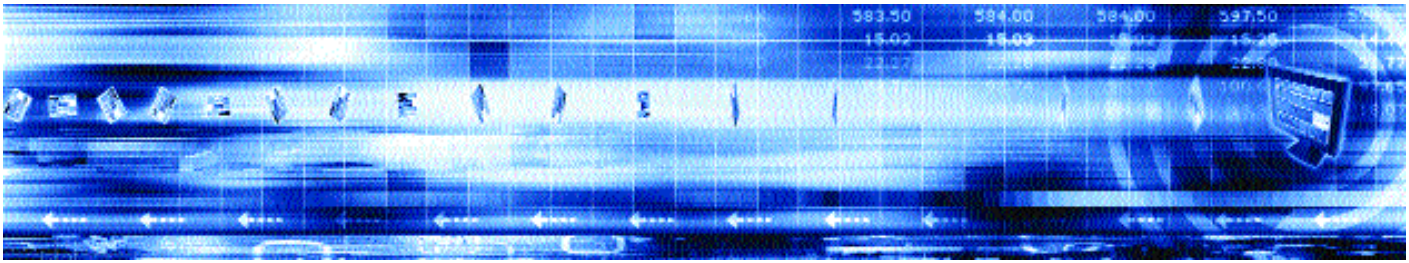


Caplin Enterprise Management Console 4.0

Customizing The Console

May 2005

1 Preface

1.1 What this document contains

This document describes how to customize Caplin's Enterprise Management Console.

1.2 Who should read this document

This document is intended for people who want to customize the Enterprise Management console to monitor Caplin's real-time data architecture.

1.3 Typographical conventions

This document uses the following typographical conventions to identify particular elements within the text.

Type	Use
Arial Bold	Other sections and chapters within this document.
Arial Italic	Parameter names and other variables.
Times Italic	File names, folders and directories.
`Courier`	Program output and code examples.
❖	Information bullet point
■	Instruction

1.4 Feedback

Customer feedback can only improve the quality of Caplin product documentation, and we would welcome any comments, criticisms or suggestions you may have regarding this document.

Please email your thoughts to documentation@caplin.com.

2 Overview

The Caplin Enterprise Console has been designed to allow customization of its displays and in particular the inclusion of user defined views. Configuration of the console is defined via a set of XML files. By implementing certain interfaces and writing a small amount of XML a user can install their own views into the console.

User views are hosted as tabbed pages within the console, each tab can have its own name, icon, tooltip, menu options, toolbar options and help text. Views are loaded dynamically and on demand for performance reasons. To add a new view to the console the user must perform the following actions (described in more detail in the rest of this document):

1 Create a **JPanel** derived class that implements the View interface. View Interface:

```
void init(Console console, Properties properties)
void save()
Properties getProperties()
List getActions()
Console getConsole()
void processMessage(String messageId, Map messageData)
```

This view should use the Console object passed into its **init()** method to obtain a JMXConnection. The actions (Swing Action objects) exposed via the **getActions()** method can be used to populate menu and toolbar options. Properties, to configure the view, can be loaded and saved via the **init()** and **getProperties()** methods.

2 Create an xml file named *view_<NameOfView>.xml* that defined the view, its classname, tabname, icon, menus, toolbar, help topic etc.

3 Edit one or more of the *console_<datasourceType>.xml* files to reference this view, i.e. specify that a console of the given type should load and display this view.

4 Link in the help text.

3 Implementing Required Java Classes

There is one interface that must be implemented in order to create a new view. There is also an optional interface which is described below. The required interface is *com.caplin.view.View.*

More details on this interface can be found by looking at the JavaDoc that is supplied with the Enterprise Console. The interface comprises of the following methods:

void init(Console console, Properties properties);

void save();

Properties getProperties();

List getActions();

Console getConsole();

void processMessage(String messageId, Map messageData);

The class that implements this interface should extend a suitable JComponent class such as **javax.swing.JPanel**. So for example the declaration of a class that implements a view for a component would have the following declaration:

```
public class ExampleView extends JPanel implements View
```

This class should be specified in the *view_viewName.xml* file. See the section **Creating a Help Guide** on page 13 for more details. In this file, the developer may specify a number of actions to be performed. Each of these actions need to be implemented in the Java code. The following section deals with this implementation.

3.1 Actions

Each `<Action Name="..."/>` tag in the *view_componentName.xml* file needs to correspond to an action in the Java code. This should be achieved by constructing an inner class within the class that implements **com.caplin.view.View**. This inner class should extend **javax.swing.AbstractAction**.

So for example, if a developer creates a *view_example.xml* file with the action element `<Action Name="SayHello"/>` then the Java code in the inner class could be similar to the following:

```
class RefreshAction extends AbstractAction
{
      public RefreshAction()
      {
            this.putValue(Action.ACTION_COMMAND_KEY, "SayHello");
            this.putValue(Action.NAME, "Say Hi");
            this.putValue(Action.SHORT_DESCRIPTION,"Say hi to the user");
            this.putValue(Action.MNEMONIC_KEY, new
            Integer(java.awt.event.KeyEvent.VK_F5));
            this.putValue(Action.SMALL_ICON,
            ResourceManager.getInstance().getImage(
            "resources/images/Refresh16.gif"));
      }
      public void actionPerformed( ActionEvent e )
      {
            //perform processing to achieve goals of action
      }
}
```

The first line of the constructor tells the Enterprise Console to call this class to perform the action described in the `<Action ../>` element. Notice that the two strings are the same. This is very important as otherwise the Enterprise Console will not be able to tell what class should be used to perform the task and will output an error to the command prompt and act as if the user never asked for the action to be performed.

The second line tells the Enterprise Console what text to display on the drop down menu.

The third line tells the Enterprise Console what text to display in the tool tip for the action. The tool tip will be displayed for both a drop down menu option and a toolbar button.

The fourth line specifies a keyboard shortcut key to use for this action. The Enterprise Console will listen for this keystroke and will call the class when it receives it.

The fifth line specifies an icon to be used for this action. If this action is to be added to the toolbar then the icon will act as a button. If the action is to be added to a drop down menu then the icon will be displayed next to the text as specified in the second line.

The code in **actionPerformed(ActionEvent e)** should perform the action. The Enterprise Console will call this method when the user requests the action corresponding to this class.

3.2 Implementing ConsoleListener

The interface **com.caplin.console.ConsoleListener** should be implemented if the view needs to perform processing when the Enterprise console starts up or shut downs. The interface should also be implemented if the view needs to be notified when another view is updated or the JMX connection changes.

A detailed description of each method to be implemented is available in the JavaDoc which is supplied as standard with the Enterprise Console. For completeness the methods declarations are also supplied here. They are as follows:

void viewChanged(View view);

void viewLoaded(View view);

void connectionStateChanged(boolean connected);

void closing();

3.3 Inter-view communication

Since each view class is loaded dynamically and on demand (via the class name in the associated *view_<ViewName>.xml* file) you should not directly access one view from another.

Inter-view communication is handled via the following three methods, two on the Console and one on the View:

Console Methods:

```
      /**
      * Shows the given view, the view must be defined in the console xml
configuration file
      * @param toViewClassName
      * @return true if view shown
      */
      boolean showView(String toViewClassName);
```

```
      /**
      * Posts a message to the given view, used for inter-view communication
      * @param toViewClassName the class name of the view to receive the
message
      * @param messageId a String identifying the message
      * @param messageData a Map of name, value pairs representing messages
data
      */
      void postMessage(String toViewClassName, String messageId, Map
messageData);
```

View Method:

```
/**
      * Processes a message from another view, used for inter-view
      communication
      * @param messageId a String identifying the message
      * @param messageData a Map of name, value pairs representing messages
data
       */
      void processMessage(String messageId, Map messageData);
```

This mechanism is used throughout the existing Caplin supplied views to link items to the associated MBean tab in the Explorer view, the following code snippet shows how to perform this link from a view class:

```
protected void onShowInExplorer( ObjectName objName )
{
      getConsole().showView("com.caplin.view.explorer.ExplorerView");
      HashMap dataMap = new HashMap();
      dataMap.put("MBeanName", objName);
      getConsole().postMessage("com.caplin.view.explorer.ExplorerView",
      "ShowMBean", dataMap);
}
```

4 Writing the Required XML

If a custom view is to be created for a particular component (e.g. Liberator) then a developer will need to perform the following tasks in order to get the Enterprise Console to use the custom view.

4.1 View

A new file called *view_viewName.xml* will need to be created. This file defines what a view will look like. The file will take the following form.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE View SYSTEM "dtd/View.dtd">
<View>
      <Class>com.company.view.OverviewViewTab</Class>

      <Name>Overview</Name>
      <LongDescription>This tab gives an overview of the new
component...</LongDescription>

      <HelpTopic>overview.mainpage</HelpTopic>

      <Icon>resources/myIcon.gif</Icon>

      <Menu Name="Tools">
            <Action Name="PerformAction"/>
      </Menu>

      <ToolBar>
            <Action Name="PerformAction"/>
            <Divider/>
            <Action Name="PerformOtherAction"/>
      </ToolBar>
</View>
```

<Class> tag

The `<Class>...</Class>` element defines the class that will be used to display this view. For information on implementing this class please see the section on implementing the required Java classes. The Enterprise Console will call this class when a user double-clicks the component in the main page of the Enterprise Console. If it cannot find/load the class then it will output an

error message to the command prompt and fail to launch the component (i.e. it will be as if the user had performed no action).

<Name> tag

The `<Name>...</Name>` and `<LongDescription>...</LongDescription>` tags are used by the Enterprise Console to display information about this view to the user. The text entered between the `<Name>…</Name>` tags is displayed on the top of the tab used to display this view.

<LongDescription> tag

If the user hovers over the tab then the text between the `<LongDescription>...</LongDescription>` tags will be displayed as a tool tip. This will give the user more information about the view without having to open it.

<HelpTopic> tag

The `<HelpTopic>...</HelpTopic>` tags are used by the Enterprise Console to reference the proper section in the help guide for this view. A developer can easily add help for a custom component to the Enterprise Console. This is covered in more detail in the creating help section.

<Icon> tag

The `<Icon>...</Icon>` tags are used to specify the location of an image file that the Enterprise Console will display next to the name of the view as specified between `<Name>...</Name>`. This file can be a relative path or an absolute path. The base directory will be the directory from where the Enterprise Console was started.

<Menus> tag

The `<Menus>...</Menus>` tags are used by the Enterprise Console to display the different drop down menu options that are available for this view. In the example above, the Enterprise Console will create a drop down menu called Edit. The name of the operation displayed in the menu will be defined by the Java code. See the section **Implementing Required Java Classes** on page 6 for more information on this topic.

It is possible to have more than one set of `<Menus>...</Menus>` tags as the developer may wish to have several drop down menus. Each set of `<Menus>...</Menus>` tags should have its own `<Action … />` element(s) defined.

<Divider> tag

It is also possible to have dividers in drop down menus. So for example if you wanted to separate out different types of operations in a single menu, then the developer could add a `<Divider/>` element between two `<Action ... />` elements. This will cause the Enterprise Console to insert a horizontal line between the two operations in the drop down menu. The tag is used in the same was as it is in the `<ToolBar>...</ToolBar>` section.

<Toolbar> tag

The `<ToolBar>...</ToolBar>` tags are used by the Enterprise Console to display different buttons on the toolbar. The developer also needs to implement this action in the class specified for this view. More information can be found on this in **Implementing Required Java Classes** on page 6. The Enterprise Console will display the button on the toolbar and when the user

clicks the button the Enterprise Console will call the relevant Java code. As with the menu section, dividers can be added to separate different groups of buttons from each other. A small vertical line will be inserted between the two buttons by the Enterprise Console in this case.

4.2 Console

Firstly the developer needs to go to the correct XML file for the component. This file will be called console_componentName.xml (e.g. *console_liberator.xml*). This file tells the Enterprise Console what views to create for the component. The format of the file will be similar to the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ConsoleConfig SYSTEM "dtd/ConsoleConfig.dtd">
<ConsoleConfig>
     <Properties>
          <Property name="propsfile" value="./properties/view.props" />
          ...

     </Properties>
     <Views>
          <View Id="explorer"/>
          <View Id="view2"/>
     </Views>
</ConsoleConfig>
```

For every view defined in the `<Views>...</Views>` section, an accompanying file needs to be created. This is dealt with in more detail in the View section. The developer will need to add an entry in the `<Views>...</Views>` section for the Enterprise Console to use. The developer should also add any required properties that the new view will need in the `<Properties>...</Properties>` section of file. The Enterprise Console will create a *java.util.Properties* object from the `<Properties>...</Properties>` section of this file and pass the object to each view (see **Implementing Required Java Classes** on page 6 for further details).

5 Creating a Help Guide

As a final but important section on creating a customized view, it is important to create a help guide for the new view, that users can refer to in order to learn how to use the view or if they encounter problems. The Enterprise Console uses JavaHelp (http://java.sun.com/products/javahelp/) to provide an interactive help guide. This guide does not go into the details of how to create the various files required, instead it describes how to modify the help files that are distributed with the Enterprise Console to reference the help files that a developer needs to create.

In the view_customView.xml file the developer needs to specify a help section for the view. The Console will then use this to provide help to the user on the current view they are using.

The developer will also need to add entries to the following files:

- *resources/help/ParentHelpMap.jhm*
- *resources/help/ParentHelp.hs*

ParentHelpMap.jhm

In this file, the developer will need to add a link to the HelpSet file (*file.hs*) that should be created for the custom view.

The file will look similar to the following:

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE map
     PUBLIC "-//Sun Microsystems Inc.//DTD JavaHelp Map Version 1.0//EN"
          "http://java.sun.com/products/javahelp/map_1_0.dtd">

<map version="1.0">
     <mapID target="toplevelfolder" url="images/toplevel.gif" />
     <mapID target="pricemaster" url="PM4_help/IdeHelp.hs" />
     <mapID target="explorer.tab" url="html/explorer.html" />
</map>
```

The developer needs to add another `<mapID ... />` element to point to the new HelpSet file created for the custom view.

ParentHelp.hs

This file is quite large but the developer only needs to edit one section within the file.

Under the `<view...>...</view>` section there is an element called 'subhelpset', the developer needs to add the help set file that has been created for the custom view to this section. For example:

```
...
<subhelpset location="CMCHelp.hs" />
<subhelpset location="PM4_help/IdeHelp.hs" />
<subhelpset location="customView_help/IdeHelp.hs" />
...
```

The location of the help set file should be specified if it is not in the same directory as `ParentHelp.hs`. An example of this can be seen above. The help set file *IdeHelp.hs* is located in a directory under the current one called PM4_help.

CustomHelpTOC.xml

In the Table of Contents for the custom view, two `<tocitem>` elements should be present. One for an introduction section to the custom view and one providing the actual help for the custom view. An example of this can be seen in the file *CMCHelpToc.xml*. In this file the table of contents items are as follows:

```
<tocitem text="Introduction">
      <tocitem text="Management Console" target="top"/>
</tocitem>

<tocitem text="Tabs" target="tab.folder">
</tocitem>
```

The target values are defined in *CMCMap.jhm*.

The developer should not need to make any further changes to provide help for the custom view.

6 Appendix A

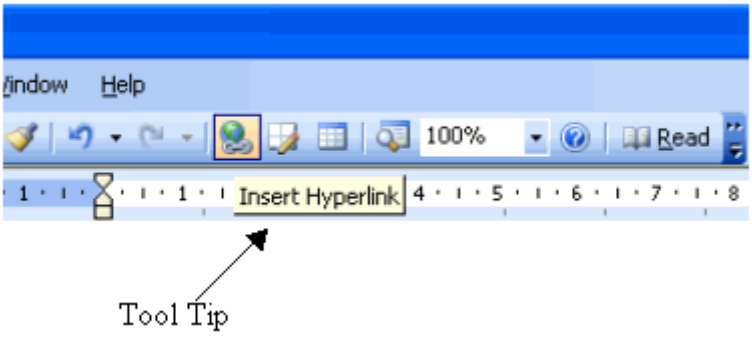
6.1 Glossary

View

A view is defined as a panel displayed in the Enterprise Console for an individual component. As default a view is implemented as a tab (e.g. the Explorer view is implemented as a tab).

Tool Tip

This is a common feature used in Java to provide extra information without cluttering up the user interface. If a user moves a mouse pointer over a component that has a tool tip and the user leaves the pointer rest over the component for a second of two then a text description of the component appears at the end of the mouse pointer. This is called a tool tip.



6.2 File locations

Java class files should be rooted in the same directory as the *emc.jar* file (i.e. if your class is *com.acme.MyView* then you should place the class file in the directory *com/acme* under the *EnterpriseManagementConsole* directory.

XML configuration files go in the *conf* directory and help files should be rooted in the *resources/help* directory (which must be created under the *EnterpriseManagementConsole* directory).

6.3 Example View class

```
package com.caplin.view;

import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.ArrayList;
import java.util.Enumeration;
import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.logging.Logger;

import javax.swing.AbstractAction;
import javax.swing.Action;
import javax.swing.JButton;
import javax.swing.JPanel;

import com.caplin.console.Console;
import com.caplin.console.ConsoleListener;

/**
 * A simple Test class that implements View:
 *
 * <pre>
 *
 *  - logs the calls to View methods
 *  - logs property values passed in
 *  - uses property to load/save background colour
 *  - exposes action to set background color
 *  - listens and logs console events
 *
 * </pre>
 *
 */
```

```
public class TestView extends JPanel implements View
{
      private static Logger log = Logger.getLogger(TestView.class.getName());

      private Console console;
      private List actions;
      private Color backColor;

      public TestView()
      {
            backColor = Color.WHITE; // set default value

            // load list of actions to be exposed
            actions = new ArrayList();
            actions.add(new SetBlueAction());

            // put something on panel
            JButton b = new JButton("make it white!");
            add(b);
            b.addActionListener(new ActionListener()
            {
                  public void actionPerformed( ActionEvent e )
                  {
                        backColor = Color.white;
                        setBackground(backColor);
                        console.setDirty(TestView.this, false);
                  }
            });
      }

      /**
       * Called by console to initialize this view
       */
      public void init( Console console, Properties properties )
      {
            this.console = console;
            log.info("---");
```

```
// add delay to simulate time taken to load state from server
delay();

// try to read color property
String colorProperty = (String)properties.get("Color");
if (colorProperty != null)
{
     if (colorProperty.equals("White"))
     {
          backColor = Color.WHITE;
     }
     else if (colorProperty.equals("Blue"))
     {
          backColor = Color.BLUE;
     }
}

setBackground(backColor);

// listen and log console events
console.addConsoleListener(new ConsoleListener()
{
     public void viewChanged( View view )
     {
          log.info("viewChanged");
     }

     public void viewLoaded( View view )
     {
          log.info("viewLoaded");
     }

     public void closing()
     {
          log.info("closing");
     }
```

```
                public void connectionStateChanged( boolean connected )
                {
                      log.info("connection changed");
                }
           });

           // log all properties passed in
           Enumeration en = properties.keys();
           while (en.hasMoreElements())
           {
                 String name = (String)en.nextElement();
                 String value = (String)properties.get(name);
                 log.fine(name + "= " + value);
           }

      }

      /**
       * Called by console to save this view (ie save data to JMX server) if it
has set its state to dirty
       */
      public void save()
      {
           log.info("---");
           delay();
      }

      /**
       * Helper for 1 second delay
       *
       */
```

```
      private void delay()
      {
            try
            {
                  Thread.sleep(1000);
            }
            catch (InterruptedException e1)
            {
                  e1.printStackTrace();
            }
      }

      /**
       * Called by console to retrieve the set of properties that must be saved
in the xml for this view
       */
      public Properties getProperties()
      {
            log.info("---");

            // save background color as property
            Properties properties = new Properties();

            if (backColor == Color.WHITE)
            {
                  properties.put("Color", "White");
            }
            else if (backColor == Color.BLUE)
            {
                  properties.put("Color", "Blue");
            }

            return properties;
      }
```

```
      /**
       * Called by console to retrieve the list of actions exposed by this
view.
       * The action names of these actions are used, via the view xml file, to
populate menu and toolbar options
       */
      public List getActions()
      {
           log.info("---");
           return actions;
      }

      /**
       * convenience method for passing console instance to other classes
       */
      public Console getConsole()
      {
           return console;
      }

      /**
       * Called by the console to send messages from other views, not
implemented here
       */
      public void processMessage( String messageId, Map messageData )
      {
      }

      /**
       * An example action to set the background color to blue
       *
       */
```

```
      class SetBlueAction extends AbstractAction
      {
            public SetBlueAction()
            {
                  this.putValue(Action.ACTION_COMMAND_KEY, "SetBlue");
                  // this is the action name used in the view xml to
                  // populate menu and toolbar options
                  this.putValue(Action.NAME, "Set Blue");
                  // menu item name
                  this.putValue(Action.SHORT_DESCRIPTION, "Set the background
blue");
                  // tooltip
                  this.putValue(Action.MNEMONIC_KEY, new
Integer(java.awt.event.KeyEvent.VK_B));
            }

            public void actionPerformed( ActionEvent e )
            {
                  backColor = Color.BLUE;
                  TestView.this.setBackground(backColor);
                  console.setDirty(TestView.this, true);
            }
      }

}
```

6.4 Example View xml configuration file

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE View SYSTEM "dtd/View.dtd">

<View>
      <Class>com.caplin.view.TestView</Class>
      <Name>A Test View</Name>
      <LongDescription>This is a test view</LongDescription>
      <HelpTopic>testview</HelpTopic>
      <Icon>/conf/folder_view.png</Icon>

      <Menus>
            <Menu Name="Edit">
                  <Action Name="SetBlue"/>
            </Menu>
      </Menus>

      <ToolBar>
            <Action Name="SetBlue"/>
      </ToolBar>
</View>
```

6.5 Example Console configuration file with test view added

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ConsoleConfig SYSTEM "dtd/ConsoleConfig.dtd">
<ConsoleConfig>
      <Properties>
      </Properties>
      <Views>
            <View Id="datasource_overview"/>
            <View Id="datasource_peers"/>
            <View Id="datasource_logging"/>
            <View Id="liberator_users"/>
            <View Id="liberator_objects"/>
            <View Id="explorer"/>
            <View Id="test"/>
      </Views>
</ConsoleConfig>
```

London

Triton Court
14 Finsbury Square
London EC2A 1BR
UK
Telephone: +44 20 7826 9600
Fax: +44 20 7826 9610

www.caplin.com

New York

111 5th Avenue
New York
NY 10003-1005
USA
Telephone: +1 212 994 1770
Fax: +1 212 994 1777

info@caplin.com