# 

# Using the RazorTransform Tool

(aka Artie)

Jim Wallace

May 31, 2014

# Contents

[Using the RazorTransform Tool 1](#_Toc352220944)

[Contents 2](#_Toc352220945)

[1 Quick Start 3](#_Toc352220946)

[2 What is Artie? 4](#_Toc352220947)

[3 Technical Overview 5](#_Toc352220948)

[4 Creating a Model with Artie 6](#_Toc352220949)

[5 The Model Definition File -- RtObject.xml 7](#_Toc352220950)

[5.1 <group> Element 7](#_Toc352220951)

[5.2 <item> Element 8](#_Toc352220952)

[5.2.1 Types 9](#_Toc352220953)

[5.3 <enum> Element 10](#_Toc352220954)

[5.4 Arrays 11](#_Toc352220955)

[6 The Model Object 12](#_Toc352220956)

[7 Razor Templates 14](#_Toc352220957)

[7.1 SimpleXml.xml Sample 14](#_Toc352220958)

[7.2 ItemBXml.xml Sample 14](#_Toc352220959)

[7.3 NestedTest.txt Sample 15](#_Toc352220960)

[7.4 ItemBTransforms.txt 15](#_Toc352220961)

[8 RTValues.xml 16](#_Toc352220962)

[9 Command Line Switches 17](#_Toc352220963)

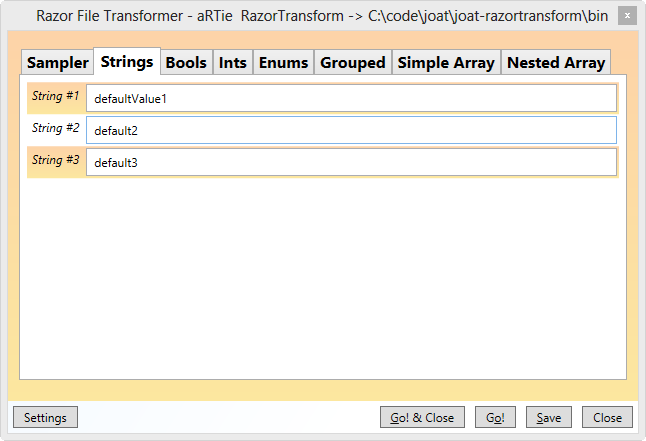
[9.1 PowerShell 17](#_Toc352220964)

[10 Artie Customization 18](#_Toc352220965)

[Appendix A. Document Revisions 19](#_Toc352220966)

# Quick Start

1. Install Artie and its sample data
2. Run RazorTransform.exe from its install folder.



1. Click around and enter values in Artie.
2. Click Go! to run the Razor Transforms.
3. Look with wonder at the Output folder that was created with the files in the Templates folder.

# What is Artie?

Artie is long for RT which is short for RazorTransform. Artie takes template files and generates output from them using the Razor view model introduced in ASP.NET 4.0. Output files can be ASP.NET web pages, XML, or any other text file, such as a PowerShell script.

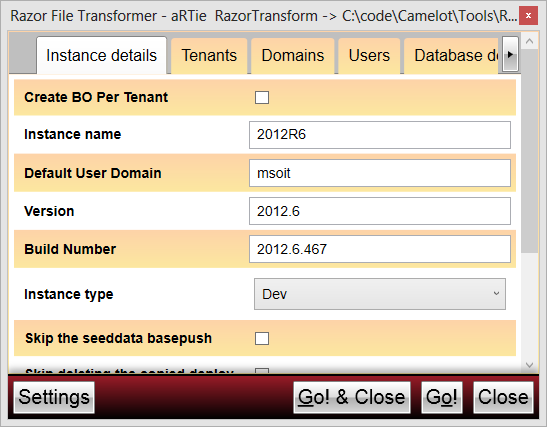


Figure 1 -- Typical Screen Shot of Data Entry

The need for Artie arose after painfully watching the IT department do a deployment of our product that required editing a myriad of XML and PowerShell files. Many of the files were very similar, and much of the data between them was redundant creating a very tedious and error-prone process just to create the files needed to begin the deploy.

Artie solves the problem by having user interface for easily entering in all the values once, in one place, then using templates with embedded C# code to generate the output files. The values are stored so they can be easily edited later for another deploy, or copied to use as the basis of another deploy.

# Technical Overview

Artie uses the [Razor](http://weblogs.asp.net/scottgu/archive/2010/07/02/introducing-razor.aspx) syntax used in .NET 4 ASP.NET pages to transform a set of template files into output files. It is a completely generic tool driven by one XML file that define what the GUI looks like, and another to store values. [Razor](http://www.microsoft.com/web/category/razor) allows Code and markup, or any other text for that matter, to produce an output file. Anything that can be done in C# can be done in a Razor transform file.

The Model can be as simple as a handful of properties, or arrays of object, or nested arrays of objects. Artie knows about many data types such as string, number, etc., and also allows extensions to add new types.

A GUI allows a user to easily enter various values for all of the properties of the Model. After entering in values for the Model, the user starts the Razor transforms. Artie transforms all the Razor templates in a *templates* folder. Each file is transformed using the values the user entered in the UI. Normally there is a one-to-one relationship between Razor template and output file with the output file having the same name as the template file. In an advanced sample below, a template creates multiple files.

There are a few buttons along the bottom of the Artie UI. The Go button runs the transforms. The Go & Close buttons run the transforms, and if successful, closes Artie. Save will save the current values without doing the transform. Close will close Artie without saving any changes. The Settings button allows changing a few of the settings, all of which can be set via command line parameters (see below). When the transform is running all the buttons are disabled, except a Cancel button.

# The Model Definition File -- RtObject.xml

This file defines the object that is used for the transforms and how the model is shown in the UI. By default this file is read from the current folder. It is made up of groups, which are made up of other groups and items. The following table lists the elements and where they can appear in the file, and the following sections describe each element in detail. Sample RtObject.xml files are includes with the source.

|  |  |  |
| --- | --- | --- |
| Element Name | Can Contain Elements | Contained in |
| RtObject (root element) | group, enum, custom | n/a |
| group | group, item | RtObject, group |
| item | n/a | group |
| enum | value | RtObject |
| value | n/a | enum |
| custom | n/a | RtObject |
| regex | n/a | RtObject |

Table 1—RtObject.xml Elements

## <group> Element

This element is shown in the UI as a tab. The attributes for the *group* element are shown in the table below. If the *group* has an *arrayValueName* attribute, it is an array, otherwise, it simply groups simple types in a tab. See below for array details.

|  |  |  |
| --- | --- | --- |
| Name | Values | Description |
| name | Any string | Shown in the UI |
| description (optional) | Any string | This is used as the tooltip in the UI |
| min (optional) | Numbers | Minimum number of items required if the group is an array. |
| expanded (optional) | n/a | No longer used, but may appear in older files. |
| hidden (optional) | true/false | If true, does not show the tab or its contents in the UI. |
| arrayValueName (optional) | Case-sensitive member name on the C# Model object | Must be a valid C# identifier. This will be used in the Razor to access the array, e.g. @Model.itemB |
| key (optional) | Any string with names of items embedded in it. | For arrays, this is the value shown in the list of items. If not supplied, the key is the first item in the group. (See below for details.) |
| sort (optional) | true, false | If an array, sort the key values when the list is displayed. |
| unique (optional) | true, false | If an array, enforce the key name is unique when validating |

Table 2 – <group> Attributes

The XML below shows *group* elements and the image after it shows the tabs they create in the UI.

<group name="Strings" description="A few strings" hidden="false">

</group>

<group name="Bools" description="A few booleans">

</group>

<group name="Ints" description="Some integers">

</group>

<group name="Enums" description="Enumerations are fun!">

</group>

<group name="Grouped" description="Array of itemB objects with min number of 3."

</group>

<group name="Simple Array"

description="Array of itemAS objects with a group. Min 2" arrayValueName="itemAS" min="2">

</group>

<group name="Nested Array"

description="Array of itemA objects nested

with ItemBs with min number of 0."

arrayValueName="itemA" min="0">

</group>

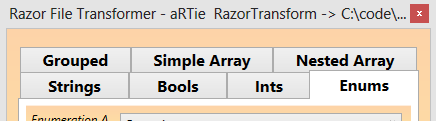


Figure 3 -- <group> elments as Tabs in UI

## <item> Element

All the *item* elements describe simple types associated with the Model, or an array item, and must be under a *group* element. *items* have the following attributes:

|  |  |  |
| --- | --- | --- |
| Name | Values | Description |
| name | Case-sensitive member name on the C# Model object | Must be a valid C# identifier. This will be used in the Razor to access members, e.g. @Model.enumD1 |
| type | folder, uncpath, guid, webport, serverport, bool, int32, password, string, <enumname>, or <customname> | This is used to create the member on the model. See the table below for details of each type. |
| displayName | Any string | This is the name shown in the UI to prompt the user |
| description (optional) | Any string | This is used as the tooltip in the UI |
| min/max (optional) | Numbers | For int32 limits on values. For string, limits on length. |
| defaultValue (optional) | Depends on type | A default value for the item if there’s no previously saved value in RtValues.xml |
| hidden (optional) | true/false | If true, does not show the tab or its contents in the UI. |
| regex (optional) | <name of regex> | This must match the name of a regex element in the XML. If supplied the value entered must match the regular expression. See below for details. |

Table 3 – Item Attributes

### Types

There are several builtin types that Artie already knows about as shown in the following table. These values may be used at the *type* attribute for an item.

|  |  |
| --- | --- |
| Name | Description |
| folder/uncpath | A string value that has a button for browsing for a folder |
| guid | A Guid type with a button to generate a new Guid value |
| bool | Boolean value shown as a checkbox in the UI |
| int32 | Integer value |
| password | String that uses password input control. Note password available for the transform, but never saved in RtValues.xml |
| string | String value |
| hyperlink | Displayed as a read-only hyperlink. The description used as the text shown for the hyper link. If empty, the hyperlink is shown. |

Table 4 – Builtin Item Type values

The following XML shows *item* elements in a *group* followed by the image of what the group looks like in the UI.

<group name="Sampler" description="One of each" hidden="false">

<item name="stringS1" type="String" description="String with a size range of 10-20" defaultValue="defaultValue" displayName="String #1" min="10" max="20"/>

<item name="boolS1" displayName="Boolean #1" type="Bool" description="This is a boolean value defaulting to False." defaultValue="False" />

<item name="intS1" type="Int32" description="Int32 with a range of 10-20" defaultValue="1" displayName="Int32 #1" min="10" max="20"/>

<item name="enumS2" type="EnumB" description="EnumB description" defaultValue="enumKeyZ" displayName="Enumeration B" />

<item name="uncS1" type="UncPath" description="Unc Description" defaultValue="c:\" displayName="UNC Folder name" />

<item name="folderS1" type="Folder" description="Folder Description" defaultValue="c:\" displayName="Folder name" />

<item name="passwordS1" type="Password" description="enter a password" defaultValue="" displayName="Password" />

<item name="guidS1" type="Guid" description="enter a Guid" defaultValue="" displayName="Guid" />

</group>

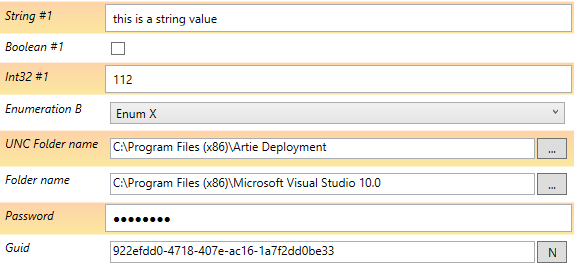


Figure 4 -- Sampler of Types

In addition to the builtin types, enums (see below) may be used to create drop down lists of values in Artie. Finally, Artie can be extended by writing a custom class to create items in the UI. See below for details.

## <enum> Element

This allows you to define values for drop down list boxes in the UI. The enum element has a *name* attribute that is used as the *type* attribute of an item element. Then it has two or more value elements defining the enumeration. Each of these has a *value* attribute that is shown in the combo box and can by any string. The *key* attribute is will be what is set on the Model’s member.

|  |  |  |
| --- | --- | --- |
| Name | Values | Description |
| value | Any string | Name shown in the dropdown list in the UI |
| key | C# identifier | Must be a valid C# identifier. This will be used in the Razor to set the value on an item. |

Table 5 – Enum Value Attributes

<group name="Enums" description="Enumerations are fun!">

<item name="enumD3" type="EnumC"

description="EnumC description"

defaultValue=""

displayName="Enumeration C" /> />

</group>

<enum name="EnumC">

<value key="enumKeyQ" value="Enum Q" />

<value key="enumKeyP" value="Enum P" />

</enum>

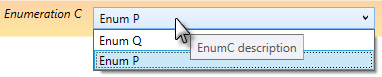


Figure 5 -- Enumeration

## Arrays

Arrays are used when you need to nest another object under the Model object. To create an array of objects, use the *group* element and set the *arrayValueName* attribute to a valid C# identifier name. This will be the name of the array member on the model. In the RtObject.xml file shown below itemA is an array and would be referenced as Model.itemA in a foreach loop in the Razor. See the examples for looping over arrays.

The *group* below shows a group that is an array. Notice that the key is the nameA value for the array item prefixed with “>”. You may include multiple values in the key and other text to create a user-friendly key name for each item in the array.

<group name="Nested Array"

description="Array of itemA objects nested with

ItemBs with min number of 0."

arrayValueName="itemA" min="0" key=">nameA">

<group name="A">

<item displayName="Name for A" name="nameA" type="string"

description="Name, no spaces" />

…

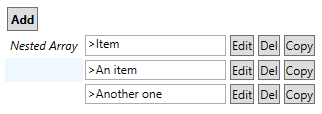


Figure 6 -- Array of three Nested Array items

# The Model Object

The previous section showed all the various elements and how they appeared in the UI. Those elements also describe a .NET object that will be the Model that is used in the Razor transformations described in the next section. All the items in the RtObject file become properties of the Model object. For arrays, the array is a property of the Model, and the objects in the array are described by the items under it. The following is a simple RtObject file that has several properties and an array.

<RtObject>

<group name="Sampler" >

<item name="stringS1" type="String" displayName="String #1" />

<item name="boolS1" displayName="Boolean #1" type="Bool" />

<item name="intS1" type="Int32" displayName="Int32 #1" min="10" max="20"/>

</group>

<group name="Simple Array" min="2" arrayValueName="itemAS" >

<group name="Basic" >

<item displayName="Name for A" name="nameA" type="string" />

<item displayName="Description for A" name="descriptionA" type="string" />

</group>

<group name="Other" description="this is other stuff">

<item displayName="String" name="stringInA" type="string" />

<item displayName="Int" name="intInA" type="Int32" />

</group>

</group>

</RtObject>

This creates a UI that has two groups (tabs) as follows:

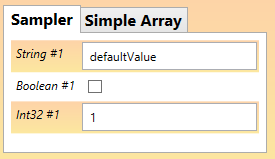
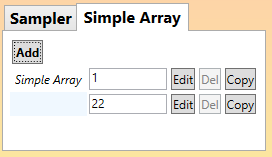
 

Figure 7 -- Sample Groups

And the Simple Array item has two groups (tabs) and looks like this when clicking Edit or Add:

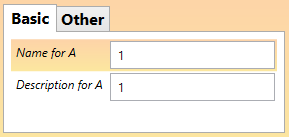
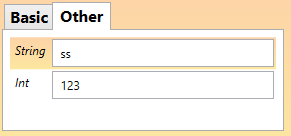
 

Figure 8 -- Sample Nested Groups

The Model object created from this RtObject file will look something like this:

class itemAS

{

public string nameA;

public string descriptionA;

public string stringInA;

public Int32 intInA;

}

class Model

{

public string stringS1;

public bool boolS1;

public Int32 intS1;

public IList<itemAS> itemAS;

}

In the templates described in the next section an instance of Model will be available as Model. So you can reference items with syntax as follows:

Model.stringS1

Model.intS1

Model.itemAS.Count

Mode.itemAS[0].intInA

# Creating a Model with Artie

Since Artie generates XML files using a friendly GUI, why not have Artie generate its own model files. Hence the RtThySelf.xml model. Starting Artie with –object RtThySelf.xml you can use Artie to create a Model for your own system.

# Razor Templates

Usually Razor templates are XML of some type such as HTML, but can really be anything. See the links earlier in the document for help on Razor syntax. Phil Haack also has a nice [quick reference](http://haacked.com/archive/2011/01/06/razor-syntax-quick-reference.aspx).

By default templates are read from the Templates folder under the current folder. Artie processes each file in the folder, writing output to a file of the same name as the template to an Output folder. The remainder of this section shows some of the samples templates that come with Artie. In Artie click to Go button to run the transforms and look at the output each of these templates generates.

Using XML Razor templates usually work quite well since that is what it was designed for. It does start to get tricky when doing plain text or PowerShell since Razor some times gets confused when it needs to drop in and out of C#. I’ve tried to include a wide range of samples that cover many of these idiosyncrasies.

## SimpleXml.xml Sample

This is perhaps the most typical use of Razor, in an XML or HTML file. Razor is very good at understanding what is XML and what is C#.

@\*

This is a simple XML transformation showing how to use Razor

\*@

<root>

<Node name="@Model.stringA1" />

<Node name="@Model.stringA2" />

<SomethingElse>

<More>

<@Model.enumD1 value="true"/>

</More>

</SomethingElse>

</root>

## ItemBXml.xml Sample

This is a typical example for processing XML and an array of items.

<?xml version="1.0" encoding="utf-8" ?>

<!--

This test shows transforming an array to XML

This also sleeps for a bit, allowing you to cancel the transform

-->

@{ System.Threading.Thread.Sleep(3000); }

<test name="we slept so you can cancel" count="@Model.itemB.Count">

@foreach ( var b in @Model.itemB )

{

<itemb name="@b.name" description="@b.description"/>

}

</test>

## NestedTest.txt Sample

This sample shows printing out a hierarchy of items from the RtObject file above. Note how for nested arrays, a Parent member is set on the object to be able to access its parent. There is also a Root member that references the Root model to avoid Parent.Parent.Parent

@\*

This test shows how nested arrays work in Artie

itemA is an array that has itemAB nested within it

\*@

Count of itemA is @Model.itemA.Count

@foreach ( var a in @Model.itemA )

{

@:Name @a.nameA

@:Desc @a.descriptionA

@:Count of itemAB is @a.itemAB.Count

@foreach ( var b in a.itemAB )

{

@: ParentName @b.Parent.nameA

@: Parent.Desc @b.Parent.descriptionA

@: Parent.Parent.stringA1 @b.Root.stringA1

@: Name @b.nameB

@: Desc @b.descriptionB

}

}

## ItemBTransforms.txt

This is a pretty advance sample that really is using C# as a scripting language. It iterates of all the itemB array items and does an additional transform on each one of those, creating additional output files.

@using System.IO

@\*

this sample shows creating a new file

for every object in the Model.itemB array

\*@

Creating output file for each Domain object of which there are @Model.itemB.Count

@foreach ( var x in Model.itemB)

{

// to make things interesting, we do a transform in the transform!

var contents = RazorEngine.Razor.Parse(@"<Domain name=""@Model.name"" provider=""@Model.description"" />", x);

var fname = Path.Combine(Model.CurrentSettings.OutputFolder,x.name);

File.WriteAllText(fname,contents);

// below we switch out of C#

@: Wrote output to @fname

}

# RTValues.xml

You should never have to edit this file. It is written to the current folder by default and contains all the values you entered in the UI so the next time you use Artie, it will have all the same values. It is useful to copy this file around to bootstrap a new system, especially ones with arrays of items since they don’t have default values in RTObject.xml.

# Command Line Switches

From Artie’s help:

RazorTranformation Tool transforms an object described in <objectFile>into all the files in a <templateFolder>

Command line options are:

[-run] [-object <objectFile>] [-values <valuesFile>] [-output <outputFolder>] [-template <templateFolder> [-logFile <logFile>] [-test] [-nosave] [-h|-?] [<overrides....>]

-h -? show this help

-test runs the transforms without saving the output to make sure Razor views work

-nosave does not save the override values in <objectFile>, useful when using overrides

-run runs transforms without pulling up the UI, and the file and folder must exist. Output is written to <logfile>

<objectFile> defaults to .\RtObject.xml

<valuesFile>defaults to .\RtValues

<outputFolder> defaults to ..

<templateFolder> defaults to .\Template

<logFile> defaults to RazorTransformTool.log

<overrides> allows you to override values before the transform using name=value any number of times.

Most of the parameters are self-explanatory and provide overrides to default values. The Overrides parameter deserves some explanation. If you are automating Artie, or just want to be able to pass in new values from the command line to override those in RtValues.xml, you can pass in any number of them as trailing parameters.

For example, in the sample RtObject.xml, we can set the stringA and boolA1 parameters as follows:

razortransform.exe stringA=test boolA1=True

Then values for those two members of the Model in RtValues.xml are replaced with these values and show up in the UI. This is useful when using –run and want to override just a few values. Often in those cases you also use the –nosave option since you don’t want to save the override values in RtValues.xml

## PowerShell

Since Artie is often used in conjunction with running PowerShell scripts, Artie was extended to run the PowerShell within it when a –powershell parameter is passed in. To keep with the generic spirit of Artie, this is totally XML driven and allows PowerShell scripts to be run before Artie loads to possibly populate RtValues.xml (or anything else), then scripts can run after the transforms. It will execute any number of PowerShell snippets, all hosted within Artie.

# Artie Customization

Artie can be extended to allow you to add new data types, or responding to events in the system. One example of an addin is a server-port type that pre-populated new properties with the next available free port for a server. Another could be an IP address control. There is an addin the RtThySelf uses to populate new datatypes on-the-fly.

Details for customization of forthcoming.

1. Document Revisions

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Description** | **Made by** | **Date** |
| 1.0 | Initial version | Jim Wallace | 11/8/12 |
| 2.0 | Update after v2.0 of Artie | Jim Wallace | 3/28/13 |
| 2.1 | Add hyperlink | Jim Wallace | 5/31/14 |
|  |  |  |  |