25

Customizing Visual Studio for ASP.NET MVC

This chapter covers:

* Creating custom T4 templates
* Using custom T4 templates
* Exporting a custom test project template
* Adding custom test project templates

Tooling within Visual Studio can make building ASP.NET MVC applications faster. Any task that you perform over and over is a candidate for automation and tooling. A computer can perform a task faster and with more accuracy than a human can, especially when the task is performed over and over. We will look at two quick ways of customizing these tools, specifically the controller, view, and project generators.

25.1 Creating custom T4 templates

T4 is a little-known feature of Visual Studio that stands for Text Template Transformation Toolkit. It is a code generation toolkit. Its templates allow you to customize how files are generated using a familiar syntax. Chapter 21 covered T4MVC, which is a set of T4 extensions provided by the CodePlex Foundationk project, MvcContrib. Under the covers, when you install ASP.NET MVC on top of Visual Studio, you get templates for adding items such as areas, views, and controllers. For instance, if you right click on an action, you’ll see an option to open the Add View dialog, shown in figure 25.1. In this dialog, you can choose the name of the view, the view model type, and the master page. If you select a strongly typed view, you have the option of choosing an automatic view template. The options are Empty, List, Create, Details, Delete. Figure 25.1 shows us selecting Create for our view content and Product for our view model.

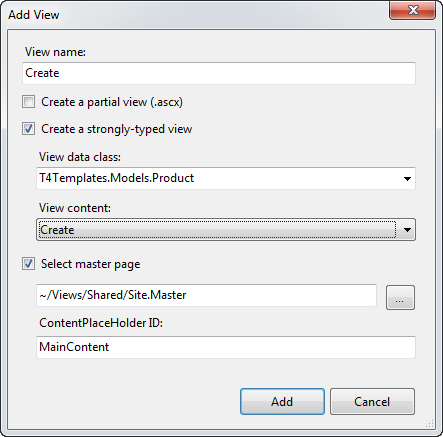


Figure 25.1 The Add View dialog allows you to auto-generate scaffolding for your model.

The options in the View Content dropdown list are T4 templates that are located on 64-bit system in

C:\Program Files (x86)\Microsoft Visual Studio 9.0\Common7\IDE\ItemTemplates\CSharp\Web\MVC 2\CodeTemplates

If we press Add, we’re given a complete form, generated for us by Visual Studio using the default template. Our view now looks like listing 25.1

Listing 25.1 The autogenerated Create view based on the Product object

<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"

Inherits="System.Web.Mvc.ViewPage<T4Templates.Models.Product>" %> #1

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">

Create

</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

<h2>Create</h2>

<% using (Html.BeginForm()) {%> #2

<%= Html.ValidationSummary(true) %>

<fieldset>

<legend>Fields</legend>

<div class="editor-label">

<%= Html.LabelFor(model => model.Id) %>

</div>

<div class="editor-field">

<%= Html.TextBoxFor(model => model.Id) %>

<%= Html.ValidationMessageFor(model => model.Id) %> #3

</div>

<div class="editor-label">

<%= Html.LabelFor(model => model.Name) %>

</div>

<div class="editor-field">

<%= Html.TextBoxFor(model => model.Name) %>

<%= Html.ValidationMessageFor(model => model.Name) %> #3

</div>

<div class="editor-label">

<%= Html.LabelFor(model => model.Description) %>

</div>

<div class="editor-field">

<%= Html.TextBoxFor(model => model.Description) %>

<%= Html.ValidationMessageFor(model => model.Description) %> #3

</div>

<div class="editor-label">

<%= Html.LabelFor(model => model.ActiveDate) %>

</div>

<div class="editor-field">

<%= Html.TextBoxFor(model => model.ActiveDate) %>

<%= Html.ValidationMessageFor(model => model.ActiveDate) %> #3

</div>

<div class="editor-label">

<%= Html.LabelFor(model => model.RetireDate) %>

</div>

<div class="editor-field">

<%= Html.TextBoxFor(model => model.RetireDate) %>

<%= Html.ValidationMessageFor(model => model.RetireDate) %> #3

</div>

<p>

<input type="submit" value="Create" /> #4

</p>

</fieldset>

<% } %>

<div>

<%=Html.ActionLink("Back to List", "Index") %> #5

</div>

</asp:Content>

As you can see, lots of code is generated for us. It contains the strongly-typed declaration for Product **(1)**, a basic form **(2)**, with fields corresponding to the object, complete with validation **(3)**, Submit button **(4)**, and back link **(5)**. This can get us started building the application quickly. Of course this is just a starting point, and you’re free to customize it from here. This template is static, and you can create a different, application-specific template for the Create view.

Add a folder in your project called CodeTemplates. Into this folder, copy the contents of the default template folder. You can create subfolders corresponding to the different types of templates (figure 25.2).

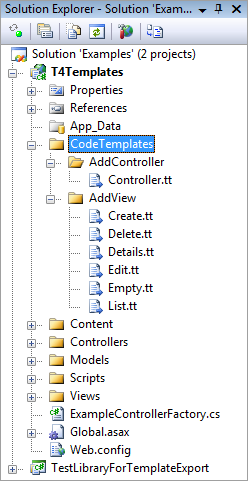


Figure 25.2 Copy the templates from C:\Program Files (x86)\Microsoft Visual Studio 9.0\Common7\IDE\Templates\CSharp\Web\MVC into your project under a CodeTemplates folder to customize them.

These templates will be effective for the current project only. You are free to alter the templates here for your project. You can also add more items to this list. Adding another .tt file in this folder will enable it for selection in the Add View dialog as show in figure 25.3.

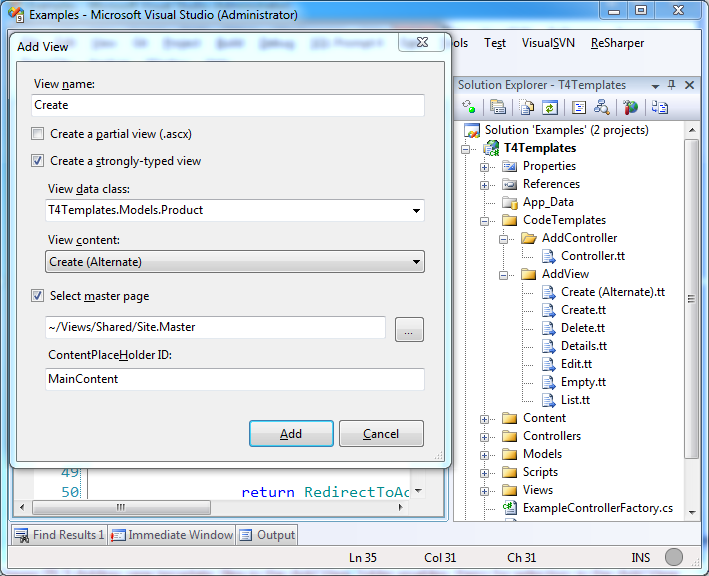


Figure 25.3 Adding new template files in the Add View folder enables them for selection in the Add View dialog.

The templates themselves are fairly complex. Here is an excerpt from the Controller.tt template:

<#@ template language="C#" HostSpecific="True" #>

<#

MvcTextTemplateHost mvcHost = (MvcTextTemplateHost)(Host);

#>

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace <#= mvcHost.NameSpace #>

{

public class <#= mvcHost.ControllerName #> : Controller

{

//

// GET: <#= (!String.IsNullOrEmpty(mvcHost.AreaName)) ? ("/" + mvcHost.AreaName) : String.Empty #>/<#= mvcHost.ControllerRootName #>/

...... more ....

As you can see, code blocks are denoted by <# #> blocks. Each template has a Host property that contains basic context information. For MVC templates, this is actually of type MvcTextTemplateHost, so we can see here that the template is casting the Host property and storing it in a variable called mvcHost for use later in the template.

Caution about T4 code generation

As interesting as T4 code generation seems, it is only one type of code generation. Code generation can be a terrific tool to leverage in order to produce software functionality quickly and of higher quality. There are two main types of code generation. The first is using a technique that produces code that is meant to be versioned in a version control system (VCS). The second type uses a technique to generate the code in the build of the software. In other words, the generated code would never be tracked in a version control system.

T4 generation is the first type. It allows you to quickly generate file, but then you have to version and maintain them yourself. I caution you to limit this type of code generation. Code generation makes laying down code very easy. A downside is that it accelerates the rate at which you accumulate duplicate code.

Overall, favor code generation techniques where you never version or maintain the generated code. In this way, you can modify the templates and regenerate code at build time. Some code generators actually work at runtime: generating code right before executing it.

T4 templates can be a little intimidating but you can do a lot of things with them. If you are interested in customizing the templates, download Visual T4 Editor for Visual Studio 2008 Community Edition (free) from Clarius Consulting. This will give you syntax highlighting, which is really helpful when you find yourself writing code that writes code! The tool can be downloaded at <http://www.visualt4.com/downloads.html>. To learn more about T4 Template syntax and the ASP.NET MVC integration, check out <http://blogs.msdn.com/webdevtools/archive/2009/01/29/t4-templates-a-quick-start-guide-for-asp-net-mvc-developers.aspx>.

25.2 Adding a custom test project template to the new project wizard

When you first create an ASP.NET MVC project, you’re eventually greeted with the dialog shown in figure 25.4:

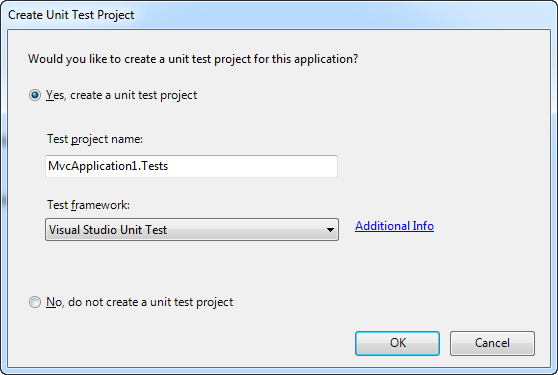


Figure 25.4 When you create a new project, you are asked if you want to create a unit test project.

Unfortunately, the only available test framework that is provided out of the box is the Visual Studio Unit Test framework. Developers who are experienced with testing will no doubt prefer NUnit, MbUnit, or xUnit.NET. There is hope! You can add your framework of choice to this dialog box (and simultaneously implement a custom project template).

The first step is to create a project that represents what you want when you create new ASP.NET MVC applications with the test project included. Make sure all third-party references (such as NUnit, MvcContrib.TestHelper, Rhino Mocks) are set to Copy Local. Then go to File > Export Template. Follow the wizard here, which will result in a single zip file. Copy this zip file to:

C:\Program Files (x86)\Microsoft Visual Studio 9.0\Common7\IDE\ProjectTemplates\CSharp\Test.

(If you’re running on a 32-bit machine, then adjust the path to C:\Program Files\ accordingly). Now that you’ve got the template in the right place, close all instances of Visual Studio, open up the Visual Studio 2008 Command Prompt (as Administrator if UAC is enabled), and run:

devenv /installvstemplates

This will take a few seconds. Now that you have a project template installed into Visual Studio, open regedit and navigate to one the following based on your computer processor architecture:

* HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\VisualStudio\9.0\MVC2\TestProjectTemplates
* HKEY\_LOCAL\_MACHINE\SOFTWARE\Wow6432Node\Microsoft\VisualStudio\9.0\MVC2\TestProjectTemplates

Here in table 25.1, you’ll find the default Visual Studio Unit Test key. To create a custom entry, make a new key here, and add the following String values:

Table 25.1 These registry values configure the unit test project settings.

|  |  |
| --- | --- |
| Value | Description |
| Package | Leave blank unless you have a custom Visual Studio package GUID to register here |
| Path | Usually CSharp\Test |
| TestFrameworkName | The name that you want to appear in the Unit Test Framework dropdown |
| AdditionalInfo | A URL that provides the user more information about your framework or template. When the user clicks on Additional Info, the browser will navigate to this URL |
| Template | The name of the zip file that contains the template |

Note

On 32-bit machines, the registry path is slightly different (Remove Wow6432Node). In addition, the Program Files path is actually C:\Program Files\. Be sure to adjust accordingly for your system as shown in figure 25.5.

Figure 25.5 shows a new template installed in this location.

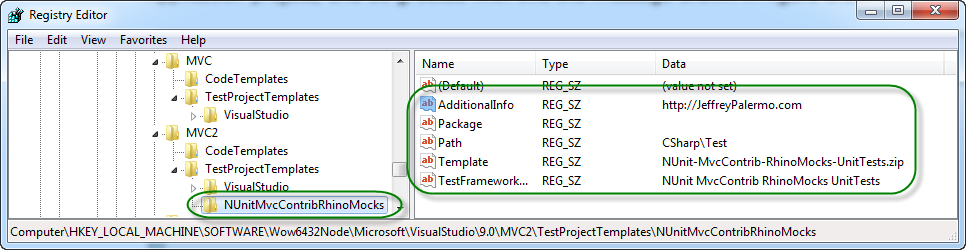


Figure 25.5 Adding a registry entry for a new custom test project template. Note that this registry path is for 64-bit machines.

With all of this in place, we can launch Visual Studio, create a new ASP.NET MVC Web Application project, and be greeted with this the message shown in figure 25.6.

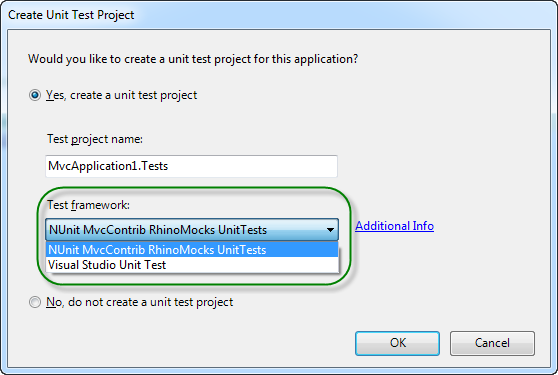


Figure 25.6 Our new test template is now available in the Create Unit Test Project dialog box.

25.3 Summary

In this chapter, you have seen some of the ways to modify Visual Studio as it relates to the ASP.NET MVC Framework. You have seen how to use the built-in T4 templates to create controllers and actions. You have also seen how to modify and create new T4 templates for new and interesting types of code files. You have also seen how to create and install custom test project templates. The industry will never standardize on a single test framework. You now know the steps necessary to create and install the test template that you need for your project. Now that we have seen how to customize Visual Studio, the next chapter will cover some best practices to apply while working with ASP.NET MVC.