running in parallel to the normal request-response thread. Looking on pure J2EE specifications you are not allowed to open threads on server-side on your own. Of course many J2EE implementations (e.g. Tomcat) do not have problems, but some have.

So, in case you want to stick 100% to J2EE rules you need to re-think how to decouple processing on server side - e.g. one way might be the usage of JMS...

In addition: working with threads etc. adds a certain additional complexity into your server side development. You need to be aware about this and you need to put extreme care into your implementation! - Finally you need to make sure that your network configuration and cluster configuration supports what you want to do.

---

## Constant Polling - The TIMER way

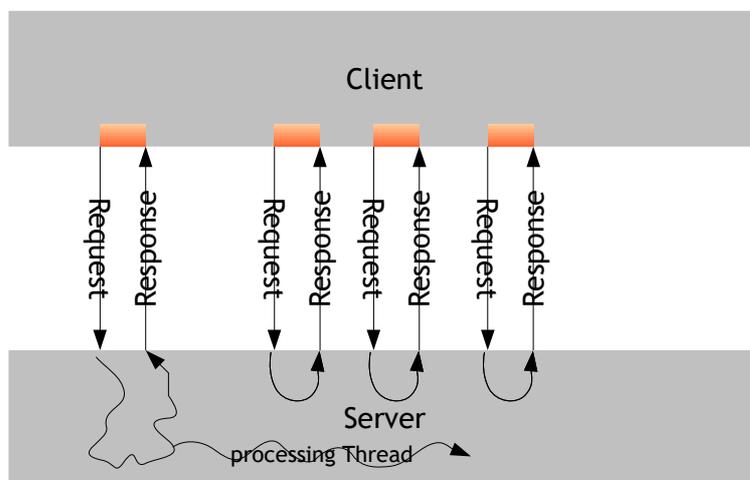
If you only want to have strict request-response driven communication then this is the only way: you set up a TIMER component within the client that regularly polls information “every xx seconds”. It's like a button being pressed regularly by the user.

The TIMER component provides two significant attributes:

- The DURATION - i.e. the polling rate in milliseconds
- The ACTIONLISTENER - i.e. the method to be called on server side

You can update the duration by binding it to a property - if you set value “0” then the TIMER component will stop sending requests.

As result you can build scenarios like the following:



The first request from the client starts a processing thread on the server side - and returns back to the client - so that the user interface is “freed up” quite fast. A timer will

afterwards regularly send requests to check the status of the processing thread. When finished, the timer may de-activate itself.

Of course there are some disadvantages with polling:

- The number of requests is quite high.
- The user interface is permanently blocked for short time because of the poller communicating to the server side.

As result we recommend timer based polling for scenarios in which you want to regularly but not very often (e.g. every 2 minutes) check for things happening on server side.

---

## Long Polling - True, synchronous Event Coupling

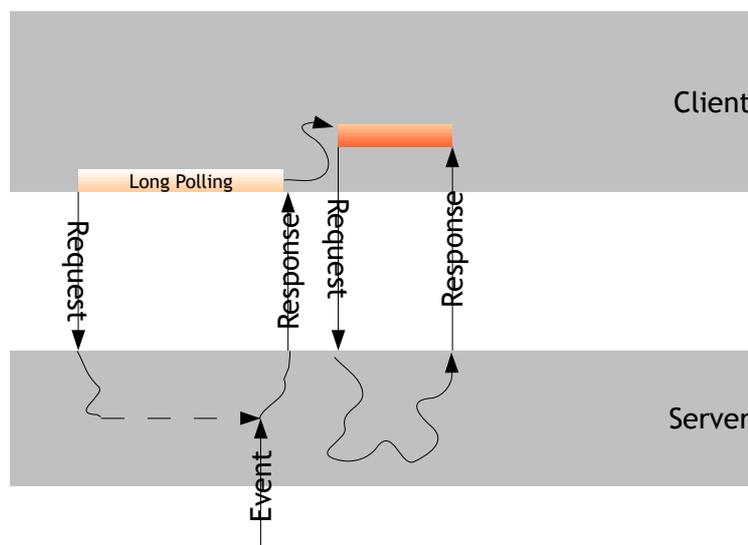
The principles behind long polling are:

- The client sends a request to the server.
- The server does not immediately respond, but waits until some event occurred on server side.
- The server sends back the response, so that the client gets notified.
- Immediately afterwards the client sends a new request waiting for the next event.

You see, long polling means that there is constantly one connection kept open - until something happens on server side. Long polling both occupies one connection and occupies one thread, that is in waiting status until being woken up by a certain server side event.

While keeping many connections open is not too much a problem for up to date networks, keeping a high number of threads up all the time may be some problem for certain environments. So long polling can nicely be used if there are some special workplaces which need synchronous event coupling, while the rest does not need it.

In CaptainCasa long polling is implemented in the following way:



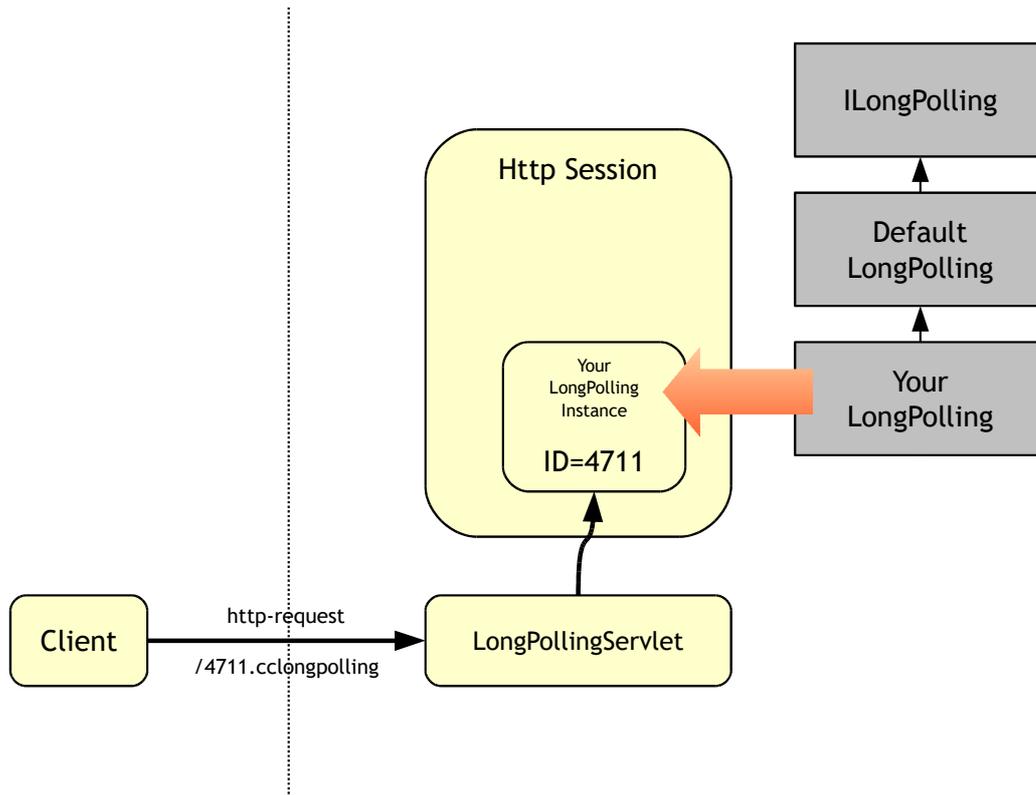
On client side a special request from a long polling component (LONGPOLLING) is sent to the server. When being woken up the response is communicated back. As consequence the client side triggers a real, normal request-response communication. - What is not shown in the picture: after having processed the response, the long polling directly starts a new request to the server side, waiting for the next event.

You can think of the LONGPOLLING component as virtual button, that is pressed by receiving a response from the server side.

OK, so there are two levels of communication:

- The “event channel” - i.e. the long polling
- The “processing channel” - i.e. the normal data transfer by request-response.

The event channel now requires a URL to go to, behind this URL there needs to be the possibility to add some Java code. So for this purpose CaptainCasa introduces a concept that is very similar to the concept of providing URLs for dynamic down/upload of data:



Within the HttpSession context an instance of “DefaultLongPolling” (or sub-class) is registered with a certain id. The instance passes back a URL, containing this id. The client is calling the URL, the central servlet dispatches the request to the LongPolling instance.

Have a look into the following codes:

```

package workplace;

public class DemoLongPolling
    extends DemoBase
    implements Serializable
{
    // -----
    // inner classes
    // -----

    class EventCreatorThread extends Thread
    {
        boolean i_continue = true;
        public void run()
        {
            while (i_continue == true)
            {
                try
                {

```

```

        Thread.currentThread().sleep(5000);
        if (i_continue == true)
            m_longPolling.wakeup(true);
        else
            break;
    }
    catch (Throwable t)
    {
        CLog.L.log(CLog.LL_ERR,"Error occurred when waking up
thread",t);
        break;
    }
}
}

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

DefaultLongPolling m_longPolling = new DefaultLongPolling();
EventCreatorThread m_eventCreatorThread = new EventCreatorThread();

String m_message = new String();

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

public DemoLongPolling(IworkpageDispatcher dispatcher)
{
    super(dispatcher);
    LongPollingMgr.add(m_longPolling);
    m_eventCreatorThread.start();
    getworkpage().addLifecycleListener(new workpageDefaultLifecycleListener()
    {
        @Override
        public void reactOnDestroyed()
        {
            super.reactOnDestroyed();
            CLog.L.log(CLog.LL_INF,"Cleaning up threads...");
            LongPollingMgr.remove(m_longPolling);
            m_eventCreatorThread.i_continue = false;
        }
    });
}

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

public String getLongPollingURL() { return m_longPolling.getURL(); }

public void onLongPollingAction(ActionEvent event)
{
    m_message = "Call to server...! " + System.currentTimeMillis() + "\n" +
m_message;
}

public String getMessage() { return m_message; }
}
}

```

An instance of DefaultLongPolling is created and registered.

Whenever a long polling request is started, then this waits within the DefaultLongPolling instance, until the method “wakeup(true/false)” is called. By passing back “true” you define that long polling will continue, i.e. the client will send the next long polling request immediately after having processed the response of the current one.

For a better understanding, what's going on internally, we show you some code of DefaultLongPolling:

```

public boolean waitForEvent()
{

```

```

...
...
    synchronized(this)
    {
        try
        {
            CLog.L.log(CLog.LL_INF,"Now waiting for event to wakeup this long
polling thread");
            m_justWaiting = true;
            this.wait();
            CLog.L.log(CLog.LL_INF,"Event woke up this thread");
            m_justWaiting = false;
            return m_continuePolling;
        }
        catch (Throwable t)
        {
            CLog.L.log(CLog.LL_ERR,"Error occurred when falling to
sleep.",t);
            throw new Error(t);
        }
    }
    ...
}

public void wakeup(boolean continuePolling)
{
    ...
    {
        ...
        synchronized(this)
        {
            try
            {
                CLog.L.log(CLog.LL_INF,"wakeup was called for the long polling
thread, continuePolling = " + continuePolling);
                m_continuePolling = continuePolling;
                this.notify();
            }
            catch (Throwable t)
            {
                m_continuePolling = false;
                CLog.L.log(CLog.LL_ERR,"Error occurred when falling to sleep.");
                throw new Error(t);
            }
        }
    }
    ...
}
}

```

You see that the thread waits within the `waitForEvent` method (which is called by the `LongPollingServlet`) until someone calls the `wakeup` method.

Finally have a look onto the JSP definition:

```

<t:beanprocessing id="g_1">
  <t:longpolling id="g_2"
    actionListener="#{d.DemoLongPolling.onLongPollingAction}"
    longpollingurl="#{d.DemoLongPolling.longPollingURL}" />
</t:beanprocessing>
<t:rowdemobodypane id="g_3" objectbinding="#{d.DemoLongPolling}">
  <t:row id="g_4">
    <t:textarea id="g_5" enabled="false" height="100%"
      text="#{d.DemoLongPolling.message}"
      width="100%" />
  </t:row>
</t:rowdemobodypane>

```

You see the `LONGPOLLING` component's main configuration consists of:

- the URL it calls (the one provided by `DefaultLongPolling` on server side)
- the `ACTIONLISTENER` it calls when receiving a response due to a server side event.

---

## Long Polling - The “Comet-Way”

### The normal, “traditional” way - One waiting thread per Long Polling

The server side part of long polling that was described in the previous chapter is based on the normal J2EE-servlet processing:

- a long polling request is sent from the client to the server
- on server side the request processing is executed in a corresponding thread
- the long polling processing within the thread does not immediately send back a response to the client but waits for an event. The corresponding thread is on status “wait”.
- an event on server side triggers the wake up of the thread - so that the response to the client is sent

From client side point of view the long polling processing is the same as waiting for a very slow response. From server side point of view per long polling a thread is blocked until a certain event releases the thread to respond to the client.

Consequence on server side: there is a potentially high number of threads that is to be managed in the server. If each client for some reason opens up one long polling synchronization - then there is always one thread per client on server side to be managed. This is fine for a low number of clients - but it is a load problem if you want to use long polling for e.g. hundreds of clients.

### The Comet Way - No waiting thread per Long Polling

Luckily there is an alternative processing on server side. This is based on the so called “Comet” protocol, which is a special handling of http-requests on server side. It internally is based on Java NIO functions (non blocking input/output).

Advantage: there is no blocking of threads anymore. Threads are only “blocked” when the request enters the system and when the response is written to the client - but threads are NOT blocked when waiting for an event on server side.

As consequence you can in principle manage many more parallel long polling connections, so connecting hundreds of clients is no load problem (at least not from threading point of view).

### Using the Comet Way

From API point of view there is nearly difference at all when switching to the Comet way. Instead of creating an instance of “DefaultLongPolling” you need to pass an instance of “DefaultLongPollingComet” - which has exactly the same interface.

“DefaultLongPollingComet” and “DefaultLongPolling” both share the interface “ILongPolling”, so there is no difference at all from usage point of view.

There are some configuration issues that you need to apply:

- You need to tell Tomcat that there is a new servlet, please check the “web.xml\_template” that is part of your project. The relevant parts are:

```
<!--  
  Optional, only Tomcat, comet based LongPollingServlet  
  <servlet id="LongPollingServletComet">  
    <servlet-name>LongPollingServletComet</servlet-name>  
    <servlet-  
class>org.eclnt.jsfserver.polling.comet.LongPollingServletComet</servlet-class>
```

```

        <load-on-startup>1</load-on-startup>
    </servlet>
    ...
    ...
    <!--
    optional, only Tomcat, comet based LongPollingServlet
    <servlet-mapping>
        <servlet-name>LongPollingServletComet</servlet-name>
        <url-pattern>*.cclongpollingcomet</url-pattern>
    </servlet-mapping>
    →

```

You need to remove the comments in order to activate the servlet that manages the long polling requests

- In the Tomcat configuration you have to tell Tomcat that it should use Java NIO functions for processing http requests. This is done in the <tomcat>/conf/server.xml file:

```

<Connector port="8080" protocol="org.apache.coyote.http11.Http11NioProtocol"
    connectionTimeout="20000"
    redirectPort="8443"
    maxPostSize="1000000"/>

```

In this case the main http-connector for port 8080 was switched to NIO. You may also leave the port 8080 on its standard port and create a parallel connector (e.g. port 8081) in which you use Java NIO. In this case you have to tell the LONGPOLLING component to use a different port (attribute LONGPOLLING-LONPOLLINGPORT).

## Comet Disadvantage - Only Tomcat...!

You may already have noticed when reading the text above: we talk quite a lot about Tomcat... Well, the reason behind is: the Comet processing is not part of J2EE processing, but is dependent from the servlet container that you use.

You only may use the Comet processing within a Tomcat (>= version 6) environment as consequence.

## The up to date “Servlet 3.0” way - No waiting thread per Long Polling

With update 20150504 a third way was introduced how to handle long polling on server side. This way is based on the servlet 3.0 standard features in the area of asynchronous request processing. If using it: make sure that your servlet container supports servlet 3.0!

In the Tomcat environment, servlet 3.0 was introduced with the Tomcat 7 version.

Similar to the comet processing a servlet request that is processed in the server can tell the servlet engine that it is passing the response in an asynchronous way. The waiting for some asynchronous event is then taking place without blocking the request thread.

The following things need to be done in order to use the servlet 3.0 asynchronous processing:

- Update the web.xml in the following way:

```

    <servlet id="LongPollingServlet30API">
        <servlet-name>LongPollingServlet30API</servlet-name>
        <servlet-
class>org.eclnt.jsfserver.polling.LongPollingServlet30API</servlet-class>
        <load-on-startup>1</load-on-startup>
        <async-supported>true</async-supported>
    </servlet>
    ...
    ...
    <servlet-mapping>
        <servlet-name>LongPollingServlet30API</servlet-name>

```

```
<url-pattern>*.cclongpolling30API</url-pattern>
</servlet-mapping>
```

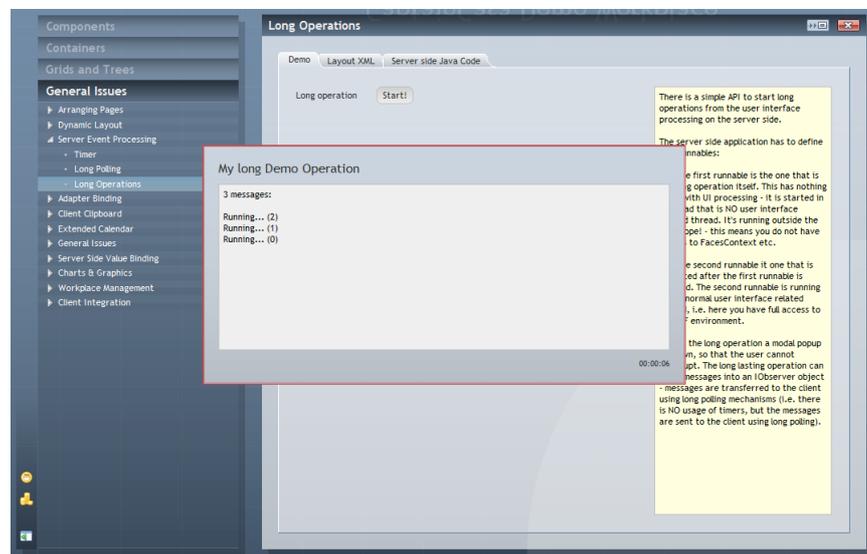
Please note: you may copy the XML for the file “web.xml\_template” that is located in the directory WEB-INF/ as well.

- Inside your server side code use class “DefaultLongPollin30API” instead of “DefaultLongPolling”. Both support interface “ILongPolling” - so the usage is exactly the same.

## Long Operations with Messages

Based on long polling and on the usage of a modal popup there is a nice way of defining long operations that run within a processing thread on server side and which pass text messages into a observer-object.

Please have a look into the demo workplace “General Issues > Server Events > Long operations” for viewing an example.



The application using “long operations” needs to define two “Runnable” instances:

- The first, obligatory one is the long operation itself. This operation is started in a thread on its own so it should not perform any access to JSF processing (e.g. call functions related to the FacesContext).
- The second, optional one is an operation that is executed after the long operation has finished. This second operation is executed within the JSF processing thread.

For writing messages, so that the user gets notified about the status of the processing, there is an observer object, that provides a corresponding interface. The messages are transferred to the client, internally using long polling as described in the previous chapter.

The following shows the code of the example that is available within the demo workplace:

```
public void onStartOperation(ActionEvent event)
{
    final IObserver observer = LongOperationwithObserverPopup.prepare("My
long Demo Operation");
    Runnable longOperation = new Runnable()
    {
        public void run()
```

```

        {
            for (int i=0; i<5; i++)
            {
                observer.addMessage("Running... ("+i+)");
                try
                {
                    Thread.currentThread().sleep(3000);
                }
                catch (Throwable t) {}
            }
        }
    };
    Runnable finishOperation = new Runnable()
    {
        public void run()
        {
            StatusBar.outputSuccess("Finished!");
        }
    };
    LongOperationWithObserverPopup.run(longOperation, finishOperation);
}

```

The long operation is a simple loop, containing some sleep-statements in order to simulate some long processing. The second operation is a simple output to the status bar.

Please pay attention: with the utility class “LongOperationWithObserverPopup” you can only start one long operation on server-side, within the scope of one user session. Nesting of long operations will lead into an error situation.

# Integration Scenarios

This chapter contains information about how to integrate the User Interface Client of CaptainCasa Enterprise Client into existing applications.

## HTML Page Integration (static)

The integration is done by running CaptainCasa Enterprise Client within an applet. The applet contains several parameters (listed in the appendix of this documentation), the most important being the address of the .jsp page to be loaded. Example:

```
<html>

<body leftmargin=0 rightmargin=0 topmargin=0 bottommargin=0 style="overflow:
hidden">

<applet code="org.ecInt.client.page.PageApplet.class"
        archive="ecInt/lib/ecInt.jar"
        width="100%"
        height="100%">
  <param name='page' value='faces/workplace/workplace.jsp'>
  <param name='headerline' value='false'>
  <param name='loglevel' value='INFO'>
  <param name='sizefactor' value='1'>
  <param name='fontfactor' value='1'>
  <param name='restrictedconfig' value='true'>

  <!-- This text appears in case the applet does not show up. -->
  The applet could not be loaded - please install the Java runtime
  plugin: <a
href="http://www.java.com/de/download">http://www.java.com/de/download</a>
</applet>

</body>
</html>
```

### Libraries to be included

The list of libraries, that is a comma separated list within the “archive” attribute of the applet, depends on the components that you used within your pages. By default only the library “elcnt.jar” is required on client side.

There are special situations in which you need to include further libraries:

- You use the ACTIVEX or BROWSER integration
- You use the JRVIEWER or JRPRINT component for the direct output of JasperReports data.

By using the Tool “Create HTML/JNLP”, that is part of the Layout Editor, the list of libraries is automatically adjusted - dependent from the configuration that you input.

### Passing Parameters into the Server Side Application

By using just normal http-parameters you can pass information from the applet definition into the server side application which is contacted by the applet. Parameters can be added using the normal way of adding parameters to a URL, i.e. Using “?name=value&name=value” extensions.

Example: in the following .html page the JSP page, that is addressed within the applet definition, has some parameters:

```
<html>
```

```

<body leftmargin=0 rightmargin=0 topmargin=0 bottommargin=0 style="overflow:
hidden">

<applet code="org.ecInt.client.page.PageApplet.class"
        archive="ecInt/lib/ecInt.jar,ecInt/lib/looks-2.1.4.jar,ecInt/lib/swt.jar"
        width="100%"
        height="100%">
  <param name='page'
        value='faces/workplace/demohttpparams.jsp?param1=Test1&param2=Test2'>

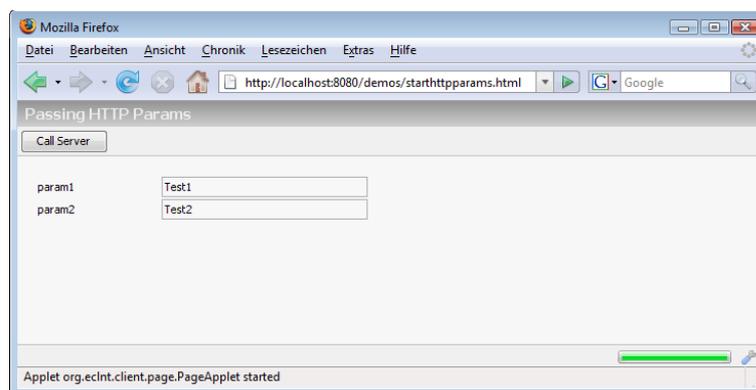
  ...

</applet>

</body>
</html>

```

The result is the following page:



You see: the parameters somehow have found their way from the applet definition into the page... This is the server side program behind:

```

package workplace;

import java.io.Serializable;

import javax.faces.event.ActionEvent;
import javax.servlet.http.HttpServletRequest;

import org.ecInt.editor.annotations.CCGenClass;
import org.ecInt.jsfserver.util.HttpSessionAccess;

@CCGenClass (expressionBase="#{d.DemoHttpParams}")
public class DemoHttpParams implements Serializable
{
    public DemoHttpParams()
    {
        HttpServletRequest request = HttpSessionAccess.getCurrentRequest();
        m_param1 = request.getParameter("param1");
        m_param2 = request.getParameter("param2");
    }

    public void onCallServer(ActionEvent event)
    {
    }

    protected String m_param2;
    public String getParam2() { return m_param2; }
    public void setParam2(String value) { m_param2 = value; }

    protected String m_param1;
    public String getParam1() { return m_param1; }
    public void setParam1(String value) { m_param1 = value; }
}

```

You see that within the constructor of the server side bean the servlet request is

accessed. The servlet request provides functions to access http parameters.

In the coding above the helper “HttpSessionAccess” is used to get access to the http request. The code that internally accesses the http request is:

```
public static HttpServletRequest getCurrentRequest()
{
    HttpServletRequest request = (HttpServletRequest)FacesContext
        .getCurrentInstance()
        .getExternalContext()
        .getRequest();
    return request;
}
```

You see: everything is “plain JSF”, no magic behind!

## “Special Parameters” since Java 1.6 Update 10 - Important!

Since Java 1.6 Update 10 there are dedicated parameters provided by Java runtime environment that allow to influence the way the applet is started. The following is the “startdemos.html” page within the demos-project:

```
<html>
<body leftmargin=0 rightmargin=0 topmargin=0 bottommargin=0 style="overflow:
hidden">
<applet code="org.eclnt.client.page.PageApplet.class"
    archive="eclnt/lib/eclnt.jar" width="100%" height="100%">
    <param name='page' value='faces/workplace/workplace.jsp'>
    <param name="image" value="eclnt/images/splash.png">
    <param name="centerimage" value="true">
    <param name="java_arguments" value="-Xmx256m">
    <param name="java_version" value="1.6+ ">
    <param name="separate_jvm" value="true">
    <!-- This text appears in case the applet does not show up. -->
    The applet could not be loaded - please install the Java runtime
    plugin: <a
href="http://www.java.com/de/download">http://www.java.com/download</a>
</applet>
</body>
</html>
```

There are three parameters:

- “java\_arguments” passes any type of virtual machine parameters that are used when starting the virtual machine. One very important one is the sizing of the memory: by default an applet’s heap size is limited to 64 MBytes, in the example this maximum heap size is defined to be 256 MBytes.
- “java\_version” passes the required version number
- “separate\_vm” defines if/that the applet should be started in an own virtual machine instances. We definitely advise to set this parameter to “true” because it is the only way to properly separate the client side virtual machines.

The three “special” parameters are set in the way you see it from the example automatically when using the “.ccapplet” way of starting a CaptainCasa page as applet. They are also set when creating an applet-html page using the toolset (function in Layout Editor to create applet/jnlp).

Please check the Java 1.6 Update 10 documentation in order to read about details and further parameters.

## Accessing the Applet Context

If a Java program is started as applet within a browser environment then there is the possibility to talk to the browser using the so called “applet context”. The main function that is provided by the applet context is the one to open documents within the browser - either in the main frame or in frame targets.

For talking to the applet context there is an invisible client component “APPLETCONTEXTSHOWDOCUMENT”. The component is able to open up URLs that you pass in the browser.

Typical areas of usage are:

- You want to switch from the current applet page to the next page (e.g. a log off page).
- You want to open documents (e.g. help texts) in a frame that runs aside the frame in which the applet was started.

Of course the usage of the component only makes sense when the CaptainCasa Enterprise Client is started as applet within a browser environment. In other situations (e.g. web start) it will behave “in a quiet way” - it will e.g. not throw any exceptions.

Please check the demo workplace, section “General Issues, Client Integration” for more details.

---

## HTML Page Integration (dynamic)

The dynamic page integration bases on the same concepts as the static page integration - but uses a servlet on server side to dynamically generate the HTML text containing the applet definition.

The servlet is invoked by sending a URL from the browser that has the following format:

- `http(s)://<host>:<port>/<webapp>/<jspname>.ccapplet`

The “<jspname>” is the name of the jsp page to be started as applet. All directory-slashes within the directory name need to be replaced by “.”. Example:

- <http://localhost:50000/demos/workplace.workplace.ccapplet> will start the jsp page “workplace/workplace.jsp”

## Defining the Client Libraries to be loaded via URL Parameter

By default the html page that is generated contains an applet definition that itself only referenced the “eclnt.jar” library to be loaded. This is sufficient for most scenarios, but e.g. not sufficient when using native component integration (ACTIVEX, BROWSER) or JasperReports components (JRVIEWER, JRPRINT).

There is a URL parameter “cclibs” that you can use for listing the libraries that you want to be loaded. Just append the parameter to the URL:

```
http://localhost:50000/demos/workplace.workplace.ccapplet&cclibs=...
```

The parameter value is a comma separated listing of the following words:

- “swt” for loading swt.jar (ActiveX, Browser integration)
- “jasperreports” for loading jasper reports basic functions
- “jasperreportsall” for loading all jasper reports functions including the export to .pdf and .xls files
- “pdf” for loading the PDF renderer library

- “comm” for working with the serial interface (CLIENTSERIALRECEIVER component)
- “osm” for working with the open street map component
- “simplehtmleditor” for working with the SIMPLEHTMLEDITOR component.

Example:

```
http://localhost:50000/demos/workplace.workplace.ccapplet&cclibs=swt,jasperreportsall,pdf
```

...will start the workplace page with a combination of libraries listed as “cclibs”-value.

If you want to load all existing client libraries you simply can define “cclibs=all” - but please be aware of the fact that now quite a big package of libraries is sent to the client.

## Centrally defining the “cclibs”-Parameter

Instead of appending the cclibs parameter to the URL you can define a central cclibs-Parameter within the configuration file “eclntjfsserver/config/system.xml”.

```
...
...
<ccappletccwebstart
  cclibs="swt,jasperreportsall,pdf"
/>
...
...
```

In this case the libraries will be loaded by a .ccapplet/.ccwebstart request - without having to explicitly list them within the URL of the request.

The valid values for the cclibs-content can be found in the JavaDoc documentation for the server side class “AppletStarter”.

## Centrally defining the Client-Parameters

In case that you always want to start the client with certain start parameters, then you can centrally define these parameters in the “eclntjfsserver/config/system.xml”-file as well:

```
...
...
<ccappletccwebstart ...>
  <clientparam name="..." value="..." />
  <clientparam name="..." value="..." />
  <clientparam name="..." value="..." />
</ccappletccwebstart>
...
...
```

## Specifying a Template File by URL Parameter

You may also define a template file for building the HTML page containing the applet to start Enterprise Client.

By URL...

- <http://localhost:50000/demos/workplace.workplace.ccapplet?cctemplate=com/abc.html>

... the HTML template defined in the file “com/abc.html” is used. The template file must be part of you server side coding, it is referenced by CaptainCasa using the classloader.

The template may look like:

```

<html>
<body leftmargin=0 rightmargin=0 topmargin=0 bottommargin=0 style="overflow:
hidden">

<applet code="org.ecInt.client.page.PageApplet.class"
        archive="ecInt/lib/ecInt.jar,ecInt/lib/swt.jar"
        width="100%"
        height="100%">
  <param name='page' value='faces/@@pagename@@'>

  <!-- This text appears in case the applet does not show up. -->
  The applet could not be loaded - please install the Java runtime
  plugin: <a
href="http://www.java.com/de/download">http://www.java.com/download</a>

</applet>

</body>
</html>

```

At runtime the literal “@@pagename@@” is replaced by the page that is part of the “.ccapplet”-URL.

In the template file you may pass additional client parameters (e.g. look and feel) to the client.

## Specifying the default Template File by Configuration

You may also specify a template file that is used by default by adding a corresponding definition to the /ecIntjsfserver/config/system.xml configuration file:

```

<system>
  ...
  ...
  <!--
  Default templates that are used for building the .html / .jnlp
  file when resolving a .ccapplet/.ccwebstart request. If not specified
  then default values will be chosen.
  -->
  <starttemplates
    ccapplet="org/ecInt/jsfserver/starter/applettemplate.html"
    ccwebstart="org/ecInt/jsfserver/starter/webstarttemplate.jnlp"
  />

</system>

```

The ccapplet-definition points to a resource that is loaded via class loader at runtime.

## Passing additional URL Parameters

You can pass further http-parameters to the URL just by appending them to the “.ccapplet”-URL. Example:

- <http://localhost:50000/demos/workplace.workplace.ccapplet?param1=v1&param2=v2>

The parameters will automatically be appended to the URL of the applet-page-parameter in the generated page, i.e. they will be appended to the URL that is used by the Enterprise Client for starting the page.

This is quite powerful because it allows you to pass parameters from outside (the one building the URL) into the server side application processing. The server side application processing accesses these parameters in the following way:

```
String s = HttpSessionAccess.getCurrentRequest.getParameter("param1");
```

## Passing additional URL Parameters that are always Part of the Request

Imagine the following: you integrate a CaptainCasa page into some kind of portal page. Certain information (e.g. about a single sign-on) is passed using URL parameters. Now, the CaptainCasa page runs inside environment - and after a while is timed out, because the user did not use the system for a while. What happens?

The servlet container will create a new, cleaned session on server side. All session data will not be available anymore. And: because normal URL parameters are only transferred with the first request, there is no change on server side to recover them.

To solve this situation there is a special “URL-Parameter-Feature”: when a URL parameter ends with “\_always”, then this parameter is always sent as part of every request that is sent from client to server. This means the information does not get lost in case of session timeouts.

Consequence: name your parameter so that it ends with “\_always” for URL parameters that are not only sent with only the first, initial request from the client to the server, but that are sent with every request.

---

## Webstart Integration

The Webstart integration is very similar to the applet integration - instead of defining an “.html” file containing the start configuration you need to define a “.jnlp” definition - either in a static or dynamic way:

### Static JNLP Definition

The JNLP file looks the following way:

```
<?xml version="1.0" encoding="utf-8"?>
<jnlp spec="1.0" codebase="$$codebase" href="$$name">
  <information>
    <title>CaptainCasa Enterprise Client</title>
    <vendor>CaptainCasa GmbH</vendor>
    <icon href="ecInt/images/splash.png" kind="splash"/>
  </information>
  <security>
    <all-permissions />
  </security>
  <resources>
    <j2se version="1.6+" java-vm-args="-esa -xnoclassgc"/>
    <jar href="ecInt/lib/ecInt.jar"/>
  </resources>
  <application-desc main-class="org.ecInt.client.page.PagewebStart">
    <argument>$$codebase</argument>
    <argument>faces/workplace/workplace.jsp</argument>
  </application-desc>
</jnlp>
```

We recommend to use the Tool “Create HTML/JNLP” within the Layout Editor to create the file. The tool automatically created the correct list of client libraries to be loaded.

### Dynamic Definition

As with applets you can use a URL, now ending with “.ccwebstart” in order to dynamically create the JNLP definition on server side.

- <http://localhost:50000/demos/workplace.workplace.ccwebstart> will start the jsp page “workplace/workplace.jsp”

In the same way as with applets you can...

- ...use URL parameter “cclibs” in order to define the libraries to be loaded into the client
- ...select the template by referencing it via parameter “?cctemplate=...”. You need to define a corresponding template-jnlp-file that contains the literal “@@pagename@@”:

```
<?xml version="1.0" encoding="utf-8"?>
<jnlp spec="1.0" codebase="$$codebase">
  <information>
    <title>CaptainCasa Enterprise Client</title>
    <vendor>CaptainCasa GmbH</vendor>
    <icon href="eclnt/images/splash.png" kind="splash"/>
  </information>

  <security>
    <all-permissions />
  </security>

  <resources>
    <j2se version="1.6+" java-vm-args="-esa -xnoclassgc"/>
    <jar href="eclnt/lib/eclnt.jar"/>
  </resources>

  <application-desc main-class="org.eclnt.client.page.PagewebStart">
    <argument>$$codebase</argument>
    <argument>faces/@@pagename@@</argument>
    <argument>loglevel=INFO</argument>
  </application-desc>
</jnlp>
```

- ...append parameters to be passed to the application processing: “? param1=v1&param2=v2”, and pickup the parameters by accessing the http request on server side
- ...define the default template that is used for building the .jnlp file by adding a corresponding definition into the /eclntjsfserver/config/system.xml configuration file:

```
<system>
...
...
<!--
Default templates that are used for building the .html / .jnlp
file when resolving a .ccapplet/.ccwebstart request. If not specified
then default values will be chosen.
-->
<starttemplates
  ccapplet="org/eclnt/jsfserver/starter/applettemplate.html"
  ccwebstart="org/eclnt/jsfserver/starter/webstarttemplate.jnlp"
/>

</system>
```

The ccwebstart-definition points to a resource that is loaded via class loader at runtime.

## Webstart / JNLP: the “codebase” Parameter

In the JNLP file there are two occurrences of an attribute value “\$\$codebase”. What's the background behind this parameter?

The codebase parameter is a URL that points to the directory which is the “main directory” of the web application. It is required inside the .jnlp file because the web start on client side needs to know which server to contact for loading libraries and for sending the first page request.

In normal .jnlp files this parameter is hard coded, so that the .jnlp file can also be dragged and dropped from the browser onto the desktop - and then can be started independently from a browser. In the default .jnlp file generated by the CaptainCasa environment the parameter is defined at runtime, i.e. the “\$\$codebase” attribute is replaced accordingly. The base for replacements is the original URL that is used in the browser to access the .jnlp file.

You need to pay attention when defining scenarios in which the client is not directly connected to the web server in which the CaptainCasa server part is running. Maybe there are some proxies and routers between client and server - not only delegating requests, but also changing the URL of the request.

In this case you need to manually replace the “\$\$codebase” attribute by the codebase-URL that is required to access the application from the client side.

Example: the normal URL to the Tomcat based system may be: [“http://tomcat:8080/demos/xyz.ccaplet”](http://tomcat:8080/demos/xyz.ccaplet), the “\$\$codebase” is replaced by [“http://tomcat:8080/demos/”](http://tomcat:8080/demos/).

Now, there is an Apache proxy between client and server which routes [“http://apache/cc”](http://apache/cc) to [“http://tomcat:8080/”](http://tomcat:8080/). The “\$\$codebase” resolution does not know this, because this happens without letting Tomcat know. In this case you need to manually use the codebase [“http://apache/cc/demos/”](http://apache/cc/demos/) - because this is the one that is usable from client perspective.

## Passing Cookies

There are situations in which you want to pass cookie information from the page that contained the “.ccwebstart” link into the CaptainCasa client processing. You may switch this on by adding query parameter “cctransfercookies=true” to your URL. Your URL may look like: <http://localhost:8080/demos/mypage.jsp?cctransfercookies=true>”.

In this case all the cookies of the .ccwebstart-request will be read and will be added as cookies to the request that is sent from the CaptainCasa Swing/FX Client to the server side processing.

## Closing the Client

By default, when the user closes the client (e.g. by pressing alt-F4 or by pressing the corresponding close-icon) a question “Do you really want to exit?” is shown. When the user presses “Yes” then the client is closed, otherwise the client continues to run.

The first way to influence this behavior is by setting a client parameter:

- The client parameter “confirmexit” can be set to “false” - in this case the client will immediately close without any question to the user.

But you can also control the closing of the client completely through your server side application: there is an explicit component CLIENTCLOSER that you use in the following way:

- Every time the client should be closed by the user (e.g. after pressing alt-F4) then the client is not closed, but sends a request to the ACTIONLISTNER-method of the CLIENTCLOSER component. - As consequence the server gets notified about the wish of the user to close the client.
- The server side application can react “just normally”, as with any other action listener call. E.g. the application may check if there is unsaved data.
- If the application really wants to close the client then it does so by assigning a Trigger-instance to the CLIENTCLOSETRIGGER attribute of the CLIENTCLOSER component. When

the trigger is activated (via its trigger() method), then this is a the signal for the client to now really close.

Place the CLIENTCLOSER component in your most outside page - then the full control about closing the client is fully on your application's side.

Reated topics:

- Please check if you want to use the SESSIONCLOSER component in addition (which typically makes sense). The SESSIONCLOSER component triggers a request to the server, when the client is really closed - as consequence the server side session will automatically be cleaned up and does not wait for server-side session time-out.

---

## Embedding Applet into existing HTML Application

### Embedding the Applet + Session Considerations

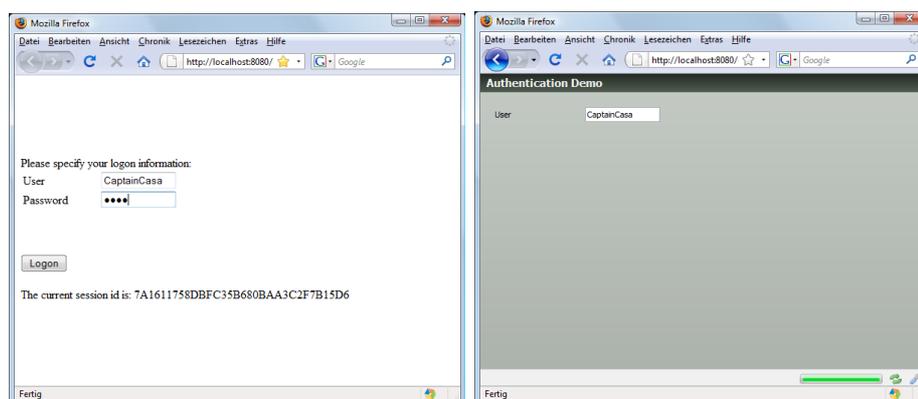
The previous chapter talked about how to start CaptainCasa Enterprise Client as applet or webstart application. The main scenario was: starting the Enterprise Client via a URL that typically is input into the URL line of your browser.

Now let's concentrate on integrating your Enterprise Client into an existing application that already has some HTML screens. At a certain part you want to embed the CaptainCasa client using applet so that it runs optically integrated within your application's frontend.

Of course, the typical way is "clear": you create an HTML-IFRAME inside your application and start the Enterprise Client passing the URL to start into the IFRAME's "src"-attribute. By using URL parameters ("?name=value&name=value") you may pass information to the Enterprise Client application processing.

You can reach an even tighter integration (if you want so) by even sharing the http-session between the outside HTML application and the inside Enterprise Client application. There is an additional parameter with the "ccapplet" and "ccwebstart" processing that tells that the Enterprise Client's application should use the same session as the calling HTML-application.

Let's look onto the following example:



The user inputs some logon information. On the server side there is corresponding servlet that checks the logon (one need to input "test" as password...) and writes the user's name into the session context. After successful logon an Enterprise Client screen is started in which the user name is output.

Let's take a look onto the servlet:

```

package workplace;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class DemoAuthenticationServlet extends HttpServlet
{
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException
    {
        // force creation of session
        req.getSession().setAttribute("usingSession","true"); // dummy value
        String user = req.getParameter("pUser");
        String password = req.getParameter("pPassword");
        if ("test".equals(password) &&
            user != null)
        {
            req.getSession().setAttribute("puser",user);
            resp.getWriter().write
            (
                "<html>" +
                "<body leftmargin=0 topmargin=0 rightmargin=0 bottommargin=0"
                style='overflow:hidden'>" +
                "<iframe"
                src='"+resp.encodeURL("workplace.demoauthentication.ccapplet" +
                "?ccsamesession=true") // otherwise the
                Enterprise Client runs in an own http session
                +"' width='100%' height='100%' style='border:0'>"
            +
                "</body>" +
                "</html>"
            );
        }
        else
        {
            if (user == null) user = "";
            if (password == null) password = "";
            resp.getWriter().write
            (
                "<html>" +
                "<body>" +
                "<br>" +
                "<br>" +
                "<br>" +
                "<br>" +
                "<br>" +
                "Please specify your logon information: " +
                "<form id='form1' method='post'"
                action='"+resp.encodeURL("demoauthenticationervlet")+"' enctype='application/x-
                www-form-urlencoded'>" +
                "<table border='0'>" +
                "<tr>" +
                "<td width='100'>User</td>" +
                "<td width='100'><input name='pUser' type='text' value='"+user+"'"
                style='width:100'></td>" +
                "</tr>" +
                "<tr>" +
                "<td width='100'>Password</td>" +
                "<td width='100'><input name='pPassword' type='password'"
                value='"+password+"'" style='width:100'></td>" +
                "</tr>" +
                "</table>" +
                "<br>" +
                "<br>" +
                "<br>" +
                "<input type='submit' value='Logon'>" +
                "<br>" +
                "<br>" +
                "The current session id is: " + req.getSession().getId() +
                "</form>" +
                "</body>" +
                "</html>"
            );
        }
    }
}

```

```

        );
    }
}

@Override
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException
{
    doGet(req, resp);
}
}

```

There are two important things:

- Use the URL parameter “ccsamesession=true” when calling the Enterprise Client.
- Use the “HttpResonse.encodeURL()” method for building a proper URL to be passed into the IFRAME. Only this will allow to use http session management without being forced to use cookies.

Now the following Enterprise Client application can itself contact the session context and use the information from there. The picking of the information is done in the constructor of the managed bean:

```

package workplace;
import java.io.Serializable;
import javax.servlet.http.HttpSession;
import org.eclnt.editor.annotations.CCGenClass;
import org.eclnt.jsfserver.util.HttpSessionAccess;

public class DemoAuthentication implements Serializable
{
    public DemoAuthentication()
    {
        HttpSession session = HttpSessionAccess.getCurrentHttpSession();
        m_user = (String)session.getAttribute("pUser");
    }

    protected String m_user;
    public String getUser() { return m_user; }
    public void setUser(String value) { m_user = value; }
}

```

One word on cookies: we definitely recommend to not use cookies. Cookies are likely to be shared among different browser instances - and different instances sharing the same server side http session typically in not healthy at all. CaptainCasa will log out some information every time it finds out that the same session is used by different browser instances.

Well, maybe your existing application is relying on cookies - so you cannot change. But if your are in the phase of providing a new application: provide URL encoding based session management!

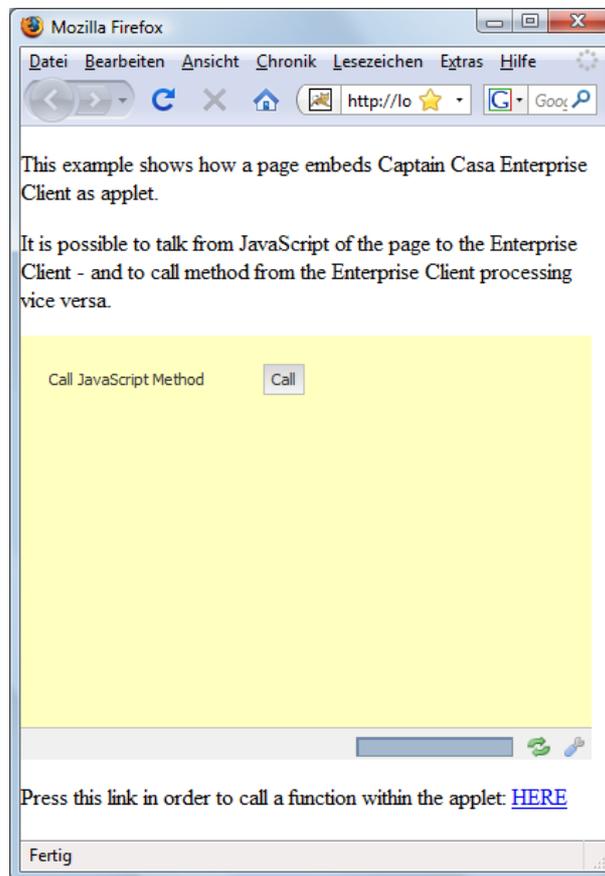
Avoiding of cookies typically can be done by maintained the file META-INF/context.xml - there is a template file that comes with CaptainCasa in the same directory in which you see how to use.

And, one word on user information: in the example the user name was passed as http-session parameter. We know, there is a authentication methodology with web applications, that automates the gathering for login information and that uses the “principal” of the http session context. But: in the example we just wanted to show how to share the session context. Which part of the session context you actually use, this is up to you.

## JavaScript Integration

When embedding the applet into an existing HTML application then there are often situations in which you want the outside HTML page to call functions of the inner applet - and vice versa.

This is possible by using the JavaScript interface of the CaptainCasa Enterprise Client. Have a look onto the following page:



The Enterprise Client applet (the yellow one...) runs embedded into some HTML page. When the user presses the “Call” button in the applet then a JavaScript function of the page is executed. Vice versa, when the user presses the link “HERE” then a method is invoked within the Enterprise Client applet.

First have a look onto what happens when the user presses the “HERE”-link:

```
<html>
<body leftmargin=0 rightmargin=0 topmargin=0 bottommargin=0 style="overflow:
hidden">
<script type="text/javascript">
// this code is used by APPLETCONTEXTSHOWDOCUMENT - if this is configured to
// use java script instead of directly contacting the context
function ccOpenPageInTarget(ccTarget,ccURL)
{
    var ccwindow = window.open(ccURL,ccTarget);
    ccwindow.focus();
}
</script>

<script type="text/javascript">
function callApplet()
{
    document.applets[0].passMessageToClient("jsjsjs(Hello Java!)");
}
```

```

}
</script>

<p>
This example shows how a page embeds Captain Casa Enterprise Client as applet.
</p>
<p>
It is possible to talk from JavaScript of the page to the Enterprise Client - and
to call method from the Enterprise Client processing vice versa.
</p>

<p>
<applet id="captaincasa"
  code="org.ecInt.client.page.PageApplet.class"
  archive="ecInt/lib/ecInt.jar" width="400" height="300" MAYSCRIPT>
  <param name='page' value='faces/workplace/demojsintegration.jsp'>
  <param name="image" value="ecInt/images/splash.png">
  <param name="centerimage" value="true">
  <param name="java_arguments" value="-Xmx256m">
  <param name="java_version" value="1.6+ ">
  <param name="separate_jvm" value="true">
  <param name="MAYSCRIPT" value="true">

  <!-- This text appears in case the applet does not show up. -->
  The applet could not be loaded - please install the Java runtime
  plugin: <a
href="http://www.java.com/de/download">http://www.java.com/download</a>

</applet>
</p>

<p>
Press this link in order to call a function within the applet: <a
href="javascript:callApplet();">HERE</a>
</p>

</body>
</html>

```

When the user presses the “HERE”-Link then the JavaScript-function “callApplet()” is executed. This function call's the applet's method “passMessageToClient(String)”. The String parameter contains a message in the format “messageCommand(param1,param2,...)”. The method “passMessageToClient” of the applet transfers the message to the message listener system that is part of the client framework. Pages can listen to the messages by defining MESSAGELISTENER components. That's exactly what the page, running inside the client, does:

```

<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>

<%@taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@taglib prefix="h" uri="http://java.sun.com/jsf/html"%>

<%@taglib prefix="demo" uri="/WEB-INF/democontrols"%>
<%@taglib prefix="t" uri="/WEB-INF/ecInt"%>

<!-- ===== CONTENT BEGIN ===== -->
<f:view>
<h:form>
<f:subview id="workplace_demojsintegration_sv">
<t:beanprocessing id="g_1" >
<t:clientjsmethodcaller id="g_2" jsCall="alert(HELLO!)"
trigger="#{d.DemoJsIntegration.jsCallTrigger}" />
<t:messageListener id="g_3"
actionListener="#{d.DemoJsIntegration.onJsMessageReceived}"
commandFilter="jsjsjs" />
</t:beanprocessing>
<t:rowbodypane id="g_4" bgpaint="rectangle(0,0,100%,100%,#FFFFFFC0)" >
<t:row id="g_5" >
<t:label id="g_6" text="Call JavaScript Method" width="150" />
<t:button id="g_7" actionListener="#{d.DemoJsIntegration.onCallJsMethod}"
text="Call" />
</t:row>
</t:rowbodypane>

```

```

<t:rowstatusbar id="g_8" />
<t:pageaddons id="g_pa"/>
</f:subview>
</h:form>
</f:view>
<!-- ===== CONTENT END ===== -->

```

The MESSAGELISTENER listens to messages of type “jsjs” - you see, that this is exactly matching the message that is sent from JavaScript (“jsjs(Hello Java!)”). In case a message is received then the corresponding action listener is called on server side.

The Java program for the page is:

```

package workplace;

import java.io.Serializable;

import javax.faces.event.ActionEvent;

import org.eclnt.editor.annotations.CCGenClass;
import org.eclnt.jsfserver.defaultscreens.Statusbar;
import org.eclnt.jsfserver.elements.BaseActionEvent;
import org.eclnt.jsfserver.elements.events.BaseActionEventMessage;
import org.eclnt.jsfserver.elements.util.Trigger;

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

public class DemoJsIntegration implements Serializable
{
    protected Trigger m_jsCallTrigger = new Trigger();
    public Trigger getJsCallTrigger() { return m_jsCallTrigger; }
    public void setJsCallTrigger(Trigger value) { m_jsCallTrigger = value; }

    public void onCallJsMethod(ActionEvent event)
    {
        m_jsCallTrigger.trigger();
    }

    public void onJsMessageReceived(ActionEvent event)
    {
        if (event instanceof BaseActionEventMessage)
        {
            BaseActionEventMessage baem = (BaseActionEventMessage)event;
            Statusbar.outputSuccess("Message was received: " +
baem.getMessageCommand());
        }
    }
}

```

Now let's check how to invoke JavaScript processing from the CaptainCasa Enterprise Client. There's a special component to do so: CLIENTJSMETHODCALLER. It is embedded in the page that is shown above:

```

<%@page contentType="text/html"%>
<%@page pageEncoding="UTF-8"%>

<%@taglib prefix="f" uri="http://java.sun.com/jsf/core"%>
<%@taglib prefix="h" uri="http://java.sun.com/jsf/html"%>

<%@taglib prefix="demo" uri="/WEB-INF/democontrols"%>
<%@taglib prefix="t" uri="/WEB-INF/eclnt"%>

<!-- ===== CONTENT BEGIN ===== -->
<f:view>
<h:form>
<f:subview id="workplace_demojsintegration_sv">
<t:beanprocessing id="g_1" >
<t:clientjsmethodcaller id="g_2" jsCall="alert(HELLO!)
trigger="#{d.DemoJsIntegration.jsCallTrigger}" />
<t:messageListener id="g_3"
actionListener="#{d.DemoJsIntegration.onJsMessageReceived}"

```

```

commandfilter="jsjsjs" />
</t:beanprocessing>
<t:rowbodypane id="g_4" bgcolor="rectangle(0,0,100%,100%,#FFFFFFC0)" >
<t:row id="g_5" >
<t:label id="g_6" text="Call JavaScript Method" width="150" />
<t:button id="g_7" ActionListener="#{d.DemoJsIntegration.onCallJsMethod}"
text="Call" />
</t:row>
</t:rowbodypane>
<t:rowstatusbar id="g_8" />
<t:pageaddons id="g_pa"/>
</f:subview>
</h:form>
</f:view>
<!-- ===== CONTENT END ===== -->

```

Inside the definition of CLIENTJSMETHODCALLER there is a JSCALL attribute. This contains the calling of a JavaScript function - that is expected to be available within the page that includes the applet definition. In the example the “alert(...)” method is used.

The invoking of the component is done by a trigger. In the example the trigger is called by the Java program, when processing the method that is called when the user presses the “Call” button:

```

public void onCallJsMethod(ActionEvent event)
{
    m_jsCallTrigger.trigger();
}

```

### Security Issues

Since Java 1.7 there are certain extended security mechanisms that you need to pay attention to, when communicating from JavaScript to the applet processing: by default the first time when a JavaScript program calls the applet a security dialog pops up, asking the user if the communication should be blocked or if it should be executed.

To prevent this message coming up you need to update the client jar-files - the ones you find in the directory “/eclnt/lib” of your web application. The applet that is embedded into the HTML page is part of the file “eclnt.jar” within this directory. This jar-file needs to be updated, so that the manifest file (META-INF/MANIFEST.MF) contains a list of server addresses: pages that are loaded from this server are allowed to open up a communication between their JavaScript and the applet, without the user having to confirm explicitly.

The following text is taken from a MANIFEST.MF file:

```

Manifest-Version: 1.0
Ant-Version: Apache Ant 1.9.1
Application-Name: CC Enterprise Client
Built-By: CaptainCasa
Built-Timestamp: 20131016_071441
CC-signing-info: Signed with validated CaptainCasa certificate
Permissions: all-permissions
Created-By: 1.6.0_25-b06 (Sun Microsystems Inc.)
Caller-Allowable-Codebase: www.captaincasademo.com 127.0.0.1 127.0.0.1
:8080 localhost:8080
Codebase: *
...
...

```

Please find more information about the “Called-Allowable-Codebase” attribute - and also further information on security issues at <http://docs.oracle.com/javase/7/docs/technotes/guides/jweb/manifest.html>.

When updating a client side JAR file then you have to sign all client JAR files with a signature of your own. Please find corresponding information about signing within the documentation area of [www.CaptainCasa.com](http://www.CaptainCasa.com) (“tech doc” section).

## Opening HTML Pages from the CaptainCasa Enterprise Client Applet

In case you “just” want to open other HTML pages from your CaptainCasa application, there is a special component to support you directly: APPLICATIONCONTEXTSHODOCUMENT.

In the component you need to define the URL and the target of the page that you want to show. When invoking the component's trigger then the CaptainCasa Enterprise Client applet will by default talk to the applet context on client side, and invoke the applet context's function to show the document.

...unfortunately we experienced situations in which Microsoft Internet Explorer crashed completely after having accessed the applet context and after switching to a different HTML page. For this reason there is a way to call the opening of pages by using JavaScript. Please check the attributes of APPLICATIONCONTEXTSHOWDOCUMENT.

When using JavaScript then the applet talks to the following JS-function:

```
<script type="text/javascript">
// this code is used by APPLETCONTEXTSHOWDOCUMENT - if this is configured to
// use java script instead of directly contacting the context
function ccOpenPageInTarget(ccTarget,ccURL)
{
    var ccwindow = window.open(ccURL,ccTarget);
    ccwindow.focus();
}
</script>
```

The function is automatically part of the HTML page when opening CaptainCasa pages via “.ccapplet”. And it is automatically generated into the HTML code that is generated using the corresponding tool within the Layout Editor.

---

## Embedding the CaptainCasa Client into Portals

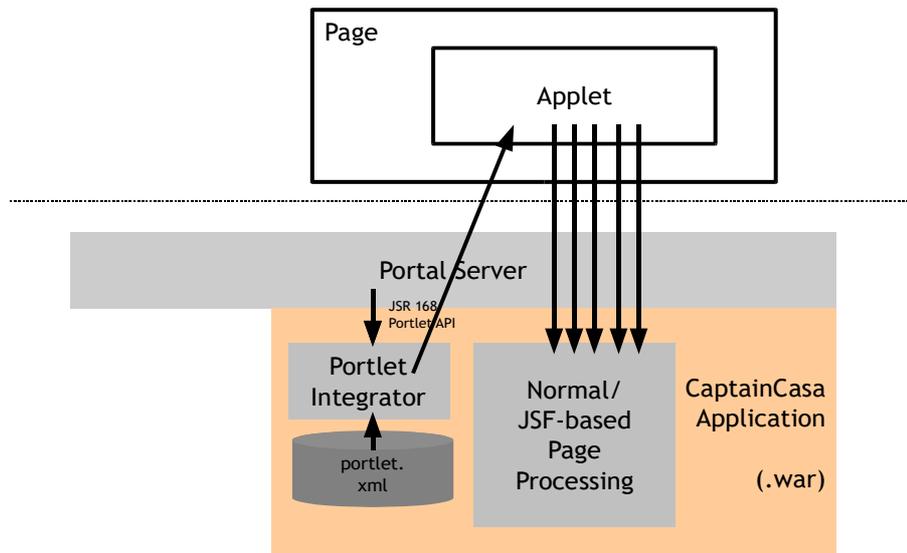
It is possible to integrate the CaptainCasa applet client into a portal server as well. By default the portal standard “JSR 168” is supported.

### Overview

The CaptainCasa Enterprise Client is a plug-in based user interface concept - integration into the browser is done using applet (or webstart) technology.

Portal server are HTML based by default - i.e. they render “big” portal pages out of “small” page fragments. Consequence: plug-in based user interfaces need to integrate by rendering their plug-in definition into the portal page.

This is exactly what the default portlet integration does:



A portlet is provided (called “Portlet Integrator”) that internally renders a frame definition. The frame definition is the one that is rendered into the portal page. Within the frame the applet is created, itself then communicating back to the application.

### Generic Portlet “PortletIntegrator”

It's the task of the portlet implementation to render the frame and internally open up the CaptainCasa Client applet and pass the information to the applet which page is the one to be shown. This could be done in a hard coded way - or in a configure-able way, so that the integration of a certain page is done by configuration only.

The following code is the one of the generic “PortletIntegrator” class. You may either directly use this class (it is part of ecIntjsfserver.jar) or you may write your own portlet integration, taking the code as example implementation:

```

package org.ecInt.jsfserver.portlet;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.Date;

import javax.portlet.GenericPortlet;
import javax.portlet.PortletConfig;
import javax.portlet.PortletContext;
import javax.portlet.PortletException;
import javax.portlet.PortletRequest;
import javax.portlet.PortletSession;
import javax.portlet.RenderRequest;
import javax.portlet.RenderResponse;
import javax.portlet.WindowState;

import org.ecInt.jsfserver.util.HttpSessionAccess;
import org.ecInt.util.log.CLog;

/**
 * Generic portlet inetgrator: the CaptainCasa application will be
 * started as applet within an IFRAME. Page name and page attributes
 * can be passed by specifying a corresponding portlet.xml file.
 */
public class PortletIntegrator extends GenericPortlet
{
    // -----
    // members
    // -----

    public static String PREF_CCPAGE = "ccpage";
    public static String PREF_NORMALWIDTH = "normalwidth";
    public static String PREF_NORMALHEIGHT = "normalheight";
    public static String PREF_MAXWIDTH = "maxwidth";
    public static String PREF_MAXHEIGHT = "maxheight";

    public static String CCPORLETSESSIONID = "ccportletsessionid";

```

```

public static String CCLASTPORTLETREQUEST = "cclastportletrequest";

static PortletContext s_portletContext;
static PortletConfig s_portletConfig;

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

/** */
public void init() throws PortletException
{
    super.init();
    s_portletContext = getPortletContext();
    s_portletConfig = getPortletConfig();
}

// -----
// static utility methods that can be used from CaptainCasa page processing
// -----

public static PortletContext getCurrentPortletContext()
{
    return s_portletContext;
}

public static PortletConfig getCurrentPortletConfig()
{
    return s_portletConfig;
}

public static PortletRequest getCurrentPortletRequest()
{
    return
(PortletRequest)HttpSessionAccess.getCurrentHttpSession().getAttribute(PortletIntegrator.CC
LASTPORTLETREQUEST);
}

public static String getCurrentPortletSessionId()
{
    return
(String)HttpSessionAccess.getCurrentHttpSession().getAttribute(PortletIntegrator.CCPORTLETS
SESSIONID);
}

// -----
// portlet "page" generation
// -----

protected void doView(RenderRequest request,
                      RenderResponse response) throws PortletException, IOException
{
    // read page url from preferences
    String width;
    String height;
    if (request.getWindowState().equals(WindowState.MAXIMIZED))
    {
        width = request.getPreferences().getValue(PREF_MAXWIDTH, "100%");
        height = request.getPreferences().getValue(PREF_MAXHEIGHT, "600");
    }
    else
    {
        width = request.getPreferences().getValue(PREF_NORMALWIDTH, "100%");
        height = request.getPreferences().getValue(PREF_NORMALHEIGHT, "400");
    }
    // response
    String pageName = request.getPreferences().getValue(PREF_CC_PAGE, "/empty.jsp");
    if (pageName.startsWith("/") == false)
        pageName = "/" + pageName;
    String routedURL = request.getContextPath() + pageName;
    // append portlet session id
    if (routedURL.indexOf('?') < 0)
        routedURL += "?" + CC_PORTLET_SESSION_ID + "=" + request.getPortletSession().getId();
    else
        routedURL += "&" + CC_PORTLET_SESSION_ID + "=" + request.getPortletSession().getId();
    // writing request into session context so that it can be accessed from http
    session // context later on

    request.getPortletSession().setAttribute(CC_PORTLET_SESSION_ID, request.getPortletSession().get
    Id(), PortletSession.APPLICATION_SCOPE);

    request.getPortletSession().setAttribute(CCLASTPORTLETREQUEST, request, PortletSession.APPLIC
    ATION_SCOPE);
    // creating iframe to hold the applet
    CLog.L.log(CLog.LL_INF, "Starting applet in portlet: " + routedURL);
    response.setContentType("text/html");
    PrintWriter pw = response.getWriter();
}

```

```

        pw.println("<iframe width='"+width+"' height='"+height+"' src='"+routedURL+"'
style='border:0;padding:0'>");
        pw.println("</iframe>");
    }

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

```

The most important part of the class is the rendering of the frame, at the very bottom of the code. The portal page received back an iframe definition with a certain height, width and URL to show.

All the height/width/URL parameters are dynamically derived from the portlet preferences, that are part of the portlet.xml definition.

## Configuration via portlet.xml

In the portlet.xml definition you defined the portlets of your application. Each portlet definition need to contain certain information as portlet preferences, so that the generic PortletIntegrator can render the portal-frame definition correctly.

There is a “portlet.xml\_template” below each's /WEB-INF/ directory that you can use as template. The following definition is an example:

```

<portlet-app xmlns="http://java.sun.com/xml/ns/portlet/portlet-app_1_0.xsd"
version="1.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/portlet/portlet-app_1_0.xsd
http://java.sun.com/xml/ns/portlet/portlet-app_1_0.xsd"
id="CC Demo Portlets">

  <portlet>
    <description>Demoworkplace</description>
    <portlet-name>Demoworkplace</portlet-name>
    <display-name>Demoworkplace</display-name>
    <portlet-class>org.ec1nt.jsfserver.portlet.PortletIntegrator</portlet-class>
    <expiration-cache>0</expiration-cache>
    <supports>
      <mime-type>text/html</mime-type>
      <portlet-mode>view</portlet-mode>
    </supports>
    <portlet-info>
      <title>Demoworkplace</title>
      <short-title>Demoworkplace</short-title>
    </portlet-info>
    <portlet-preferences>
      <preference><name>ccpage</name><value>workplace.workplace.ccaplet?
ccsession=true</value></preference>
      <preference><name>normalwidth</name><value>100%</value></preference>
      <preference><name>normalheight</name><value>400</value></preference>
      <preference><name>maxwidth</name><value>100%</value></preference>
      <preference><name>maxheight</name><value>700</value></preference>
    </portlet-preferences>
  </portlet>

  <portlet>
    <description>DemoHelloworld</description>
    <portlet-name>DemoHelloworld</portlet-name>
    <display-name>DemoHelloworld</display-name>
    <portlet-class>org.ec1nt.jsfserver.portlet.PortletIntegrator</portlet-class>
    <expiration-cache>0</expiration-cache>
    <supports>
      <mime-type>text/html</mime-type>
      <portlet-mode>view</portlet-mode>
    </supports>
    <portlet-info>
      <title>DemoHelloworld</title>
      <short-title>DemoHelloworld</short-title>
    </portlet-info>
    <portlet-preferences>
      <preference><name>ccpage</name><value>workplace.demoHelloworld.ccaplet?
ccsession=true</value></preference>
      <preference><name>normalwidth</name><value>100%</value></preference>
      <preference><name>normalheight</name><value>300</value></preference>
      <preference><name>maxwidth</name><value>100%</value></preference>
      <preference><name>maxheight</name><value>300</value></preference>
    </portlet-preferences>
  </portlet>

```

```
</portlet>
</portlet-app>
```

There are two portlets defined:

- Each portlet definition uses the class “PortletIntegrator” to provide the portlet interface.
  - Each portlet definition contains preferences in which the page and sizes are defined.
  - The page that you pass as “ccpage” parameter is defined in the same way as explained with “Dynamic Applet Integration”. Example: the page “/workplace/demohelloworld.jsp” is transferred into “workplace.demohelloworld.ccaplet”.
  - You typically add the parameter “ccsamesession=true” to the page that you pass, otherwise each rendering of the applet will create a new http-session on server side!
- ...and that's it!

## Accessing Portal Information from CaptainCasa JSF Processing

When running a CaptainCasa page as portlet, then you may want to access the portal server's API, e.g. in order to get the single sign on credentials.

The class PortletIntegrator provides corresponding static methods to do so:

```
public static PortletContext getCurrentPortletContext() {...}
public static PortletConfig getCurrentPortletConfig() {...}
public static PortletRequest getCurrentPortletRequest() {...}
public static String getCurrentPortletSessionId() {...}
```

## Pay Attention...!

Well, now you can integrate CaptainCasa pages as portlets into portal pages. That's a big advantage, of course! The performance of the application running within the CaptainCasa Client is the same as running the client “just normally”.

But: you need to be aware of certain disadvantages as well. These disadvantages apply to all plug-in based user interface client concepts:

The main problem of portal pages is that they are quite frequently re-rendered completely within the client: a certain operation on one portlet (i.e. one fragment of the whole portal page) typically causes a roundtrip to the server. Consequence: the whole page is re-built on server side and sent to the browser client.

In case of plain HTML this means that the browser needs to re-render quite a lot of HTML. If this HTML contains a plug-in to be rendered then this plug-in is re-opened/-rendered with every roundtrip of the portal page.

---

## Starting Browser Window out of CaptainCasa Client

There are scenarios in which you may want to start a native browser (Internet Explorer, Mozilla, ...) with a certain URL from your CaptainCasa application. The browser opens up in an own window - running completely decoupled from the CaptainCasa client processing.

Use the component JSHOWURL to do so: in this component you pass a URL as parameter + can trigger the opening of the browser. The opening can be controlled in several modes:

- (A) You can open the browser by using the client side Java Desktop API. The API is

available both within the webstart and the applet environment.

- (B) You can open the browser by using the client side Applet Context API - this API is only available if the client is started as applet.
- (C) You can open the browser by using the JavaScript interface, explained in the previous chapter.

The mode that you should select by default is (A) - it is the usage mode which is independent from the way the client is started. The desktop API selects your favorite browser, i.e. it follows your operating system preferences.

Why are there other modes as well? - (B) has the mini-feature, that the browser is started out of the existing browser environment. This means, that if the user uses CaptainCasa client as applet within Internet Explorer, then the browser that will be started, will also be Internet Explorer - regardless of the operating system default browser settings.

Well, and (C) uses JavaScript, i.e. you can also use this mode to start the page within a dedicated target frame, so that the page does not show up as popup but in a certain area of your screen.

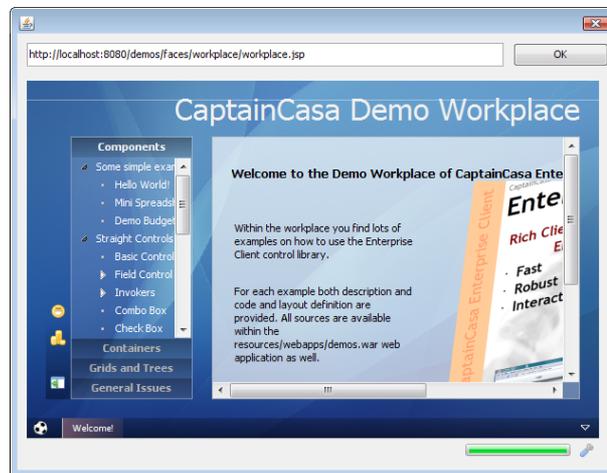
By the way: how can you find out at server side, if the user is running the client as applet or as web start application? - By accessing: `HttpSessionAccess.getCurrentClientType()`.

---

## Swing Integration

In some cases you may have an existing Swing based application, into which you want to place the CaptainCasa Enterprise Client. To do so you need to work with the object "PageBrowser", which is the central component to contain a rich client page.

Example: the following Swing application contains one field, one button and the Enterprise Client:



The code of the Swing program is:

```
package org.ecInt.client.ztest;

import java.awt.Color;
import java.awt.EventQueue;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.ComponentEvent;
import java.awt.event.WindowEvent;
import java.util.HashMap;
import java.util.Map;

import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JFrame;
import javax.swing.JTextField;
import javax.swing.UIManager;
```

```

import org.eclnt.client.controls.util.DefaultComponentListener;
import org.eclnt.client.controls.util.DefaultWindowListener;
import org.eclnt.client.page.PageBrowser;
import org.eclnt.client.util.valuemgmt.Scale;

public class IntegrationExample extends javax.swing.JPanel
{
    static class MyWindowListener extends DefaultWindowListener
    {
        public void windowClosing(WindowEvent e)
        {
            System.exit(1);
        }
    }

    JTextField m_textField;
    JButton m_button;
    PageBrowser m_pageBrowser;
    JFrame m_frame;
    IntegrationExample m_this = this;

    public static void main(String[] args)
    {
        try
        {
            UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        }
        catch (Exception e) {}
        JFrame frame = new JFrame();
        IntegrationExample inst = new IntegrationExample(frame);
        frame.add(inst);
        frame.setBounds(0,0,800,600);
        frame.setVisible(true);
    }

    public IntegrationExample(JFrame frame)
    {
        m_frame = frame;
        initGUI();
    }

    private void initGUI()
    {
        try
        {
            setLayout(null);
            setSize(1024,768);
            // field
            m_textField = new JTextField();
            m_textField.setBounds(10,10,500,25);

m_textField.setText("http://localhost:8080/demos/faces/workplace/demomanyfields.jsp");
            add(m_textField);
            // button
            m_button = new JButton();
            m_button.setText("OK");
            m_button.setBounds(520,10,100,25);
            m_button.addActionListener(new ActionListener()
            {
                public void actionPerformed(ActionEvent e)
                {
                    m_pageBrowser.switchToUrl(m_textField.getText());
                }
            });
            add(m_button);
            EventQueue.invokeLater(new Runnable()
            {
                public void run()
                {
                    // page browser:
                    Map<String,String> configParams = new HashMap<String, String>();
                    PageBrowser.initClientParameters(configParams);
                    m_pageBrowser = new PageBrowser("http://localhost:8080", // server/port
port of URL
"/demos/faces/workplace/demomanyfields.jsp", // page part of URL
0,
m_frame, // owning window
m_frame, // owning frame
false, // show header
true); // show footer

                    add(m_pageBrowser);
                    // resize...
                    sizeContent();
                    addComponentListener(new DefaultComponentListener()
                    {

```

```

        public void componentResized(ComponentEvent e)
        {
            sizeContent();
        }
        public void componentShown(ComponentEvent e)
        {
            sizeContent();
        }
    });
});
}
catch (Exception e)
{
    e.printStackTrace();
}
}
private void sizeContent()
{
    m_pageBrowser.setBounds(10,
                            50,
                            getWidth()-20-getInsets().left-getInsets().right,
                            getHeight()-70-getInsets().top-getInsets().bottom);
    m_frame.validate();
}
}
}

```

## Page not coming up properly?

There are situations in which the PageBrowser component does not start up as expected - or it only is properly sized when explicitly resizing the window.

When looking into the code above you see that the generation of the PageBrowser instance is done not directly within the “initGUI()” method, but is done asynchronously, by being embedded in an EventQueue.invokeLater() call.

Please follow this type of invoking the CaptainCasa client.

## Passing Parameters to the Server side Application at Startup

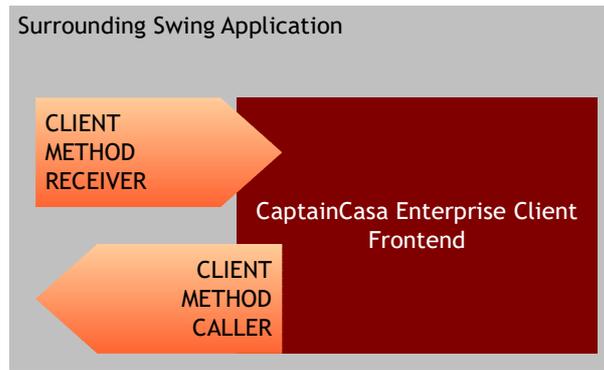
Once again you may use the http-parameters to transfer information to the server side processing - just as explained in the previous chapter (Applet Integration). Just append the parameters via “?name=value&name=value” to the URL that you pass into the PageBrowser instance - and read the parameters using the HttpServletRequest instance on server side.

## Exchanging Data with the Server side Application at Runtime

There is a certain mechanism that allows to exchange data between the Enterprise Client Application and the embedding Swing application. This may sound strange because the Enterprise Client application is running on server side, and only the frontend part of the application is running on “Swing-side”.

The solution is simple: the exchange of data is done by special, invisible UI components.

- CLIENTMETHODRECEIVER is a component that receives a message on client side by providing a simple Java API to the surrounding Swing application.
- CLIENTMETHODCALLER is a component that calls a method on server side, triggered by a server response.



## CLIENTMETHODRECEIVER Component

Place this component into your page, best located inside the out-most page, in case you use ROWINCLUDEs to nest one page into the next. Make sure that there is only one representation of the component available within your page(s).

Now your Swing application can call the following JAVA API:

```
CLIENTMETHODRECEIVER.sendMessageToServer(String message);
```

The message that you pass as parameter is passed to the CLIENTMETHODRECEIVER component. The component triggers a server roundtrip - the same way as a roundtrip is triggered by a user event (e.g. button pressed). On server side the CLIENTMETHODRECEIVER's action listener is called and receives an event of type "BaseActionEventClientMethodReceive". From the event you get the message by calling getMessage() function.

## CLIENTMETHODCALLER Component

This is the other direction: you want to pass a message from your server side Java application to your client side Swing application.

In the CLIENTMETHODCALLER component you need to define:

- The class name of your Swing application that should be called.
- The message that you want to send.
- A trigger that actually triggers the data sending of the component instance to the surrounding Swing application.

In the client the following happens:

- With each trigger a new Object of the Swing class (defined by class name) is created. The object is required to support interface "ICLIENMETHODOCALLERTarget". The message is passed through the method "processMessageFromServer":

```
public interface ICLIENMETHODOCALLERTarget
{
    public void processMessageFromServer(String message);
}
```

## ACTIVEX Integration

You can embed ActiveX controls into a page by using the ACTIVEX component. The ACTIVEX component provides an interface that allows to define the ActiveX control and to set/get String parameters. It is possible to call methods with string parameters as well.

When using the ACTIVEX component you must be aware of:

- An additional client side library is required (swt.jar) which needs to be referenced within the Applet (.html) or Web Start (.jnlp) definition. When using the tool for creating the applet/web start definitions, that is part of the Layout Editor, then please switch on the corresponding check box in order to use native components.
- Of course the usage of ActiveX controls is limited to the operating system “Windows”. You should only use the ActiveX integration for dedicated purposes, e.g. integration of sub devices. Be aware as well that the ActiveX is running in the same process as the Java client itself. A crash of the ActiveX control will also a crash of the Java client.

An example for using the AXTIVEX component is part of the demo workplace:

```

...
<t:row id="g_2">
  <t:col distance id="g_3" width="120" />
  <t:activex id="g_4"
    actionListener="#{d.demoActivex.onReceiveData}"
    command="#{d.demoActivex.activexCommand}"
    commandCallback="true"
    height="100%"
    initCommand="Navigate(http://www.google.com)"
    progid="Shell.Explorer"
    propertyInterest="get(Application)"
    width="100%" />
</t:row>
...

```

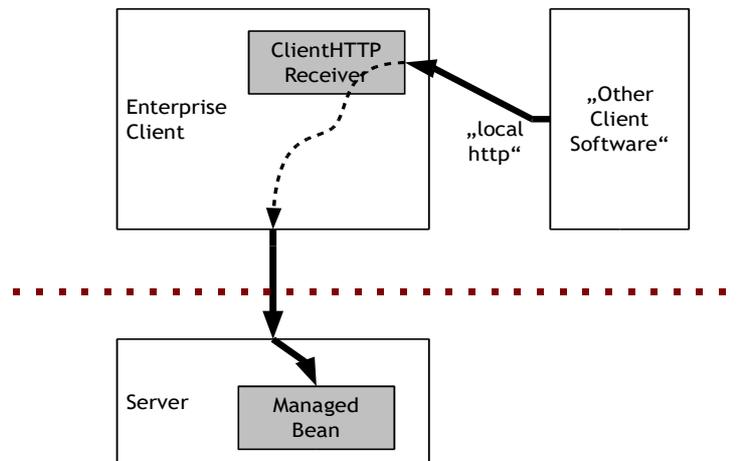
Let's take a closer look onto the attributes:

- PROGID - this is the ActiveX program id for the ActiveX control. In the example the Internet Explorer is integrated, having the id “Shell.Explorer”. Check the documentation of your ActiveX control to obtain the corresponding program id.
- INITCOMMAND - this is a command that is executed after the creation of the ActiveX control on client side. In the example the command “Navigate(<http://www.google.com>)” is sent to the ActiveX control, causing the embedded Internet Explorer to render the corresponding page. The command is a method of the ActiveX component. The method may have parameters, currently only String parameters are supported. You can concatenate several commands by semicolon: “command1;command2”.
- COMMAND - this is a command that is passed during the life cycle of the component. The structure is the same as with the INITCOMMAND parameter, i.e. it contains one or more method calls that are passed to the ActiveX control. The COMMAND is executed on client side every time it changes.
- PROPERTYINTEREST - it is possible to read string properties from the ActiveX control and send them to the server side processing. The reading of ActiveX parameters is done by default every time the client communicated to the server. You can specify attributes to be collected (“get(Application)”) or you can specify methods that return a string value of the ActiveX control. As with the command you can concatenate several property interests by semicolon: “get(prop1);get(prop2)”. If there's a change in one of the properties then all of the property values are sent to server side and cause the invoking of the ACTIONLISTENER method behind the ACTIVEX component. The values can be accessed through the API of BaseActionEventActiveX.
- COMMANDCALLBACK - normally the collecting of property values (specified by PROPERTYINTEREST) is done every time the client talks to the server. You can enforce the communication from client to server by setting COMMANDCALLBACK to “true”. In this case a server call back is done directly after invoking the COMMAND of the ActiveX control.

---

## Integrating Client Devices by Usage of Client Side http

It is possible to add a component to the user interface client that itself provides a mini-http server. This server listens to http requests on a defined port and transfers the content of the http request to the server side processing.



The component's name is CLIENTHTTPRECEIVER. Every time a request is received by the component is invokes its action listener and passes the query string of the request to the server side processing. Please have a look into the demo workplace for seeing how to use.

You can compare the component with an “invisible button”: the button is not activated by the user but is activated by an http request.

---

## Integration of Client Devices by Usage of Serial Interface

The component CLIENTSERIALRECEIVER allows to listen to a serial port and transfer data coming from the serial port to the server side.

The following JSP code shows the usage of the component:

```
<t:beanprocessing id="g_1">
  <t:clientserialreceiver id="g_2"
    actionListener="#{d.DemoSerialScanner.onSerialAction}"
    rendered="#{d.DemoSerialScanner.serialScannerRendered}" />
</t:beanprocessing>
<t:rowbodypane id="g_3">
  ...
  ...
</t:rowbodypane>
```

There are certain attributes (like port name, baud rate, etc.) that you can maintain with the component - and that you are familiar with when working with the serial interface. Please check the attributes and the values that you can define within the Layout Editor.

On server side the value that is passed through the serial interface is passed as part of the action event:

```
public class DemoSerialScanner implements Serializable
{
  ...
  ...
  public void onSerialAction(ActionEvent event)
  {
    if (event instanceof BaseActionEventClientSerialReceive)
    {
      BaseActionEventClientSerialReceive baecsr =
(BaseActionEventClientSerialReceive)event;
      m_scannedvalue = baecsr.getSerialValue();
    }
  }
}
```

```
    ...  
    ...  
}
```

The connection from Java to the serial interface is done through a certain “.jar” file, a certain library (“.dll”) and certain property settings. Due to the fact that this is a quite “native level of computing”, it is not possible to automatically transfer and install all this on client side. You need to...

- ...extend the list of libraries that are required by the applet (or by webstart) and need to include the library “/ecInt/lib/commapi.jar” as well. - When calling the applet/webstart by “.ccapplet/.ccwebstart” then add “?cclibs=comm” to the URL.
- ...transfer two files into the Java Virtual Machine that you are using. Both files are contained within the CaptainCasa delivery - within the “resources/commapi” folder:
  - The file “win32com.dll” needs to be copied into the /bin directory of your Java runtime.
  - The file “javax.comm.properties” needs to be copied into the /lib directory of your Java runtime.

When working with other operating systems, please check the information on the “Java Comm API”.

---

## Integration of Client Devices by Usage of generic Interface “ISubDevice”

### Component CLIENTSUBDEVICE, Interface ISubDevice

While some of the components for integrating client side subdevices are serving a certain communication protocol (e.g. serial protocol, http protocol), there is also a generic subdevice component, that is not bound to any technical communication protocol - but just is an interface to an abstract subdevice.

A subdevice has the following life cycle:

- It is created.
- It is used.
  - Messages are sent from the subdevices to the client processing and are transferred to the server side of CaptainCasa, so that the application logic can react correspondingly,
  - Messages are sent to the subdevice - triggered by the server processing.
- It is closed. Either closing is done explicitly (by the application logic) or implicitly (by the application logic re-creating the subdevice).

In principal an abstract subdevice is nothing else than an instance to which certain String-content is sent, and from which certain String content is received. The content protocol of what the meaning of a string is, is up to the user of the subdevice: if you e.g. pass XML data via the string(s) or comma separated values - this is your definition!

The client side interface that you have to implement when adding own subdevice implementations is:

```
public interface ISubDevice  
{  
    public void init(String initData, ISubDeviceCallback callback);  
    public void sendData(String sendData);  
}
```

```

    public void close();
}

```

You need to add an implementation of this interface to the client processing, i.e. you need to implement a class with a default constructor (no parameters!) and with an implementation of ISubDevice.

An instance of this class is created when adding component CLIENTSUBDEVICE to you page layout. The parameters are:

- SUBDEVICECLASSNAME - the class name of your implementation, which is taken by the component for dynamically creating an instance of your class at runtime
- INITDATA - the string that is passed into the init() method of ISubDevice.
- INITTRIGGER - by default an instance of ISubDevice is created when the component is created on client side - you do not have to explicitly trigger the first time creation. But, you may later on decide from your application logic processing to re-open the subdevice. In this case you need to trigger this explicitly.
- SENDVALUE, SENDDATATRIGGER - these are the attributes to send data content to the subdevice (which is passed to method ISubDevice.sendData(...)). SENDVALUE contains the content of what is sent. SENDTRIGGER actually triggers the client component to send the data.
- CLOSETRIGGER - with this trigger you can explicitly close the subdevice. The method ISubDevice.close() is called on client side.
- ACTIONLISTENER - the action listener that is called on server side when client side data is received.

Sending of data from the subdevice to the component (and then to the server side processing) is done by using the interface "ISubDeviceCallback". An instance of this interface is passed to your processing when method ISubDevice.init(..) is called:

```

public interface ISubDeviceCallback
{
    public void reactOnSendDataToServer(String receivedData);
}

```

The client side processing will call the action listener, passing an event of type "BaseActionEventClientSubDevice".

## Example

The following code is the client side implementation of ISubDevice:

```

package org.eclnt.client.util.subdevice;

public class TestSubDevice implements ISubDevice
{
    /**
     * Thread simulating some subdevice events - sending data to
     * the server.
     */
    class MyThread extends Thread
    {
        boolean i_continue = true;
        @Override
        public void run()
        {
            while (i_continue)
            {
                try { this.sleep(5000); } catch (Throwable t) {}
                if (i_continue)
                {
                    m_callBack.reactOnSendDataToServer("## Message from subdevice,
timestamp " + System.currentTimeMillis());
                }
            }
        }
    }
}

```

```

    }
}

MyThread m_thread;
ISubDeviceCallback m_callBack;

public void init(String initData, ISubDeviceCallback callBack)
{
    System.out.println("#####");
    System.out.println("## TestSubDevice - INIT");
    System.out.println("## initData - " + initData);
    System.out.println("#####");
    m_callBack = callBack;
    m_thread = new MyThread();
    m_thread.start();
}

public void sendData(String sendData)
{
    System.out.println("#####");
    System.out.println("## TestSubDevice - SEND");
    System.out.println("## sendData - " + sendData);
    System.out.println("#####");
}

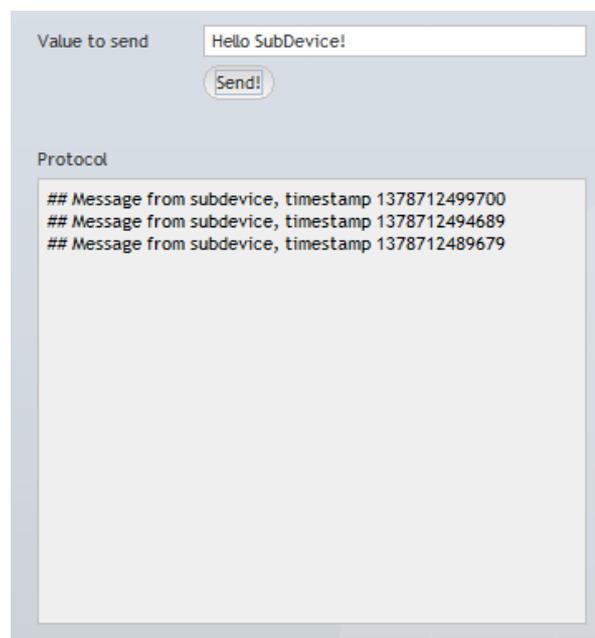
public void close()
{
    System.out.println("#####");
    System.out.println("## TestSubDevice - CLOSE");
    System.out.println("#####");
    m_thread.i_continue = false;
    m_thread = null;
}
}
}

```

You see...:

- Messages/Calls from the server side (init, sendData, close) are output to the console.
- A thread simulates events coming from a technical subdevice (e.g. a scanner). The interface ISubDeviceCallback is used to send this information.

The interface is referenced in the following page:



The page definition is:

```

<t:beanprocessing id="g_1">
  <t:clientsubdevice id="g_2"

```

```

        actionListener="#{d.DemoClientSubDevice.onSubDeviceAction}"
        initvalue="Some init value"
        sendtrigger="#{d.DemoClientSubDevice.sendTrigger}"
        sendvalue="#{d.DemoClientSubDevice.sendValue}"
        subdeviceclassname="org.ecInt.client.util.subdevice.TestSubDevice" />
</t:beanprocessing>
<t:rowdemobodypane id="g_3" rowdistance="5"
  objectbinding="#{d.DemoClientSubDevice}">
  <t:row id="g_4">
    <t:label id="g_5" text="value to send" width="100" />
    <t:field id="g_6" text="#{d.DemoClientSubDevice.sendValue}"
      width="100%" />
  </t:row>
  <t:row id="g_7">
    <t:coldistance id="g_8" width="100" />
    <t:button id="g_9"
      actionListener="#{d.DemoClientSubDevice.onSendAction}"
      text="Send!" />
  </t:row>
  <t:rowdistance id="g_10" height="20" />
  <t:row id="g_11">
    <t:label id="g_12" text="Protocol" />
  </t:row>
  <t:row id="g_13">
    <t:textarea id="g_14" enabled="false" height="100%"
      text="#{d.DemoClientSubDevice.protocol}" width="100%" />
  </t:row>
</t:rowdemobodypane>

```

The CLIENTSUBDEVICE component is listed below the tag BEANPROCESSING.

The server side implementation is:

```

package workplace;

import java.io.Serializable;

import javax.faces.event.ActionEvent;

import org.ecInt.editor.annotations.CCGenClass;
import org.ecInt.jsfserver.elements.events.BaseActionEventClientSubDevice;
import org.ecInt.jsfserver.elements.util.Trigger;
import org.ecInt.workplace.IWorkpageDispatcher;
import org.ecInt.workplace.WorkpageDispatchedBean;

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

public class DemoClientSubDevice
  extends DemoBase
  implements Serializable
{
  String m_sendValue;
  public String getSendValue() { return m_sendValue; }
  public void setSendValue(String value) { this.m_sendValue = value; }

  Trigger m_sendTrigger = new Trigger();
  public Trigger getSendTrigger() { return m_sendTrigger; }

  String m_protocol = "";
  public String getProtocol() { return m_protocol; }
  public void setProtocol(String value) { this.m_protocol = value; }

  public DemoClientSubDevice(IWorkpageDispatcher workpageDispatcher)
  {
    super(workpageDispatcher);
  }

  public void onSendAction(ActionEvent event)
  {
    m_sendTrigger.trigger();
  }

  public void onSubDeviceAction(ActionEvent event)
  {
    if (event instanceof BaseActionEventClientSubDevice)
    {
      m_protocol = ((BaseActionEventClientSubDevice) event).getSubDeviceSendValue() +
        "\n" + m_protocol;
    }
  }
}

```

## Deployment

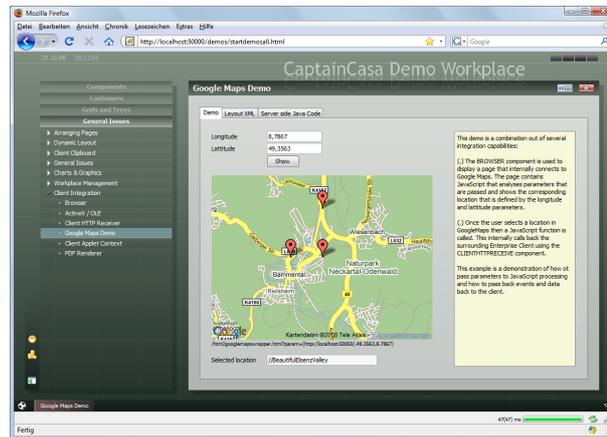
Your implementation of ISubDevice needs to be added to the client processing of CaptainCasa Enterprise Client.

- Package your implementation into an own JAR-file.
- Sign all client jar files that you use within the client processing. All JAR-files need to have the same signature! So you both have to sign standard JAR-files from CaptainCasa and your own one(s). Pay attention: first un-sign existing JAR-files before signing them.
- Add you library within the applet or jnlp definition that you use to start your client.

## Integration of HTML Pages with embedded JavaScript

Using the BROWSER component and the CLIENTHTTPRECEIVER component you can build up powerful scenarios in which you can build up communication between the Enterprise client and the embedded JavaScript.

There's an example added to the demo workplace showing the coupling of an Enterprise Client page and a Google Maps application that is shown inside:



The principles behind are quite simple. First have a look onto the page definition:

```
...
...
<t:row id="g_4">
  <t:label id="g_5" text="Longitude" width="100" />
  <t:formattedfield id="g_6" format="bigdecimal" formatmask="dec4"
    value="#{d.DemoGoogleMaps.longitude}" width="100" />
</t:row>
<t:row id="g_7">
  <t:label id="g_8" text="Latitude" width="100" />
  <t:formattedfield id="g_9" format="bigdecimal" formatmask="dec4"
    value="#{d.DemoGoogleMaps.latitude}" width="100" />
</t:row>
<t:row id="g_10">
  <t:coldistance id="g_11" width="100" />
  <t:button id="g_12" actionListener="#{d.DemoGoogleMaps.onShow}" text="Show" />
</t:row>
<t:rowdistance id="g_13" height="10" />
<t:row id="g_14">
  <t:browser id="g_15" height="300" url="#{d.DemoGoogleMaps.url}" width="400" />
</t:row>
<t:row id="g_16">
  <t:label id="g_17" font="size:9" text="#{d.DemoGoogleMaps.url}" />
</t:row>
<t:rowdistance id="g_18" height="10" />
<t:row id="g_19">
```

```

<t:label id="g_20" text="Selected location" width="100" />
<t:field id="g_21" enabled="false"
      text="#{d.DemoGoogleMaps.selectedLocation}" width="200" />
</t:row>
...
...

```

The URL that is passed into the BROWSER component is built in the onShow-method:

```

public class DemoGoogleMaps
    extends DemoBase
    implements Serializable
{
    protected String m_selectedLocation = "Not yet selected.";
    public String getSelectedLocation() { return m_selectedLocation; }
    public void setSelectedLocation(String value) { m_selectedLocation = value; }

    protected String m_url = "about:blank";
    public String getUrl() { return m_url; }

    protected String m_latitude = "49.3563";
    public String getLatitude() { return m_latitude; }
    public void setLatitude(String value) { m_latitude = value; }

    protected String m_longitude = "8.7867";
    public String getLongitude() { return m_longitude; }
    public void setLongitude(String value) { m_longitude = value; }

    public DemoGoogleMaps(IDispatcher dispatcher)
    {
        onShow(null);
    }

    public void onShow(ActionEvent event)
    {
        m_url = "/html/googlemapswrapper.html?
param=(http://localhost:50050/,"+m_latitude+" ," +m_longitude+)";
    }

    public void onClientReceive(ActionEvent event)
    {
        if (event instanceof BaseActionEventClientHttpReceive)
        {
            m_selectedLocation =
((BaseActionEventClientHttpReceive)event).getQueryString();
        }
    }
}

```

It contains an appendix “?param=(...,...,...)” with three paramaters:

- The callback URL base - the port is the same as provided by the CLIENTHTTPRECEIVER. As result the embedded page knows how to call back in case of events
- The latitude and longitude

The page that is called is an HTML page opening Google Maps. In addition there is some JavaScript code for taking the parameters out of the URL and interpreting them:

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8"/>
    <title>Google Maps JavaScript API Example</title>
    <script src="http://maps.google.com/maps?
file=api&v=2&key=ABQIAAAAUcxc5t7Z6kYUH4wXxuN6xT718c0Zo6t1Tj76RFh0kYw6QOP
SxRLnp--KuPAQaErCRE1DovJKgfOMw"
      type="text/javascript"></script>
    <script type="text/javascript">
      var urlString = document.location.href;
      var indexStart = urlString.indexOf("param=");

```

```

var indexEnd = urlString.indexOf(")", indexStart);
var urlparam = urlString.substring(indexStart+7, indexEnd);
var urlparams = urlparam.split(",");
var m_urlBase = urlparams[0];
var m_lat = urlparams[1]*1;
var m_lng = urlparams[2]*1;
function load()
{
  if (GBrowserIsCompatible())
  {
    var map = new GMap2(document.getElementById("map"));
    map.setCenter(new GLatLng(m_lat,m_lng), 13);
    // marker
    var point = new GLatLng(49.3563,8.7867);
    var marker = createMarker(point, 'CaptainCasa');
    map.addOverlay(marker);
    point = new GLatLng(49.3663,8.7867);
    marker = createMarker(point, 'CaptainCasaNeighbour');
    map.addOverlay(marker);
    point = new GLatLng(49.3563,8.7767);
    marker = createMarker(point, 'BeautifulElsenzValley');
    map.addOverlay(marker);
  }
}
function createMarker(point, callbackparam)
{
  var marker = new GMarker(point);
  GEvent.addListener(marker, "click",
  function()
  {
    var divcomm = document.getElementById("DIVCOMM");
    var divurl = "<iframe src='" + m_urlBase + "/" + callbackparam + ""
width='500' height='50'>";
    divcomm.innerHTML = divurl;
  });
  return marker;
}
</script>
</head>
<body onload="load()" onunload="GUnload()" topmargin="0" leftmargin="0"
rightmargin="0" bottommargin="0" style="padding:0" style="overflow:hidden"
scroll="no">
  <div id="map" style="width: 400px; height: 300px"></div>
  <div style="width: 0px; height: 0px" id="DIVCOMM">
  </div>
</body>
</html>

```

The callback to the Enterprise Client's CLIENTHTTPRECEIVER component is done using a hidden frame. The hidden frame is dynamically built in the createMarker-function. It uses the URL-base that was passed as client parameter.

In order to increase security you may also pass some security tokens into the JavaScript processing. In this case only callbacks are accepted by the CLIENTHTTPRECEIVER processing that contain a valid token.

---

## PDF Document Integration

There are two ways to go when integrating PDF documents into your Enterprise Client frontend:

- The “native way”: using the BROWSER component
- The “Java way”: using the PDFRENDERER component

The “native way” is quite simple: a plain, native browser is embedded into the Enterprise Client - the URL that you pass is one pointing to a PDF file. The browser will load the corresponding plug-in (e.g. Acrobat Reader) and will render the PDF file. - In case you dynamically create PDF data by your application on server side then you need to store the PDF “somewhere” and access it through a servlet.

The “Java way” means, that on client side a Java-PDF-renderer is used. The renderer is taken from the “PDF Renderer Project” (<https://pdf-renderer.dev.java.net>) - as consequence the PDFRenderer.jar file is required on client side. Please check the documentation available on <https://pdf-renderer.dev.java.net/> for getting detailed information about the PDF version that may be used for PDF documents.

When going the “Java way” then you need to integrate the component PDFRENDERER into your layout definition. The PDFRENDERER provides an attribute PDF - this is the one in which you need to fill the pdf document to be displayed on client side. Because everything which is sent to the client needs to be “in String” you need to convert the pdf document’s byte[] array into a hexadecimal String before.

Please check the demo within the demo workplace for coding details.

## Client-Id Management

In some cases you want to keep information that is associated with the client that is just logged on. In other words: you want to know “which client is actually talking to your server application”.

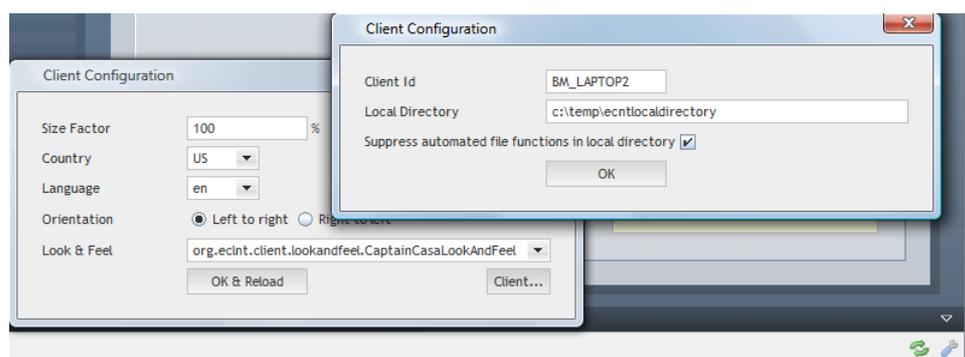
There are two ways of defining and storing a client id:

- The client id may be stored within the client’s temporary directory. This is a special area of the client’s file system - the exact location is defined by the operating system.
- The client id may be stored by usage of cookies. This only works when starting the client via “.ccapplet” and “.ccwebstart” URL. It does NOT work when starting the client via “.html” or “.jnlp”:

The setup of client ids by using the client’s temporary directory is much easier - the setup by usage of cookies was introduced first and is kept for compatibility reasons.

### Defining and Storing the Client Id in the temporary Directory of the Client

This is quite simple: the user just needs to open the configuration dialog, then the “Client...” configuration:



There the user can define the Client Id directly.

### Usage of Cookies

**Please note: the usage of cookies is only available when using the Swing Client - it is NOT available when using the FX Client.**

The following concept that is based on browser cookies only is applied when calling the

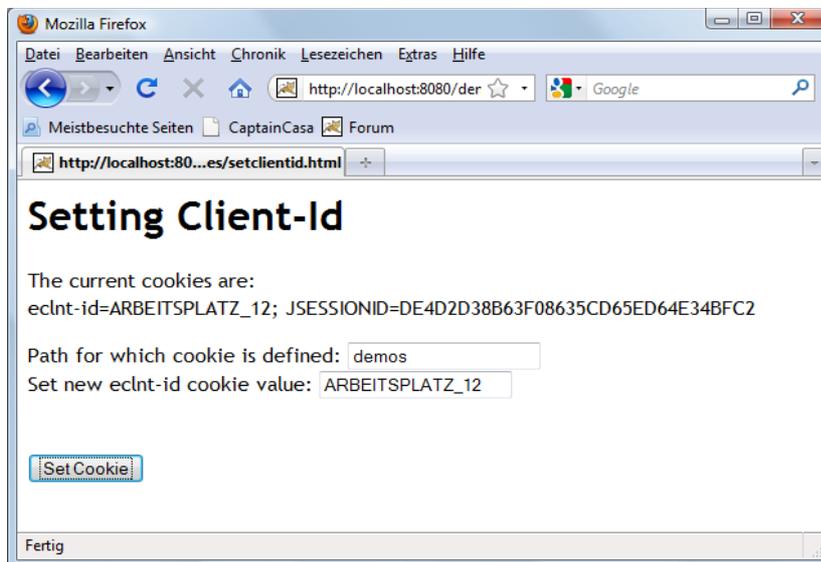
client via “.ccapplet” or “.ccwebstart” URLs. Please check the chapter “Intergation Scenrios” for more details on “.ccapplet”/“.ccwebstart”.

The way it works is:

- In the browser you request a “.ccapplet” or “.ccwebstart” URL in order to open up the CaptainCasa client.
- Both “.ccapplet” and “.ccwebstart” are transferred to a certain servlet provided by CaptainCasa Enterprise Client's server side. In the servlet it is examined if a cookie with the name “eclnt-id” was sent with the request from the client. If yes, then the cookie is read - its content is the id that was already assigned to the client. If not, then a new id is generated and a cookie “eclnt-id” is sent back to the client side - together with the page starting the client.
- The client-id is made available to the application that is opened within the CaptainCasa client, as http-header-parameter “eclnt-id”.

You can explicitly set the client id by calling the following page:

```
http://<host>:<port>/<webapp>/eclntjsfserver/utilpages/setclientid.html
```



In this page you can specify two values:

- The path for the cookie.
- The value of the cookie, i.e. the value of your client-id.

The path typically is set to be the context root of your web application.

## Accessing the Client-Id on Server Side

You can access the client id within your application directly by accessing the http-header information:

```
HttpServletRequest request = getCurrentRequest();  
String clientId = request.getHeader("eclnt-id");
```

There is a convenience function

```
HttpSessionAccess.getCurrentClientId()
```

## Storing dependent Data

The client-id management only assigns one id to the client computer accessing CaptainCasa pages.

The strategy now is, to store all client-specific data that you require from your application's point of view on server side - dependent from the client-id.

---

## Client-Instanceid Management

In addition to the “eclnt-id” that is explained in the previous chapter there is a second client identification, the client instance-id which is transferred via http header parameter “eclnt-instanceid”.

```
HttpServletRequest request = getCurrentRequest();  
String clientId = request.getHeader("eclnt-instanceid");
```

This id is randomly generated for each client instance when the client starts up. It is stable throughout the whole life-cycle of the client.

If you start the client multiple times on one physical client computer, then the “eclnt-id” is the same for all the instances - where as the “eclnt-instanceid” is different.

A typical use case is, if you want to invalidate the current http-session on server side and by this force the client to reload a new session. In this case you can save data on server-side for the “eclnt-instanceid” before invalidating the session - and you can re-access the data later on, if the request for a new session is triggered by the client.

This use case is sometimes used in certain security scenarios, in which you want the logon session to be different from the session that you later on use for running the application. In this case you invalidate the session on server-side, after having parked the authentication information in a way, that you can access it later. The binding id between both sessions is the “eclnt-instanceid”.

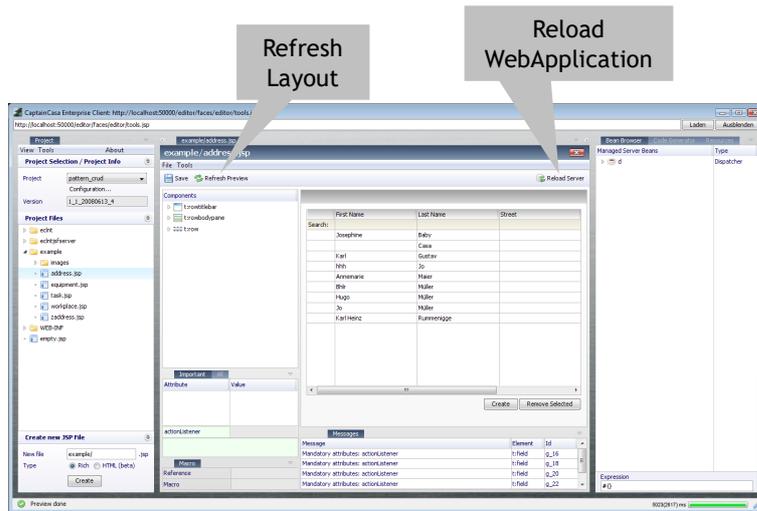
# Using “Hot UI Deployment”

This chapter provides information about an optional to use feature of the server infrastructure of CaptainCasa Enterprise Client: “Hot Deployment”.

## Overview + When to Use

The typical development process when developing user interfaces is:

- You design the .jsp pages within the Layout Editor
- You develop classes which are referenced by the pages via expressions



While changes in .jsp files are recognized by the tool environment immediately (with the next refreshing), changes in the Java classes are recognized in the following way:

- The developer needs to reload the web application.
- During the reload typically a copying of the webcontent directory and a restarting of the web application is done.

Due to the restarting of the web application the runtime environment works on refreshed classes, i.e. the changes that you did within your Java code are visible.

Now, where's the problem?

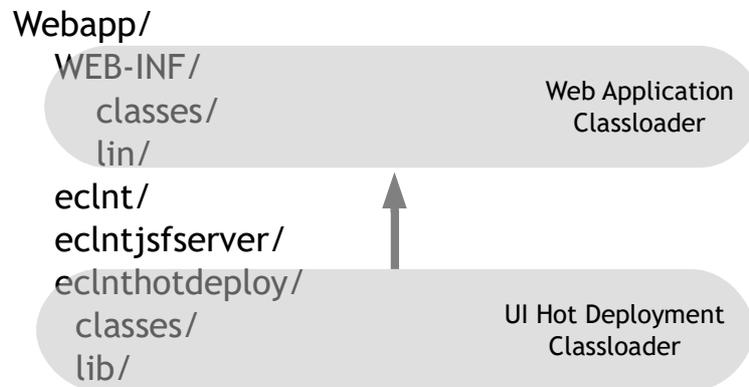
- Well, the reload of the web application may just take some time. In many applications a reload means a re-initialization of the whole application. As a result each class change and each testing of changed classes in the UI area mean a quite long waiting time for the developer.
- On the other hand also the Java VM processing has some limits: because classes that are loaded into the VM are typically not released from the VM memory, there is a constant growth because of constant reloading of classes. Well, there's no way to bypass this fact, but in case of reloading whole applications quite oftenly, the load put onto the memory is quite high. Typical messages you will receive as result are “OutOfMemory” messages, talking about “PermGen Space” that is not sufficient.

The solution of this problem is quite easy: these classes that have to do with UI processing and that are very likely to be changed during the development process need to be separated from these classes which are not changed so frequently. Typical scenarios are:

- You have business logic classes, accessing the database and providing business functions. These classes are “quite stable”.

- You have UI classes (e.g. the beans referenced by expressions of JSP pages) that change quite oftenly.

The way to separate the classes is to use different class loaders. In short: the “stable” classes are running in the normal web application class loader (the one using WEB-INF/classes and WEB-INF/lib), and the “volatile” classes are running in an extra class loader, named “Hot UI Deployment Class Loader”.

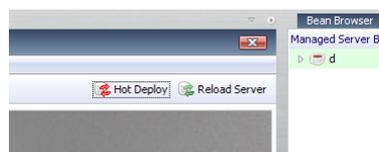


The UI Hot Deployment Classloader is a child of the Web Application Classloader. This means it knows all the classes from its parent. But, vice versa, the Web Application Classloader does not know any classes of the UI Hot Deployment Classloader.

This typically makes sense and is in synch with your normal class dependencies: the UI classes can access logical classes, but logical classes never must access UI classes.

Now having separated UI classes and logical classes you already might guess what the advantage is: you do not have to restart the whole web application in order to see changes that you did within your UI classes. It's enough to restart the UI Hot Deployment Classloader. This takes much less time than restarting the web application.

In the Layout Editor there will be a “Hot Deploy” button next to the “Reload” button for the fast hot-deploying of UI classes:



Summary: you should think about isolating UI classes into an own classloader environment, if reloading the web application is too heavy and as result your UI development process is slowed down significantly. The price you pay is:

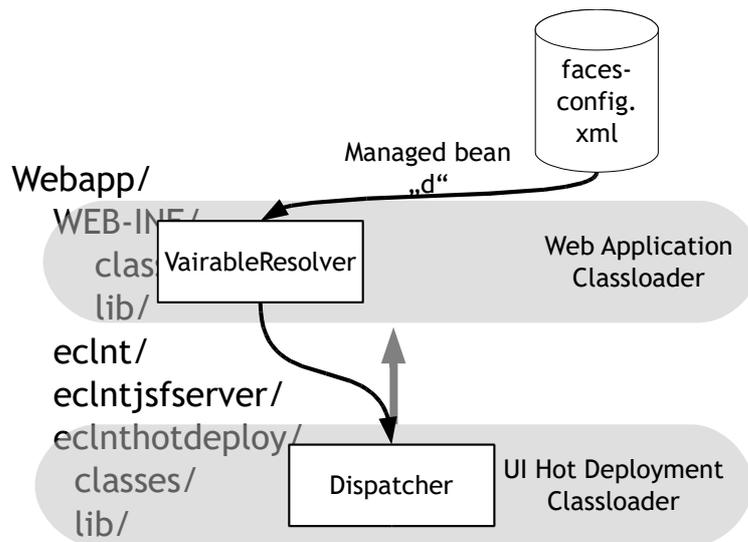
- You need to more consequently separate your UI processing from the rest.
- You need to compile UI classes into a different directory than your logical classes. This means you need to some effort within the configuration of your project within your development environment (e.g. Eclipse).

## Hot UI Deployment Framework Details

### Resolution of Managed Beans

The central location where the separated class loader is used is the resolution of managed beans that are defined in the faces-config.xml file. In the JSF / CaptainCasa server side

framework there is a so called “VariableResolver” that creates managed bean instances for names defined in face-config.xml. CaptainCasa comes with an own implementation of “VariableResolver” that loads resolved objects within the context of an own classloader.



## Configuration

The “UI Hot Deployment Classloader” is not used by default. It requires some extra configuration: there is a configuration file “eclntjsfserver/config/hotdeploy.xml” that controls the class loading. If it is available then class loading will be done using the “UI Hot Deployment Classloader”.

```

<!--
Configuration for the optional hot deployment framework. The directories
that are listed + their containing jar files are added to the classpath
of the hot deployment classloader.
-->

<hotdeploy>
  <webappdir name="/eclnthotdeploy/classes"/>
  <webappdir name="/eclnthotdeploy/lib"/>
</hotdeploy>

```

In the file the directories where the classloader searches for UI classes are listed. The directory can be defined in two ways:

- by “webappdir”: in this case the directory is interpreted as directory of the web content directory. At runtime the webcontent directory is determined via an API provided by the servlet container (e.g Tomcat).
- by “dir”: in this case the directory is interpreted as “absolute directory”. You may either specify a full directory path (e.g. in Windows: “c:\xyz\xyz”) or you may define a relative path (e.g. Windows: “..\xyz\xyz”) - the relative path is then relative to the call of the java-runtime (Windows: java.exe).

That's it!

## (Eclipse) Project Configuration

In your web application you now have two places to store “.jar” and “.class” files:

- the normal “WEB-INF/classes” and “WEB-INF/lib” directories
- the “eclnthotdeploy/classes” and “eclnthotdeploy/lib” directories.

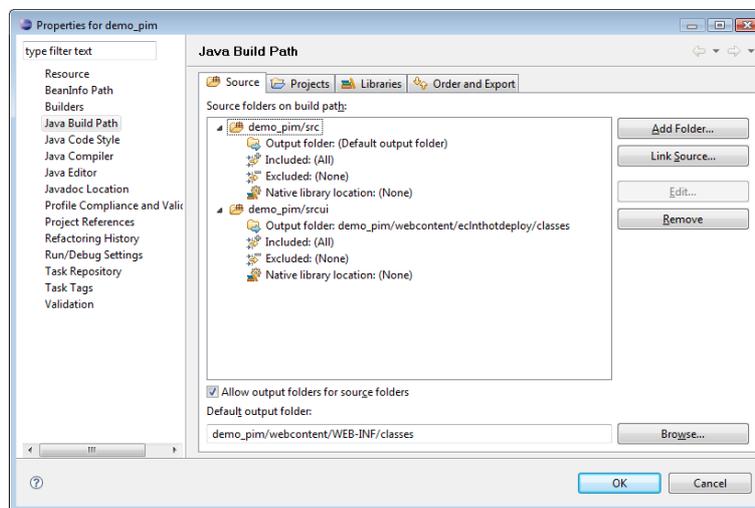
Consequently you have to structure the compilation of the project defined in your

development environment as well. The following explanation is done on base of Eclipse, but can be transferred to any other development environment as well.

There are two ways to go:

- (1.) Either you define two projects, one for the logical side, the other one for the UI side. The logical project passes its “.jar” libraries into the WEB-INF/lib folder that is maintained in the webcontent directory of the UI project. - The UI project compiles its classes into the hot deployment directories.
- (2.) Or you have both logical and UI code inside one project, but in different directories. Now you have to tell within your project definition that the logical code should be compiled to WEB-INF/classes and that the UI code should be compile to “eclnthotdeploy/classes”.

The configuration of the second way is a bit tough within Eclipse. Have a look onto the following project configuration:



The Java code below “/src” is compiled into the default output folder (which is WEB-INF/classes). The Java code below “/srcui” is comiled into the hot deployment folder (which is eclnthotdeploy/classes). In order to do this type of configuration you need to switch on the option “Allow output folders for source folders”.

---

## Design Time <=> Runtime

We recommend to restrict the usage of the “Hot UI Deployment Classloader” to be used in design time scenarios only. On the one hand it costs a bit of performance (not a lot, but a bit...) on the other hand it's a thing that you do not actually need for runtime purposes - so: leave it out!

For runtime purposes all classes need to be in WEB-INF/classes and WEB-INF/lib and the file “eclntjfsserver/config/hotdeploy.xml” should be removed.

# Security Issues

CaptainCasa Enterprise Client is based on standards:

- http(s) is used as communication protocol between client and server
- The server side is based on default servlet /JSF processing. The session management on server side is just the normal http session management.

This does not completely unburden you from checking certain security considerations which have to be applied to any http scenarios. This chapter lists these issues that typically are checked when going through security audits and tells you how to react on technical side correspondingly.

---

## The JSESSIONID appendix

### Basics on session-id management and risks involved

The transfer of the http-session-id is the base of assigning client side activities to the corresponding session data on server side. There are two standard ways to transfer the session-id:

- Transfer via cookie
- Transfer via URL encoding

CaptainCasa is using the “URL encoding” way. The main reason behind: cookies are shared between browser instances: when starting a browser out of an existing browser instances then cookies are shared between both browsers. - As consequence the same page (or applet in CaptainCasa) is shown twice to the user, both instances referring to the same session-id.

The URL encoding does not come with this significant disadvantage. With URL encoding the session-id is appended to any URL that is transferred to the client side - the typical “;jsessionid=xxxx” that you might already have seen when looking into what is transferred on http level between client and server.

While URL encoding is the (only) answer to correct session management across multiple browser instances, it comes with the disadvantage, that URLs that are requested by the client contain the appendix “;jsessionid=xxxx”. - Within CaptainCasa no URL is directly shown to the user at any place, so there is no risk of cutting/pasting URLs and send them e.g. via mail to others. But URLs are potentially log-able on network level. So, there is the danger, that someone gets to know your “;jsessionid=xxxx” appendix, and tries to hack into your server side session.

### Solution

CaptainCasa provides a default solution for this. By applying a certain servlet filter a parallel id is generated on server side for each session. This “session-check-id” is transferred to the client side. The “session-check-id” contains random information and is not predictable in any way as consequence.

Every JSF-request is an http-post request, which now contains the “session-check-id” in its post-part. (The post-part is the one that is encrypted when using https.) On server side the filter is applied in front of any application processing and checks if the “session-check-id” that is sent from the client is reflecting the one that is kept on server side.

The invoking of the filter needs to be done by adding the following filter:

```
<filter>
  <filter-name>org.eclnt.jsfserver.util.SecurityFilter</filter-name>
  <filter-class>org.eclnt.jsfserver.util.SecurityFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>org.eclnt.jsfserver.util.SecurityFilter</filter-name>
  <url-pattern>*.jsp</url-pattern>
</filter-mapping>
```

The definition is part of the web.xml\_template file in the WEB-INF directory of your project, so you might copy/paste this section from there.

The solution is both apply-able for “normal” CaptainCasa processing and for using the HT\*-controls (see “Developing plain HTML Pages”).

---

## Avoiding the Replaying of http - CLIENTSECID

There are security scenarios in which you have to explicitly avoid that the network activity of a certain http-request-processing is recorded and replayed somewhere afterwards.

To avoid this, CaptainCasa introduced a component CLIENTSECID that is bound to a server side object of type CLIENTSECIDBinding. The component requests from the CLIENTSECIDBinding a new id within each request-response processing on server side. This id is sent to the client side and re-sent to the server with the next request.

On server side a request is checked if the id that is received corresponds to the last id that was sent to the client. Then a new id is generated and sent with the response to the client.

The ids that are generated are random ids, that are not predictable to anyone trying to replay a certain recorded http-sequence against your server.

The CLIENTSECID component (arranged below the component BEANPROCESSING) should be placed on the most-outest page of your scenario, i.e. the one that is your starting page.

# Developing plain HTML Pages

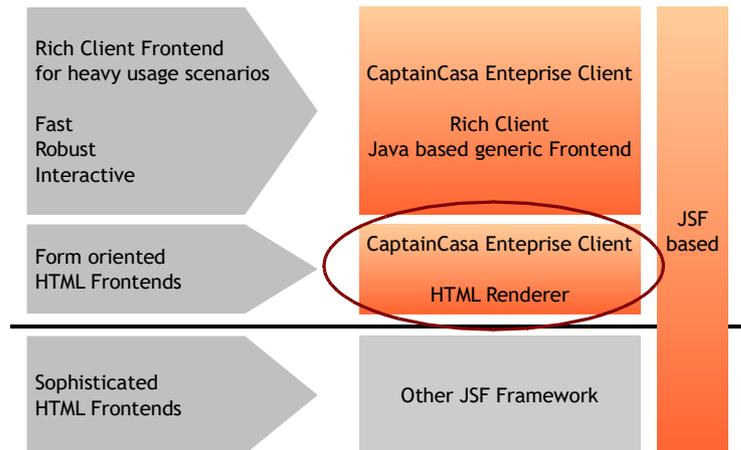
The main purpose of CaptainCasa Enterprise Client is the providing of rich, interactive and fast pages for your server side application - that's the reason why we chose Swing as technology on client side.

Since release 2.0 of CaptainCasa Enterprise Client there is also a framework for rendering HTML pages - this chapter provides detail information about this framework.

---

## Positioning

Why did we add a plain HTML framework? Have a look onto the following image:



Well, even an application, providing a wonderful rich client UI to the user, needs to provide for a solution to render a certain part of the application in HTML technology. Examples:

- Some screens should be accessed by anonymous users - and you do not want them to install the Java runtime environment prior to accessing the screens.
- Some screens may be shown on browser devices, e.g. sub-devices running on hand-helds, mobile phones, etc. For these devices very often a browser is available.

In order to meet these scenarios there are two ways to go:

- (1) CaptainCasa is built on top of JSF. One of the important reasons why we did this was to run on the same server side standard technology that is very commonly used for providing HTML pages. At the end you use the same JSF technology to build HTML pages than you use for building rich faces. Of course you need to select an appropriate JSF component library, you need to get into the details of this library - but the water used for cooking is the same as with CaptainCasa Enterprise Client.
- (2) CaptainCasa provides an own component library for building form oriented, simple HTML screens. With this library you can cover the “typical screens” of HTML applications. The building of pages is the same as the one you know from building rich pages - you use the same tools, just with a different set of components.

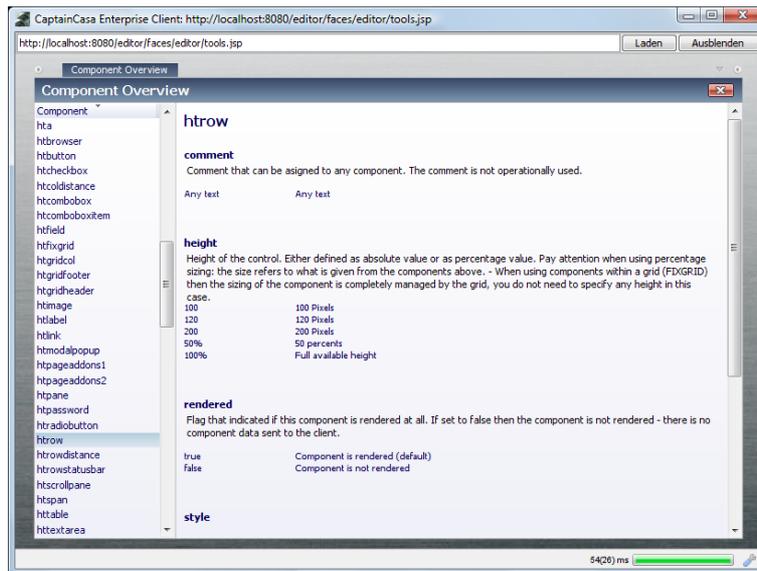
The 2<sup>nd</sup> way is the one described within this chapter.

---

## HT\* Components

All the components starting with the prefix “HT” are HTML components. I.e. they do not render into XML to be interpreted by a client side generic Swing GUI, but they render into HTML which is interpreted by a client side HTML Browser.

Have a look into the Component Overview which is part of the CaptainCasa tools to see which component are available, and have a look into the corresponding Demo Workplace.



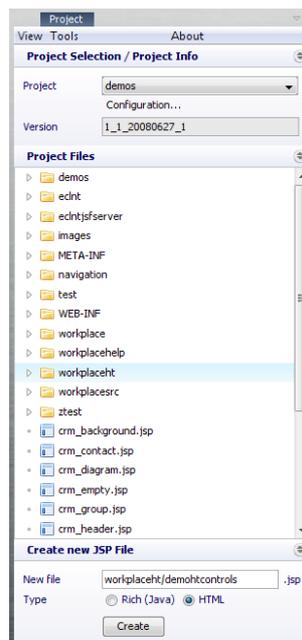
All components render into plain HTML - i.e. JavaScript is only used as minimum as possible. Currently there is only one component using a little bit JavaScript at all (HTLINK). Background: the components should run in “any type” of browser, as result we do not want to get into these areas where browsers differ from one another.

Our internal testing is based on Internet Explorer 7 and Mozilla Firefox 2.

## Writing HTML Pages

Writing HTML pages is exactly the same as writing rich pages. You use the Layout Editor for the design and you use your development environment (e.g. Eclipse) for the coding.

In the Layout Editor, when creating a page, you need to specify this page to be an HTML page, instead of using the default page type “rich”.



The definition is done by using the radio button on the very bottom.

Inside the Layout Editor you then see the ht\* components in the tree on the left. The



---

## Styling

### Default Style

The default style for all components is kept in the file `/eclntjsfserver/htstyle/htstyle.css`. This file serves as reference for styles that you may define on your own.

### Statically exchanging the default Style

You should not update the default style on your own - otherwise your updates will be overwritten with every update of your CaptainCasa environment.

But: you may define an own style and then register this file within your project definition:

```
<project ...
  ...
  htstylepath="/xyz/abc.css"
  ...
  ...>
  ...
  ...
</project>
```

Please pay attention: the fix default style is written into the JSP pages that you generate by using the layout editor. After updating the default style you have to re-save at least the “most outside” pages of your application. (The style that is used is automatically taken over for all pages that you include by using ROWINCLUDE or ROWPAGEBEANINCLUDE - so you do not have to re-save these.)

### Dynamically assigning the Style

In certain situations you may want to dynamically assign the style to be used. In this case you may append the parameter “ccstyle” to the URL, calling you page. Example:

```
http://localhost:8080/demos/faces/mypage.jsp?ccstyle=nicestyle
```

In this case the style is automatically routed to the following style sheet:

```
<webapp>/eclntjsfserver/styles/<ccstyle>/style.css
This is:
http://localhost:8080/demos/eclntjsfserver/styles/nicestyle/style.css
```

...you see: the style management for HTML is then very similar to the style management that is used for the rich client (pointing to the style.xml file in the same directory). You can select in the layout editor, which style to use for previewing you page.

### Individual styling

In addition you can use the STYLECLASS or STYLE attributes that are available on all components in order to update the style of certain components individually.

---

## Special HT\* Components

Most of the HT\* components are similar to their “rich” counterparts. They cover the very basic features but do not provide any special interactive usage patterns. For example there is a HTFIELD component used for text input - similar to the rich FIELD component. But: the HTFIELD does not provide flushing (triggering communication to the server

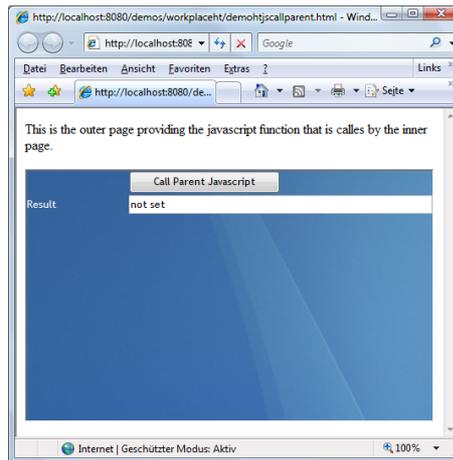
without explicitly pressing a button), it does not provide self-defined drag&drop management, it does not provide right mouse button popup menus, etc. etc.

We do not list all HT\* components in this chapter - please check the demo workplace and the online component documentation. We concentrate on special components only.

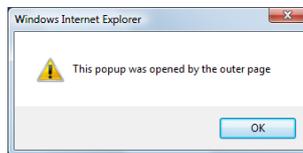
## Component HTJSCALL

The component HTJSCALL is used to execute a JavaScript method on browser client side. It is used to contact JavaScript functions which for example are not part of the page that is rendered by JSF.

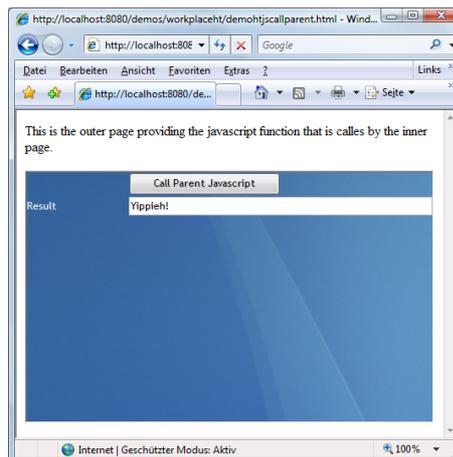
Example:



When pressing the “Call Parent JavaScript” button, then a popup shows up:



...and the page content is updated by the result of JavaScript processing:



There is a outside page which is defined in the following way:

```
<html>
<script>
function callDemo()
{
    alert("This popup was opened by the outer page");
    var result = new Object();
    result.callback = true;
    result.returnValue = "Yippieh!";
}
```

```

        return result;
    }
</script>
<body>
    <p>
        This is the outer page providing the javascript function that
        is called by the inner page.
    </p>
    <iframe src=" ../faces/workplaceht/demohtjsCall.jsp" width="100%" height="300"
    style="overflow:hidden">
</body>
</html>

```

The “JSF page” is embedded into the outer page by using an IFRAME tag. The outer page provides a JavaScript function “callDemo()”. The function does something and returns back a result with two variables: result.callback and result.returnValue.

Now let's take a look onto the inner page's jsp file:

```

...
<t:htrow id="g_2">
    <t:htcoldistance id="g_3" width="120" />
    <t:htbutton id="g_4" actionListener="#{d.DemoHtJsCall.onJsCall}"
        text="Call Parent Javascript" />
</t:htrow>
<t:htrow id="g_8">
    <t:htlabel id="g_9" text="Result" width="120" />
    <t:htfield id="g_10" text="#{d.DemoHtJsCall.result}" width="100%" />
</t:htrow>
<t:htrowdistance id="g_11" height="100%" />

<t:htjsCall id="g_12" actionListener="#{d.DemoHtJsCall.onJsResult}"
    jscode="#{d.DemoHtJsCall.jscode}"
    resulttext="#{d.DemoHtJsCall.jsresult}" />

```

The HTJSCALL component should be positioned at the very end of the JSP layout definition. It provides the following attributes:

- JSCODE - the JavaScript code to be executed. This is the Javascript statement that is actually executed when the page is rendered within the client. Typically this code is not defined statically but dynamically using an expression. In the example the value behind the JSCODE expression is only set when the user presses the button.
- ACTIONLISTENER - the JavaScript method that is invoked on client side may return a special return value, providing the internal properties “callback” and “returnValue”. When “callback” returns “true” (boolean) then immediately after executing the JavaScript method a callback to the server is done. I.e. the form of the JSF page is submitted. Background: sometimes the JavaScript that is invoked collects some data and immediately sends the data back to the server side. When being submitted then the ACTIONLISTENER is called on server side.
- RESULTTEXT - the JavaScript method that is invoked on client side may return a result providing the property “returnValue”. This value is passed back into the RESULTTEXT expression. The format of the string, and if it contains some structured information, is completely up to you.

Now let's have a look onto the managed bean serving the page:

```

public class DemoHtJsCall implements Serializable
{
    protected String m_result = "not set";
    public String getResult() { return m_result; }
    public void setResult(String value) { m_result = value; }

    protected String m_jscode;
    public String getJscode() { return m_jscode; }

    protected String m_jsresult;
    public String getJsresult() { return m_jsresult; }
}

```

```

public void setJsresult(String value) { m_jsresult = value; }

public DemoHTJSCall()
{
}

public void onJsResult(ActionEvent event)
{
    m_result = m_jsresult;
}

public void onJsCall(ActionEvent event)
{
    m_jscode = "parent.callDemo()";
    PhaseManager.runAfterRenderResponsePhase(new Runnable()
    {
        public void run()
        {
            m_jscode = null;
        }
    });
}
}

```

When the method “onJsCall” is called (i.e. the user presses the button) then “parent.callDemo()” is passed as JavaScript method to be invoked. The value of “m\_jscode” is set back to null when the request is processed, so that by default no further JavaScript processing is initiated with the next request. - This is quite important to avoid loops in some situations, when using the callback-feature!

The method “onJsResult” is called when the JavaScript execution on client side is finished. Why? Because the JavaScript execution of the example returns back “result.callback == true”. - The text that is returned from the JavaScript processing is passed into the property “jsresult”.

### **HTJSCALL - Only use once!**

The component HTJSCALL must only occur one time within you page. Multiple occurrences will result in non-predictable behavior!

### **HTJSCALL - Darkening the Page from your Code**

Sometimes you may want to “darken” the page from your Javascript code. “Darkening” normally is implicitly done when the JSF page sends back a request to the server: in this case a transparent coloring is put on top of the screen and the screen is blocked for any input.

From outside you can “darken” by calling the method “ccDarken(true/false)”. Have a look into the HTML/JavaScript code of a page (by looking into the page source within your browser environment) to see details about this method.

# Selected Topics

## Configuration

### Definition of Error Screen for Server-side Errors

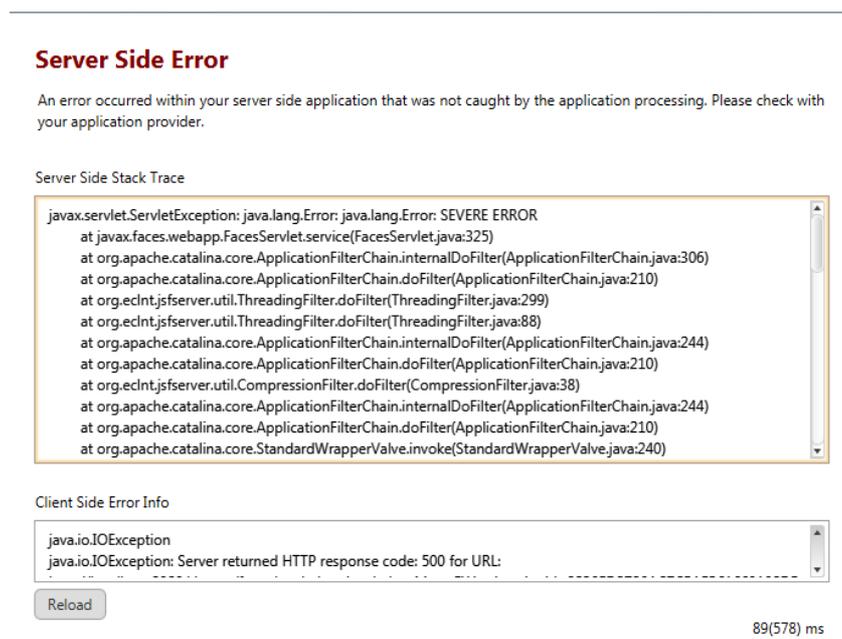
By default the server side application is responsible for catching errors.

Example: inside the implementation of an `actionListener`-method you may catch errors in the following way:

```
public void onSaveAction(ActionEvent event)
{
    try
    {
        ...
    }
    catch (Error e)
    {
        ...
    }
    catch (RuntimeException re)
    {
        ...
    }
}
```

You may use the interface `IErrorAware` within you bean implementation - which automatically delegates uncaught exceptions/errors to some centralized processing as well. You may for example implement “`IErrorAware`” on level of the dispatcher class (the `#{d}` instance...) and then centrally process error situations.

...but: in reality it may happen that you still do not catch all error situations. In this case the CaptainCasa framework shows an error page:



The error page is defined in the following `.jsp` file:  
`ecIntjsfserver/includes/showservererror.jsp`

You may override the default error screen for server side errors in the following way:

- Make a copy of “eclntjfsfserver/includes/showservererror.jsp” and store the copy inside your .jsp files (e.g. “mypages/showservererror.jsp”).
- Now you can modify the .jsp file. The managed bean behind the .jsp file is the class “ShowServerError” in the package “org.eclnt.jsfserver.defaultscreens”. You may use all get-properties that are available in this class and reference them from your screen. And you can of course design the page any way you like.
  - Please note: your page must be bound to the data available in the class “ShowServerError” - this is a runtime the object which the CaptainCasa server runtime fills with corresponding error information (e.g. the exception that came with the error)
- You now have to tell the client by setting the client parameter “errorscreen” that in case of recognizing some server side errors it switches to you error page. E.g. the parameter value could be “errorscreen=/faces/mapges/showservererror.jsp”.

## Definition of Error Screen for Client-side Errors

The client may run into error situations in which it shows up an error screen on its own - which is independent from server side processing.

Example: if the client is started and the server is not reachable via http, then the client must show up some error screen. In this case it is not possible, of course, to reference to some server side definition of a error screen because the server is not available - the screen definition of the corresponding error screen must be part of the client processing.

There are certain steps inside the client processing to update the client error screen:

- In case of an error the client requires an implementation of interface ILocalErrorScreenProvider. The interface is:

```
public interface ILocalErrorScreenProvider
{
    public void passurl(String url);
    public String getXMLofErrorScreen(String comment,Throwable exc);
}
```

- The default implementation is “DefaultLocalErrorScreenProvider” - you may use this one and extend it to you needs. If implementing your own interface then you have to pass the class name of your implementation with client parameter “errorscreenproviderclassName”. (Please pay attention: there is a capital “N” inside the parameter name!)

In the default implementation the screen that shows up is built using some client side template. The template contains the XML together with some placeholders that are replaced with error information at runtime. The default template is:

```
<form id="_id0" method="post" action="lastvalidsessionid"
    enctype="application/x-www-form-urlencoded">
    <row id="eror_r0">
        <scrollpane id="error_p1" width="100%" height="100%"
        bgpaint="rectangle(0,0,100%,100%,#FFFFFF)" border="#000000" padding="20"
        rowdistance="20" iswindowmover="true">
            <row id="error_r1">
                <label id="error_l1" text="@@title@"
                font="size:20;weight:bold"/>
            </row>
            <!--
            <row id="xxerror_r1">
                <label id="ddderror_l1" text="@@titleExtension@"
                font="size:16;weight:bold"/>
            </row>
```

```

-->
<row id="error_r2">
  <textpane id="error_t1" width="100%" text="@@text@" />
</row>
<row id="aerror_n2">
  <pane id="aah_1" padding="10"
bgpaint="rectangle(0,0,100%,100%,#00000010)" width="100%">
    <row id="aah_2">
      <!-- @@reconnectbegin@@
      <reloadbutton id="aat_a1" width="200"
text="@@reconnect@" trytokeepsession="true"/>
      <coldistance id="hh3" width="10"/>
      @@reconnectend@@ -->
      <reloadbutton id="aerror_n3" width="150" text="@@reload@"
hotkey="@@hotkey@" requestfocus="creation"/>
      <coldistance id="hh1" width="100"/>
      <systemicon id="hh2222" text="@@configure@"
systemfunction="configure"/>
      <coldistance id="hh1111" width="5"/>
      <systemicon id="hh2" text="@@close@" systemfunction="close"/>
    </row>
  </pane>
</row>
<rowdistance id="bbb" height="50"/>
<row id="l1lerror_r1">
  <label id="l1lerror_l1" text="@@titleDetail@"
font="size:16;weight:bold"/>
</row>
<row id="ssserror_r3">
  <textarea id="ssserror_ta1" width="100%" height="100%;100"
background="#FFFFFF" text="@@stacktrace@" />
</row>
</scrollpane>
</row>
</form>

```

You may pass an own template by specifying the template via client parameter “clienterrorscreen”. The default value is “org/eclnt/client/comm/http/errorscreen.xml”.

The template is loaded via the client side class loader and must be part of the client side jar files as consequence.

In any case: for updating the error screen you have to update the client side files. This means you e.g. have to add an own jar-file to be loaded via html-applet/jnlp-webstart. And this means, you have to re-sign all client side jar files with some own signature of yours.

## Appendix - Stream Store Persistence

CaptainCasa is a frontend-related technology and itself does not have too much to do with persistence. Of course applications, built with CaptainCasa, have to deal with persistence! - but this is typically something in some deep layers, well behind the user interface processing.

But: there are some situations in which CaptainCasa wants to persist data as well. Examples are:

- When a user changes the sequence of columns, then this may be stored, so that the column layout is updated for this user.
- In the workplace management, there are some configuration files, that are to be defined at runtime - e.g. the definition of a function tree for a specific user.
- In the workplace management the user may update the perspective's layout by drag & drop. Again this needs to be persisted...

In order to persist all this data at runtime, there is a persistence mechanism, called "Stream Store" (the name of the corresponding Java class is "StreamStore").

---

### Stream Store API

Imagine the stream store to be a file system, that allows to store UTF8-based files (typically XML files). For each file/content you want to persist you define:

- ...the directory, in "slash"-notation, e.g. "/aaa/bbb/"
- ...the name of the file, e.g. "ccc.xml"
- ...the content of the file as UTF8-String

This is represented by the stream store API that is available through the interface "IStreamStore":

```
public interface IStreamStore
{
    public List<String> getContainedStreams(String dirpath, boolean withError);
    public List<String> getContainedFolders(String dirpath, boolean withError);
    public String readUTF8(String path, boolean withError);
    public void writeUTF8(String path, String xml, boolean withError);
    public void removeStream(String path, boolean withError);
    public boolean checkIfStreamExists(String path, boolean withError);
}
```

Based on this API, the individual functions define where and how to store their information within the stream store. For example, the workplace perspective management stores its information in the following directory:

```
/ccworkplace
 /perspectives
  /<username>
   /<perspectiveName>.xml
```

You may use the stream store API for persisting own data as well. Please keep in mind that all directory names starting with "cc" are reserved for use by CaptainCasa only.

And (of course...): only use the stream store for UI related data. Do NOT use the stream store for any other purpose. The stream store's persistence is not built for mas—data-management and does not ensure any type of transactional consistence.

---

## Where is the Data stored?

Now, let's get to the “most important” question: where is the data stored? The answer is: this depends on the stream store implementation...!

In principal any implementation of the `IStreamStore` interface can be activated. You need to register the corresponding class in the configuration file `/eclntjsfserver/config/system.xml` - that's it:

```
<system>
  <streamstore
    name="org.ec1nt.jsfserver.streamstore.StreamStoreFile"
  />
</system>
```

There are two default implementations that come with the CaptainCasa runtime:

- A file system based persistence
- A JDBC based persistence

### File based Persistence

This is the persistence that is activated by default - i.e. if not doing any specific definitions within the `system.xml` configuration file.

The directory structure of the stream store is directly reflected into the directory structure of a certain area of the file system. If not doing any specific definition, then this area is selected in the temporary work area of the web application.

(In case of Tomcat this is the directory in `tomcat/work/Catalina/localhost/<webapp>`.)

But you can override this directory and explicitly set an own one, by doing the following definition in `system.xml`:

```
<system>
  <streamstore
    name="org.ec1nt.jsfserver.streamstore.StreamStoreFile"
    rootdir="c:/streamstoredir"
  />
</system>
```

The file based stream store is positioned for “small scenarios”, without clustered application servers only! - Imagine: when having multiple application servers then the only way to store data at one common place is to define one central file server (“[x:/streamstore](#)”) that is accessed from each application server node remotely.

### JDBC based Persistence

There is a JDBC based implementation of `IStreamStore` as well. It is invoked in the following way:

```
<system>
  <streamstore
    name="org.ec1nt.jsfserver.streamstore.StreamStoreJDBC"
  />
  <jdbcconnectionprovider
    name="demostreamstore.DemoConnectionProvider"
  />
</system>
```

The first “streamstore” definition in `system.xml` tells the stream store management to use the JDBC based implementation. Because the JDBC based implementation requires a JDBC connection to a database, the second “jdbcconnectionprovider” definition is required as well.

The JDBC connection provider is a class of your application, implementing the interface `IJDBCConnectionProvider`. A (very simple) example is:

```
package demostreamstore;

import java.sql.Connection;
import java.sql.DriverManager;

import org.eclnt.jsfserver.streamstore.IJDBCConnectionProvider;
import org.eclnt.util.log.CLog;

public class DemoConnectionProvider implements IJDBCConnectionProvider
{
    static
    {
        try
        {
            Class.forName("org.hsqldb.jdbcDriver");
        }
        catch (Throwable t)
        {
            CLog.L.log(CLog.LL_ERR, "", t);
        }
    }

    public Connection createConnection()
    {
        try
        {
            Connection result =
DriverManager.getConnection("jdbc:hsqldb:hsqldb://localhost/PIM", "sa", "");
            return result;
        }
        catch (Throwable t)
        {
            throw new Error(t);
        }
    }
}
```

---

## Design Time vs. Run Time Data

There are often situations in which configuration on the one hand comes from the program (i.e. it is part of your design time) and on the other hand comes from definitions that are made during runtime.

The stream store is able to handle this - by always taking the directory `/webresource/config` of your application into consideration as well. If the stream store is asked for the content of `"/ccworkplace/perspectives/default.xml"`, then...

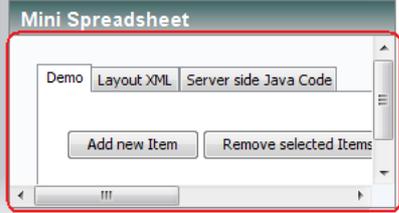
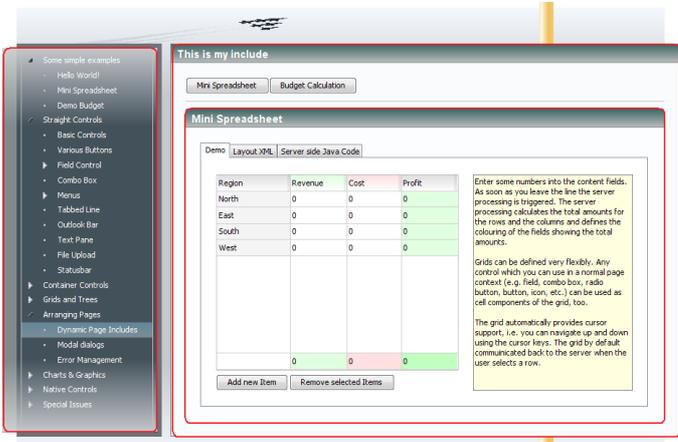
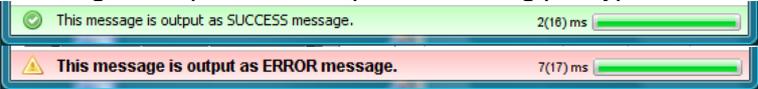
- ...it first checks the "normal" place of persistence, e.g. the database when using the JDBC based stream store implementation
- ...and then - if the first access was not successful checks the web application for the file `"/eclntjsfserver/config/ccworkplace/pespectives/default.xml"`.

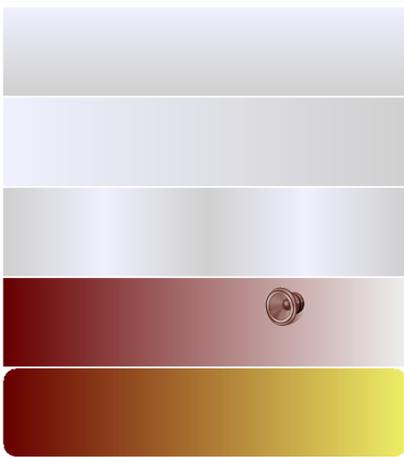
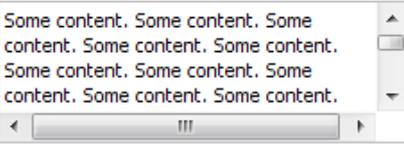
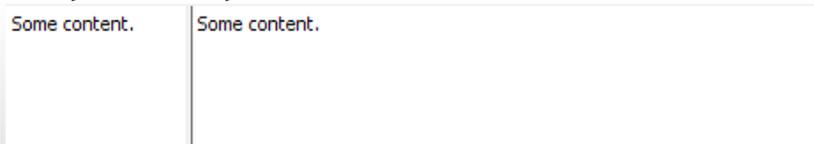
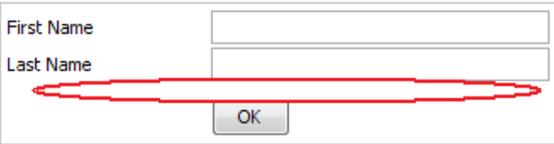
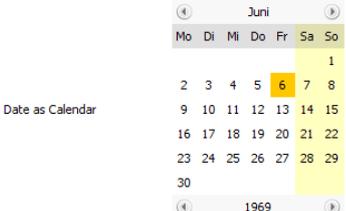
This means that you always can integrate design time data into your stream store.

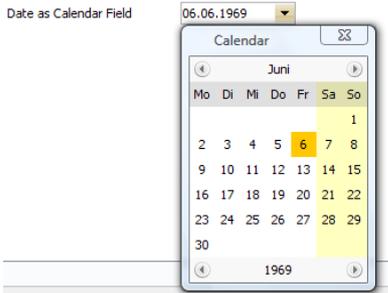
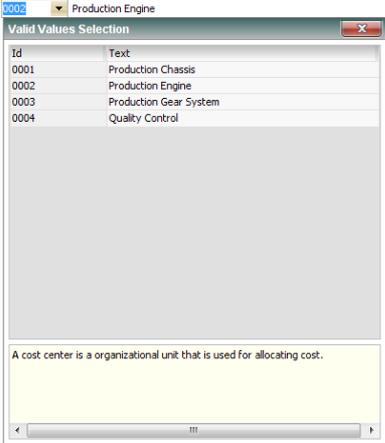
Of course the design time data is read-only. When writing back data into the stream store that was picked from the design time part, then automatically the updated file will be saved into the "normal" stream store persistence (e.g. the JDBC database), and will override the design time data as consequence.

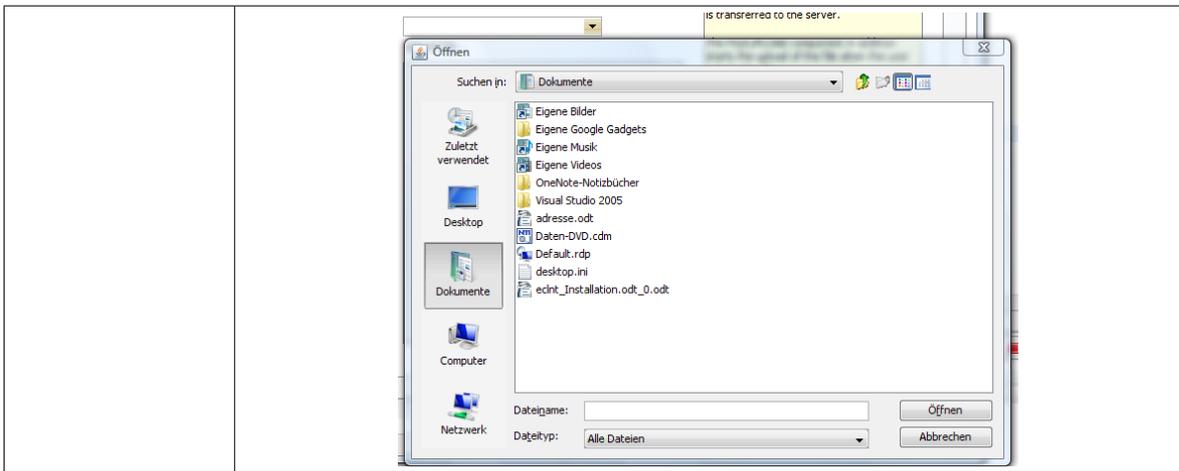
# Appendix - Component Overview

This chapter lists all the components available and - very briefly! - describes what they are used for. For detailed explanations please have a look onto the demo workplace and the detailed chapters of this documentation.

Typical Page Components	
ROWBODYPANE	<p>Container that holds the typical body part of the page. Provides scrollbars if the content of the page exceeds the ROWBODYPANE container's size.</p> 
ROWINCLUDE	<p>Area which includes another page. Central mechanism for assembling pages out of pages.</p> 
ROWTITLEBAR	<p>Title bar. Can be filled with additional components (e.g. icons) which are places on the very right of the bar.</p> 
ROWSTATUSBAR	<p>Status bar. Typically occurs exactly one time for the page. Knows certain types of messages and provides adequate rendering per type.</p> 
Container Components	
ANIMATEDPANE	<p>Pane - normally invisible - that provides a certain animation (flashing, i.e. the pane's area is overlaid with a certain color, that is dimmed out) on server request. Used to define areas which "flash!" e.g. when data changes.</p>
PANE	<p>Container that allows to arrange other components within rows. The PANE is by default invisible itself, but also may be colored in many ways.</p>

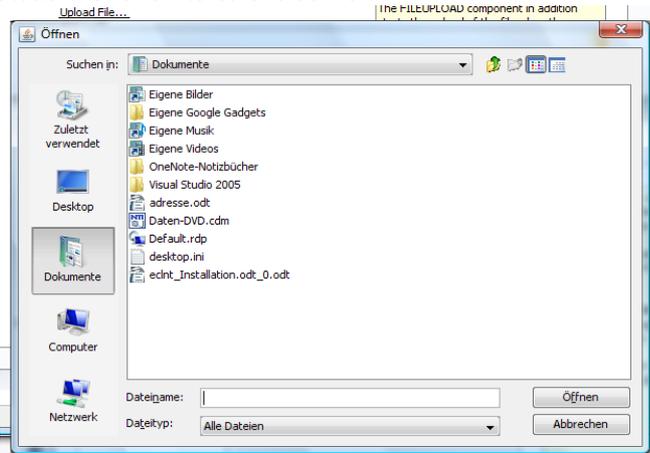
	
SCROLLPANE	<p>Container that has a certain size. If the content exceeds this size then scrollbars are shown.</p> 
SPLITPANE	<p>Container that divides up into sub-containers. Division either is done horizontally or vertically.</p> 
SPLITPANESPLIT	Sub-container part of a SPLITPANE component.
Row Components	
ROW	Row inside a container. A row holds other components.
ROWDISTANCE	<p>Distance between two rows.</p> 
Components	
BUTTON	<p>Invokes a method on server side.</p> 
CHECKBOX	<p>Allows to edit boolean values.</p> <input type="checkbox"/> Text aside a checkbox
CALENDAR	<p>Calendar that is directly displayed in the page. The calendar component receives and edits a Date-value which is associated with a time zone.</p> 

<p>CALENDARFIELD</p>	<p>Input field which checks user input to be a valid date - according to the Locale definition on client side. Provides calendar popup when user presses the icon or pressed the F4-key.</p> 
<p>COLDISTANCE</p>	<p>Puts a distance between controls of a row.</p> 
<p>COMBOBOX</p>	<p>Selection of one out of multiple values. Each values is represented as id or as id with a text.</p> <p>Credit Card <input type="text" value="0002"/> <input type="text" value="Production Engine"/></p> <p>Credit Card <input type="text" value="0002"/> <input type="text" value="Production Engine"/></p> <p>Credit Card <input type="text" value="0002"/> <input type="text" value="Production Engine"/></p>
<p>COMBOBOXITEM</p>	<p>Item of a COMBOBOX. Used when you want to hard-wire the items of a COMBOBOX within the layout definition.</p>
<p>COMBOFIELD</p>	<p>Field with valid value support. When clicking the icon or pressing F4 you can open up any popups for value selection. Default popups for simple lists are provided.</p> 
<p>FIELD</p>	<p>Input field with various configuration possibilities.</p> <p>Example "mandatory field" <input type="text"/></p>
<p>FILECHOOSER</p>	<p>Allows to select a client file name via the default file selection popup. Screenshot: see FILEUPLOAD.</p>
<p>FILEUPLOAD</p>	<p>Allows to upload a client file to the server.</p>



FILEUPLOADLINK

Allows to upload a client file to the server.



FORMATTEDFIELD

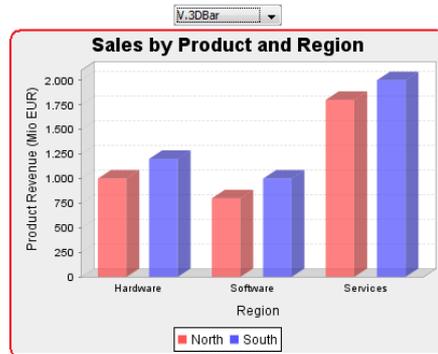
Field that checks the input value against data types (int, float, decimal, long, date, time) according to the client's localization

Integer Value	<input type="text" value="100"/>
Float Value	<input type="text" value="99,99"/>
Date Value	<input type="text" value="11.01.2008"/>
Time Value	<input type="text" value="11.01.2008 08:19:58"/>
Big Decimal	<input type="text" value="99,99"/>

HEXIMAGE

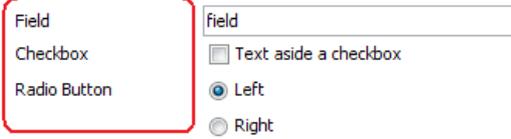
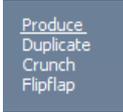
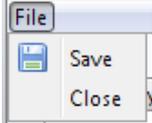
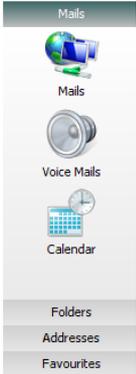
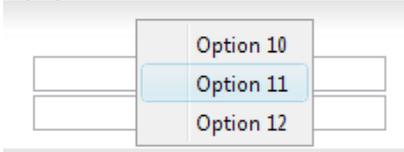
Image output. The image is not loaded from a URL but is provided by a hexadecimal string representation. Example of usage: images are stored in a database.

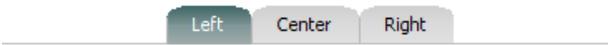
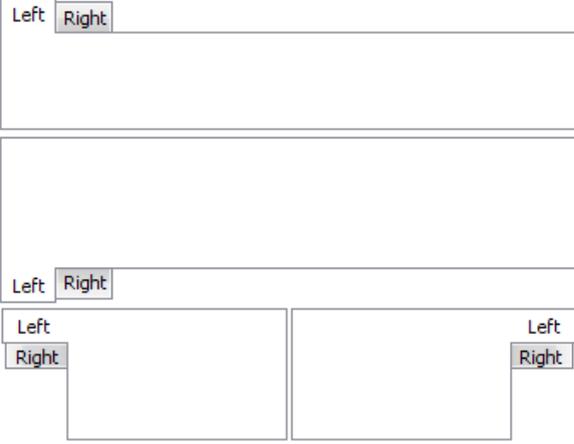
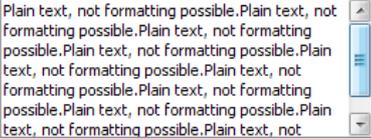
Product	Revenue North	Revenue South
Hardware	1.000	1.200
Software	800	1.000
Services	1.800	2.000

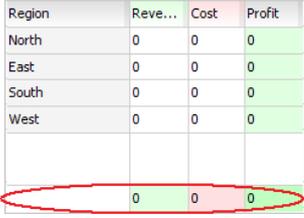
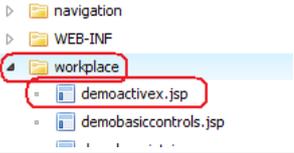


IMAGE

Image output the image is loaded from a URL which is part of the web

	application.
IMAGECAPTURE	Special control for managing certain image information aspects for an object.
LABEL	Text Label for any purpose. 
LINK	Invoke of a server side function (like BUTTON), but rendered as link. Rollover underline effects. 
MENU	Folder component of a MENUBAR or a POPUPMENU.
MENUBAR	Menu bar that can be placed everywhere on the page. A page may have multiple menubars.  A MENUBAR holds MENU components. MENU components may hold MENU components themselves or hold MENUITEM components.
MENUITEM	End node of menu.
OUTLOOKBAR	Outlook bar which allows to switch between various aspects. 
OUTLOOKBARCONTENT	Content part of the OUTLOOKBAR component.
OUTLOOKBARITEM	Item part of the OUTLOOKBAR component.
PASSWORD	Password input field.
POPUPMENU	Right mouse button popup. Multiple POPUPMENU components can be defined per page. Other components reference to the POPUPMENU components in order to define their popup menu. 
RADIOBUTTON	Radio button that represents a certain value. Several radio buttons that are maintaining the same property are grouped by using the GROUP attribute.

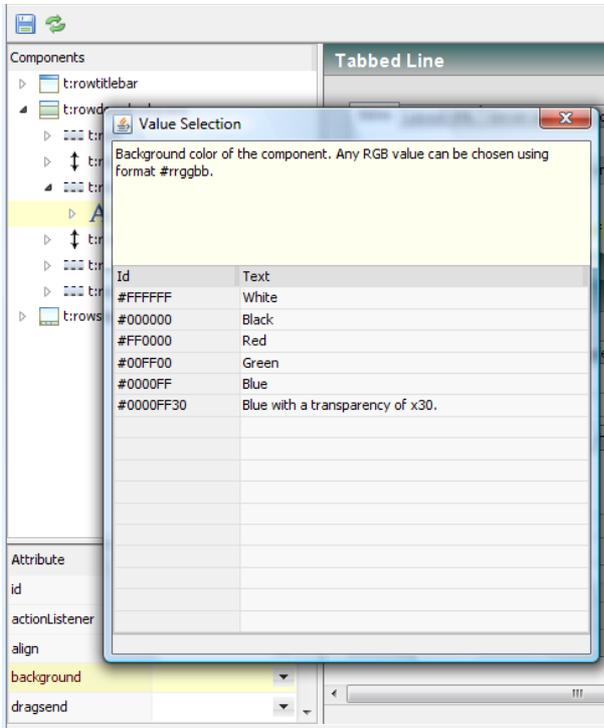
	<input checked="" type="radio"/> Left <input type="radio"/> Right																												
TABBEDLINE	<p>Sequence of “tabs” that invoke a function on server side when clicked. The currently selected tab is highlighted. Multiple ways of defining the rendering.</p> 																												
TABBEDLINETAB	Item of a TABBEDLINE component.																												
TABBEDPANE	<p>Tabbed pane.</p> 																												
TABBEDPANETAB	Item of a TABBEDPANE component.																												
TEXTAREA	<p>Multi line edit component. Only plain text editing is enabled.</p> 																												
TEXTPANE	<p>Text output. The text type is either plain text or HTML text or RTF text. Only simple formatting rules of HTML and RTF are provided.</p> <p>Some <b>HTML Text</b>. Not everything is possible, but the basic things...</p>																												
<b>Grid Components</b>																													
FIXGRID Grid Usage	<p>Grid that contains any type of column components.</p> <table border="1" data-bbox="536 1469 1094 1765"> <thead> <tr> <th>Region</th> <th>Revenue</th> <th>Cost</th> <th>Profit</th> </tr> </thead> <tbody> <tr> <td>North</td> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>East</td> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>South</td> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>West</td> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td></td> <td></td> <td></td> <td></td> </tr> <tr> <td></td> <td>0</td> <td>0</td> <td>0</td> </tr> </tbody> </table>	Region	Revenue	Cost	Profit	North	0	0	0	East	0	0	0	South	0	0	0	West	0	0	0						0	0	0
Region	Revenue	Cost	Profit																										
North	0	0	0																										
East	0	0	0																										
South	0	0	0																										
West	0	0	0																										
	0	0	0																										
FIXGRID Tree Usage	Tree with / without additional columns of any type.																												
GRIDCOL	Columns of a FIXGRID. Each GRIDCOL component may hold exactly one component that is the one rendered for each cell of the grid column.																												
GRIDCOLFOOTER	A FIXGRID may contain several footer lines. These lines are optically part of the grid, but from data binding point of view they are decoupled.																												

	
GRIDCOLHEADER	Same as GRIDCOLFOOTER, now for the header lines.
TREENODE	<p>Tree node within a GRIDCOL within a FIXGRID component. The TREENODE allows dynamic assignment of text and images.</p> 
<b>Special Components</b>	
ACTIVEX	<p>Allows to integrate ActiveX components into a layout. You can dynamically set properties and call methods of the component. You can take data from the ActiveX component back to the server processing.</p> 
ANIMATEICON	<p>Icon that allows to toggle another components property from one value to another when being clicked. This processing happens on client side only. Example of usage is the icon on the right top within a FOLDABLEPANE component:</p> 
BROWSER	Native browser that you can include into a layout.
<b>Non visible Components</b>	
BEANPROCESSING	Folder node for keeping BEANMETHODINVOKER and BEANPROPERTYSETTER inside. No functional meaning.
BEANMETHODINVOKER	Allows to call a method of a managed bean during certain phases of the JSF request processing.
BEANPROPERTYSETTER	Allows to set a property of a managed bean during certain phases of the JSF request processing.

# Appendix - Component Attribute Reference

Please note: explicit online documentation is available for each attribute within the layout editor environment:

When editing the value of an attribute then you can open a value help dialog by pressing F4 within the field, or by clicking onto the valid values icon of the field.



The value help dialog contains both textual information and provides a list of example values.

---

## Attribute Value Formats

Some attributes may provide the possibility to define complex parameters. Example: the BGPAIN attribute provides a list of graphical background painting commands for a component. Its value is a semicolon separated concatenation of method specifications. Example:

```
<pane ...  
  bgpaint="rectangle(0,0,100,100,#FF0000);rectangle(100,100,50,50,#00FF00)"  
  />
```

This chapter tells about the rules to form these complex parameters.

### Semicolon separated List

The following examples are valid comma separated list strings:

```
A;B;C ==> "A","B","C"  
A; B ; C ==> "A"," B "," C"  
A;BB\x3BBB;C ==> "A","BB;BB","C"  
A;BB, BB;C ==> "A","BB, BB",C
```

The rules are:

- Any character is a valid one - there is not automated trimming
- A “;” within a string is represented as “\x3B” (pay attention: “B” is in uppercase!). There are no other special representation of characters

The class “org.eclnt.util.valuemgmt.ValueManager” provides some functions to simplify the creation of semicolon separated strings:

```
public static String encodeCSV(String[] values) { ... }
public static String[] decodeCSV(String csvValue) { ... }
```

Please check for more information within the corresponding Java API documentation.

## Method Value

Sometimes an attribute value represents a method or a command to be executed. The command consists out of a name and a list of parameters.

Valid methods are:

```
rectangle(0,0,100,100,#FF0000)
image(0,0,/images/hallo.gif,lefttop)

command(a,b;b,c) ==> command! “a”, “b;b”, “c”
command( a ;b;c ) ==> command! “ a “, “b”, “c “
command(a,b\x2Cb,c) ==> command! “a”, “b,b”, “c”
```

The rules are:

- The command name is starting with the first character and ending with the first “(“ occurrence.
- The parameters are separated by commas. If a parameter itself contains a “,” then this is represented as “\x2C”. Pay attention: “C” must be in uppercase.
- Any character is valid, there is not automated trimming of strings

The class “org.eclnt.util.valuemgmt.ValueManager” provides some functions to simplify the building of method values:

```
public static String decodeMethodName(String methodValue) { ... }
public static String[] decodeMethodParams(String methodValue) { ... }
public static String encode(String methodName, String[] methodParams) { ... }
```

## Value List

In some cases an attribute value is a collection of diverse “name”-“value” combinations. Valid examples are:

```
top:1;bottom:1;left:2;right:2;color:#FF0000
color:#FF0000
```

The rules are:

- name-value-pairs are separated by “;”. See the rules on semicolon separated strings for more information.
- Name and value are separated by “:”. Names and values must not contain a “;” character.
- All characters are valid. There is no automated trimming of spaces at the begin or end of a value.

The class “ValueManager” provides some useful methods that simplify dealing with complex value strings:

```
public static Map<String,String> decodeComplexValue(String value) { ... }
public static String encodeComplexValue(Map<String,String> valueMap) { ... }
```

## Color Values

Color values can be specified in two ways:

- #RRGGBB ==> red/ green/ blue specification
- #RRGGBBTT ==> same as above, now together with transparency definition. Transparency “00” means “100% transparent, background shines through”, transparency “FF” means “no shining through of background”.

---

## Some Information about some Attributes

### action

- Reference to a faces action. Faces actions are the JSF framework for triggering navigation between commands.
- An action either is a literal which is referred within the navigation section of faces-config.xml.
- Or it is a reference to a managed bean method which itself returns a literal that triggers navigation. The corresponding bean method must be defined in the following way:

```
public String onXYZ()
{
    ....
    return navigationResult;
}
```

### actionListener

- Reference to a method processing of a managed bean.
- The bean's method has an ActionEven-parameter:

```
public void onSave(ActionEvent event)
{
    ....
}
```

- Pay attention: there is a java.swt-ActionEvent and a javax.faces-ActionEvent. You need to select the Faces-Event within your code!
- Example: “#{address.onSave}”

### background

- Color code of the background color.

### bgpaint

- Defines the background painting of a pane.
- Semicolon separated list of paint commands. The commands are executed on client side in the sequence of definition.

- Available Commands
  - `rectangle(x,y,width,height,color)` - draws filled rectangle
  - `rectangle(x,y,width,height,color1,color2,direction)` - draws filled rectangle with color from color1 to color2. Direction is either “vertical” or “horizontal” and defines the direction of color change.
  - `image(x,y,imageUrl,anchor)` - draws an image. Anchor defines how x,y are interpreted. Possible values are: “lefttop”, “leftmiddle”, “leftbottom”, “centertop”, “centermiddle”, “centerbottom”, “righttop”, “rightmiddle”, “rightbottom”
  - `image(x,y,width,height,imageUrl,anchor)` - draws an image now with passing the width and the height.
- X,y,width and height either are defined as pixel values or as percentage values.

### border

- Defines the border of a control.
- Style definition consisting out of the style elements:
  - left, top, right, bottom - pixel size of the border
  - color - color of the border
- Example: `border="left:1;top:1;right:1;bottom:1;color:#C0C0C0"`, default = ""

### contenttype

- Text content type of text which is rendered inside the control. As consequence the control will render the text accordingly.
- Use “text/html” in order to use the control for rendering HTML text. - The HTML rendering is limited to “simple HTML” formatting - sophisticated style management is not supported.
- Possible values are “text/plain”, “text/html” and “text/rtf”. The default is “text/plain”.

### enabled

- Activates the control. If not enabled (“false”) then a control does not allow any change of data to the user.
- Example: “true”, default is “true”

### image

- URL of an image file that is available on the server. Images of type .gif, .jpg and .png are supported.
- The addressing of the URL either is absolute or relative to the page definition. The decision is done based on the first character - if it's a slash (“/”) then the URL is interpreted as absolute URL, if not then the URL is interpreted as relative URL.
  - “images/xyz.png” or “../images/xyz.png” are loaded relative to the pages they are defined inside
  - “/images/xyz.png” is loaded from the “images” directory of the web application.

This means the full URL is “http://host:port/<webapp>/images/xyz.png”. Note, that “absolute” means “absolute within the web application”.

- Pay attention: the Enterprise Client frontend by default only can talk back to the server it was loaded from. Do not refer to images which are available on other servers.
- Example: “images/disk.png”

## padding

- The padding of the container content relative to the border of the container.
- Example: “20”, default is “0” for default pane controls, and “20” for the ROWBODYPANE control.

## page

- Name of a referenced JSP-page. The page must reside in the same web application, i.e. It is not allowed to refer to a page outside the web application.
- The URL can be either relative (no leading “/”) or absolute inside the web application (with leading “/”).
- Example: “../xyz/xyz.jsp” is a relative URL, “/xyz/xyz.jsp” is an absolute URL, pointing to the file “\$webapp\$/xyz/xyz.jsp”.

## stylevariant

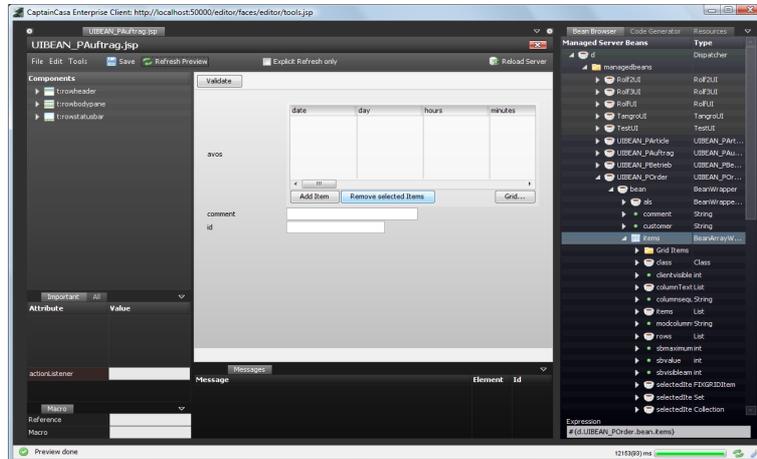
- For each control you can define a standard style that is used.
- A style is an XML file located in the ecIntjsfserver/styles folder of the web application. Each style may contain several variants for a control - the “stylevariant” selects the corresponding style variant.
- The setup of style variants is up to the user of the Enterprise Client.
- Default style is the style variant “default”.

## tooltip

- String value that is output as tool tip on a control.
- A tool tip is shown to the user when the user moves the mouse onto a control and waits for a short while without moving the mouse.

# Appendix - Dynamic Introspection within the Bean Browser

On the right of the Layout Editor there is a tool “Bean Browser”:



The Bean Browser shows the managed beans of your runtime and allows to drag & drop corresponding expressions into the attributes of components.

By default the Bean Browser uses straight Java introspection in order to show the properties and action listener methods of a selected bean. But: you may use dynamic properties (i.e. Map or Array/List implementations) - and as a result you may also want to show these properties within the bean browser.

## Method “introspectDynamically”

When resolving the bean hierarchy then the Bean Browser always checks if the currently selected element's class provides the following method:

```
public static DynamicIntrospectionInfo introspectDynamically(List references,
List<String> pathList)
{
    return ...;
}
```

If the class supports this static method then the information returned in the result of the method is used for navigating into the next “bean-level”.

The class DynamicIntrospection is contained in interface IdynamicIntrospectionSupported:

```
package org.ecInt.util.dynaccess;

import java.util.ArrayList;
import java.util.List;

/**
 * Classes implementing this interface need to provide the static method
 * "public static DynamicIntrospectionInfo introspectDynamically(List references,
 * List<String> pathList)".
 * which is used at design time by the bean browser.
 */
public interface IDynamicIntrospectionSupported
{
    public static class DynamicIntrospectionInfo
    {
        List<DynamicPropertyInfo> m_properties = new
        ArrayList<DynamicPropertyInfo>();
        List<DynamicMethodInfo> m_methods = new ArrayList<DynamicMethodInfo>();
    }
}
```

```

        public List<DynamicPropertyInfo> getProperties() { return m_properties; }
        public List<DynamicMethodInfo> getMethods() { return m_methods; }
    }

    public static class DynamicPropertyInfo
    {
        String m_name;
        Class m_propClass;
        List m_references = new ArrayList();
        public void setName(String value) { m_name = value; }
        public String getName() { return m_name; }
        public void setPropClass(Class value) { m_propClass = value; }
        public Class getPropClass() { return m_propClass; }
        public List getReferences() { return m_references; }
        public void addReference(Object reference) { m_references.add(reference); }
    }
}

    public static class DynamicMethodInfo
    {
        String m_name;
        public void setName(String value) { m_name = value; }
        public String getName() { return m_name; }
    }
}
}

```

In the result you can pass back properties and action listener methods.

---

## Parameters of “introspectDynamically”

### “references”

The first parameter “references” contains additional information about a property's class. By default (i.e. coming from the introspection mode) the list may be filled with one object - representing the type class of a generic property type.

Example:

```

public class XYZBean
{
    ...
    public AddInfo<Article> getArticleInfo()
    {
        ...
    }
    ...
}

```

In case the Bean Browser wants to show the properties and methods below “articleInfo” then the references-list that is passed into the method “introspectDynamically” holds as element the “Article.class” instance. If there is no generic type available then the list is empty.

Later on you may return “DynamicPropertyInfo” objects as part of the result: each object again provides a list of references that you may set on your own. These references are used, when continuing to drill down the bean object hierarchy.

### “pathList”

The second parameter “pathList” contains the expressions of all Bean Browser nodes, starting with the top node and ending with the currently selected one. The list may look like...

```

#{d}
#{d.ArticleEditUI}
#{d.ArticleEditUI.article}

```

...assuming that the user is just about to open the Bean Browser tree below the “article” property.

# Appendix - Non Page Bean based Navigation

In the chapter “Page Navigation” the clear emphasis is on presenting navigation concepts based on the Page Bean framework. The framework was made available with release 3.0 of Enterprise Client. With release 5.0 we tool the non-page-bean-based navigation documentation out and now moved into this appendix.

As stated in the chapter “Page Navigation”: the page bean based navigation has a number of great advantages when comparing to the non page based navigation. So we definitely recommend to use page beans for any new development that you do!

---

## The ROWINCLUDE Component

Nesting page is very simple: you can nest one page into another page by using the “t:rowinclude” component within a page definition:

```
...  
<t:rowinclude page="def.jsp"/>  
...
```

The included page will be embedded into the including page - as if the XML definition was copied into the including page.

Of course the page-parameter can be defined dynamically:

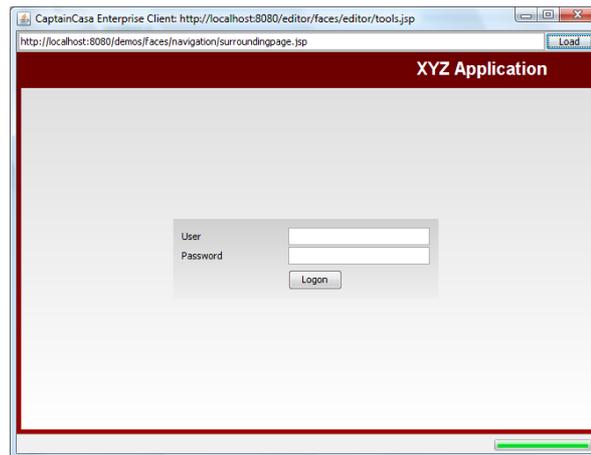
```
...  
<t:rowinclude page="#{bean.pageName}"/>  
...
```

It is possible to include pages into the included page as well, there is no limit of what you can do.

---

## Navigation Example

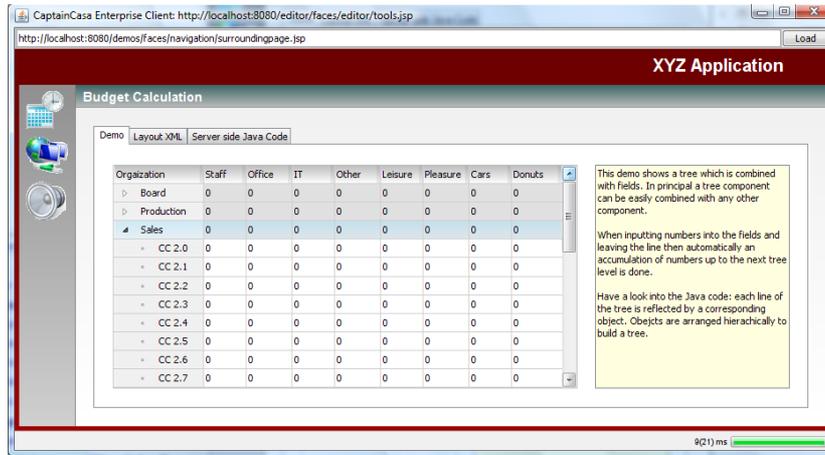
To demonstrate the usage of ROWINCLUDE have a look onto an example. The example starts with the following page:



After logging on successfully the page changes to look like:

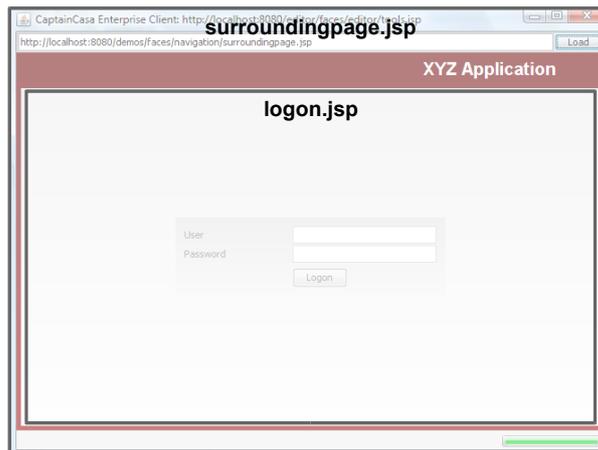


After clicking the top left icon the screen looks like:



## The most outside Page

The whole application is surrounded by a red frame with some text “XYZ Application” on the right top. Inside this frame there is an included page, which is first the “login page” and then the “workplace page”.



The surrounding page looks like:

```
<t:rowbodypane id="g_2"
bgpaint="rectangle(0,0,100%,100%,#6c0000,#A00000,vertical)" padding="5" >
  <t:row id="g_3" >
    <t:coldistance id="g_4" width="100%" />
    <t:label id="g_5" font="weight:bold;size:20" foreground="#FFFFFF" text="XYZ
Application" />
    <t:coldistance id="g_6" width="50" />
```

```

</t:row>
<t:rowdistance id="g_7" height="10" />
<t:row id="g_8" >
  <t:pane id="g_9" bgpaint="rectangle(0,0,100%,100%,#C0C0C0,#FFFFFF,vertical)"
height="100%" width="100%" >
  <t:rowinclude id="g_10" page="#{d.surroundingpageBean.contentPage}" />
</t:pane>
</t:row>
</t:rowbodypane>
<t:rowstatusbar id="g_11" />

```

The most important part is the ROWINCLUDE statement. The name of the included page is not defined statically but by expression reference.

The class implementation is quite simple:

```

public class SurroundingpageBean implements Serializable
{
  // -----
  // members
  // -----

  Dispatcher m_dispatcher;
  String m_contentPage = "logon.jsp";

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

  public SurroundingpageBean(Dispatcher dispatcher)
  {
    m_dispatcher = dispatcher;
  }

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

  public String getContentPage()
  {
    return m_contentPage;
  }

  public void switchToPage(String nextContentPage)
  {
    m_contentPage = nextContentPage;
  }
}

```

The initial “contentPage” is “logon.jsp” - so that's why the logon page is shown inside. By using the method “switchToPage()” the content can be changed.

## The logon Page

Inside the logon page the user/password input is checked. When correct the content of the surrounding page is exchange to no longer display the logon page but to display the workplace page.

The layout of the logon page is:

```

<t:rowbodypane id="g_2" >
  <t:rowdistance id="g_3" height="50%" />
  <t:row id="g_4" >
    <t:coldistance id="g_5" width="50%" />
    <t:pane id="g_6" bgpaint="rectangle(0,0,100%,100%,#D0D0D0,#F0F0F0,vertical)"
padding="10" rowdistance="2" width="300" >
      <t:row id="g_7" >
        <t:label id="g_8" text="User" width="120" />
        <t:field id="g_9" text="#{d.logonBean.user}" width="100%" />
      </t:row>
      <t:row id="g_10" >
        <t:label id="g_11" text="Password" width="120" />
        <t:password id="g_12" text="#{d.logonBean.password}" width="100%" />
      </t:row>
    </t:pane>
  </t:row>
</t:rowbodypane>

```

```

</t:row>
<t:rowdistance id="g_13" />
<t:row id="g_14" >
  <t:coldistance id="g_15" width="120" />
  <t:button id="g_16" actionListener="#{d.logonBean.onLogon}"
text="Logon" />
</t:row>
</t:pane>
<t:coldistance id="g_17" width="50%" />
</t:row>
<t:rowdistance id="g_18" height="50%" />
</t:rowbodypane>

```

The code behind the logon page is:

```

public class LogonBean implements Serializable
{
  // -----
  // members
  // -----

  Dispatcher m_dispatcher;
  String m_user = "";
  String m_password = "";

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

  public LogonBean(Dispatcher dispatcher)
  {
    m_dispatcher = dispatcher;
  }

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

  public void setUser(String value) { m_user = value; }
  public String getUser() { return m_user; }

  public void setPassword(String value) { m_password = value; }
  public String getPassword() { return m_password; }

  public void onLogon(ActionEvent ae)
  {
    if (m_user.equalsIgnoreCase("captain") &&
        m_password.equalsIgnoreCase("casa"))
    {
      // page sequence in outsidepage
      m_dispatcher.getSurroundingpageBean().switchToPage("workplace.jsp");
    }
    else
    {
      Statusbar.outputError("Logon data is not correct. Please retry.");
    }
  }
}

```

You see: during the onLogon method the bean of the surrounding page is contacted and the content page of the surrounding page is exchanged to be “workplace.jsp”.

## The Dispatcher Bean

You may wonder, how the “logon bean” and the “surroundingpage bean” know from one another. Looking into the code above there is a dispatcher bean which is passed into the page beans.

The dispatcher looks the following way:

```

public class Dispatcher implements Serializable
{
  // -----

```

```

// members
// -----

SurroundingpageBean m_surroundingpageBean;
LogonBean m_logonBean;
workplaceBean m_workplaceBean;

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

public SurroundingpageBean getSurroundingpageBean()
{
    if (m_surroundingpageBean == null)
        m_surroundingpageBean = new SurroundingpageBean(this);
    return m_surroundingpageBean;
}

public LogonBean getLogonBean()
{
    if (m_logonBean == null)
        m_logonBean = new LogonBean(this);
    return m_logonBean;
}

public workplaceBean getworkplaceBean()
{
    if (m_workplaceBean == null)
        m_workplaceBean = new workplaceBean(this);
    return m_workplaceBean;
}
}

```

The dispatcher is registered as managed bean in the faces-config.xml:

```

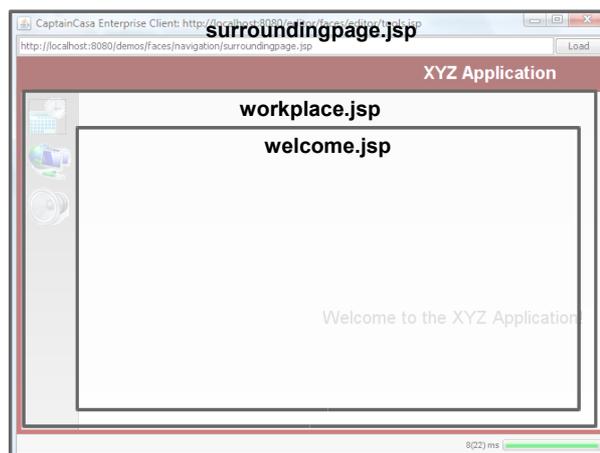
<managed-bean>
  <managed-bean-name>d</managed-bean-name>
  <managed-bean-class>navigation.Dispatcher</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

```

The dispatcher is the central access point, both for building expressions (“#{d.logon.user}”) and for internally accessing information.

## The Workplace Page

You will already imagine how the workplace bean itself is built:



On the left there are some icons, on the right there is ROWINCLUDE area. The ROWINCLUDE area shows the “welcome page” at the beginning. When pressing one of the icons, then a corresponding application page is replacing the “welcome page”. The principals of doing this are exactly the same as you got to know with the “surrounding page”.

---

## Nesting Pages - Conclusion

Looking at the example you see that it's possible to build up any type of nesting. As consequence the nesting and navigation structure is one of the most important design issues to become aware of. Dependent on your application you can end up in very flexible scenarios, in which at least a big part of the navigation logic can be flexibly outsourced into declarations (e.g. XML files).

Example: In the example shown above all page names are hard coded into the code of the beans. This means of course that changing page names directly has an impact on the bean implementation.

---

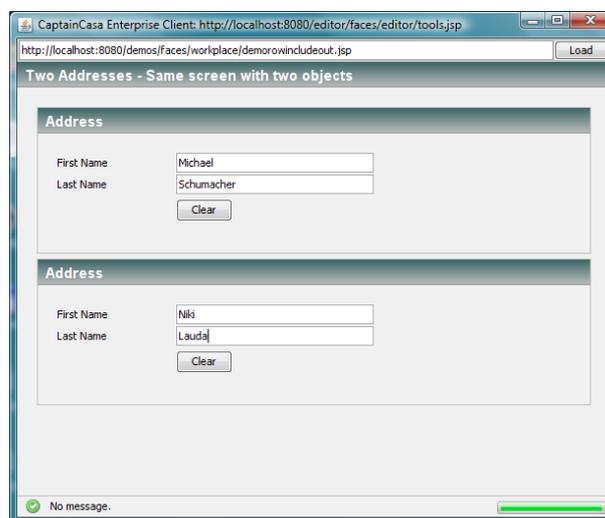
## ROWINCLUDE - Extended Usage

The ROWINCLUDE component provides an interesting attribute - the attribute CONTENTREPLACE. The meaning of the attribute is quite simple: it allows to change the attribute values of the included page just at the point of time when the page is embedded into the surrounding page.

Please imagine what internally happens: on server side the ROWINCLUDE component reads the JSP definition of the included page, i.e. It parses the XML tag definitions and builds a corresponding JSF component tree. During this reading and parsing each value is scanned for replacements when being transferred into an attribute value of a JSF component.

So, that's what CONTENTRPLECE is used for... - the result of what you can do now, is very powerful.

Have a look onto the following example:



You see one "outside page" that includes two "inside pages". Both "inside pages" are the same page layout (the representation of an address) - but: they are bound to two different objects. You see that the data within the "inside pages" are not the same, and: when pressing the clear button, only the corresponding address data is cleared, the other data remains untouched.

How is this possible? - Normally (e.g. you may know about `jsp:include`) it is NOT possible to have one page definition binding two different objects, because the binding is part of the page definition...

Let's have a closer look: the "inside page" definition is as follows:

```
<f:view>
<h:form>
```

```

<f:subview id="workplace_demorowincludeing_31">
<t:rowtitlebar id="g_2" text="Address" />
<t:rowbodypane id="g_3" >
  <t:row id="g_4" >
    <t:label id="g_5" text="First Name" width="120" />
    <t:field id="g_6" text="#{d.address1.firstName}" width="200" />
  </t:row>
  <t:rowdistance id="g_7" />
  <t:row id="g_8" >
    <t:label id="g_9" text="Last Name" width="120" />
    <t:field id="g_10" text="#{d.address1.lastName}" width="200" />
  </t:row>
  <t:rowdistance id="g_11" height="5" />
  <t:row id="g_12" >
    <t:coldistance id="g_13" width="120" />
    <t:button id="g_14" actionListener="#{d.address1.onClear}" text="Clear" />
  </t:row>
</t:rowbodypane>
</f:subview>
</h:form>
</f:view>

```

The binding of the fields is defined to bind to “wp.address1”-properties. The binding of the button action listener references to a “wp.address1”, too.

Now let’s have a look onto the outside page:

```

<f:view>
<h:form>
<f:subview id="workplace_demorowincludoutg_21">
<t:rowtitlebar id="g_2" text="Two Addresses - Same screen with two objects" />
<t:rowbodypane id="g_3" >
  <t:row id="g_4" >
    <t:pane id="g_5" border="#C0C0C0" height="150" padding="1" width="100%" >
      <t:rowinclude id="g_6" page="demorowincludin.jsp" />
    </t:pane>
  </t:row>
  <t:rowdistance id="g_7" height="5" />
  <t:row id="g_8" >
    <t:pane id="g_9" border="#C0C0C0" height="150" padding="1" width="100%" >
      <t:rowinclude id="g_10" contentreplace="address1:address2"
page="demorowincludin.jsp" />
    </t:pane>
  </t:row>
</t:rowbodypane>
<t:rowstatusbar id="g_11" />
</f:subview>
</h:form>
</f:view>

```

The first ROWINCLUDE component is defined “just normal”: it references the page “demorowincludin.jsp”, which is the address page.

Now, the second ROWINCLUDE component is “special”: it references the same page as the first ROWINCLUDE component, but in addition specifies a content replacement: “address1:address2”. This means that in the included page, every value occurrence of “address1” is replaced by “address2”. Because “address1” only occurs in the binding definition of fields and buttons the result of the content replacement is a page that has the same layout as the normal page, but binds to “address2” instead of “address1”.

One page - used two times in parallel with two different objects! This is the result.

Please note: the definition of CONTENTREPLACE allows you to concatenate several replacement definitions by separating them via semicolon. A valid definition containing three replacements may look like:

```
address1:address2; $i$:1; Hugo:Harry
```

## ROWINCLUDE - Important Note!

Please note that the ROWINCLUDE component may only be used with 100% straight “Enterprise Client” pages - this means: pages, that internally contain a pure XML tag definition, without JSP code mixed in.

---

## Popups

### Modal Popups

Any page can be called as modal popup. From the calling page you need to access the popup management via a set of Java APIs.

For opening a popup there are three possibilities:

- There is a “low level API” available that directly talks to the popup components.
- There is a “high level API” when using popups in a workplace environment. The workplace environment is an optional feature on its own that is described in a separate chapter of this documentation.
- And there is a “high level API” when opening pages that are based on PageBean modularization.

The rule is simple: use the “high level APIs” when using page beans or the workplace management, otherwise use the “low level API”.

This is the code to open a popup using the “low level API”:

```
public class DemoPopup
    implements IModalPopupListener
{
    ModalPopup m_modalPopup;

    public void onOpenPopup(ActionEvent e)
    {
        m_modalPopup = ModalPopup.createInstance();
        m_modalPopup.open("/workplace/demobudget.jsp",
            "Budget Calculation",
            800,
            600,
            this);
    }

    ...

    public void reactOnPopupClosedByUser()
    {
        m_modalPopup.close();
    }
}
```

The popup page will be opened accordingly. The last parameter in the open-Method is a pointer to an interface “IModalPopupListener”. Via this interface you get notified when the user wants to close the popup using the close-corner-button:

```
public interface IModalPopupListener
{
    public void reactOnPopupClosedByUser();
}
```

Closing a popup is done using the ModalPopup.close() method.

**PAY ATTENTION** when specifying the name of the page that you want to open within the popup: **the name must be the absolute address of the page within your web application, starting with “/”**.

When using the “high level API” within the context of a workspace then everything is the same but just the opening of the popup is done in a different way:

```
public class DemoPopup
    extends DemoBase
    implements IModalPopupListener
{
    public void onOpenPopup(ActionEvent ae)
    {
        m_modalPopup = getworkspace().
            createModalPopupInWorkspaceContext();
        m_modalPopup.open("/workspace/demohelloworld.jsp",
            "Hello world!",
            800,
            300,
            this);
    }
}
```

Please check the documentation about PageBean modularization to see how popups are opened in this context.

## Modeless Popups

Modeless popups can be called in the same way as modal Popups - just by using the corresponding Java API with the name “ModelessPopup” instead of “ModalPopup”.

In short, this means:

- “Low level API”: You open instances of modeless popups by calling the function “ModelessPopup.createInstance()”
- “High level API”: you open instances of modeless popups by calling the function “Workspace.createModelessPopupInWorkspaceContext()”
- You open, close them by using the “open()” and “close()” functions of the popup instance
- You implement interface “ImodelessPopupListener” in order to get notified when the user requests to close the popup.

## Popup Closing

Note that it's always your server side program that closes a popup. A popup will never close itself without your program being the trigger for closing.

A popup that is shown in default, decorated mode will not close itself when the user presses the close-icon on the top right. Instead it will send a request to the server side. The server side will then call back the interface IModal/IModelessPopupListener - in the implementation of this interface you can add code to close the popup.

Also note that it is the opener's responsibility to close the popup. The popup does not close itself, it needs to be closed. The advantage of this architecture is, that any page can be put into a popup by another page without being aware of running in a popup at all. It's the opening page that is responsible for the popup management.

## Popup Decoration

Popup decoration by default is set to “true”: a popup comes up with a border, with a title bar, with a close-icon, etc. When specifying the attribute “decorated” with false, then a popup will open up without this default decoration.

```

m_modalPopup = ModalPopup.createInstance();
m_modalPopup.open("demobudget.jsp",
    "Budget Calculation",800,600,this);
m_modalPopup.setDecorated(false); // no decoration

```

The same is possible for modeless popups as well, of course.

## Popup Data Synchronization

Because popups run within a window on their own you may assume that popup windows talk to the server independently from their underlying page. But: that's not true at all, by intention!

A request that is triggered by a popup (e.g. when pressing a button in a popup) always is a request of the complete page (underlying page + popup) to the server. Advantage: you may update data of the underlying page as part of the popup processing. The changed data will be visible to the user, though e.g. working in a modal popup.

Imagine the popup being an optical area which is an “outsourced” part of the page. It runs inside a different window, but still is a part of the page, just as a straight PANE component also is a part of the page.

## Popup Background Style Settings

When working with transparent colors then all pages rely on a certain background painting - which normally comes from the out-most page of your screen. Popup dialogs are windows on their own: they do not take over coloring from the background. As consequence there is a way of telling popups which background to use:

```

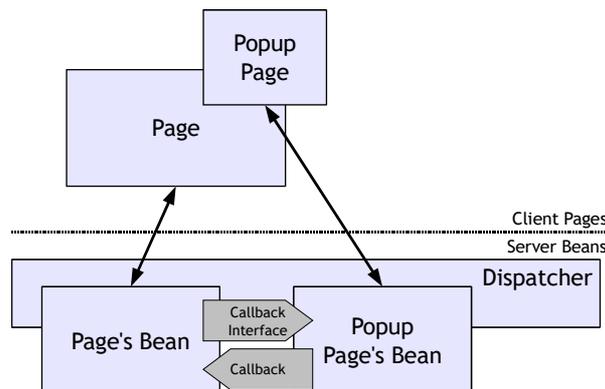
BasePopup.initialize("rectangle(0,0,100%,100%,#F0F0F0,#D0D0D0,vertical)");

```

By using the “initialize()”-method of the class “BasePopup” you can pass a BGPAIN expression that is used for rendering the background of all popup dialogs.

## Writing generically usable Popups

In many cases you want to create popup pages which are generically usable. For example you want to create a popup for searching employees and you want to use this popup from many screens.



On server side the calling page and the popup page are independent beans. The calling page's bean can reach the popup page's bean, e.g. due to a dispatcher bean that owns both.

It is useful to work with a interface based callback structure. This means: the popup page's code may look like:

```

public class EmployeeSelectionBean ...
{
    public interface IEmployeeCallback
    {
        public void employeeSelected(String employeeId);
    }

    IEmployeeCallback m_callback;

    public void prepare(IEmployeeCallback callback)
    {
        m_callback = callback;
    }

    ...

    public void onEmployeeSelect(ActionEvent ae)
    {
        // call back!
        m_callback.employeeSelected(...);
    }
}

```

The calling page's code may look like:

```

public class CallingPageBean ...
{
    ModalPopup m_popup;

    String m_employeeId;
    public String getEmployeeId() { return m_employeeId; }
    public void setEmployeeId(String id) { m_employeeId = id; }

    ...

    public void onOpenEmployeeSelection(ActionEvent event)
    {
        // data navigation
        EmployeeSelectionBean esb = ...; // connect to other bean, e.g. by
dispatcher
        esb.prepare(new IemployeeCallback()
        {
            public void employeeSelected(String employeeId)
            {
                setEmployeeId(employeeId);
                m_popup.close();
            }
        });
        // page navigation
        m_popup = getWorkpage().createModalPopupInWorkpageContext();
        m:popup.open(...);
    }
}

```

You see:

- The popup defines a callback interface. In the example this is an innter interface class - it could of course be a “real” class as well.
- The popup defines a method that allows the caller to pass a callback interface implementation. The callback is done when the employee is selected in the popup.
- The calling page passes an interface implementation instance and reacts accordingly, if the callback is executed.

Result: calling code and popup code are isolated from one another. All communication is going through a callback interface that does not contain any semantics of the calling bean. The popup itself does not know anything about the calling page.

## Sizing Popups

By default the size of the popup is passed within its corresponding open method:

```
m_modalPopup.open("/workplace/demobudget.jsp",  
                  "Budget Calculation",  
                  800,  
                  600,  
                  this);
```

As usual the “pixels” that are passed are translated at client side into visual pixels, dependent from the scaling on client side. (By default the scaling is 100%, so that one pixel is really represented by one physical pixel.)

You may also pass a size of “0,0” - in this case the popup will occupy exactly this space that is required by the content of the contained page.

But, pay attention: in many cases you “by default” put the content of a page into a ROWPAGEBODY or into a SCROLLPANE. Consequence: these components from an outside view do not occupy the space that is required by their content - because they hide the content. If the content is too big, then these components start scrolling the content, rather than requesting more space on their owns.

Consequence: pages that you want to include into popups, that should be sized automatically, need to be defined without an inner ROWPAGEBODY or SCROLLPANE component.

## Positioning Popups

By default the client tries to position the popup window as good as possible - this means it tries to position next to the component from which the last request that triggered the popup was initiated.

You may also explicitly set the x,y - position of the popup or you may explicitly position the popup next to a dedicated other component of the screen.

Please view the examples in the demo workplace for more information.

# Appendix - Starting Enterprise Client Pages

## Three Possibilities

Pages that you create by using the Enterprise Client Framework can be started in three ways:

- as applet
- as web start application
- as command line application

## Applet

The applet tag looks like:

```
<html>

<body leftmargin=0 rightmargin=0 topmargin=0 bottommargin=0 style="overflow:
hidden" onhelp="ccDoNothing();">

<script type="text/javascript">
// this code is used by APPLETEXTSHOWDOCUMENT - if this is configured to
// use java script instead of directly contacting the context
function ccOpenPageInTarget(ccTarget,ccURL)
{
    var ccWindow = window.open(ccURL,ccTarget);
    ccWindow.focus();
}
function ccDoNothing()
{
    alert("ccDoNothing");
    event.returnValue = false;
}
</script>

<applet code="org.ecInt.client.page.PageApplet.class"
        archive="ecInt/lib/ecInt.jar" width="100%" height="100%"
        MAYSCRIPT">
    <param name='page' value='faces/workplace/workplace.jsp'>
    <param name="image" value="ecInt/images/splash.png">
    <param name="centerimage" value="true">
    <param name="java_arguments" value="-Xmx256m">
    <param name="java_version" value="1.6+ ">
    <param name="separate_jvm" value="true">
    <param name="MAYSCRIPT" value="true">

    <!-- This text appears in case the applet does not show up. -->
    The applet could not be loaded - please install the Java runtime
    plugin: <a
href="http://www.java.com/de/download">http://www.java.com/download</a>

</applet>

</body>
</html>
```

Additional parameters can be passed via “<param name=” value=”>”. Example:

```
<applet code="org.ecInt.client.page.PageApplet.class"
        archive="ecInt/lib/ecInt.jar" width="100%" height="100%"
        MAYSCRIPT">
    <param name='page' value='faces/workplace/workplace.jsp'>
    <param name="image" value="ecInt/images/splash.png">
    <param name="centerimage" value="true">
    <param name="java_arguments" value="-Xmx256m">
    <param name="java_version" value="1.6+ ">
    <param name="separate_jvm" value="true">
    <param name="MAYSCRIPT" value="true">
```

```

<param name="disabledcolor" value="#00000030">
<param name="loglevel" value="INFO">

<!-- This text appears in case the applet does not show up. -->
The applet could not be loaded - please install the Java runtime
plugin: <a
href="http://www.java.com/de/download">http://www.java.com/download</a>
</applet>

```

## Web Start

The JNLP file looks like:

```

<?xml version="1.0" encoding="utf-8"?>
<jnlp spec="1.0" codebase="$$codebase" href="$$name">
  <information>
    <title>CaptainCasa Enterprise Client</title>
    <vendor>CaptainCasa GmbH</vendor>
    <icon href="ecInt/images/splash.png" kind="splash"/>
  </information>

  <!-- -->
  <security>
    <all-permissions />
  </security>
  <!-- -->

  <resources>
    <j2se version="1.6+" java-vm-args="-esa -xnoclassgc"/>
    <jar href="ecInt/lib/ecInt.jar"/>

    <!-- ONLY REQUIRED WHEN USING BROWSER OR ACTIVEX COMPONENT! -->
    <!-- -->
    <jar href="ecInt/lib/swt.jar"/>
    <nativelib href="ecInt/lib/swt.jar"/>
    <!-- -->
  </resources>

  <application-desc main-class="org.ecInt.client.page.PagewebStart">
    <argument>$$codebase</argument>
    <argument>faces/workplace/workplace.jsp</argument>
  </application-desc>
</jnlp>

```

Additional parameters can be added after the two default arguments withing the “application-desc” section. Example:

```

<application-desc main-class="org.ecInt.client.page.PagewebStart">
  <argument>$$codebase</argument>
  <argument>faces/workplace/workplace.jsp</argument>
  <argument>orientation=rtl</argument>
  <argument>sizefactor=1.5</argument>
  <argument>fontfactor=1.5</argument>
</application-desc>

```

The first two parameters' meaning is fix, whereas the follow on parameters can be in any sequence.

## Command Line

You require a local installation of JRE 1.6 in order to run the client by command line.

```

set TCP=..\resources\webappaddons\ecInt\lib\ecInt.jar

rem following jar is only needed when using native components
rem set TCP=%TCP%;..\resources\webappaddons\ecInt\lib\swt.jar

start ire\bin\iavaw.exe -cp %TCP% org.ecInt.client.page.PageBrowser

```

```
http://localhost:50000 /demos/faces/workplace/workplace.jsp
```

Additional parameters can be passed in the following way:

```
start jre\bin\javaw.exe -cp %TCP% org.eclnt.client.page.PageBrowser  
http://localhost:50000 /demos/faces/workplace/workplace.jsp orientation=rtl  
sizefactor=1.5 fontfactor=1.5
```

---

## Usage with .ccapplet and .ccwebstart

When using “.ccapplet” and “.ccwebstart” for dynamically creating the .jnlp and .html file for starting your page, then you may maintain the client start parameters within the configuration file “eclntjsfserver/config/system.xml”. Please take a look into the template file within the same directory in order to see the corresponding XML definition:

```
<system>  
...  
...  
  <ccappletccwebstart ...>  
    <clientparam name="loglevel" value="INFO"/>  
    <clientparam name="xxx" value="yyy"/>  
  </ccappletccwebstart>  
...  
...  
</system>
```

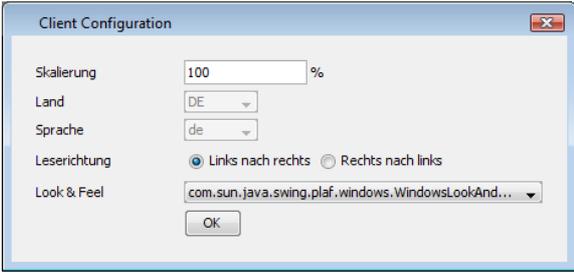
---

## Passing Client Parameters

### List of Parameters

A couple of parameters can be passed in order to start a page. These parameters are listed in the following table:

Parameter	Description
orientation	Either “ltr” (default) or “rtl”, for “left to right” and “right to left”  (optional)
language	Language that is set within the client's locale definition. ISO code for language. See JavaDoc of class “Locale” for more information.  (optional)
country	ISO Code for the country that is set within the client's locale definition. See JavaDoc of class “Locale” for more information.  (optional)
timezone	The timezone in which the client is started. If not explicitly specified then the default timezone of the client is determined by the Java environment of the client.  The timezone is extremely important for all components that deal with “Date”-objects, e.g. CALENDARFIELD. All these components allow the direct definition of the timezone, but fall back to default timezone, if no explicit definition is done.  (optional) - We recommend to set this parameter and/or to at least have a very clear understanding about where time-related component instances receive their time zone from.
sizefactor	Float value by which all pixel sizes are multiplied in the client. Default is “1.0”.  (optional)
fontfactor	Float value by which all font sizes are multiplied in the client. Default is “1.0”.  (optional)
imagefactor	Float value by which all image sizes are multiplied in the client. Default is “1.0”. This factor is a quite important one: you may decide to in general show your application with a sizefactor of e.g 1.5. This means, that as result (when not explicitly setting the sizefactor) all images will be upscaled accordingly. As result of upscaling the quality of the image will suffer...  With defining imagefactor you now can provide all images in a big, “150%” version - and tell the system to keep this versions by setting imagefactor to “1”.  (optional)
loglevel	Log level for client side logging  Valid values are “ALL”, “INFO”, “WARNING”, “SEVERE”
clientlogfile	Name of file, in which log information is written on the client side. Should only be set in development test scenarios!  The log by default is written to the client console.
headerline	Boolean value that indicates if a browser-like URL section is displayed at the top of the client window. Default is “false”.  (optional)

footerline	<p>Boolean value that indicates if footer icons are rendered, typically located on the bottom right of the screen. Default is “false”</p> <p>(optional)</p>
progressbar footerlinereload footerlinetools footerlineduration	<p>Swing client: if “footerline” is defined to “true” then you may define if the individual parts of the footer line are rendered (“true”, default) or not (“false”). The individual parts are: ...a progress bar ...the reload icon ...the configuration/tools icon ...the message showing the durations for the last request processing</p>
loglevel	<p>Granularity of client side console logging. Valid values are “SEVERE”, “WARNING”, “INFO”, “FINE” or “ALL”.</p>
lookandfeel	<p>Class name of the Java look and feel that you want to use for the client. Pass the value “system” for using the client’s system look and feel - this is the one that is closest to the outside operating system environment.</p> <p>By default the “CaptainCasaLookAndFeel” is used - that provides a cross-system compatibility. Screens will look exactly the same in any operating system environment.</p> <p>Example: “org.eclnt.client.lookandfeel.CaptainCasaLookAndFeel”.</p>
lookandfeelcheckduringruntime	<p>When setting “lookandfeelcheckduringruntime” to “true” then with each response processing within the client the client checks if the current look and feel really is the one that should be used with the client. - This parameter is useful when integrating the CaptainCasa client into existing Swing applications: there may be cases in which the Swing application updates the look and feel on its own - consequence: components in the CaptainCasa client are drawn with the wrong look and feel.</p>
restrictedconfig	<p>Defines if the client configuration dialog allows the user to set language and country settings. If “false” (default) then the user can input these parameters, if “true” then the user can only see them:</p>  <p>Background: the decision about language and country can be done by the server side application, by using the CLIENTCONFIG component.</p>
startx starty startwidth startheight	<p>Coordinates for positioning the window when using web start. If not set then the window will be started at (0,0,1024,768). For startwidth and startheight you may pass “max” as value - in this case the screen’s maximum size will be chosen as size.</p>
showsplashurl	<p>(Swing client only) If set to “false” then the URL is not output below the splash image during start up of the client.</p>
maximized	<p>If set to “true” then the client is started in maximized mode. Only applicable when started by web start or as application.</p>

confirmexit	If set to “false” then the user will not be asked if he/she really wants to close the client. Only applicable for web start / application scenario (not for applet scenario).
errorscreen	In case the server side responds with an error then an error page is shown in the client. This error page can be configured by passing the errorpage parameter.  The default error page is “/faces/eclntjsfserver/includes/showservererror.jsp”.  Keep the same syntax as with the default error page when defining your own page, i.e. start with “/faces/” as well.
imagebuffersize	Images (e.g. icon images) are loaded from the client by http access. In order to reduce the traffic between client and server images are buffered within the client side memory for the duration of one client session. The buffer has a maximum number of images that is defined by the IMAGEBUFFERSIZE parameter. Once the client reaches the defined size then a clean-up will happen: images are removed from the buffer according to their last usage point of time.
selectioncolor1 selectioncolor2 rollovercolor1 rollovercolor2	Color definitions for all grid selection and rollover effects. The color should be defined with transparency (#rrggbbtt) so that the original background has some chance to shine through. The first color is the top-color, the second color is the bottom color. There will be a corresponding gradient painting.
selectionbgpaint	By default the background of a selected grid item is a gradient coloring from one color (selectioncolor1) to another color (selectioncolor2). But you can also define a specific bgpaint-command that e.g. does not do any coloring but some drawing of borders. The selectionbgpaint-command is applied to each cell of the selected line. If it is defined then it overrides any definitions of selectioncolor1 and selectioncolor2.
gridscrolldelay	When moving the scrollbar of a (grid) with the mouse then there is a certain duration called “gridscrolldelay”: if the user does not move the scrollbar while still clicking the mouse then the system will load the corresponding data at the current scroll position. If the value is too low then there is the risk of having a lot of round trips between client and server when the user scrolls by dragging/dropping the scroll bar.
gridcellpadding	Definition of the padding that is used for grid cells. Example: “left:2;top:2;bottom:2;right:2” (<= this is the default).
disabledcolor	Background color that is used for disabled components. The disabled color should be defined as transparent color so that the original background color has some chance to shine through. Default value is “#00000008” (= a quite transparently defined black)
kiosk	Removes all decoration from the window in which the Enterprise Client is started. Often used in combination with startx=0 starty=0 startwidth=max startheight=max.
touch	If set to “true” then scrollbars will be drawn wider/higher than normally in order to allow a usage via touch screen
jsessionIdname	The literal that is used to indicate the http session id within a request's URL. The default is “jsessionid” - the session id is encoded by inserting a string “;jsessionid=xxxxxxxx” into URLs being requested from client side to server side.  Please only change if your web server processing explicitly is configured to use a different session-id identifier.

clientid	The id that identifies the client “logically”. This client-id typically is set by using the .ccapplet / .ccwebstart way of starting the client. The client id is transferred as http-header parameter with every request.
animate	If set to “false” then all implicit animations are switched off. This may be useful is accessing the client via “remote terminal”, in order to reduce the data volume transferred to the terminal server.
sound	If set to “false” then no sound will be played.
reconnectpopup	<p>If the client sends data to the server then the server might not respond. The reason may be a temporary blocking of the server (e.g. due to high load) or temporary network problems.</p> <p>The client reacts by trying to reconnect to the server. (Of course there is a certain mechanism to prevent double-processing of the same request on sever side...=. After a certain number of reconnect attempts by default a popup comes up to the user, asking if the user wants to continue with trying to reconnect.</p> <p>By setting reconnectpopup to “false” you can switch off this popup - and directly show the error screen to the user, that the server is not reachable.</p>
numberofreconnects	Number of attempts the client re-tries to connect to the server in order to send a request, after having failed to do so. The default is “1”. You can pass any integer value >= 0.
urlconnectiontimeoutconnect	Timeout in milliseconds that the client waits when opening a URL connection to the server. Default is “0”, which means “infinite”.
urlconnectiontimeoutread	Timeout in milliseconds that the client waits for receiving the response afetr having connected to the server. Default is “0” which means “infinite”.
downloadopenimmediately	<p>When downloading files from the server to the client there is an option that these files are directly opened within the client environment. This immediate opening only is executed if the client is configured to accepts this - by defining “true” with this parameter.</p> <p>The default is “false” - i.e. no files are opened directly after download. As result the user by default always needs to explicitly open the file in his/her own.</p>
focusbackgrounddrawing focusdarkbgpaint focuslightbgpaint	<p>Parameters defining the background drawing of focused components. By default components holding the focus are painted with a certain background shading so that the user better can identify where the current focus is located.</p> <p>focusbackgrounddrawing: by default is “true”, but can be set to “false”. In this case no background drawing at all is active. All other parameters in this case are irrelevant.</p> <p>BGPAINt rendering commands for rendering diverse component backgrounds:</p> <p>focus* ==&gt; rendering of field's, ... background when the component is focused</p> <p>There are two classes of components...: “light” components are the ones typically holding a light (e.g. white background), e.g. fields. Dark components typically hold a dark background.</p>

<p>regexerrorbgpaint errorbgpaint errorbgpaintcombo mandatorybgpaint mandatorybgpaintcombo</p>	<p>BGPAINT rendering commands for rendering diverse component backgrounds:</p> <p>error* ==&gt; explicit definition of how the error()-BGPAINT-command is executed mandatory* =&gt; explicit definition of how the mandatory()-BGPAINT-command is executed</p>
<p>focusdrawing focusdrawingcontrols focusdrawingcolors</p>	<p>focusdrawing: By default certain focused components are surrounded with some rectangle, that moves with the currently focused component. You can switch off this rectangle by defining “false”. In this case all other parameters are irrelevant.</p> <p>focusdrawingcontrols: A semicolon separated list of client controls, for which an explicit focus border is drawn. The controls must be written in capital letters, e.g. “FIELD;LABEL;COMBOFIELD”. Purpose behind this parameter: the drawing of a rectangle around the focused component does not make sense for all components - e.g. it does not look nicely with buttons. As consequence, all components for which a rectangle is to be drawn, are listed in this parameter. You only need to set the parameter, if you do not like the default selection of components, for which the rectangle is drawn.</p> <p>focusdrawingcolors: The definition of colors that are used for drawing the border around the controls - as semicolon-separated list of color values. The default is “#00000040;#00000028;#00000010;#00000008”: as result the border width is 4 pixels, the first border line around the control is drawn with “#00000040”, the second is drawn with “#00000028”, etc.</p>
<p>imagetreenodeopened, imagetreenodeclose, imagetreenodeendenode</p>	<p>General definition of the images that are used within the tree node components. Pass as parameter the link to a server side image (e.g. “/images/xxx.png” with “/images” being a directory within you web content.</p>
<p>imagesortup imagesortdown  sortimagecentered</p>	<p>Explicit definition of the images that are shown within the header column of grids for indicating the sort status. Pass as parameter the link to a server side image (e.g. “/images/xxx.png” with “/images” being a directory within you web content.</p> <p>By default the sort image is aligned in the center of the grid column header. By defining “sortimagecentered” as “false”, the image will be aligned at the opposite direction of the text alignment of the grid column header.</p>
<p>transferxmlinputupdate</p>	<p>This is a very special one: during communication we provide a certain logic to check if the XML received by the server is correct. In some situations we found that a certain proxy changed the XML content returned by transferring “&lt;input.../&gt;” into “&lt;input...&gt;” (slash was removed) - which is fine from an HTML perspective but not from an XML perspective. The corresponding check logic by default is switched on, you can switch off explicitly with this parameter.</p> <p>Default value is “true”, you can set to “false”.</p>
<p>errorscreenproviderclassName</p>	<p>This is a very special one: in the client processing an error screen is built up in certain situations, e.g. in which the server is not available. You can - by explicitly adding code to the client - manipulate this error screen by implementing the interface “ILocalErrorScreenProvider” and by registering your class name via this parameter.</p>
<p>unfilledbuttonpressedbgpain</p>	<p>BGPAINT command that is used for drawing the background of buttons</p>

t unfilledbuttonmouseoverbgp aint	that have set CONTENTAREAFILLED to “false”.
filledbuttoncolor1 filledbuttoncolor2 filledbuttonpressedcolor1 filledbuttonpressedcolor2 filledbuttonmouseovercolor1 filledbuttonmouseovercolor2 filledbuttonradius	Customization of button look and feel for the CaptainCasaLookAndFeel: the button's color definition consists out of two colors - the one on the top, the other on the bottom of the button. In addition you can define a radius defining the rounding that is applied within the edges of the button. Color values need to be passed in the format #rrggb or #rrggbtt, the radius is a pixel value, the default being “13”.
embeddedwebappdir	When starting the client with embedded tomcat (see documentation “Embedded Usage Mode”) then by default the directory “embeddedserver/webapp” is chosen as webapp directory. You can assign an own directory, if not keeping to the standard usage scenario.
dateresolutionintopast	By default is “true”, you can set the parameter to “false”.  This attribute indicates that the automated resolution of a date when being define with a date controls (CALENDARFIELD) is done in the following way: If the current date is “15.05.2010” and the user types “13” then by default the 13st of the current month, i.e. “13.05.2020” is resolved. When setting the parameter to “false” then no date which is in the past will be resolved, but the next date in the future, i.e. “13.06.2010”.
century19resolution	Integer value between 0 and 99 that indicates if a two-digit date input should be resolved into “19xx” or “20xx”. Example: if defining “70” then all abbreviated year input which is higher or equal to “70” is resolved to “19xx”, otherwise to “20xx”.
tabonenter	If set to “true” then “enter” in fields etc. will cause a “tab”-event. Result: the focus is moved to the next focus-able component.
hotkeyrowexecute	Hotkey definition for the key that triggers the “row-execute” processing in grids. By default the “row-execute” is triggered with the “enter”-key. You can select a different key by using this attribute.
userhintfont	Font definition (“family:...;size:...”) that is used for rendering the user hints. (the small yellow popup dialogs that come up, when navigating into a corresponding component)
textselectionforeground textselectionbackground	Selection color (foreground and background) for all text components (FIELD, FORMATTEDFIELD, TEXTPANE, ...). Define color values (#RRGGBB or #RRGGBBTT).
reloadonsessiontimeout	By default an error screen is shown when a session timeout occurs on server side. When setting the parameter “reloadonsessiontimeout” to “true” then just after showing the error screen a new session is requested from server side. Consequence: the application re-occurs with its start screen.
configcountries configlanguages	Semicolon-separated list of country/language codes that are available in the configuration popup. If undefined then the full list of ISO values is shown.
popupmenucopyclipboard	By default text components provide a popup menu (right mouse button menu) providing the function to copy the text content to the client clipboard. If you do not want to see this automatically created popup menu, then define this parameter as “false”.
tooltipfont	Font that is used for tooltips. This is a “normal” CaptainCasa font definition, e.g. as used in FIELD-FONT etc.  Example: “size:12”.

scrollbarrounding	By default the CaptaiCasa Look and Feel renders scrollbars with a certain rounding. You may define the rounding on your own. If defining "0" then there is no rounding at all - scrollbars will be rendered as rectangle.
scrollbarcolorthumb scrollbarcolorthumb2 scrollbarcolorthumbrollover scrollbarcolorthumb2rollover scrollbarcolortrack scrollbarcolorborder scrollbarcolorborderrollover scrollbarcolorarrow scrollbarcolorfocsuarrow scrollbarcolorbutton	Explicit definition of colors ("#rrggbb","#rrggbbtt") for default scrollbars.
flatmode	Defines if dialogs are rendered with the normal, operating system's window decoration (titlebar, borders, ...) or if dialogs are rendered with a special, CaptainCasa decoration - by default providing a flat design.  "true" - switch flat mode on "false" - normal, operating system dialogs (default)
flatmodeborderframe flatmodeborderdialogmodal flatmodeborderdialogmodeless flatmodetitlebgpaintframe flatmodetitlebgpaintdialogmodal flatmodetitlebgpaintdialogmodeless flatmodetitleforeground flatmodetitlefont flatmodetitleborder	Fine controlling the visual appearance of the dialog decoration, if flat mode is switched on.  ...for "**border**" use normal border definitions (e.g. "color:#FF0000;left:1;right:1;top:1;bottom:1")  ...for "**bgpaint**" use background painting commands  ...for "**foreground**" use color definition ("#rrggbb")  ...for "**font**" use fond definition (e.g. "size:14;color:#FFFFFF")
authenticationuser authenticationpassword	Username and password that are passed via basic authentication within the header of http requests that are sent from the client to the server
clientliteralsextensionclassname	Name of a client class implementing interface "IClientLiteralsExtension". By using the interface all internal requests for reading client literals are first transferred to the interface before they are read using property files. The interface is used in order to influence the text literals that are used and shown inside the client.

## Defining Client Parameters to be copied into http-Header

When defining a client parameter the following way...

```
http-header:<name>=<value>
e.g.: "http-header:tokenId=4711"
```

...then the corresponding information (i.e. the value of "<name>" and "<value>") will be added to each http-request as http-heaser attribute that is sent from the client to the server for data/event-synchronization.

On server side you might access the values using class "HttpSessionAccess", method "getCurrentHttpRequest".

## Defining Client Parameters to be transferred into cookies

When defining a client parameter in the following way...

```
http-cookie:<name>=<value>
```

```
e.g.: "http-cookie:sessiontokenid=0815"
```

...then the corresponding information is added as a cookie to the request(s) that are sent from the client to the server processing.

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

[www.CaptainCasa.com](http://www.CaptainCasa.com)  
[info@CaptainCasa.com](mailto:info@CaptainCasa.com)