crystal decisions™

Crystal Enterprise 8 Customization

A Lesson in Using the Report Plug-in for Scheduling

Contents

Learning Objectives

This document describes the *ReportPlugin* object and how to use it in Crystal Enterprise's Crystal Server Pages (CSP) to get and set report object properties. It will serve as a lesson and will walk you through developing an "enhanced" scheduling application that uses the features exposed by the *ReportPlugin* object.

Requirements for this Lesson

Before working through this document, you should be familiar with Crystal Enterprise's Crystal Server Pages (CSP) as well as have a clear understanding of the Crystal Enterprise product. This document also requires a good understanding of Microsoft Active Server Pages (ASP) development and some JavaScript knowledge. A good overview of the differences between ASP and CSP is included in the document, Crystal Enterprise 8 – ASP vs. CSP: Deployment Issues, which is available on the Crystal Decisions web site at:

http://support.crystaldecisions.com/communityCS/TechnicalPapers/CE8_ASP_VS_CSP.pdf

The Operating System

For the purposes of this lesson, Crystal Enterprise 8 should be installed on a computer running Microsoft Windows NT 4 or 2000, Server or Workstation. Some of the exercises included will be easier to complete when using the Server version of the operating system. The operating system should have the latest service packs applied as well as a supported web server. The web server can be either Microsoft Internet Information Server 3.x\4.x\5.x or Netscape Enterprise Server 3.x\4.x. Installing Microsoft Office 97 or 2000 or at least Microsoft Access 97 or 2000 will be helpful. For more information on what platforms are supported with Crystal Enterprise 8, please refer to *Platforms.txt* located in the root directory of the CE CD.

A Complete Crystal Enterprise 8 Installation

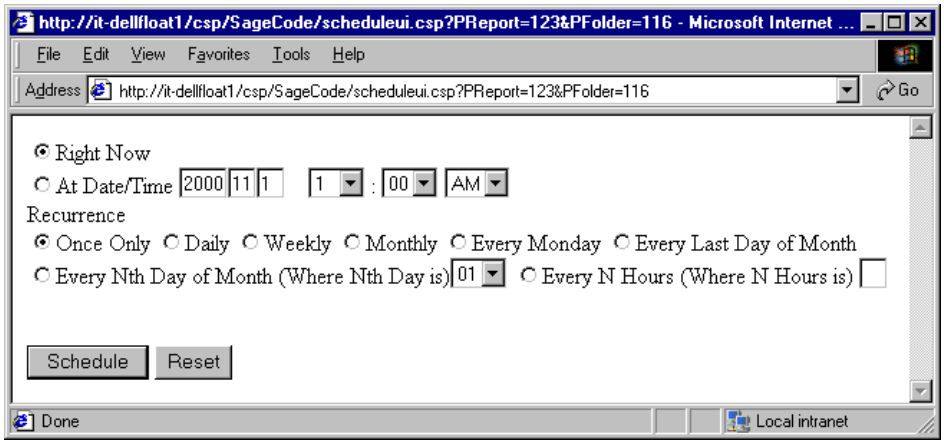
The computer you are working at should have a complete installation of Crystal Enterprise. Ideally, the Crystal APS will be using the default Microsoft SQL Server 7 to store the system database.

Scheduling

The scheduling page will prove to be something of a challenge due to all the factors that need to be considered when scheduling a report. Scheduling can be immediate, or set for a specific date and time. Therefore, we must provide the user with that choice. If a report contains parameter fields, either we can use the values currently set for the report, or we will need to collect values from the user using parameters. If the parameter contains default values, we will need to allow the user to select from those values. If the report requires database logon information we can use the values stored with the report or prompt the user to provide new database logon credentials. These are just a few of the concerns a developer of a scheduling page needs to consider.

For this lesson, we will build a scheduling page in stages. We will code and then test to make certain the desired functionality is working correctly before adding functionality that is more advanced. We will need to build a user interface (UI) page as well as an actual scheduling page. Starting simply, we will ask the user for the date and time for the object to be scheduled. We will provide a "Right Now" option as well as a date\time option. We will call our UI page *ScheduleUI.csp* and our actual scheduling page *Schedule.csp*

Our first scheduling UI page will look like this:



Even though we are creating an HTML page to collect information for the user, you will notice that we have called the file *ScheduleUI.csp*. The reason that we are creating a CSP page is that we have to create some of the HTML dynamically. For example, we need to keep track of the ID of the report object we want to schedule. By using CSP, we can also show today's date in the year, month and day text boxes rather than leaving them empty. We'll leave the time drop downs at their default value of 1:00 AM.

ScheduleUI.csp Code and Analysis

```
<%
ParentReport = Request.QueryString("PReport")
ParentFolder = Request.QueryString("PFolder")

Response.Write("<FORM Method=Post Action=Schedule.csp?PReport=" & ParentReport &
"&PFolder=" & ParentFolder & ">")
%>

<INPUT type="radio" name="When" Value="Now" Checked>Right Now<BR>
<INPUT type="radio" name="When" Value="At Time">At Date/Time

<%
CurrentDate = Now
CurrentYear = Year(CurrentDate)
CurrentMonth = Month(CurrentDate)
CurrentDay = Day(CurrentDate)

Response.Write("<INPUT Size=3 Name='Year' Value=" + Cstr(CurrentYear) + ">")
Response.Write("<INPUT Size=1 Name='Month' Value=" + Cstr(CurrentMonth) + ">")
Response.Write("<INPUT Size=1 Name='Day' Value=" + Cstr(CurrentDay) + ">")
%>

   
<Select name=hours>
<Option>1<Option>2<Option>3<Option>4<Option>5<Option>6<Option>7<Option>8<Option>9
<Option>10<Option>11<Option>12
</Select>
:
<Select name=minutes>
<Option>00<Option>01<Option>02<Option>03<Option>04<Option>05<Option>06<Option>07
```

```
<Option>08<Option>09<Option>10<Option>11<Option>12<Option>13<Option>14<Option>15
<Option>16<Option>17<Option>18<Option>19<Option>20<Option>21<Option>22<Option>23
<Option>24<Option>25<Option>26<Option>27<Option>28<Option>29<Option>30<Option>31
<Option>32<Option>33<Option>34<Option>35<Option>36<Option>37<Option>38<Option>39
<Option>40<Option>41<Option>42<Option>43<Option>44<Option>45<Option>46<Option>47
<Option>48<Option>49<Option>50<Option>51<Option>52<Option>53<Option>54<Option>55
<Option>56<Option>57<Option>58<Option>59
</Select>
<Select name=AMPM>
<Option>AM<Option>PM
</Select>
<BR>
Recurrence<BR>
<INPUT TYPE="RADIO" NAME="frequency" VALUE="0" Checked>Once Only
<INPUT TYPE="RADIO" NAME="frequency" VALUE="2">Daily
<INPUT TYPE="RADIO" NAME="frequency" VALUE="3">Weekly
<INPUT TYPE="RADIO" NAME="frequency" VALUE="4">Monthly
<INPUT TYPE="RADIO" NAME="frequency" VALUE="6">Every Monday
<INPUT TYPE="RADIO" NAME="frequency" VALUE="7">Every Last Day of Month
<BR>
<INPUT TYPE="RADIO" NAME="frequency" VALUE="5">Every Nth Day of Month
(Where Nth Day is)<Select Name=NthDay>
<Option>01<Option>02<Option>03<Option>04<Option>05<Option>06<Option>07<Option>08
<Option>09<Option>10<Option>11<Option>12<Option>13<Option>14<Option>15<Option>16
<Option>17<Option>18<Option>19<Option>20<Option>21<Option>22<Option>23<Option>24
<Option>25<Option>26<Option>27<Option>28<Option>29<Option>30<Option>31
</select>
<INPUT TYPE="RADIO" NAME="frequency" VALUE="1">Every N Hours
(Where N Hours is) <INPUT Size=1 Name='NthHour'>
<BR><BR><BR>
<INPUT type="Submit" Value="Schedule">
<Input type=Reset>
</FORM>
```

Notice that there are very few lines of CSP code. The rest is HTML code. The bulk of the code is the <*Option*> tags used to populate the drop-down list boxes! Starting from the top, the CSP code is used to collect the ID values for the report and folder objects. This is pretty common to all CE pages. However, in this page we use those values to create an HTML Form tag. The opening <*form*> tag requires an "Action", which is the URL that the form tells the browser to call, and a method (*Get* or *Post*). We want our scheduling UI page to call the file "*Schedule.csp*" but we need to pass the ID of the report and folder as well. Therefore you will notice that the URL to *Schedule.csp* has the "*PReport="* and "*PFolder="* query string parameters set dynamically. This enables us to tell *Schedule.csp* which report object we are scheduling.

From there, we use VBScript date functions to obtain the current year, month and day. The "*Now*" function provides us with the exact date and time when it was called. Then use the *Year*, *Month* and *Day* functions to extract the respective pieces of the date that you want. Once the desired portions of the date have been extracted, we will create text boxes with the date values as their defaults.

When the user clicks the Schedule button on the *ScheduleUI.csp* page, all of the information that was filled in will be sent to our *Schedule.csp* page. Our *Schedule.csp* page will accept all of this information and use it to submit a schedule to the Crystal APS. The following is the code that we will use to submit the schedule, *Schedule.csp*.

Schedule.csp Code and Analysis

```
<%
ParentFolder = Request.Querystring("PFolder")
ParentReport = Request.Querystring("PReport")
```

```
'Retrieve Scheduling Info

SchedYear = Request.Form("Year")
SchedMonth = Request.Form("Month")
SchedDay = Request.Form("Day")
When = Request.Form("When")
SchedHours = Request.Form("Hours")
SchedMinutes = Request.Form("Minutes")
AMPM = Request.Form("AMPM")
Frequency = Request.Form("Frequency")
NthDay = Request.Form("NthDay")
NthHour = Request.Form("NthHour")

'Retrieve the logon cookie

apstoken = Request.Cookies("LogonToken")
If apstoken = "" then
      response.redirect("LogonForm.csp")
End If

'Create a session manager

Set SessionManager = Server.CreateObject("CrystalEnterprise.SessionMgr")

'Logon using the token.  Will fail if the token is no longer valid.


Set oEntepriseSession = SessionManager.LogonWithToken(apstoken)

Set oIStore = oEntepriseSession.Service("", "InfoStore")

Set cReports = oIStore.Query("Select SI_SCHEDULEINFO from CI_INFOOBJECTS Where SI_ID
= " & Cint(ParentReport))

Set oReporttoSchedule = cReports.Item(1)

Set SchedulingInfo = oReporttoSchedule.SchedulingInfo

If When <> "Now" then
If AMPM = "PM" then
SchedHours = Cint(SchedHours) + 12
Else
SchedHours = "0" + Cstr(SchedHours)
End If

SchedDate = SchedYear & "/" & SchedMonth & "/" & SchedDay & " " & SchedHours & ":" &
SchedMinutes & ":" & "00"
SchedulingInfo.BeginDate = Cdate(SchedDate)
Else
SchedulingInfo.RightNow = 1
End if

SchedulingInfo.Type = Cint(Frequency)

If Cint(Frequency) = 1 Then
SchedulingInfo.IntervalHours = NthHour
End If

If Cint(Frequency) = 5 Then
SchedulingInfo.IntervalNthDay
End If

oIStore.Schedule(cReports)
```

```
URL = "listinstances.csp?PReport=" + Cstr(ParentReport) + "&PFolder=" +
Cstr(ParentFolder)
Response.Redirect(URL)
%>
```

The *Schedule.csp* page is started very similar to other CSP pages by collecting the values passed. However, you will notice that in *Schedule.csp* we have to use both *Request.QueryString* and *Request.Form*. This is because some of the information (in this case *PFolder* and *PReport*) was passed on the *QueryString* while the other information was passed in via the HTTP header.

After collecting all of the parameters, we create the Session Manager, logon to the APS with the logon token and create the *InfoStore*. We then query the *InfoStore* for the report object by its ID. Once we have the report collection created (consisting of the one report that we want to schedule), we extract that report from the collection using the *Item* property of the collection.

Now we have our report that we want to schedule. To set all of the scheduling information for this report we create a *SchedulingInfo* object by invoking the *SchedulingInfo* property of the report object. Once we have the *SchedulingInfo* object instantiated we can go about setting the various properties of our scheduled job. We first check the "*when*" variable. If the value is not "Right now" then we set the date and time specific information, otherwise we set the *SchedulingInfo RightNow* property to "True" (1).

Once we have set the schedule time, we set the frequency of the schedule (the recurrence). If the frequency is set to *N hours* then we set the *ScheduleInfo IntervalHours* property to the value collected from the form (stored in the *NthHour* variable). If the frequency was not *N hours*, we do not set the *IntervalHours* property. We perform the same check for *N Day* frequency and then set the *IntervalNthDay* property, if required. Finally, to commit the schedule we call the schedule method of the *InfoStore* object, passing the collection rather than the object. Therefore, a collection of objects is actually scheduled rather than a single object.

Displaying the Status of the Schedule

We will use a *ListInstances.csp* page to display the status of the schedule once we have given it to the APS. To get to the list of instances we will use *ListFolders.csp, FullFolderpath.csp, and ListReports.csp to call ListInstances.csp. S*amples of each are included below. After logging on to the APS and creating the *InfoStore* object as an interface to extract objects from the APS, we will query the *InfoStore*. What we query the *InfoStore* for is really dependant on the specification of the CSP application that you are developing. This could range from creating an entire desktop application, to simply providing a list of hyperlinks to preview reports. The design of any one application is not as important as how to interact with the *InfoStore* to retrieve the information that you need.

The three types of APS objects that you will likely be interested in are: folders, reports, and instances. We will look at these objects, and how they are related. Sample CSP pages are provided to extract these objects from the APS database and display them to the browser. For more in-depth information, please refer to the Web Developer's Guide included in the \Doc directory of the CE CD.

ListFolders.csp Code and Analysis

ListFolders.csp will list all the folders defined on the APS. The first thing that we want to do is modify *Logon.csp* to call *ListFolders.csp*.

ListFolders.csp will also take advantage of the LogonWithToken method and logon with the token created in *Logon.csp*. To call the *ListFolders.csp* from *Logon.csp* add the following line just above the last "end if" statement of the page:

Response.Redirect "ListFolders.csp?FolderID=0"

This redirects the user's browser to *ListFolders.csp,* specifically the query string portion of that request (the "?FolderID=0" part). In order to list folders in *ListFolders.csp* we must first reconnect to the APS. To reconnect to the APS we must retrieve the token that we stored on the browser (in the cookie called LogonToken) and use the LogonWithToken method to pass that information to the APS (to "remind" it about us).

The following is a sample *ListFolders.csp*:

```
<!-- #Include file=FullFolderPath.csp -->
<%
ParentID = Request.Querystring("FolderID")

'Retrieve the logon cookie

apstoken = Request.Cookies("LogonToken")
If apstoken = "" then
       response.redirect("LogonForm.csp")
End If

'Create a session manager

Set SessionManager = Server.CreateObject("CrystalEnterprise.SessionMgr")

'Logon using the token.  Will fail if the token is no longer valid.


Set oEntepriseSession = SessionManager.LogonWithToken(apstoken)

Set oIStore = oEntepriseSession.Service("", "InfoStore")
FolderPath = RetrieveFullPath(ParentID, oIStore, Success)

Response.Write(FolderPath & "<BR><BR>")

If cFolders.Count = 0 Then
       Response.Write("No SubFolders in Current Folder")
End if

Set cFolders = oIStore.Query("Select SI_ID, SI_NAME From CI_INFOOBJECTS Where
SI_OBType=1 AND SI_PARENTID=" & Cint(ParentID))

For each oFolder in cFolders
Set cSubFolders = oIStore.Query("Select SI_NAME From CI_INFOOBJECTS " &_
"Where SI_OBType=1 AND SI_PARENTID=" & Cint(oFolder.ID))

set cReports = oIStore.Query("Select SI_NAME from CI_INFOOBJECTS " &_
"Where SI_OBType=2 AND SI_INSTANCE=0 AND SI_PARENTID=" & Cint(oFolder.ID))

If cSubFolders.Count = 0 then
Response.Write("<B>Folder Name:</B>  " & oFolder.Properties("SI_Name") & "<BR>")

If cReports.Count > 0 then
      Response.Write("           ")
Response.Write "<a href='listreports.csp?PFolder=" & oReport.Properties("SI_ID")
&_"'>View Report Objects in Folder</a>"
Response.Write "<BR><BR>"
else
```

```
      Response.Write("           ")
Response.Write "No Reports in Folder"
Response.Write "<BR><BR>"
End If
  Else
Response.Write("<B>Folder Name:</B>  " & "<a href='ListFolders.csp?FolderID=" &_
oFolder.Properties("SI_ID") & "'>" & oFolder.Properties("SI_Name") &_
"</a> <BR>")

If cReports.Count > 0 then
    Response.Write("           ")
Response.Write "<a href='listreports.csp?PFolder=" & oReport.Properties("SI_ID") &_
"'>View Report Objects in Folder</a>"
Response.Write "<BR><BR>"
Else
Response.Write("           ")
Response.Write "No Reports in Folder"
Response.Write "<BR><BR>"
End If
End If
Next
%>
```

FullFolderPath.csp Code and Analysis

You will notice the first line of *ListFolders.csp* (<!-- #Include file=FullFolderPath.csp -->) calls *FullFolderPath.csp*. The #Include directive copies the contents of another file at execution time. Therefore, when we call *ListFolders.csp*, the contents of *FullFolderPath.csp* are copied into *ListFolders.csp*. The *FullFolderPath.csp* page defines a function that we can use to display a hyper linked folder tree. Somewhat of a "breadcrumb path" of where we are in the directory structure and how we got there. You can copy this code and save it as *FullFolderPath.csp* and save it in the same directory as *ListFolders.csp* so that when *ListFolders.csp* is called, *FullFolderPath.csp* can be included. This code works by using the property bag SI_Path. The SI_Path contains a few properties that we use to determine the parent folder structure of the folder we are currently "in". We use the SI_NUM_FOLDERS property to determine the number of parent folders our current folder has. We can use that number to loop through each parent folder displaying its name and hyper linking to *ListFolders.csp* using its ID. We add each folder's name hyper linked separated by a "/" character. Therefore, at the top of our folder list, we have a nice way to navigate directly to the home folder or any parent folder of our current folder. When complete the string that is written will have this form:

Home/SubFolder/SubFolder/CurrentFolder

This code will provide a complete folder-listing page, with a convenient "breadcrumb" navigation scheme as well as identify which folders contain report objects and instances. If a folder contains report objects and instances, it will provide a link to click to view the list of report objects.

The code for *FullFolderPath.csp* is as follows:

```
<%
Function RetrieveFullPath(FolderID, IStore, Success)

Dim Query
Dim Result
Dim Path

Path = ""
```

```
If FolderID = 0 Then

RetrieveFullPath = "<a href='ListFolders.csp?FolderID=0'>Home</a>"
       Success=TRUE
       Exit Function
End If

Query = "Select SI_PATH, SI_NAME From CI_INFOOBJECTS Where " & ”SI_ID =" &
Cstr(FolderID)

On Error Resume Next

Set Result = IStore.Query(Query)
If Err.Number <> 0 Then

Success=FALSE
Exit Function
End If

Dim FullPath

If Result.Count > 0 Then

Dim NumFolders
NumFolders = Result.Item(1).Properties("SI_PATH").Properties("SI_NUM_FOLDERS")

HomeLink = "<A HRef='ListFolders.csp?FolderID=0' >Home</A>"

       Dim CurrentName
CurrentName = Server.HTMLEncode(Result.Item(1).Properties("SI_NAME"))

Dim CurrentFolder
CurrentFolder = "<A HRef='ListFolders.csp?FolderID="&Cstr(FolderID) & "' >" &_
CurrentName & "</A>"
              FullPath = HomeLink & " / " & CurrentFolder

If  NumFolders > 0 then
                    Dim k, ParentID, ParentName

              FullPath = ""
              For k = 1 to NumFolders

ParentId = Result.Item(1).Properties("SI_PATH").Properties("SI_FOLDER_ID" & Cstr(k))

ParentName = Result.Item(1).Properties("SI_PATH").Properties("SI_FOLDER_NAME" &
Cstr(k))
ParentName = Server.HTMLEncode(ParentName)
FullPath = "<A HRef='ListFolders.csp?FolderID=" &_ Cstr(ParentID) & "' >" &
_ParentName & "</A>" & " / " & FullPath
              Next

FullPath = HomeLink & " / " & FullPath & CurrentFolder
       End If
       Success=TRUE
Else
       Success=FALSE
End If

RetrieveFullPath = FullPath

End Function
%>
```

ListReports.csp Code and Analysis

For our listing reports page we will provide the ability to view the report, or view a list of instances if there are any. We will also provide a link to schedule the report. Our report-listing page will also make use of the *FullFolderPath.csp* so that we can include our "breadcrumb" navigation links in the report listings.

To list the report objects, our csp page will need to know what folder we are currently in. Therefore, we will need to collect the folder ID from the *ListFolders.csp* page. We built the *ListFolders.csp* page such that it hyper linked to *ListReports.csp* passing this information. We will need to reconnect to the APS, which will require collecting the LogonToken cookie from the user. In addition, we will need to create the *InfoStore* object to get a list of report objects. Therefore, *ListReports.csp* starts out looking a lot like our *ListFolders.csp*. We will query the APS for the list of reports for our folder. In the ListFolders page, we asked the *InfoStore* to find all of the folder objects for a given parent. For this page, we need to select all of the objects for a given parent where the type of object is "report", but not report instances. Instances will be listed later.

The code for *ListReports.csp* is as follows:

```
<!-- #Include file=FullFolderPath.csp -->
<%
ParentID = Request.Querystring("PFolder")

'Retrieve the logon cookie

apstoken = Request.Cookies("LogonToken")
If apstoken = "" then
      response.redirect("LogonForm.csp")
End If

'Create a session manager

Set SessionManager = Server.CreateObject("CrystalEnterprise.SessionMgr")

'Logon using the token.  Will fail if the token is no longer valid.

Set oEntepriseSession = SessionManager.LogonWithToken(apstoken)

Set oIStore = oEntepriseSession.Service("", "InfoStore")
FolderPath = RetrieveFullPath(ParentID, oIStore, Success)

Response.Write(FolderPath & "<BR><BR>")

Set cReports = oIStore.Query("Select SI_Name, SI_Children from CI_INFOOBJECTS Where
SI_PARENTID = " & Cint(ParentID) & " And SI_OBType = 2 and SI_Instance = 0")

For each oReport in cReports
Response.Write("<B>Report Name: </B>" & oReport.Properties("SI_Name") & "<BR>")
Response.Write("    <a href='viewrpt.cwr?id=" &_
oReport.ID & "&apstoken=" & apstoken & "'>" & "Run Report</a>")
Response.Write("    " & "<a href='schedule.csp?PReport=" &_
oReport.ID & "'>Schedule Report</a>")

If oReport.Properties("SI_Children") > 0 then
Response.Write("    " & "<a href='listinstances.csp?PReport=" &_
oReport.ID & "'>View Instance List</a><BR>")
Else
Response.Write("    View Instance List <BR>")
End If
Next
%>
```

ListInstances.csp Code and Analysis

Let's consider a few things about instances. Does it make sense to list it by name? Not really. An instance is a copy of a report object, only saved with data. Therefore, the name will be the same as the report object itself. Are all instances viewable? No. What if we list a failed instance? Another consideration is since an instance is a report with saved data, when was the instance created\run? It would be good if we could figure out which instance we wanted to view based on a time stamp. Using these considerations, we will modify our For…Each loop to display the instance status, its last updated time stamp, and a link to view the instance. But only if the instance was successful. If the instance's status was "Failed" or "Error", we will display the error message associated with the instance.

The code for *ListInstances.csp* is as follows:

```
<!-- #Include file=FullFolderPath.csp -->
<%
ParentFolder = Request.Querystring("PFolder")
ParentReport = Request.Querystring("PReport")

'Retrieve the logon cookie

apstoken = Request.Cookies("LogonToken")
If apstoken = "" then
      response.redirect("LogonForm.csp")
End If

'Create a session manager

Set SessionManager = Server.CreateObject("CrystalEnterprise.SessionMgr")

'Logon using the token.  Will fail if the token is no longer valid.


Set oEntepriseSession = SessionManager.LogonWithToken(apstoken)

Set oIStore = oEntepriseSession.Service("", "InfoStore")
FolderPath = RetrieveFullPath(ParentFolder, oIStore, Success)

Response.Write(FolderPath & "<BR><BR>")

Set cInstances = oIStore.Query("Select SI_UISTATUS, SI_Update_TS, SI_Error_Message
from CI_INFOOBJECTS Where SI_PARENTID = " & Cint(ParentReport) & " And SI_OBType = 2
and SI_Instance = 1")

For each oInstance in cInstances
If oInstance.Properties("SI_UISTATUS")=1 then
Response.Write("<a href='viewrpt.cwr?id=" &_
oInstance.ID & "&apstoken=" & apstoken & "'>" &_
"View Instance</a><BR>")

      Else
            Response.Write("View Instance<BR>")
      End If

Select Case oInstance.Properties("SI_UISTATUS")
      Case 0
Response.Write("<B>Status: </B> Processing")
      Case 1
Response.Write("<B>Status: </B> Success")
      Case 3
Response.Write("<B>Status: </B> Failed ")
Response.Write(" (" & oInstance.Properties("SI_Error_Message") & ")")
```

```
        Case 4
Response.Write("<B>Status: </B> Loaded")
        Case 5
Response.Write("<B>Status: </B> Loading")
        Case 6
Response.Write("<B>Status: </B> Not Responding")
        Case 7
Response.Write("<B>Status: </B> On Hold")
        Case 8
Response.Write("<B>Status: </B> Paused")
        Case 9
Response.Write("<B>Status: </B> Scheduled")
        Case 10
                Response.Write("<B>Status: </B> Error ")
                Response.Write(" " & SI_Error_Message)
        Case 11
Response.Write("<B>Status: </B> Queued")
        Case 12
Response.Write("<B>Status: </B> Run Today")
        Case 13
Response.Write("<B>Status: </B> Terminated")
        Case 13
Response.Write("<B>Status: </B> Terminating")
        Case Else
Response.Write("<B>Status Unavailable</B>")
End Select

Response.Write("    <B>Updated: </B>" &
oInstance.Properties("SI_Update_TS") & "<BR>")

Next
%>

<%
ParentFolder = Request.Querystring("PFolder")
ParentReport = Request.Querystring("PReport")

'Retrieve the logon cookie

apstoken = Request.Cookies("LogonToken")
If apstoken = "" then
        response.redirect("LogonForm.csp")
End If

'Create a session manager

Set SessionManager = Server.CreateObject("CrystalEnterprise.SessionMgr")

'Logon using the token.  Will fail if the token is no longer valid.


Set oEntepriseSession = SessionManager.LogonWithToken(apstoken)

Set oIStore = oEntepriseSession.Service("", "InfoStore")
FolderPath = RetrieveFullPath(ParentFolder, oIStore, Success)

Response.Write(FolderPath & "<BR><BR>")

Response.Write("<a href='listinstances.csp?PReport=" & ParentReport &_    "&PFolder="
& ParentFolder & "'>Refresh Instance List</a><BR><BR>")

Set cInstances = oIStore.Query("Select SI_UISTATUS, SI_Update_TS, SI_Error_Message
from CI_INFOOBJECTS Where SI_PARENTID = " & Cint(ParentReport))
```

```
For each oInstance in cInstances
Response.Write("<B>Updated: </B>" & oInstance.Properties("SI_Update_TS")
&_"    ")

If oInstance.Properties("SI_UISTATUS")=1 then
Response.Write("<a href='viewrpt.cwr?id=" &_
oInstance.ID & "&apstoken=" & apstoken & "'>" &_
"View</a><BR>")
       Else
              Response.Write("<BR>")
       End If

Select Case oInstance.Properties("SI_UISTATUS")
       Case 0
Response.Write("    <B>Status: </B> Processing")
       Case 1
Response.Write("    <B>Status: </B> Success")
       Case 3
Response.Write("    <B>Status: </B> Failed ")
Response.Write(" (" & oInstance.Properties("SI_Error_Message") & ")")
       Case 4
Response.Write("    <B>Status: </B> Loaded")
       Case 5
Response.Write("    <B>Status: </B> Loading")
       Case 6
Response.Write("    <B>Status: </B> Not Responding")
       Case 7
Response.Write("    <B>Status: </B> On Hold")
       Case 8
Response.Write("    <B>Status: </B> Paused")
       Case 9
Response.Write("    <B>Status: </B> Scheduled")
       Case 10
Response.Write("    <B>Status: </B> Error ")
             Response.Write(" " & SI_Error_Message)
       Case 11
Response.Write("    <B>Status: </B> Queued")
       Case 12
Response.Write("    <B>Status: </B> Run Today")
       Case 13
Response.Write("    <B>Status: </B> Terminated")
       Case 13
Response.Write("    <B>Status: </B> Terminating")
       Case Else
Response.Write("    <B>Status Unavailable</B>")
End Select

Response.Write("<BR>")

Next
%>
```

Scheduling Revisited

We have developed a page that will allow us to schedule a report object. However, [JT1]our scheduling page does not provide the functionality to change the properties of the object. For example, if the report object contains parameters, or requires database logon information, our scheduling page will not allow the user to set any of these values. If the report

had default values for parameters or database logon, these defaults would be used when scheduled from our page. Otherwise, no values would be used.

Therefore, when developing a desktop application that provides scheduling functionality, you must take into consideration any parameters, database or selection formula information that the object has. Otherwise scheduling may not work properly, if at all.

Report Plug-in

To modify report object properties, such as parameters, database logon\location or selection criteria, the *ReportPlugin* object, *Report.dll*, is used. To access the Report Plug-in you use the "*PluginInterface*" property of a report object. For example, [JT2]the following code will create a collection containing a single report object:

```
Set session("cReports") = oIStore.Query("Select SI_Name, SI_Children from
CI_INFOOBJECTS Where SI_PARENTID = 111 and SI_OBType = 2 and SI_Instance = 0")
```

To create the Report Plug-in interface, you first extract the report object from the collection.

```
set oTheReport = session("cReports").Item(1)
```

Then you create a new object by referencing the *PluginInterface* property of the report object.

```
set ReportPlugin = oTheReport.PluginInterface
```

Once you have created the *ReportPlugin* object, you have access to a few different collections and objects to get or set attributes of the report. There are three main parts of the Report Plug-in interface: the *Report* object (which is essentially the *PluginInterface* object itself), the *Report Parameters* collection and the *Database Logons* collection. We will start out with the report object, then look at the *Database Logons* collection object, and finish with the *Report Parameters* collection, as it is the most complex.

Once you have an understanding of the various collections and objects exposed by the Report Plug-in, you [JT3]will learn how to modify our original scheduling page to provide for parameter, database logon and selection formula modification functionality.

The Report Object

The report object contains a few properties that can be modified, but also serves as the interface to the *Report Parameters* and *Database Logons* collections. The report object contains the selection formula properties (record and group). So if the user wants to add, edit or view either of the selection formulas it is just a matter of accessing the *RecordFormula* or *GroupFormula* property.

For example:

To display the record selection formula set for the report object, you would use the following code (assuming continuation of the previous code samples):

```
Response.Write(ReportPlugin.RecordFormula)
```

To set the Record Selection formula you would use the same property.

```
ReportPlugin.RecordFormula = "{table.field = 'Value' or table.different_field =
'Different Value'}"
```

Getting or setting the Group Selection formula would require similar code, only using the *GroupFormula* property.

ReportLogons Collection

The report object contains a property that allows the developer to check if the report's database(s) requires logon credentials. This is [JT4]essential if you need to authenticate a user to the database. You can test to see if the report object requires a database logon and if it does, then provide [JT5]the user a logon page.

For example:

```
If ReportPlugin.NeedsLogon then
  Response.Write("<a href = 'GetDatabaseInfo.csp?PReport=" & ParentReport &
"&PFolder=" & ParentFolder & "'>Click Here for Database Logon Information</a>")
End If
```

This code checks the *NeedsLogon* property, and if "True", a hyperlink is created for the user to click. On this new page, the user can enter database, user name and password information.

On this new page, *GetDatabaseInfo.csp*, [JT6]we would use the *ReportLogons* collection to get or set the database logon and\or location information. To instantiate the *ReportLogons* collection:

```
Set cLogons = ReportPlugin.ReportLogons
```

Since a report can contain more that one table, the *ReportLogons* property returns a collection of objects representing the tables used in the report. Now that you have access to the collection, you can reference a particular table by "extracting" an object from the collection. For example:

```
Set oLogon = cLogons.item(1)
```

The other option, of course,[JT7] would be to loop through each of the objects in the collection in order to [JT8]get or set the values. For example, [JT9]if you wanted to display current server, database, and user name information for each table, [JT10]you could use the following loop. ***(Note: Database passwords, even those saved with the report, cannot be retrieved, only set.)***

```
counter = 1
For Each oLogon in cLogons
  Response.Write("Logon Information for Table " & counter)
  Response.Write("<BR>")
  Response.Write("Database Server: <Input Name=dbServer Size=30 Value='" &
oLogon.ServerName & "'>")
  Response.Write("<BR>")
  Response.Write("Database: <Input Name=dbName Size=36 Value='" & oLogon.DatabaseName
& "'>")
  Response.Write("<BR>")
  Response.Write("Logon Username:  <Input Name=dbUID Size=30 Value='" &
oLogon.UserName & "'>")
  Response.Write("<BR>")
  Response.Write("Logon Password: <Input type=password Name=dbpwd Size=30>")
  Response.Write("<BR>")
```

```
  Response.Write("<Input Type=Hidden Name=id Value=" & ObjectID & ">")
  counter = cint(counter) + 1
next
```

The above code is a good example of how to both display and retrieve logon\location information as the current data is displayed in editable text boxes. The user name and password properties will likely be the most used, with the *ServerName* and *DatabaseName* required only for changing the database location information at schedule time. Each of these properties is Read\Write so changing the value is simply a matter of assigning a value directly to the property.

ReportParameters Collection

The *ReportParameters* collection is accessible by using the *ReportParameters* property of the *ReportPlugin* object.

```
Set cParams = ReportPlugin.ReportParameters
```

At this point, you have access to the collection of parameters contained in the report. However, from this point it gets a little tricky. The goal of your application is probably going to be addressing the parameters, and either displaying their value(s) or setting their value(s). Likely, it will be some combination of both.

Even though all parameters generally have the same properties, there are two different types of parameters: discrete value and range value. Both parameter types have slightly different handling requirements for[JT11] getting and setting their values. Therefore [JT12]when you access a parameter object you gain access to a number of properties that are Read-Only. Unfortunately, it is not as simple as accessing the parameter object and setting its value property. Therefore, [JT13]before we get into the code needed to get or set a parameter's value, [JT14]we must review the object model as it pertains to parameters.

The top-level object is the *ReportParameters* collection, which contains all of the parameter objects in the report. Next [JT15]is the parameter object itself. The parameter object contains many properties that describe the parameter. To access a parameter object you will extract the object from the collection.[JT16]

```
Set oParam = cParams.Item(1)
```

However, as far as value(s) of the parameter, they are contained in separate collections. To access a parameter's current value(s) you use the *CurrentValues* property. Likewise, [JT17]to access any default value(s) a parameter contains, you would use the *DefaultValues* property. There is also a *PickListValues* property that will return a collection of values contained in a pick list associated with a parameter. Calling any of these properties will return a *ReportParameterValues*, populated with the appropriate information. For example, to display the parameter's default value(s):

```
Set cDefaultValues = oParam.DefaultValues
If cDefaultValues.Count > 0 then
  Response.Write("Default Values for" & oParam.Parametername)
  for each oDefaultValue in cDefaultValues
    Response.Write(oDefaultValue.MakeDisplayString)
  Next
Else
  Response.Write("Parameter has no default values")
End If
```

When displaying a parameter's values (default or current) the *MakeDisplayString* method is used to display a string representation of that value. Not all values are stored as strings, but it is somewhat easier to use the string

convention to display the value, so that all values are displayed as strings. Range values use set notation to illustrate the inclusion or exclusion of bounding values. Since a *ReportParameterValues* collection is returned when calling the *DefaultValues* and *CurrentValues* properties, y[JT18]ou could use almost the same code to display a parameter's current values.

In order to set a current value you [JT19]have to [JT20]determine whether or not the parameter accepts discrete values or range values. The *ReportParameter* object contains the *SupportsDiscreteValues* and *SupportsRangeValues* boolean properties that allow us to test the parameter to determine what type it is. From here, we can create new current value objects.

Values for a parameter are also objects. Since discrete values and range values have different properties that describe them, the object model has both *ReportParameterSingleValue* and *ReportParameterRangeValue* objects. Once you determine what type of value the parameter requires, you must create a new value object of the correct type. The *ReportParameter* object contains properties to create the two types of value objects. For example:

```
If oParam.SupportsDiscreteValues then
  Set oCurrentValue = oParam.CreateSingleValue
Else
  Set oCurrentValue = oParam.CreateRangeValue
End If
```

Once you have created a new value object of the correct type, you can modify its properties. The discrete (single value) object is a bit easier because it has a "value" property that you use to set its value, whereas a range value requires several properties that must be set. You will actually [JT21]set the *FromValue* and *ToValue* properties to set the values for the range.

However, you also set the boolean properties *IncludesLowerBound* and *IncludesUpperBound* to say whether or not the *FromValue* and *ToValue* are inclusive or exclusive, respectively. Finally,[JT22] you set the boolean properties *HasNoLowerBound* and *HasNoUpperBound* if you do not wish to include one bound or another.

You must take caution in dealing with range value objects. If you do not set the properties correctly, it can result in incorrect data or cause the schedule to fail. Some of the properties must or must not be set, [JT23]depending on other properties.

For example, [JT24]if you set the *NoLowerBound* property to "True", you cannot set the *IncludesLowerBound* property to "True" and you should not set anything for the *FromValue* property. Furthermore, [JT25]you cannot set both the *NoLowerBound* and *NoUpperBound* properties to "True" at the same time. Unbounded ranges are not supported. Finally,[JT26] if you set values for the *ToValue* and *FromValue* you must set the *IncludesLowerBound* and *IncludesUpperBound* properties. The parameter needs to know if these values are to be included or not.

Once you have set the appropriate value-related properties for the *ParameterValue[JT27]* object, [JT28]you add it to the *CurrentValues* collection using the *Add* method. So lets extend the "if…then…else" statement used above to set a value for the parameter. Lets assume that if our parameter accepts discrete values we want to set its value to "1". But if it accepts range values we want to set the range of "1 to 10", including "1" but not "10". Here is the code that we would use:

```
If oParam.SupportsDiscreteValues then
  Set oCurrentValue = oParam.CreateSingleValue
  oCurrentValue.Value = 1
Else
  Set oCurrentValue = oParam.CreateRangeValue
  oCurrentValue.FromValue = 1
  oCurrentValue.ToValue = 10
  oCurrentValue.IncludesLowerBound = "true"
  oCurrentValue.IncludesUpperBound = "false"
  oCurrentValue.HasNoLowerBound = "false"
```

```
  oCurrentValue.HasNoUpperBound = "false"
End If
```

Now that we have created a current value object[JT29] of the correct type, and set the appropriate properties, the final step is to add that current value object to the parameter's current value collection. To do this you create the *CurrentValues* collection and call the *Add* method.

```
Set cCurrentValues = oParam.CurrentValues
CCurrentValues.Add(oCurrentValue)
```

Calling the *Add* method is the final step in setting a value for the parameter. So to review, the steps are:

1. Create the *ReportPlugin* interface.
2. Create the *ReportParameters* Collection.
3. Test the value type of the parameter (using the *SupportsDiscreteValues* or *SupportsRangeValues* properties).
4. Create a new *ParameterValue* object (of the correct type based on the results of step 3).
5. Set the appropriate properties of the *ParameterValue* object.
6. Call the *Add* method of the *CurrentValues* collection to add the new value object as a current value.

A *ValueObject[JT30]* can be addressed once it has been added to the collection. However, [JT31]you must be cautious in modifying a parameter that is already part of the collection. This is where the *SupportsDiscreteValues* and *SupportsRangeValues* properties come in handy again.

For example, lets say that we want to modify an existing current value to "10" if it is a discrete value[JT32] parameter, or "10 through 20" (including both bounds) if it is a range. The code would look like this:

```
Set cCurrentValues = oParam.CurrentValues
Set oCurrentValue = cCurrentValues.Item(1)
If oCurrentValue.SupportsDiscreteValues then
  oCurrentValue.Value = 10
Else
  oCurrentValue.FromValue = 10
  oCurrentValue.ToValue = 20
  oCurrentValue.IncludesLowerBound = "true"
  oCurrentValue.IncludesUpperBound = "true"
  oCurrentValue.HasNoLowerBound = "false"
  oCurrentValue.HasNoUpperBound = "false"
End If
```

Here, instead of creating a *ParameterValue[JT33]* object using the *CreateSingleValue[JT34]* or *CreateRangeValue* property, we have used the item property of the *CurrentValues* collection to create a *ParameterValue[JT35]* object. But this time we used the *SupportsCurrentValues* and *SupportsRangeValues* properties to determine what type of value we got from the collection, instead of what kind of value we were going to create for the collection.

The last thing about setting parameter values is the use of the *EnableMultipleValues* property. Both discrete and range value parameters can be configured to contain multiple values. However, [JT36]this setting is done when the parameter is created (in the Crystal Report Designer). The *EnableMultipleValues* property is Read-Only. This is the reason for having a *CurrentValues* collection, because some parameters can have more than one current value.

Setting multiple current values is exactly the same as setting a single value. When setting a single value you:

1. Create the ReportPlugin interface.
2. Create the *ReportParameters* collection.

3. Test the value type of the parameter (using the *SupportsDiscreteValues* or *SupportsRangeValues* properties).
4. Create a new *ParameterValue* object (of the correct type based on the results of step 3).
5. Set the appropriate properties of the *ParameterValue* object.
6. Call the *Add* method of the *CurrentValues* collection to add the new value object as a current value.
7. To set multiple values for a parameter, repeat steps 4 through 6. However, you must ensure that the *EnableMultipleValues* property returns "True". If the parameter is not designed to accept multiple values, you could end up overwriting the previously set value or getting errors in your application.

Now that we understand setting, what if a value is set that you do not want to keep? If you can add a value object to the *CurrentValues* collection, [JT37]you should also be able to remove it.[JT38] This is where the *Delete* and *Clear* methods of the *ParameterValues* collection come in. The *Delete* method takes an index parameter allowing you to delete a specific value. The *Clear* method clears the *ParameterValues* collection completely, removing any value objects.

To remove the second current value:

```
Set cCurrentValues = oParam.CurrentValues
call cCurrentValues.Delete(2)
```

To remove all current values:

```
Set cCurrentValues = oParam.CurrentValues
cCurrentValues.Clear
```

That pretty much wraps up the basics of parameter handling. We have learned how to obtain the default values associated with a parameter. We have also seen how to handle the different parameters value types and [JT39]how to actually set a value (and multiple values) for a parameter. We have seen how to modify an existing [JT40]parameter value. Finally,[JT41] we have learned how to remove a single parameter value or clear the entire collection. There are properties and methods that we have not touched on, so be sure to have a look at the parameter object model in the Crystal Enterprise Web Developer's Help to see all the other things you can do. The help guide is located in the \Doc directory of CE CD and is called *WebDeveloperHelp.chm*.

Enhanced Scheduling Page

Now that we have an idea of how to handle selection formulas, database logon and parameters,[JT42] let's examine methods for enhancing our scheduling page to provide functionality to get and set these report properties. We will take our existing[JT43] scheduling page, *Schedule.csp*, and use it as a starting point. Some of the code can be reused, but for the most part, we will be building a new page. We will also be adding separate pages to handle the enhanced functionality. Pages should be separated based on functionality to avoid having one page doing too much, thus taking too long to process. Therefore, previous scheduling page will look very different when we are done.

We will follow the same progression in adding functionality to the scheduling page as we did in learning the new Report Plug-in. We will start by adding selection formula (record and group) functionality, then [JT44]we will add database logon\location functionality, [JT45]and finally[JT46] look at parameter handling.

Before we start writing code, we should decide how we are going to provide this functionality. The main consideration will be the UI, as it will primarily dictate how the pages need to be developed. In deciding how the UI will look, we should consider what exists currently, as consistency[JT47] is a good thing.

In the Seagate Info Desktop for Windows, selection formula, database logon, and parameter information are all displayed\set on separate tabs of the schedule dialog. In the Crystal Enterprise ePortfolio desktop, there is a single scheduling page to collect date\time, selection formula, database logon\location and parameter information. However, [JT48]the page does not contain all of this UI at the same time as the user selects the information they want using a drop-down select control.

To remain[JT49] somewhat consistent with ePortfolio (but actually become a little simpler), [JT50]we will use a single scheduling page that displays Date, Time and Recurrence options (basically our previous scheduling page), but provide links on the page to allow the user to get and set the selection formula, database logon\location and parameter information. We will provide a link to get and set selection formula, but only provide links for the database logon\location and parameters if the report requires them.

The scheduling portion of our current desktop consists of two pages: *[JT51]ScheduleUI.csp* (which allowed the user to set scheduling options) and *Schedule.csp* (which collected the information from *ScheduleUI.csp*, and set the scheduled job).

We are going to use the *ScheduleUI.csp* we created earlier as the base page to allow the user to specify the date, time, and recurrence options. Our new *ScheduleUI.csp* will also provide a link to selection formulas, database logon\location and parameter information pages. As mentioned previously, [JT52]the database logon and parameter page links will only be provided if required for the report.

New Date\Time\Recurrence Interface

It is a good idea to separate pages by functionality. This is because code becomes hard to maintain when a page performs too many and\or non-related operations. In addition, since CSP is an interpreted[JT53] language, the smaller the page, and the faster it can be interpreted and executed.

Therefore, [JT54]we will try to keep our pages small to make them as manageable and efficient as possible. To this end, *ScheduleUI.csp* will be somewhat of a homepage for all of the scheduling functionality. We will either link to, or include other separate pages to get or set the property information. The first thing that we are going to do is modify how the date, time and recurrence information is handled. We will create two new pages that we will integrate into *ScheduleUI.csp*. They will be *GetDateTime.csp* and *SetDateTime.csp*. *GetDateTime.csp* will be the UI. It will show any current date\time\recurrence options set, or display default information for the user to accept or change. *SetDateTime.csp* will set the information submitted[JT55] by the user[JT56] from *GetDateTime.csp*.

GetDateTime.csp Code and Analysis

```
<%
Response.Write("<FORM Name=the_form Method=POST Action=ScheduleUI.csp?PReport=" &
ParentReport & "&PFolder=" & ParentFolder & ">")
If SchedulingInfo.RightNow then
  Response.write("<INPUT type='radio' name='When' Value='Now' Checked>Right Now<BR>")
  Response.write("<INPUT type='radio' name='When' Value='At Time'>At Date/Time")
ElseIf Not IsNull(SchedulingInfo.BeginDate) then
  Response.write("<INPUT type='radio' name='When' Value='Now'>Right Now<BR>")
  Response.write("<INPUT type='radio' name='When' Value='At Time' Checked>At
Date/Time")
Else
  Response.write("<INPUT type='radio' name='When' Value='Now' Checked>Right Now<BR>")
  Response.write("<INPUT type='radio' name='When' Value='At Time'>At Date/Time")
End If
CurrentDay = Day(Now())
SchedYear = Year(cdate(SchedulingInfo.BeginDate))
SchedMonth = Month(cdate(SchedulingInfo.BeginDate))
SchedDay = Day(cdate(SchedulingInfo.BeginDate))
SchedHours = Hour(SchedulingInfo.BeginDate)
SchedMinutes = Minute(SchedulingInfo.BeginDate)
Response.Write("<INPUT Size=3 Name=Year Value=" + cstr(SchedYear) + ">")
```

```
Response.Write("<INPUT Size=1 Name=Month Value=" + cstr(SchedMonth) + ">")
Response.Write("<INPUT Size=1 Name=Day Value=" + cstr(SchedDay) + ">")
%>
   
<Select name=hours>
<%
If cint(SchedHours) > 12 then
  SchedHours = SchedHours-12
  AMPM = "PM"
ElseIf cstr(SchedHours) = "0" then
  SchedHours = "12"
  AMPM = "AM"
ElseIf cstr(SchedHours) = "12" then
  SchedHours = "12"
  AMPM = "PM"
ElseIf cint(SchedHours) < 10 then
  SchedHours = Right(cstr(SchedHours), 1)
  AMPM = "AM"
Else
  AMPM = "AM"
End If
Response.write("<option selected>" & cstr(SchedHours))
%>
<Option>1<Option>2<Option>3<Option>4<Option>5<Option>6<Option>7<Option>8<Option>9
<Option>10<Option>11<Option>12
</Select>
:
<Select name=minutes>
<%
If SchedMinutes < 10 then
  Response.write("<option selected>0" & cstr(SchedMinutes))
Else
  Response.write("<option selected>" & cstr(SchedMinutes))
End If
For i = 0 to 9
  Response.Write("<Option>0"&i)
Next
For j = 10 to 59
  Response.Write("<Option>"&j)
Next
%>
</Select>
<Select name=AMPM>
<%
If AMPM = "AM" then
  Response.write("<option selected>AM")
  Response.write("<option>PM")
ElseIf AMPM = "PM" then
  Response.write("<option selected>PM")
  Response.write("<option>AM")
End If
%>
</Select>
<BR>
Recurrence<BR>
<%
Select Case SchedulingInfo.Type
  Case "0"
    CheckedorNot0 = "Checked"
  Case "2"
    CheckedorNot2 = "Checked"
  Case "3"
    CheckedorNot3 = "Checked"
```

```
  Case "4"
    CheckedorNot4 = "Checked"
  Case "6"
    CheckedorNot6 = "Checked"
  Case "7"
    CheckedorNot7 = "Checked"
  Case "5"
    CheckedorNot5 = "Checked"
  Case "1"
    CheckedorNot1 = "Checked"
  Case Else
    CheckedorNot0 = "Checked"
End Select

Response.Write("<INPUT TYPE='RADIO' NAME='frequency' VALUE='0'" & CheckedorNot0 &
">Once Only")
Response.Write("<INPUT TYPE='RADIO' NAME='frequency' VALUE='2'" & CheckedorNot2 &
">Daily")
Response.Write("<INPUT TYPE='RADIO' NAME='frequency' VALUE='3'" & CheckedorNot3 &
">Weekly")
Response.Write("<INPUT TYPE='RADIO' NAME='frequency' VALUE='4'" & CheckedorNot4 &
">Monthly")
Response.Write("<INPUT TYPE='RADIO' NAME='frequency' VALUE='6'" & CheckedorNot6 &
">Every Monday")
Response.Write("<INPUT TYPE='RADIO' NAME='frequency' VALUE='7'" & CheckedorNot7 &
">Every Last Day of Month")
%>
<BR>
<%
Response.Write("<INPUT TYPE='RADIO' NAME='frequency' VALUE='5'" & CheckedorNot5 &
">Every Nth Day of Month")
%>
(Where Nth Day is)<Select Name=NthDay>
<%
If SchedulingInfo.Type = 5 then
  Response.write("<option selected>" & cstr(SchedulingInfo.IntervalNthDay))
Else
  Response.write("<option selected>" & cstr(CurrentDay))
End If
For i = 1 to 9
  Response.Write("<Option>"&i)
Next
For j = 10 to 31
  Response.Write("<Option>"&j)
Next
%>
</select>
<%
Response.Write("<INPUT TYPE='RADIO' NAME='frequency' VALUE='1'" & CheckedorNot1 &
">Every N Hours")
%>
(Where N Hours is)
<%
If NthHour <> "" Then
  Response.Write("<INPUT Size=1 Name='NthHour' Value='" & NthHour & "'>")
Else
  Response.Write("<INPUT Size=1 Name='NthHour'>")
End If
%>
<BR><BR>
<Input Type=Hidden Name=SetDateTime Value=1>
<Input type=Submit Value="Set Date/Time">
<Input type=Reset>
```

```
<Input type=Button Value="Schedule Report" onClick="Schedule()">
</FORM>
```

The code used to collect the date, time and recurrence information should look somewhat similar to the code used in *ScheduleUI.csp*. Much of the code has been retained, but enhanced. One thing to notice is that when creating the input controls to collect information from the user, we are using what is currently stored in the *SchedulingInfo* object if values are set; [JT57]otherwise, [JT58]we are displaying defaults (based on the current date and time).

So, *GetDateTime.csp* is now used to create the UI form to collect information from the user, just like *ScheduleUI.csp*. This time, [JT59]instead of calling *GetDateTime.csp* directly[JT60], we will include *GetDateTime.csp* in to *ScheduleUI.csp*. The form tag that *GetDateTime.csp* creates calls *ScheduleUI.csp*, which includes *GetDateTime.csp*, forming[JT61] a bit of a circle. How does the Date\Time\Recurrence data get set? The hidden form control named "SetDateTime" serves as a flag to *ScheduleUI.csp*. If *SetDateTime* is equal to "1", [JT62]we know that the Set Date\Time button was clicked in the *GetDateTime.csp*.

So if *SetDateTime* is equal to "1", we want to set the date\time information by instantiating the *SchedulingInfo* object, [JT63]setting the appropriate values for its properties based on the input from the user. We do that in another page that we will call *SetDateTime.csp*.

SetDateTime.csp Code and Analysis

```
<%
SchedYear = Request.Form("Year")
SchedMonth = Request.Form("Month")
SchedDay = Request.Form("Day")
When = Request.Form("When")
SchedHours = Request.Form("Hours")
SchedMinutes = Request.Form("Minutes")
AMPM = Request.Form("AMPM")
Frequency = Request.Form("Frequency")
NthDay = Request.Form("NthDay")
NthHour = Request.Form("NthHour")
If When <> "Now" then
  If AMPM = "PM" and SchedHours <> "12" then
    SchedHours = cint(SchedHours) + 12
  ElseIf SchedHours = "12" and AMPM = "AM" then
    SchedHours = "00"
  ElseIf SchedHours = "12" then
    SchedHours = "12"
  ElseIf SchedHours < 10 then
    SchedHours = "0" + cstr(SchedHours)
  End If
  SchedDate = SchedYear & "/" & SchedMonth & "/" & SchedDay & " " & SchedHours & ":"
& SchedMinutes & ":" & "00"
  SchedulingInfo.BeginDate = cdate(SchedDate)
Else
  SchedulingInfo.RightNow = 1
End if
SchedulingInfo.Type = cint(Frequency)
If cint(Frequency) = 1 Then
  SchedulingInfo.IntervalHours = cint(NthHour)
End If
If cint(Frequency) = 5 Then
  SchedulingInfo.IntervalNthDay = NthDay
End If
%>
```

SetDateTime.csp takes care of populating the *SchedulingInfo* object's properties based on the *GetDateTime.csp* form values. So, should we create a separate *SetDateTime.csp*? Why not just put this code in *ScheduleUI.csp*? To review, we only want to set this information if the user chooses to by clicking the Set Date\Time button. We use a flag called *SetDateTime*, setting its value equal to "1" if the user submits the *GetDateTime.csp* form data. Therefore, if *SetDateTime* is equal to "1" we want to call *SetDateTime.csp*. This means that[JT64] in our *ScheduleUI.csp* we include *SetDateTime.csp* only if *SetDateTime* is equal to "1". This is a performance gain as the code for *SetDateTime.csp* is only evaluated when necessary.

Adding Selection Formula functionality

Let's move on to handling selection formulas. There are two types of selection formulas; group and record. Both [JT65]will be handled on the same page. We will create a *SelectionFormula.csp* page to allow the user to see the current value of the group or record selection formula (if a current value exists) and[JT66] modify it, or set a new value if no current value exists. The code for *SelectionFormula.csp* is:

```
<HTML>
<HEAD><TITLE>Selection Formula</TITLE>
</HEAD>
<BODY>
<%
Response.ExpiresAbsolute = Now() - 1
ParentReport = Request.QueryString("PReport")
ParentFolder = Request.QueryString("PFolder")
Response.Write("<FORM Name=the_form Method=Post Action=ScheduleUI.csp?PReport=" &
ParentReport & "&PFolder=" & ParentFolder & ">")
Response.Write("Record Selection Formula:")
Response.Write("<BR>")
set oTheReport = session("cReports").Item(1)
set ReportPlugin = oTheReport.PluginInterface
Response.Write("<textarea name='recordsf' rows=5 cols=40>")
Response.Write(ReportPlugin.RecordFormula)
Response.Write("</textarea>")
%>
<BR><BR>
<%
Response.Write("Group Selection Formula:")
Response.Write("<BR>")
Response.Write("<textarea name='groupsf' rows=5 cols=40>")
Response.Write(ReportPlugin.GroupFormula)
Response.Write("</textarea>")
%>
<br>
<input type=hidden name=sfchanged value=''>
<input type=button value=Update OnClick="UpdateSF(1);">
<input type=button value=Cancel OnClick="UpdateSF(0);">
</form>
<script language = "JavaScript">
<!-- hide me from non-functional browsers
function UpdateSF(changed)
{
  document.the_form.sfchanged.value = changed;
  document.forms["the_form"].submit();
}// show me -->
</script>
</BODY>
</HTML>
```

SelectionFormula.csp creates two text areas that display the current value of the record and group selection formula if a current value is set. The user can then modify the existing selection formula, or if no value currently

exists, create a new selection formula. *SelectionFormula.csp* is designed to call *ScheduleUI.csp*, thereby passing the newly set selection formula values. We do not need to create a *SetSelectionFormula.csp*, [JT67]as setting the selection formula value(s) simply requires setting the *RecordFormula* and *GroupFormula* properties of the *ReportPlugin* object. We can add those two lines directly into *ScheduleUI.csp*, however, we need to know if the user intended to set the values or not.

To deal with this we use another hidden form control, named "sfchanged", as a flag variable that is set in *SelectionFormula.csp* and used in *ScheduleUI.csp*. The value is set at[JT68] "1" if the user chooses to set or update the selection formula values, or set to "0" if the user decides not to set or update the values. A simple Submit or Reset button will not work in this approach, and so we will [JT69]turn to some JavaScript. We use the *Document* object's *Forms* collection, which [JT70]allows us to access [JT71]elements of the form and set their value. So executing the "*onClick*" event for the "Update" button we call our *UpdateSF* JavaScript function passing a "1" to set for the value of the *sfchanged* form control and submit the form. Conversely, "*OnClick*" of the "Cancel" button we call the *UpdateSF* function passing a "0" to set for the value of the *sfchanged* form control and then submit the form.

Submitting the form means calling our home (or "container" page) *ScheduleUI.csp*. Back in *ScheduleUI.csp,* we get the value of *sfchanged*. If *sfchanged* is equal to "1", [JT72]we set the *ReportPlugin* object's *RecordFormula* and *GroupFormula* properties to the values set in text area controls from *SelectionFormula.csp*.

That is how to add selection formula functionality in our scheduling CSP page!

Adding Database Logon\Location functionality

Database information is contained in a collection of "Report Logons". A collection of report logon objects is used because a report can contain more than one database (table, view or stored procedure) each with its own location and logon information. In order[JT73] to set database information we have to instantiate the *ReportLogons* collection. Each object within the collection represents a database table's logon and location information. To instantiate the *ReportLogons* collection, you again use the *ReportPlugin*. Example:

```
set ReportPlugin = oTheReport.PluginInterface
set cReportLogons = ReportPlugin.ReportLogons
```

Now that we know how to access the *ReportLogons,* we can display any current values stored with the report object, and collect any values that the user enters. We will use the same UI strategy as we did with the record, group, and record selection formulas. We will also [JT74]display any current values in editable text controls and allow the user to both modify the values and submit the changes, [JT75]or abandon them. However, in this scenario we will be using two separate pages;[JT76] one to get any current values, *GetDatabaseInfo.csp,* and one to set new values, *SetDatabaseInfo.csp*.

The code for *GetDatabaseInfo.csp* looks like this:

```
<HTML>
<HEAD><TITLE>Database Information</TITLE>
</HEAD>
<BODY>
<%
Response.ExpiresAbsolute = Now() - 1
ParentReport = Request.QueryString("PReport")
ParentFolder = Request.QueryString("PFolder")
Response.Write("<FORM Name=the_form Method=POST Action=ScheduleUI.csp?PReport=" &
ParentReport & "&PFolder=" & ParentFolder & ">")
Set ReportPlugin = session("cReports").Item(1).PluginInterface
Set cLogons = ReportPlugin.ReportLogons
If cLogons.Count > 0 Then
  Response.Write("Database Logon Information:")
```

```
  Response.Write("<BR><BR>")
End If
counter = 1
For Each oLogon in cLogons
  Response.Write("Logon Information for Table " & counter)
  Response.Write("<BR>")
  Response.Write("Database Server: <Input Name=dbServer Size=30 Value='" &
oLogon.ServerName & "'>")
  Response.Write("<BR>")
  Response.Write("Database: <Input Name=dbName Size=36 Value='" & oLogon.DatabaseName
& "'>")
  Response.Write("<BR>")
  Response.Write("Logon Username:  <Input Name=dbUID Size=30 Value='" &
oLogon.UserName & "'>")
  Response.Write("<BR>")
  Response.Write("Logon Password: <Input type=password Name=dbpwd Size=30>")
  Response.Write("<BR>")
  Response.Write("<Input Type=Hidden Name=id Value=" & ObjectID & ">")
  counter = cint(counter) + 1
next
%>
<Input Type=Hidden Name=SetDB Value=''>
<input type=button value=Update OnClick="UpdateDB(1);">
<input type=button value=Cancel OnClick="UpdateDB(0);">
</form>
<script language = "JavaScript">
<!-- hide me from non-functional browsers
function UpdateDB(changed)
{
  document.the_form.SetDB.value = changed;
  document.forms["the_form"].submit();
}
// show me -->
</script>
</BODY>
</HTML>
```

In this page we instantiate the *Report Plug-in* object and then use it to instantiate the *ReportLogons* collection. From the *ReportLogons* collection (which we have called *cLogons*), we loop through (using a For…Each loop) accessing all of the *ReportLogon* objects. For each *ReportLogon* object (which we have called *oLogon*), we display any default values for the server name, database name, and user name in editable text boxes and a blank text box for the password (password values cannot be retrieved from the *ReportLogon* object).

In our *GetDatabaseInfo.csp* page, we have also used a hidden control, *SetDB*, to handle state as to whether the user has chosen to commit or abandon any changes made to the database logon or location information. If the user chooses to commit the changes, the *SetDB* form control will be set to "1";[JT77] if the user chooses to abandon the changes the control's value will be set to "0". We will use a JavaScript function, *UpdateDB*, for the "OnClick" events of the Update or Cancel buttons. Basically, the user will be calling the action of the form (*ScheduleUI.csp*), [JT78]passing all the form data. However, if the user clicks the Update button, the *SetDB* flag is passed as "1", signifying that in our *ScheduleUI.csp* we will need to call on *SetDatabaseInfo.csp*.

The code for *SetDatabaseInfo.csp* looks like this:

```
<%
dbServer = Request.Form("dbServer")
dbName = Request.Form("dbName")
dbUID = Request.Form("dbUID")
dbpwd = Request.Form("dbpwd")
Set cLogons = ReportPlugin.ReportLogons
For each oLogon in cLogons
```

```
  oLogon.ServerName = dbServer
  oLogon.DatabaseName = dbName
  oLogon.UserName = dbUID
  oLogon.Password = dbpwd
Next
%>
```

The *SetDatabaseInfo.csp* [JT79]collects the form control values for server name, database name, user name and password and, creates the *cLogons* collection and loops through it, [JT80]setting the values for the matching properties of[JT81] each *oLogon* object. Why did we use two separate pages? The problem is that to accept input from the user, you need a form, and a form needs an action. Therefore, this means another page needs to be called from the form page. As a result, we have to perform a page transition, passing the form data. For this we use our trusty container page *ScheduleUI.csp*. Why not just put this code directly into *ScheduleUI.csp* rather than a separate file, such as *SetDatabaseInfo.csp*? The reason is in the way that the *SetDatabaseInfo.csp* is called (or loaded). Here is how *SetDatabaseInfo.csp* is handled within *ScheduleUI.csp*:

```
SetDB = Request.Form("SetDB")
If SetDB="1" then
%>
  <!-- #include file="SetDatabaseInfo.csp" -->
<%
End If
```

B[JT82]BBasically, this says if the *SetDB* variable (which is one of the values passed in from the *GetDatabaseInfo.csp*) is equal to "1", include the code from *SetDatabaseInfo.csp*. The benefit of this approach is that if the user wants to abandon any changes made to the database logon information, [JT83]the *SetDatabaseInfo.csp* page is not parsed (as *SetDB* will be "0").[JT84]

Adding Parameter functionality

For parameter values, we will again use a separate "get" and "set" page. Our "get" page will display any current values set for the parameter field, and post any new\modified values to *ScheduleUI.csp*, [JT85]which will "call on" the "set" page to handle the values. If you recall, the two main types of parameters are discrete values and range values each being able to handle single or multiple values depending on the definition.

If we break down our "get" page, we basically have four main parameter scenarios to handle: single discrete values, multiple discrete values, single range values and multiple range values. Another issue is that each parameter type could also contain default values, so in our "get" page we must provide a UI that will not only display, but also accept input relevant for each parameter type.

User Interface definition

We will use a similar parameter interface provided in ePortfolio. The specifications are:

Single discrete values:

An editable text box will be used for the value to be set. A drop-down list box will be used for any default values stored with the parameter. If a user selects a value from the drop-down list box, that value will automatically populate the text box. "Set Value" and "Clear Value" buttons are provided to either set the value currently in the text box or clear the parameter's current value.

Multiple discrete values:

An editable text box will be used to add the value. A drop-down list box will be used for any default values stored with the parameter. If a user selects a value from the drop-down list box, that value will automatically populate the text box, as the value in the text box is always used for the input value. An "Add Value" button will add the value contained in the text box to the parameter's current values collection. A (non-drop-down style) list box will be used to display current values for the parameter. A value can be removed by selecting it from within the list box and clicking the remove button.

Single range values:

For range values, [JT86]three controls need to be used for both the lower and upper bound values. A text box will be used to collect the actual value, and checkboxes will be used to collect the "Include this value" and "Range has no upper\lower bound" flags. If the parameter contains default or current values, the text boxes will be populated accordingly. A "Set Value" button is used to submit the new value, and a [JT87]"Clear Value" is provided to clear any current or default value.

Multiple range values:

A text box will be used to collect the actual value, and checkboxes will be used to collect the "Include this value" and "Range has no upper\lower bound" flags. If the parameter contains default or current values, the text boxes will be populated accordingly. An "Add Value" button will add the value (and appropriate flags) to the parameter's current values collection. A (non-drop-down style) list box will be used to display current values for the parameter (in range notation format). A range can be removed by selecting it from within the list box and clicking the remove button.

GetParamInfo.csp Code and Analysis

```
<HTML>
<HEAD><TITLE>Parameter Values</TITLE>
</HEAD>
<BODY>
<%
'begin section 1
Response.ExpiresAbsolute = Now() - 1
ParentReport = Request.QueryString("PReport")
ParentFolder = Request.QueryString("PFolder")
Response.Write("<FORM Name=the_form Method=Post Action=ScheduleUI.csp?PReport=" &
ParentReport & "&PFolder=" & ParentFolder & ">")
Response.Write("Report Parameter Information:")
Response.Write("<BR>")
paramnum=1
set oTheReport = session("cReports").Item(1)
set ReportPlugin = oTheReport.PluginInterface
Set cParams = ReportPlugin.ReportParameters
'end section 1
'begin section 2
For Each oParam in cParams
  Response.Write("Parameter Name: " & oParam.ParameterName)
  If oParam.Prompt <> "" then
    Response.Write("<BR>" & oParam.Prompt)
  End If
  Select Case oParam.ValueType
    Case 3
      Response.Write("<BR>Date Parameters Values entered in the form:  Date(YYYY,
MM, DD)")
```

```
    Case 4
      Response.Write("<BR>Time Parameters Values entered in the Form:  Date(HH,
MM, SS)")
    Case 5
      Response.Write("<BR>Date Time Parameters Values entered in the Form: 
DateTime(YYYY, MM, DD, HH, MM, SS)")
  End Select
  Response.Write("<BR>")
  If oParam.SupportsDiscreteValues Then
    If Not oParam.EnableMultipleValues Then
      If oParam.IsDefaultValueSet Then
        Set cCurrentValues = oParam.CurrentValues
        Response.Write("<select name='choice" & paramnum & "'
onChange='PopulateTextBox(this.options[selectedIndex].text,
document.the_form.selected" & paramnum & ")'>")
        For Each oEntry in oParam.DefaultValues
          Response.Write("<option>" & oEntry.Value)
        next
       Response.Write("</select>")
        Response.Write("<input name='selected" & paramnum & "' size=20>")
      Else
        Response.Write("<input name='selected" & paramnum & "' size=20>")
      End If
      Response.Write("<input type=button value='Set Value' OnClick='PutValue(" &
chr(34) & "selected" & paramnum & chr(34) & ", " & paramnum & ")'>")
      Response.Write("<input type=button value='Clear Value' OnClick='RemoveValue(" &
paramnum & ",0)'>")
      Response.Write("<BR>")
      If cCurrentValues.Count>0 then
        Response.Write("Current Value: " & cCurrentValues.Item(1).MakeDisplayString &
"<BR>")
      Else
        Response.Write("Current Value: <BR>")
      End If
'end section 2
'begin section 3
    Else
      If oParam.IsDefaultValueSet Then
        Set cCurrentValues = oParam.CurrentValues
        Response.Write("<select name='choice" & paramnum & "'
onChange='PopulateTextBox(this.options[selectedIndex].text,
document.the_form.selected" & paramnum & ")'>")
        For Each oEntry in oParam.DefaultValues
          Response.Write("<option>" & oEntry.Value)
        next
       Response.Write("</select>")
      End If
      Response.Write("<input name='selected" & paramnum & "' size=20>")
      Response.Write("<input type=button value='Add Item' OnClick='PutValue(" &
chr(34) & "selected" & paramnum & chr(34) & ", " & paramnum & ")'>")
      Response.Write("<input type=button value='Remove Item' OnClick='RemoveValue(" &
paramnum & ",document.the_form.tarea" & paramnum & ".selectedIndex" & ")'>")
      Response.Write("<BR>")
      Response.Write("Current Values: <BR>")
      Response.Write("<select name='tarea" & paramnum & "' size=4>")
      Set cCurrentValues = oParam.CurrentValues
      For each oCurrentValue in cCurrentValues
        Response.Write("<option>" & oCurrentValue.MakeDisplayString & "</option>")
      Next
      Response.Write("</select>")
    End If
'end section 3
'begin section 4
```

```
  ElseIf oParam.SupportsRangeValues Then
    Set cCurrentValues = oParam.CurrentValues
    If Not oParam.EnableMultipleValues Then
      Response.Write("Lower Limit:  ")
      If oParam.IsDefaultValueSet Then
        Response.Write("<input name='lower" & paramnum & "' size=25 Value='" &
oParam.DefaultValues.Item(1).Value & "'><BR>")
      Else
        Response.Write("<input name='lower" & paramnum & "'size=25" & "><BR>")
      End If
      Response.Write("Include This Value")
      Response.Write("<input type='checkbox' name='inclwr" & paramnum & "'
Value=true>")
      Response.Write("  No Lower Bound")
      Response.Write("<input type='checkbox' name='nolwr" & paramnum & "' Value=true
onclick='DisableCheckBoxes(document.the_form.nolwr" & paramnum & ",
document.the_form.inclwr" & paramnum & ",document.the_form.noupr" & paramnum &
")'><BR>")
      Response.Write("Upper Limit:  ")
      If oParam.IsDefaultValueSet Then
        Response.Write("<input name='upper" & paramnum & "'size=25 Value='" &
oParam.DefaultValues.Item(2).Value & "'><BR>")
      Else
        Response.Write("<input name='upper" & paramnum & "'size=25" & "><BR> ")
      End If
      Response.Write("Include This Value")
      Response.Write("<input type='checkbox' name='incupr" & paramnum & "'
Value=true>")
      Response.Write("  No Upper Bound")
      Response.Write("<input type='checkbox' name='noupr" & paramnum & "' Value=true
onclick='DisableCheckBoxes(document.the_form.noupr" & paramnum & ",
document.the_form.incupr" & paramnum & ",document.the_form.nolwr" & paramnum & ")'>")
      Response.Write("<input type=button value='Set Value' OnClick='PutValue(" &
chr(34) & chr(34) & "," & paramnum & ")'>")
      Response.Write("<input type=button value='Clear Value' OnClick='RemoveValue(" &
paramnum & ",0)'>")
      If cCurrentValues.Count>0 then
        Response.Write("<BR>Current Value: " &
cCurrentValues.Item(1).MakeDisplayString & "<BR>")
      Else
        Response.Write("<BR>Current Value: <BR>")
      End If
'end section 4
'begin section 5
    Else
      Response.Write("Lower Limit:  ")
      If oParam.IsDefaultValueSet Then
        Response.Write("<input name='lower" & paramnum & "' size=25 Value='" &
oParam.DefaultValues.Item(1).Value & "'><BR>")
      Else
        Response.Write("<input name='lower" & paramnum & "'size=25" & "><BR>")
      End If
      Response.Write("Include This Value")
      Response.Write("<input type='checkbox' name='inclwr" & paramnum & "'
Value=true>")
      Response.Write("  No Lower Bound")
      Response.Write("<input type='checkbox' name='nolwr" & paramnum & "' Value=true
onclick='DisableCheckBoxes(document.the_form.nolwr" & paramnum & ",
document.the_form.inclwr" & paramnum & ",document.the_form.noupr" & paramnum &
")'><BR>")
      Response.Write("Upper Limit:  ")
      If oParam.IsDefaultValueSet Then
```

```
        Response.Write("<input name='upper" & paramnum & "'size=25 Value='" &
oParam.DefaultValues.Item(2).Value & "'><BR>")
      Else
        Response.Write("<input name='upper" & paramnum & "'size=25" & "><BR> ")
      End If
      Response.Write("Include This Value")
      Response.Write("<input type='checkbox' name='incupr" & paramnum & "'
Value=true>")
      Response.Write("  No Upper Bound")
      Response.Write("<input type='checkbox' name='noupr" & paramnum & "' Value=true
onclick='DisableCheckBoxes(document.the_form.noupr" & paramnum & ",
document.the_form.incupr" & paramnum & ",document.the_form.nolwr" & paramnum & ")'>")
      Response.Write("<input type=button value='Add Value' OnClick='PutValue(" &
chr(34) & chr(34) & "," & paramnum & ")'>")
      Response.Write("<input type=button value='Remove Value' OnClick='RemoveValue("
& paramnum & ",document.the_form.tarea" & paramnum & ".selectedIndex" & ")'><BR>")
      Response.Write("<select name='tarea" & paramnum & "' size=4>")
      For each oCurrentValue in cCurrentValues
        Response.Write("<option>" & oCurrentValue.MakeDisplayString & "</option>")
      Next
      Response.Write("</select>")
    End If
  End If
  Response.Write("<BR><BR>")
  paramnum=cint(paramnum)+1
next
'end section 5
%>
<input type=hidden name=paramvalue value=''>
<input type=hidden name=paramnumber value=''>
<input type=hidden name=keepsetting value=''>
<input type=hidden name=remove value=''>
<input type=hidden name=removevalnumber value=''>
<Input Type=Hidden Name=SetParam Value=1>
<INPUT type=button Value="Done" onclick="done();">
<INPUT type=button Value="Reset All" onclick="abandon();">
</FORM>
<script language = "JavaScript">
<!-- hide me from non-functional browsers

function PopulateTextBox(choicevalue, textbox)
{
  if (choicevalue != "")
    textbox.value = choicevalue;
}
function PutValue(param, paramnum)
{
   document.the_form.paramvalue.value = param;
   document.the_form.paramnumber.value = paramnum;
   document.the_form.keepsetting.value = 1;
   document.forms["the_form"].submit();
}
function DisableCheckBoxes(control1, control2, control3)
{
  if (control1.checked == true)
  {
    control2.disabled = true;
    control3.disabled = true;
  }
  else
  {
    control2.disabled = false;
    control3.disabled = false;
```

```
  }
}
function AddRange()
{
  document.forms["the_form"].submit();
}
function RemoveValue(paramnum, index)
{
  if (index == -1)
  <%Response.Write("location.href = 'GetParamInfo.csp?PFolder=" & ParentFolder &
"&PReport=" & ParentReport & "';")%>
  else
  {
    document.the_form.removevalnumber.value = index;
    document.the_form.remove.value = 1;
    document.the_form.paramnumber.value = paramnum;
    document.the_form.keepsetting.value = 1;
    document.forms["the_form"].submit();
  }
}
function done()
{
<%
Response.Write("location.href = 'scheduleui.csp?PFolder=" & ParentFolder &
"&PReport=" & ParentReport & "';")
%>
}
function abandon()
{
<%
Response.Write("location.href = 'scheduleui.csp?PFolder=" & ParentFolder &
"&PReport=" & ParentReport & "&new=1" & "';")
%>
}
// show me -->
</script>
</BODY>
</HTML>
```

This is probably the most daunting CSP page we have looked at so far. It seems long and complex but really is not when you break it down. Let's look at what this code actually does. The code has been separated into sections for analysis. The beginning of each section in the code above is marked with a green arrow in the left margin for your reference.

Section 1:

The first few lines are pretty standard. They collect the query string parameters for folder and report id, and then set up the form to call *ScheduleUI.csp* (just like the Selection Formula and Database Logon pages) and create the *ReportPlugin* and *ReportParameter* collection (which we have called *cParams*). We have also initialized a parameter number counter (called *paramnum*) that we will use to keep track of the "nth" parameter we are currently working with. Then we create the *ReportPlugin* object, which we then use to create the parameters collection.

Section 2:

Then we start looping through the *cParams* collection. The For…Next loop is used to create the HTML interface (as defined previously) to collect values for each parameter. For each *oParam* object in the *cParams* collection,

we test the parameter type and create the HTML Form controls to collect and\or display the appropriate data for the parameter.

The first thing we do in the For…Each loop is to write the Parameter's name, then check to see if the parameter has any prompting text. If the parameter has prompting text, we display it. We then check the parameter type for date, time and date\time parameters. If the parameter is any of these types, we give the users instructions on the format for entering these values. This helps in absence of any prompting text[JT88], as date, time and date\time values must be entered in a specific way.

Once we have written out any instructional text to the users, we begin handling the parameters. Our first "if" condition tests the parameter's type. We handle discrete parameters first by testing the *SupportsDiscreteValues* property. If this property returns a value of "True", the parameter is a discrete value parameter (therefore not a range parameter). Then we check if the parameter supports multiple values so we know what form controls we need to create based on our UI specification. Once we determine that the parameter does not accept multiple values (*If Not oParam. EnableMultipleValues*), we then check to see if it has any default values. If the parameter contains any default values, we populate a drop-down list box with them, as we must provide the user with the choice to use those values.

The **<select>** tag is created with the name "choice" concatenated with the value of the *paramnum* variable. It also has an *OnChange* event that calls the JavaScript function "*PopulateTextBox(choicevalue, textbox)*" where "choicevalue" refers to [JT89]the value selected from the drop-down list [JT90]and "textbox" is the name of the text box control in which this value is to be placed. As an example, if the first parameter in the collection is a discrete value parameter[JT91] that has default values, a drop-down list box will be created with the tag:

```
<select name= 'choice1' onChange=PopulateTextBox(this.options[selectedIndex].text,
document.the_form.selected1)>
```

This tag means: "This is a drop-down list box called 'choice1' and when a value is selected from it, call the function *PopulateTextBox* passing text of the selected value and the text box control 'selected1'. We then loop through each value in the *DefaultValues* collection, writing an <option> tag for each, thus creating an entry in our drop-down list for each default value in the collection.

When we are done looping through the collection we then write a closing </select> tag. When finished with the drop-down, we create the text box form control to hold the value selected from the drop-down list box. We create it as *Selected1*, again [JT92]using the name "Selected" and appending the value of the *paramnum* variable. Incidentally, this is the form control passed as the second parameter for the JavaScript function *PopulateTextBox()*. The JavaScript function basically takes the textual value selected from the drop-down list box and makes it the value of the text box, if the value from the drop-down was not "" (an empty string).

Once we have the drop-down and text box controls, we add two buttons to the form. The first button is [JT93]labeled "Set Value" and [JT94]will be used to set the parameter's current value to the value currently in the text box. This is accomplished by the button's onClick event calling our JavaScript method *PutValue()* which takes the value of the parameter and the nth parameter number (what our *paramnum* value is equal to), sets the value of our "keepsetting" hidden control to "1", [JT95]and submits the form.

When you click the add button, you are really calling *ScheduleUI.csp*, passing the value you want to set for the parameter and the parameter's ordinal in the collection (value that *paramnum* is set to). Why? *ScheduleUI.csp* is designed to load our *SetParamInfo.csp* when it is called from *GetParamInfo.csp*. We do this [JT96]the same way that we did with the selection formula and database logon pages [JT97]with a hidden form control. In this case, it's[JT98] called *SetParam*. If *SetParam* is equal to "1" then *ScheduleUI.csp* "loads" (with a server-side include) *SetParamInfo.csp*, which we will analyze later. After the buttons have been created we then write the heading: "Current Value:" and display the parameter's current value, which will be the first item in the *CurrentValues* collection. We know it is the first value because this particular parameter did not support multiple values; therefore, the *CurrentValues* collection can only have one item.

Section 3:

From here we move on to the "Else" portion of our "If the parameter does not support multiple values" condition, which means it does support multiple values. We are basically going to perform the same set of steps as the previous case, except for two additions:

1. Add a non-drop-down list box to list all the current values.
2. Add functionality to remove a current value.

We start out by creating the *CurrentValues* collection, we check to see if the parameter has any default values. If so, [JT99]we create the drop-down list box containing the default values and create a text box to hold our selected (or typed) value, exactly the same as we did in the previous case.

This time, however, we create "Add Item" and "Remove Item" buttons, where "Add Item" [JT100]works exactly the same way as the "Set Value" button in the previous example (calls the JavaScript function "*PutValue*"), which takes the value of the parameter and the nth parameter number. The "Remove Value" button, however, calls the JavaScript function *RemoveValue()*. We pass the nth parameter number and the numerical index of the item that was clicked in the list box.

These values are (as in the previous case) posted to *ScheduleUI.csp*, where they will be handled by our *SetParamInfo.csp* page to remove the selected value from the *CurrentValues* collection. *SetParamInfo.csp* knows that this is a remove value request because of the hidden form control "remove" which is set to "1" by the *RemoveValue* function.

After the buttons are created, we create the (non-drop-down) list box used to hold the parameter's current values. In the previous case, we needed only to display the first item in the *CurrentValues* collection. In this case we have to loop through that collection of however many values there are. To avoid this [JT101] we create an opening select tag (dynamically named *tarea1*, where "1" comes from our nth parameter number variable) and for each value in the current values collection create an <Option> tag and call the value object's *MakeDisplayString* to get a string representation of the value. When we have iterated through the entire collection of current values we then close off the <select> tag.

It is important to note that when we click the add button, [JT102]the value does not really "get placed" in the drop-down by the Add button's JavaScript as it appears to. The value is sent to *ScheduleUI.csp*, where *SetParamInfo.csp* then adds that value to the *CurrentValues* collection. Then when we come back to *GetParamInfo.csp* and iterate through the *CurrentValues* collection and the value is [JT103]displayed in the list box. The same goes for the removal of a value. We pass the nth parameter number and the index of the current value to be removed, [JT104]and have *ScheduleUI.csp* call *SetParamInfo.csp* to remove it. Then the *GetParamInfo.csp* is called again to display the results and accept more input. We call *GetParamInfo.csp* again because of the hidden form control "KeepSetting", [JT105]which is set to "1" when the functions *PutValue()* and *RemoveValue()* are called.

With that, we have completed the handling of discrete value parameters, which means we are half[JT106]way done. Now we have to deal with the UI to collect single and multiple range parameter values. Then we will look at the code behind *SetParamInfo.csp*, [JT107]and will finally examine *ScheduleUI.csp* in its entirety. [JT108]

Section 4:

As with the discrete values, we check the parameter to see if it supports range values. Although there are not any other "types" of parameters besides discrete and range, it is still good to make certain. We then check if the parameter supports multiple values or not, by testing the negative first. If the parameter does not support multiple values then check to see [JT109]if a default value is present. If a default value is present we create the "Lower

Limit" input text box value and populate it with the default value, otherwise we leave[JT110] the "Lower Limit" input text box empty.

The text box is named *lower#* where "#" is the value of our nth parameter number variable "*paramnum*". We then create two checkboxes to collect the additional "flag" values, "Include this value" and "No Lower\Upper Limit", required for range parameters. In this case, they are named *inclwr#* and *nolwr#* respectively. We then[JT111] perform the same steps for the Upper value text box and checkboxes [JT112]naming them *upper#, incupr#,* and *noupr#* respectively.

You will notice from the code that the *nolwr#* (No Lower Value) and the *noupr#* (No Upper Value) checkboxes have "OnClick" events calling *DisableCheckboxes(),* passing the control's name as well as the name of two other controls. The *DisableCheckboxes()* function checks the state of the checkbox clicked and if it is "True" (checked), it [JT113]disables the two other controls passed to the function. We do this because of restrictions on range parameter values. For example, if the user checks the No Lower Value checkbox, the *DisableCheckBoxes()* function disables the "Include Lower Value" and "No Upper Value" checkboxes. This is because you cannot include the lower value if there is no lower limit. The "No Upper Value" checkbox is disabled because you cannot have an unbounded range (no lower value and no upper limit). So to prevent a user from trying this we disable the options.

Once the input controls are created (text box and two checkboxes for each of the lower and upper values of the range), we create the "Set Value" and "Clear Value" buttons. These buttons act exactly the same as in our discrete value parameter case. The "Set Value" button calls the *PutValue()* function which sets the "keepsetting" hidden control value to "1" and the "*paramnumber*" hidden control value to the nth parameter variable's value, [JT114]then submits the form.

The "Clear Value" button calls *RemoveValue()* which sets the "remove" hidden control value to "1" and also sets the "*paramnumber*" hidden control value to the nth parameter variable's value; then submits the form. Both would be handled, again, by *ScheduleUI.csp*, which would actually call *SetParamInfo.csp*. Finally, below the buttons, we will show the current value of the parameter. We do this by calling the *MakeDisplayString* method of the first (and only) item of the *CurrentValues* collection, which will display the value using range notation.

The last parameter type to be handled is the multiple range value parameter. The way we handle this parameter type is pretty much the same way as we handled multiple discrete value parameters, only by substituting the input fields for a range value.

Section 5:

We start out by checking if the parameter contains any default values. If a default value is present, [JT115]we create the "Lower Limit" text box populated with that value; otherwise, we create the text box as empty. From this point[JT116] we create the two "Lower Limit" flag value checkboxes, then we create the "Upper Limit" text, populating it with a default value if one is present;[JT117] otherwise, [JT118]creating it as [JT119]blank. Then the "Upper Limit" flag value checkboxes are created. Once the input controls are created, we add the action buttons. Since this is a multiple-value parameter, we create an "Add Value" button and a "Remove Value" button. The "Add Value" button calls the same *PutValue()* JavaScript function to set our hidden form control values and then submit the form; it passes an empty string for the first parameter.

If you recall, the first parameter passed to the *PutValue()* function is the actual value to be set for the parameter. A [JT120]range value is different, as a complete range parameter value consists of two actual values (Lower and Upper bound) and the two flags (Include Value or No Bound) for each value. We cannot pass all of this information into the parameter for the *PutValue()* function, so instead we rely on *PutValue()* submitting the form controls containing these values in to *ScheduleUI.csp*, [JT121]where we will use *SetParamInfo.csp* to collect and manipulate the values. The "Remove" button, however, works exactly the same way [JT122]as it did in the discrete value case, as we do not deal directly with the value of the parameter. Once the buttons are set up, we create a (non-drop-down) list box to show any current range values for the parameter. We do this by creating a <select>

tag and then looping for each *CurrentValue* object in the *CurrentValues* collection, calling the object's *MakeDisplayString* method to display a string representation of the parameter's value. In this case, the string is [JT123]in the form of range notation.

We then finish the page with the HTML required to create the hidden form controls (set by the JavaScript functions). This is done using both [JT124]a "Done" button, calling the JavaScript function *Done(),* that redirects us back to the *ScheduleUI.csp,* [JT125]and a "Reset All" button, calling the JavaScript function *Abandon(),* that redirects us back to the *ScheduleUI.csp* page with "new=1" attached to the query string. This tells *ScheduleUI.csp* that this is a new schedule request, which effectively clears anything that has been set for the schedule so far (including all current values that have been set for parameters). The rest of the page is the code for all of the JavaScript functions that we use to handle the various form control events.

Now we have analyzed the *GetParamInfo.csp* from top to bottom. The main thing to remember about this page is that it is designed to create a UI required to collect values for the parameters in the report. Since there are different types of parameters, each with different handling requirements, the UI we create depends on the parameter. *GetParamInfo.csp* creates HTML that allows the user to enter information and submit that information to *ScheduleUI.csp*, which will then call *SetParamInfo.csp* to handle the submitted information and set the parameters accordingly.

SetParamInfo.csp Code and Analysis

```
<%
removevalue = Request.Form("remove")
paramnumber = Request.Form("paramnumber")
Dim rngParams()
ReDim rngParams(1)
Dim rngParamsOptions()
ReDim rngParamsOptions(3)
Set cParams = ReportPlugin.ReportParameters
Set oParam = cParams.Item(cint(paramnumber))
Set cCurrentValues = oParam.CurrentValues
If removevalue="1" then
       valuetoremove = Request.Form("removevalnumber")
       call cCurrentValues.Delete(cint(valuetoremove)+1)
Else
       If oParam.SupportsDiscreteValues then
                     torequest = Request.Form("paramvalue")
                     paramvalue = Request.Form(torequest)
       If Not oParam.EnableMultipleValues then
              cCurrentValues.Clear
       End If
       Set oCurrentValue = oParam.CreateSingleValue
       oCurrentValue.Value = cstr(paramvalue)
       cCurrentValues.Add(oCurrentValue)
  Else
If oParam.SupportsRangeValues then
             rngParams(0) = Request.Form("lower" & paramnumber)
             rngParams(1) = Request.Form("upper" & paramnumber)
If Request.Form("inclwr" & paramnumber) <> "" then
       rngParamsOptions(0) = "true"
Else
       rngParamsOptions(0) = "false"
End if
If Request.Form("nolwr" & paramnumber) <> "" then
       rngParamsOptions(1) = "true"
Else
       rngParamsOptions(1) = "false"
End if
If Request.Form("incupr" & paramnumber) <> "" then
```

```
       rngParamsOptions(2) = "true"
Else
                  rngParamsOptions(2) = "false"
            End if
      If Request.Form("noupr" & paramnumber) <> "" then
             rngParamsOptions(3) = "true"
Else
             rngParamsOptions(3) = "false"
      End if
      If Not oParam.EnableMultipleValues then
        cCurrentValues.Clear
      End If
      Set oCurrentValue = oParam.CreateRangeValue
      oCurrentValue.FromValue.Value = rngParams(0)
      oCurrentValue.IncludesLowerBound = rngParamsOptions(0)
      oCurrentValue.HasNoLowerBound = rngParamsOptions(1)
      oCurrentValue.ToValue.Value = rngParams(1)
      oCurrentValue.IncludesUpperBound = rngParamsOptions(2)
      oCurrentValue.HasNoUpperBound = rngParamsOptions(3)
      cCurrentValues.Add(oCurrentValue)
    End If
  End If
End If
%>
```

SetParamInfo.csp is not quite as large as *GetParamInfo.csp*, and it does not create any HTML either, which makes it somewhat easier to read. *SetParamInfo.csp* collects most of the information sent from *GetParamInfo.csp* to handle setting the parameter values. Some information coming from *GetParamInfo.csp* is "state" information or flags created [JT126]specifically to tell *ScheduleUI.csp* what to do. *SetParamInfo.csp* does not collect or use this information.

We start out by collecting the "remove" and "paramnumber" form control values. The next few declarations are designed [JT127]to handle range parameter values. Since a range parameter value consists of six separate pieces of information (a lower bound, flag whether to include the lower value, flag to set no lower bound, an upper bound, flag whether to include the upper value, flag to set no upper bound) we have to associate each of these pieces of information together as a range parameter "value". To handle this kind of organization of the values, [JT128]we use two arrays to capture and handle the data. We create a "value" array, and an "options" array. The "value" array will be used to hold the upper and lower bound values, and the "options" array will be used to hold the "flag" values.

Therefore, we create (dimension) an array called *RngParams* and then re-dimension the array to have two elements (arrays are zero-based). This will hold our lower value and upper value. We then do the same steps for the *RngOptions* array, [JT129]only re-dimensioning it to four elements. This array will hold the four "flag" values (2 lower value, and 2 upper value).

Then we start working with the parameter. We create the parameters collection (calling it *cParams*) using the Report Plug-in. Notice, though, [JT130]that we did not create the Report Plug-in[JT131]. That is because *SetParamInfo.csp* is "included" into *ScheduleUI.csp* when it is needed. As a result, the code becomes "inline" with the rest of the code in *ScheduleUI.csp*. The *ReportPlugin* object is created in *ScheduleUI.csp* so we can use it. Once we have the collection created, [JT132]we need to instantiate the parameter object for which [JT133]we want to set the value [JT134]. This is where the parameter index value that we kept passing around in *GetParamInfo.csp* becomes necessary[JT135]. The numeric index of the[JT136] parameter we were dealing with in *GetParamInfo.csp* was passed over in the form control "paramnumber". We use the *Item* property of the *cParams* collection (and the paramnumber variable) to create the parameter object we want to modify. From here, we create the *CurrentValues* collection so that we can add to or remove from it.

Now we test to see [JT137]if we are removing a value or adding a value, using the "removevalue" variable. If "removevalue" was set to "1", [JT138]that means the user clicked the "Clear Value" or "Remove Value" button, which means that we need to delete a current value. So if "removevalue" is equal to "1", we request the form control "removevalnumber", which identifies which value was selected from the list box of current values. We then call the *Delete* method of the *CurrentValues* collection, passing the value of our "removevalnumber" variable plus one. The reason why we increment by one is that the *CurrentValues* collection is "one-based" (meaning the first value is item 1), whereas the list box where the current values were displayed is "zero-based" (meaning the first value is item 0). The item in the *CurrentValues* collection will always be one higher than the index of the selection from the list box.

If *removevalue* is not equal to "1" it means that we are adding a new value to the *CurrentValue* collection. We then test what type of parameter object we are dealing with, testing the discrete value case first. If the parameter supports discrete values, we create a variable called "*torequest*" and populate it with the value of the form control "*ParamValue*". Then we create a variable called "*paramvalue*" and request the value from the form control represented by our "*torequest*" variable. Why is it done this way?

Recall how our *GetParamInfo.csp* worked when you clicked the "Set Value" or "Add Value" buttons. When the button was clicked, the name of the text box control (and the index of what parameter we were setting) was passed to the function *PutValue()* that set the hidden form controls "*ParamValue*" and "*ParamNumber*". Therefore, the form control *ParamValue* contains the name of the text box that contains the value we want. Since all form control values are sent when we submit the form, we figure out which form control has the value we actually want this way. We ask for the name of the text box that has the value, and then we ask for the value of that text box.

Once we have the value from the text box, [JT139]we test if the parameter supports multiple values. If the parameter does not support multiple values, we call the *Clear* method of the *CurrentValues* collection. This removes any values that are already set. We only want to do this if the parameter does not support multiple values, [JT140]though. Then we create a new *SingleValue* (discrete value) object, set the *CurrentValue* object's *Value* property to our "*paramvalue*" variable, and finally we call the *CurrentValue* collection's *Add* method to add this value to the collection. That is a discrete value set.

To our "Else" case now:[JT141] if the parameter supports range values, we have to request all the form data that comprises the range values. This represents a unique challenge because we used checkboxes in our form. A checkbox form control only reports its value in the "True" (or checked) case. Said another way, if you have a checkbox form control and do not check it, the value is not sent. This means we have to collect all the values passed from the form but also trap the ones that were not passed (unchecked checkboxes).

To that end, we have to be creative in how we collect the information passed in from the form. We not only have to request what was passed in, but we have to look for what was not passed in. We start by[JT142] requesting the lower and upper bound values. Notice that we use *Request.Form* passing in the value of "lower" or "upper" concatenated to the value of the "*paramnumber*" variable. This is so that we request the correct form controls just in case there were multiple range parameters displayed on the *GetParamInfo.csp* page.

Then we move on to the flag values. We need values for the controls *inclwr#, nolwr#, incupr#,* and *noupr#.* Where "#" is the value of the "*paramnumber*" variable, but because these controls were checkboxes, they may or may not be present (depending if their values were "True" or not). So we test if the *Request.Form* of *inclwr#* is not equal to "" (an empty string). If the value for *inclwr#* is not an empty string, that means the checkbox was checked so we set the array value for the "include lower value" flag to "True". If the checkbox was not checked, we set the value to "False". We then perform these same steps for the other three flag values.

Once we have all the range value "parts" stored in our arrays, we test to see [JT143]if the parameter supports multiple values. If the parameter does not support multiple values, we call the *Clear* method of the *CurrentValues* collection, which removes any values that are already set. We only want to do this if the parameter does not support multiple values though. We then [JT144]create a new *RangeValue* object, and set the properties of the

range value objects to the appropriate values stored in our arrays. Finally, we call the *CurrentValue* collection's *Add* method to add this value to the collection. We are now done with handling parameters.

Review of Parameter Handling

Now that we have completed the analysis of both *GetParamInfo.csp* and *SetParamInfo.csp*, we should take a step back and review how they work together. At a high level, *GetParamInfo.csp* iterates through the collection of parameters, [JT145]creating the UI necessary to add or delete values from the parameter. When you click the Set, Add, Clear or Remove buttons in the HTML created by *GetParamInfo.csp*, [JT146]the form information collected on that page is posted to *ScheduleUI.csp*. *ScheduleUI.csp* then calls *SetParamInfo.csp*. *SetParamInfo.csp* figures out whether you want to delete a value, and deletes that value by its ordinal position in the *CurrentValues* collection. If you are not deleting a value, you are adding one. In which case *SetParamInfo.csp* determines what type this parameter is (discrete or range value), collects the required form data, clears the collection (if the parameter does not support multiple values), creates a new value object (of the required type), sets the new value object's properties to the collected form data, and adds the new value object to the *CurrentValues* collection.

Bringing it all together

We have looked at how Selection Formula, Database Logon and Parameter Fields are handled in their respective pages, and [JT147]we have briefly mentioned our "container" page that manages each of these other pages and coordinates the requests. Now we will [JT148]look at the code behind *ScheduleUI.csp* and bring our scheduling application together.

Revised *ScheduleUI.csp* Code and Analysis

```
<HTML>
<HEAD>
<TITLE>Schedule Report</TITLE>
</HEAD>
<BODY>
<%
Response.ExpiresAbsolute = Now() - 1
ParentReport = Request.QueryString("PReport")
ParentFolder = Request.QueryString("PFolder")
NewSchedule = Request.QueryString("New")
KeepSettingParams = Request.Form("KeepSetting")
Response.Write("<a href='listreports.csp?PFolder=" & ParentFolder & "'>Back to
Reports List</a><br>")
SetDateTime = Request.Form("SetDateTime")
SetParam = Request.Form("SetParam")
SetDB = Request.Form("SetDB")
apstoken = Request.Cookies("logontoken")
Set SessionManager = CreateObject("CrystalEnterprise.SessionMgr")
Set oEnterpriseSession = SessionManager.LogonWithToken(apstoken)
Set oiStore = oEnterpriseSession.Service("", "InfoStore")
ReportQuery = "Select * from CI_INFOOBJECTS where SI_OBTYPE = 2 AND SI_ID=" &
cint(ParentReport)
If NewSchedule = "1" then
       set session("cReports") = nothing
       set session("cReports") = oiStore.Query(cstr(ReportQuery))
End If
set oTheReport = session("cReports").Item(1)
Set SchedulingInfo = oTheReport.SchedulingInfo
If SetDateTime = "1" then
       %>
       <!-- #include file="SetDateTime.csp" -->
       <%
       End If
```

```
        %>
        <!-- #include file="GetDateTime.csp" -->
        <%
        set ReportPlugin = session("cReports").Item(1).PluginInterface
        'Handle Parameters:
        If SetParam="1" then
                %>
                <!-- #include file="SetParamInfo.csp" -->
                <%
        End If
If KeepSettingParams = "1" then
Response.Redirect("GetParamInfo.csp?PFolder=" & ParentFolder & "&PReport=" &
ParentReport)
End If
If ReportPlugin.ReportParameters.Count > 0 Then
Response.Write("<a href='GetParamInfo.csp?PReport=" & ParentReport & "&PFolder=" &
ParentFolder & "'>Click Here for Parameter Information</a><BR><BR>")
End If
'Handle Selection Formula
sfchanged = Request.Form("sfchanged")
If sfchanged=1 then
        NewRecordSF = Request.Form("recordsf")
        NewGroupSF = Request.Form("groupsf")
        ReportPlugin.RecordFormula = NewRecordSF
        ReportPlugin.GroupFormula = NewGroupSF
End If
Response.Write("<a href='SelectionFormula.csp?PReport=" & ParentReport & "&PFolder="
& ParentFolder & "'>Click Here for Selection Formula</a><BR><BR>")
'Handle Database
If SetDB="1" then
        %>
        <!-- #include file="SetDatabaseInfo.csp" -->
        <%
End If
If ReportPlugin.NeedsLogon then
Response.Write("<a href='GetDatabaseInfo.csp?PReport=" & ParentReport & "&PFolder=" &
ParentFolder & "'>Click Here for Database Logon Information</a>")
End If
%>
<BR><BR><BR>
<script language = "JavaScript">
function Schedule()
{
<%
  Response.Write("location.href='Schedule.csp?PReport=" & ParentReport & "&PFolder="
& ParentFolder & "'")
%>
}
</script>
</BODY>
</HTML>
```

As usual, we start out by collecting form information passed on from the previous page. We are mainly looking for form data that identifies which page (*SetDateTime.csp*, *SetDatabaseInfo.csp* or *SetParamInfo.csp*) called *ScheduleUI.csp* so we know what scheduling information we are setting. We first request the parent folder parent report information, so that [JT149]we know what folder the call came from and what report we are scheduling. Then we request the "New" form parameter, to see if this is a new schedule request. Then we get the "*KeepSetting*" parameter, which tells us if we are setting parameters and are not finished yet.

We write a hyperlink to go back to the reports list (in case the user wants to exit the schedule interface), and then we collect the "set" parameters. Each of the *Set*.csp* pages passes a "set" parameter to *ScheduleUI.csp* in their form. So if *SetDateTime.csp* was used to call *ScheduleUI.csp*, the form parameter *SetDateTime* would be set to "1". We collect the *SetDateTime*, *SetParam* and *SetDB* parameters so we can find out which page called *ScheduleUI.csp*. From here, we load the CE SDK, logon to the APS with the user's logon token, and create the *InfoStore* object. We then [JT150]set up a query to select the report that we want to schedule from the *InfoStore*, but we do not execute the query yet.

Now we start using that state information that we collected [JT151]from the calling form. We first check the "*NewSchedule*" variable, which [JT152]tells us if the "New" form parameter was set to one, indicating that this was a new schedule request. If the "*NewSchedule*" variable is equal to "1" we destroy the existing report collection stored in a session variable, and query the APS for the collection again. This is our first use of a session variable. The reason for this approach is that our report collection (our report object) needs to persist across multiple pages manipulating its properties. This is the way we discard any previous settings. Once we have created the collection, or preserved the existing one[JT153], we create the report object, [JT154]the scheduling info which holds date, time and recurrence information.

Now we start checking to see which page called *ScheduleUI.csp*. We first check the *SetDateTime* variable. If *SetDateTime* is equal to "1" it means that the user clicked the Set Date\Time button on the form created by *GetDateTime.csp*. If this is the case then we include the code for *SetDateTime.csp* to handle requesting and setting the date, time and recurrence information for the schedule. We then [JT155]include *GetDateTime.csp* to display the current date, time and recurrence information for the schedule.

Then we move on to check the parameter information. We create the *ReportPlugin* object and check the *SetParam* variable. If *SetParam* is equal to "1" it means that the user clicked a "Set" or "Add" button created by *GetParamInfo.csp*. This means that [JT156]we include the code from *SetParamInfo.csp* to handle the request. Once *SetParamInfo.csp* is done executing, we check the "*KeepSetting*" variable to see if it is "1". If so, [JT157]this means that the user did not click the "Done" button from *GetParamInfo.csp*, meaning that[JT158] they are not finished setting parameter information. Therefore, we redirect the browser back to *GetParamInfo.csp*.

Then we check to see [JT159]if the *ReportPlugin* parameter collection count is greater than "0". If the parameter collection count is greater than "0", it means that the report has parameter fields and we need to create a hyperlink to call *GetParamInfo.csp*. [JT160]Checking to see if the report has parameters after we call on the page to process parameter input may seem a bit backwards. Nevertheless, the key here is that not all parts of *ScheduleUI.csp* are processed each time it is called. The *SetParamInfo.csp* will only be called if our *SetParam* variable is set to "1". What our *ScheduleUI.csp* page does really depends on what page called it, and more specifically, which flag variables were set. The other side of the coin [JT161]is that we want to make any updates (setting) before displaying (getting) the list to preserve consistency.

Now we check the selection formula, starting with [JT162]the "*sfchanged*" flag. If "*sfchanged*" is equal to "1" it means that the user clicked the "Update" button on the *SelectionFormula.csp*. If this is the case, then we request the new record and group selection formulas from the form and set the appropriate *ReportPlugin* properties to these new values. Then we write a link to the *SelectionFormula.csp* page to allow the user to get and set the values. Again, we set the value (if need be) before allowing the user to get the value and change it. Then we move on to the database information, checking [JT163]the *SetDB* flag to see if it is set to "1". If it is, [JT164]it means that the user clicked the "Update" button in the *GetDatabaseInfo.csp*, so we include *SetDatabaseInfo.csp* to handle that request. We then check to see [JT165]if the *ReportPlugin NeedsLogon* property is "True". If this is so, it means that the report has database logon requirements, and we need to create a hyperlink to call *GetDatabaseInfo.csp*. All of this is very similar to the way we handled parameters.

Finally, we finish off the *ScheduleUI.csp* page with some HTML. We create the JavaScript code to drive the "Schedule" button created for this page's form (which is actually created in *GetDateTime.csp*). When clicked, the Schedule button calls on the function that directs the browser to *Schedule.csp*.

Revised *Schedule.csp* Code and Analysis

```
<%
ParentFolder = Request.Querystring("PFolder")
ParentReport = Request.Querystring("PReport")
'Retrieve the logon cookie
apstoken = Request.Cookies("LogonToken")
If apstoken = "" then
       response.redirect("LogonForm.csp")
End If
Set SessionManager = CreateObject("CrystalEnterprise.SessionMgr")
Set oEnterpriseSession = SessionManager.LogonWithToken(apstoken)
Set oiStore = oEnterpriseSession.Service("", "InfoStore")
oiStore.Schedule(session("cReports"))
set session("cReports") = nothing
Response.Redirect("listinstances.csp?PReport=" + cstr(ParentReport) + "&PFolder=" +
cstr(ParentFolder))
%>
```

Schedule.csp is basically used to commit the schedule to the APS. *ScheduleUI.csp*, *GetDateTime.csp*, **ParamInfo.csp*, **DatabaseInfo.csp* and *SelectionFormula.csp* are all used to get and set various properties of the scheduled job, but *Schedule.csp* is the one that actually "schedules" it. *Schedule.csp* gets the ID[JT166] of the current folder and report from the query string and [JT167]collects the user's logon token cookie. Then we load the CE SDK, logon to the APS with the user's logon token, and create the *InfoStore* object. Once the *InfoStore* has been created, [JT168]we call the Schedule method, passing our collection (which is stored in a session variable). The session variable is then destroyed, as we are done with it. We then redirect to our *ListInstances.csp* so the user can check the status of the newly scheduled job.

Conclusion

In this document, we have examined the functionality of the *ReportPlugin* object. The *ReportPlugin* allows us to get and set various properties of a report that are stored on the APS. We can access record and group selection formula, database location and logon, and report parameter information. We also looked at how we could develop a scheduling application that makes use of the *ReportPlugin* object to get and set these properties. The scheduling page described in this document is for demonstration purposes.

Getting More Information

For more information, review the following documentation or contact Technical Support.

Product Documentation

Available for download from the Crystal Decisions Support Site (http://support.crystaldecisions.com/docs) in portable document format (PDF):

- ASP vs. CSP: Deployment Issues (CE8_ASP_vs_CSP.pdf)
- Understanding Session Handling within Crystal Enterprise (CE8_Session_Handling.pdf)

Contacting Crystal Decisions for Technical Support

We recommend that you refer to the product documentation and that you visit our Technical Support web site for more resources.

Self-serve Support:

http://support.crystaldecisions.com/

Email Support:

http://support.crystaldecisions.com/support/answers.asp

Telephone Support:

http://www.crystaldecisions.com/contact/support.asp

****

Page: 14
[JT1]comma
Page: 15
[JT2]should read, For example, the following code…a single report object:
Page: 15
[JT3]comma
Page: 16
[JT4]delete
Page: 16
[JT5]semi-colon
Page: 16
[JT6]comma
Page: 16
[JT7]delete
Page: 16
[JT8]by?
Page: 16
[JT9]comma
Page: 16
[JT10]comma
Page: 17
[JT11]for?
Page: 17
[JT12]Delete, add a comma, so
Page: 17
[JT13]delete period, add a comma, so
Page: 17
[JT14]comma
Page: 17
[JT15]just Next
Page: 17
[JT16]comma
Page: 17
[JT17]comma
Page: 18
[JT18]
Page: 18
[JT19]In order to set a current value, you have to . . .
Page: 18
[JT20]delete
Page: 18
[JT21]actually
Page: 18
[JT22]Finally,
Page: 18
[JT23]comma
Page: 18
[JT24]comma
Page: 18
[JT25]comma
Page: 18
[JT26]Finally,
Page: 18
[JT27]either parameter value object; or ParameterValue object
Page: 18
[JT28]comma

Page: 19
[JT29]delete comma
Page: 19
[JT30]either ValueObject, or value object
Page: 19
[JT31]comma
Page: 19
[JT32]DiscreteValue?
Page: 19
[JT33]ParameterValue or parameter value
Page: 19
[JT34]delete comma
Page: 19
[JT35]as above
Page: 19
[JT36]comma
Page: 20
[JT37]comma
Page: 20
[JT38]comma
Page: 20
[JT39]delete extra space
Page: 20
[JT40]existing
Page: 20
[JT41]Finally,
Page: 20
[JT42]comma
Page: 20
[JT43]existing
Page: 20
[JT44], then
Page: 20
[JT45]comma
Page: 20
[JT46]finally we will
Page: 20
[JT47]consistency
Page: 21
[JT48]However,
Page: 21
[JT49]remain
Page: 21
[JT50]comma
Page: 21
[JT51]colon
Page: 21
[JT52]comma
Page: 21
[JT53]At this point, I have stopped correcting your spelling errors, as these have been highlighted for you by Spell Check. Please correct them yourself.
Page: 21
[JT54]comma
Page: 21
[JT55]delete comma

Page: 21
[JT56]delete comma
Page: 24
[JT57]semi-colon
Page: 24
[JT58]comma
Page: 24
[JT59]this time,
Page: 24
[JT60]delete, keep comma
Page: 24
[JT61]delete period, sounds like. add comma, forming…
Page: 24
[JT62]comma
Page: 24
[JT63]comma
Page: 25
[JT64]This means that…
Page: 25
[JT65]delete period, add comma both of which will be handled on the same page.
Page: 25
[JT66]Delete extra space
Page: 26
[JT67]comma
Page: 26
[JT68]at
Page: 26
[JT69]will
Page: 26
[JT70]delete period, add comma, which
Page: 26
[JT71]delete
Page: 26
[JT72]comma
Page: 26
[JT73]In order
Page: 26
[JT74]also
Page: 26
[JT75]comma
Page: 26
[JT76]semi-colon
Page: 27
[JT77]semi-colon
Page: 27
[JT78], comma
Page: 28
[JT79]which what? Use “this” or “this code”
Page: 28
[JT80]comma
Page: 28
[JT81]delete
Page: 28
[JT82]This
Page: 28
[JT83]comma

Page: 28
[JT84]delete.
Page: 28
[JT85]Comma, space
Page: 29
[JT86]comma
Page: 29
[JT87]delete period, add comma, and a….
Page: 34
[JT88]delete
Page: 34
[JT89]refers to the value…?
Page: 34
[JT90]dropdown list
Page: 34
[JT91]delete comma
Page: 34
[JT92]delete
Page: 34
[JT93]is
Page: 34
[JT94]and
Page: 34
[JT95]comma
Page: 34
[JT96]delete comma
Page: 34
[JT97]delete comma, add -
Page: 34
[JT98]it’s
Page: 35
[JT99]comma
Page: 35
[JT100]delete extra space
Page: 35
[JT101]To avoid this, we create…
Page: 35
[JT102]comma
Page: 35
[JT103]delete
Page: 35
[JT104]comma
Page: 35
[JT105]comma
Page: 35
[JT106]delete space
Page: 35
[JT107]comma
Page: 35
[JT108]delete
Page: 35
[JT109]to see
Page: 35
[JT110]leave
Page: 36
[JT111]delete

Page: 36
[JT112]comma
Page: 36
[JT113] comma, it disables…
Page: 36
[JT114]comma
Page: 36
[JT115]comma
Page: 36
[JT116]this point
Page: 36
[JT117]semi-colon
Page: 36
[JT118]comma
Page: 36
[JT119]as
Page: 36
[JT120]Delete But, capitalize a
Page: 36
[JT121]comma
Page: 36
[JT122]way
Page: 36
[JT123]is
Page: 36
[JT124]both
Page: 37
[JT125]semicolon
Page: 38
[JT126]created
Page: 38
[JT127]designed
Page: 38
[JT128]comma
Page: 38
[JT129]comma
Page: 38
[JT130]comma though comma
Page: 38
[JT131]delete
Page: 38
[JT132]comma
Page: 38
[JT133]delete, for which
Page: 38
[JT134]delete
Page: 38
[JT135]becomes necessary
Page: 38
[JT136]delete which, add the
Page: 38
[JT137]to see
Page: 38
[JT138]comma
Page: 39
[JT139]comma

Page: 39
[JT140]comma
Page: 39
[JT141]colon
Page: 39
[JT142]delete with, add by
Page: 39
[JT143]To see
Page: 39
[JT144]We then
Page: 39
[JT145]comma
Page: 40
[JT146]comma
Page: 40
[JT147]comma and
Page: 40
[JT148]will
Page: 41
[JT149]that
Page: 41
[JT150]We then
Page: 41
[JT151]delete
Page: 41
[JT152]which
Page: 42
[JT153]delete
Page: 42
[JT154]comma, followed by the scheduling…
Page: 42
[JT155]We then
Page: 42
[JT156]This means that …
Page: 42
[JT157]comma
Page: 42
[JT158]comma, meaning that…
Page: 42
[JT159]to see
Page: 42
[JT160]delete – see below
Page: 42
[JT161]of the coin, or ..the other point
Page: 42
[JT162]comma, starting with the …
Page: 42
[JT163]comma, checking the…
Page: 42
[JT164]comma
Page: 42
[JT165]to see
Page: 43
[JT166]ID
Page: 43
[JT167]The what? collects…

Page: 43
[JT168]comma