Create Your Own Report Connector

Last Updated: 15-December-2009.

The URS Installation Guide documents how to compile your own URS Report Connector. This document provides a guide to what you need to create in your connector code. We'll use the XtraReportsConnector as a starting point.

Creating A Report Connector

This is the entire version 1.1 Report Connector code for DevExpress XtraReports, with commentary:

```
/*
 * Copyright (c) 2008-2009 by VersaReports, LLC.  All rights, including the rights to copy and
distribute, are reserved.
 */
using System;
using System.Collections.Generic;
using System.Text;
using System.Collections.Specialized;
using VersaReports.ReportConnector;

using System.IO;
using System.Drawing;
using System.Reflection;
using DevExpress.XtraReports.UI;
using DevExpress.XtraPrinting;
using DevExpress.XtraReports.Parameters;
```

The above highlighted lines are going to change based on the requirements of the report designer that you are using.

The next section defines the report connector class. The highlighted inheritance is required and provides the properties you'll need to work with the report to be run.

```
namespace XtraReportsConnector
{
    public class XtraReportsDllConnector : ReportsConnectorInterface
    {
        /***** Public inherited properties
            public string ReportFileLocation = Report file location (if stored on disk);
            public byte[] ReportFileContents = Report contents (if stored in database);
            public string[] ExtraDllsToLoad = Full paths of extra DLLs to load (if needed);
            public string ExtraInfo = extra info defined when report was configured
            public string DataSourceInfo = Data source information (if needed);
            public string UserName = user name for report's database connection (if needed)
            public string Password = password for report's database connection (if needed)
            public string ReportName = The report's name;
            public string ReportDescription = The report's description;
            public string ScheduleName = The name of the schedule using this report;
            public string SaveFormat = The format of the report output to generate (defined in
the configuration files);
            public List<ReportParameterValue> Parameters = list of report parameter values
(include description, type, and name of parameter in report)
        */
```

The properties are listed in the comments, but let's provide some more depth:

- **ReportFileLocation** - the location that was selected by the report administrator for the report file (e.g., the DLL or RPT file). If reports of this type are stored in the database, this value will be the name of the file and you are expected to use the report contents stored as a byte array in **ReportFileContents** (see below). If **ReportFileContents** is null, then you are expected to use the report contents stored at the location specified by **ReportFileLocation**.
- **ReportFileContents** - the actual contents of the report that the report administrator uploaded when the report was defined to URS. If reports of this type are stored on disk, this value will be null and you are expected to get the report's contents via the **ReportFileLocation** property.
- **ExtraDllsToLoad** - an array of strings listing absolute paths to any DLLs that are going to be needed to run a report of this type. This array gets set if the site administrator did not load these DLLs into the GAC.
- **ExtraInfo** - this is the string provided by the report administrator who administers this report and is something the report connector writer will want to define. For example, if your report is stored in a DLL, this property might define the fully qualified report class name to instantiate.
- **DataSourceInfo**, **UserName**, **Password** - these are used for reports that can vary their data source (e.g., Crystal Reports allows you to change the data source at run-time). The structure for **DataSourceInfo** -- like **ExtraInfo** -- is defined by the report connector writer; **UserName** and **Password** are the user name and password for the data source connection and are specified only if needed.
- **ReportName** - the name of the report as defined by the report administrator when the report is created within URS.
- **ReportDescription** - the description string defined by the report administrator when the report is created within URS.
- **ScheduleName** - the name of the schedule defined by the report's scheduler when the report is scheduled within URS.
- **SaveFormat** - this property is a string defining the finished report format to be created. The possible choices are defined in the report_types.xml file for each type of report that can be run by URS. Often, this property will contain the file extension for the resulting report output file (e.g., *pdf*); this string is limited to 10 characters.
- **Parameters** - a *List* of type *ReportParameterValue* that contains the parameters and values to be passed to the report as defined by the report scheduler for this report. *ReportParameterValue* is a class containing the following properties:
    - **Description** - the prompt string used to request the value from the report scheduler.
    - **ParameterNameInReport** - the string containing the parameter's name as specified in the report to be run.
    - **Type** - an enumeration which can be one of *Boolean*, *Integer*, *Float*, *String*, C*hoices*, *Date_And_Time* and specifies the type of parameter that was defined in URS (which might be a different type than the report's parameter needs, so you'll want to be careful of that).
    - **Value** - an object containing the actual value to be passed to the report. The object's value will match what the **Type** says it will be, with the exception that a *Choices* parameter results in a *String* value.

CreateReport is the first routine that you will override and is the main routine to run a report:

```
        public override bool CreateReport(out string errorMessages, out byte[] results)
        {
            errorMessages = string.Empty;
            results = new byte[] { };
            string fileName = string.Empty;
```

The first step is to get the report contents into a place where you can actually run them:

```
            try
            {
                try
                {
                    if (ReportFileContents != null)
                    {
                        fileName = Path.GetTempFileName();
                        File.Move(fileName, Path.ChangeExtension(fileName,
Path.GetExtension(ReportFileLocation)));
                        File.WriteAllBytes(fileName, ReportFileContents);
                    }
                    else
                    {
                        fileName = ReportFileLocation;
                    }
                }
                catch (Exception ex)
                {
                    errorMessages = "Error writing report DLL to a temporary file in
XtraReportDllConnector.  Error is: " + ex.Message;
                    return false;
                }
```

Some report types will need to reside on disk to run, but might be stored in the database for security and control. If the report file is stored in the database but needs to run from disk, use **ReportFileLocation** as the name of the resulting file and **ReportFileContents** as the contents to be written into that file to run the report. Some reports can just be run directly from memory, so either load the report from **ReportFileLocation** or use the byte array in **ReportFileContents**, whichever applies. If the report is stored on disk and runs from there, just read **ReportFileLocation** to get the report. This connector handles both situations: the report DLL stored in the database or the report DLL stored on the web server's disk.

This particular report type is a DLL, so it is written to a temporary file and then loaded by file name. At the same time, any additional DLLs that aren't in the GAC that are defined in report_types.xml are also loaded. The highlighted line below shows that we are using **ExtraInfo** as the name of the report class to instantiate.

```
                XtraReport rpt = null;
                MemoryStream mem = new MemoryStream();

                try
                {
                    // Now get the DLL assembly
                    Assembly thisAssembly = Assembly.LoadFile(fileName);
                    foreach (string dll in ExtraDllsToLoad)
                        Assembly.LoadFrom(dll);

                    rpt = (XtraReport)thisAssembly.CreateInstance(ExtraInfo);
                }
```

```
                catch (Exception ex)
                {
                    errorMessages = "Error loading report DLL or extra DLLs in
XtraReportDllConnector.  Error is: " + ex.Message;
                    return false;
                }

                if (rpt == null)
                {
                    errorMessages = "Error loading the class/method for this report.  Was the
report configured correctly for the class/method needed?";
                    return false;
                }
```

In the next section, we search for the report parameters by name and assign values to them from the **Parameters** list that was described above.

```
                // If parameters defined for report, then see if we can pass them
                if (Parameters.Count > 0)
                {
                    int parmsPassed = 0;
                    for (int i = 0; i < rpt.Parameters.Count; i++)
                    {
                        for (int j = 0; j < Parameters.Count; j++)
                        {
                            if
(rpt.Parameters[i].Name.Equals(Parameters[j].ParameterNameInReport,
StringComparison.CurrentCultureIgnoreCase))
                            {
                                rpt.Parameters[i].Value = Parameters[j].Value;
                                parmsPassed++;
                                break;
                            }
                        }
                    }
                    rpt.RequestParameters = false;
                }
```

In the next section, we take the **SaveFormat** value to determine the way to run/export the report. For the standard XtraReportsConnector included with URS, **SaveFormat** is the resulting file extension and we use that to determine how to run the report. XtraReports allows us to save all resulting output to a MemoryStream, so we use that for this connector; other report designers may not have this feature, so you will instead save the output to disk and then read it back into memory. The end result of this step must be an array containing the report output that is then passed back to the calling routine for storing in URS.

```
                if (SaveFormat.Equals("xls", StringComparison.CurrentCultureIgnoreCase))
                {
                    // Set XLS-specific export options.
                    XlsExportOptions xlsOptions = rpt.ExportOptions.Xls;
                    xlsOptions.ShowGridLines = true;
                    xlsOptions.UseNativeFormat = true;

                    // Export the report to XLS.
                    rpt.ExportToXls(mem);
                }
                else if (SaveFormat.Equals("pdf", StringComparison.CurrentCultureIgnoreCase))
                {
                    // Set PDF-specific export options.
                    PdfExportOptions pdfOptions = rpt.ExportOptions.Pdf;
                    pdfOptions.Compressed = true;
                    pdfOptions.ImageQuality = PdfJpegImageQuality.High;
                    pdfOptions.DocumentOptions.Title = ReportName;
                    pdfOptions.DocumentOptions.Subject = ReportDescription;
```

```
                    pdfOptions.ShowPrintDialogOnOpen = false;

                    // Export the report to PDF.
                    rpt.ExportToPdf(mem);
                }
                else if (SaveFormat.Equals("html", StringComparison.CurrentCultureIgnoreCase))
                {
                    // Set HTML-specific export options.
                    HtmlExportOptions htmlOptions = rpt.ExportOptions.Html;
                    htmlOptions.ExportMode = HtmlExportMode.SingleFile;
                    htmlOptions.Title = ReportName;

                    rpt.ExportToHtml(mem);
                }
                else if (SaveFormat.Equals("txt", StringComparison.CurrentCultureIgnoreCase))
                {
                    // Set TXT-specific export options.
                    TextExportOptions txtOptions = rpt.ExportOptions.Text;
                    txtOptions.Separator = ",";
                    txtOptions.QuoteStringsWithSeparators = true;

                    rpt.ExportToText(mem);
                }
                else if (SaveFormat.Equals("rtf", StringComparison.CurrentCultureIgnoreCase))
                {
                    // Set RTF-specific export options.
                    RtfExportOptions rtfOptions = rpt.ExportOptions.Rtf;
                    rtfOptions.ExportMode = RtfExportMode.SingleFile;

                    rpt.ExportToRtf(mem);
                }
                else if (SaveFormat.Equals("tiff", StringComparison.CurrentCultureIgnoreCase))
                {
                    // Set Image-specific export options.
                    ImageExportOptions imgOptions = rpt.ExportOptions.Image;
                    imgOptions.ExportMode = ImageExportMode.SingleFilePageByPage;
                    imgOptions.Format = System.Drawing.Imaging.ImageFormat.Tiff;
                    imgOptions.Resolution = 150;

                    rpt.ExportToImage(mem);
                }
                else if (SaveFormat.Equals("prnx", StringComparison.CurrentCultureIgnoreCase))
                {
                    rpt.CreateDocument(false);
                    rpt.PrintingSystem.SaveDocument(mem);
                }

                results = mem.ToArray();
            }
            catch (Exception ex)
            {
                errorMessages = "Exception detected running report in XtraReportDllConnector: " +
ex.Message;
                return false;
            }
```

Make sure you delete any files you created before exiting. The routine returns *true* if the report ran successfully and the *results* contains a byte array with the report's output. Otherwise, the routine returns *false* and sets *errorMessages* to contain the error messages to place into the URS log for assisting the report administrator with debugging what went wrong.

```
            finally
            {
                if (fileName.Length > 0)
                {
                    try
                    {
                        // Delete the temp file containing the DLL
```

```
                    File.Delete(fileName);
                }
                catch { };
            }
        }

        return true;
    }
```

The next routine that you can override gets the list of parameters from the report. Some report designers will support this, others will not. If not, you don't need to override the routine, because the base class handles this situation properly.

The resulting *List* of type *ReportParameter* contains the information URS needs to schedule a report. *ReportParameter* is a class with the following properties:

- **Description** - the prompt string used to request the value from the report scheduler. Some report types have a parameter description separate from the parameter name. Others do not.
- **ParameterNameInReport** - the string containing the parameter's name as specified in the report.
- **Type** - an enumeration which can be one of *Boolean*, *Integer*, *Float*, *String*, *Choices*, *Date_And_Time* and specifies the type of parameter to define in URS (which might be a different type than the report's parameter needs, so you'll want to be careful of that). You should derive this value from the type of parameter that is defined in the report (e.g., if the parameter in the report is a string, use *String* as the value for **Type**).
- **RangeLow** - the minimum value that can be assigned by the scheduler to this parameter. This property only applies to *Integer* and *Float* parameter types.
- **RangeHigh** - the maximum value that can be assigned by the scheduler to this parameter. This property only applies to *Integer* and *Float* parameter types.
- **Choices** - some report types support a list of choice strings for a parameter and this array would be used to store those choice values.

```
    public override List<ReportParameter> GetReportParameters(out string errorMessage)
    {
        errorMessage = string.Empty;
        List<ReportParameter> parms = new List<ReportParameter>();

        string fileName = string.Empty;
        string destinationFilename = string.Empty;
        XtraReport rpt = null;
```

Same setup as above: we need to get to the report and load all the necessary DLLs to continue. See the notes above for specifics.

```
        try
        {
            try
            {
                if (ReportFileContents != null)
                {
                    fileName = Path.GetTempFileName();
                    File.Move(fileName, Path.ChangeExtension(fileName,
Path.GetExtension(ReportFileLocation)));
```

```
                        File.WriteAllBytes(fileName, ReportFileContents);
                    }
                    else
                    {
                        fileName = ReportFileLocation;
                    }
                }
                catch (Exception ex)
                {
                    errorMessage = "Error writing report DLL to a temporary file in
XtraReportDllConnector.  Error is: " + ex.Message;
                    return null;
                }

                MemoryStream mem = new MemoryStream();

                try
                {
                    // Now get the DLL assembly
                    Assembly thisAssembly = Assembly.LoadFile(fileName);
                    foreach (string dll in ExtraDllsToLoad)
                        Assembly.LoadFrom(dll);

                    rpt = (XtraReport)thisAssembly.CreateInstance(ExtraInfo);
                }
                catch (Exception ex)
                {
                    errorMessage = "Error loading report DLL or extra DLLs in
XtraReportDllConnector.  Error is: " + ex.Message;
                    return null;
                }

                if (rpt == null)
                {
                    errorMessage = "Error loading the class/method for this report.  Was the
report configured correctly for the class/method needed?";
                    return null;
                }
```

For each parameter found in the report, create a *ReportParameter* object, fill in the appropriate properties, and then add it to the *List*.

```
                foreach (Parameter field in rpt.Parameters)
                {
                    ReportParameter parm = new ReportParameter();
                    parm.ParameterNameInReport = field.Name;
                    if (field.Description == null || field.Description.Length == 0)
                        parm.Description = field.Name;
                    else
                        parm.Description = field.Description;
                    switch (field.ParameterType)
                    {
                        case ParameterType.Boolean:
                            parm.Type = ReportParameterType.Boolean;
                            break;

                        case ParameterType.DateTime:
                            parm.Type = ReportParameterType.Date_And_Time;
                            break;

                        case ParameterType.String:
                            parm.Type = ReportParameterType.String;
                            break;

                        case ParameterType.Decimal:
                        case ParameterType.Double:
                        case ParameterType.Float:
                            parm.Type = ReportParameterType.Float;
                            parm.RangeHigh = float.MaxValue;
```

```
                            parm.RangeLow = float.MinValue;
                            break;

                        case ParameterType.Int32:
                            parm.Type = ReportParameterType.Integer;
                            parm.RangeHigh = float.MaxValue;
                            parm.RangeLow = float.MinValue;
                            break;
                    }
                    parms.Add(parm);
                }
            }
            catch (Exception ex)
            {
                errorMessage = "Error interpreting parameters in XtraReportDllConnector.  Error
is: " + ex.Message;
                return null;
            }
```

Finally, dispose the report or delete any report files and return the parameters.

```
            finally
            {
                if (rpt != null)
                    rpt.Dispose();

                if (fileName.Length > 0)
                {
                    try
                    {
                        // Delete the temp file containing the DLL
                        File.Delete(fileName);
                    }
                    catch { };
                }
            }

            return parms;
        }

    }
}
```

Compile the connector into a DLL as described in the URS Installation Guide and place it on your URS server in the *bin* directory for the URS web site as well as the *ReportsRunner* directory in the URS installation.

How does Report_Types.xml Work With This?

Once you have a report connector, you'll need to tell the report_types.xml file about it. The structure of this file is described in the *URS Installation Guide*, but we'll talk about it in more depth here.

This is a typical report_types.xml created for the DevExpress XtraReports Report Connector described above:

```
<?xml version="1.0" encoding="utf-8" ?>
<report_types>
  <report_type id="XtraReportsDll" name="DevExpress XtraReports (DLL file)">
    <connector
assembly="C:\VersaReports\Connectors\XtraReportsConnector\bin\Debug\XtraReportsConnector.dll"
class="XtraReportsConnector.XtraReportsDllConnector"/>
    <report_file extensions=".dll" storage="disk" extra_info_prompt="Class Name of Report"
pass_connection="false" />
    <supporting_assemblies>
```

```
      <add assembly="C:\Program Files\Developer Express .NET
v8.3\Sources\DevExpress.DLL\DevExpress.Charts.v8.3.Core.dll" />
      <add assembly="C:\Program Files\Developer Express .NET
v8.3\Sources\DevExpress.DLL\DevExpress.Data.v8.3.dll" />
      <add assembly="C:\Program Files\Developer Express .NET
v8.3\Sources\DevExpress.DLL\DevExpress.Utils.v8.3.dll" />
      <add assembly="C:\Program Files\Developer Express .NET
v8.3\Sources\DevExpress.DLL\DevExpress.Web.v8.3.dll" />
      <add assembly="C:\Program Files\Developer Express .NET
v8.3\Sources\DevExpress.DLL\DevExpress.XtraCharts.v8.3.dll" />
      <add assembly="C:\Program Files\Developer Express .NET
v8.3\Sources\DevExpress.DLL\DevExpress.XtraPrinting.v8.3.dll" />
      <add assembly="C:\Program Files\Developer Express .NET
v8.3\Sources\DevExpress.DLL\DevExpress.XtraReports.v8.3.dll" />
      <add assembly="C:\Program Files\Developer Express .NET
v8.3\Sources\DevExpress.DLL\DevExpress.XtraReports.v8.3.Web.dll" />
    </supporting_assemblies>
    <output_formats>
      <format extension="pdf" name="Acrobat Reader" mime_type="application/pdf" native="false"
icon="images/pdf.png" />
      <format extension="html" name="HTML Web Page" mime_type="text/html" native="false"
icon="images/html.png"/>
      <format extension="rtf" name="Rich Text File" mime_type="application/rtf" native="false"
icon="images/rtf.png"/>
      <format extension="xls" name="Excel 2003" mime_type="application/vnd.ms-excel"
native="false" icon="images/xls.png" />
      <format extension="prnx" name="Preview-Only Format" native="true"
display_page="XtraReportsViewer.aspx?id={0}" icon="images/report.png" />
    </output_formats>
  </report_type>
</report_types>
```

Let's tie some pieces together now. In this line, the **name** attribute is what appears in the Report Definition Wizard when a report administrator selects this type of report.

```
<report_type id="XtraReportsDll" name="DevExpress XtraReports (DLL file)">
```

This line contains the report connector's compiled assembly location and the class defined in it:

```
    <connector
assembly="C:\VersaReports\Connectors\XtraReportsConnector\bin\Debug\XtraReportsConnector.dll"
class="XtraReportsConnector.XtraReportsDllConnector"/>
```

The following line defines some additional information that makes it possible for URS to properly call and run a report of this type:

```
<report_file extensions=".dll" storage="disk" extra_info_prompt="Class Name of Report"
pass_connection="false" />
```

In the above line, the extensions are the list of extensions that are acceptable for a report of this type. XtraReports also supports a report format that can be sent without coding (RPX format), but we didn't use that for this connector, so the extensions are only ".dll". Because we're using DLL-based reports for this connector, we need to know the fully-qualified class name of the report in the DLL that is going to be provided to the Report Definition Wizard.

The `<supporting_assemblies>` block defines the array of strings that end up in the **ExtraDllsToLoad** property that's inherited by the Report Connector class you create.

The `<output_formats>` block is critical and defines everything you'll need to know when you generate the report in your connector routines. Let's look at two of the lines and see why this is so critical:

```
      <format extension="xls" name="Excel 2003" mime_type="application/vnd.ms-excel"
native="false" icon="images/xls.png" />
      <format extension="prnx" name="Preview-Only Format" native="true"
display_page="XtraReportsViewer.aspx?id={0}" icon="images/report.png" />
```

In these lines, the extension attribute is a value that could be placed into the **SaveFormat** property that the class inherits. For exported formats that don't require a special viewer page (e.g., Excel, PDF), you will set the native property to "false". For formats that have a "native" viewer (e.g., XtraReports includes a viewer control that you can embed in a web page), the native property will be set to "true" and the report will be saved in a format that's "native" to this viewer.

How Should I Debug My Report Connector?

Once you have the report connector code compiled, we recommend that you test the code with ReportsRunner operating in interactive mode. If ReportsRunner has been started as a service, temporarily stop the service and run ReportsRunner from a Command Prompt window. Then use Visual Studio to attach to the ReportsRunner program and debug your connector as needed.