Getting started with the XCRICAP-NET base library

Please note that this library is not maintained by, endorsed by, or in any other way officially connected with, the xcri.org website or the people and organisations involved with XCRI in the UK.

Aims

The aim of this project is to give providers who may already have base data access code written in .NET an easy method to output a valid XCRI feed with minimal effort. The codebase allows provides the ability to create XCRI-CAP documents using the XCRI-CAP 1.1 namespace plus additional namespaces such as the XCRI-CAP 1.1 Terms (built-in) or others (through extension).

The library aims to support 100% of the XCRI-CAP 1.1 standard including the delta-update-pattern.

Project structure

The project is contained within one single DLL named "xcri.dll"; the source of this is available via the Google Code website. The project itself aims to allow the output of the base XCRI-CAP specification with minimal work, whilst also allowing for the output of customised XML extensions where applicable.

Creating a new XCRI feed

Please note that this documentation deals with using the DLL from within Visual Studio and concentrates upon using C# as the programming language.

Setting up a project within Visual Studio

This will concentrate upon creating a sample Class Library and ASP.NET Web Application to use it from.

1. Create a new ASP.NET Web Application project within Visual Studio. Locate this wherever makes sense and choose an applicable name for the solution and project such as "XCRISampleWebApplication".
2. Right-click on the solution node and select "Add | New project" to add a new Class Library project. Locate this in a similar vein to the project above and choose an applicable name such as "MyCollegeXCRIAdapter".
3. Add a reference within the Class Library project to the release XCRI.dll file which can be downloaded from http://xcricap-net.googlecode.com. You may also need to add a reference to your base data access layer, if you already have one.
4. Add a reference within the Web Application to both the XCRI.dll file and your Class Library project.

At this point we're ready to start creating a class hierarchy that extends the one provided and allows you to quickly produce an XCRI feed. For the sake of this document we're not going to investigate integrating with any existing data access layer, we're going to just create sample objects.

1. In the Class Library, remove the "Class1.cs" file which is automatically generated.
2. Add a class file and name it "MyCollegeProvider". This represents a logical provider of courses – for example, this may be a single college or university. If your college or university also has an associated entity that offers courses to businesses, as an example, we could create a second provider for their data.
3. Open up the provider class within Visual Studio and modify the class to extend the XCRI.BaseProvider class. Add in a new default constructor and set the minimum required set of values to defaults. You may have another methodology for this, or you may want to populate these values programmatically at runtime – in which case, use your own judgement.

```
public MyCollegeProvider()
            : base()
        {

            this.Url = new Uri("http://www.craighawker.co.uk");
            this.Titles.Add(new XCRI.Title() { Value = "Craig Hawker College
(test)" });
            this.ReferenceNumber = 12345678;
            this.Address = new XCRI.Address()
            {
                Street = "123 Fake Street",
                Town = "My town",
                Postcode = "AB12 1AB"
            };
            this.Descriptions.Add(new XCRI.Description()
            {
                Value = "This is an example provider to show what must be
implemented for a valid XCRI feed."
            });

        }
```

4. For the purpose of this document we're going to fake loading the Course information from a database and, instead, manually populate the course and presentation information. In a real scenario, this would interact with your current course datasource.

```
// This is generic information about the course
            XCRI.Course course1 = new XCRI.Course()
            {
                Uri = new Uri("http://www.craighawker.co.uk/courses/hnc/business-
computing/")
            };
            course1.Identifiers.Add(new XCRI.Identifier()
            {
                Value = course1.Uri.ToString()
            });
            course1.Titles.Add(new XCRI.Title()
            {
                Value = "HNC Business Computing"
            });
            course1.Descriptions.Add(new
XCRI.Vocabularies.XCRICAP11.Terms.Description()
            {
                XsiTypeValue =
XCRI.Vocabularies.XCRICAP11.Terms.DescriptionTypes.prerequisites,
                Value = "5 GCSE grades A-C, preferably including a computing
element"
            });
            // This is a means in which a student can take the course.
            course1.Presentations.Add(new XCRI.Presentation()
            {
                AttendanceMode = new
XCRI.Vocabularies.XCRICAP11.Terms.AttendanceMode()
                {
                    Value =
XCRI.Vocabularies.XCRICAP11.Terms.AttendanceModeTypes.Campus
                },
                AttendancePattern = new
XCRI.Vocabularies.XCRICAP11.Terms.AttendancePattern()
                {
                    Value =
XCRI.Vocabularies.XCRICAP11.Terms.AttendancePatternTypes.Daytime
                },
                Start = new DateTime(2011, 10, 01)
            });
```

5. At this point your object model is complete and returning (sample) data. Next it's on to modifying the web application to return a valid XCRI feed.

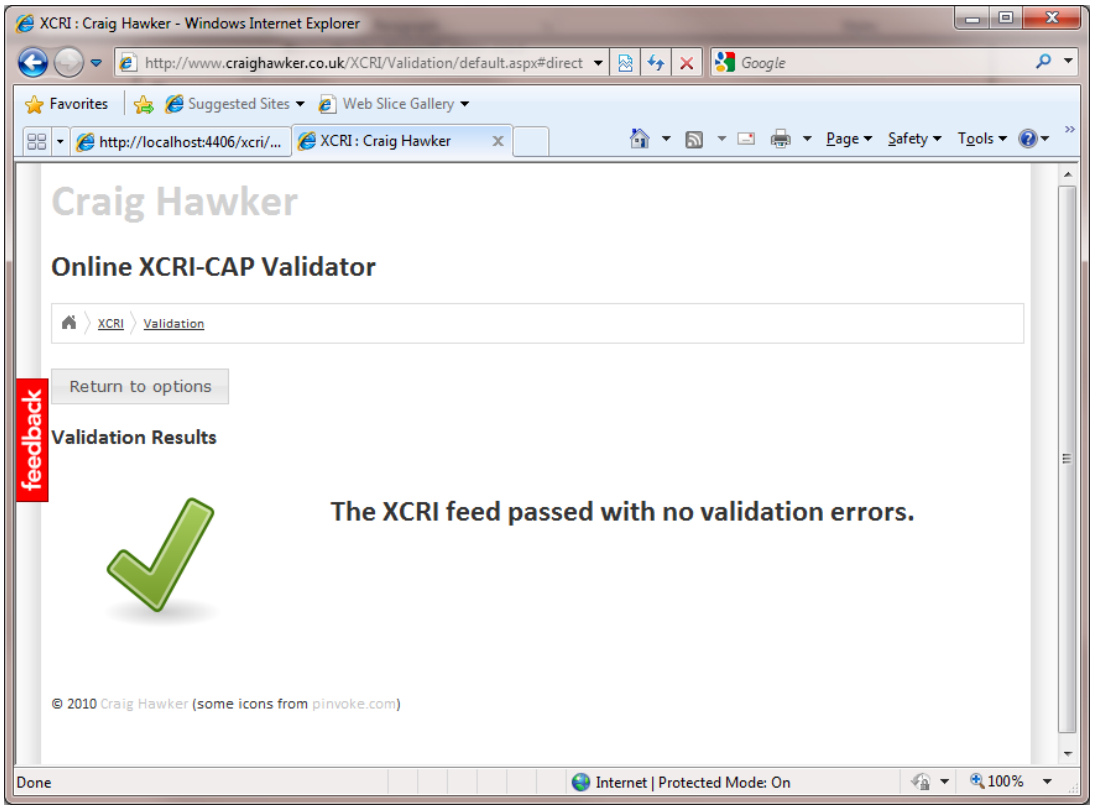
The next step is to modify the sample web application to output your XCRI feed when someone requests it. Obviously this stage is artificial and you would incorporate this step into your public-facing website. Please note that, due to the potential size of the XCRI feed, you may also want to investigate caching this information.

1. Create a new folder within the application named "xcri" and create a Default.aspx page within it (read more about the reliable feed location pattern).

2. Ensure that the content type for the page is set to “text/xml”.
3. Override the Render method and output the XML to the response stream.

```
protected override void Render(HtmlTextWriter writer)
        {
            // Create an XML generator for XCRI-CAP 1.1
            XCRI.XmlGeneration.Interfaces.IXmlGenerator
                gen =
XCRI.XmlGeneration.XmlGeneratorFactory.GetXmlGenerator(XCRI.XCRIProfiles.XC
RI_v1_1);
            // This will be the root element (valid in most scenarios)
            XCRI.Catalog catalog = new XCRI.Catalog();
            // Add our provider
            catalog.Providers.Add(new
MyCollegeXCRIAdapter.MyCollegeProvider());
            // Set the root element
            gen.RootElement = catalog;
            // Force generation
            gen.Generate(writer);
        }
```

4. Just to go full-circle, let’s view the source of the page and put it through the online validator at http://www.craighawker.co.uk/xcri/:



Next, a couple of notes on the XCRI location: there’re two recommended XCRI location patterns mentioned within the blog. Both auto-discovery features can be tested using the online XCRI validator:

1. The “reliable feed location” pattern (http://www.xcri.org/wiki/index.php/Reliable_feed_location)
   The XCRI feed is always in a location which is expected – this is defined as http://www.myeducationalinstitution.ac.uk/xcri/. Please note that Uri locations are case-sensitive (even though IIS/Windows implementations aren’t).
2. The “feed autodiscovery” pattern (http://www.xcri.org/wiki/index.php/Feed_autodiscovery)
   The XCRI feed is located through the use of a HTML “meta tag”. To use it, the provider includes a <link /> tag within the <head> section of their HTML source code. This then, in turn, points to the XCRI feed.