Razor Template System Documentation

Version: 1.0

Razor Template System

1 Table of contents

Razor Template System

2 Introduction

Razor Template System is a powerful template system to convert inputted files (or manual data objects) into other formats based on templates. Currently the following input file types are supported:

- CSV
- XML

Other formats are also possible but need to be implemented.

The templates syntax is either C# or VB.NET syntax, they are cshtml or vbhtml files known from ASP.NET MVC. The name Razor Template System comes from the Razor View Engine, a project from Microsoft to create dynamic web sites with ASP.NET (MVC – Model -View-Controller). With the Razor View Engine you can pass a dynamic model of data into a template and use pure C# or VB.NET code (with Razor-Syntax) to control and publish it.
I personally recommend the C#-Razor-Syntax because it's much more clear compared to VB.NET.

3 Use case

Razor Template System is used to easily convert input files such as CSV and XML or a your own data model to output types defined by yourself. It is not made for converting large files (a CSV with many thousands of rows) because the data must stay in memory until it is processed and written to the template. Razor Template System is made for converting XML files to other needed formats or converting each CSV row in a set of XML tags.

There is currently no limitation to the output file types, because you can define the file structure and layout by yourself in the template. The input files are currently limited to CSV and XML.

4 Input reading

This chapter describes how the data can be read from input files, but first the data structure must be explained with what Razor Template System operates. Razor Template System has one model in which the data is hold of an inputted file. That model contains so called container and every container contains fields. A container is identified by a unique name and can hold multiple fields, each field is identified by a unique name and holds the value. A container can be of type array, that means that a fields can occure more than once. If you transfer that to a CSV, the first row (with column names) is a normal container and each field contains the columns name. Now the second row to the end of the file is a container of type array and each field in each row contains the cell value. This is of course also possible with a XML.

4.1 CSV

The row number (or numbers if array) of a CSV is the starting point for a container and the column number in is the indicator for a field.

4.2 XML

When using XML as input type, there is only one information needed to read data out of the XML – the XPath. Each field value is found by a XPath expression and the container type (array or single) indicates whether there is more than one result value on that XPath.

5 Manual object

You may of course enter a manual object into a template. The object can be of any type (an ExpandoObject for a completely dynamic object or an class type defined by yourself), but in the template the object the object you access is named “TemplateData”. So if you inject a list (List<MyClass>) named MyClassList, then in the template it is called “TemplateData” and you would access it like this: *foreach (MyClass myClass in TemplateData)*.

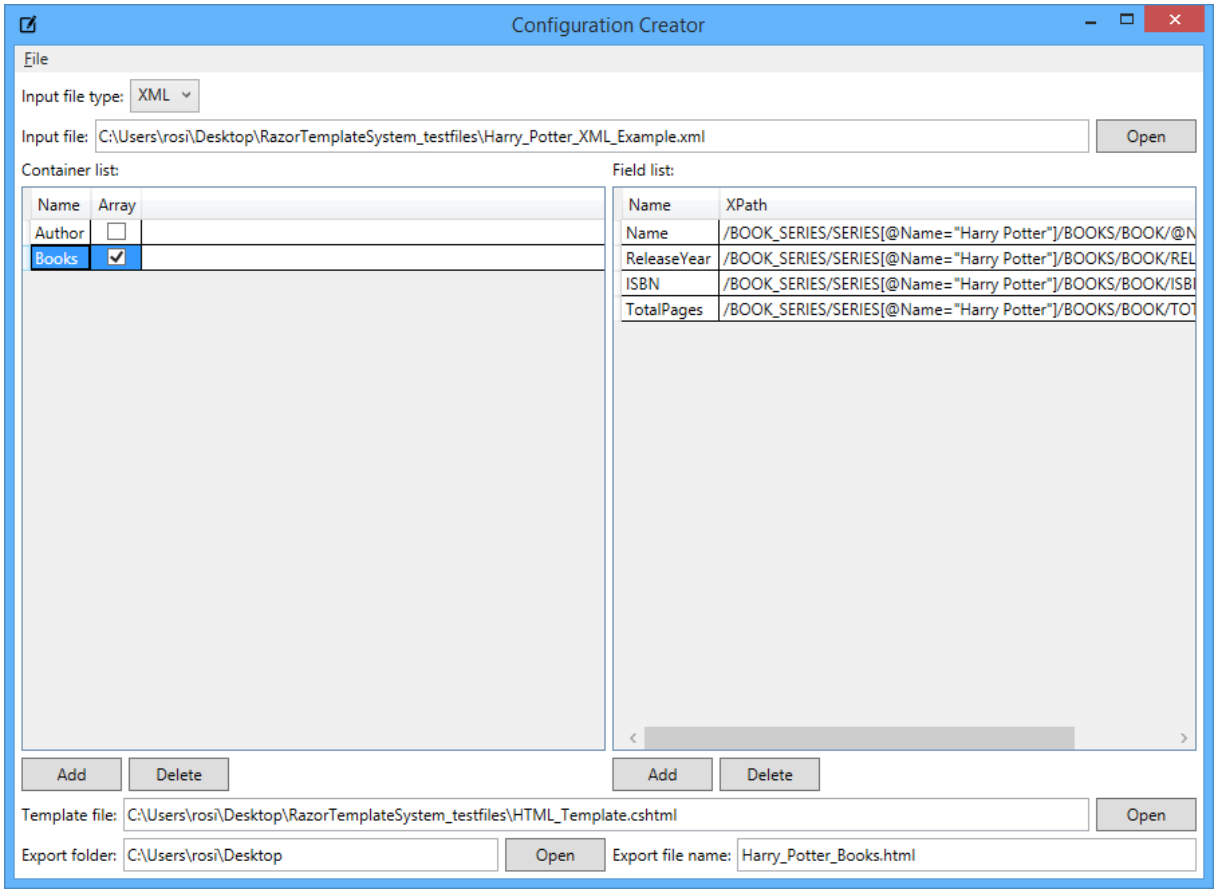
6 Configuration

To make Razor Template System work, you need a Mapping Configuration file. It is a serialized XML file with different properties, when loading it in Razor Template System it deserializes to an object. You may of course create this object in code, set the properties manually and give it to Razor Template System. The Mapping Configuration file contains the following:

- The path to the input file (CSV or XML) (not necessary when using a manual object)
- The input file type (CSV or XML) (not necessary when using a manual object)
- The path to the template used (cshtml or vbhtml)
- The path to the export folder
- The export file name
- A list of containers (not necessary when using a manual object)
- A list of fields for each container (not necessary when using a manual object)

6.1 Configuration Creator

In this project is also a WPF GUI application included for creating the Mapping Configuration files. With this application you can create a complete Mapping Configuration XML file and use it in Razor Template System:



The workflow is as follows:

1. Load an existing mapping configuration (see “File” menu) or start with a new one.
2. Define the input file type (CSV or XML).
3. Select the input file.
4. Add at least one container and choose whether it is of type array or single.
    a. When input file type is CSV you have to define the Line (if it is single container) or the Start and End Line (if it is a array container). The Line, End and Start Line is one-based (you can leave the End Line property 0, that means it’ll read to the end of the file)!
5. Add at least one field. If the field is for a CSV you have to define the column where this field should read from (one-based). If the field is for a XML you have to define an XPath to your value.
6. Select the template file for your project.
7. Define the export folder and your export file name for your result file.

8. Save the mapping configuration (see “File” menu).

7 How to use it

Open up the RazorTemplateSystem solution and enter the "Examples" folder. In it you will find multiple projects to for the following use cases to get you started:

1. Convert a XML file into a HTML file*
2. Convert a CSV file into a TAB-seperated file
3. Convert a manual object into a JSON file

*you probably think: “Are you kidding me? XML to HTML? HTML is kind of XML and you could easily parse it with XSLT”
Yes, you may also parse a XML with XSLT to an HTML, but you don’t have the full blown method set of .NET ☺.

8 How does it work

RazorTemplateSystem converts (with the help of the Razor View Engine) your template into a compileable Code (see below for example). Basically it creates a dynamic source code file from the template and the inputted data. After that it compiles the source file into a binary and executes it. The returned string is then written to a file.

Example code from the HTML test project:

```
namespace RazorEngine
{
    using System;
    using System.Collections.Generic;
    using System.Text.RegularExpressions;


    public class Template_1 : RazorTemplateSystem.RazorEngine.TemplateBase
    {

#line hidden

        public Template_1()
        {
        }

        public override void Execute()
        {
            WriteLiteral("<html>\r\n    <head>\r\n        <title>Harry Potter
Example</title>\r\n        <link");
            WriteLiteral(" rel=\"stylesheet\"");
            WriteLiteral("
href=\"https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/css/bootstrap.min.css\"");
            WriteLiteral(">\r\n    </head>\r\n    <body>\r\n        <div");
            WriteLiteral(" class=\"container\"");
```

Razor Template System

```
            WriteLiteral(">\r\n            <h1>Harry Potter Example</h1>\r\n
<h3>Author</h3>\r\n      " +
            "      <table");
            WriteLiteral(" class=\"table table-striped\"");
            WriteLiteral(">\r\n                <tr>\r\n
<td>Name</td>\r\n                    <" +
            "td>");
            Write(TemplateData.Author.Name);
            WriteLiteral("</td>\r\n                </tr>\r\n                <tr>\r\n
<td>Birth" +
            " date</td>\r\n                    <td>");
            Write(TemplateData.Author.BirthDate);
            WriteLiteral("</td>\r\n                </tr>\r\n                <tr>\r\n
<td>Occup" +
            "ation</td>\r\n                    <td>");
            Write(TemplateData.Author.Occupation);
            WriteLiteral("</td>\r\n                </tr>\r\n                <tr>\r\n
<td>Natio" +
            "nality</td>\r\n                     <td>");
            Write(TemplateData.Author.Nationality);
            WriteLiteral("</td>\r\n                </tr>\r\n            </table>\r\n
<h3>Books</h3>\r\n " +
            "           <table");
            WriteLiteral(" class=\"table\"");
            WriteLiteral(@">
                <thead>
                    <tr>
                        <th>Name</th>
                        <th>Release Year</th>
                        <th>ISBN</th>
                        <th>Total pages</th>
                    </tr>
                </thead>
                <tbody>
");
            foreach (var book in TemplateData.Books)
            {
                WriteLiteral("                    <tr>\r\n
<td>");
                Write(book.Name);
                WriteLiteral("</td>\r\n                        <td>");
                Write(book.ReleaseYear);
                WriteLiteral("</td>\r\n                        <td>");
                Write(book.ISBN);
                WriteLiteral("</td>\r\n                        <td>");
                Write(book.TotalPages);
                WriteLiteral("</td>\r\n                    </tr>\r\n");
            }

            WriteLiteral("                </tbody>\r\n            </table>\r\n
</div>\r\n    </body>\r\n</ht" +
            "ml>");

        }
    }
}
```

You may wonder if it is fast? Well it performs of course a lot worse than a full blown statically XML to HTML converter. The main aim of RazorTemplateSystem is to provide an easy way to create different kind of templates quick for different kind of tasks. If you have changing input and export types, then RazorTemplateSystem is suited for you (e.g. for EDI message parsing).
If your input and export types are kind of static, then you should write your own parser which will be faster.