Cognitier SimpleIPC

Siebel COM Data Control ASP Page Sample

Version 1.0.0.1

May 04, 2009

Copyright © 2009 Cognitier, Inc. All rights reserved.

Cognitier and the Cognitier logo are trademarks of Cognitier, Inc. All other company and product names referred to may be trademarks of their respective owners. This documentation is copyrighted material and is intended exclusively for use by licensed users of Cognitier software. The information in this document is subject to change without notice.

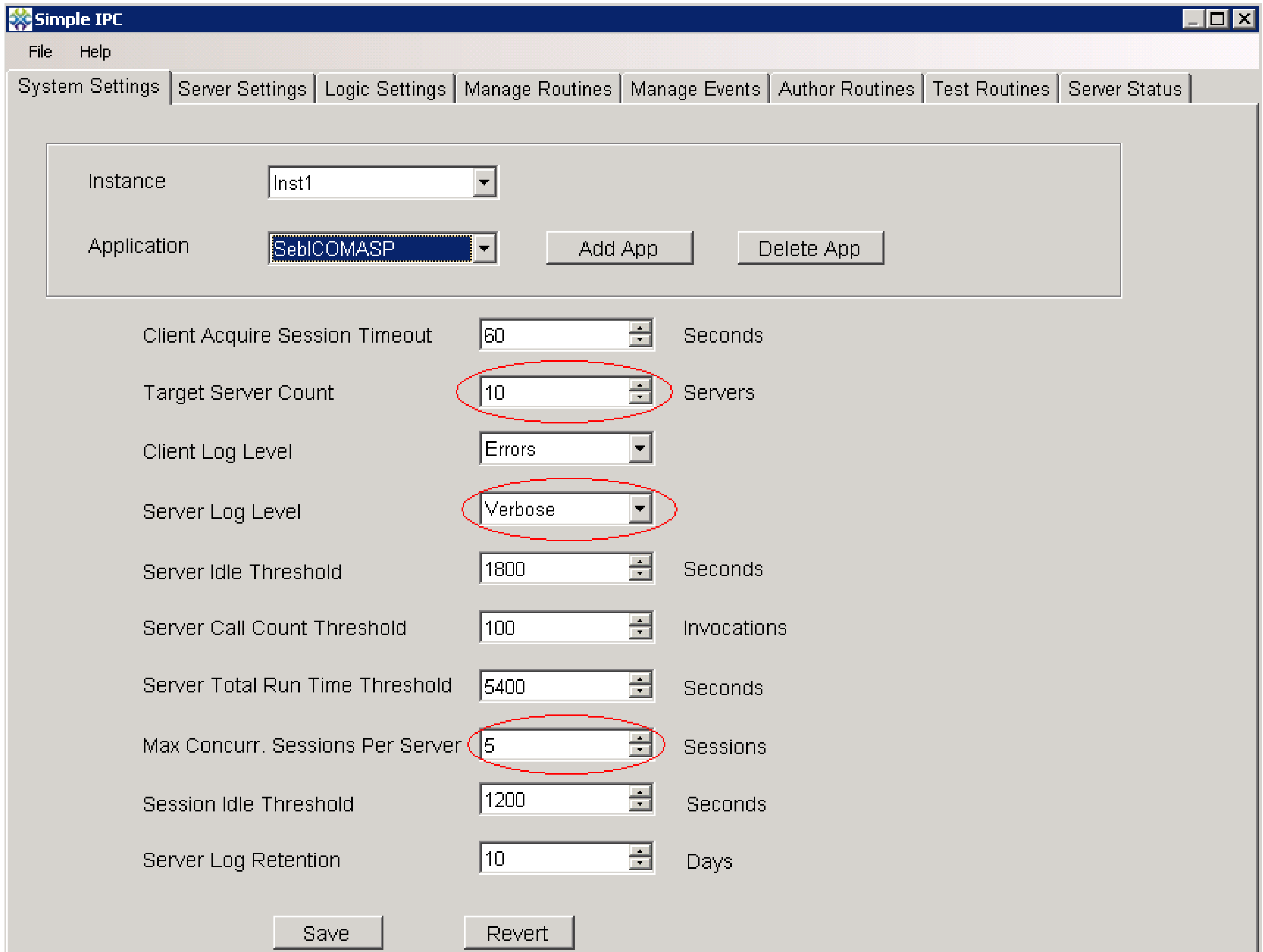
Siebel COM Data Control ASP Page Sample

Incorporating COM components into a web application can be challenging because the identity of the calling user, the threading model, and the number of concurrent users may be different from the situation where the same COM components are used in a traditional desktop application. In this example, we will incorporate a COM component into a SimpleIPC application. This will allow us to keep the environmental conditions the same whether the calling application is a Windows form or a web application.

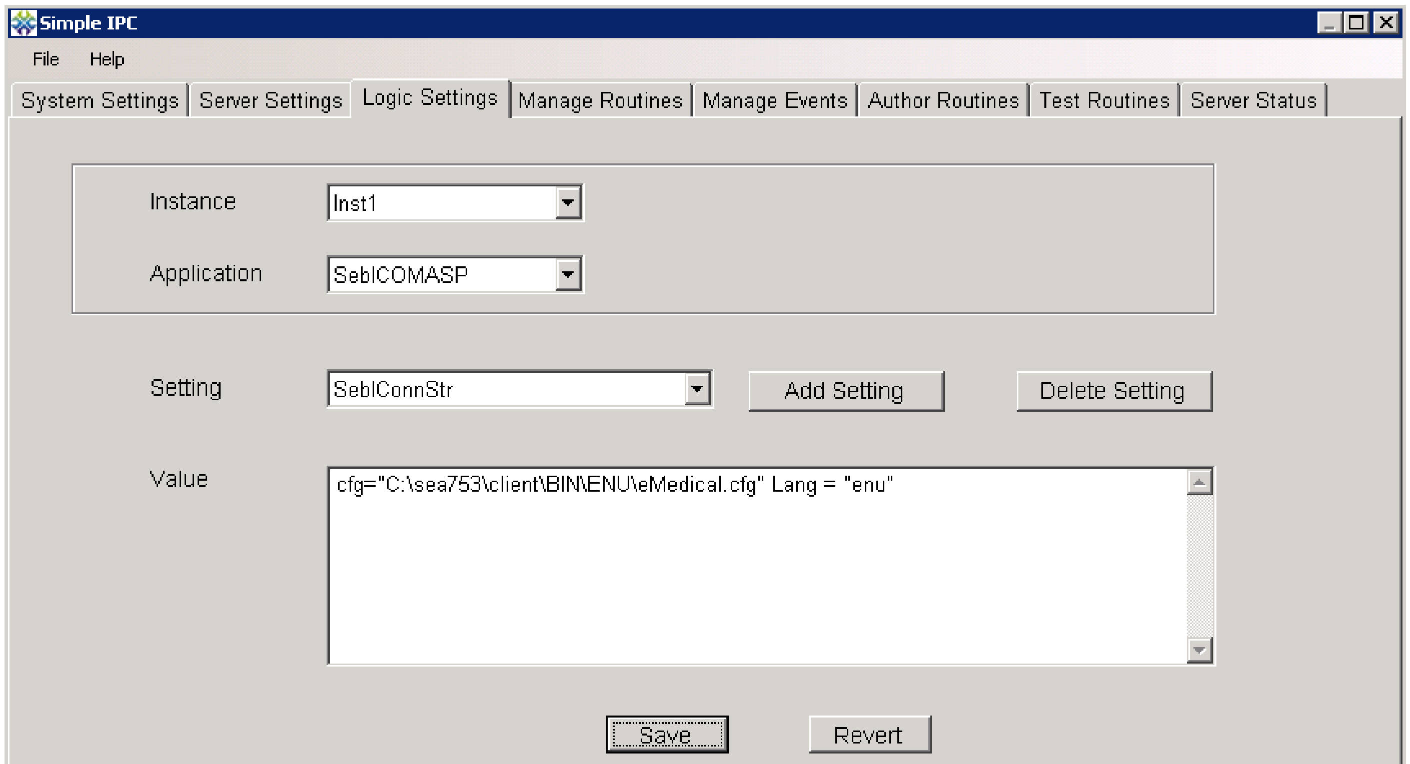
In this example, we will use the Siebel COM Data Control in “thick client” mode. We will provide a connection string that identifies a local configuration file. The Siebel thick client is installed locally, and the Siebel COM Data Control will use local object definitions and make its own connection to the Siebel Repository – rather than connecting to the Siebel server. In this way, we will be putting a load on our web server which would otherwise have been placed on the Siebel server. When the Siebel COM Data Control is used in thick client mode, there can only be one instance of the component per process, and the component can only service one client request at a time. In this exercise, we will enter server code to handle incoming requests serially, and we will configure our application to allow multiple IPC servers. In this way we will still be able to accommodate a concurrent load of web users.

The sample code omits many error-checking operations in order to make the code more readable. Be aware of word-wrapping if you copy and paste sample code from this document.

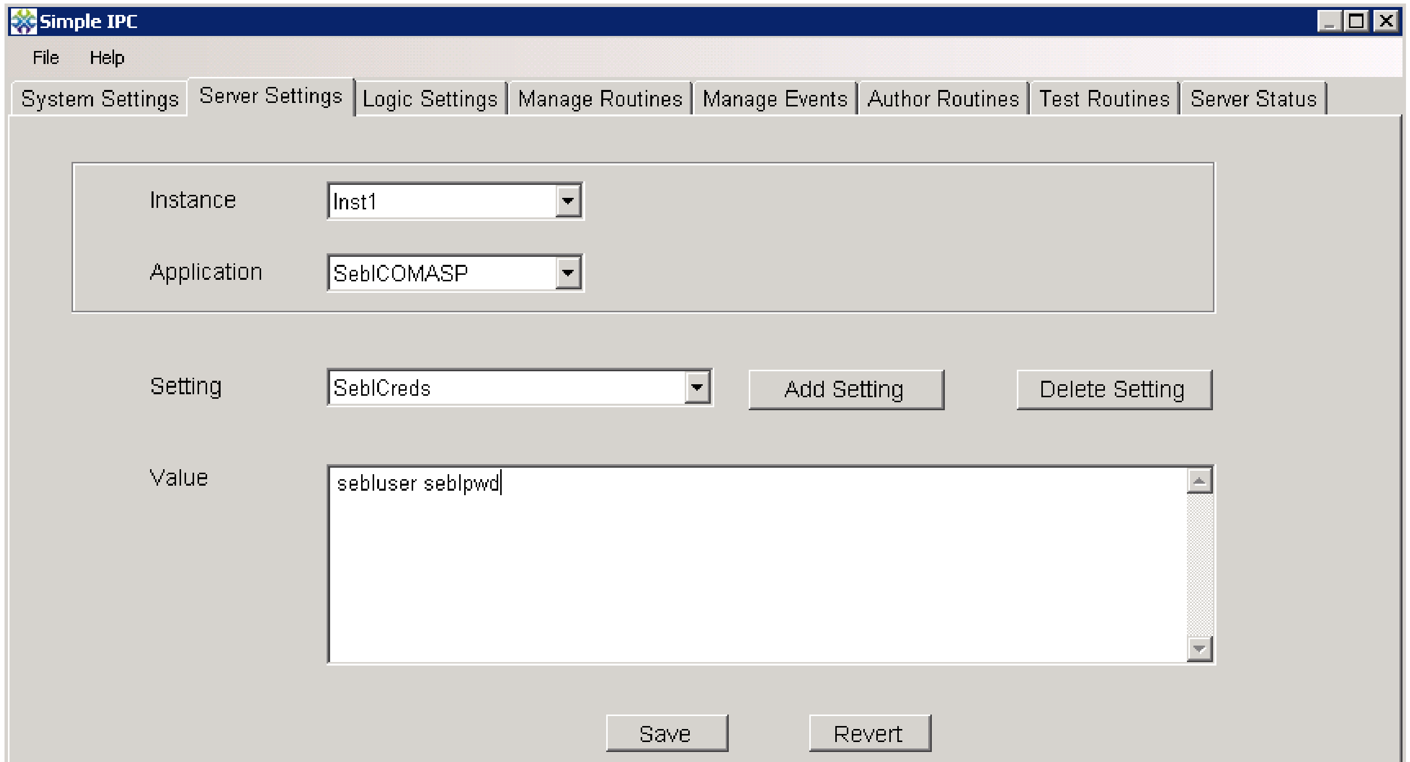
Step 1: Go to the System Settings tab in the CTConsole and create a new application in SimpleIPC for this exercise. Call it SeblCOMASP. Designate that there may be 10 servers for this Instance/Application combination, and five sessions per server. Set the server log level to verbose.



Step 2: Create a Logic Setting for this new application with the connection string to use when connecting to Siebel via the COM Data Control in thick client mode. Go to the Logic Settings tab and toggle the Instance selection in order to refresh the list of Applications. Select the SeblCOMASP application and enter the new Logic Setting. Call the Logic Setting SeblConnStr, enter a value appropriate for your environment, and save the value.



Step 3: Create a Server Setting for this new application with the credentials to use when connecting to Siebel. Go to the Server Settings tab and toggle the Instance selection in order to refresh the list of Applications. Select the SeblCOMASP application and enter the new Server Setting. We could create separate settings for the username and password, but in this case we will combine both values into one setting delimited by the space character. Call the Server Setting SeblCreds, enter a value appropriate for your environment, and save the value.



Step 4: Go to the Author Routines tab and create the routine that will execute when an IPC server starts. Create a new routine called SeblASPServStart using the BasicServerOp1 template. Add a reference to C:\WINNT\Microsoft.NET\Framework\v2.0.50727\System.XML.dll (adjust the path for your environment). Enter the following code to start a worker thread which connects to Siebel and then processes incoming client requests.

```
import System;
import System.Collections;
import System.Reflection;
import System.Xml;

[assembly:AssemblyVersion("1.0.0.0")]

public class SeblASPServStart implements CTSHost.IBasicServerOp1
{
  static var mServerContextAccessor: CTSHost.ServerContextAccessor;
  public function RunRoutine(oServerContextAccessor:
CTSHost.ServerContextAccessor) : System.Boolean
  {
     SeblASPServStart.mServerContextAccessor = oServerContextAccessor;
     var oLogUtil: CTCommon.LogUtil;
```

```
    var bRet: System.Boolean;

    oLogUtil = CTCommon.LogUtil.Instance;
    bRet = false;
    try
    {
     //Start a worker thread to keep running through the server's lifespan
     var oWorkerThread: System.Threading.Thread = new
System.Threading.Thread(System.Threading.ThreadStart(ProcessInboundRequests));
     oWorkerThread.SetApartmentState(System.Threading.ApartmentState.STA);
     oWorkerThread.Start();

     bRet = true;
    }
    catch (e)
    {
     //Write errors to the server log file
     oLogUtil.LogOutput(CTCommon.Constants.LogLevel.Errors,
CTCommon.Constants.LogOpType.RoutineExecution, e);
    }
    return bRet;
  } //end function

  static function ProcessInboundRequests()
  {
    var oLogUtil: CTCommon.LogUtil;
    oLogUtil = CTCommon.LogUtil.Instance;
    try
    {
     //Cast the argument object back to the ServerContextAccessor
     //var oServerContextAccessor: CTSHost.ServerContextAccessor =
System.Convert.ChangeType(oParamObj, CTSHost.ServerContextAccessor);
     var oServerContextAccessor: CTSHost.ServerContextAccessor =
SeblASPServStart.mServerContextAccessor;

     //Put a boolean variable in the context to tell us when to end thread
execution
```

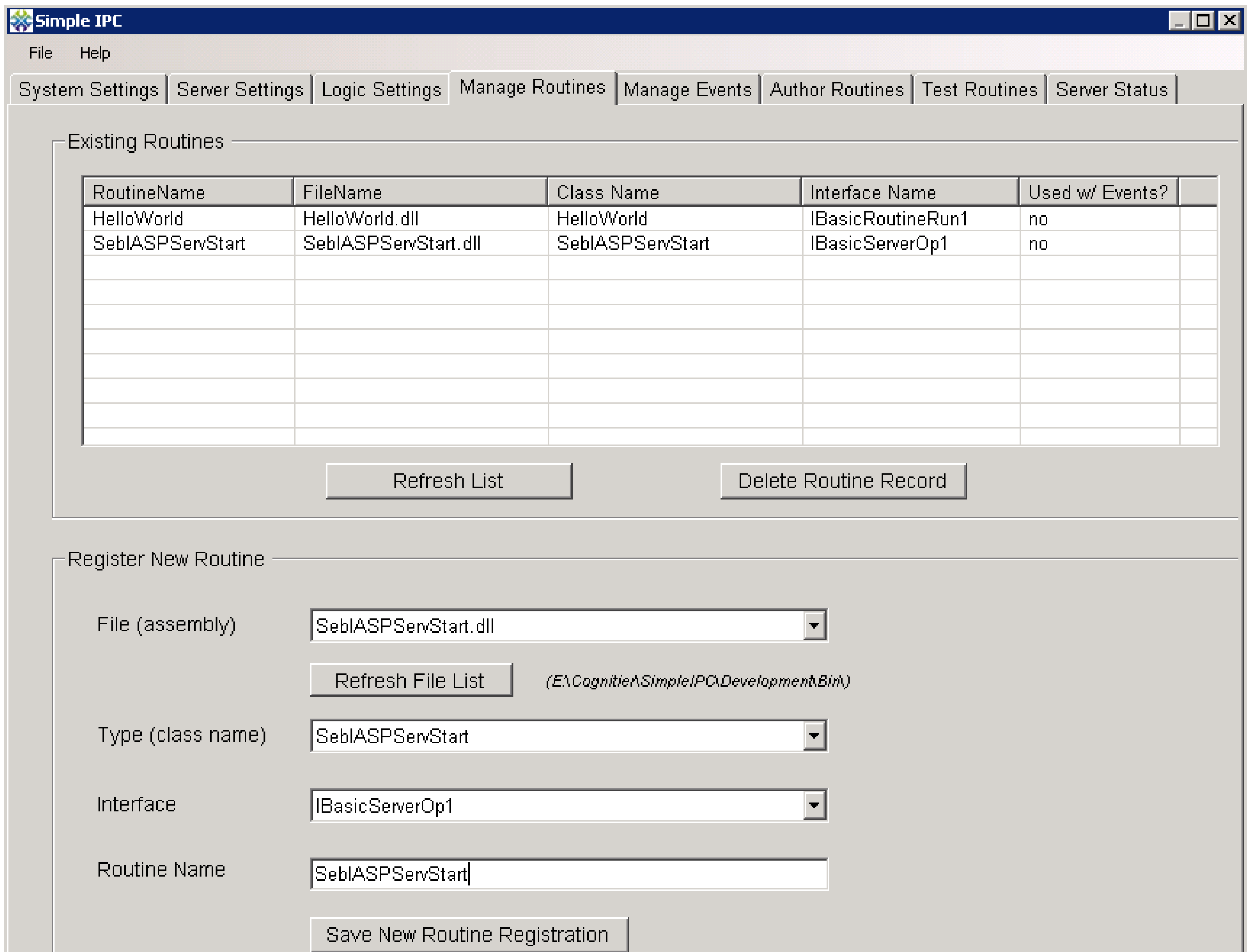
```
      var bServerShuttingDown: System.Boolean = false;
      oServerContextAccessor.SetServerContextItem("ServerShuttingDown",
bServerShuttingDown);

      //Instantiate the COM Data Control
      var oSiebelApp = new ActiveXObject("SiebelDataControl.SiebelDataControl");
      //Get the Logic Setting value with the connection string
      var sConnStr: System.String =
oServerContextAccessor.GetLogicSetting("SeblConnStr");
      //Get the Server Setting value with the concatenated username and password
      var sCreds: System.String =
oServerContextAccessor.GetServerSetting("SeblCreds");
      //Parse the concatenated credentials
      var sCredArray: System.String[] = sCreds.Split(" ");
      //Make the connection to Siebel
      oSiebelApp.Login(sConnStr, sCredArray[0], sCredArray[1]);

      //Get a reference to the Employee Business Object
      var oEmpBO = oSiebelApp.GetBusObject("Employee");
      //Get a reference to the Employee Business Component
      var oEmpBC = oEmpBO.GetBusComp("Employee");

      //Prepare variables for handling requests and return strings
      var sRequestID: System.String = String.Empty;
      var sRequestPayload: System.String = String.Empty;
      var oXMLDoc:  XmlDocument = null;
      var oRecordXMLDoc:  XmlDocument = null;
      var oRecordXMLNode:  XmlNode= null;
      var iRet: int = 0;
      var iRecordCount: int = 0;

      //Loop while the server doesn't appear to be shutting down
      var htServerContextItems: Hashtable;
      htServerContextItems = oServerContextAccessor.GetServerContextItems();
      bServerShuttingDown =
System.Boolean(htServerContextItems["ServerShuttingDown"]);
```

```
      while (bServerShuttingDown== false)
      {
            //Look for new requests queued in the server context
            sRequestID = String.Empty;
            for (var sKeyName in htServerContextItems.Keys)
            {
             if (sKeyName.StartsWith("REQ::"))
             {
                  sRequestID = sKeyName;
                  break;
             }
            }

            if (sRequestID.Length > 0)
            {
             //Get the request details from the server context item and then take
the item out of the context
             //In the present example, the request details don't matter because
we always run the same query
             sRequestPayload = System.String(htServerContextItems[sRequestID]);
             oServerContextAccessor.RemoveServerContextItem(sRequestID);
             //For our example, run the query for Employees without constraints
             oEmpBC.ClearToQuery();
             oEmpBC.SetViewMode(3);
             oEmpBC.ActivateField("First Name");
             oEmpBC.ActivateField("Last Name");
             iRet = oEmpBC.ExecuteQuery(1);
             //Create a base XML document - many ways to do this
             oXMLDoc= new XmlDocument();
             oXMLDoc.LoadXml("<?xml version=\"1.0\" encoding=\"utf-8\"
?><Employees></Employees>");
             iRet = oEmpBC.FirstRecord();
             iRecordCount = 0;
             while (iRet != 0 && iRecordCount < 10)
             {
                  oRecordXMLDoc = new XmlDocument();
```

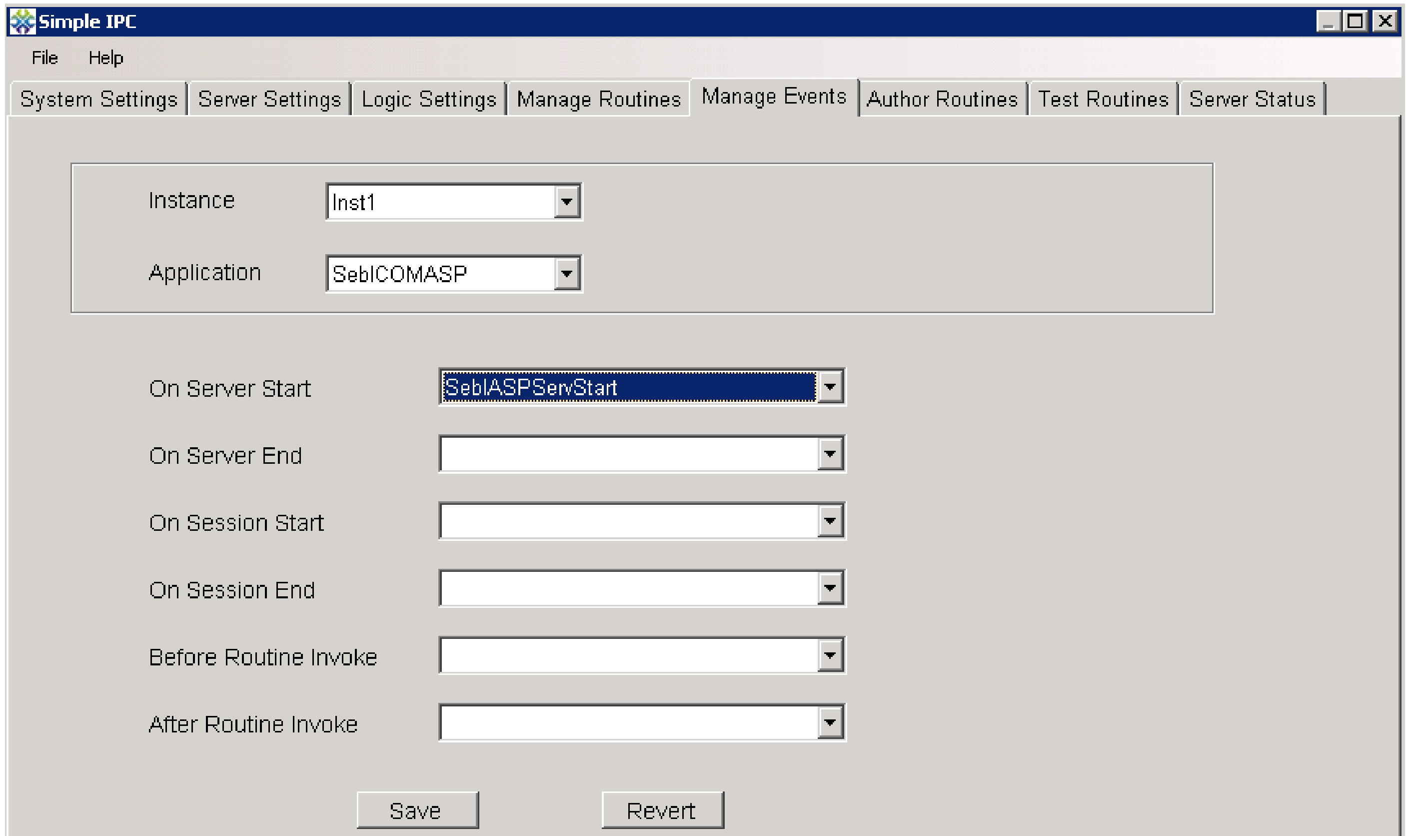
```
                oRecordXMLDoc.LoadXml("<Employee><FirstName /><LastName
/></Employee>");

oRecordXMLDoc.SelectSingleNode("//Employee/FirstName").InnerText =
oEmpBC.GetFieldValue("First Name");
                oRecordXMLDoc.SelectSingleNode("//Employee/LastName").InnerText
= oEmpBC.GetFieldValue("Last Name");
                oRecordXMLNode = XmlNode(oRecordXMLDoc.DocumentElement);

oXMLDoc.SelectSingleNode("//Employees").AppendChild(oXMLDoc.ImportNode(oRecordX
MLNode, true));
                iRet = oEmpBC.NextRecord();
                iRecordCount++;
           }

           //Put a response corresponding to the request into the server
context with the resulting XML string
           oServerContextAccessor.SetServerContextItem("RESP" +
sRequestID.Substring(3, sRequestID.Length - 3), oXMLDoc.InnerXml);
           }
           System.Threading.Thread.Sleep(1000);
           htServerContextItems =
oServerContextAccessor.GetServerContextItems();
           bServerShuttingDown =
System.Boolean(htServerContextItems["ServerShuttingDown"]);
      }

      //When the loop exits, perform cleanup operations - in production, do this
when an exception is thrown as well
      oEmpBC = null;
      oEmpBO = null;
      oSiebelApp.Logoff();
      oSiebelApp = null;
     }
     catch (e)
     {
      //Write errors to the server log file
```

```
      oLogUtil.LogOutput(CTCommon.Constants.LogLevel.Errors,
CTCommon.Constants.LogOpType.RoutineExecution, "Error in ProcessInboundRequests
thread: " + e);
      }
  } //end function
} //end class
```

Step 5: Register the routine we just created. Go to the Manage Routines tab of the CTConsole and click the Refresh File List button under the "File (Assembly") drop-down. Select the SeblASPServStart.dll dll, enter a name of SeblASPServStart and click Save New Routine Registration.



Step 6: Associate the server start-up routine with the “On Server Start” event for our application. Go to the Manage Events tab and toggle the Instance selection to refresh the list in the Applications drop-down. Select the SeblCOMASP application and associate the SeblASPServStart routine with the “On Server Start” event.



Step 7: Go to the Author Routines tab and create a new source code file with a name of SeblQueryEmps. Use the BasicRoutineRun1 template. Add a reference to C:\WINNT\Microsoft.NET\Framework\v2.0.50727\Microsoft.VisualBasic.dll (needed for some date/time functions). For this exercise, enter the following code to set a request into the server context to be picked up by the worker thread, and then periodically check for a response and return the text of the response in the output array.

```
import System;
import System.Collections;
import System.Reflection;
import System.Xml;
import Microsoft.VisualBasic;

[assembly:AssemblyVersion("1.0.0.0")]
```

```
public class SeblQueryEmps implements CTSHost.IBasicRoutineRun1
{
  public function RunRoutine(oServerContextAccessor:
CTSHost.ServerContextAccessor, oSessionContextAccessor:
CTSHost.SessionContextAccessor, oCallParamsAccessor:
CTSHost.CallParamsAccessor) : System.Boolean
  {
     var REQUESTTIMEOUT: int = 120;
     var oLogUtil: CTCommon.LogUtil;
     var bRet: System.Boolean;

     oLogUtil = CTCommon.LogUtil.Instance;
     bRet = false;
     try
     {
      //Make up a request id from the session id and the tick count
      //Put it in the server context with an arbitrary value for extra request
information
      var sBaseRequestID: System.String = oSessionContextAccessor.SessionID +
"::" + System.Environment.TickCount;
      var sRequestID: System.String = "REQ::" + sBaseRequestID;
      oServerContextAccessor.SetServerContextItem(sRequestID,
"extra_request_info");

      //Wait for the server's worker thread to process the request and put the
response in the server context
      var sResponseID: System.String = "RESP::" + sBaseRequestID;
      var sResponsePayload: System.String = String.Empty;
      var dStart: System.DateTime = System.DateTime.Now.ToUniversalTime();
      var iDateDiffSecs: int = 0;
      var htServerContextItems: Hashtable;
      while ( iDateDiffSecs < REQUESTTIMEOUT)
      {
           htServerContextItems =
oServerContextAccessor.GetServerContextItems();
           if (htServerContextItems[sResponseID] != null)
           {
```

```
            //If the response is present, remove it from the context and exit
the loop
            sResponsePayload = System.String(htServerContextItems[sResponseID]);
            oServerContextAccessor.RemoveServerContextItem(sResponseID);
            break;
           }
           System.Threading.Thread.Sleep(1000);
           iDateDiffSecs = int(DateAndTime.DateDiff(DateInterval.Second, dStart,
System.DateTime.Now.ToUniversalTime()));
      }

      //Return the response to the calling program - it could be empty if the
request timed out
      oCallParamsAccessor.OutputArgs.Add(sResponsePayload);

      bRet = true;
      }
      catch (e)
      {
      //Set errors into errors collection - must be strings
      oCallParamsAccessor.OutputErrors.Add(e.Message);
      //Also write errors to the server log file
      oLogUtil.LogOutput(CTCommon.Constants.LogLevel.Errors,
CTCommon.Constants.LogOpType.RoutineExecution, e);
      }
      return bRet
  } //end function
} //end class
```

Step 8: Register the routine we just created. Go to the Manage Routines tab of the CTConsole and click the Refresh File List button under the "File (Assembly") drop-down. Select the SeblQueryEmps.dll dll, enter a name of SeblQueryEmps and click Save New Routine Registration.

Step 9: Go to the Author Routines tab and create the routine that will execute when an IPC server exits. Create a new routine called SeblASPServStop using the BasicServerOp1 template. Enter the following code to set a Boolean variable in the server context which indicates to the worker thread that it should log off from Siebel and exit.

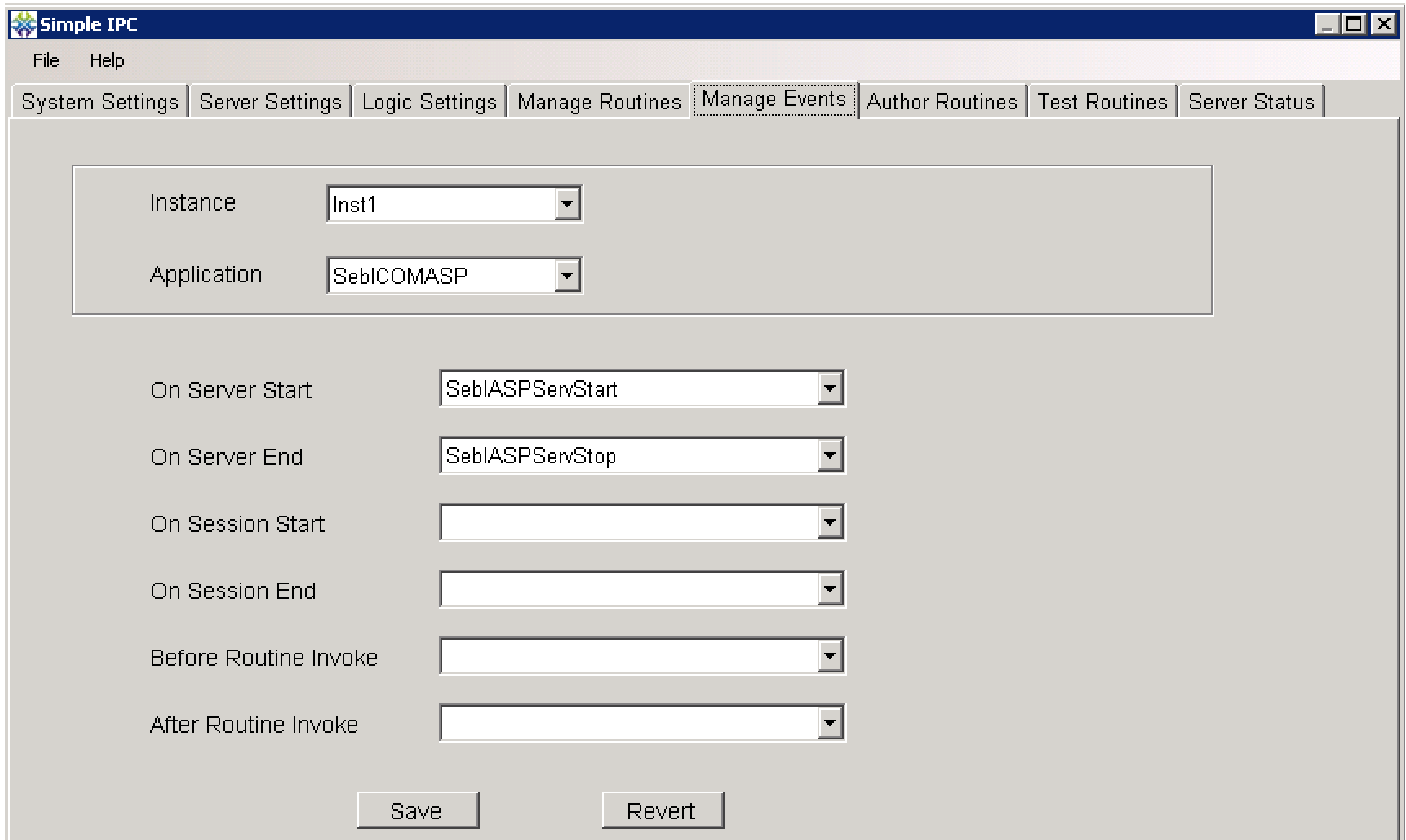
```
import System;
```

```
import System.Collections;
import System.Reflection;

[assembly:AssemblyVersion("1.0.0.0")]

public class SeblASPServStop implements CTSHost.IBasicServerOp1
{
  public function RunRoutine(oServerContextAccessor:
CTSHost.ServerContextAccessor) : System.Boolean
  {
     var oLogUtil: CTCommon.LogUtil;
     var bRet: System.Boolean;

     oLogUtil = CTCommon.LogUtil.Instance;
     bRet = false;
     try
     {
      //Set the boolean context variable to true to let the worker thread know
the server is exiting
      var bServerShuttingDown: System.Boolean = true;
      oServerContextAccessor.SetServerContextItem("ServerShuttingDown",
bServerShuttingDown);

      //Wait for a minute to let the worker thread finish what it is doing and
then exit
      System.Threading.Thread.Sleep(60000);

      bRet = true;
     }
     catch (e)
     {
      //Write errors to the server log file
      oLogUtil.LogOutput(CTCommon.Constants.LogLevel.Errors,
CTCommon.Constants.LogOpType.RoutineExecution, e);
     }
     return bRet;
  } //end function
```

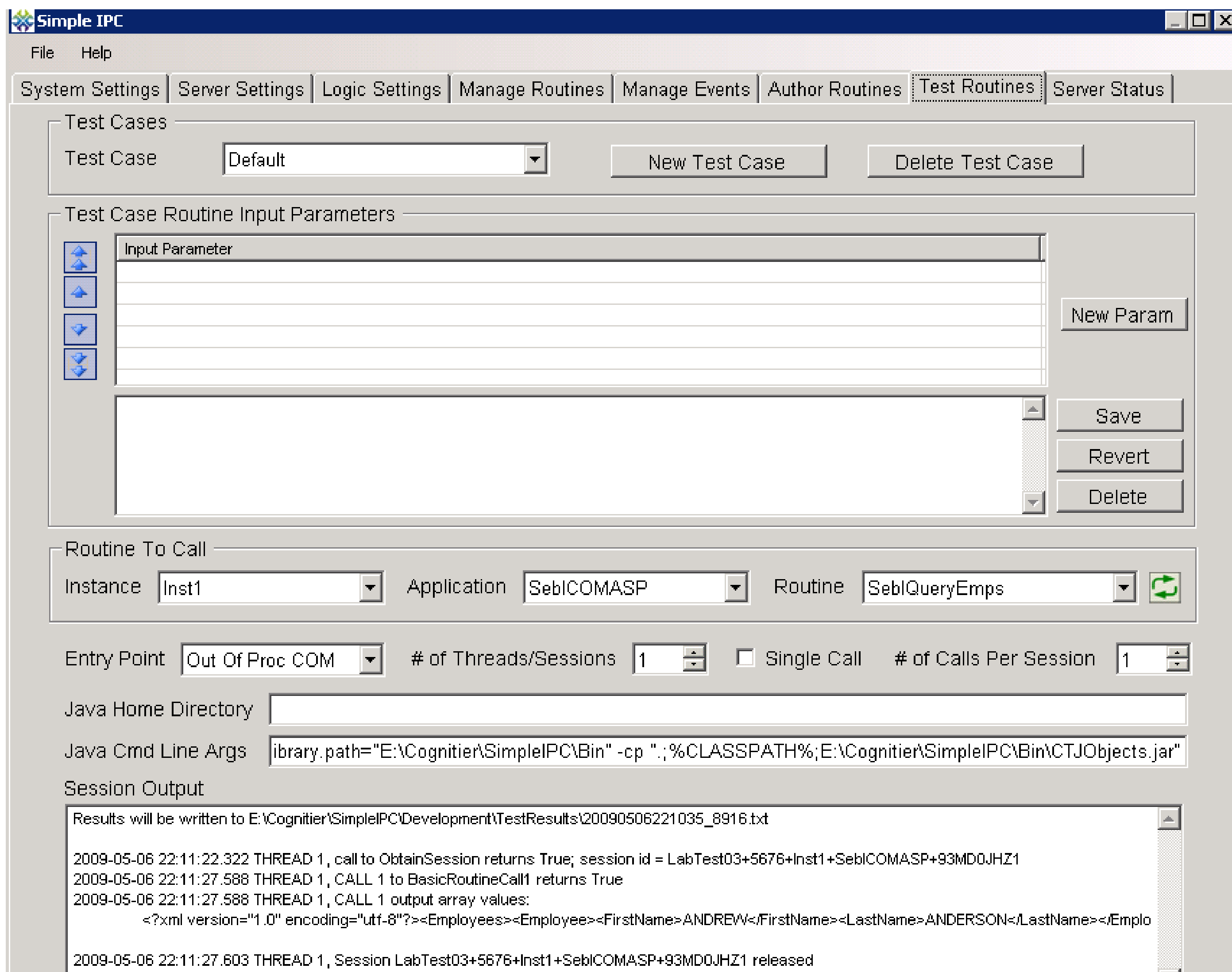
```
} //end class
```

Step 10: Register the routine we just created. Go to the Manage Routines tab of the CTConsole and click the Refresh File List button under the "File (Assembly") drop-down. Select the SeblASPServStop.dll dll, enter a name of SeblASPServStop and click Save New Routine Registration.

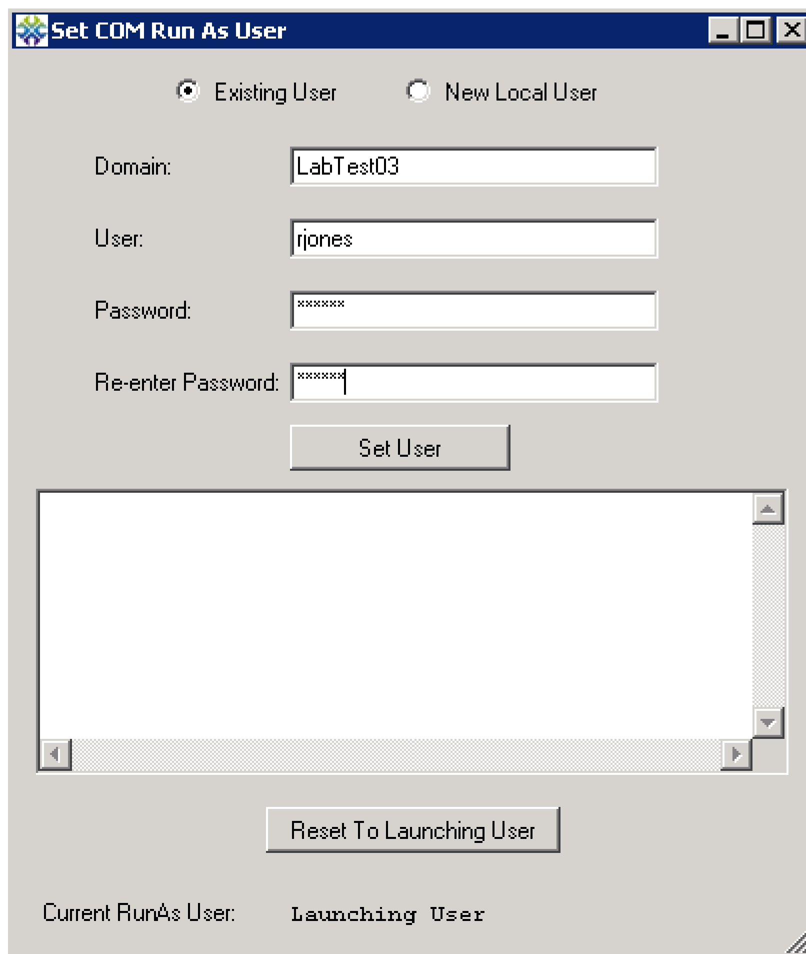
Step 11: Associate the server exit routine with the "On Server End" event for our application. Go to the Manage Events tab and toggle the Application selection to "App1" and then "SeblCOMASP" to refresh the selections in the routine drop-downs. Associate the SeblASPServStop routine with the "On Server End" event of the SeblCOMASP application.



Step 12: Test your work up to this point. Go to the Test Routines tab and toggle the Instance selection from Inst1 to Inst2 and back to Inst1 to refresh the list of Applications. Select the SeblCOMASP application. Click the refresh button to the right of the Routines drop-down and then select the SeblQueryEmps routine. Designate one session making one call to the routine. The other selections on the form are not important. Click the Start button to run the test and make sure you obtain valid values in the result.

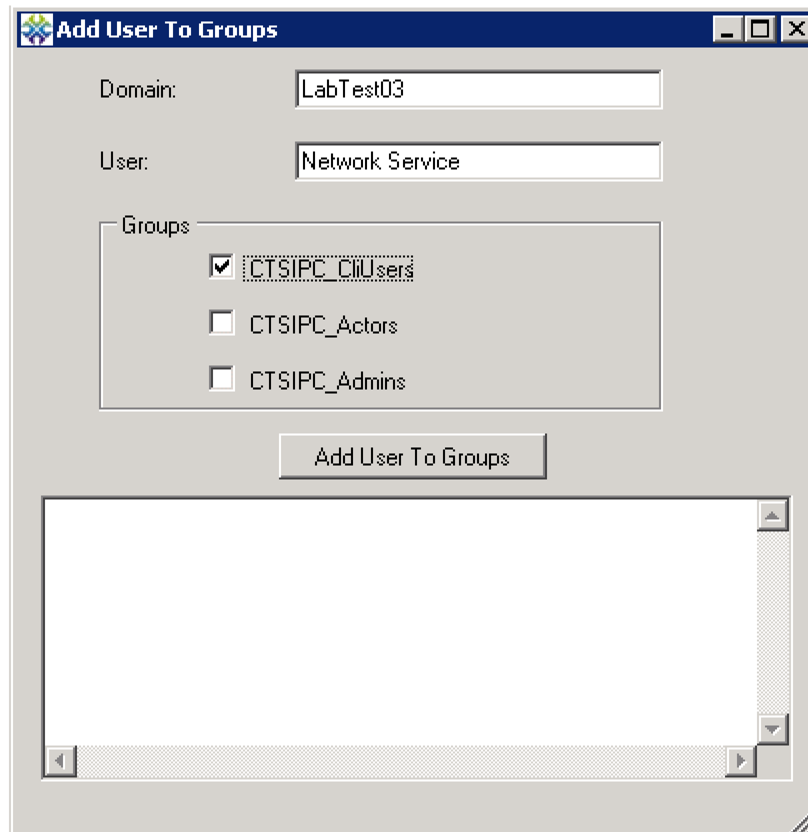


Step 13: A web application is an example of a situation where we want the client application and the IPC server running under different user accounts. We want the IPC server to run under a fixed account with membership in the CTSIPC_Actors group, and we want the user account under which the web application is running to be a member of the CTSIPC_CliUsers group. If you have not already done so, configure the out-of-process COM servers to run under a fixed identity. Select File>>Set COM Run As User from the CTConsole menu. Enter the credentials for either an existing user account, or create a new local user.



Since we have changed the COM server configuration, the computer should be rebooted at this point. When the computer is booted up again, go to the Test Routines tab of the CTConsole and select the HelloWorld routine. Run a test with the Out Of Proc COM Entry Point with the Inst1 instance, and then repeat the test with the Inst2 instance. This will verify that the COM server registration information is intact after the configuration change.

Step 14: In this exercise, the client application is actually the web server application pool. The application pool is running under the “Network Service” account. We will need to make the “Network Service” account a member of the CTSIPC_CliUsers group. However, we do not want to grant any further permissions to that user account. From the CTConsole menu, choose File>>Add User To Groups. Add the “Network Service” user to the CTSIPC_CliUsers group. Also add the IUSR_[Machine Name] account to the CTSIPC_CliUsers group. After making these group membership changes, recycle the IIS application pool you intend to use with your ASP page.



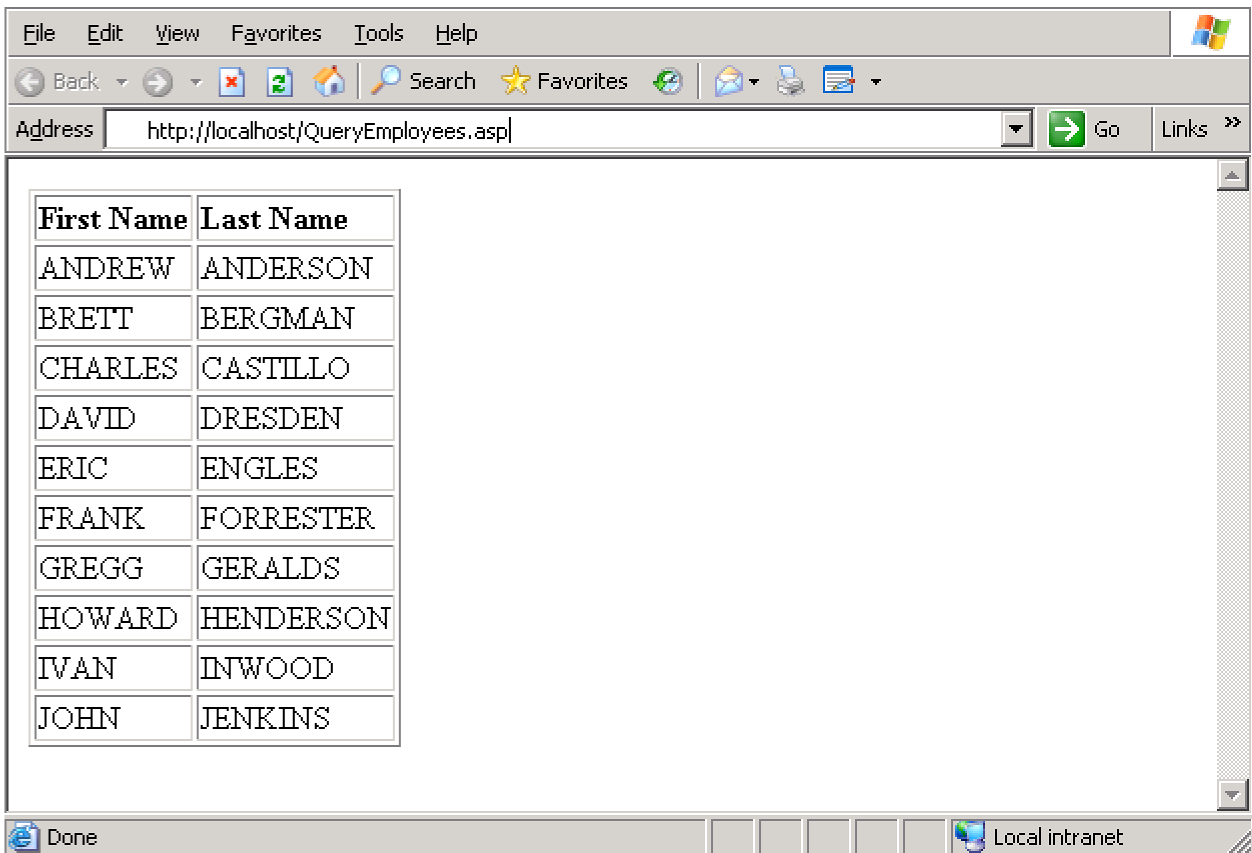
Step 15: Use Notepad or another text editor to create an Active Server Page which will invoke the SeblQueryEmps routine. Call your new file QueryEmployees.asp. Place your ASP page file in a directory for a web application that is serviced by the application pool you previously recycled. For this exercise, use the following ASP code to invoke the SeblQueryEmps routine, parse the returned XML, and display the results in a simple HTML table.

```
<%@LANGUAGE="VBSCRIPT"%>
<HTML>
<%
'Declare variables
Dim oXMLDoc, sXML, oNodeList, iCount, iNodeCount, sHTML
dim oComp, sSessionID, iRejectCode, sRoutine, bRet
dim oInputArray, oOutputArray, oErrorArray, oWarningArray

'Initialize variables
sAppID = "SeblCOMASP"
sRoutine = "SeblQueryEmps"
sSessionID = ""
iRejectCode = 0
SESSIONINVALIDORTIMEOUT = "12001"
```

```
SERVERCOMMUNICATIONERR = "12004"
ERRRESUMINGSESSION = "12005"
sXML = ""
sHTML = ""

'Get reference to COM object
Set oComp = CreateObject("CTOutProcCOMInst1.CTSessionComp1")

'Make call to obtain a session
bRet = oComp.ObtainSession(sSessionID, sAppID, iRejectCode)

'Make call to BasicRoutineCall1
bRet = oComp.BasicRoutineCall1(sSessionID, sAppID, sRoutine, oInputArray,
oOutputArray, oErrorArray, oWarningArray)

If UBound(oOutputArray) >= 0 Then
      sXML = oOutputArray(0)
End If

If Len(sXML) > 0 Then
      sHTML = "<table border=1><tr><td><B>First Name</B></td><td><B>Last
Name</B></td></tr>"
      Set oXMLDoc = Server.CreateObject("Msxml2.DOMDocument")
      oXMLDoc.Async = false
      oXMLDoc.loadXML(sXML)
      Set oNodeList = oXMLDoc.selectNodes("//Employees/Employee")
      iNodeCount = oNodeList.Length
      iCount = 0
      While iCount < iNodeCount
            sHTML = sHTML & "<tr><td>" & oNodeList(iCount).childNodes(0).text &
"</td><td>" & oNodeList(iCount).childNodes(1).text & "</td></tr>"
            iCount = iCount + 1
      Wend
      Set oXMLDoc = Nothing
      sHTML = sHTML & "</table>"
      Response.Write sHTML
Else
```

```
      Response.Write "<B>No Employee Records Returned</B>"
End If

oComp.ReleaseSession(sSessionID)
Set oErrorArray = Nothing
Set oWarningArray = Nothing
Set oOutputArray = Nothing
Set oComp = Nothing
%>
</HTML>
```

Step 16: Use Internet Explorer to invoke your ASP page and ensure that valid results are returned. If you do not receive the expected results then check the NT Event Viewer for permissions-related errors. Most importantly, the account under which your web application is acting must have access to the SimpleIPC COM server and the IPC channel (which should be accomplished via membership in the CTSIPC_CliUsers group).



First Name	Last Name
ANDREW	ANDERSON
BRETT	BERGMAN
CHARLES	CASTILLO
DAVID	DRESDEN
ERIC	ENGLES
FRANK	FORRESTER
GREGG	GERALDS
HOWARD	HENDERSON
IVAN	INWOOD
JOHN	JENKINS

Step 17: We will want to be able to simulate many users invoking the ASP page at nearly the same time. Create a new routine for this purpose. Go to the Author Routines tab of the CTConsole and create a new source code file. Call

the new routine ASPEmpQuery and use the BasicRoutineRun1 template. For this exercise, enter the following code to invoke the ASP page and put the text contents of the returned page into the output array.

```
import System;
import System.Collections;
import System.Reflection;

[assembly:AssemblyVersion("1.0.0.0")]

public class ASPEmpQuery implements CTSHost.IBasicRoutineRun1
{
  public function RunRoutine(oServerContextAccessor:
CTSHost.ServerContextAccessor, oSessionContextAccessor:
CTSHost.SessionContextAccessor, oCallParamsAccessor:
CTSHost.CallParamsAccessor) : System.Boolean
  {
     var oLogUtil: CTCommon.LogUtil;
     var bRet: System.Boolean;

     oLogUtil = CTCommon.LogUtil.Instance;
     bRet = false;

     var oResponse: System.Net.HttpWebResponse = null;
     var oResponseReader: System.IO.StreamReader = null;
     try
     {
      //Create a web request object for invoking our ASP page
      var oRequest: System.Net.HttpWebRequest = null;
      var sURL: System.String = "http://localhost/QueryEmployees.asp";
      var sPageContents: System.String = String.Empty;
      oRequest = System.Net.HttpWebRequest(System.Net.WebRequest.Create(sURL));
      oRequest.KeepAlive = false;
      oRequest.Timeout = 120000;

      //Invoke the web page and read the page contents into a string variable
      oResponse = System.Net.HttpWebResponse(oRequest.GetResponse());
      oResponseReader = new
System.IO.StreamReader(oResponse.GetResponseStream());
      sPageContents = oResponseReader.ReadToEnd();
      oResponseReader.Close();
      oResponse.Close();
```

```
      //Return the contents of the ASP page to the calling program in the output
array
      oCallParamsAccessor.OutputArgs.Add(sPageContents);

      bRet = true;
      }
      catch (e)
      {
      //Set errors into errors collection - must be strings
      oCallParamsAccessor.OutputErrors.Add(e.Message);

      //Also write errors to the server log file
      oLogUtil.LogOutput(CTCommon.Constants.LogLevel.Errors,
CTCommon.Constants.LogOpType.RoutineExecution, e);

      //If we caught an exception, confirm that we're closing the response and
response reader
      if (oResponseReader != null)
      {
           oResponseReader.Close();
      }
      if (oResponse != null)
      {
           oResponse.Close();
      }
      }
      return bRet;

  } //end function
} //end class
```

Step 18: Register the routine we just created. Go to the Manage Routines tab and click the "Refresh List" button so that our new dll will appear in the dlls drop-down. Select ASPEmpQuery.dll, enter a routine name of ASPEmpQuery, and save it.

Step 19: Test your work. Go to the Test Routines tab of the CTConsole. Select the "App1" application. It is important not to select the SeblCOMASP application because we want to simulate a user invoking the ASP page and we do not need to have the server start-up routine execute for this part of the test. Click the Refresh button to the right of the Routines drop-down and select the ASPEmpQuery routine. For this exercise, it does not matter which Entry Point is selected. Designate one thread/session making one call per session. Click the “Start” button and observe the output in the text area beneath the button. The output should be the HTML contents of the ASP page. If

this test produces the expected results, then increase the load such that 50 threads are each making 10 invocations of the routine (a total of 500 requests for the new ASP page). Review the output displayed in the text area in the CTConsole, and also check the IPC server logs to see if any errors occurred. Likewise check memory and CPU utilization on the machine.