# Configuring Patient Access

Jonathan Lent

# Overview

## How Does This Technique Work?

The technique used to configure the Patient Access solution is a variant of the technique that uses XML and XSLT to generate context-specific (by environment, by branch, etc.) configuration files for the solution. It uses existing technology and tools rather than relying on custom code and can generate nearly any kind of text-based configuration file the team might need in a project.

To make this technique work for you, you need four things: an XSLT template describing the overall format of your configuration file, an XML file containing context information, an XSLT processor, and some way to combine all three (an MSBuild script, in our case).

### XSLT Template

The XSLT language is definitely the heavy-lifter of all the components. Some aspects of the language can be a little hard to follow, especially the declarative aspect, so we’ll stick to using the simplest and most-useful parts for our needs.

When creating a new configuration template, you will often have a sample or existing configuration file to use as a starting point. The basic creation pattern is to take that sample and paste it inside the xsl:template element in XSLT that looks like the following:

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

    2 <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    3   <xsl:param name="Environment" />

    4   <xsl:param name="Branch" />

    5   <xsl:param name="Version">1.0.0.0</xsl:param>

    6   <xsl:output indent="yes" />

    7   <xsl:template match="/">

    8

    9     Your Configuration Goes Here...

   10

   11   </xsl:template>

   12 </xsl:stylesheet>

Notice that the template declares three parameters: Environment, Branch, and Version. Environment is the name of the deployment environment and can be “Local”, “Dev”, “Test”, “Model”, “Beta”, or “Prod”. Branch refers to the name of the source code branch and can be either “Trunk” or “Support”. Finally, Version is the version number of the build. Version is generally only really useful to the automated build process and defaults to “1.0.0.0” for your local development workstation.

These three parameters should be standard on any new templates you create, even if they aren’t all used in the template at the time of creation. The build script will always pass at least these three parameters to the template so that it can use them to determine which values and sections to fill-in for the configuration file.

There are a couple of additional conventions to follow for these templates. The name of the template should be the name of the configuration file you wish to generate plus the “.xslt” extension. So, “web.config.xslt” would be used to generate a file named “web.config”. Once created, these template files should be stored in a folder named “Configuration” in the root of the project (not solution) to which it applies.

### XML Context File

Since XSLT is used to transform XML files, we need a target file to which we can apply our XSLT templates. In our project, this XML file is named “Settings.xml” and can be found in the “Solution Items\Configuration” folder. We refer to this file as a “context” file because its job is to provide the specific values needed for the various combinations of environment/branch configurations needed when building and deploying the solution. The following snippet is a heavily-edited example of the Settings.xml file used by Patient Access:

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

    2 <Settings>

    3   <Databases Description="DBMS systems used by the application. Will be encrypted in final form.">

    4     <Db2 Environment="Local,Dev,Test">Datasource={0}; LibraryList=PADEV,\*USRLIBL...</Db2>

    5     <Db2 Environment="Model,Beta,Prod">Datasource={0}; LibraryList=PACCESS,\*USRLIBL...</Db2>

    6

    7     <SqlServer Environment="Local" Branch="Trunk">Database=PatientAccessTrunkLocal;...</SqlServer>

    8     <SqlServer Environment="Local" Branch="Support">Database=PatientAccessSupportLocal;...</SqlServer>

    9

   10     <SqlServer Environment="Dev" Branch="Trunk">Database=PatientAccessTrunkDev;...</SqlServer>

   11     <SqlServer Environment="Dev" Branch="Support">Database=PatientAccessSupportDev;...</SqlServer>

   12     <SqlServer Environment="Test" Branch="Trunk">Database=PatientAccessTrunkTest;...</SqlServer>

   13     <SqlServer Environment="Test" Branch="Support">Database=PatientAccessSupportTest;...</SqlServer>

   14

   15     <SqlServer Environment="Model" Branch="Any">Database=PatientAccessModel;...</SqlServer>

   16

   17     <SqlServer Environment="Beta" Branch="Any">Database=PatientAccessBeta;...</SqlServer>

   18

   19     <SqlServer Environment="Prod" Branch="Any">Database=PatientAccessProd;...</SqlServer>

   20   </Databases>

   21   <ExternalSystems Description="Systems on which Patient Access is dependent">

   22     <Edv Description="Perot system used for validation of patient benefits, credit, address information">

   23

   24       <AddressValidationServiceUrl Environment="Local,Dev,Test">

   25         http://172.27.135.47/edv/services/AddressValidationService</AddressValidationServiceUrl>

   26       <BenefitsValidationServiceUrl Environment="Local,Dev,Test">

   27         http://172.27.135.47/edv/services/BenefitsValidationService</BenefitsValidationServiceUrl>

   28       <BenefitsValidationFusServiceUrl Environment="Local,Dev,Test">

   29         http://172.27.135.47/edv/services/BenefitsValidationFusService</BenefitsValidationFusServiceUrl>

   30       <ComplianceCheckerServiceUrl Environment="Local,Dev,Test">

   31         http://172.27.135.47/edv/services/ComplianceCheckerService</ComplianceCheckerServiceUrl>

   32       <CreditValidationServiceUrl Environment="Local,Dev,Test">

   33         http://172.27.135.47/edv/services/CreditValidationService</CreditValidationServiceUrl>

   34       <PriorAccountBalanceServiceUrl Environment="Local,Dev,Test">

   35         http://172.27.135.47/edv/services/PriorAccountBalanceService</PriorAccountBalanceServiceUrl>

   36

   37       <AddressValidationServiceUrl Environment="Model">

   38         http://edv.mdltenethealth.net/edv/services/AddressValidationService</AddressValidationServiceUrl>

   39       <BenefitsValidationServiceUrl Environment="Model">

   40         http://edv.mdltenethealth.net/edv/services/BenefitsValidationService</BenefitsValidationServiceUrl>

   41       <BenefitsValidationFusServiceUrl Environment="Model">

   42         http://edv.mdltenethealth.net/edv/services/BenefitsValidationFusService</BenefitsValidationFusServiceUrl>

   43       <ComplianceCheckerServiceUrl Environment="Model">

   44         http://edv.mdltenethealth.net/edv/services/ComplianceCheckerService</ComplianceCheckerServiceUrl>

   45       <CreditValidationServiceUrl Environment="Model">

   46         http://edv.mdltenethealth.net/edv/services/CreditValidationService</CreditValidationServiceUrl>

   47       <PriorAccountBalanceServiceUrl Environment="Model">

   48         http://edv.mdltenethealth.net/edv/services/PriorAccountBalanceService</PriorAccountBalanceServiceUrl>

   49

   50     </Edv>

   51   </ExternalSystems>

   52 </Settings>

Notice a couple of things here. The innermost elements in the file have Environment and/or Branch attributes on them. These are used by XSLT to determine which values should be used when generating a configuration. The environment attribute on line 24, for example, indicates that this same value can be used for any configuration generated for the Local, Dev, or Test environments. This particular setting does not vary by branch, so there is no branch attribute and the value will be used regardless of which source branch (trunk or support) is being targeted. Lines 7-8 show a scenario where the value does vary by source branch. In this case, the SqlServer 2005 databases vary both by environment and branch (notice the database name), so we need to factor in the branch value when choosing the value to place in our generated configuration file.

There is no formal schema for this XML context file, but there are a few practices that should be followed when possible. First, try to avoid creating new groups inside the “Settings”, “ServerSettings”, and “ClientSettings” elements if you can logically place your new setting on one of the existing groups. Second, use the “text” node (the area between tags) to hold the value and reserve attributes for the Environment and Branch values. Finally, group your settings logically and don’t feel constrained by the format of the configuration files you are trying to generate. That’s a bit nebulous, so let me give you an example. Say you have two values pertaining to the eDV system. One will go in appSettings in the generated configuration file, and one will go in some custom configuration section in the same file. Even though the values will end up in very different places in the generated output, you should put them both in the eDV section of the Settings.xml file instead of just lumping them together in the more general ServerSettings section. This tends to make the values easier to understand and also makes them a little more generally applicable across multiple configuration files.

### XSLT Processor

There is not a lot to say on the topic of the actual XSLT processor. For Patient Access, we are using an open source MSBuild task called NXSLT from the Mvp.Xml project to process our templates. This processor was chosen over others because it provides the cleanest output and most extended features when compared to some of the other tools.

### MSBuild Script

The previous three components of this technique are useless unless we have some convenient way to “glue” them all together. For our purposes, we use MSBuild. This could pretty easily be done with any number of other mechanisms, but MSBuild is guaranteed to be on the system if DotNET is installed and it integrates nicely into the overall build process.

As mentioned earlier, all XSLT files can be found in the Configuration directory under the root folder of the project which is being configured. You will also find a file named Generate.proj in this folder. This is the MSBuild script used to generate all the configurations needed for the project. In general, this file is pretty much the same across projects, save for the following section:

   59 <ItemGroup>

   60   <ConfigurationFile Include="$(MSBuildProjectDirectory)\Log4Net.config.xslt">

   61     <OutFile>$(MSBuildProjectDirectory)\..\Log4Net.config</OutFile>

   62   </ConfigurationFile>

   63   <ConfigurationFile Include="$(MSBuildProjectDirectory)\Web.config.xslt">

   64     <OutFile>$(MSBuildProjectDirectory)\..\Web.config</OutFile>

   65   </ConfigurationFile>

   66   <ConfigurationFile Include="$(MSBuildProjectDirectory)\QuartzJobs.xml.xslt">

   67     <OutFile>$(MSBuildProjectDirectory)\..\QuartzJobs.xml</OutFile>

   68   </ConfigurationFile>

   69   <ConfigurationFile Include="$(MSBuildProjectDirectory)\Downloads.xml.xslt">

   70     <OutFile>$(MSBuildProjectDirectory)\..\Downloads.xml</OutFile>

   71   </ConfigurationFile>

   72   <ConfigurationFile Include="$(MSBuildProjectDirectory)\PatientAccessClientManifest.xml.xslt">

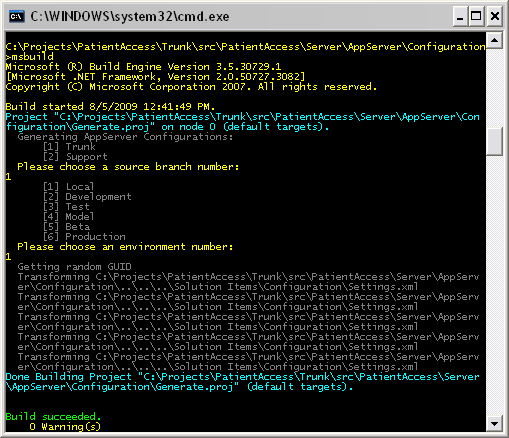
   73     <OutFile>$(MSBuildProjectDirectory)\..\Manifests\PatientAccessClientManifest.xml</OutFile>

   74   </ConfigurationFile>

   75 </ItemGroup>

This ItemGroup section tells the rest of the script what the input and outputs are for a particular configuration file. On line 60, the Include attribute gives the relative path to the XSLT configuration template for a Log4Net configuration file. Line 61 tells the script to place the generated configuration in a file named Log4Net.config one directory up from the current Configuration directory. So, if you need to add additional configurations to a particular project, this ItemGroup is where you would add the new entry needed to tell MSBuild how to process it.

Executing the MSBuild script is fairly simple. Open a Visual Studio command prompt in the Configuration directory and type msbuild. Since there is only one project file in the directory, MSBuild will automatically choose it for you and run the default target. The script will ask you which Environment and Branch values you wish to provide to the XSLT process and will then proceed to generate the configuration files for you. It looks something like this:



## Why Do It This Way?

There are a couple of advantages to doing the configuration this way. First, there isn’t any source code to maintain other than the configuration files themselves. This should help avoid the scenario where the original tool-builder is not around and nobody really wants to maintain somebody’s weird, old source code. Second, unlike the Enterprise Library tools we’ve tried out, it is very easy to customize all parts of a configuration file with this technique – not just certain sections. Finally, the build process uses the same scripts to configure the solution as the developer -- which helps keep things consistent and predictable.

## Why Are There Little Red Dots and Yellow Triangles In Visual Studio?

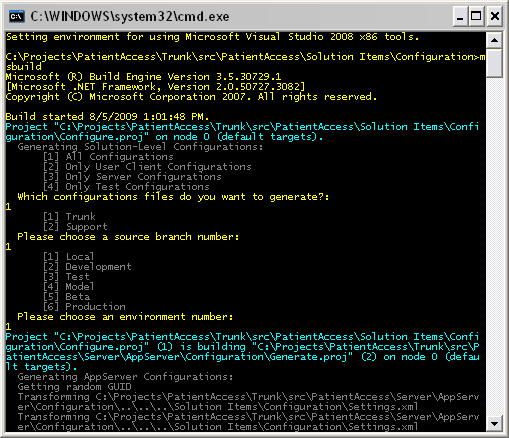
Since the configuration files are generated, we really do not want to store them in source control. This lets you tweak and play with the generated files as much as you would like and not have to worry about accidentally committing a bad value or setting to the source code repository. Still, even though we don’t want the generated files in source control, it’s really convenient to be able to see them as part of the solution. To do this, we include the file in the project, but tell Visual Studio to exclude it from source control. Once this is done, the file remains in the project but is marked with a red dot to indicate that it is not under source control.

The yellow warning triangle is Visual Studio indicating that it cannot find a file on the file system. This will typically appear if you have a clean copy of the source or one that has been scorched and you open the solution before you generate the configuration files. It’s totally harmless and can be remedied by simply generating a set of configuration files. Note that the warning triangles will not go away unless you double click the file after generating it or re-open the project.

# Running the Scripts

## Configuring the Entire Solution At One Time

Many times during the course of your work you are going to want to generate a set of configuration files for the solution. This might be because you have made changes to the Settings.xml file or you might just want to make sure you have a consistent set of configurations. Regardless of the reason, the Configure.proj script found in Solution Items\Configuration exists to help configure the entire solution in one step. It doesn’t actually generate any of the configurations itself, it simply asks you a few questions and then uses the answers to cascade through the solution and run the Generate.proj scripts. Using this script is very similar to using the Generate.proj scripts except that it asks you an additional question about the scope of the configuration:



Frankly, there has not been much use for any other scope choices other than “All Configurations”, so don’t be shocked if that goes away in the near future.

## Configuring an Individual Project

The MSBuild script section actually covered the mechanics of configuring an individual project, so here we’ll cover the reason why you might want to do it.

From time to time, you might find it useful to configure a portion of your local application to point to an environment other than the local one. This allows you to debug the process on your local machine but to have it behave as though it were deployed in another environment. This is invaluable if the development environment does not have the necessary information or services to reproduce a bug.

The most common form of this kind of re-configuration is to configure the AppServer project to point to a different environment. To do this, you should first run the Configure.proj file and reset the solution to a local configuration – just to be safe. Next, you go to the AppServer Configuration directory and run the Generate.proj file, selecting the environment you would like it to mimic instead of the local one. At this point, everything is pointing to the local environment, but your AppServer is running as though it were in a different one (test, for example).

One note: be careful when using this technique to point to Beta or Prod. This is useful in some analysis scenarios but is very risky as you can accidentally alter production data when using these environments.

# Adding New Configuration Values

## Adding Static Values

A lot of the configuration values you will want to create will be the same regardless of environment or branch. These static values should be added directly to the XSLT template. In the sample below, the description value (line 12) is a good example of such a static value.

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

    2 <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    3   <xsl:param name="Environment" />

    4   <xsl:param name="Branch" />

    5   <xsl:param name="Version">1.0.0.0</xsl:param>

    6   <xsl:output indent="yes" />

    7   <xsl:template match="/">

    8   <downloads  xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

    9     <download>

   10       <name>Patient Access for Windows Version <xsl:value-of select="$Version"/></name>

   11       <installfile>PatientAccess.msi</installfile>

   12       <description>Installs the client software for the Patient Access system</description>

   13     </download>

   14   </downloads>

   15   </xsl:template>

   16 </xsl:stylesheet>

## Adding Per-Environment/Branch Values

Of course, if we could only generate static values this technique would be pretty worthless. The example below shows how to code XSLT in a template (this is from the AppServer’s web.config.xslt file) to get a value based on a supplied environment and/or branch value:

1 <connectionStrings>

2  <add name="ConnectionString"

 3       connectionString="{/Settings/Databases/SqlServer[contains(@Environment,$Environment) and (@Branch=$Branch or @Branch='Any')]/text()}"/>

 4  <add name="DB2ConnectionTemplate"

 5       connectionString="{/Settings/Databases/Db2[contains(@Environment,$Environment)]/text()}"/>

 6 </connectionStrings>

XSLT Note: The curly brace syntax can be used inside any attribute and essentially tells the XSLT processor to evaluate the expression inside the braces and place the resulting value in the attribute. This same shortcut does not work for text nodes (the space between a set of tags). For that scenario you will need to use the XSLT value-of tag:

  <name>PurgeOldQueuedEmployerAddressTrigger-<xsl:value-of select="@Name"/></name>

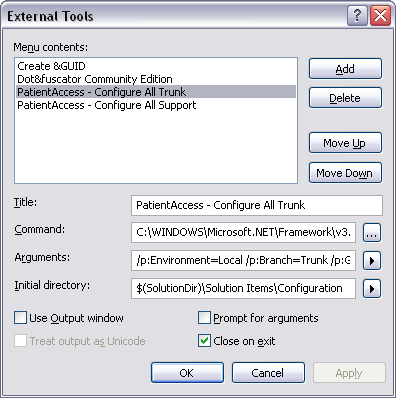
In the first example, the DB2ConnectionTemplate value is retrieved from the Settings/Databases/Db2 element whose Environment attribute contains the value assigned to the $Environment parameter (which was passed to the XSLT processor by MSBuild). The ConnectionString value is very similar, but adds the additional qualification that the Branch must either equal the value provided by MSBuild in the Branch parameter or be equal to “Any”. That last part is important because Patient Access has a peculiar deployment environment where the branch is important only up to the Model environment.

This is very rudimentary XSLT code, to say the least. It is possible to do a lot more with the language than is explained in these samples. I would caution against getting too fancy with the syntax, however, as it tends to increase the learning curve for others.

# Tips and Tricks

## Adding External Tools in Visual Studio 2008

It would seem that one of the most common configuration tasks is configuring the entire solution at once. While this is not exactly rocket science, it can be a little tedious to go through the whole Visual Studio command prompt process. To make this easier, you can create two external tool entries for both your Local/Trunk and Local/Support environments using Visual Studio’s “External Tools” menu option:



The title can be anything you like, of course. The Command should be the full path to the MSBuild.exe file in the DotNet 3.5 framework directory. The initial directory should be exactly like shown in the picture. The really important part here is the Arguments line:

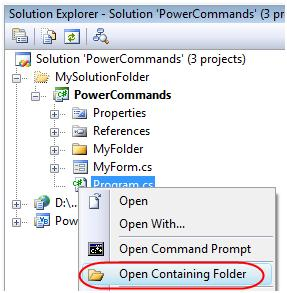
For Local Trunk: /p:Environment=Local /p:Branch=Trunk /p:GenerationOption=All

For Local Support: /p:Environment=Local /p:Branch=Support /p:GenerationOption=All

Once you have these two external tool entries added, configuring the solution should be as simple as selecting the appropriate one from the Tools menu.

## Adding PowerCommands in Visual Studio 2008

One of the nicest freebies around for Visual Studio 2008 is the [PowerCommands](http://code.msdn.microsoft.com/PowerCommands) suite. In addition to a bunch of other really great tools, it adds the ability to open a Visual Studio command prompt from the solution explorer (the picture is from the tool site, so ignore the red circle around the Open Containing Folder item):



The Open Command Prompt action can be very useful if you want to configure a single project. All you need to do is find that project’s Configuration directory in Solution Explorer and then select “Open Command Prompt”.

## Reading More About XSLT

The hardest part of this technique is picking up the XSLT language if you are not already familiar with it. Before you run out and buy a book, however, you might want to take a look at the following sites for some more inspiration and guidance:

[W3Schools.com XSLT Tutorial](http://www.w3schools.com/xsl/)

[W3c XSLT 1.0 Specification (painful to read, but often useful)](http://www.w3.org/TR/xslt)

[XSL Transformations Article on Wikipedia](http://en.wikipedia.org/wiki/XSL_Transformations)