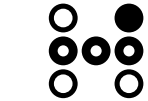
Essbase® Administration Services

Release 7.1

Developer's Guide



Hyperion Solutions Corporation

U.S. Patent Number: 5,359,724

Hyperion Solutions Corporation
1344 Crossman Avenue
Sunnyvale, CA 94089

Printed in the U.S.A.

Contents

Preface

Welcome to the Essbase Administration Services Developer's Guide. This preface discusses the following topics:

Purpose

This guide provides you with all the information that you need to extend Essbase Administration Services. It explains the Essbase Administration Services concepts, processes, procedures, formats, tasks, and examples that you need to customize the software.

Audience

This guide is for software developers who are responsible for extending Essbase Administration Services.

Document Structure

This document contains the following information:

Chapter 1, "Introduction", introduces the Essbase Administration Services plug-in components and shows you how you can use them. It also contains a list of packaged APIs for Essbase Administration Services and explains client and server concepts.

Chapter 2, "Writing Client Plug-ins," provides general guidelines for developing an Essbase Administration Services Java plug-in and describes how to add your Java plug-in to Essbase Administration Services.

Chapter 3, "Writing Server-side Command Listeners," explains how to write a command listener for the Essbase Administration Services mid-tier web server.

The Index contains a list of Hyperion Essbase Administration Services terms and their page references.

Where to Find Documentation

All Essbase Administration Services plug-in documentation is accessible from the following locations:

- The HTML Information Map is located at:

  EASPATH`/eas/doc_launcher.htm`

  Launch this file, and then provide the name of the computer where Administration Server is installed. Administration Server must be started when you launch the Information Map.

- The Hyperion Solutions Web site is located at `http://www.hyperion.com`.
- The Hyperion Download Center can be accessed from `http://hyperion.subscribenet.com` or from `http://www.hyperion.com`.

➤ To access documentation through the Hyperion Solutions Web site:

1. Log on to `http://www.hyperion.com`.
2. Select the **Support** link and type your username and password to log on.

   Note: New users must register to receive a username and password.

3. Click the **Hyperion Download Center** link and follow the on-screen instructions.

➤ To access documentation from the Hyperion Download Center:

1. Log on to `http://hyperion.subscribenet.com`.
2. In the **Login ID** and **Password** text boxes, enter your assigned login ID name and password.
3. In the **Language** list box, select the appropriate language and click **Login.**
4. If you are a member on multiple Hyperion Download Center accounts, select the account that you want to use for the current session.
5. Perform one of the following actions:
   - To access documentation online, from the **Product List**, select the appropriate product and follow the on-screen instructions.
   - To order printed documentation, from the **Information** section in the left frame, select **Order Printed Documentation**, then follow the on-screen instructions

➤ To order printed documentation if you do not have access to the Hyperion Download Center:

- In the United States, call Hyperion Solutions Customer Support at 877-901-4975.
- From outside the United States, including Canada, call Hyperion Solutions Customer Support at 203-703-3600. Clients who are not serviced by support from North America should call their local support centers.

Conventions

The following table shows the conventions that are used in this document:

Table i: Conventions Used in This Document

Item	Meaning
	Arrows indicate the beginning of a procedure consisting of sequential steps or one-step procedures.
Brackets [ ]	In examples, brackets indicate that the enclosed elements are optional.
Bold	Bold in procedural steps highlights major interface elements.
CAPITAL LETTERS	Capital letters denote commands and various IDs. (Example: CLEARBLOCK command)
Ctrl + 0	Keystroke combinations shown with the plus sign (+) indicate that you should press the first key and hold it while you press the next key. Do not type the + sign.
`Example text`	Courier font indicates that the material shown is a code or syntax example.
`Courier italics`	Courier italic text indicates a variable field in command syntax. Substitute a value in place of the variable shown in Courier italics.
ARBORPATH	When you see the environment variable *ARBORPATH* in italics, substitute the value of `ARBORPATH` from your site.
EASPATH	This environment variable is set to the directory path of the Administration Services installation. The default is `C:\Hyperion` on Windows platforms and `/home/hyperion` on UNIX platforms. When you see the environment variable *EASPATH* in italics, substitute the value of `EASPATH` from your site.
n, *x*	Italic *n* stands for a variable number; italic *x* can stand for a variable number or an alphabet. These variables are sometimes found in formulas.

Table i: Conventions Used in This Document (Continued)

Item	Meaning
Ellipses (...)	Ellipsis points indicate that text has been omitted from an example.
Mouse orientation	This document provides examples and procedures using a right-handed mouse. If you use a left-handed mouse, adjust the procedures accordingly.
Menu options	Options in menus are shown in the following format. Substitute the appropriate option names in the placeholders, as indicated. ***Menu name*** > ***Menu command*** > ***Extended menu command*** For example: **1.** Select **File** > **Desktop** > **Accounts**.

Additional Support

In addition to providing documentation and online help, Hyperion offers the following product information and support. For details on education, consulting, or support options, click the Services link on the Hyperion Web site at `http://www.hyperion.com`.

Education Services

Hyperion offers instructor-led training, custom training, and eTraining covering all Hyperion applications and technologies. Training is geared to administrators, end users, and information systems (IS) professionals.

Consulting Services

Experienced Hyperion consultants and partners implement software solutions tailored to clients' particular reporting, analysis, modeling, and planning requirements. Hyperion also offers specialized consulting packages, technical assessments, and integration solutions.

Technical Support

Hyperion provides enhanced electronic-based and telephone support to clients to resolve product issues quickly and accurately. This support is available for all Hyperion products at no additional cost to clients with current maintenance agreements.

Documentation Feedback

Hyperion strives to provide complete and accurate documentation. We value your opinions on this documentation and want to hear from you. Send us your comments by clicking the link for the Documentation Survey, which is located on the Information Map for your product.

Chapter 1

Introduction

This chapter includes the following topics:

About Essbase Administration Services

Essbase Administration Services is the new cross-platform framework for managing and maintaining Essbase. Administration Services provides a single point of access for viewing, managing, and maintaining Analytic Servers (formerly OLAP Servers), applications, and databases. This new product incorporates the functionality of Essbase Application Manager along with other new administrative features and with Essbase administration tools, such as MaxL.

Administration Services consists of two components:

- Administration Services Console (client tier)

  This component is a Java client console that enables administrators to manage the Essbase environment from a robust graphical user interface.

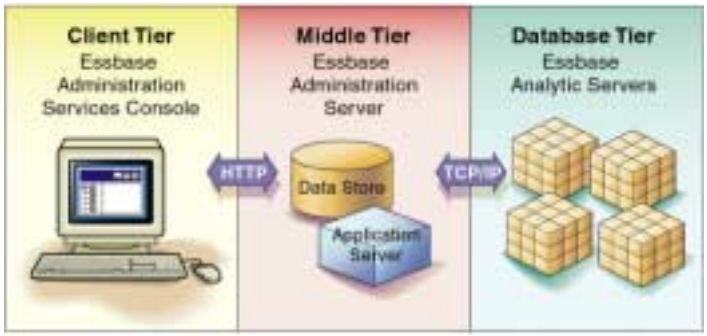
- Administration Server (middle tier)

  This component is a Java based web application server that communicates with both Administration Services Console and Essbase Analytic Servers. Administration Server maintains communication and session information for

each connection to Analytic Servers. Administration Server also stores documentation files so that console users can access the documentation without having to install it locally.

Administration Server serves as the middle tier between Administration Services Console and Essbase Analytic Servers, as shown in Figure 1.

Figure 1: Essbase Administration Services Architecture



About Java Plug-in Components

Essbase Administration Services Java plug-ins are installable components. They provide the following benefits to users:

- Enable the Essbase Administration Services development team to easily provide additional functionality to end users
- Allow other Hyperion internal development groups to easily integrate their products with Essbase Administration Services
- Enable partners and customers to easily integrate their processes into Essbase Administration Services
- Allow customers to accomplish more because they are not launching several applications at once

The following list describes how you can use Essbase Administration Services plug-ins:

- Customize the Essbase Administration Services Enterprise Tree
- Customize the Essbase Administration Services File > Open dialog box

- Customize the Essbase Administration Services File > New dialog box
- Customize the Essbase Administration Services File > Save As dialog box
- Change the Essbase Administration Services menus

For each of these tasks, there are a set of classes, interfaces, and methods that must be implemented by a plug-in author. There are also a set of guidelines to follow when implementing plug-ins.

For information about performing the preceding tasks, see "Writing Client Plug-ins" on page 19.

Requirements for Using Administration Services Java Plug-ins

The following list describes the requirements necessary to use Essbase Administration Services Java plug-in components:

- Java SDK Version 1.4.1_b06 or later
- Hyperion Essbase Release 7.1 or later
- Essbase Administration Services Release 7.1 or later

Prerequisite Knowledge

Developers using this guide must have the following prerequisite knowledge:

- XML (Extensible Markup Language)
- HTTP (Hypertext Transfer Protocol)
- Java 2
  - Introspection
    Introspection is a Java technique that Essbase Administration Services uses to interact and communicate with plug-in components.
  - Exception handling
  - Packaging of applications (`.jar` files)
- Swing

Swing is a graphical user interface (GUI) component kit, part of the Java Foundation Classes (JFC) integrated into Java 2 platform, Standard Edition (J2SE). Swing simplifies deployment of applications by providing a complete set of user-interface elements written entirely in the Java programming language. Swing components permit a customizable look and feel without relying on any specific windowing system.

Because Swing is incorporated in the Java 2 platform, there is no need to download or install it.

Framework Concepts

Packaged APIs for Administration Services

Administration Services consists of several packages. For detailed information about these packages, see the *Essbase Administration Services Java API Reference* for the packages and classes described in this guide.

Administration Services Java Packages

com.essbase.eas.ui.* (all packages)

com.essbase.eas.framework.* (all packages)

Example Classes

ConsoleTreeHandler

ConsoleMenuHandler

MiscellaneousHandler

NewDialogHandler

OpenDialogHandler

OptionsDialogHandle

About the Sample Code in this Guide

The code snippets and examples contained in this guide are intended to demonstrate how plug-ins interact with the Administration Services framework. They are intended to show how to get an aspect of the interaction to work and, in some cases, omit details that are not relevant to the topic being discussed. In addition, while the techniques shown will work, the Java techniques shown may in some cases not be the best implementation method when scaling up to a production quality product.

For example, in the section on context menu items, “Adding Context Menu Items To Tree Nodes” on page 27, the example creates new menu items and action listeners each time the getContextMenuItems() method is called; this might not be the best mechanism for handling this task. Please consult the appropriate Java resources (books, Web pages, documentation) for other techniques; in particular, when dealing with Swing objects, the Swing event model, and associating Swing event listeners to objects.

Chapter 2

Writing Client Plug-ins

This chapter explains how to write a plug-in for Administration Services Console. Plug-ins are the mechanism for extending the functionality of Administration Services Console. This chapter includes the following topics:

Preliminaries

We make the following user presumptions:

- You have some Java experience
- You have access to the *Essbase Administration Services Java API Reference*

- Since different developers use different build tools and environments, we do not discuss how to do anything for specific development environments. Rather, we describe the desired results, leaving it to the developer to know how to achieve these results with their specific development tools.

Note: For the purposes of this documentation, the terms “client framework”, “Administration Services Console”, “console”, “Administration Services client”, and simply, “the client” can generally be taken to refer to the client application.

Access Point for Plug-ins

The implementation of the Administration Services client is contained in the `eas_client.jar` and `framework_client.jar` files that are installed with Administration Services. Additional classes are found in the `eas_common.jar` and `framework_common.jar` files. The Analytic Services plug-in to Administration Services Console is contained in the `essbase_common.jar` and `essbase_client.jar` files.

Class Packages

Administration Services Console consists of several packages. The public classes in these packages are available to the implementor of plug-ins. In particular, the user interface, print, and mail-related classes. For detailed information about the packages and classes described in Table 1, see the *Essbase Administration Services Java API Reference*.

Table 1: Administration Services Console Class Packages

Package or Class Name	Description
com.essbase.eas.client.intf	The classes and interfaces that provide an interface to the console
com.essbase.eas.client.manager	The classes that provide “management” services for parts of the console; such as, LoginManager, CommandManager, ConsoleManager, and so on
com.essbase.eas.client.plugins	The classes that the client framework uses to install plug-ins, track plug-ins, and so on

Table 1: Administration Services Console Class Packages (Continued)

Package or Class Name	Description
com.essbase.eas.framework.client.defs.command	The client-specific classes related to sending commands to the mid-tier. As of Release 7.1, this consists only of the UICommandManager class.
com.essbase.eas.framework.client.defs.login	This is the default login dialog box provided by the console. It displays if no plug-in has registered a different login dialog or if any command is sent to the Administration Services mid-tier and a mid-tier server name has not been provided.
com.essbase.eas.framework.client.ui.filedlgs	Implements dialog boxes associated with a file menu. For example, New, Open, Save As
com.essbase.eas.ui	Another package with several user interface components used by the console and by the Analytic Services plug-in
com.essbase.eas.ui.ctable	An implementation of a standard extension to the JTable control
com.essbase.eas.ui.ctree	An implementation of an extension to the JTree control. This is the control that is used in the Enterprise Tree and in the custom views of the console.
com.essbase.eas.ui.editor	An implementation of a standard text editor with syntax highlighting. This control is used as the base class for the calculation script editor, maxL editor, and report script editor in the Analytic Services plug-in.
com.essbase.eas.ui.email	An implementation of some e-mail related classes. The framework provides a service for sending e-mail; this package contains the implementation of the service.
com.essbase.eas.ui.font	The classes that provide the font-related utility
com.essbase.eas.ui.print	The classes that provide the print-related utility
com.essbase.eas.ui.ptable	An extension to the JTable control for editing properties. This table provides extensive editing, sorting capabilities, and is used by many windows and dialogs in the Analytic Services plug-in.

Table 1: Administration Services Console Class Packages (Continued)

Package or Class Name	Description
com.essbase.eas.ui.ptree	An extension to the JTree control for editing tree-oriented properties. This tree provides extensive editing capabilities and is used by many windows and dialogs in the Analytic Services plug-in.
com.essbase.eas.ui.tree	The generic utility routines for working with JTree-based controls
com.essbase.eas.framework.defs	This package and the packages under it provide services for transferring commands from the mid-tier to the client, packaging/unpackaging data to be transferred, a logging mechanism, and so on
com.essbase.eas.i18n	The internationalization utility classes
com.essbase.eas.utils	Various utility classes spanning a range of uses: file utilities, compression, encryption, array utilities, and so on
com.essbase.eas.utils.print	Utility classes dealing with printing

How the Client Locates Plug-ins

The client tracks plug-ins by maintaining a list of jar files that the user has selected using the Configure Plugin Components dialog box. To display this dialog box, from Administration Services Console, select **Tools > Configure components**.

When a jar file is selected, the dialog scans through each package in the jar file looking for a class called MiscellaneousHandler.class. When a class with this name is found, the jar file name and the package name containing that class file are retained by the plug-in manager. Therefore, each jar file must contain exactly one package with a MiscellaneousHandler class in it.

When Administration Services Console starts, the plug-in manager scans each jar file in its stored list, looking for the MiscellaneousHandler.class file in the specified package. If this class is found, the plug-in manager adds this plug-in to its list of plug-ins. Other parts of the application, or any other plug-in can then call the plug-in manager to get a list of all plug-ins.

Basically, each plug-in consists of the following:

A jar file containing a package with a MiscellaneousHandler class

For the rest of this document, we will use the term “plug-in root” to refer to the package containing the MiscellaneousHandler class.

For example, the rest of this document uses a plug-in with a class named com.MyPlugin.MiscellaneousHandler; the plug-in root refers to the package com.MyPlugin.

Adding Functionality

There are many ways to add functionality to Administration Services Console. The following sections describe how this is currently implemented:

Semantic Rules

Many of the following sections have a description of semantic rules. In most cases, Administration Services Console does not enforce these rules. We expect that developers writing plug-ins for Administration Services will be “well-behaved citizens”; philosophically, this means that a lot of the console is open, accessible, and plug-ins can have an adverse effect on the application by taking actions that break these semantic rules.

Adding a Branch to the Enterprise Tree

When Administration Services Console starts, a panel is created called the “Enterprise View”. This panel contains an instance of the CTree class. The text for the root node is called “Enterprise View”. Each plug-in gets the opportunity to add children to the root node. This permits each plug-in to have its own branch in the Enterprise Tree view.

In the plug-in root, add a class called ConsoleTreeHandler. In our example, this would be com.MyPlugin.ConsoleTreeHandler. Add a method called “populateTree()” to this class. The source code should look something like the following example:

```
public class Console TreeHandler {
  //a no-argument constructor is required by the framework.
  public ConsoleTreeHandler() {
  }

  public void populateTree(CTreeModel model) {
    Object root=model.getRoot();

    //strictly speaking, this next check should not be
    //necessary; however, we do this to make sure some other
    //plug-in hasn’t replaced the root node with something
    //unexpected.
    if ((root!=null) && (root instanceof CTreeNode))
      //create any CTreeNode-derived objects, adding them
      //as children of the root node.
    }
  }
}
```

There are some unenforced semantic rules associated with CTree objects:

- The only action a plug-in should perform on the CTreeModel is to get the root. The plug-in should never replace the root node, traverse the tree model, or make changes to any other descendants of the root node.

- Every object added as child of the root node must be derived from a CTreeNode. Theoretically, any object can be added as a child of the root; however, other parts of the framework will not respond to those objects in any meaningful way.

Note: A plug-in can be called more than once if the console disconnects from the current server. The code needs to check that the node has already been added and only append nodes that have not been added previously. The source code should look something like the following Essbase ConsoleTreeHandler code:

```
/**
    * populates the model with information required.
    */
    public void populateTree(CTreeModel model) {
        Object root=model.getRoot();
        CTreeNode rootNode=null;
        boolean firstTime=true;
        if (root instanceof CTreeNode) {
            rootNode=(CTreeNode) root;
            if (rootNode.getChildCount()!=0) {
                CTreeNode node=(CTreeNode) rootNode.getFirstChild();
                while (node !=null) {
                    if (node instanceof ServersContainerNode) {
                        firstTime=false;
                        UIFactory.refreshServerList();
                        break;
                    }
                    node=(CTreeNode rootNode.getChildAfter(node);
                }
            }
        }
        if (firstTime) {
            CTreeNode essnode=new ServersContainerNode(null);
            rootNode.add(essnode);
            final CTreeNode containerNode=essnode;

            ConsoleManager.getConsoleInstance().addFrameListener(new
            WindowAdapter() {
                public void windowClosed(WindowEvent e) {
                    //signal that we are simply disconnecting instead of
                    //closing
                    if (e.getNewState() == WindowEvent.WINDOW_OPENED &&
                        e.getOldState() == WindowEvent.WINDOW_OPENED) {
                        Server[] servers = UIFactory.getServers();
                        for (int ii=0; ii<servers.length; ii++) {
                            UIFactory.removeServerInstance(servers[ii]);
```

```
                }
            }
            UIFactory.disconnectAll();
        }
    })
  }
}
```

Adding Children to Other Tree Nodes

When a CTreeNode object is expanded for the first time, each plug-in gets the opportunity to add child nodes to the CTreeNode being expanded.

In the plug-in root, add a class called ConsoleTreeHandler. In our example, this would be com.MyPlugin.ConsoleTreeHandler. Add a method called "getTreeNodeChildren()" to this class. The source code should look something like the following example:

```
public static CTreeNode[] getTreeNodeChildren(CTreeNode node) {
  // strictly speaking, this check for null should never be
  // necessary
  if (node == null)
    return new CTreeNode[0];
  if (node instanceof SomeSpecificTreeNode) {
    CTreeNode[] theChildren = new CTreeNode[5];
    theChildren[0] = new ChildNode();
    theChildren[1] = new AnotherChildNode();
    // and so on...
    return theChildren;
  }
  else if (node instanceof SomeOtherTreeNode) {
    // different set of children here.
  }
  // and if we're not interested in any other types.
  return new CTreeNode[0].
}
```

Item of interest for this operation:

- This method could be declared public Object[] getTreeNodeChildren(CTreeNode node) and it would still get called. The CTreeNode method that handles this checks the return value for null and also checks each item returned in the array to ensure that it is an instance of a CTreeNode object. Declaring the method as in the example enforces to the implementer of the plug-in that the items returned must be items derived from the CTreeNode class.

- The only arrangement that currently is done is that child nodes that cannot have children are placed before the child nodes that can have children. Nodes from plug-ins are placed after the nodes that the parent node already knows about.

Permitting Plug-ins To Add Children To Your Tree Nodes

By default, all CTreeNode based objects that can have children have this feature enabled. Currently, there is no way to prevent plug-ins from adding children to a tree node if that tree node can have children.

Adding Context Menu Items To Tree Nodes

When the CTree control detects that a popup menu needs to be displayed, it calls the instance of the CTreeNode and asks it for a list of items to display in the context menu. The following are rules or guidelines for how CTreeNode objects should build this array:

- The signature for the CTreeNode method is:

  ```
  public Component[] getContextMenuItems();
  ```

  Even though this method is declared to return an array of Component objects, it is highly recommended that the objects returned all be instances of the JMenuItem class (or classes derived from JMenuItem).

- The state of any menu items returned from the getContextMenuItems() method must be properly initialized; that is, enabled/disabled, checked.

- The JMenuItem objects (or whatever objects) must be properly linked to the specific CTreeNode object that is being called. The event passed in the actionPerformed() call will contain none of this contextual information.

The CTree then calls each plug-in, retrieving any additional menu items for the specified CTreeNode object. If there are additional items, the CTree places a separator after the original menu items, then places all of the plug-in items in the popup menu, and then, if the CTreeNode can be put on custom views, puts another separator and the menu items related to custom views.

For a plug-in to respond to the CTree properly in this case, add a class called ConsoleTreeHandler to the plug-in root package. In our example, this would be com.MyPlugin.ConsoleTreeHandler. Add a method called “getContextMenuItemsFor()” to this class. The source code could look something like the following example:

```
public static Component[] getContextMenuItmsFor(CTreeNode node) {
   // strictly speaking, this check for null should never be
   // necessary
   if (node == null)
      return new Component[0];
   if (node instanceof SomeSpecificTreeNode) {
      JMenuItem theItem = new JMenuItem("Walk");
      JMenuItem anotherItem = new JMenuItem("Don't walk");
      theItem.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            // take action here.
         }
      }
      return new Component[] { theItem, anotherItem };
   }
   else if (node instanceof SomeOtherTreeNode) {
      // different set of menu items here.
   }
   // and if we're not interested in any other types.
   return new Component[0].
}
```

Items of interest for this operation:

- This method can be declared to return anything. For instance, for better type safety within your own code, you could declare the method to be “public static JMenuItem[] getContextMenuItemsFor(CTreeNode node)”; however, the CTree object making the call will only use items that are derived from the Component class.
- This example is very bare bones; for instance, the returned JMenuItem object does not know which CTreeNode object it should be working with; even worse, one of the items does not have an action listener associated with it. For a complete example of this, please see the sample plug-ins developed by the Administration Services development team.
- CTreeNode (being derived from DefaultMutableTreeNode) objects have a user object. This is available through the getUserObject() method. The intent is that the user object for a node represents that data that the node has been created for and this is the data that would need to be associated with the menu item. For instance, a node might have an object representing an Analytic Services application. In the above example, we would then perform a node.getUserObject() call to obtain this Analytic Services application object

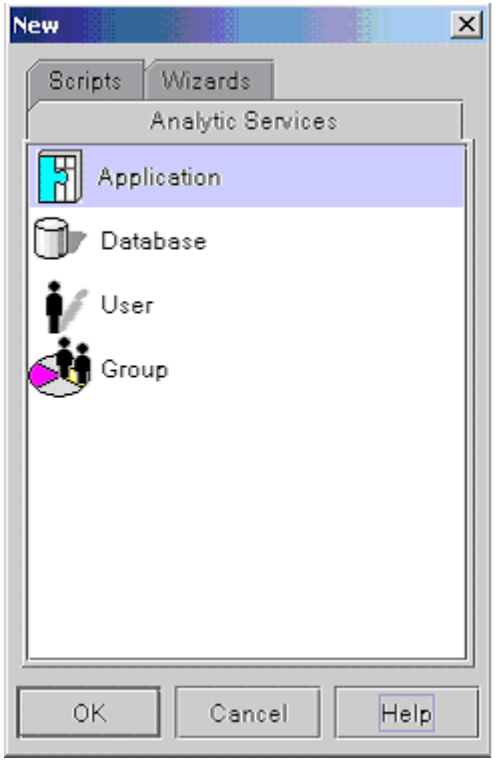
- Because plug-ins are called in the order that the user has arranged them in the Component Manager dialog box, there currently is no way to force the menu items from one plug-in to appear before the menu items of another plug-in.

Handling the File > New Menu Item

Obviously, it makes sense that the framework would provide a single **File** > **New** dialog box; then the issue becomes, “How do we get every conceivable object that can be created into the **File** > **New** dialog box?”.

When the **File** > **New** menu item is invoked, the framework creates and displays an instance of the com.essbase.eas.framework.client.ui.filedlgs.NewDialog.java class. The results look something like the following dialog box:

Figure 2: Example File > New Dialog Box



In Figure 2, there are three tabs on this dialog box:

- Analytic Services

- Scripts
- Wizards

These tabs were added, in this case, by the Analytic Services plug-in and the Administration Services plug-in. The dialog box itself provides the following items:

- The OK, Cancel, and Help buttons
- An instance of a JTabbedPane to act as a container for each of the other panels
- Actions for the OK, Cancel, and Help buttons that make the appropriate calls into the plug-in that provided the active panel

To add a panel and tab to the New dialog box, add a class called NewDialogHandler to the plug-in root package. In our example, this would be com.MyPlugin.ConsoleTreeHandler. Add a method called "populatePanel()" to this class. The source code could look something like the following code:

```
public void populatePanel(JTabbedPane panel) {
    // create an instance of the right kind of panel
    CNewDialogScrollPanel s = new CNewDialogScrollPane();
    s.setHorizontalScrollBarPolicy(JscrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED);
    s.setVerticalScrollBarPolicy(JscrollPane.VERTICAL_SCROLLBAR_AS_NEEDED);

    // create a list model that has some items in it.
    DefaultListModel model = new DefaultListModel();
    model.addElement(new JLabel("XTD Connection");
    model.addElement(new Jlabel("SQL Connection");

    // make sure the list box has a selected item
    list.setSelectedIndex(0);

    // toss the list into the scroll pane and ensure that the new
    // dialog box will call this instance when the OK button is
    // clicked.
    s.getViewport().add(list);
    s.setOkHandler(this);

    // add this panel to the tabbed panel we were given
    panel.add("My Objects", s);

}
```

For this to work correctly, you would also need to add the following method to the class:

```
public void handleOk(Component component) {
    if (component instanceof CNewDialogScrollPane) {
        CNewDialogScrollPane scroller = (CNewDialogScrollPane) component;
```

```
        Component control = scroller.getViewport().getComponent(0);
        if (control != null) && (control instanceof JList)) {
            // extract the selected item in the JList.
            // ensure that it is one of the ones we added.
            // take the appropriate action.
        }
    }
}
```

Items of interest for this operation:

- Items added to the JTabbedPane must be derived from the CNewDialogScrollPane class.
- Since CNewDialogScrollPane is derived from JScrollPane, the components that give the best visual presentation when displayed in the new dialog box are components that are derived from JTable, JTree, and JList.
- For the best visual presentation, the component added to the scroller can have custom renderers, event handlers, and so on.
- For the best behavior, this list would need a MouseListener added to it to listen for double click events. This MouseListener then would need to call the enclosing dialog box's handleOk() method.
- A plug-in can add more than one panel to the JTabbedPane instance.

Adding Items To Menus

Menu items are typically displayed in three ways:

- Static
- From an internal frame
- From a CTreeNode on the console tree

Static Menu Items

Static menu items are always displayed. The following example is for a static menu item:

```
public class XYZ {
    private CMenu editorsMenu = new CMenu("Scripts",
    Console.ID_ACTIONS_MENU - 1, this);
    private CMenuItem outline  = new CMenuItem("Outline", null, 0, this);
    private CMenuItem report  = new CMenuItem("Report", null, 1, this);
```

```
    private CMenuItem calc   = new CMenuItem("Calc", null, 2, this);
    private CMenuItem maxl   = new CMenuItem("Maxl", null, 3, this);
    private CMenuItem mdx    = new CMenuItem("Mdx", null, 4, this);
    private CMenuItem dataprep  = new CMenuItem("DataPrep", null, 5, this);

       void createMenu() {
report.addActionListener(new AbstractAction("createReport") {
    public void actionPerformed(ActionEvent e) {
    }
});

calc.addActionListener(new AbstractAction("createCalc") {
    public void actionPerformed(ActionEvent e) {
    }
});

maxl.addActionListener(new AbstractAction("createMaxl") {
    public void actionPerformed(ActionEvent e) {
    }
});

mdx.addActionListener(new AbstractAction("createMdx") {
    public void actionPerformed(ActionEvent e) {
    }
});

outline.addActionListener(new AbstractAction("createOutline") {
    public void actionPerformed(ActionEvent e) {
    }
});

dataprep.addActionListener(new AbstractAction("createDataPrep") {
    public void actionPerformed(ActionEvent e) {
    }
});

editorsMenu.add(outline);
editorsMenu.add(dataprep);
editorsMenu.add(calc);
editorsMenu.add(report);
editorsMenu.add(maxl);
editorsMenu.add(mdx);

LocalizeUtils.localizeMenu(resources, editorsMenu);
ConsoleManager.getConsoleInstance().mergeMenus(new Component[] {
editorsMenu});
```

```
        }
}
```


Internal Frame Menu Items

Menu items from an internal frame only display when the internal frame is active. If the internal frame is deactivated or or closed, then these menu items no longer are displayed. The following example is for an internal frame menu item:

```
public class XYZ extends CInternalFrame {
  public Component[] getFrameMenus() {
// Like the example above
    return (new Component[] { editorsMenu});
  }
}
```

Console Tree Menu Items

These menu items only display when a node is selected. The following example is for a console tree menu item:

```
public XYZ extends CTreeNode {
  public Component[] getActionMenuItems() {
    return (new Component[] { editorsMenu});
  }
}
```

In general, there are predefined menu positions defined in the Console interface:

```
  public static final int ID_FILE_MENU = 0;
  public static final int ID_EDIT_MENU = 1;
  public static final int ID_VIEW_MENU = 2;
  public static final int ID_ACTIONS_MENU = 10;
  public static final int ID_TOOLS_MENU = 20;
  public static final int ID_WIZARD_MENU = 30;
  public static final int ID_WINDOW_MENU = 90;
  public static final int ID_HELP_MENU = 99;
```

If the CMenu item's (that is returned from the above example) position matches with one of the predefined ones, then that CMenu item's submenus are merged in else that CMenu is inserted based on the position. So if the CMenu has a position of ID_ACTIONS_MENU, then the items are merged in to the action menu item that is already on the main menubar. If the CMenu has a position (ID_ACTIONS_MENU - 1), then the CMenu is inserted before the action menu.

Handling Save As

Save As requires the plug-in to implement the interface SaveAsRequestor. The following example uses an inner class:

```
  if (saveAsAdapter == null) {
  saveAsAdapter = new SaveAsAdapter();
  }
  SaveAsDialog.showDialog(resources.getString("exportTitle"),
  (SaveAsRequestor) saveAsAdapter);

}
```

The initSaveAsDialog is called to allow the dialog/frame to initialize the SaveAsDialog as it needs to. By default a file system chooser is added to mainPanel at index 0. A plug-in can add other panels to save to other places in this method.

When an object is selected from any panel, then the saveAsObject method is called with the selected object. If the file system panel is selected the object will be a File if the plug-in adds a panel of their own it they will have to perform the steps to save the object.

```
private class SaveAsAdapter implements SaveAsRequestor {
    public void initSaveAsDialogComponents(JTabbedPane mainPanel) {
        String xmlString = ResourceUtilities.getStringSafely(resources, XML_FILES);
        DefaultFileFilter xmlFilter =  new DefaultFileFilter(xmlString, "xml",
        resultAction);
        JFileChooser jfc = (JFileChooser) mainPanel.getComponentAt(0);
        jfc.setFileSelectionMode(JFileChooser.FILES_ONLY);
        if (jfc.isAcceptAllFileFilterUsed() == true)
            jfc.setAcceptAllFileFilterUsed(false);
        jfc.setFileFilter(xmlFilter);
    }

    public void initExtraComponents(JPanel extraPanel) {
    }

    public boolean saveAsObject(Object saveObject) {
        boolean saved = false;
        if (saveObject instanceof File) {
            File file = (File) saveObject;
            String exportFile = file.getPath();
            if (exportFile != null) {
                String msg = "";
                if (AdminServerPropertiesHelper.requestExportDB(exportFile))
                {
                    msg = resources.getString("sucEXDBMsg");
```

```
                    StandardMessages.showMessage(resources, "exportTitle", msg,
                    JOptionPane.DEFAULT_OPTION, JOptionPane.INFORMATION_MESSAGE);
                    saved = true;
                }
                else
                {
                    msg = resources.getString("failEXDBMsg");
                    StandardMessages.showMessage(resources, "exportTitle", msg,
                    JOptionPane.DEFAULT_OPTION, JOptionPane.ERROR_MESSAGE);
                }
            }
        }
        return saved;
    }
    public void setFocusComponent() {
    }
}
```

Standard Controls

While it is not required that plug-ins use the standard controls provided by the framework classes, there are some benefits to using them. Namely, some consistency of look and feel is provided, some housekeeping tasks are performed by the standard controls, there is support for internationalization, accessibility, and so on.

The StandardDialog Class

The StandardDialog class is an extension of the JDialog class and was introduced for the following reasons:

1. Standardize the mechanism for internationalization and localization handling
2. Standardize the position, location, and behavior of dialog “action” buttons
3. Standardize some of the accessibility handling for modal dialogs
4. Standardize the handling of results

The StandardDialog class contains the following protected (or private) fields:

Table 2: Fields in the StandardDialog Class

Field	Description
`okBtn`	An instance of an OK button. This is one of the standard controls described in "Dialog Initialization" on page 37.
`cancelBtn`	An instance of a Cancel button. This is one of the standard controls described in "Dialog Initialization" on page 37.
`helpBtn`	An instance of a Help button. This is one of the standard controls described in "Dialog Initialization" on page 37.
`buttons`	An instance of a ButtonPanel. The ButtonPanel is one of the standard controls described in "Dialog Initialization" on page 37.
`resources`	An instance of a ResourceBundle object. This resource bundle is used for internationalization purposes.
`adapter`	An instance of a StandardDialogAdapter.
`dialogResult`	An instance of a DialogResult object.
`saveDialogBounds`	A boolean value indicating whether the bounds (location and size) of this dialog should be saved when it is closed.

Name of Standard Dialog Class

The name of the Standard Dialog class is StandardDialog. It is in com.essbase.easui.StandardDialog.class.

Dialog Creation

There are at least 11 constructors for the StandardDialog class; most of these chain to another constructor. The two constructors that should be invoked by derived classes are the ones with the following signatures:

StandardDialog(Frame owner, String title, boolean modal, DialogResult result);

StandardDialog(Dialog owner, String title, boolean modal);

Most of the other constructors exist only to match constructor names of the JDialog class.

Dialog Initialization

During the call to the StandardDialog constructor, the following initialization steps will occur:

- An OK button, a Cancel button, and a Help button are created

  These are the standard buttons used by most dialogs. If the dialog being implemented uses a different set of buttons (for instance, Close, Apply, Next, and so on) the derived class should implement instances of those buttons.

- A ButtonPanel containing the OK, Cancel, and Help buttons is created

  If the dialog being implemented wants the button panel to contain a different set of buttons, it should call buttonPanel.changeButtons(new Jbutton[] { closeBtn, helpBtn }); // as an example.

- A ResourceBundle instance is created

  This resource bundle is used to perform localization work within the dialog. It is important to know where the standard dialog looks for the instance of the resource bundle. For example, if the dialog class is MyFunnyDialog, then the resource bundle must be in a file called resources/MyFunnyDialog.properties.

- A StandardDialogAdapter is created and is added as a window listener to the dialog

---

CAUTION: Because of the implementation of the StandardDialogAdapter class, there should never be a reason for a descendant class of StandardDialog to attach a WindowListener to itself. Routing of all window events should be handled by the StandardDialogAdapter. If the descendant class needs to take action when a window close, window open, and so on, event occurs then override the methods in StandardDialog that the StandardDialogAdapter calls.

- Sets the instance of the dialog result to the value passed in, if any

  To understand how this works, see “Dialog Results” on page 39.

- Sets the dialog’s default close operation to DISPOSE_ON_CLOSE

In most cases, this is the desired behavior; for a dialog that needs a different behavior, this can be changed by the constructor in the descendant class.

- Sets the dialog's content pane layout to be a BoxLayout oriented vertically

  If necessary, this can be changed by the derived class.

- Adds entries to the action and input maps of the dialog's root pane to take a "default action" when the Enter key is pressed by the user

  For more information on what this default action is, and why this step is necessary, see the section of this document titled "Dialog default action".

Dialog Default Action

The Microsoft Windows operating environment has the concept of a default button when modal dialog windows are open. The default button is painted in a way that makes it stand out visually to the user. Normally, that is the OK button; however, it can be any action button on the dialog. To handle this concept, the StandardDialog adds entries to the action and input maps of its root pane for handling the enter keystroke.

If your dialog box does not have an OK button or, if at any time, the default button should be some other button, then a call like the following needs to be performed:

```
dlg.getRootPane().setDefaultButton(closeBtn);
```

Dialog Keyboard Handling, Focus Order, Action Maps, and So On

Depending on which buttons are inserted into a dialog, certain keystrokes will be mapped automatically:

- The Enter key
- The Esc key
- The F1 key (for help)

These are the primary keystrokes that are mapped by the standard dialog and the standard buttons.

To add handling when these keystrokes are pressed, do the following:

- For the Enter key, override the handleOk() method. If everything finishes correctly and the dialog needs to be released, then call super.handleOk(). This will ensure that the dialog shuts down properly.
- For the Esc key, override the handleCancel() method. The standard dialog behavior closes the dialog, releases all the controls, disposes of contained components, and so on. In most cases, this method will not need to be overridden.
- For the F1 key, override the handleHelp() method. If the dialog has been connected via the Administration Services help system via the normal manner, this step should not be necessary.

By default, the Java Swing implementation sets the focus order of controls to correspond to the order in which they were added to their container, and then those container's to their container, and so on. This can be overridden by making a call to the method DialogUtils.setFocusOrder(). This mechanism should be used in all dialogs to ensure the focus order of controls is correct and doesn't rely on how the code for building the containment models was written.

Dialog Results

In many cases, a dialog needs to return a significant amount of information to the calling mechanism. Unfortunately, the method Dialog.show() is declared as void and does not return any data.

If, when implementing a dialog, results from the dialog are needed, the recommended way to get those is by doing the following tasks:

- Extend the DialogResult class to contain references and additional data needed by the dialog and/or returned by the dialog.
- Before creating the dialog, create an instance of the DialogResult class.
- Ensure that the dialog has at least one constructor that accepts an instance of a DialogResult object.
- In the constructor for the dialog class derived from StandardDialog, pass the DialogResult object to the correct StandardDialog constructor.
- During the handling of the OK button, set the results back into this instance.

Methods to Override

The StandardDialog class has a set of methods that can be overridden. Whether each of these methods are overridden will depend on the needs of each derived class. See the *Essbase Administration Services Java API Reference* for detailed information about each of the following methods:

- dispose()
- handleCancel()
- handleOk()
- handleWindowClosed()
- handleWindowClosing()
- handleWindowOpened()

Standard Buttons and Other Controls

There are a large number of standard controls provided by the client framework. The following is a representative list; for more complete information, see the *Essbase Administration Services Java API Reference* for the com.essbase.eas.ui package and descendant packages.

Note: This is not a complete list of controls. The plug-in developer should browse the Java API Reference for the com.essbase.eas.ui package and other packages under this one for additional standard components.

- ActivateButton
- ApplyButton
- BackButton
- BooleanComboBox
- ButtonPanel
- CancelButton
- CloseButton
- DoneButton
- FinishButton
- HelpButton

- ListMoverPanel
- NextButton
- NumericTextField
- OkButton
- ReadOnlyTextFrame
- RefreshButton
- ResetButton
- SimpleWizardPanel
- VerticalPairPanel
- WizardPanel

Administration Services Console Services

The client framework provides the following Administration Services Console services:

- Retrieving the CSS Token from the Console
- Sending E-mail

Retrieving the CSS Token from the Console

The CSS token is retrieved from the FrameworkUser object which is returned on successful login to Administration Server.

```
import com.essbase.eas.client.intf.Login;
import com.essbase.eas.client.manager.LoginManager;
import com.essbase.eas.admin.defs.*;
import com.essbase.eas.admin.client.*;
import com.essbase.eas.framework.defs.FrameworkUser;

private String getToken() {
        String loginToken = null;
        Login login = LoginManager.getLoginInstance();
        if (login != null) {
            FrameworkUser u = (FrameworkUser)
login.getProperty("FrameworkUser");
            if (u != null) {
                loginToken = u.getToken();
```

```
                    }
                }
                return loginToken;
            }
```

Sending E-mail

Administration Services Console has integrated support for sending e-mail using the JavaMail API. We have wrapped the classes and provide a dialog for sending e-mail. There is also support in the InternalFrame class to send from any class derived from the CInternalFrame class.

The following is a simple example of how to send the contents of a text area in an e-mail from a dialog.

```
Import com.essbase.eas.ui.email.*;

public void email() {
            JFrame fr = ConsoleManager.getConsoleFrame();

            SendEmail email = new SendEmail(fr, fr.getTitle(), new
            Object[] {
getTextArea().getText()} );
            email.send();
      }
```

The following example is for a window derived from CInternalFrame. The methods, isEmailable() and getObjectsToEmail, are methods in the CInternalFrame class.

```
      public boolean isEmailable() {
            return true;
      }

      public Object[] getObjectsToEmail() {
            HTMLDoc doc = new HTMLDoc();

            doc.setTitle(getTitle());
            doc.addObject(doc.getHeading(2, doc.getStyleText(getTitle(),
doc.BOLD | doc.UNDERLINE), doc.CENTER));

            doc.addObject(doc.BR);

doc.addObject(TableUtilities.getHTML((DefaultTableModel)locksTable.getMod
el()));
```

```
            return (new Object[] { new EmailAttachment(doc.toString(),
"Locks.htm", EmailAttachment.HTMLTEXT, "", EmailAttachment.ATTACHMENT)});
      }
```

Note: Sending an e-mail puts an entry in the background process table showing the outcome of the e-mail.

Internationalization

The framework provides a set of internationalization and localization utilities in the package com.essbase.eas.i18n. These classes provide a mechanism for locating resources associated with a window or dialog box, loading resource bundles based on the locale, localizing collections, arrays of components, or containers. There is also an i18n-friendly string collator class.

Packaging the Plug-in

The only packaging requirement is that all classes and resources necessary for a client plug-in must be contained in the same jar file. You must include an entry in the jar file which defines the other jar files it depends on. For example, lets say the plug-in jar file `xyz.jar` depends on `abc.jar` and `cde.jar`, include the following entry in the manifest file for the plug-in jar file:

Class-Path: `xyz.jar cde.jar`

Chapter 3

Writing Server-side Command Listeners

This chapter explains how to write a command listener for the Essbase Administration Services mid-tier web server. Installable command listeners are the mechanism for extending the functionality of the Administration Services Web server. This chapter includes the following topics:

Preliminaries

We make the following user presumptions:

- You have some Java experience
- You have access to the *Essbase Administration Services Java API Reference*
- Since different developers use different build tools and environments, we do not discuss how to do anything for specific development environments. Rather, we describe the desired results, leaving it to the developer to know how to achieve these results with their specific development tools.

Note: For the purposes of this documentation, the terms "Administration Services web server", "Administration Services servlet", "Administration Services mid-tier", "Administration Services framework", and, simply, "the framework" can generally be taken to refer to the same object.

The *framework* is the Administration Services servlet and associated classes that receive commands, handle housekeeping duties, return results, and route commands to the registered listener.

Command Listeners

A *command listener* is an instance of any class that implements the CommandListener interface; however, for practical purposes, all plug-in command listeners should extend one of these classes:

- EssbaseCommandListener
- AppManCommandListener
- AbstractCommandListener

The framework uses command listeners as the mechanism to properly route commands to be handled.

When the Administration Services servlet starts up, it builds a table of command listeners, the commands that each command listener can handle, and the method in the command listener for that command. As client applications send commands (http requests), the Administration Services servlet uses the command's operation parameter to determine the command listener and method to route the request to.

For example, a typical command might be to log in to the Administration Services servlet. When expressed as an http request, this command will look something like this:

```
http://LocalHost/EAS?op=login&name=user1&password=hello
```

When all of the http information is parsed out, the part that would be of interest to the Administration Services servlet are the following parameters:

op=login

name=user1

password=hello

The framework uses the “op” parameter to route the command to the correct command listener. If the command listener has been registered correctly, the framework will also collect the “name=” and “password=” parameters and pass them as arguments to the method in the command listener.

Class Hierarchy

The class hierarchy for the command listeners is:

```
com.essbase.eas.framework.server.application.AbstractCommandListener
   com.essbase.eas.server.AppManCommandListener
   com.essbase.eas.essbase.server.EssbaseCommandListener
```

All three of these classes are declared as abstract. You must extend from one of these three classes in order to have the framework find your command listener.

The AbstractCommandListener class provides the basic functionality that is needed for the framework. Most of the methods in this class are either final or protected; for most practical purposes, implementers of derived classes should not override the protected methods of this class. For a description of those methods that can be useful to implement in a derived class, see the section "Which Methods to Override" on page 47.

The AppManCommandListener class adds some small functionality to the AbstractCommandListener, mostly dealing with EAS servlet session validation and exception handling during command routing.

The EssbaseCommandListener class adds some Analytic Services-specific functionality, primarily Analytic Services session validation.

Which Class To Extend

Do not extend the AbstractCommandListener class, even though it is declared public. The EssbaseCommandListener.handleEventPrep() method checks some standard parameters for an Analytic Server name, application name, and database name and ensures a connection to that database if those parameters exist. If the implementer of the new command listener wishes to take advantage of the session handling performed by the EssbaseCommandListener, then they should extend this class; however, if this isn't necessary, the new command listener can extend the AppManCommandListener class.

Which Methods to Override

AbstractCommandListener.getCommands() must be overridden. We explain more about this method in the section, "Registering Commands" on page 48.

The handleEventPrep(), handleEventPost(), and handleEventException() methods may be overridden. These three methods, along with AbstractCommandListener.handleEvent(), form the core processing for any command received by the framework.

Once the framework determines which command listener to route a command to, it calls that command listener's handleEvent() method. Since the AbstractCommandListener declares this method as final, the framework always calls the method in AbstractCommandListener. This method then performs the following sequence of steps:

1. Calls handleEventPrep(); if this method returns true, then continues with step 2.
2. Gets the command listener's method that handles this specific command. If this method cannot be located, logs an error with the logging utility.
3. Converts the arguments from the http command into an array of Java objects.
4. Using Java introspection, invokes the method.
5. If no exceptions were thrown, invokes handleEventPost().
6. If exceptions were thrown in steps 4 or 5, calls handleEventException().

Any change to the processing of events before they arrive at a specific method in the command listener must be done by overriding the handleEventPrep() method. For instance, this is where the EssbaseCommandListener class checks Essbase sessions and the AppManCommandListener checks for a valid servlet session.

In most cases, the handleEventPost() method is empty and the handleEventException() method is empty.

Registering Commands

After a command listener is instantiated by the framework, the framework calls the getCommands() method. This method returns an array of CommandDescriptor objects. The CommandDescriptor objects describe each command that the CommandListener is designed to handle. The CommandDescriptor object consists of three main parts:

- A string for the command
- The method in the command listener to call
- The list of arguments expected for this command.

The next few sections describe the classes used by the framework when registering commands.

Note: All of these classes are in the package com.essbase.eas.framework.defs.command.

CommandString Class

When most people write a command listener, they think of it handling commands like “GetDatabaseList”, “GetUsers”, “DeleteUsers”, and so on. Since each command must be unique, it is easy to see how this would lead to confusion. The CommandString class was introduced to let each programmer of command listeners think of their commands in the simplest way. The CommandString class is declared as:

```
public abstract class CommandString
```

The only constructors are declared as:

```
private CommandString() { ... }
protected CommandString(String original) { ... }
```

These two declarations combined mean that instances of this class can never be instantiated and derived classes must call the CommandString(String original) constructor with a valid String object as the parameter.

The most important action that instances of this class do is take the original String object and prepend the class name, including the package name, to the front of the String. This new value is then returned when the object’s toString() method is called.

CommandArgument Class

The CommandArgument class describes individual arguments to commands. It contains the following fields:

- String name (available through the getName() method)

  This is the name of the http parameter corresponding to this argument.

- boolean required (available through the isRequired() method)

  Indicates whether this argument is required. The intent is that the framework can check this field when routing a command and return a pre-defined error status to the client if a required field is missing.

- Class ClassType (available through the getClassType() method)

  This is used so the framework can convert the incoming text value to an appropriate object type.

- Object defaultValue (available through the getDefaultValue() method)

  The framework will substitute this object for the argument if the argument is missing from the command.

- Boolean hidden (available through the isHidden() method)

  The framework can log the retrieval and routing of commands and their parameters. Setting this field to true means the framework will not echo the value of this argument in the log file. This would be useful for passwords, and so on.

These fields are all declared as private and, since there are no setXXX() methods, cannot be changed after a CommandArgument object is constructed.

CommandDescriptor Class

The CommandDescriptor class combines the CommandArgument and CommandString classes into a cohesive value so that the framework can construct its internal tables and route the commands as they are received.

The examples in the following sections show how all of this fits together.

Examples

This section includes the following sample code:

- "Example.java" on page 51
- "ExampleCommandString.java" on page 52
- "ExampleDescriptor.java" on page 53
- "ExampleCommandListener.java" on page 54

Example.java

```
// this is a simple class used as a parameter to show how the
// framework can separate out command arguments that are object
// types embedded in XML. For more information on how the
// framework uses XML to transport "generic" objects between the
// mid-tier and the client, please see the Java Docs references
// for the XMLTransferObject class.
public Example extends Object {
  private String name = "";
  private String[] text = new String[0];
  // no-argument constructor. Must be public for XML Transfer
  // to work.
  public Example() {
  }

  public String getName() {
    return name;
  }

  public void setName(String value) {
    name = value;
  }

  public String[] getSampleText() {
    String[] result = new String[text.length];
    for (int i = 0; i < result.length; ++i)
      result[i] = text[i];
    return result;
  }

  public void setSampleText(String[] values) {
    if (values != null) {
      text = new String[values.length];
      for (int i = 0; i < values.length; ++i)
        text[i] = values[i];
    }
    else {
        text = new String[0];
    }
  }
}
```

ExampleCommandString.java

```
public ExampleCommandString extends CommandString {
   // declare some static String objects in a way that we know these
   // objects do not need to be translated to different locales.
   public static final String GET_EXAMPLES_TEXT = "GetExamples";
   public static final String ADD_EXAMPLE_TEXT = "AddExample";
   public static final String DELETE_EXAMPLE_TEXT = "DeleteExample";

   // now we declare the actual commands
   public static final ExampleCommandString GET_EXAMPLES =
                           new ExampleCommandString(GET_EXAMPLES_TEXT);
   public static final ExampleCommandString ADD_EXAMPLE =
                           new ExampleCommandString(ADD_EXAMPLE_TEXT);
   public static final ExampleCommandString DELETE_EXAMPLE =
                           new ExampleCommandString(DELETE_EXAMPLE_TEXT);

   // for organizational purposes, we also declare the parameters for each
   // of these commands in this file.
   public static final String PARAM_LOCATION = "location";
   public static final String PARAM_EXAMPLE = "example";
   public static final String PARAM_NAME = "examplename";

   // declare a CommandArgument object for each of these parameters
   private static final CommandArgument ARGUMENT_LOCATION =
                           new CommandArgument(PARAM_LOCATION,
                           true,
                           String.class,
                           null);
   private static final CommandArgument ARGUMENT_EXAMPLE =
                           new CommandArgument(PARAM_EXAMPLE,
                           true,
                           Example.class,
                           null);
   private static final CommandArgument ARGUMENT_NAME =
                           new CommandArgument(PARAM_NAME,
                           true,
                           String.class,
                           null);

   // declare an array of arguments for each command.
   public static final CommandArgument[] GET_EXAMPLES_ARGS =
                           new CommandArgument[] { ARGUMENT_LOCATION };
   public static final CommandArgument[] ADD_EXAMPLE_ARGS =
                           new CommandArgument[] { ARGUMENT_LOCATION,
                              ARGUMENT_EXAMPLE };
   public static final CommandArgument[] DELETE_EXAMPLE_ARGS =
```

```
                        New CommandArgument[] { ARGUMENT_LOCATION,
                            ARGUMENT_NAME };
}
```

This class declares command strings and describes the arguments for three commands that will be supported by the ExampleCommandListener class. If the toString() method of each ExampleCommandString object declared in this source code file were called, the results would be:

```
ExampleCommandString.GetExamples
ExampleCommandString.AddExample
ExampleCommandString.DeleteExample
```

Every CommandDescriptor object contains a reference to an object derived from CommandString; it is through this mechanism that the framework guarantees every command name is unique.

ExampleDescriptor.java

```
public class ExampleDescriptor extends CommandDescriptor {
    private static final String GET_EXAMPLES_METHOD = "getExamples";
    private static final String ADD_EXAMPLE_METHOD = "addExample";
    private static final String DELETE_EXAMPLE_METHOD = "deleteExample";

    public static final CommandDescriptor GET_EXAMPLES =
                new CommandDescriptor(ExampleCommands.GET_EXAMPLES,
                            GET_EXAMPLES_METHOD,
                            ExampleCommands.GET_EXAMPLES_ARGS);
    public static final CommandDescriptor ADD_EXAMPLE =
                new CommandDescriptor(ExampleCommands.ADD_EXAMPLE,
                            ADD_EXAMPLE_METHOD,
                            ExampleCommands.ADD_EXAMPLE_ARGS);
    public static final CommandDescriptor DELETE_EXAMPLE =
                new CommandDescriptor(ExampleCommands.DELETE_EXAMPLE,
                            DELETE_EXAMPLE_METHOD,
                            ExampleCommands.DELETE_EXAMPLE_ARGS);
}
```

ExampleCommandListener.java

```
public class ExampleCommandListener extends AppManCommandListener {
   // the method called when the GetExamples command is received.
   public boolean getExamples(CommandEvent theEvent,
                              ServiceContext theContext,
                              String theLocation) {
          // the details will be filled in later
          return true;
   }

   // the method called when the AddExample command is received.
   Public Boolean addExample(CommandEvent theEvent,
                              ServiceContext theContext,
                              String theLocation,
                              Example theExample) {
          // the details will be filled in later
          return true;
   }

   // the method called when the DeleteExample command is
   // received.
   public boolean deleteExample(CommandEvent theEvent,
                              ServiceContext theContext,
                              String theLocation,
                              String theName) {
          // the details will be filled in later.
          return true;
   }

   // the framework calls this method to get the descriptors for
   // the commands supported by this command listener.
   public CommandDescriptor[] getCommands() {
      return new CommandDescriptor[] {
                              ExampleDescriptor.GET_EXAMPLES,
                              ExampleDescriptor.ADD_EXAMPLE,
                              ExampleDescriptor.DELETE_EXAMPLE };
   }
}
```

The preceding example shows the skeleton of a command listener:

1. Extend the correct class
2. Add the command handling methods
3. Override the getCommands() method to return the descriptors for those commands.

The difficulty is in the details of the command handling methods, which is covered in the next section.

Command Handling Methods

This section includes the following topics:

Method Signatures

If you were looking carefully at the example code in the preceding section, you might be saying something along the lines of, "Wait a minute, in GET_EXAMPLES_ARGS, I defined one argument, the location argument. What are these other two arguments, theEvent and theContext? Where did they come from and what do I do with them?" The answer partly lies in the older version of the Administration Services framework. The first version of the framework did not do all the type checking and parameter parsing that the new level does, so all command handling methods had the following signature:

```
public boolean handlerMethod(CommandEvent theEvent) { }
```

It was up to each method to then extract and handle the arguments along the lines of:

```
String theLocation = theEvent.getRawParameter("Location");
if (theLocation == null) {
  // oops - this shouldn't happen!
  return false;
}
```

Or, if the parameter was supposed to be a numeric value:

```
int theNumber = 0;
String theValue = theEvent.getRawParameter("Value");
if (theValue == null) {
  // oops - this shouldn't happen!
  return false;
}
try {
```

```
    theNumber = Integer.parseInt(theValue)
  }
  catch (Exception ex) {
    return false;
  }
```

In most cases, theEvent object was used mostly to get the parameters for the command. When the framework was upgraded, theEvent object was retained as the first argument to the command handler methods, even though it is rarely used.

The second argument, theContext, is actually a field in theEvent object; if you want to return results to the client, you must do so through the ServiceContext reference. Since every command handling method at some time would call theEvent.getServiceContext(), we decided to add it as a second parameter to every command handling method.

As a result of these decisions, every command handling method has the following signature:

```
public boolean handlerMethod(CommandEvent theEvent,
        ServiceContext theContext,
        Class0 value0,
        ...,
        ClassN, ValueN);
```

Where the ClassX parameters are described by the CommandDescriptor for the method.

In addition, even though the method is declared boolean, the framework never looks at the return value from a command handler method. Return values are handled within each method by a mechanism explained later in this document.

Grabbing Command Arguments

In most cases, the command arguments will have been extracted and parsed by the framework; however, special circumstances can arise whereby extra arguments are sent with each command that, for whatever reason, the programmer doesn't want to include in the CommandDescriptor object.

An example is the EssbaseCommandListener; the EssbaseCommandListener.handleEventPrep() method calls a validateSession() method that looks for the standard parameters “servername”, “appname”, “dbname”, then attempts to validate an EssbaseSession against those parameters. If this fails, then the handleEventPrep() method returns a standard error status to the client. In most cases, any EssbaseCommandListener will need these arguments

when handling commands. However, there are cases (such as in the outline editor) when those arguments aren't used. If, during implementation of a command listener method, a similar situation arises, the parameters can be retrieved by the following call:

```
String aValue = theEvent.getRawParameter("SomeParameterName");
```

This should be a rare necessity and should raise caution alarms if an implementer finds themselves needing to do this.

Sending Results Back to the Client

There are two types of results to return to a client:

1. Status of the command
2. Data that the client needs to display

The CommandStatus class is used to return the success/failure of the command to the client. The CommandStatus class only understands two types of status: SUCCESS and FAILURE. The original intent of this class was to indicate whether a command was routed successfully by the framework. However, this wasn't made explicit and, as a result, many existing command handling methods use this SUCCESS/FAILURE to indicate the status of their specific processing.

It would be a good practice to always extend this class to enable returning more specific error codes than just SUCCESS/FAILURE.

So, let's return to our example and fill in one of the command handling methods to return data and a SUCCESSFUL status to the client.

```
public boolean getExamples(CommandEvent theEvent,
          ServiceContext theContext,
          String theLocation) {
  //object used to transmit results back to the client
  XMLTransferObject xto=new XMLTransferObject();
  Example [] theResults=someMethod(theLocation);
  if (theResults == null) {
    //this is simplistic, but it shows what we need
    xto.setCommandStatus(CommandStatus.SIMPLE_FAILURE);
  }
  else {
    if (theResults.length != 0)
      xto.addAll(theResults);
    xto.setCommandStatus(CommandStatus.SIMPLE_SUCCESS);
  }
  this.storeService.set(theContext,
```

```
            DefaultScopeType.REQUEST_SCOPE,
            AppManServlet.RESULT,
            xto.exportXml());
    return true;
}
```

The XMLTransferObject is used to transmit the data and the command status back to the client; we use the defined CommandStatus.SIMPLE_FAILURE or CommandStatus.SIMPLE_SUCCESS objects to return the correct status. If results were available, they were then added to the XMLTransferObject using the addAll() method. The results were then placed in the command listener's store service using the REQUEST_SCOPE and using the key AppManServlet.RESULT. After this method returns to the framework, the framework will take any data stored using the combination DefaultScopeType.REQUEST_SCOPE and AppManServlet.RESULT and send that data back to the client as the results of the command.

Storing Temporary Data Using the Framework

In the preceding section, we gave an example of how to place data in the framework's storage so that the data would be returned to the client as the results of a command. The storeService field in each command manager can store data for additional purposes. There are six defined DefaultScopeTypes:

1. CONFIG_SCOPE

   This is used by the framework as it is initializing. It should never be used by command handler methods.

2. BUILDER_SCOPE

   This is used by the framework as it is initializing. It should never be used by command handler methods.

3. APP_SCOPE

   Using this scope type will cause the data to be stored for the life of the servlet. This should be very, very rarely used by command listeners.

4. SESSION_SCOPE

   Using this scope type will cause the data to be stored until the current client/server session is no longer valid. At that point, the framework will remove all data stored in this scope. If storing information in this scope that needs to be recovered during processing of a later command.

5. USER_SCOPE

   Using this scope makes the data available to any client connected using the same EAS user id. When all sessions associated with this user are no longer valid, the framework will remove data stored in this scope. In the current implementation, this is never used and it probably will never be used very often.

6. REQUEST_SCOPE

   Using this scope makes the data available until the framework has bundled the results of the command and returned them to the client. The framework then removes all data stored in this scope associated with the request that just finished.

Storing data is done through a command listener's store service, as in the preceding example. The StoreService interface has several get(), set(), and remove() methods. However, there is only one of each of these methods that a command listener (or other plug-in code) should call; the other methods were put in place for use by some of the framework code itself. The three method signatures are:

```
public Object get(ServiceContext context, ScopeType type, Object key);
public Object set(ServiceContext context, ScopeType type,
                              Object key, Object value);
public Object remove(ServiceContext context, ScopeType type, Object key);
```

For more information about these methods, see the *Essbase Administration Services Java API Reference*.

Packaging the Code

When packaging the code into jar files for a plug-in, follow these guidelines:

- Separate the code into three distinct pieces:
  - Code that is only used on the client
  - Code that is only used on the server
  - Code that is used in both places
- Set up the build tools to compile and package these different pieces separately in order to prevent crossover compilation. For example, the framework is packaged into the following jar files:

  ```
  framework_client.jar
  framework_common.jar
  ```

`framework_server.jar`

- Package the command listener classes in the server jar
- Package the command descriptor classes in the server jar. This is because they contain references to the method names in the command listeners and this should not be publicly available on the client.
- Package the CommandString derived classes in the common jar file. While the framework does not currently take advantage of this on the client, it will be upgraded to do the packaging of parameters and commands for client applications.
- Place any classes extending CommandStatus in the common jar file.
- Place any specialized classes (such as `Example.java`) in the common jar file.

The server jar file must contain a manifest file. Each command listener must have an entry in this manifest file that looks like the following:

Name: `ExampleCommandListener.class`

EAS-Framework-CommandListener: True

If, as is likely, the command listener has a package name that must be prepended to the name in the example above, like this:

Name: `com/essbase/eas/examples/server/ExampleCommandListener.class`

EAS-Framework-CommandListener: True

Note: Even though this is a class name, use slashes ("/") instead of dots (".") to separate the package names.

Loading the Code

After all of this, to get the framework to recognize the command listeners and route the commands to the correct place, the jar file containing the command listeners and any other jar files that this code depends on must be placed into Administration Server's `lib` directory. The exact location is dependent upon each particular installation. However, in the default installation using the Tomcat web server, the location will be:

`EAS_HOME/server/tomcat/webapps/eas/WEB-INF/lib`

After putting the jar files in this location, you must stop and restart Administration Server. To determine if the new command listeners have been installed, set the Administration Services logging level to between 5000 and 9999.

Utility Classes

There are many utility classes provided by the Administration Services framework. In particular, there are utility classes in some of the following packages:

```
com.essbase.eas.framework.defs
com.essbase.eas.framework.server
com.essbase.eas.utils
com.essbase.eas.ui
com.essbaes.eas.i18n
com.essbase.eas.net
```

The *Essbase Administration Services Java API Reference* makes it easy to navigate through these classes and learn what is available.

Index

A

B

C

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z