CaptainCasa Enterprise Client

Tool Extensions

BETA 2.2

tainCasa Enterprise Client

This is a feature of CaptainCasa Enterprise Client. The current status is “beta”. We appreciate your feed back!

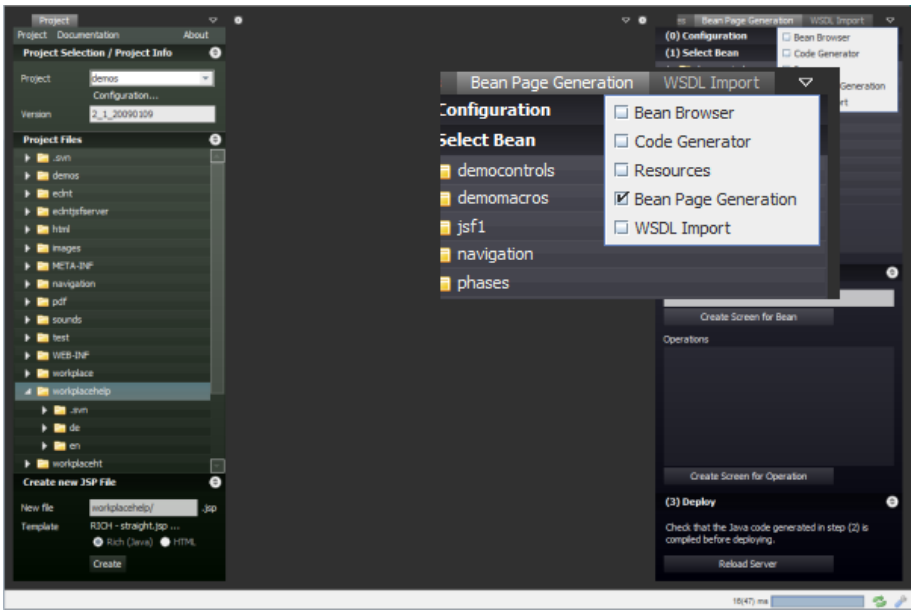
Please let us know, when you become active in the area of “Tool Extensions” - for receiving explicit support during the beta phase.

Table of Contents

Tool Extensions....3
- Registration of Tool Extensions....3
- Writing Pages to be used as Tool Extensions....4
  - Dispatcher “#{t}”....4
  - Hints for developing....5
- Interface IToolUI....5
- Class ProjectInfo....5
  - Retrieving Information about the Project....5
  - Triggering Reload of Project....6
  - Triggering Eclipse Reload....6
- Drag & Drop into the Editor....6
- Project Injection....7
- Styling Issues....7
- Tool Extensions – Eclipse Plug-ins....8

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.

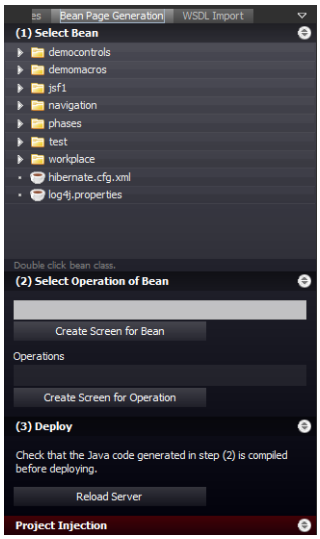
Project Injection

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.