CaptainCasa Enterprise Client

Tool Extensions

Table of Contents

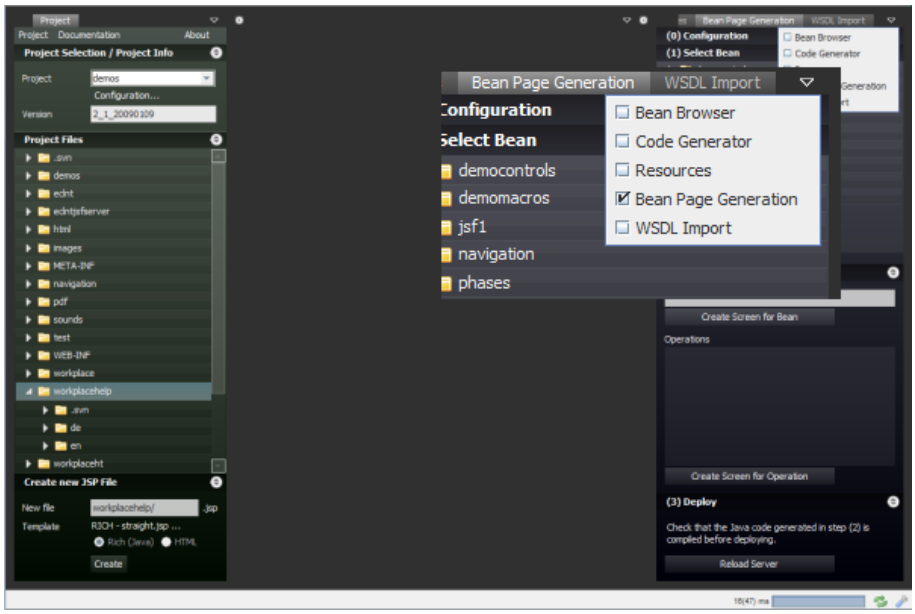
There are three concepts available for adding new features to the CaptainCasa Enterprise Client toolset:

- You may add new tool user interfaces into the tool area “on the right”. These new pages may having nothing to do with existing, other tools – there design and function is completely up to you. These tool user interface get notified about certain events within the toolset (e.g. project is loaded) and you can drag & drop information from the tool user interface into the Layout Editor.
- You may influence the Bean Browser to display the “expression tree” following your needs. The normal Bean Browser does introspect the classes of your project and follows the set/get methods for drill down. When using generic beans (e.g. beans supporting a Map-based interface for accessing properties) then the class introspection is not adequate and can be replaced by your own introspection.
- You may define the expression hierarchy of the so called Expression Browser – which is from result point of view similar to what the Bean Browser does, but which in certain cases is much easier for you to build up.

All three concepts are not linked in any way. If you are just interested in manipulating the Bean Browser hierarchy: skip the “Tool Extensions” and the “Expression Browser” - and vice versa.

Tool Extensions

The CaptainCasa Enterprise Client toolset can be extended by adding own user interface components:



You see that in the tool area on the right there are not only the three default tools available (bean browser, code generator, resources), but also two additional tools (bean page generation, wsdl import).

There is a certain, thin interface that allows you to add own tools in a simple way. And there are certain conventions that we ask you to follow in order to run each tool within its own environment when being started as part of the toolset.

Registration of Tool Extensions

First have a look into the web tool's web application. The layout editor is an Enterprise Client application – so there is a straight web application behind.

```
/webapps
    /editor
        /eclnt
        /eclntjsfserver
        /editor
        /tools
        /WEB-INF
        /META-INF
        ...
```

Inside the web application there is a tools directory. Each tool needs to add...:

- A subdirectory “toolxyz” into the tools directory.
- A corresponding “.jar” file “tool_toolxyz” into the WEB-INF/lib directory.

The directory structure now looks like:

```
/webapps
    /editor
        /eclnt
```

```
        /eclntjsfserver
        /editor
        /tools
            /toolxyz
        /WEB-INF
            /lib
                /tool_toolxyz.jar
        /META-INF
        ...
```

The subdirectory “toolxyz” looks the following way:

```
/webapps
    /editor
        /tools
            /toolxyz
                tool.xml      - tool registration
                page1.jsp     - jsp pages of tool
                page2.jsp
                page3.jsp
                /dir1         - additional directories of the tool
                /dir2
    /WEB-INF
        /lib
            tool_toolxyz.jar - code of the tool
```

The tool typically provides one or more pages (the one to be added into the tool section of the toolset). And there may be additional subdirectories for any purpose that is specific to the tool.

There is one important file that needs to exist: the “tool.xml” file. This one tells the CaptainCasa toolset which pages to include in which way:

```
<tool>
    <page text="XYZ Tool"
          page="/tools/toolxyz/page1.jsp"
          class="com.company.toolxyz.Page1UI"/>
    <page text="..."
          page="..."
          class="..."/>
    ...
    ...
</tool>
```

You can add any number of pages that should be integrated into the right tool section of the CaptainCasa Enterprise Client toolset. Each page definition consists out of the following information:

- text: text to be shown in the tab-line of the toolset
- page: link to jsp-page to be integrated
- class: class that is associated with the page.

Writing Pages to be used as Tool Extensions

Pages that you add to the CaptainCasa Enterprise Client toolset are “just normal” Enterprise Client pages, which are embedded into the tool's workplace. The following rules need to be followed so that your tool page is properly integrated into the tool's workplace processing.

Dispatcher “#{t}”

The dispatcher that is used within the toolset is an extension of the normal WorkpageDispatcher-class. If you are not familiar with workplace concepts then please

read details in the chapter “Workplace Management” of the Developers' Guide.

The dispatcher is registered via faces-config.xml using the name “t”, i.e. the expression that is used for accessing the dispatcher is “#{t}”.

You need to follow these conventions, so that your managed beans are running in the context of the dispatcher. This means:

- The managed beans serving your tool pages either are plain beans or they are extensions of DefaultDispatcher-class or Workpage-Dispatcher class.
- All expressions need to start with “#{t}”, so that they are reachable through the tool's dispatcher.
- The tool's dispatcher will bind the name following the “t” with regards to the tool.xml definitions, in which you pass per jsp-page a corresponding managed bean class.

Hints for developing

When developing your tool extension's user interface you just need to define a normal dispatcher inside the package of your managed bean implementations, and reference the dispatcher with “t” from faces-config.xml.

Later on, when your tool runs inside the toolset, then this dispatcher will not be used any longer, but the toolset's dispatcher will be used.

Interface IToolUI

The managed bean class that is associated with a tool page is directly started (corresponding object is created) when the CaptainCasa Enterprise Client toolset is started.

The class may implement the interface IToolUI:

```
package org.eclnt.editor.tools;


public interface IToolUI
{
    public void prepare(ProjectInfo projectInfo);
}
```

Each time a project is selected within the project area of the toolset then the corresponding “prepare” function is called.

Class ProjectInfo

Retrieving Information about the Project

The parameter “projectInfo” is of type “ProjectInfo” and contains a couple of get-methods that allow to you to access the tool's resources:

```
public class ProjectInfo
{
    ...
    ...
    public String getName() { ... }
    public String getWebcontentdirectory() { ... }
    public String getWebcontentdeploydirectory() { ... }
    public String getWebcontextroot() { ... }
    public String getWebhostport() { ... }
    public String getJavasourcedirectory() { ... }
```

```
    public boolean getSupportsHotDeploy() { ... }
    public String getHotDeployFileName() { ... }
    ...
    ...
}
```

Consequently you can gather for any information within the current project's directory structure.

Triggering Reload of Project

Your tool can initiate a reloading of the project by calling the reloadProject()-method. You may optionally pass a reference object (“trigger”) so that you can identify later by whom the reload was triggered:

```
public class ProjectInfo
{
    ...
    ...
    public void reloadProject(Object trigger) { ... }

    public void addProjectReloadListener(IProjectReloadListener prl)
    { ...}
    public void removeProjectReloadListener(IProjectReloadListener prl)
    { ... }
}
```

There's a listener mechanism that allows to you to listen to other tools that initiate a reload. The interface “IProjectReloadListener” looks as follows:

```
public interface IProjectReloadListener
{
    public void reactOnReload(ProjectInfo pi, Object trigger);
}
```

The “reload” typically is a significant step when the user uses the toolset: it means that the development directories of the project are copied into the runtime and that the corresponding runtime web application is re-started.

Triggering Eclipse Reload

When supporting Eclipse as development environment then there is a nice way to automatically synchronize the Eclipse project with the file system.

```
public class ProjectInfo
{
    ...
    ...
    public void triggerSrcUpdateInEclipse() { ... }
    ...
    ...
}
```

By calling this method a “signal file” is written into the design time project. This signal file is scanned by the CaptainCasa Eclipse Plug-in, which then initiates a synchronization with the file system.

Every time you write a file or you update a file within the user's project you should call the “triggerSrcUpdateInEclipse()”-function.

Drag & Drop into the Editor

You can use the normal CaptainCasa drag and drop functions in order to drop expressions

from your tool page into the layout editor.

The DRAGSEND-string that you need to use is: “expression:<expression>”, e.g. “expression:#{d.xxx.yyy}”. The value of the expression will be transferred into the attribute values of component – just as you know it from using the “Bean Browser”-tool.

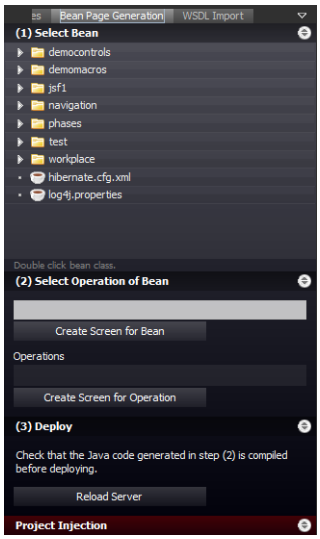
Project Injection

In many cases a tool that you provide is part of a framework that you want to embed into the creation of JSP pages. The framework not only consists of tools but also of libraries that you need to add into the project as runtime part. - This process of adding libraries (and other resources) is called “Project Injection”.

Project injection can be simply done as part of you tool's pages:

- You add the libraries and resources into the /tools/toolxyz folder, e.g. into a folder “/tools/toolxyz/injection”.
- You provide a button within your tool's page that allows the user to start the injection.

All add-on tools that are provided by CaptainCasa follow the convention that the Project Injection is done in the bottom area of a tool's page:



The functions having to do with project injection are placed into a FOLDABLE, that is closed by default. Once the user opens the FOLDABLE component then the corresponding functions will show up in order to start the injection.

There is a special FOLDABLE-STYLEVARIANT that you can use (“INJECTION”).

Styling Issues

The CaptainCasa Enterprise Client toolset uses two styles:

- cceditor
- cceditorlight

Please have a look into the style definition(s) and use the corresponding style variants for styling the page that you integrate into the tool environment.

Tool Extensions – Eclipse Plug-ins

When adding an extension that somehow simplifies the creation of your application, then there are two possibilities in general:

- You write an extension to the CaptainCasa toolset environment.
- You use the extensibility of your development environment (e.g. Eclipse) in order to add extensions there.

The CaptainCasa toolset is not a “multi-purpose” development environment, but is a set of tools that explicitly support the creation of user interfaces.

Consequently we recommend to only add these tools that really have to do with UI processing at all – e.g. from which you can drag & drop information into the Layout Editor.

For other purposes, that are not UI related, we clearly recommend to first check the usage of plug-in technology that is provided by the development environment.

Preconfigured Tool Extensions

In some cases you want to show a certain page of your real application within the Layout Editor's tool area. In order to do so, there is a predefined, simplified way that is described within this chapter.

Overview

The idea is quite simple: within the editor's tool area a sub-page is embedded, that directly talks to the application of the currently selected project.

Background: the CaptainCasa client is able (due to its SUBPAGE component) to embed a client on its own into a page. You may compare with the IFRAME tag in the HTML world: you can design pages, in which a certain part of the page is an IFRAME pointing to an other server than the rest of the page.

Exactly this is done in the area of the preconfigured tool extensions: within the page that talks to the editor application a sub-page is embedded, that directly talks to the application of the currently selected project.

Configuration

Pages within your Application

Within your application you may define a page with the exact name...

```
/eclnteditorextension/ext1.jsp
```

This is the page to be shown within the editor later on. You may test the page before embedding into the editor by opening the page in the browser via "http://localhost:50000/project/ecnteditorextension.ext1.ccapplet".

Registration within the Layout Editor

Within the application of the Layout Editor you need now to tell, that you want to use the pre-configured plugin.

This is done by editing /editor/tools/ccappextension/tool.xml. By default there is only a template file available within the same directory – just copy the file to "tool.xml" and then edit the content:

```
<tool>
    <page text="Your Extension"
          page="/tools/ccappextension/ext1.jsp"
          class="org.eclnt.editor.Ext1UI"
          bindingname="Ext1UI"/>
</tool>
```

The only thing you have to do is to exchange the text. This is the text that will be later on displayed within the tool section of the Layout Editor. The other parameters need to be kept as defined by the template.

That's it!

There's nothing more to do. You now can re-start the Layout Editor an will see your page being part of the tool area on the right.

There are three preconfigure-able Extensions

If you have more than one page that you want to integrate into the layout editor, use the same mechanism as described before, but use index “2” (or “3”) instead of “1”:

The tool.xml file in this case looks the following way...

```
<tool>
    <page text="Your first Extension"
          page="/tools/ccappextension/ext1.jsp"
          class="org.eclnt.editor.Ext1UI"
          bindingname="Ext1UI"/>
    <page text="Your second Extension"
          page="/tools/ccappextension/ext2.jsp"
          class="org.eclnt.editor.Ext2UI"
          bindingname="Ext2UI"/>
</tool>
```

...and the page that you need to implement within your project is:

```
/eclnteditorextension/ext2.jsp
```

Parameters that are passed into your Extension

When your page is loaded into the Layout Editor environment then a couple of parameters are appended to the corresponding URL – which you can access at runtime:

You can access these parameters – e.g. in the constructor of your extension class – in the following way:

```
    HttpServletRequest req = HttpSessionAccess.getCurrentRequest();
    String projectName = req.getParameter(“projectname”);
```

The parameters that are passed are:

- “projectname” - id of the current project
- “projectdirectory” - the directory in which the project is located
- “webcontentdirectory” - the directory in which the project's webcontent is stored
- “sourcedirectory” - the directory in which the project's Java sources are stored
- “editorinstance” - an id that is uniquely created for each editor instance and which is stable throughout the licecycle of this editor instance.

Session Considerations

The page that is running as editor extension is started in a session of its own. The session is opened when the user opens the extension (i.e. selects corresponding tab within the tool environment). When switching to another editor extension and then back into the extension again, then the page is created “from the scratch” - and a new session is opened by the page.

There are two issues which you may do as consequence:

- Embed a SESSIONCLOSER component into your page, so that the session is automatically closed when the user leaves the extension. As consequence the session is closed immediately and does not have to wait for a time out.
- You may use the “editorinstance”-id in order to store certain information across several sessions. Example: in your page you may logon to a certain system / database in order to retrieve certain information. You now may store the logon data with the editinstance-id so that the user does not have to re-logon when leaving the page and getting back to the page again.

Message Communication “through the Screen”

The CaptainCasa client provides a nice feature: you can send messages to the client through a server side API – using class “AsynchMessageBus”. (Well, actually “send message to the client” means, that the corresponding messages are included in the response of the current request processing that is sent back to the client...)

A message has a method name and parameters: “method(param1,param2,...)”.

On client side you can define MESSAGELISTENER components that listen to messages being sent – and that themselves call an action listener once receiving a message with a certain method-name.

This messaging was introduced to provide a mechanism allowing to communicate between various parts of applications “through the client”. And this feature is exactly used by the Layout Editor: the Layout Editor sends out certain messages that can be processed by other applications' screens.

The message format is:

- cceditor(load,currentPageName) – when a file is shown in the Layout Editor

This means: in the page that you embed into the pre.configured tool area, you may add a MESSAGELISTENER component and as consequence get notified every time a page is loaded into the editor. The example-jsp might contain the following:

```
<t:beanprocessing id="g_1">
      <t:messagelistener id="g_2"
            actionListener="#{d.EditorExtension1UI.onMessageAction}"
            commandfilter="cceditor"
            delay="#{d.EditorExtension1UI.onMessageAction}" />
</t:beanprocessing>
```

The attribute COMMANDFILTER ensures that only messages with the name “cceditor” are registered. When a message of this type is received on client side then the action listener will be called.

In the action listener of your application the code might look like:

```
public void onMessageAction(ActionEvent event) throws Exception
{
    if (event instanceof BaseActionEventMessage)
    {
        String page = ((BaseActionEventMessage) event).getParams()[2];
    }
}
```

The first parameter is the name of the message “cceditor”, the second parameter is the function that is currently executed (“load”) the third parameter is the name of the page – which is checked in this example.

In short: your plugin gets notified when the layout is opened within the working area of the editor. As consequence you may now use e.g. naming conventions in order to display these pieces of information within your plugin that are adequate to the page that is currently edited.

Drag & Drop Communication

There's a niche possibility to allow layout-XML to be dropped from your plugin into the layout editor preview area. It is based on the normal drag&drop protocol that is used within the CaptainCasa client – in which the sender of drag&drop has to specify the information as “type:value” String, whereas the receiver defines a number of types, that

are valid to be dropped (“type1;type2”).

The Layout Editor preview supports two types:

- “cceditor_newcol” - for adding column-oriented XML
- “cceditor_newrow” - for adding row-oriented XML

Example: your plugin may provide components with drag information “cceditor_newcol:<t:field text='Hello' width='100'/>” or “cceditor_newrow:<t:row><t:field text='Hello' width='100'/></t:row>”.

In this case this information is directly embedded into the layout editor preview when dragging & dropping it from you plugin.

Bean Browser Adaptations

The bean browser is the central tool “on the right” that show the tree of reachable managed bean objects. It is commonly used to drag & drop JSF expressions into attributes of user interface components.

The default bean browser first scans the “faces-config.xml” for finding the root nodes of your hierarchy of managed beans. It then introspects the classes and drills down following the set/get property methods.

Adaptation Process

The bean browser introspects the classes. While introspecting it checks with every class if it supports a special method:

```
public static DynamicIntrospectionInfo introspectDynamically(List
references, List<String> pathList);
```

If the class implements this static method then the bean browser calls the method.

Static Method

Why is this a static method? - Because the bean browser must not create any object instances. Typically, in order to create valid instances of your class, you need to provide a certain environment. Maybe your objects need access to the database, and consequently will fail when trying to be created from outside. Remember: the bean browser runs within the context of the Layout Editor – a completely different application, which in principal has nothing to do with your project's application at all.

Please keep this in mind: the bean browser opens up a classloader for the project, that looks into the WEB-INF/classes and WEB/INF/lib directories of your web application. In case you use hot-deploy management then it will also look into eclnthotdeploy/classes and eclnthotdeploy/lib. It does not run your application, i.e. it does not stat it as web application – it just introspects it from outside.

DynamicIntrospectionInfo

Let's start with the result of the method “introspectDynamically”. The class “DynamicIntrospectionInfo” is part of the interface “IDynamicIntrospectionSupported”:

```
public interface IDynamicIntrospectionSupported
{
    public static class DynamicIntrospectionInfo
    {
        boolean m_continueWithNormalIntrospection = false;
        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 void setContinueWithNormalIntrospection(boolean value)
{ m_continueWithNormalIntrospection = value; }
        public boolean getContinueWithNormalIntrospection() { return
m_continueWithNormalIntrospection; }
    }

    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; }
    }

}
```

As you can see from the interface's source code the result object contains a list of dynamic properties and a list of dynamic methods – the ones that you add based on your knowledge about the object structure.

There is also a flag “continueWithNormalIntrospection”: if this is defined to be false, then your result object's information is the only that is displayed within the bean browser. Otherwise, if defined to be true, then the normal introspection is continued just as normal – the bean browser will show a mixture out of properties that you dynamically added and properties that are available due to introspection.

Each property info can be associated with a list of objects - the references. Here you can add any information that you want to store with the property information. Background: when the user opens a property that you dynamically created in order to see the contained sub properties of this object, then the introspection is repeated just as normal: the list of references is passed to the next introspection layer. - In other words: you may attach information that is required on the next introspection layer, when the user is continuing the drill down in the bean browser.

Input Parameters

Now, it is clearer what the input parameters are used for:

- The “references” parameter transfers the information of the previous property layer.
- The “pathList” is a list of strings representing the expression names: starting with the expression of the root expression, down to the node that is just introspected.

Expression Explorer

The expression explorer is a tool “on the right side of the editor” as well. From what it does it is very similar to the bean browser: it allows to show an hierarchy of expressions that the user can drag and drop from.

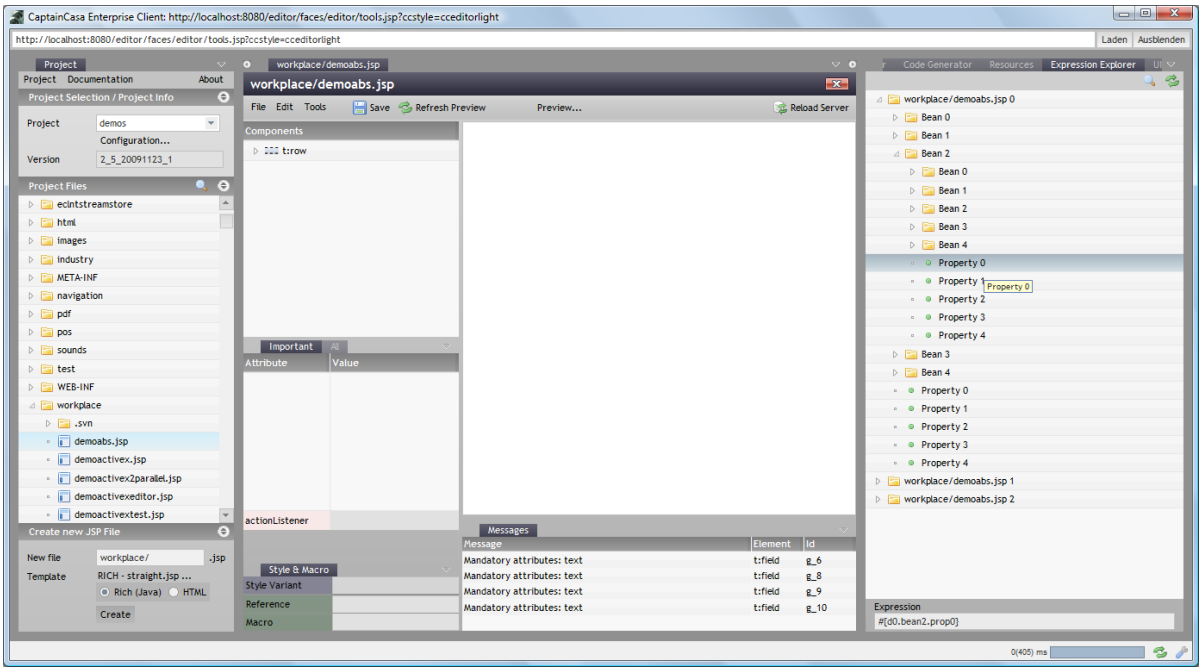
The main differences to the bean browser are:

- The Expression Explorer gets notified about the page that is currently shown within the Layout Editor – as consequence it can react accordingly and only show these items that are reasonable to be used in the page. - The rules, what expressions are usable for what page are completely part of your implementation.
- The Expression Explorer is an hierarchy that you completely define on your own. Whether you use bean introspection or whether you use any internal meta data that is available as part of your application: the Expression Explorer is not interested in. As a result, the interface of the Expression Explorer does have nothing to do with bean introspection but is a pure hierarchy-interface.

For using the Expression Explorer you need to implement the logical part of the Expression Explorer by extending two predefined classes. The core aspect of these classes is to define the hierarchy of expressions for a page and make it available for the Expression Explorer.

Example

The example that will show how to implement the logical part of the Expression Explorer looks as follows:

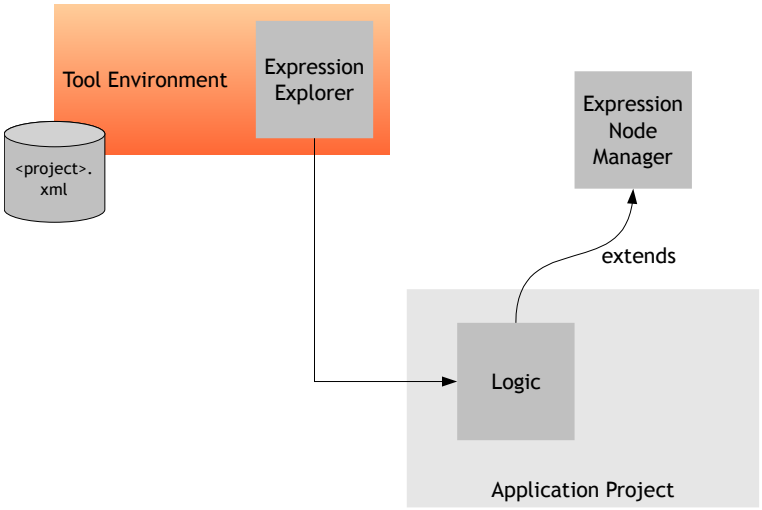


Of course the hierarchy on the right does not reflect a proper expression hierarchy – but this is by intention! What and how you define the expression hierarchy: this is completely left to you.

Concepts

Logic implemented as Extension of ExpressionNodeManager

The Expression Explorer is a certain user interface implementation that is part of the CaptainCasa tool environment. It links to a logical part, which is an extension of class “ExpressionNodeManager”. This class (at runtime: object) is responsible for building the hierarchy.



Logic is registered in CaptainCasa Project File

The name of the class providing the logic is registered within the CaptainCasa project file. At runtime of the tool environment a corresponding object is created by the tool environment.

Pay Attention: ClassLoading Issues

The logic implementation is part of your normal application project. You do not have to add it into the editor's libraries, but the tool environment will access via an extra classloader your project classes.

This is very important to point out: the tool environment creates an own class loader, using by default the WEB-INF/classes and WEB-INF/lib directories of your project. (If using hot deployment then also the corresponding classes are used.) You need to be aware of, that the runtime environment of your project is somehow different to your normal application's runtime environment, which is normally loaded by the web application loader, and which my execute certain initializations when starting up.

Example

The implementation of the logic is quite simple, because only a dummy hierarchy is built up in this example:

```
package tools;

import org.eclnt.editor.tools.ExpressionNode;
import org.eclnt.editor.tools.ExpressionNodeManager;
import org.eclnt.editor.tools.ProjectInfo;

public class DemoExpressionNodeManager extends ExpressionNodeManager
{
```

```
    DemoExpressionNodeManager m_this = this;

    public class MyExpressionNode extends ExpressionNode
    {
        String i_text;
        int i_nodeType = ExpressionNode.NODETYPE_EXPRESSIONFOLDER;
        String i_expression;
        public MyExpressionNode(ExpressionNodeManager manager,ExpressionNode parent,String
text, String expression, int nodeType)
        {
            super(manager, parent);
            i_text = text;
            i_expression = expression;
            i_nodeType = nodeType;
        }
        @Override
        public String getExpression() { return i_expression; }
        @Override
        public int getNodeType() { return i_nodeType; }
        @Override
        public String getText() { return i_text; }
        @Override
        public void loadChildNodes()
        {
            m_childNodes.clear();
            for (int i=0; i<5; i++)
            {
                MyExpressionNode en = new MyExpressionNode(m_this,
                                                           this,
                                                           "Bean " + i,
                                                           getExpression().replace("}",".bea
n"+i+"}"),
                                                           ExpressionNode.NODETYPE_EXPRESSIO
NFOLDER);
                m_childNodes.add(en);
            }
            for (int i=0; i<5; i++)
            {
                MyExpressionNode en = new MyExpressionNode(m_this,
                                                           this,
                                                           "Property " + i,
                                                           getExpression().replace("}",".pro
p"+i+"}"),
                                                           ExpressionNode.NODETYPE_EXPRESSIO
NLEAVE);
                m_childNodes.add(en);
            }
        }
    }


    @Override
    public void loadRootNodesForPage(ProjectInfo project, String pageName)
    {
        m_rootNodes.clear();
        for (int i=0; i<3; i++)
        {
            MyExpressionNode en = new MyExpressionNode(this,null,pageName + " " +
i,"#{d"+i+"}",ExpressionNode.NODETYPE_EXPRESSIONFOLDER);
            m_rootNodes.add(en);
        }
    }

    public void prepare(ProjectInfo projectInfo)
    {
    }

}
```

- The tool environment creates an instance of the logic class DemoExpressionNodeManager at the point of time when a project is selected by the user.
- When opening or showing a page within the Layout Editor then the method “loadRootNodesForPage” is called. The purpose of this method is to fill the “m_nodes” property (implemented inthe super-class ExpressionNodeManager)
- “m_nodes” is a list of node informations, each one being an extension of class “ExpressionNode”. In the example an inner class is used for implementing this extended class, you could of course also use a normal class.
- Each node provides the information about its text, its expression and its type.

- When the user toggles on node (i.e. opens its sub nodes) then the method “loadChildNodes” in the corresponding node is called. The processing now is very similar to the processing of “loadRottNodesForPage” - the next level of node list is built up.

Result

The Expression Explorer is a nice way to provide a hierarchy of expression with little coding effort. In many cases you have a certain “given hierarchy” which e.g. is built up by conventions or by accessing other met data.

Especially in environments when working with dynamic managed beans (hash map based beans) the bean introspection of the Bean Browser is not really of great help – this is when to use the Expression Explorer.