T4

Succinctly

by Nick Harrison

Syncfusion | Technology Resource Portal

T4 Succinctly

By
Nick Harrison

Foreword by Daniel Jebaraj

Technical Reviewer: Matt Duffield

Copy Editor: Courtney Wright

Acquisitions Coordinator: Hillary Bowling, marketing coordinator, Syncfusion, Inc.

Proofreader: Morgan Cartier Weston, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to "enable AJAX support with one click," or "turn the moon to cheese!"

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and "Like" us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Nick Harrison has more than 20 years of development experience and has worked with .NET since the first Community Technical Preview (CTP) went golden. He believes that .NET is a wonderful development environment that gets better with each update.

Nick also stays active in the local development community as a frequent speaker at local user groups and code camps. He is an author for the technical journal Simple-Talk and an occasional blogger on the blog community Geeks with Blogs.

In 2007, he met the true love of his life and was fortunate enough to start a family with Tracy along with Morgan Faye. To this day, they remain a profound source of joy and a strong counterbalance to the chaos of daily life.

Chapter 1 Introduction

Introducing T4

T4 has been available as a part of Visual Studio since Visual Studio 2005. Incremental improvements have been made with each subsequent release of Visual Studio, so it keeps getting better.

Note: It is possible to use T4 outside of Visual Studio hosted in your own application, but this book will focus on using the toolkit with Visual Studio.

T4 is a code generator. Actually, it's more than that. T4 stands for:

- Text
- Templating
- Transformation
- Toolkit

Notice that neither code nor generator shows up in the name. T4 is based on text in templates and how those templates get transformed into something useful. In many cases, this involves applying metadata to a template to transform it into source code in a variety of languages.

Some possible outputs from a T4 template:

- C#
- VB.Net
- SQL
- HTML
- JavaScript
- Text
- CSS

The template also has code to drive the transformation. This code can be in C# or VB.NET—depending on which language you are most comfortable with—regardless of what type of text is output.

In this book, we will use C# to drive the transformation, and when generating code, we will generate C#, but this is not the only option.

Why Generate Code?

Familiarity with code generation tools is key for modern software engineers. Much of the code that we need to write can be handed over to a trusted code generator to handle for us, and frankly, much of our time would be better spent designing software than dealing with the subtle nuances of repetitive code that the computer could write with minimal input from us.

Generating code is particularly useful in places where we have lots of repetitive steps that can easily be described with metadata, or where best practices have not yet fully gelled. Humans get bored and sloppy with repetitive, tedious tasks. Computers do not, and when the best practices are still being worked out, it makes sense to generate the code so that as the best practices change, you only have to change the generator to have all of the code built from the generator conforming to the evolving best practice.

We can also leverage code generation to kick-start a project. Instead of generating code that should never be changed and is expected to compile and run successfully without developer intervention, this type of code generation lays out the broad strokes and expects a developer to fill in the rest. This type of code generation is often referred to as scaffolding.

Scaffolding is great for providing developer guidance and making sure that you don't have to face an empty project and wonder where to start, but it does nothing to help with evolving best practices or changes to the metadata. Scaffolding generators are expected to run only once, with the developer handling everything from there. They provide a catalyst from which to start.

Actual Applications

Whether you know it or not, you're probably already using a code generator. Many tools and frameworks use them behind the scenes to make life easier for the developer.

Increasingly, T4 is becoming the tool of choice for generating this code. The template developers have control over the code that is being generated while still allowing the developer, as a user of the template, the ability to modify the generated output by customizing the templates.

MVC and Entity Framework are possibly the two biggest examples of using T4 behind the scenes. Entity Framework expects the output of its templates to be complete and usable without any developer intervention. These templates generate partial classes, so if you need to add anything, add it in a partial class in a separate file. This will protect your changes whenever the code is regenerated, and may potentially be regenerated fairly often.

MVC takes the scaffolding approach to code generation. The generator is expected to run once for any artifact that you generate. This generation is intended to help you hit the ground running, but after that, you are on your own. You can make any changes that you need to the generated code, but there is no way to re-run the generator unless you delete the outputted file and start over.

Both approaches have their strengths and weaknesses, as we shall see.

Practical Applications

As a toolkit, T4 allows you to do pretty much whatever you need to do, but there are some guidelines that can help us hone in on areas that lend themselves more easily to reaping the benefits of code generation:

- You must have a source of metadata that is easy to get to, easy to maintain, and easy to keep clean.
- The generated code must conform to a pattern that is easily defined by the available metadata.
- A generation template should be able to create a collection of artifacts based on the available metadata.

With these guidelines in mind, we can identify several possible templates, depending on the nature of your project. If you are using Entity Framework, you are already using T4, but if you are not, there are many opportunities to introduce T4 to a project without necessarily having to completely change your data access logic.

Some applications include:

- Automate binding parameters to a command object.
- Automate writing stored procedures to encapsulate access to a table.
- Automate extracting data from a data table into a POCO.

If you are already using MVC, you are already using T4 for the scaffolding—but it can do even more for you, inlcuding:

- Generate models and views (with a little extra metadata).
- Generate JavaScript to automate calling Web API methods.
- Generate the skeleton of a style sheet.

If you are not using MVC, you can get inspiration from all the ways that MVC uses T4.

Minimal Requirements

T4 has been integrated into Visual Studio since 2008, and was even available as a separate download to be added to Visual Studio 2005. As long as you are using Visual Studio 2008 or later, you have almost everything you will need. I say almost, because while T4 is built into Visual Studio, and Visual Studio understands how to run these templates, support for editing the templates is marginal without a good extension to provide some much-needed syntax highlighting and IntelliSense. I recommend starting with the community edition provided by Tangible. The community edition is free and will provide basic syntax highlighting, as well as limited IntelliSense support. This is more than enough to get you started, but as your templates get more complex, you can upgrade to a professional version that will give you full IntelliSense support, as well as the ability to debug your templates.

Note: While this is my favorite extension, it is not the only one available. A quick search through the extension manager will turn up several more.

Summary

In this chapter we have covered a lot of introductory material needed to provide context for where code generation and T4 specifically can fit into your projects. We have outlined some basic qualities that any successful application of code generation should have. We have reviewed the two different approaches to code generation: an all-inclusive approach where the generator produces code that should never be modified by the developer, and a simpler, scaffolding approach where the goal is to simply jumpstart a development effort, but rely on the developer to fill in the gaps.

We have reviewed some current, active examples where you might see T4 in action without realizing it: Entity Framework and MVC. We have also touched briefly on some example applications that we will explore more thoroughly in upcoming chapters.

Finally, we have seen the first hints at just how important metadata is to generating code. The second half of this book will focus on tracking down this metadata.

Chapter 2 Getting Started

Two Types of Templates

We have two types of templates: text templates and runtime text templates. Text templates will produce output directly controlled by the template, while runtime text templates will produce a class that is called by your code to produce the output described in your template.

This chapter will focus on text templates, but everything that we will say about the structure and syntax for a T4 template will hold true for both types. The difference will come in the types of output produced, and how they are used.

When you select **Add New Item**, you can filter to show the template items.

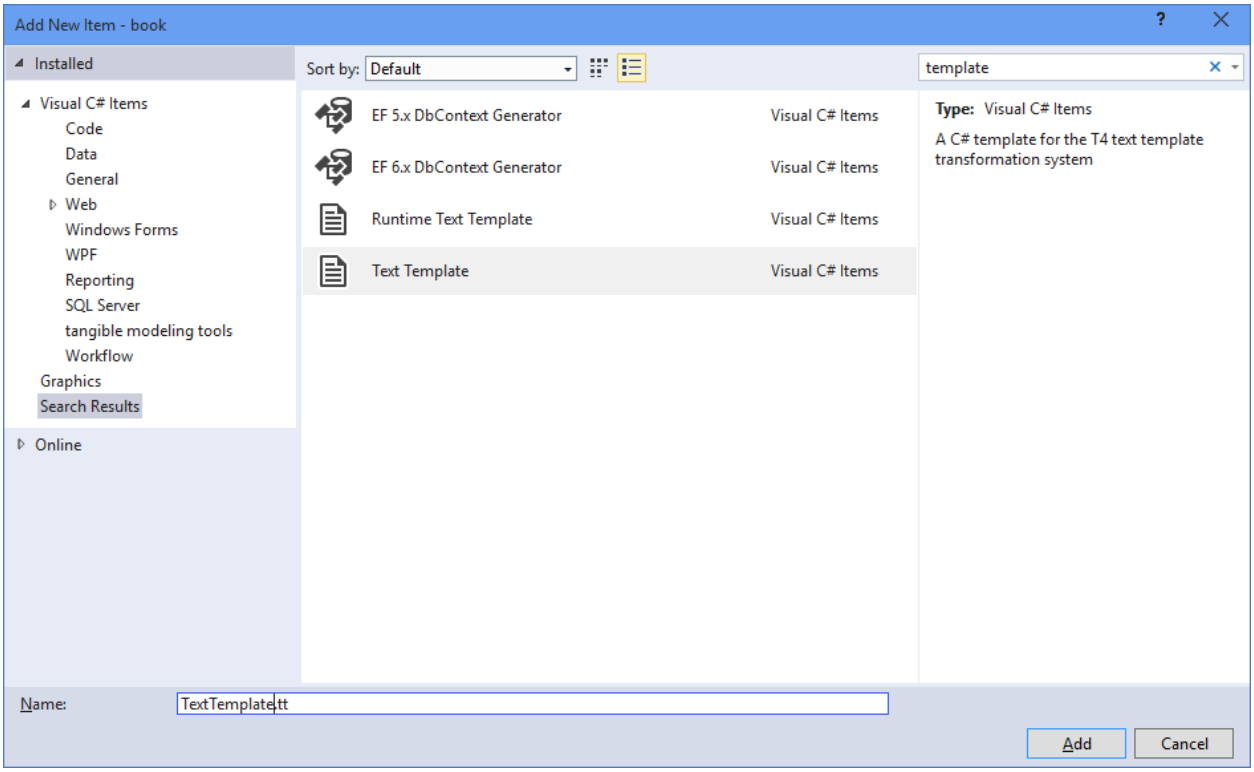

Figure 1: Add New Item Templates

Adding Templates to Projects

Let's start with a Console Application project. From the **Add New Item** dialog, select **TextTemplate**. For our first template, we will keep the unimaginative name, **TextTemplate.tt**. The initial code in the template will be very basic.

```
<#@ template debug="true" hostSpecific="true" #>
<#@ output extension=".cs" #>
<#@ Assembly Name="System.Core" #>
<#@ Assembly Name="System.Windows.Forms" #>
<#@ import namespace="System" #>
<#@ import namespace="System.IO" #>
<#@ import namespace="System.Diagnostics" #>
<#@ import namespace="System.Linq" #>
<#@ import namespace="System.Collections" #>
<#@ import namespace="System.Collections.Generic" #>
<#


#>
```

Code Listing 1: Initial Basic Template

Saving this template will run the Custom Tool, which will process the template. You can also run the Custom Tool by selecting **Run Custom Tool** in the context menu for the template in Solution Explorer.

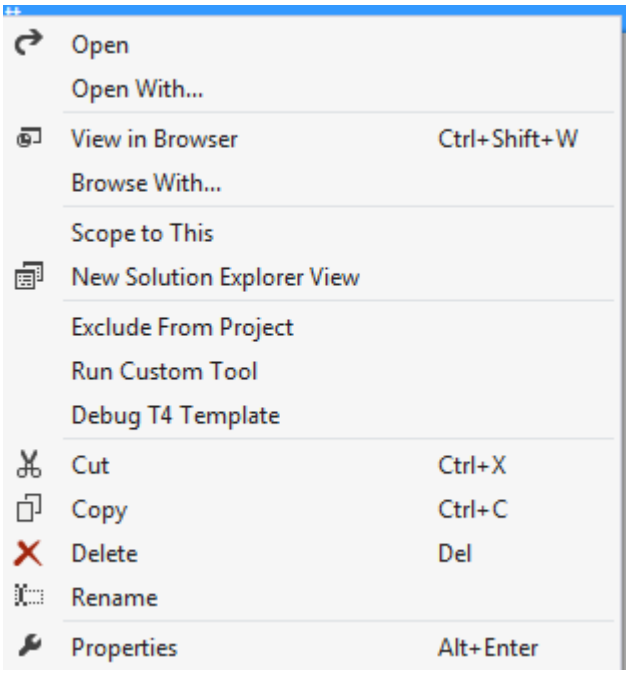

Figure 2: Solution Explorer Context Menu

After running the Custom Tool, you will find a new file nested under the template in the Solution Explorer. This file will have the same name as the template, but with a **.cs** extension. This will be the output of running the template. At this point the output will be blank because we haven't actually told our template to do anything yet.

Add the following bit of code to the template, and let's look at the output now.

```
<#@ template debug="true" hostSpecific="true" #>
<#@ output extension=".cs" #>
<#@ Assembly Name="System.Core" #>
<#@ Assembly Name="System.Windows.Forms" #>
<#@ import namespace="System" #>

using System;

namespace book
{
 class Program
 {
  static void Hello(string[] args)
  {
     Console.WriteLine("Hello World");
     Console.ReadKey();
  }
 }
}
```

Code Listing 2: Hard-Coded Template

The output is fairly straightforward, if a little uninspired, but we will work on that. Let's now see what the output looks like.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace book
{
 class Program
 {
  static void Hello(string[] args)
  {
     Console.WriteLine("Hello World");
     Console.ReadKey();
  }
 }
}
```

Code Listing 3: Template Output

This showcases the beauty and appeal of T4. Looking at the text in the template, it's easy to see how the output will look.

Now let's turn our attention to syntax and the components of this template.

Anatomy of a Template

Templates are composed of **directives**, **text blocks**, **code blocks**, and **feature blocks**. We will explore each of these in turn.

Directives

Directives are marked with **<#@ #>** delimiters.

The general syntax is:

```
<#@ DirectiveName [AttributeName = "AttributeValue"] ... #>
```

Directives should be at the top of the template file. They cannot be included in a code block or after any feature blocks.

In the template that we have been looking at, the following directives are used:

```
<#@ template debug="true" hostSpecific="true" #>
<#@ output extension=".cs" #>
<#@ Assembly Name="System.Core" #>
<#@ Assembly Name="System.Windows.Forms" #>
<#@ import namespace="System" #>
```

Code Listing 4: Directives for out Template

We will now look at the details for each of these directives.

Template Directive

This is a key directive. Every template will include a template directive, and can only include this directive once. If your template has more than one template directive, you will get a warning message like this:

> *Multiple template directives were found in the template. All but the first one will be ignored. Multiple parameters to the template directive should be specified within one template directive*.

There are no warning or error messages given if you do not have a template directive, but there are some key attributes that you will generally want to set.

Attribute	Description
language	You can specify C# or VB. The default value will be C#. This will be the language that the code blocks and expression blocks are written in, not the language generated by the template.

Attribute	Description
compilerOptions	Any valid compiler option. These options are passed to the compiler when the template is compiled. This is ignored for Run Time template.
culture	Specifies the culture to use to evaluate expression blocks as they are converted to text. This defaults to InvariantCulture.
debug	Specifies whether or not to keep the intermediate files created to compile the template. This is useful to get better error messages and to navigate to the source of the problem if there are problems compiling the template.
hostspecific	Can be true, false, or trueFromBase. The default value is false. If set to true, the base class for the template will include a property Host that will give you access to the hosting engine. This attribute is not applicable for Run Time Templates.
inherits	For text templates, this is the base class for the intermediate class used to create the template when the Custom Tool is run. This can be any class derived from Microsoft.VisualStudio.TextTemplating.TextTransformation. We will discuss the implications of this attribute in the context of run time templates in Chapter 3.
visibility	This attribute is applicable only with run time templates. This specifies the visibility for the class generated to create the run time template. The default value is public. You can also set it to internal to limit who can use the template.
linePragmas	This attribute can be true or false, and will determine whether or not linePragma directives are included in the generated code used to create the template. This allows you to get better error messages if there are problems compiling the template.

Output Directive

This directive is applicable only for text templates, and will control the file extension and encoding for the generated file.

File Extension	Description
extension	Specifies the extension for the generated file. The default value is **".cs"**. This does not influence the language for the generated file, just the extension for the file name. It is ignored for run time templates in that it does not change the file extension of the related nested file, but will not cause any errors if it is provided anyway.
encoding	Specifies the encoding for the generated file. The default value is `0` for system default. Valid values can be any of the `WebName` or `CodePage` values from calling `Encoding.GetEncodings()`.

Assembly Directive

This directive has a similar effect to adding a reference to a Visual Studio project. There is only one attribute: the name. You can specify the full name as recorded in the GAC, the fully qualified assembly name, or the absolute path to the assembly.

You can also reference Visual Studio variables with syntax like this:

```
<#@ assembly name="$(SolutionDir)\BaseTemplate\bin\debug\BaseTemplate.dll" #>
```

Or

```
<#@ assembly name="%LibraryDirectory%\Library\%Version%\BaseTemplate.dll" #>
```

Import Directive

This directive is similar to a **`using`** directive in C# or an **`Imports`** directive in VB. Any namespace that you specify must be found in one of the assemblies mentioned in the Assembly directives. This directive has a single attribute—the name of the namespace to import.

Include Directive

This directive is similar to the **`#include`** directive from classic ASP. The contents of another file are inserted into the template being worked on.

file	The file path can be a relative or an absolute path to the file to be included. The path can also include any of the variables discussed earlier for the assembly directive.
once	Can be set to `true` to ensure that the specified file is included only once

Text Blocks

Text blocks are created from straight text that is entered in a template and copied directly to the generated file. This allows us to write template code that looks like the output that will be produced.

The text block for our original template is marked in the dotted border in the following code:

```
<#@ template debug="true" hostSpecific="true" #>
<#@ output extension=".cs" #>
<#@ Assembly Name="System.Core" #>
<#@ Assembly Name="System.Windows.Forms" #>
<#@ import namespace="System" #>
<#@ import namespace="System.IO" #>
<#@ import namespace="System.Diagnostics" #>
<#@ import namespace="System.Linq" #>
<#@ import namespace="System.Collections" #>
<#@ import namespace="System.Collections.Generic" #>
-------------------------------------------------------------------------------
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace book
{
 class Program
 {
  static void Hello(string[] args)
  {
     Console.WriteLine("Hello World");
     Console.ReadKey();
  }
 }
}
-------------------------------------------------------------------------------
```

Code Listing 5: Template Highlighting the Text Block

Expression Blocks

Expression blocks are inserted in text blocks. With expression blocks, you can inject values from metadata or the results of evaluating an expression into the generated code. Before this value is injected into the generated code, the template engine will run the value through the **`ToString`** function, taking into account the culture specified in the **`Template`** attribute.

The syntax for an expression looks familiar to anyone who has done ASP programming.

<#= DateTime.Today #>

This expression will insert today's date into the generated code. Any valid expression can be used in the expression block, but not a statement. Behind the scenes, the expression will get wrapped in a call to **this.Write()** in the generated class used to create the template.

Let's add an expression to the original template and see how that changes the generated code.

```
<#@ template debug="true" hostSpecific="true" #>
<#@ output extension=".cs" #>
<#@ Assembly Name="System.Core" #>
<#@ Assembly Name="System.Windows.Forms" #>
<#@ import namespace="System" #>
<#@ import namespace="System.IO" #>
<#@ import namespace="System.Diagnostics" #>
<#@ import namespace="System.Linq" #>
<#@ import namespace="System.Collections" #>
<#@ import namespace="System.Collections.Generic" #>

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace book
{
 // Generated on <#= DateTime.Today #>
 class Program
 {
  static void Hello(string[] args)
  {
     Console.WriteLine("Hello World");
     Console.ReadKey();
  }
 }
}
```

Code Listing 6: Simple Template with an Expression Block

The output will now look like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace book
```

```
{
 // Generated on 04/23/2015 00:00:00
 class Program
 {
  static void Hello(string[] args)
  {
     Console.WriteLine("Hello World");
     Console.ReadKey();
  }
 }
}
```

Code Listing 7: Output from the Expression Block

If we change the culture, we can see the impact of this attribute to the template directive:

Let's change the template directive like this:

```
<#@ template debug="true" hostSpecific="true" culture="en-GB"#>
```

The output now looks like the following, with the British format for the date:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace book
{
 // Generated on 23/04/2015 00:00:00
 class Program
 {
  static void Hello(string[] args)
  {
     Console.WriteLine("Hello World");
     Console.ReadKey();
  }
 }
}
```

Code Listing 8: Output Using the Great Britain Culture

Code Blocks

Beyond simple expressions, we may want to execute code statements with more complex logic.

To demonstrate this, let's create a new template that will loop through and list out the valid cultures that could be passed to the **`culture`** attribute of the **`template`** directive.

In our solution, let's add a new template and call it **culture.tt**.

Add the following code to this template:

```
<#@ template debug="false" hostspecific="false" language="C#" #>
<#@ assembly name="System.Core" #>
<#@ import namespace="System.Linq" #>
<#@ import namespace="System.Text" #>
<#@ import namespace="System.Collections.Generic" #>
<#@ import namespace="System.Globalization" #>
<#@ output extension=".txt" #>

<#

      var cultures =
           CultureInfo.GetCultures(CultureTypes.SpecificCultures);
      foreach (var culture in cultures )
      {
#>
      // <#= culture.DisplayName #>
            template debug="false" hostspecific="false" language="C#"
culture="<#=culture.Name #>"

<#
      }
#>
```

Code Listing 9: Simple Template with a Code Block

The code block is delimited by **<#** and **#>**

Inside a code block, we can declare variables, have conditional statements, loop statements, call methods, etc. Anything that can go in the body of a method can go inside a code block.

If we look at **culture.txt**, the output will look like this:

```
// Arabic (Saudi Arabia)
template debug="false" hostspecific="false" language="C#" culture="ar-SA"

// Bulgarian (Bulgaria)
template debug="false" hostspecific="false" language="C#" culture="bg-BG"

// Catalan (Catalan)
template debug="false" hostspecific="false" language="C#" culture="ca-ES"

// Chinese (Traditional, Taiwan)
template debug="false" hostspecific="false" language="C#" culture="zh-TW"

// Czech (Czech Republic)
template debug="false" hostspecific="false" language="C#" culture="cs-CZ"
```

```
// Danish (Denmark)
template debug="false" hostspecific="false" language="C#" culture="da-DK"

. . .

// Chinese (Traditional, Hong Kong S.A.R.)
template debug="false" hostspecific="false" language="C#" culture="zh-HK"

// Chinese (Traditional, Macao S.A.R.)
template debug="false" hostspecific="false" language="C#" culture="zh-MO"

// Chinese (Simplified, Singapore)
template debug="false" hostspecific="false" language="C#" culture="zh-SG"

// Chinese (Traditional, Taiwan)
template debug="false" hostspecific="false" language="C#" culture="zh-TW"

// isiZulu (South Africa)
template debug="false" hostspecific="false" language="C#" culture="zu-ZA"
```

This creates a handy reference for any specific culture you may need. This also showcases that the output of the T4 can be any type of text—not just code.

Feature Blocks

Feature blocks allow us to create new features like methods, properties, and even classes.

There are a couple of rules that we have to follow when dealing with feature blocks:

- A feature block cannot be defined within a code block or a text block.
- Feature blocks must be the last thing in a template file. You cannot define a feature block and then add a new code block or text block.
- Feature blocks can contain text blocks, which can include an expression block.

Feature blocks are delimited by the **<#+ #>** characters.

Let's go back to the **Template1.tt** that we were working on earlier, and add a feature block. Add the following code to **Template1.tt**:

```
<#@ template debug="true" hostSpecific="true" #>
<#@ output extension=".cs" #>
<#@ Assembly Name="System.Core" #>
<#@ Assembly Name="System.Windows.Forms" #>
<#@ import namespace="System" #>
<#@ import namespace="System.IO" #>
<#@ import namespace="System.Diagnostics" #>
<#@ import namespace="System.Linq" #>
```

```
<#@ import namespace="System.Collections" #>
<#@ import namespace="System.Collections.Generic" #>

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace book
{
 class Program
 {
  static void Hello(string[] args)
  {
     Console.WriteLine("Hello World");
     Console.ReadKey();
  }
 }
}

<#+
 private void HelloWorld()
 {
#>
   Hello World at <#= DateTime.Now #>
<#+
 }
#>
```

Code Listing 10: Feature Block added to Template1.tt

Here we have a new feature block that defines the method **`HelloWorld`**. This feature block includes a text block that simply writes out the common phrase “Hello World”. The text block also includes an expression block that will write out the current time.

If we try to include anything after the closing delimiter for the feature block, we will get the following error message:

> *This is a helpful error message.*

A better message would be:

> *A Template containing a class feature must contain only class features once the class feature is defined.*

Take a look at the following code:

```
<#+
 private void HelloWorld()
 {
#>
  Hello World at <#= DateTime.Now #>
<#+
 }
#>

Hello world


<#+
 private void GoodBye()
 {
#>
  Bye Bye World at <#= DateTime.Now #>
<#+
 }
#>
```

Code Listing 11: Feature Block with an Error

If your template looks like this at the bottom, you will get a slew of error messages that won't make any sense. To understand what is going on, we need to look at the intermediary class that is created to support the template. As long as you have **debug** set to **true** in your template directive, Visual Studio will keep the intermediary files for you; you just have to find them.

The intermediary file will be located in the **%TEMP%** directory. By default, this will be in your **AppData/Local/Temp** directory. Navigate to this directory and sort the contents by **Date Modified**. Look for the most recently modified **.cs** file. This will be the class that the Custom Tool created to implement your template.

Open this file, and we will take a peek under the hood at what is happening.

Text Templates behind the Scenes

When you open the file, it will look like this:

```
namespace
Microsoft.VisualStudio.TextTemplating9A2BB878B08A3E9F2B95519515D1CC499C63E7
923662D35FE2106B665A51DC80F16C18CAEA399F9CBFB9C93D4CF2FD8B62D07A63C69BD85BB
9D2458BB8A5FF3A
{
 using System;
 using System.IO;
 using System.Diagnostics;
```

```
 using System.Linq;
 using System.Collections;
 using System.Collections.Generic;

 /// <summary>
 /// Class to produce the template output
 /// </summary>

 #line 1 "E:\t4\T4\project\book\book\Template1.tt"
 public class GeneratedTextTransformation :
Microsoft.VisualStudio.TextTemplating.TextTransformation
 {
#line hidden
  /// <summary>
  /// Create the template output
  /// </summary>
  public override string TransformText()
  {
  try
  {
   this.Write(@"


using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace book
{
 class Program
 {
  static void Hello(string[] args)
  {
  Console.WriteLine(""Hello World"");
  Console.ReadKey();
  }
 }
}




");
  }
  catch (System.Exception e)
  {
   e.Data["TextTemplatingProgress"] =
this.GenerationEnvironment.ToString();
```

```
    throw new System.Exception("Template runtime error", e);
   }
   return this.GenerationEnvironment.ToString();
   }
   private
global::Microsoft.VisualStudio.TextTemplating.ITextTemplatingEngineHost
hostValue;
   /// <summary>
   /// The current host for the text templating engine
   /// </summary>
   public virtual
global::Microsoft.VisualStudio.TextTemplating.ITextTemplatingEngineHost
Host
   {
   get
   {
    return this.hostValue;
   }
   set
   {
    this.hostValue = value;
   }
   }

   #line 34 "E:\t4\T4\project\book\book\Template1.tt"

  private void HelloWorld()
  {


   #line default
   #line hidden

   #line 37 "E:\t4\T4\project\book\book\Template1.tt"
this.Write("  Hello World at ");


   #line default
   #line hidden

   #line 38 "E:\t4\T4\project\book\book\Template1.tt"
this.Write(Microsoft.VisualStudio.TextTemplating.ToStringHelper.ToStringWit
hCulture(DateTime.Now));


   #line default
   #line hidden

   #line 38 "E:\t4\T4\project\book\book\Template1.tt"
this.Write("\r\n");
```

```
  #line default
  #line hidden

  #line 39 "E:\t4\T4\project\book\book\Template1.tt"

 }


  #line default
  #line hidden

  #line 41 "E:\t4\T4\project\book\book\Template1.tt"
this.Write("\r\nHello world\r\n\r\n\r\n");


  #line default
  #line hidden

  #line 46 "E:\t4\T4\project\book\book\Template1.tt"

 private void GoodBye()
 {


  #line default
  #line hidden

  #line 49 "E:\t4\T4\project\book\book\Template1.tt"
this.Write("  Bye Bye World at ");


  #line default
  #line hidden

  #line 50 "E:\t4\T4\project\book\book\Template1.tt"
this.Write(Microsoft.VisualStudio.TextTemplating.ToStringHelper.ToStringWit
hCulture(DateTime.Now));


  #line default
  #line hidden

  #line 50 "E:\t4\T4\project\book\book\Template1.tt"
this.Write("\r\n");


  #line default
  #line hidden
```

```
  #line 51 "E:\t4\T4\project\book\book\Template1.tt"

 }


  #line default
  #line hidden
 }

 #line default
 #line hidden
}
```

Code Listing 12: Intermediary Class Created by T4 to Implement the Template

Some of this should look very familiar based on the template that we have been working on, but some of it will look strange. Don't worry about the scary-looking namespace. It will change every time, and is only needed for T4 internally.

Behind the scenes, T4 has created a new class called **`GeneratedTextTransformation`** that is derived from **`Microsoft.VisualStudio.TextTemplating.TextTransformation`**.

Note that this generated class overrides the **`TransformText`** method. More importantly, the implementation of this method outputs all of the text blocks of our template and executes the code in any code blocks as part of the **`TransformText`** method.

The feature blocks are treated differently; once we get a feature block, we are no longer working in the context of the **`TransformText`** method. Instead, we are now working in the context of the methods defined in the feature block. This explains the problem reported in the error messages with compiling the template.

The "*Invalid token 'this' in class, struct, or interface member declaration*" error message refers to line 113 of the generated intermediary file.

Text blocks get converted to calls to the **`Write`** method, which makes sense in the context of the **`TransformText`** method, or the context of a method being defined in a feature block, but does not make sense outside the context of a method. Now the error message makes sense.

This peek behind the scenes shows us a couple of guiding principles that are helpful to keep in mind when writing templates:

- Text blocks are converted to calls to the **`Write`** method from the **`TextTransformation`** class.
- Expression blocks are converted to calls to the **`ToStringWithCulture`** method of the **`ToStringHelper`** class.
- Code blocks are evaluated directly in whatever method they are defined in. If a feature block has not been defined, they are evaluated in the override of the **`TransformText`** method.

Now that we know our templates are actually defining a new class derived from the **`TextTransformation`** class, let's turn our attention to what utility functions this class provides.

Utility Methods

In looking at the intermediary class created to implement our template, we have seen a couple of these utility methods as they were used in the generated class. Let's explore these methods a bit further.

Output Functions	
Write	Writes the specified text out to the generated text output. There is an overload of this method that behaves like string.Format for handling more complex string concatenation. This is the only overload. Every parameter you pass to this method must already be a string. There are no other overrides to handle other data types.
WriteLine	WriteLine behaves just like Write, except that it adds a carriage return and line feed to the end of the output.

Error-Reporting Functions	
Warning	Adds a new warning to an internal list of errors that will be displayed in the output window. This does not affect execution of the template, but simply logs a warning message.
Error	Works just like the Warning method, except the message that you pass in will be flagged as an error instead of a warning in the Error List window. This also does not affect execution of the template; it simply logs an error message. If you want to halt the execution of the template, you will need to add a return statement to exit out of the method you are in. Usually this will be the overridden TransfromText method.

Formatting Functions	
ClearIndents	Resets the CurrentIndent property to blank.

Formatting Functions	
PushIndent	Appends the specified value to the CurrentIndent property. In general this should be a string with the number of spaces to indent
PopIndent	Removes the most recently added text for the CurrentIndent property

Output Functions

Depending on what you are trying to output, it may be easier to simply use **`Write`** and **`WriteLine`** in a code block instead of creating a new text block.

Let's go back and look at the **`HelloWorld`** and **`GoodByeWorld`** methods that we added to **Template1** when we were discussing feature blocks.

These methods could be written to look like this:

```
<#+
 private void HelloWorld()
 {
     this.WriteLine(" Hello World at {0}",
     ToStringHelper.ToStringWithCulture(DateTime.Now ));
 }

 private void GoodBye()
 {
     this.WriteLine ("Bye Bye World at {0}",
     ToStringHelper.ToStringWithCulture(DateTime.Now ));
 }
#>
```

Code Listing 13: Feature Methods Written using WriteLine

Depending on what you are trying to output, this may be much easier to read than adding the text blocks and expression blocks that we originally had.

Error Reporting

You will often encounter conditions that you want to warn the developer using the template about, or error conditions that you need to complain about. This is where the **`Error`** and **`Warning`** functions come in handy.

To showcase this, let's go back to the **culture.tt** template that we created earlier. Change the template code block to look like this:

```
<#

      var cultures =
CultureInfo.GetCultures(CultureTypes.SpecificCultures);
      Warning("There were " + cultures.Count() + " cultures returned");
      foreach (var culture in cultures )
      {
#>
// <#= culture.DisplayName #>
template debug="false" hostspecific="false" language="C#"
culture="<#=culture.Name #>"

<#
      }
#>
```

Code Listing 14: Code Blocking Showing the Warning Method

When you run this template, you will get a warning that there were 524 cultures returned.

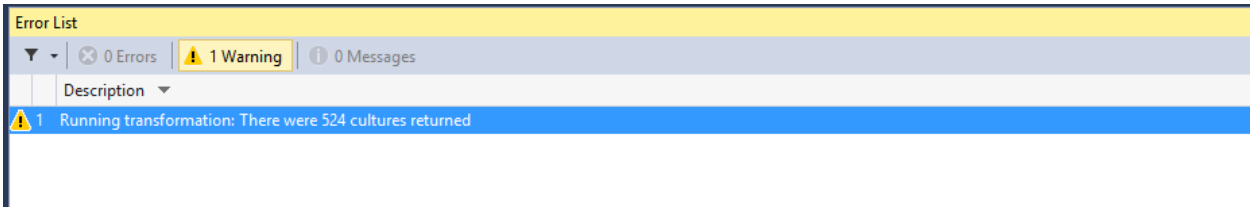

Figure 3: Output from the Call to the Warning Function

Formatting Functions

We can also take advantage of the output helpers to make this template more readable.

```
<#

      var cultures = CultureInfo.GetCultures
            (CultureTypes.SpecificCultures);
      Warning("There were " + cultures.Count() + " cultures returned");
      foreach (var culture in cultures )
      {
          WriteLine ("// " + culture.DisplayName);
          WriteLine (" template debug=\"false\" hostspecific=\"false\"
language=\"C#\" culture=\"{0}\" ",  culture.Name );
      }
#>
```

Code Listing 15: Culture Template Rewritten to use Helper Functions to Improve Formatting

A Simple Example

Now let's put all the pieces we have learned together into something useful.

Let's load the config file and find the **`appSettings`**. We will then create a class that exposes a property for each **`appSetting`** that we find.

Start by adding a new text template and naming it **appSettings.tt**.

First we have a couple of non-T4 related tasks to do. Let's start by finding the config file. Depending on the type of project you use this template in, it could be an **app.config** or a **web.config**. The first thing we need to do is find the template and then look in the root folder for the containing project for both an **app.Config** and a **web.Config**. If we don't find one, we want to log an error.

Once we find the config file, we will load it as an **`XmlDocument`** and parse it to find the elements under **`appSettings`**. We will then output a read only property for each element found that will return the current value from the config file.

To find the config file, we will use the **`EnvDte`** object. "Dte" stands for Developer Tool Extensibility. We will touch on this object briefly here, and explore it more thoroughly in Chapter 8.

Let's start with the **`FindConfigFile`** method. Add the following code to the bottom of the **appSettings.tt** file.

```
<#+
public string FindConfigFile ()
{
 var visualStudio = (this.Host as
 IServiceProvider).GetService(typeof(EnvDTE.DTE))
 as EnvDTE.DTE;
 var project =
 visualStudio.Solution.FindProjectItem
 (this.Host.TemplateFile).ContainingProject
   as EnvDTE.Project;
 foreach (EnvDTE.ProjectItem item in project.ProjectItems)
 {
    if (item.FileNames[0].EndsWith(".config"))
    {
        return item.FileNames[0];
    }
  }
  return "";
}

#>
```

Code Listing 16: FindConfigFile Method

This method will return the full path to either the **web.config** or the **app.config**, or an empty string if it cycles through every item in the project without finding a config file.

Now that we can get to the config file, we are ready to parse. First off, we will log an error if we don't find the config file, and then we are ready to parse the file that we did find.

Add the following code to the template above the feature block.

```
<#@ template debug="true" hostspecific="true" language="C#" #>
<#@ assembly name="System.Core" #>
<#@ assembly name="EnvDte" #>
<#@ assembly name="System.Xml"#>
<#@ output extension=".cs" #>
<#@ import namespace="System.Xml" #>
using System;


namespace book
{
 public class AppSettings
 {
<#
 PushIndent (" ");
 var xml = new XmlDocument();
 var configFile = FindConfigFile();
 if (string.IsNullOrEmpty(configFile))
 {
  Error("No config file found");
  return "";
 }
 xml.Load(configFile);
 var element = xml.SelectSingleNode
  ("/configuration/appSettings");
 for (int i=0; i<element.ChildNodes.Count; i++)
 {var key = element.ChildNodes[i].Attributes["key"].Value;
   var value = this.CurrentIndent
      + element.ChildNodes[i].Attributes["value"].Value.Trim();
   WriteLine(this.CurrentIndent + "public string "
      + key + "{get { return \""
      + value +"\";}}");
 }
 PopIndent();
#>
 }
}
```

Code Listing 17: appSettings.tt

There are a couple of key items to note in this code. This example showcases how to use the **`CurrentIndent`** and **`PushIndent`** with the **`PopIndent`** to format the generated code nicely.

We also see a great example of using the **`Error`** method to log an error, as well as an explicit return from the **`TransformText`** method to halt processing.

In your project, add an **app.config** with the following configuration settings:

```
 <appSettings>
 <add key="PreserveLoginUrl" value="true" />
 <add key="ClientValidationEnabled" value="true" />
 <add key="UnobtrusiveJavaScriptEnabled" value="true" />
 </appSettings>
```

Code Listing 18: Sample AppSettings

This sample **`appSettings`** will generate a class that looks like this:

```
using System;


namespace book
{
 public class AppSettings
 {
  public string PreserveLoginUrl{get { return " true";}}
  public string ClientValidationEnabled{get { return " true";}}
  public string UnobtrusiveJavaScriptEnabled{get { return " true";}}
 }
}
```

Code Listing 19: Output from the appSettings.tt Template

Summary

We've covered a lot of material in this chapter—we've covered the main components that go into making a text template, and explored all of the directives and attributes for these directives. This section will make a handy quick reference for the rest of this book.

We also explored the various sections of a template, paying close attention to what goes into text blocks, expression blocks, code blocks, and feature blocks. We explored the different roles that each of these blocks play, and how they relate to each other.

Especially when dealing with feature blocks, we explored what can go wrong when compiling the template. This prompted our peek behind the scenes to explore the intermediary class that T4 generates to implement the template.

Seeing this intermediary class gave us our first peek at utility functions provided by the base class for the template. We looked at utility functions to help create the output, report errors, and format the output.

Finally, we built our first useful template to generate a class that will expose a read-only property for every configuration setting defined in the **`appSettings`** of the configuration file. This also gave us our first peek at the **`EnvDte`** object that we will explore further in Chapter 4.

Now that we have explored text templates, let's turn our attention to how text templates are different from run time templates.

Chapter 3 Run Time Templates

Most of what we learned in Chapter 2 about text templates will still hold true for run time templates, but there a couple of big differences.

The first big difference is the custom tool used. For text templates, the Custom Tool is **`TextTemplatingFileGenerator`**. For run time templates, the Custom Tool is **`TextTemplatingFilePreProcessor`**.

The second big difference is the output from the template. With text templates, the output is the result of running the template. Behind the scenes, we saw that T4 creates an intermediary class and then compiles this class. The output we get comes from instantiating this class and calling the **`TransformText`** method. For text templates, T4 does all of this for us as a single step, and we can see the output in the nested related file.

For run time templates, our job is a bit more involved, while T4 gets off a bit easier. The output of running the Custom Tool on a run time template is the intermediary class. We are expected to instantiate this class at runtime and explicitly call the **`TransformText`** method directly to get the results. This gives us a great deal more control over calling the template.

Let's start by creating a new template that we will call **HelpMessage.tt**. Make sure to select **Run Time Text Template** from the **Add New Item** dialog box:

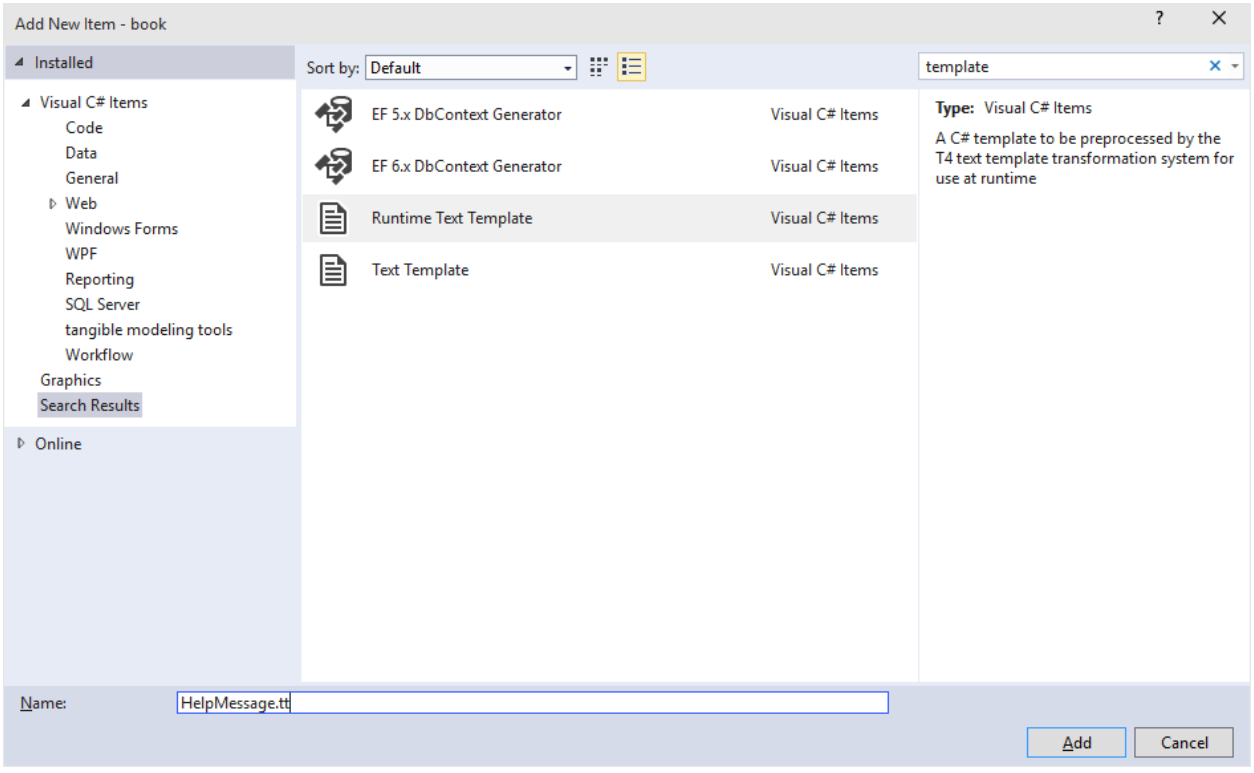

Figure 4: Adding a New Run Time Template

Also confirm, by looking at the properties for this item in Solution Explorer, that the **Custom Tool** is properly set as **`TextTemplatingFilePreprocessor`**:

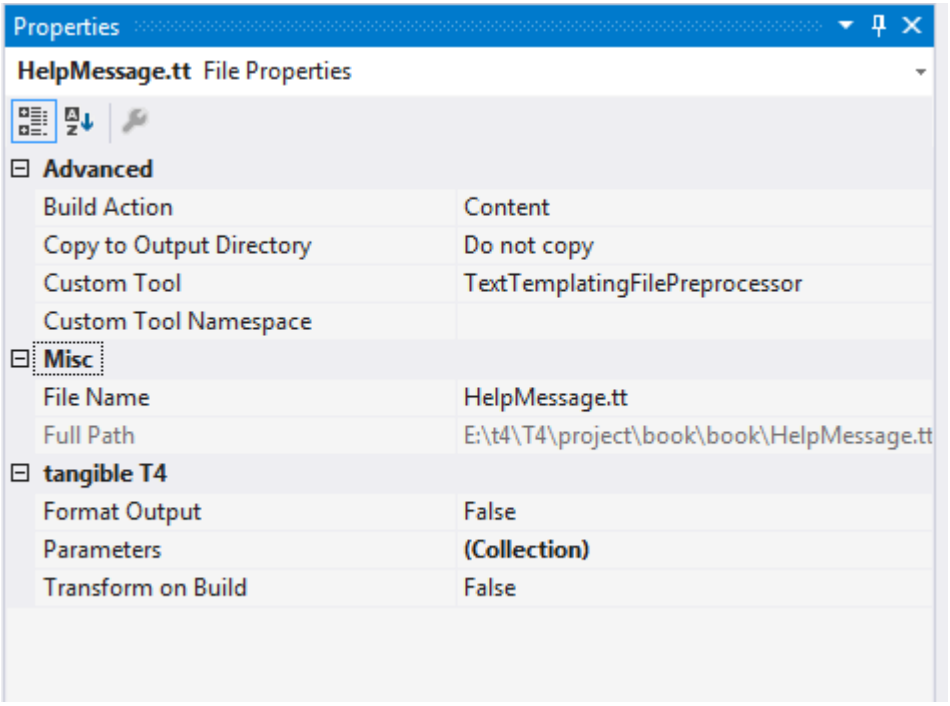

Figure 5: Properties on a Run Time Template

Add the following text to the **HelpMessage.tt** template.

```
<#@ template language="C#" #>
<#@ assembly name="System.Core" #>
<#@ assembly name="System.Reflection" #>
<#@ import namespace="System.Linq" #>
<#@ import namespace="System.Text" #>
<#@ import namespace="System.Collections.Generic" #>
<#@ import namespace="System.Reflection" #>

Message from SENDER to RECEIVER on <#= DateTime.Now #>

Need help with <#= Assembly.GetExecutingAssembly().GetName ().Name #>
```

Code Listing 20: Initial HelpMessage.tt

This simple template shows the directives that we are already familiar with, as well a text block and a couple of expression blocks. We will see in future templates that code blocks and feature blocks also continue to work as expected.

Now let's look at the output from running the Custom Tool:

```
// ------------------------------------------------------------
// <auto-generated>
//  This code was generated by a tool.
//  Runtime Version: 12.0.0.0
//
// Changes to this file may cause incorrect behavior and will be lost if
//  the code is regenerated.
// </auto-generated>
// ------------------------------------------------------------
namespace book
{
 using System.Linq;
 using System.Text;
 using System.Collections.Generic;
 using System.Reflection;
 using System;

 /// <summary>
 /// Class to produce the template output
 /// </summary>

 #line 1 "E:\t4\T4\project\book\book\HelpMessage.tt"

[global::System.CodeDom.Compiler.GeneratedCodeAttribute("Microsoft.VisualSt
udio.TextTemplating", "12.0.0.0")]
 public partial class HelpMessage : HelpMessageBase
 {
#line hidden
  /// <summary>
  /// Create the template output
  /// </summary>
  public virtual string TransformText()
  {
  this.Write("\r\n\r\nMessage from SENDER to RECEIVER on ");

  #line 10 "E:\t4\T4\project\book\book\HelpMessage.tt"
  this.Write(this.ToStringHelper.ToStringWithCulture(DateTime.Now));

  #line default
  #line hidden
  this.Write("\r\n\r\nNeed help with ");

  #line 12 "E:\t4\T4\project\book\book\HelpMessage.tt"

this.Write(this.ToStringHelper.ToStringWithCulture(Assembly.GetExecutingAss
embly().GetName ().Name));

  #line default
  #line hidden
```

```
  this.Write("\r\n\r\n");
  return this.GenerationEnvironment.ToString();
  }
 }
```

Code Listing 21: Output from HelpMessage.tt

This should look familiar from the intermediary file that we saw in Chapter 2. So now the question is, how do we get the output of the template? The answer may already be obvious from the name of the template type: we get this output at run time.

When we created the Console Application project at the beginning of Chapter 2, we also got a **`Program`** class that we have so far ignored. Now we need to add the following code to **Program.cs**:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace book
{
 class Program
 {
  static void Main(string[] args)
  {
     var template = new HelpMessage();
     Console.WriteLine(template.TransformText());
     Console.ReadLine();
  }
 }
}
```

Code Listing 22: Running the Run Time Template

This will create an instance of our template, execute the **`TransformText`** method, and simply display the output to the screen. So far, so good.

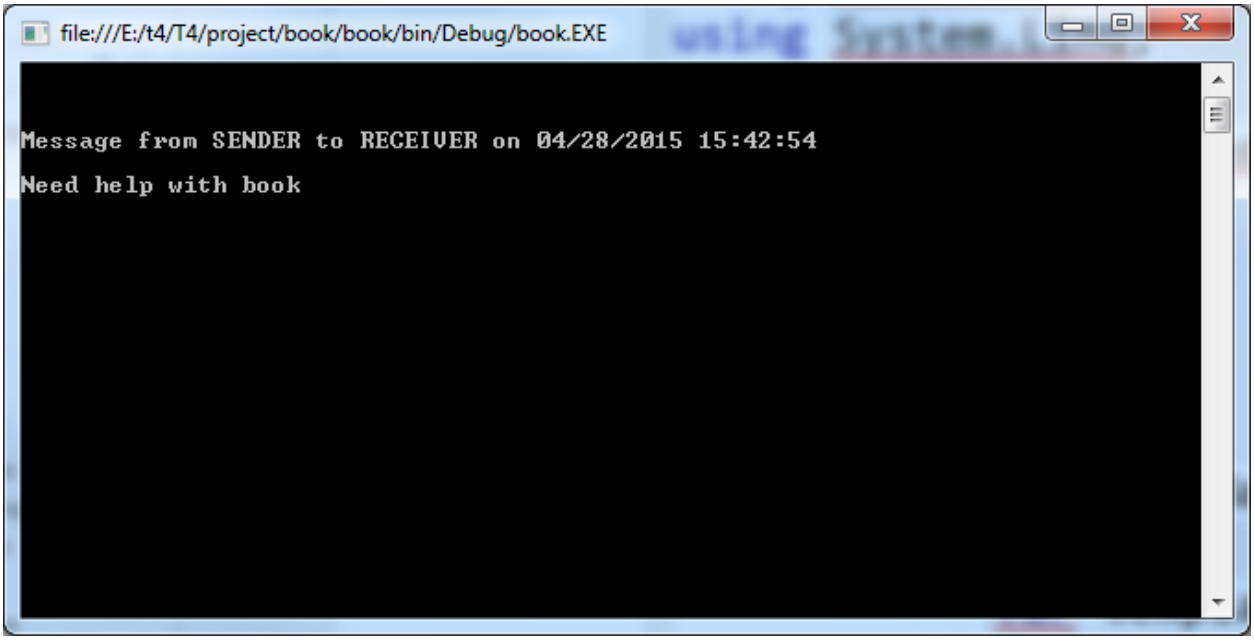

Figure 6: Running the Run Time Template

Now we are in control of setting up the environment for running the template. This opens up a lot of possibilities, and a key question. Most importantly, how can we pass data to this template?

Directives

The directives that we saw in Chapter 2 continue to work as expected, but now we have a new directive that is applicable only to run time templates.

Parameters

The **`Parameter`** directive was created specifically for passing data to the template when it is called.

Parameter	Description
type	The data type of the parameter passed in. This should be the full type name to the data type. If necessary, you may need to Assembly or Import directives so that the template understands the type you specify.
name	The name of the parameter passed in. All of the identifier naming rules for the language being used apply to the parameter name.

For our sample template, it would be nice to be able to pass in a couple of parameters to specify the sender and receiver. So let's add two parameter directives.

```
<#@ template language="C#" #>
<#@ assembly name="System.Core" #>
<#@ assembly name="System.Reflection" #>
<#@ import namespace="System.Linq" #>
<#@ import namespace="System.Text" #>
<#@ import namespace="System.Collections.Generic" #>
<#@ import namespace="System.Reflection" #>
<#@ parameter type="System.String" name="sender" #>
<#@ parameter type="System.String" name="receiver" #>

Message from <#= sender#> to <#= receiver #> on <#= DateTime.Now #>

Need help with <#= Assembly.GetExecutingAssembly().GetName ().Name #>
```

Code Listing 23: HelpMessage.tt with Two Parameter Directives

We can now access these parameters with simple expression blocks as if they were local variables. There are a couple of key items to note in the generated class that implements our template. The parameters that we specified are in fact now local properties, and we have a new **`Initialize`** method.

Let's look at the relevant sections from the generated file.

```
#line 1 "E:\t4\T4\project\book\book\HelpMessage.tt"

private string _senderField;

/// <summary>
/// Access the sender parameter of the template.
/// </summary>
private string sender
{
 get
 {
  return this._senderField;
 }
}

private string _receiverField;

/// <summary>
/// Access the receiver parameter of the template.
/// </summary>
private string receiver
{
 get
 {
  return this._receiverField;
 }
}
```

```
/// <summary>
/// Initialize the template
/// </summary>
public virtual void Initialize()
{
 if ((this.Errors.HasErrors == false))
 {
  bool senderValueAcquired = false;
  if (this.Session.ContainsKey("sender"))
  {
  this._senderField = ((string)(this.Session["sender"]));
  senderValueAcquired = true;
  }
  if ((senderValueAcquired == false))
  {
  object data = global::System.Runtime.Remoting.Messaging
    .CallContext.LogicalGetData("sender");
  if ((data != null))
  {
   this._senderField = ((string)(data));
  }
  }
  bool receiverValueAcquired = false;
  if (this.Session.ContainsKey("receiver"))
  {
  this._receiverField = ((string)(this.Session["receiver"]));
  receiverValueAcquired = true;
  }
  if ((receiverValueAcquired == false))
  {
  object data = global::System.Runtime.Remoting.Messaging
    .CallContext.LogicalGetData("receiver");
  if ((data != null))
  {
   this._receiverField = ((string)(data));
  }
  }
 }
}
```

Code Listing 24: Implementation for HelpMessage.tt with Parameter Directives

As you can see here, the **`Initialize`** method explicitly sets these local properties based on data from **`Session`** or **`CallingContext`**.

Let's run the template.

Unfortunately, we now get an **`ArgumentNullException`** because our parameters have not been initialized.

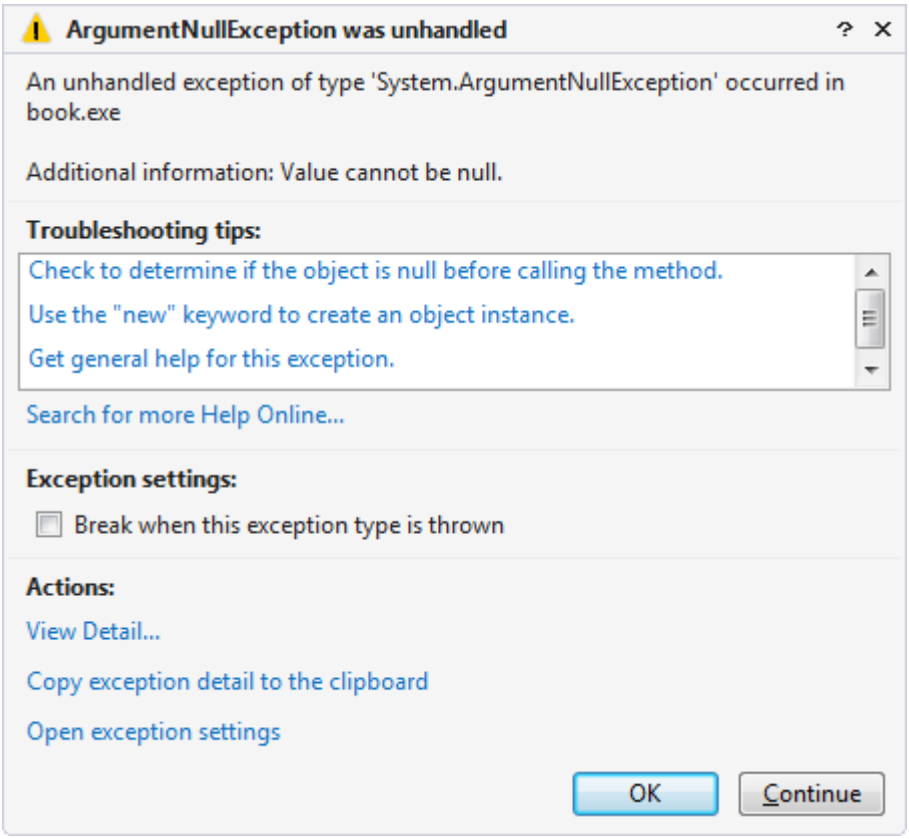

Figure 7: Error Message From Running the Template Without Initializing the Parameters

So we need to change the template to be a bit more resilient.

Add the following code just above the text block in the template:

```
<#
 if ((string.IsNullOrEmpty(sender))
  || (string.IsNullOrEmpty(receiver)))
 {
  return "Could not Transform text because parameters were missing";
 }
#>
```

Code Listing 25: Making the Template More Resilient

Now when we run the template, we still won't get the expected output, but at least we will get a meaningful message without throwing an exception.

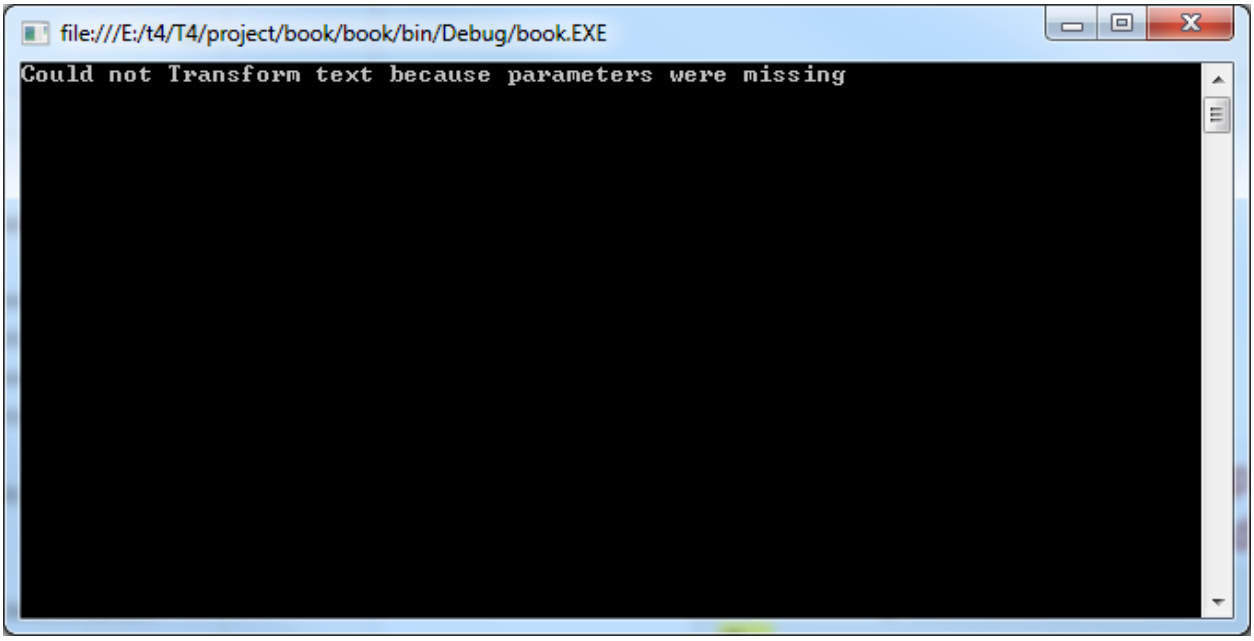

Figure 8: Helpful Error Message After Running the More Resilient Version of the Template

So we need to rework the code used to call the template to create a session with valid values for our parameters.

Add code to **Program.cs** to look like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace book
{
 class Program
 {
  static void Main(string[] args)
  {
     var template = new HelpMessage();
     template.Session = new Dictionary<string, object>();
     template.Session.Add("sender", Environment.UserName);
     template.Session.Add("receiver", "Help Desk");
     template.Initialize();
     Console.WriteLine(template.TransformText());
     Console.ReadLine();
  }
 }
}
```

Code Listing 26: Properly Calling the HelpMessage Template

As you can see, the **Session** is simply a dictionary that we can explicitly set, and then we must call the **Initialize** method so that the **Session** gets evaluated to initialize the local properties.

Now when we run the template, we get the expected results:

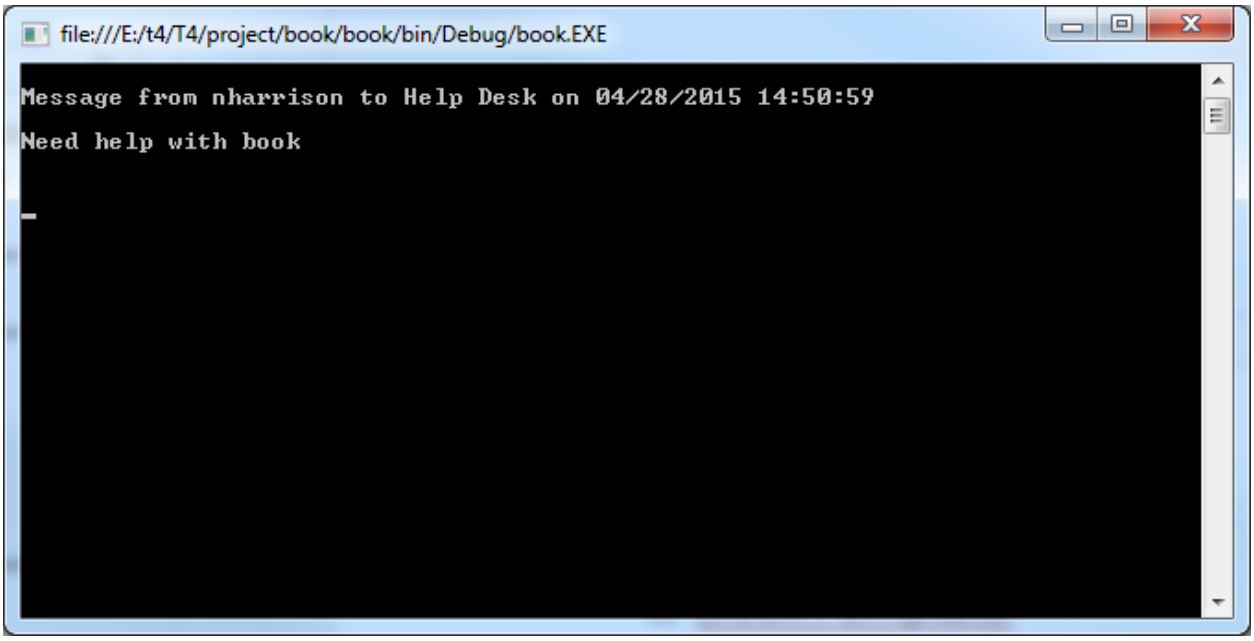

Figure 9: Output From Properly Calling the Help Message Template

Changing the Base Class

So far we have glossed over the base class for the class that gets generated as part of the run time template. If we do not specify an **inherits** attribute in the template directive, T4 will generate a base class for us that has everything we need so that there won't be a dependency on T4 when we distribute code with a run time template. I won't list the code for this generated base class here because it is long and the details for the implementation are not relevant to our current discussion, but we may still have uses for it.

Add a new class to the project and call it **HelpMessageBase**. Copy the entire base class region from the **HelpMessage1.cs** to this new class.

Now we have a couple of tweaks to make to this class. Wrap it in a namespace, and add a couple of virtual methods.

```
  public virtual string TransformText() { return ""; }
  public virtual void Initialize(){ }
```

Code Listing 27: Changes Needed to the Base Class

Now we can set the **inherits** attribute on the template directive:

```
<#@ template language="C#" inherits="book.HelpMessageBase"#>
```

When we run the Custom Tool, the generated output will no longer include a generated base class.

If we run the program now, we will get the same output as before.

At this point we have control over the base class, and can manipulate it as needed, since it is no longer a generated artifact.

There are a couple of changes that we would like to make. Start by removing the two parameter directives. Instead, we will define these as regular properties in the new base class.

Add the following code to the top of **HelpMessageBase.cs**:

```
  public string sender { get; set; }
  public string receiver { get; set; }
```

Now we can go back to the **program.cs** and modify it to use these new properties.

```
  static void Main(string[] args)
  {
     var template = new HelpMessage();

     template.sender = Environment.UserName;
     template.receiver = "Help Desk";
     Console.WriteLine(template.TransformText());
     Console.ReadLine();
  }
```

Code Listing 28: Calling the HelpMessage Template with the New Base Class

When we run the program, we get the same results without having to mess with **`Session`** or calling the **`Initialize`** method.

Note: You could also extend the constructor to accept the values as parameters instead of explicitly setting the properties. This would have the advantage of ensuring that the template could not be created without setting these critical values.

Depending on your specific requirements, this may be a much more useful way to pass data to the template.

Summary

In this chapter we explored run time templates. As we have seen, these are different from text templates in that the output is a new class that we need to call to get the output we expect from the template.

We explored a couple of different scenarios for passing data to these templates through **`parameter`** directives, and by taking control of the base class and adding our own properties.

Taking control of the base class opens up a lot of possibilities for extending reusability, which we will cover in greater detail in Chapter 4.

Chapter 4 Working with the Host

Overview of the DTE

The DTE is the object that Visual Studio provides to automate many tasks in Visual Studio. You may hear various stories for what DTE stands for. Some examples include:

- Design Time Extensibility
- Developer Tool Extensibility
- Development Tools Environment

Regardless of the name used, this object gives you access to programmatically control most aspects of Visual Studio. More important to our discussions here, this is also the Host for templates that are run from within Visual Studio.

To access this object, we start with the **`template`** directive.

```
<#@ template debug="true" hostSpecific="true" #>
```

Setting **`hostSpecific`** to true will ensure that the base class for the template includes a property called **`host`**. Now that we have the **`host`** property, we will also need the assemblies and references to use the DTE.

```
<#@ assembly name="EnvDTE" #>
<#@ assembly name="EnvDTE100" #>
<#@ import namespace="EnvDTE" #>
<#@ import namespace="EnvDTE100" #>
```

Code Listing 29: References Needed to Work with the DTE

With this in place, we access this DTE through this **`host`** property. The common pattern is:

```
IServiceProvider serviceProvider = (IServiceProvider)host ;
VisualStudio = serviceProvider.GetService(typeof(EnvDTE.DTE)) as DTE;
```

Code Listing 30: Accessing the DTE Through the Host

Now that we have the DTE object, let's see we can do with it.

Throughout this chapter, we will build up methods in a new class we will call **`DTEHelper`**.

In our solution, create a new Class Library project and name it **T4Utilities**. We will create our new **`DTEHelper`** class in this project. First, we will need to install the Visual Studio SDK. You can get the latest version from the Microsoft Download Center.

There are a couple of references we need to add to this T4Utilities project that will now be available:

- EnvDTE
- Microsoft.VisualStudio.TextTemplating.Interfaces.10.0

Because we depend on the host to get to the DTE, let's provide a constructor requiring that any user gives us the host.

```
private DTE VisualStudio { get; set; }

public DTEHelper(ITextTemplatingEngineHost host)
{
   var serviceProvider = (IServiceProvider)host ;
   VisualStudio = serviceProvider
      .GetService(typeof(EnvDTE.DTE)) as DTE;
}
```

Code Listing 31: Constructor for DTEHelper

Navigating the Solution

The DTE provides access to all of the projects in the solution. This is a little bit complicated because the DTE is actually a C++ COM object. So navigating through a collection is not nearly as straightforward as we are used to. Let's create a method to give us access to the list of projects in a format we can more easily deal with.

```
public IList<Project> GetAllProjects()
{
   var returnValue = new List<Project>();
   foreach (Project project in VisualStudio.Solution.Projects)
   {
      returnValue.Add(project);
   }
   return returnValue;
}
```

Code Listing 32: Getting a List of all the Projects in a Solution

Now that we have this in an **`IList`**, we can easily filter by any of the properties exposed by **`Project`**. Filtering by the **`Kind`** property, we can get a list of only the C# projects.

```
public IList<Project> GetCSharpProjects()
{
   return GetAllProjects().Where
         (p => p.Kind ==
            "{66A26720-8FB5-11D2-AA7E-00C04F688DDE}").ToList();
```

```
    }
```

Code Listing 33: Getting a List of CSharp Projects

"{66A26720-8FB5-11D2-AA7E-00C04F688DDE}" is the **Guid** for C# Projects. This works because the **Kind** property is actually returning the last project type **Guid** from the **ProjectTypeGuids** element in the project file. As you may know, a project may have multiple project types because they get to be fairly granular, but the language is generally considered to be the "real project" with any others being flavored. This makes sense, considering how the Add New Project dialog box is structured. You first select the language, and then the type of project within the language.

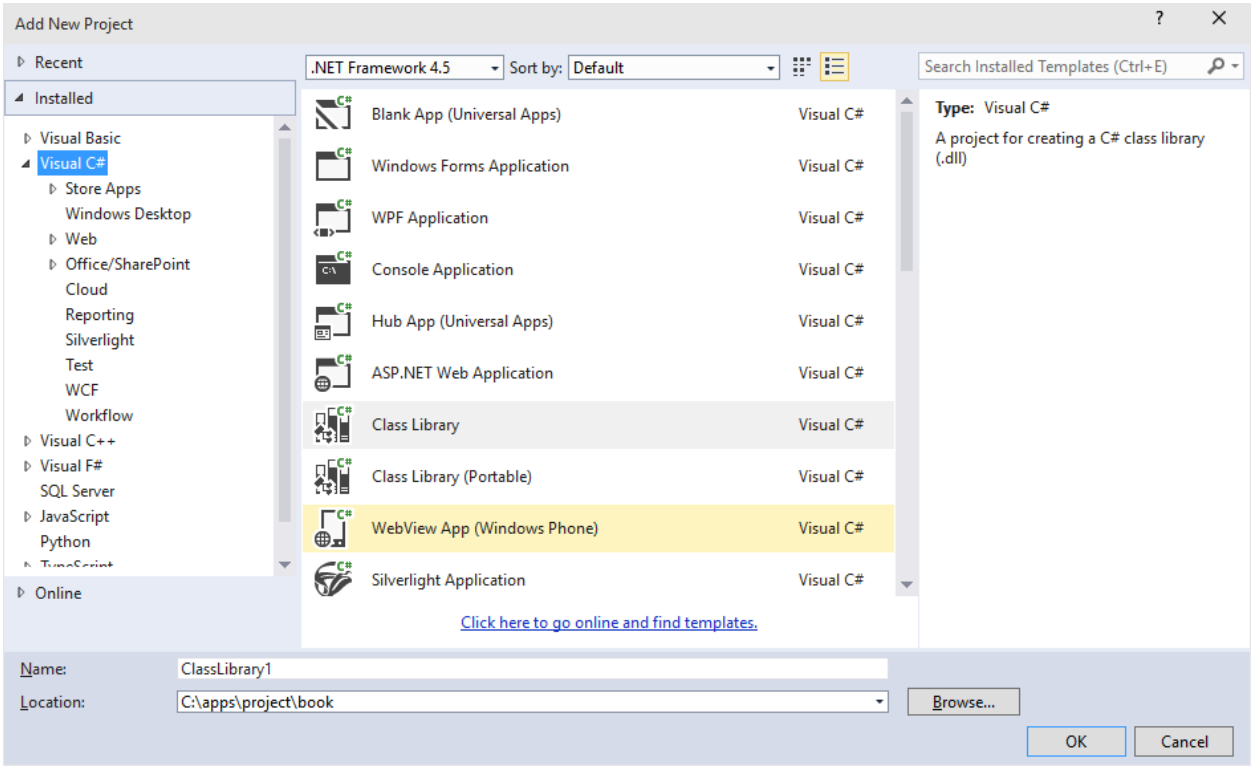

Figure 10: Add New Project Dialog Box. Select Language First.

The complete list of project types is long and subject to change, but can be found here.

In looking over this list, it is obvious that you may want to filter by more than just the language. This turns out to be a bit more complicated. To work this magic, we have to make a deeper dive into COM Interop.

```
public IList<string> GetProjectTypeGuids(EnvDTE.Project proj)
{
    string projectTypeGuids = "";
    int result = 0;
    var solution = GetService
          <Microsoft.VisualStudio.Shell.Interop.IVsSolution>();
    Microsoft.VisualStudio.Shell.Interop.IVsHierarchy hierarchy =
```

```
                    null;
    result = solution.GetProjectOfUniqueName(proj.UniqueName,
                out hierarchy);
    if (result == 0)
    {
       Microsoft.VisualStudio.Shell.Interop.IVsAggregatableProject
              aggregatableProject =
             (Microsoft.VisualStudio.Shell.Interop.IVsAggregatableProject)
             hierarchy;
         result = aggregatableProject.GetAggregateProjectTypeGuids
              (out projectTypeGuids);
    }
    return projectTypeGuids.Split(';');
 }

 private T GetService<T>() where T : class
 {
    T service = null;
    IntPtr serviceIntPtr;
    int hr = 0;
    var guid = typeof(T).GUID;
    Microsoft.VisualStudio.OLE.Interop.IServiceProvider
        serviceProvider =
              (Microsoft.VisualStudio.OLE.Interop.IServiceProvider)
                   VisualStudio;
    hr = serviceProvider.QueryService(ref guid, ref guid,
              out serviceIntPtr);
    if (hr != 0)
    {
       System.Runtime.InteropServices.Marshal
              .ThrowExceptionForHR(hr);
    }
    else if (!serviceIntPtr.Equals(IntPtr.Zero))
    {
        service = System.Runtime.InteropServices.Marshal
                  .GetObjectForIUnknown(serviceIntPtr) as T;
        System.Runtime.InteropServices.Marshal
                  .Release(serviceIntPtr);
    }
    if (service == null)
       throw new Exception("Error retrieving service");
    return service;
 }
```

Code Listing 34: Deeper Dive into COM Interop to get All Project Type GUIDS

Now we can use the **ProjectTypeGuids** method to use some more interesting filters, such retrieving all test projects. The **Guid** for a test project is **{3AC096D0-A1C2-E12C-1390-A8335801FDAB}**.

```
public IEnumerable<Project> GetTestProjects()
{
   var projects = GetAllProjects()
           .Where(project => GetProjectTypeGuids(project)
           .Any(subProject => subProject ==
               "{3AC096D0-A1C2-E12C-1390-A8335801FDAB}"));
   return projects;
}
```

Code Listing 35: Getting all Test Projects

Navigating a Project

Now that we can identify the projects that we might be interested in, let's turn our attention to what we may want to do with one of these projects.

One of the first things that we may want to do is get a list of items in the project. Because projects can contain folders that will also have project items (and in fact, any project item can have nested project items), retrieving a list of all projects will need to be recursive.

```
public IEnumerable<ProjectItem> GetAllProjectItems
                (ProjectItems projectItems)
{
   foreach (ProjectItem projectItem in projectItems)
   {
      foreach (ProjectItem subItem in
         GetAllProjectItems(projectItem.ProjectItems))
      {
         yield return subItem;
      }
      yield return projectItem;
   }
}
```

Code Listing 36: Get all Project Items in a Project

With this method, we can easily do things like find all of the templates in a project.

```
public IEnumerable<ProjectItem >GetAllTemplates
       (ProjectItems projectItems)
{
   return GetAllProjectItems(projectItems)
            .Where(p => p.Name.EndsWith(".tt"));
}
```

Code Listing 37: Finding all of the Templates in a Project

Now that we can find a list of templates, it would be nice to be able to save each of them. This will also run the custom tool, which will evaluate the template.

We can easily do this given a **`ProjectItem`**. All we have to do is save the item.

```
public void SaveItem(ProjectItem item)
{
   var needToClose = !item.IsOpen;
   item.Open();
   item.Save();
   if (item.Document != null)
      if (needToClose)
         item.Document.Close();
}
```

Code Listing 38: Saving a Project Item

Visual Studio requires that an item be open before it can be saved. We don't want to leave a bunch of templates open that were not originally open, and we don't want to close any that were already open. This does the trick.

This opens up the possibility of orchestrating some complex workflows where you can have a template control the order of execution for a collection of templates. For example, you may want to ensure that a particular template is run before another if the first one creates artifacts needed by the second one.

Creating a New Project

You can also design a template that will create a new project. To do this, you will need to add a reference to **`EnvDTE80`** in the T4Utilities project and include this to the **`using`** statements for the **`DTEHelper`** class.

Now we can add the **`CreateProject`** method:

```
public Project CreateProject(string projectName)
{
   if (GetAllProjects().Any(p => p.Name == projectName))
      return GetAllProjects()
        .FirstOrDefault(t=>t.Name == projectName);
   var solution = (Solution2)VisualStudio.Solution;
   var templatePath =
       solution.GetProjectTemplate("ClassLibrary.zip",
            "CSharp");
   var solutionPath = Path.GetDirectoryName
       (VisualStudio.Solution.FullName);
   var path = Path.Combine(solutionPath, projectName);
   return solution.AddFromTemplate(templatePath, path,
        projectName, false);
```

```
}
```

Code Listing 39: Create a Project

This method will create a C# class library. If you need to create a different type of project, simply change the parameters to the **`GetProjectTemplate`** method call.

Note that this function first checks to ensure that the project has not already been created, so this method can safely be called multiple times without causing any damage.

Creating a New Project Item

Now that you have a new project, let's explore what it takes to add an item to it.

The **`ProjectItems`** class exposes four methods of interest.

AddFolder

The **`AddFolder`** method creates a new folder in the project. With this, you could easily initialize the directory structure for an MVC project.

```
public void MimicMVCFolders(Project project)
{
   var existingProjectItems =
      GetAllProjectItems(project.ProjectItems)
         .Select(s => s.Name);
   if (!existingProjectItems.Contains("Models"))
      project.ProjectItems.AddFolder("Models");
   if (!existingProjectItems.Contains("Views"))
      project.ProjectItems.AddFolder("Views");
   if (!existingProjectItems.Contains("Controllers"))
      project.ProjectItems.AddFolder("Controllers");
}
```

Code Listing 40: Create the Folders Used by an MVC Application

AddFromFile

The **`AddFromFile`** method allows you to specify a file on disk and add it to the project. This comes in handy when we create a file and want to add it to the project.

```
public void AddFileToProject(Project project,
            string fileName)
{
   if (File.Exists(fileName))
```

```
        project.ProjectItems.AddFromFile(fileName);
}
```

Code Listing 41: Add an Existing File to the Project

AddFromDirectory

The **`AddFromDirectory`** method steps through a given directory and its subdirectories, automatically adding all of its items to the project. In effect, this recursively calls **`AddFromFile`**, and will save you from having to make repeated calls to the **`AddFromFile`** function.

AddFromTemplate

The **`AddFromTemplate`** method is not as relevant from a code-generation perspective, but it allows you to create a new project item from an existing template. We will generally create our project items in their entirety, and then add them to the solution, but this can also be useful to show what it expected to go into a folder structure for a generated project.

```
public void AddFileFromTemplateToProject
 (ProjectItems project, string fileName)
{
   string parentPath = string.Empty;
   var parent = project.Parent as Project;
   if (parent != null)
      parentPath = parent.FullName;
   else
   {
      var parentItem = project.Parent as ProjectItem;
      if (parentItem == null)
         throw new Exception
             ("Could not retrieve parent path");
      parentPath = parentItem.FileNames[0];
   }
   var projectPath = Path.GetDirectoryName(parentPath);
   if (File.Exists(Path.Combine(projectPath, fileName)))
      return;
   Solution2 solution =
         this.VisualStudio.Solution as Solution2;
   var itemPath = solution.GetProjectItemTemplate
         ("Class.zip", "csharp");
   project.AddFromTemplate(itemPath, fileName);

}
```

Code Listing 42: Adding a File from a Template

This function could be used to add a new class to a project or any folder added to a project. We have also taken care in this function to ensure that it can be called multiple times without throwing any exceptions, because it checks to ensure that the item has not already been added before adding it.

```
var proj = vs.CreateProject("book.help");
vs.MimicMVCFolders (proj);
vs.AddFileToProject(proj, path);
vs.AddFileFromTemplateToProject(proj.ProjectItems,
         "NewClass.cs");
var models = vs.GetAllProjectItems
         (proj.ProjectItems).FirstOrDefault
                            (p=>p.Name == "Models");
vs.AddFileFromTemplateToProject(models.ProjectItems,
         "SampleModel.cs");
var controller = vs.GetAllProjectItems
          (proj.ProjectItems).FirstOrDefault
                           (p=>p.Name == "Controllers");
vs.AddFileFromTemplateToProject(controller.ProjectItems,
         "SampleController.cs");
```

Code Listing 43: Pulling the Pieces Together

After running this snippet of code, we will have a new project called **book.help** with three folders called Models, Views, and Controllers. We will have a class in the root folder called **`NewClass`**, and sample classes in the Models and Controllers folders.

Generating More than One File from a Template

One annoying thing about T4 is that by default, the generated content will be stored in a nested file under the template with the same as the template, plus whatever extension you specified. This is actually rarely what we want.

In this chapter, we have seen various ways to add new content to a project, including creating a new project altogether. In the last chapter, we saw how to use run time templates, which allow us to create templates that we can run at run time and get the generated output in a string variable. We can combine these two pieces of information to be able to put the generated content wherever we want in the solution and name them whatever we want.

Let's go back to the T4Utilities project that we have been working in, create a new run time template, and call it **ClassCreater.tt**.

`ClassCreater` will be fairly simple. We will define a template that has a couple of properties that are used to generate a class. Add the following code to **`ClassCreator`**:

```
<#@ template language="C#" #>
<#@ assembly name="System.Core" #>
```

```
<#@ import namespace="System.Linq" #>
<#@ import namespace="System.Text" #>
<#@ import namespace="System.Collections.Generic" #>
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

public class <#=ClassName#>
{
<#
 foreach (var property in Properties)
 {
#>
  public <#=property.DataType #> <#= property.Name#> {get;set;}
<# }
#>
}

<#+
 public string ClassName{get ; set;}
 public IList<PropertyDescription> Properties{get;set;}
 public class PropertyDescription
 {
    public string Name {get; set;}
    public string DataType {get;set;}
 }
#>
```

Code Listing 44: Class Generator Template

Now, back in the book project, create a new text template and name it **SetupProject.tt**.

In this template, we will need to declare an instance of the class created by the **`ClassCreator`** template. We will initialize the properties on this class and call the **`TransformText`** method, storing the output to a local variable.

We'll write the local variable out to a file that we will then add to a newly created project.

```
<#@ template debug="true" hostspecific="true" language="C#" #>
<#@ assembly name="System.Core" #>
<#@ import namespace="System.Linq" #>
<#@ import namespace="System.Text" #>
<#@ import namespace="System.Collections.Generic" #>
<#@ output extension=".cs" #>

<#@ assembly name="EnvDTE" #>
<#@ assembly name="EnvDTE100" #>
<#@ import namespace="EnvDTE" #>
```

```
<#@ import namespace="EnvDTE100" #>
<#@ import namespace="System.Linq" #>
<#@ import namespace="System.Collections.Generic" #>
<#@ import namespace="System.IO" #>

<#@ assembly name="$(SolutionDir)T4Utilities\bin\Debug\T4Utilities.dll" #>
<#@ import namespace = "T4Utilities" #>


<#
 var template = new T4Utilities.ClassCreator();
 template.ClassName = "SampleModel";
 template.Properties =
  new List<ClassCreator.PropertyDescription>();
 template.Properties.Add (
  new ClassCreator.PropertyDescription
   {Name = "FirstName", DataType = "string"});
 template.Properties.Add (
  new ClassCreator.PropertyDescription
   {Name = "LastName", DataType = "string"});
 template.Properties.Add (
  new ClassCreator.PropertyDescription
   {Name = "StreetAddress", DataType = "string"});
 template.Properties.Add (
  new ClassCreator.PropertyDescription
   {Name = "City", DataType = "string"});
 template.Properties.Add (
  new ClassCreator.PropertyDescription
   {Name = "State", DataType = "string"});
 template.Properties.Add (
  new ClassCreator.PropertyDescription
   {Name = "Zip", DataType = "string"});
 template.Properties.Add (
  new ClassCreator.PropertyDescription
   {Name = "HireDate", DataType = "DateTime"});
 template.Properties.Add (
  new ClassCreator.PropertyDescription
   {Name = "LastStatusChangeDate",
   DataType = "DateTime"});
 var vs = new DTEHelper(this.Host);
 var proj = vs.CreateProject("book.help.MVC");
      vs.MimicMVCFolders (proj);
 var models = vs.GetAllProjectItems(proj.ProjectItems)
   .FirstOrDefault (p=>p.Name == "Models");
 var path = Path.Combine(models.FileNames[0],
  "SampleModel.cs");
 using (StreamWriter writer = new StreamWriter(path))
   {
      var code = template.TransformText();
      writer.Write (code);
```

```
        vs.AddFileToProject(proj, path);
    }
#>
```

Code Listing 45: SetupProject.tt

As you can see, being able to manipulate the DTE opens up some very exciting opportunities for our templates.

Chapter 5 It's All About the Metadata

Metadata is data about data. Here, it is data that will drive the code generation. This is data describing the code that we want to generate. Quite literally, metadata drives code generation.

Keeping Your Metadata Clean

If your code generation approach relies on being able to regenerate on demand, you must protect your metadata. You want to make sure that it stays current and clean. Without proper care, these two goals may be at odds with each other. We will discuss strategies to help manage this contention. We will also see shortly that there are two types of metadata that we will work with.

Intrinsic Metadata is metadata that we don't really have to worry about updating. This data stays current without any extra effort on our part. Examples include Reflection, data that the compiler will keep current every time you compile, and the database Data Dictionary which the RDMS will keep current for us. Chapter 7 will focus on Reflection and Chapter 6 will focus on the Data Dictionary

Extrinsic metadata will require more work on our part. This is metadata that we might create specifically for Code Generation or it may already exist to support other parts of the system. The key thing is that we need an external process to update and ensure that it is current and accurate. Examples here are often domain specific and will often dependent greatly on the business needs.

Keep Your Generated Code Clean

If your development process is based on being able to regenerate code on demand, you must also keep the generated code clean. This means making sure that any changes to the code are either incorporated into future regenerations or are protected from being lost during future regenerations.

We have a couple of strategies for ensuring that code changes persist across regenerations. The simplest is updating the metadata used to drive the code generation, but sometimes this is not enough. We also may need to update the template used to generate the code.

For example, we can easily change the code generated for an Entity Framework entity by updating the metadata. Add a column to a table, and the corresponding entity gets a new property. Instead of adding the property, update the metadata first, and let the code generation update the code.

Alternately, we may decide that we want to add an **`IsVisible`** property to every defined property for a **`ViewModel`**. This will be a relatively straightforward change to the template driving the code generation. Again, we don't want to make such changes directly to the generated code. Instead, update the template and let code generation update your code.

Tip: The key thing to remember is DO NOT directly change the files generated by the code generation.

Sometimes we may have a "one off" change to make to the generated code. This is a change that does not follow a pattern and may not be reproducible or needed anywhere else. Such changes cannot easily be made or described using metadata or templates. Examples may include calculated fields that are not mapped to the database, or context-specific, custom validation logic.

Even in these scenarios, we don't want to directly manipulate the code that was generated. This does not mean that we cannot change the code. We cannot change the code in the output file created by the code generation, but thanks to partial classes, we can live with this restriction.

Tip: Always mark generated classes as partial so that the implementation can be spread over multiple files, only one of which will actually be off limits. We are free to add a new file to the class definition and make whatever changes are needed there.

Finding Metadata

An initial challenge when generating code is finding metadata. Fortunately, .NET is all about metadata. We access this metadata through reflection. We will explore reflection further in Chapter 8. For now, we will simply mention that reflection gives us access to all of the data about the types in an assembly.

Reflection

Using reflection, we can load an assembly and search through the types in that assembly. For any type, we can search through its properties, events, and methods. For any method, we can loop through the parameters and examine the return type.

This gives us access to a lot of information. In Chapter 7, we will step through code showing how to access this data, as well as explore some options for using this information to drive code generation.

Data Dictionary

The data dictionary in every database is another great source for metadata. Without any intervention on your part, the Relational Database Management System (RDMS) keeps track of metadata about every table, every column in each table, every stored procedure, and each parameter for these stored procedures, as well as the details for every relationship and constraint in the database. The RDMS keeps track of even more metadata just to keep the database running, but these are the main types of metadata that we will care about for code generation. Databases are very metadata intensive, and the data dictionary is the key to this metadata.

In Chapter 6, we will explore the SQL Server Management Objects (SMO) library, which provides a rich object model for accessing this data and much more.

Code Model

The code model is an API provided by Visual Studio for walking through the code in Visual Studio before it is compiled. It gives us access to the same data that we get through reflection without having to compile to an assembly first.

The code model is a notoriously difficult API to work with, but don't worry—we will decipher its mysteries, and show how to get data we need to drive code generation.

Chapter 8 will explore the code model further.

Chapter 6 Working with SQL Server Metadata

Microsoft.SQLServer.SMO

SMO stands for Server Management Objects. SMO is a collection of objects designed to give you programmatic access to managing all aspects of SQL server. We don't need all of this power, but the library also provides an easy and straightforward way to access the data dictionary for SQL Server.

Note: Using SMO will tie you directly to SQL Server. If you're using a different RDMS, you can still retrieve this metadata by querying the tables in your database's data dictionary directly. Refer to your RDMS documentation.

As long as you opted to install the Client Tools SDL when you installed SQL Server, you already have SMO installed. If you did not, then you can download and install the SQL Server Feature Pack. Make sure to install the correct version for your version of SQL Server.

Throughout this chapter we will build up the functionality for a new class named **`SQLHelper`** in the **T4Utilities** project that we started in Chapter 4. We have a few new references that we will need to add to the T4Utilities project:

- **`Microsoft.SqlServer.Smo`**
- **`Microsoft.SqlServer.ConnectionInfo`**
- **`Microsoft.SqlServer.Management.Sdk.Sfc`**

You will also need to add references to these assemblies for any template that will use our SQL Helper.

Connect to a Server

The first step in dealing with SMO is to connect to the server. Fortunately, this is very straightforward. For the most part, from a code generation perspective, you will generally be connecting to your local server.

Let's add a constructor to the **`SQLHelper`** object that will accept the name of the server to connect to, and will initialize a connection to this server.

```
private Server server {get;set;}
public SQLHelper(string serverName)
{
    server = new Server(serverName);
```

```
}
```

Code Listing 46: Constructor for SQLHelper

Connect to a Database

Now that we are connected to a server, you want to get access to the databases on that server. Often we want to connect to a particular database.

```
public Database GetDatabase(string whichDatabase)
{
   if (!GetAllDatabases()
        .Any(d => d.Name == whichDatabase))
       throw new Exception(whichDatabase
              + "was not found on the server");
   var database = new Database(server, whichDatabase);
   database.Refresh();
   return database;

}

public IList<Database> GetAllDatabases()
{
   var returnValue = new List<Database>();
   foreach (Database db in server.Databases)
   {
      returnValue.Add(db);
   }
   return returnValue;
}
```

Code Listing 47: GetAllDatabases or Get a Specific Database

Here we add in an extra check to ensure that the database requested actually exists on the server.

Get a List of Tables

Once we have access to an existing database, we will often need to get a list of tables in that database:

```
public IList<Table> GetAllTables (Database database)
{
   var returnValue = new List<Table>();
   foreach (Table currentTable in database.Tables)
   {
       returnValue.Add(currentTable);
```

```
    }
    return returnValue;
}
```

Code Listing 48: Get all the Tables in a Database

With this data brought back as a **`List`** object, we are able to easily filter this by any of the properties exposed by the table. For our purposes, we will most often be interested in filtering by owner.

```
public IList<Table> GetAllTablesOwnedBy
  (Database database, string owner)
{
   return GetAllTables(database)
       .Where(t => t.Owner == owner).ToList();
}
```

Code Listing 49: Filter Tables by Owner

Get Details about a Table

Now that we can readily get the tables that we are interested in, there are a lot of details about the tables that we may care about. For starters, let's get a list of the columns in a table:

```
public IList<Column> GetAllColumns (Table whichTable)
{
   var returnValue = new List<Column>();
   foreach (Column currentColumn in whichTable.Columns)
   {
      returnValue.Add(currentColumn);
   }
   return returnValue;
}
```

Code Listing 50: Get All the Columns in a Table

Now we can filter this list of columns by any property exposed by **`Column`**, and **`Column`** provides some useful properties to filter by:

Column Properties	
Computed	Boolean value indicated if the column is a computer value
DataType	Details about the data type for the column

Column Properties	
InPrimaryKey	Boolean value indicating if the column is part of the primary key or a unique constraint
IsForeignKey	Boolean value indicating if the column is a foreign key
Name	Gets the name of the column

Generating Stored Procedures

Now that we have this metadata, we can easily do things like create the stored procedures to encapsulate accessing the tables in the database.

Let's start by creating a new template, and name it **StoreProcedures.tt**.

```
<#@ template debug="true" hostspecific="true" language="C#" #>
<#@ output extension=".sql" #>
<#@ assembly name="System.Core" #>
<#@ import namespace="System.Linq" #>
<#@ assembly name= "System.Xml"#>
<#@ assembly name="C:\Program Files (x86)\Microsoft SQL
Server\110\SDK\Assemblies\Microsoft.SqlServer.Smo.dll" #>
<#@ assembly name="C:\Program Files (x86)\Microsoft SQL
Server\110\SDK\Assemblies\Microsoft.SqlServer.ConnectionInfo.dll"  #>
<#@ assembly name="C:\Program Files (x86)\Microsoft SQL
Server\110\SDK\Assemblies\Microsoft.SqlServer.Management.Sdk.Sfc.dll" #>
<#@ import namespace="Microsoft.SqlServer.Management.Smo" #>
<#@ assembly  name="$(SolutionDir)T4Utilities\bin\Debug\T4Utilities.dll" #>
<#@ import namespace = "T4Utilities" #>
<#
    var smo = new SQLHelper("localhost");
    var database = smo.GetDatabase("Northwind");
    var tables = smo.GetAllTables(database).ToList();
    foreach (var table in tables)
    {
        GenerateUpdateProcedure(table);
        GenerateDeleteProcedure(table);
    }
#>
```

You may need to adjust the path for the SMO assemblies, depending on how you have them installed. You can get the path based on how the reference was added to the T4 Utilities project. You can also change the name of the parameter to the **GetDatabase** method to refer to any database that you are interested in.

The heart of the code generation takes place in **GenerateUpdateProcedure** and **GenerateDeleteProcedure**. Let's first think about what an stored update procedure should look like.

In its simplest form, an update procedure would look like this:

```
CREATE PROCEDURE Region_update
        @RegionID int,
        @RegionDescription nchar
AS
BEGIN

          UPDATE Region
          SET
              RegionDescription = @RegionDescription
        WHERE
           RegionID = @RegionID
END
GO
```

Code Listing 51: Update Procedure

Your naming conventions may vary, but the general structure will be the same. You want a parameter for every column, and then you will set each column not part of the primary key to the value passed in. You will filter to only update the intended records based on the column that makes up the primary key.

So let's look at the **GenerateUpdateProcedure**:

```
<#+
    public void GenerateUpdateProcedure(Table table)
    {
        if (table.Columns.Cast<Column>().All(p => p.InPrimaryKey))
            return;
       WriteLine(string.Format("CREATE PROCEDURE {0}_update ",
                    table.Name));

        var count = 0;
        PushIndent("        ");
        foreach (var column in table.Columns.Cast<Column>())
        {
            if (count > 0) this.WriteLine(",");
            Write("@" + column.Name + " " + column.DataType);
            count ++;
        }
```

```
        PopIndent();
        WriteLine(String.Format(@"
AS
BEGIN

            UPDATE {0}
            SET", table.Name));

        PushIndent("              ");
        count = 0;
        foreach (var column in table.Columns.Cast<Column>()
             .Where(column => !column.InPrimaryKey))
        {
            if (count > 0) Write(" , ");
            WriteLine(column.Name + " = @" + column.Name);
            count ++;
        }
        PopIndent();
        PushIndent("        ");
        WriteLine("WHERE");
        count = 0;
        foreach (var column in table.Columns.Cast<Column>()
             .Where(column => column.InPrimaryKey))
        {
            PushIndent("   ");
            if (count > 0) Write(" and ");
            WriteLine(column.Name + " = @" + column.Name);
            count ++;
        }
        PopIndent();
        PopIndent();
        WriteLine("END");
        WriteLine("GO");
        WriteLine("\n");
    }

#>
```

Code Listing 52: General Update Procedure

The first thing we do is make sure that there is at least one updateable column. If every column is part of the primary key, then we really have nothing to update, and need to just return.

We also want to keep track of how many columns we have processed. When dealing with the parameters, we want to add a comma before each column (except the first one).

For the **`set`** clause, we want to include every column that is not part of the primary key, and for the **`where`** clause, we want to include every column that is part of the primary key.

The **`GenerateDeleteProcedure`** will look similar, with the following exceptions: we only need to include parameters for columns in the primary key, and there is no **`set`** clause to worry about.

Using the **Northwinds** training database, the output would look like this:

```
CREATE PROCEDURE Categories_update
        @CategoryID int,
        @CategoryName nvarchar,
        @Description ntext,
        @Picture image
AS
BEGIN

            UPDATE Categories
            SET
              CategoryName = @CategoryName
               , Description = @Description
               , Picture = @Picture
        WHERE
           CategoryID = @CategoryID
END
GO


CREATE PROCEDURE Categories_delete
        @CategoryID int
AS
BEGIN

            DELETE FROM Categories
            WHERE
              CategoryID = @CategoryID
END
GO
CREATE PROCEDURE Categories_update
        @CategoryID int,
        @CategoryName nvarchar,
        @Description ntext,
        @Picture image
AS
BEGIN

            UPDATE Categories
            SET
              CategoryName = @CategoryName
               , Description = @Description
               , Picture = @Picture
        WHERE
           CategoryID = @CategoryID
END
GO
```

```
CREATE PROCEDURE Categories_delete
        @CategoryID int
AS
BEGIN

            DELETE FROM Categories
            WHERE
              CategoryID = @CategoryID
END
GO


CREATE PROCEDURE CustomerCustomerDemo_delete
        @CustomerID nchar,
        @CustomerTypeID nchar
AS
BEGIN

            DELETE FROM CustomerCustomerDemo
            WHERE
              CustomerID = @CustomerID
               AND CustomerTypeID = @CustomerTypeID
END
GO
```

Code Listing 53: Generate Delete Procedure

You can easily modify the template to follow any naming conventions that you like.

You can also easily follow the same pattern to create and insert stored procedures, as well as select stored procedures by the primary key and every foreign key defined.

Run Arbitrary SQL

Easy access to this metadata is very useful for generating code for a variety of scenarios, but sometimes we also need to run SQL statements. SMO provides an easy way to run SQL and return the results in a **`DataTable`**. Any time you run SQL like this, you will need to add an assembly directive to include **`System.Data`**.

We can use such SQL statements to access custom metadata that we create.

Let's consider what domain-specific metadata we could gather from the **NorthWinds** database. For your reference, here is the data model for this training database:

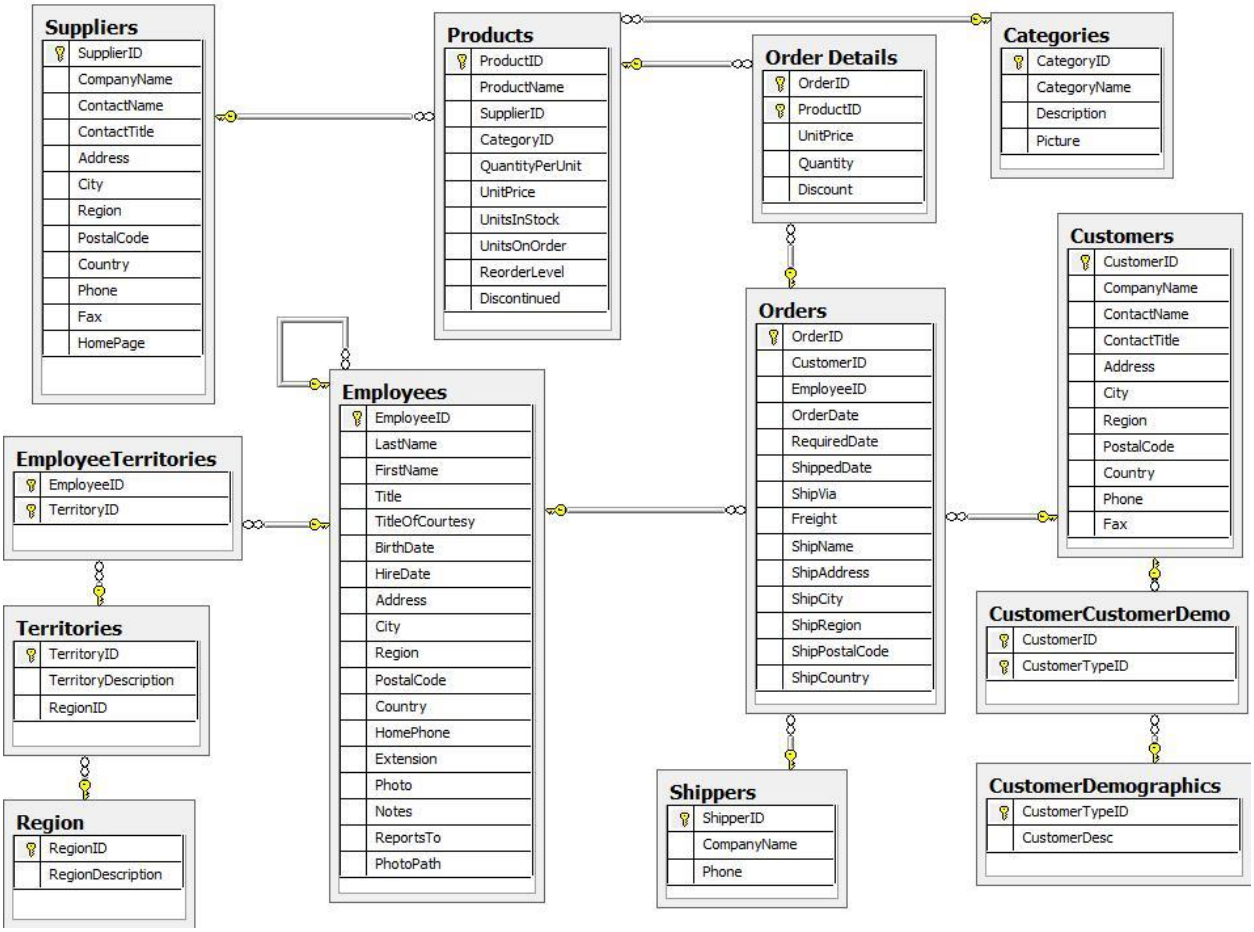

Figure 11: NorthWind Data Model

Create a new class in the **T4Utilities** project, and call it **NorthWindHelper**. We will add a few queries to this class to help automate access some of the metadata.

Let's start with a simple class to hold a region:

```
public class Region
{
   public string RegionDescription {get;set;}
   public int RegionId {get;set;}
}
```

Code Listing 54: Simple POCO to Hold Data from the Region Table

Now, to query the **Region** table and bring back an **IList<Region>**, all we have to do is this:

```
public IList<Region> GetAllRegions()
{
   var returnValue = new List<Region>();
   var sql = @"Select RegionId, RegionDescription
               from Region";
```

```
    var smo = new SQLHelper("localhost");
    var db = smo.GetDatabase("NorhWind");
    var dataset = db.ExecuteWithResults(sql);
    foreach (DataTable currentTable in dataset.Tables)
    foreach (DataRow currentRow in currentTable.Rows)
      returnValue.Add(new Region()
      {
         RegionDescription = (string)currentRow[1],
         RegionId = (int)currentRow[0]
      });
    return returnValue;
  }
```

Code Listing 55: Query the Region Table Using SMO

Now we can create a template that will read this data and create a class for each region configured in the database. This might be a precursor to creating a hierarchy of objects to handle region-specific business logic without having to revert to case statements sprinkled throughout the code.

Add a new text template to the book project that we have been working in, and name it **Region.tt**.

Add the following code to **Region.tt**:

```
<#@ template debug="true" hostspecific="true" language="C#" #>
<#@ assembly name="System.Core" #>
<#@ import namespace="System.Linq" #>
<#@ import namespace="System.Text" #>
<#@ import namespace="System.Collections.Generic" #>
<#@ output extension=".txt" #>
<#@ assembly name="EnvDTE" #>
<#@ assembly name="EnvDTE100" #>
<#@ import namespace="EnvDTE" #>
<#@ import namespace="EnvDTE100" #>
<#@ import namespace="System.Linq" #>
<#@ import namespace="System.Collections.Generic" #>
<#@ import namespace="System.IO" #>
<#@ assembly
 name="$(SolutionDir)T4Utilities\bin\Debug\T4Utilities.dll" #>
<#@ import namespace = "T4Utilities" #>

<#
 var vs = new DTEHelper(this.Host);
 var project = vs.GetAllProjects().ToList()
  .FirstOrDefault(p=> p.Name == "book");
 var existingProjectItems =
  vs.GetAllProjectItems(project.ProjectItems)
  .Select(s => s.Name);
 if (!existingProjectItems.Contains("NorthWind"))
```

```
  project.ProjectItems.AddFolder("NorthWind");
 var northWindFolder = vs.GetAllProjectItems
  (project.ProjectItems).ToList()
  .FirstOrDefault (i=>i.Name == "NorthWind");
 var northwind = new NorthWindUtils();
 var regions = northwind.GetAllRegions();
 foreach (var region in regions)
 {
  var template = new T4Utilities.ClassCreator();
  template.ClassName = region.RegionDescription;
  template.Properties =
  new List<ClassCreator.PropertyDescription>();
  template.Properties.Add (new
  ClassCreator.PropertyDescription
  {
    Name = "RegionName",
    DataType = "string"
  });
  template.Properties.Add (new
  ClassCreator.PropertyDescription
  {
   Name = "SalesProjection",
   DataType = "decimal"
  });
  var path = Path.Combine(northWindFolder.FileNames[0],
  region.RegionDescription+ ".cs");
  using (StreamWriter writer =
        new StreamWriter(path))
       {
  var code = template.TransformText();
  writer.Write (code);
       }
  vs.AddFileToProject(project, path);
 }
#>
```

Code Listing 56: Region.tt

Here we create new folder in the book project, naming it **NorthWind**. Then we connect to the **NorthWind** database and get a list of the regions. We then create a class for each region based on the **`ClassCreator`** from Chapter 4, adding properties for **`RegionName`** and **`SalesProjection`**.

Summary

SQL Server provides a wealth of metadata about the database and the structures in that database. We have seen accessing the databases on the server, the tables in a database, and the columns in a table. We have also seen running any arbitrary SQL that we might need to get more domain specific metadata.

In addition to the metadata that we have explored here, SMO exposes a wealth of additional metadata that can be helpful in generating code. Just as we accessed the tables in a database, you can also access the stored procedures and their parameters.

SMO also makes it relatively painless to automate the task of creating any database structures that you might need, making it easier to keep your code in sync with the databases you are working with.

Chapter 7 Working with Reflection

Introduction

Reflection is how we access metadata about code at run time. This opens up some wonderful dynamic runtime possibilities, but it can also be leveraged to create some exciting code generation opportunities.

In this chapter, we will look through the various types of metadata that we have available, showing how the pieces fit together. Once we have finished this survey of reflection techniques, we will tie it all together with a code generator that will load an assembly and find all of the types that are derived from **`ApiController`**. For each of these classes, we will find the actions that are exposed and create a method to automate calling that action using **`HttpClient`**.

Throughout this chapter we will add functionality to a new class called **`ReflectionHelper`** that we can use to simplify accessing eeflection data. Go ahead and a new class called **`ReflectionHelper`** in the **T4Utilities** project that we created in Chapter 4.

Load an Assembly

Loading an assembly can be a bit tricky. Without proper care, you can wind up locking the assembly that you are trying to load. If you are reflecting on an assembly being built by your current solution, you could be stuck having to shut down Visual Studio every time you run your template.

We have a couple of options to avoid this. The most straightforward approach is to read the contents of the assembly into a **`Byte`** array, and then load the array. With this approach there is no file to lock. We can see a simple implementation in Code Listing 54. Add the following method to the **`ReflectionHelper`** class that we added to the T4Utilities project.

```
public void LoadAssembly(AssemblyName assembly)
{
   var url = new Uri(assembly.CodeBase);
   var bytes = File.ReadAllBytes(url.AbsolutePath);
   TargetAssembly = Assembly.Load(bytes);
}
public Assembly TargetAssembly {get;set;}
```

Code Listing 57: Load an Assembly Without Locking

`AssemblyName` is a handy object in the **`Reflection`** namespace that simplifies the process of referring to an assembly and ensuring that you are referring to the correct one. It can be initialized like this:

```
var path = this.Host.ResolvePath (@"..\WebApi\bin\WebApi.dll");
var name = System.Reflection.AssemblyName.GetAssemblyName(assemblyPath);
```

Code Listing 58: Initializing an AssemblyName

This approach to loading an assembly will ensure that you can continue compiling the code that generated the assembly being loaded and rerunning the templates using that assembly without clashing with each other.

Note: This is very important, because nothing is more frustrating than having to exit and restart Visual Studio just to unlock an assembly so you can recompile. This is an area that has been greatly enhanced in the more recent version of Visual Studio, which used to lock assemblies behind the scenes in many places. But as of Visual Studio 2013, this should be a thing of the past. Any locked assemblies should be self-inflicted, and avoidable by following this simple trick.

Get a List of Types

While gathering the metadata to support code generation, we will usually be interested in multiple types in an assembly. This means that we will call **`GetTypes`** to get all of the types in the assembly. Depending on your metadata requirements, you may know which type you are interested in. If this is the case, you can call the **`GetType`** method, which will return a single **`Type`** object.

Tip: In general, you will be better off making a single call to GetTypes rather than several calls to GetType. Unless you are explicitly interesting in a single type, GetTypes is probably the better choice.

Getting a list of types is pretty straightforward:

```
        private IList<Type> _cachedTypes;
        public IList<Type> GetTypes()
        {
            if (_cachedTypes == null)
                _cachedTypes = TargetAssembly.GetTypes().ToList();
            if (_cachedTypes.First().Assembly != TargetAssembly)
                _cachedTypes = TargetAssembly.GetTypes().ToList();
            return _cachedTypes;
        }
```

Code Listing 59: Getting a List of Types

Note: The list of types is cached to improve performance. Getting the list of types is a fairly expensive operation. If the TargetAssembly has not changed, there is no reason to repeatedly to through the cost of building this list.

We can build on this to simplify some common filters by running LINQ queries against the results, using any of the methods or properties exposed by **Type**.

For example:

```
public IList<Type> GetTypesInNamespace (string nameSpace)
{
   return GetTypes()
      .Where(t=>t.Namespace == nameSpace).ToList();
}
```

Code Listing 60: Filtering Types By NameSpace

Or

```
public IList<Type> GetInterfaces()
{
    return GetTypes()
          .Where(t => t.IsInterface ).ToList();
}
```

Code Listing 61: Getting all Interfaces

We can add a couple of helper functions to give us even more options:

```
public bool HasBaseType (Type currentType, Type baseType)
{
   return GetBaseClasses(currentType).Any(b => b == baseType);
}
public List<Type> GetBaseClasses(Type type)
{
   var allBases = new List<Type>();
   var derived = type;
   do
   {
      derived = derived.BaseType;
      if (derived != null)
         allBases.Add(derived);
   } while (derived != null);
   return allBases;
}
```

Code Listing 62: Determing if a Class has a Particular Base Class

The **GetBaseClasses** method will walk the entire inheritance hierarchy back to object and return this as a list. **HasBaseType** will determine if any of the items returned matches the item that we are looking for.

With these helper functions, we can now include a filter like this:

```
public IList<Type > GetTypesDerivedFrom (Type baseType)
{
   return GetTypes()
       .Where(t=> HasBaseType(t, baseType)).ToList();
}
```

Code Listing 63: Filtering Types Based on Base Class

Once we have an Assembly loaded, we can easily get all of the types, a single type by name, or list of types filtered any number of ways.

Get the Properties of a Type

Once we have identified the type or types that we are interested in, we can start learning more about the members in that type.

Again, getting the properties is fairly straightforward:

```
public IList<PropertyInfo> GetProperties (Type name)
{
   return name.GetProperties().ToList();
}
```

Code Listing 64: Getting the Properties of a Type

There are some built-in filters that .NET provides directly through the **BindingFlags** that can be passed to the **GetProperties** method. As will see shortly, these **BindingFlags** can be passed to a number of places.

Depending on your metadata requirements, you may want different filters, but a common practice is to return only the public properties defined at this level of inheritance.

```
public IList<PropertyInfo> GetProperties (Type name)
{
   return name.GetProperties(
           BindingFlags.DeclaredOnly |
           BindingFlags.Public |
           BindingFlags.Instance).ToList();
}
```

Code Listing 65: Adding BindingFlags to GetProperties

This will filter the properties and only return the public ones defined at this level of inheritance that are not static.

Getting the Methods of a Type

Getting the methods defined in a type is very similar to getting the properties.

```
public IList<MethodInfo> GetMethods(Type name)
{
   return name.GetMethods(
           BindingFlags.DeclaredOnly |
           BindingFlags.Public |
           BindingFlags.Instance)
        .Where(m => !m.Name.StartsWith("get_")
                 && !m.Name.StartsWith("set_")).ToList();
}
```

Code Listing 66: Getting a List of Methods Including the Binding Flags Filtering Out Properties

We are using the binding flags that we saw earlier, but I add an extra filter. We want to remove the methods that start with **`get_`** and **`set_`**. These are the methods that implement the getters and setters for the properties in the type. As a general rule, when we are dealing with methods, we are not interested in the methods that implement a property. We will deal with those methods as we deal with the property.

Methods also have a couple of interesting properties worth exploring: **`ReturnType`** and **`Parameters`**.

We may want to filter based on the **`ReturnType`** like this:

```
public IList<MethodInfo> GetMethodsReturningVoid(Type name)
{
   return GetMethods(name)name)
        .Where(m=>m.ReturnType == typeof(void)).ToList();
}
}
```

Code Listing 67: Filtering Methods Based on Return Type

Alternately, we may want to filter based on some attribute of the parameters.

```
public IList<MethodInfo> GetMethodsWithNoParameters(Type name)
{
return GetMethods( name ))
 .
 .Where(m => m.GetParameters().Count() == 0).ToList();
}
```

Code Listing 68: Filtering Based on the Parameters

Get the Attributes of a Type, Method, or Property

The **GetCustomAttributes** method is defined as part of the **MemberInfo** class. Fortunately for us, this just so happens to be the base class for types, methods, and properties. We can easily write a method to get the custom attributes for any of these types like this:

```
public IEnumerable<Attribute> GetAttributes(MemberInfo member)
{
   var list = member.GetCustomAttributes(true).ToList();
   var returnValue = new List<Attribute>();
   foreach (var item in list)
   {
      returnValue.Add(item as Attribute);
   }
   return returnValue;
}
```

Code Listing 69: Getting Attributes

Once we have this list of attributes, we can define a handy function to easily determine if the member that we are interested in has a particular attribute.

```
public bool HasAttribute(MemberInfo member, string attr)
{
   return GetAttributes(member)
    .Any(a => a.GetType().Name == attr);
}
```

Code Listing 70: Checking to See If a Member Has a Particular Attribute

Now that we have the handy **HasAttribute** method, we could write new filters against properties or methods, taking into account their custom attributes.

```
public IList<PropertyInfo> GetRequiredProperties (Type member)
{
   return GetProperties(member)
    .Where(p => HasAttribute(p, "RequiredAttribute")).ToList();
}
```

Code Listing 71: Filtering Properties by Attribute

One key point to remember here is that even though we think of the “Required” attribute, the full name for this attribute is **“RequiredAttribute”**.

Defining a Custom Attribute

Now that we know how to access the types in an assembly and the various members of a type, as well as the custom attributes for each of these, the next question is: “How do I create my own custom attribute?”

It's pretty straightforward; all we need to do is define a new class and derive it from **`System.Attribute`**.

Let's go ahead and add a new project to our solution and name it **book.Model**. Let's add a new class to this project called **`HideMethodAttribute`**.

Add the following code to **`HideMethodAttribute`**:

```
using System;

namespace book.Models
{

   [System.AttributeUsage(System.AttributeTargets.Method)]
   public class HideMethodAttribute : Attribute
   {
   }
}
```

Code Listing 72: Defining a Simple Attribute

The **`AttributeUsageAttribute`** might seem a little strange at first. It simply instructs the compiler that this attribute is valid only on methods. Thanks to this attribute, you will get a compile error if you try to use it anywhere else.

Properties for the attribute are simple properties added to this class, defined like any other property, set when it is applied, and accessed during code generation. That is it for a simple attribute. The interpretation and meaning of the attribute comes from what we do when we detect that the attribute has been applied to a method.

Pulling it All Together

Go back to the solution that we have been working in, and add a new project. This time, add a new MVC Web API Project.

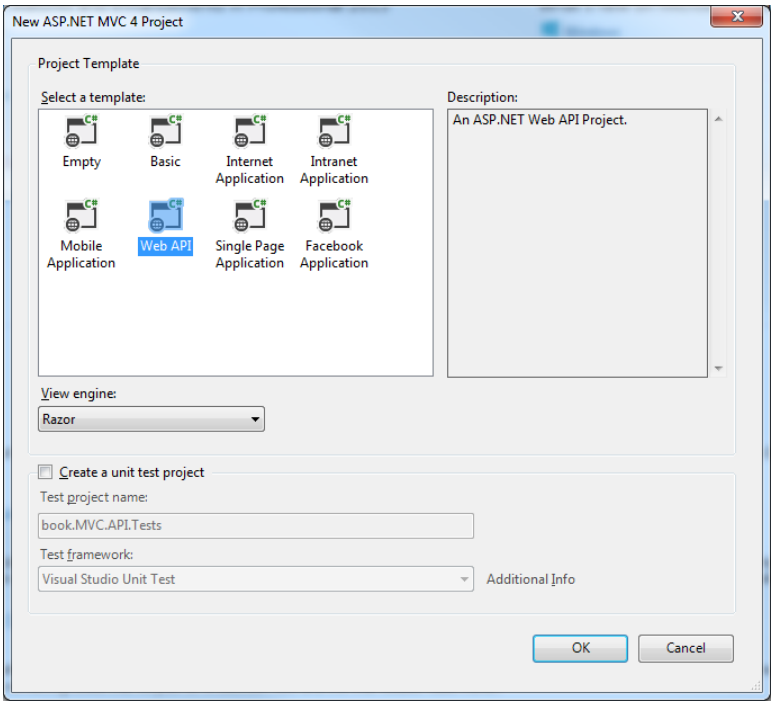

Figure 12: Adding a Web API Project

Add a reference to the **book.Model** project that we created when we discussed creating a custom attribute. In this project, let's create a new class called **`EmployeeModel`**.

Add the following code to **`EmployeeModel.cs`**:

```
namespace book.Model
{
  public class EmployeeModel
  {
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime DateOfBirth { get; set; }
    public DateTime HireDate { get; set; }
    public DateTime LastReview { get; set; }
  }
}
```

Code Listing 73: Some Common Properties for an Employee Model

Now let's go back to the Web API project, and create a controller based on this model.

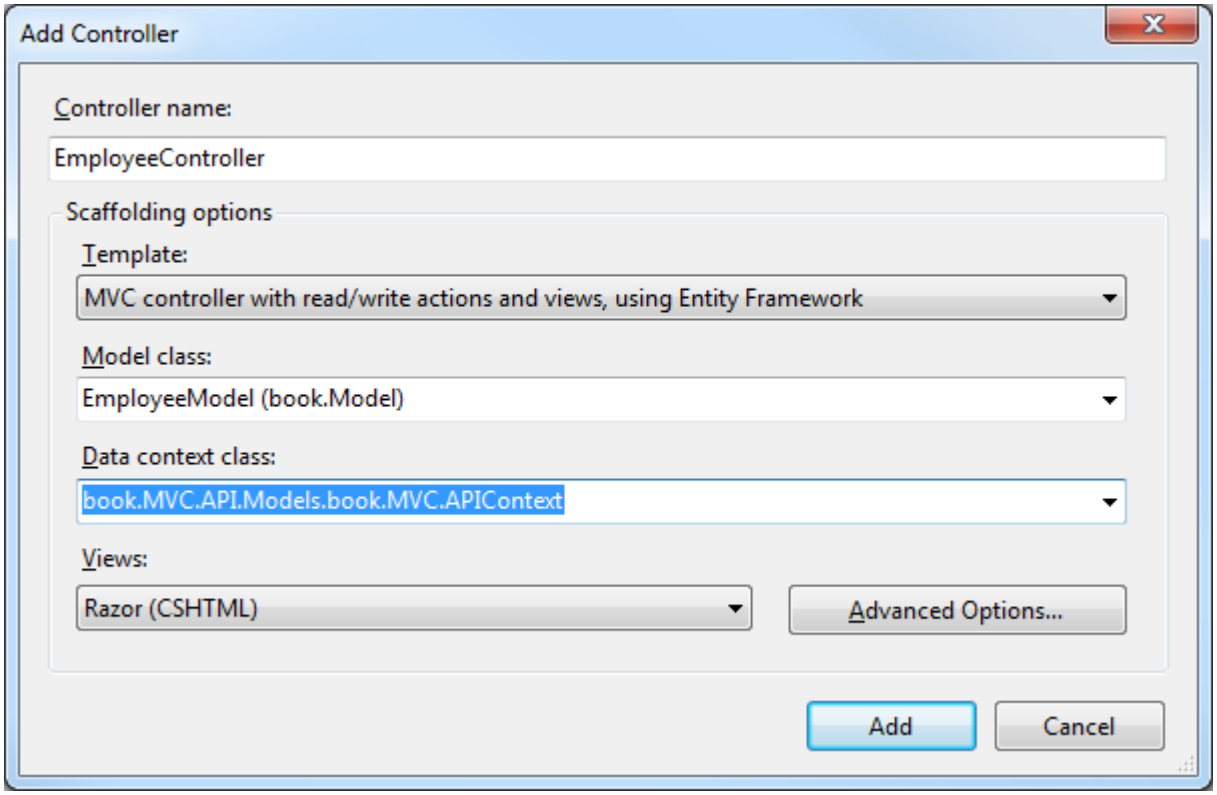

Figure 13: Add Controller Specifying Scaffolding Options

In the **Scaffolding options** section, we can specify the **Template**; this is the MVC template that will be used to generate the controller. We will select **MVC controller read/write actions and views**. We also specify the **Model class** and the **Data context class** for Entity Framework. In this case, we selected **Create a new one**, and were prompted for the name to give it.

With this minimal information, the scaffolding system will generate the full controller and the Entity Framework context.

Let's look at the code generated for the controller.

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Data.Entity;
using System.Data.Entity.Infrastructure;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Web.Http;
using System.Web.Http.Description;
using book.MVC.API1.Models;
using book.Model;

namespace book.MVC.API1.Controllers
```

```
{
 public class EmployeeController : ApiController
 {
  private ApplicationDbContext db = new ApplicationDbContext();

  // GET: api/EmployeeModels
  public IQueryable<EmployeeModel> GetEmployeeModels()
  {
  return db.EmployeeModels;
  }

  // GET: api/EmployeeModels/5
  [ResponseType(typeof(EmployeeModel))]
  public IHttpActionResult GetEmployeeModel(int id)
  {
  EmployeeModel employeeModel = db.EmployeeModels.Find(id);
  if (employeeModel == null)
  {
   return NotFound();
  }

  return Ok(employeeModel);
  }

  // PUT: api/EmployeeModels/5
  [ResponseType(typeof(void))]
  public IHttpActionResult PutEmployeeModel(int id, EmployeeModel
employeeModel)
  {
  if (!ModelState.IsValid)
  {
   return BadRequest(ModelState);
  }

  if (id != employeeModel.Id)
  {
   return BadRequest();
  }

  db.Entry(employeeModel).State = EntityState.Modified;

  try
  {
   db.SaveChanges();
  }
  catch (DbUpdateConcurrencyException)
  {
   if (!EmployeeModelExists(id))
   {
    return NotFound();
```

```
    }
    else
    {
     throw;
    }
   }

   return StatusCode(HttpStatusCode.NoContent);
   }

   // POST: api/EmployeeModels
   [ResponseType(typeof(EmployeeModel))]
   public IHttpActionResult PostEmployeeModel(EmployeeModel employeeModel)
   {
   if (!ModelState.IsValid)
   {
    return BadRequest(ModelState);
   }

   db.EmployeeModels.Add(employeeModel);
   db.SaveChanges();

   return CreatedAtRoute("DefaultApi", new { id = employeeModel.Id },
employeeModel);
   }

   // DELETE: api/EmployeeModels/5
   [ResponseType(typeof(EmployeeModel))]
   public IHttpActionResult DeleteEmployeeModel(int id)
   {
   EmployeeModel employeeModel = db.EmployeeModels.Find(id);
   if (employeeModel == null)
   {
    return NotFound();
   }

   db.EmployeeModels.Remove(employeeModel);
   db.SaveChanges();

   return Ok(employeeModel);
   }

   protected override void Dispose(bool disposing)
   {
   if (disposing)
   {
    db.Dispose();
   }
   base.Dispose(disposing);
   }
```

```
  private bool EmployeeModelExists(int id)
  {
  return db.EmployeeModels.Count(e => e.Id == id) > 0;
  }
 }
}
```

Code Listing 74: Generated Controller Code

As you can see, this controller includes actions for each of the HTTP verbs **GET**, **POST**, **PUT**, and **DELETE**.

It would be nice to have a proxy class to automate calling these methods using the HttpClient library available by using the `Microsoft.Net.Http` package available through NuGet.

Let's go back to our solution, and in the book project, use NuGet to add the `Microsoft.Net.Http` package.

```
PM> Install-Package Microsoft.Net.Http
```

Figure 14: Install Microsoft.Net.Http

Now, in the book project, we want to create a new text template and name it **services.tt**.

In order to access the `book.WebApi` assembly, we will need to add a few references to the template.

```
<#@ assembly name="C:\Program Files (x86)\Reference
Assemblies\Microsoft\Framework\.NETFramework\v4.5\System.Data.Entity.dll"#>
<#@ assembly name="$(SolutionDir)book.Model\bin\debug\book.Models.dll" #>
<#@ assembly
name="$(SolutionDir)packages\EntityFramework.5.0.0\lib\net45\EntityFramewor
k.dll" #>
<#@ assembly
name="$(SolutionDir)packages\Microsoft.AspNet.Mvc.4.0.30506.0\lib\net40\Sys
tem.Web.Mvc.dll" #>
<#@ assembly
name="$(SolutionDir)packages\Microsoft.AspNet.WebApi.Core.4.0.30506.0\lib\n
et40\System.Web.Http.dll" #>
<#@ assembly name="$(SolutionDir)T4Utilities\bin\Debug\T4Utilities.dll" #>
```

Code Listing 75: Assembly References Needed for Services.tt

To make the code a little bit simpler, we will add some basic import directives.

```
<#@ import namespace = "T4Utilities" #>
<#@ import namespace = "System.Web.Http" #>
<#@ import namespace= "System.Web.Mvc" #>
<#@ import namespace= "System.Reflection" #>
```

Code Listing 76: import Directives Needed by Services.tt

Now with the basics for our directives out of the way, let's think about the type of class that we want to generate.

Let's generate a single class with a method for each API action that it finds. We will start by initializing our **`ReflectionHelper`**, and then outputting some boilerplate code:

```
<#
 reflection = new ReflectionHelper();
 NamespaceName = "book";

#>
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Net.Http.Headers;
using Newtonsoft.Json;
using Newtonsoft.Json.Converters;
namespace <#= NamespaceName #>
{
 public class DataAPIServices
 {

<#
 var path = this.Host.ResolvePath (@"..\WebApi\bin\WebApi.dll");
 OutputActionCalls(path);
#>
 }
}
```

Code Listing 77: Initializing the Services template

The bulk of the code generated will be the calls to the various actions that we find in the assembly.

Now let's turn our attention to the feature block for the template. We need to define a couple of properties for the template that we initialized earlier, as well as the **`OutputActionCalls`** method.

```
<#+
 public ReflectionHelper reflection{get ; set;}
 public string NamespaceName {get; set;}
```

```
public void OutputActionCalls(string assemblyPath)
{
  var name = AssemblyName.GetAssemblyName(assemblyPath);
  reflection.LoadAssembly (name);
  var targetType= typeof (ApiController);
  var controllers= reflection.GetTypes()
          .Where (t=> reflection.HasBaseType(t, targetType));
  foreach (var controller in controllers)
  {
     PushIndent("  ");
     this.WriteLine( CurrentIndent+"//" + controller.Name);
     var actions= reflection.GetMethods(controller);
     ProcessGetMethods(actions);
     ProcessPostMethods(actions);
     ProcessPutMethods(actions);
     PopIndent();
  }
  PopIndent();
}
```

Code Listing 78: The OutputActionCalls Method

The properties are defined just like any other property, and the **`OutputActionCalls`** is rather straightforward. We hand most of the heavy lifting over to the **`ReflectionHelper`**. Because we want to handle the various actions differently, we will filter them out by checking their attributes to see which ones handle which verbs.

Note: This does require changing the code generated by the Web API scaffolding. If you don't want to have to make this change, you could also identify which actions go with which verbs by matching the naming conventions on the methods.

Let's go back to the controller and some action selector attributes to the generated code. Because this is generated from scaffolding, we are free to change this code as needed—this was always a onetime code generation.

Action Selector Attributes	
HttpGetAttribute	Restricts access to an action to `HTTP GET` requests
HttpPostAttribute	Restricts access to an action to `HTTP POST` requests
HttpPutAttribute	Restricts access to an action to `HTTP PUT` requests

Action Selector Attributes	
HttpDeleteAttribute	Restricts access to an action to HTTP DELETE requests

Let's look at these various process methods.

```
 private void ProcessGetMethods(IList<MethodInfo> actions)
 {
    this.WriteLine(this.CurrentIndent + "// Get");
    foreach (var action in actions
          .Where (a=>reflection
                .HasAttribute(a, "HttpGetAttribute")))
    {
        this.Write(CurrentIndent + "public " +
        reflection.FormatType( action.ReturnType)+ " "
                  + action.Name);
        var parameterString = reflection
               .GetParameterString (action);
        if (string.IsNullOrWhiteSpace(parameterString))
              parameterString = " string url";
        else
              parameterString += ", string url";
        this.WriteLine("(" + parameterString + ")");
        this.WriteLine(CurrentIndent + "{");
#>
  HttpClient client = new HttpClient();
  var response = client.GetAsync(url).Result;
  var data =
JsonConvert.DeserializeObject<<#=reflection.FormatType(action.ReturnType)#>
>(response.Content.ReadAsStringAsync().Result);
  return data;
<#+
        this.WriteLine(CurrentIndent + "}");
    }
 }
```

Code Listing 79: Supporting Process Methods

The actual implementation of this method is not necessarily very exciting, but there are a couple of things to note. First, we make a call to get the parameters as a string, and we want to add an extra parameter for the URL. We first need to do a little bit of conditional logic to properly handle the cases where there were originally no parameters.

The other item potentially of interest is the use **HttpClient**. For each verb, we will call the appropriate method corresponding to the verb that we are working with. Here, we call **GetAsync**. For **POST**, we will call **PostAsJsonAsync**. For **PUT**, we will call **PutAsJsonAsync**, and for **DELETE**, we will call **DeleteAsync**.

For your reference, here are the remaining methods defined in this template:

```
 private void ProcessPostMethods(IList<MethodInfo> actions)
 {
    this.WriteLine(this.CurrentIndent + "// Post");
    foreach (var action in actions
              .Where (a=>reflection.HasAttribute(a,
                     "HttpPostAttribute")))
    {
       this.Write(CurrentIndent + "public "
              + reflection.FormatType( action.ReturnType)+ " "
              + action.Name);
       var parameterString = reflection
          .GetParameterString (action);
       if (string.IsNullOrWhiteSpace(parameterString))
           parameterString = " string url";
       else
           parameterString += ", string url";
       this.WriteLine("(" + parameterString + ")");
       this.WriteLine(CurrentIndent + "{");
#>
  var client = new HttpClient();
  client.DefaultRequestHeaders.Accept.Add(
   new MediaTypeWithQualityHeaderValue("application/json"));
  var response = client.PostAsJsonAsync(url,
<#=action.GetParameters()[0].Name #>).Result;
  return response;
<#+
       this.WriteLine(CurrentIndent + "}");
    }
 }

 private void ProcessPutMethods(IList<MethodInfo> actions)
 {
    this.WriteLine(this.CurrentIndent + "// Put");
    foreach (var action in actions
             .Where (a=>reflection.HasAttribute(a,
                     "HttpPutAttribute")))
    {
       this.Write(CurrentIndent + "public "
              + reflection.FormatType( action.ReturnType)+ " "
              + action.Name);
       var parameterString= reflection
    .      GetParameterString (action);
```

```
        if (string.IsNullOrWhiteSpace(parameterString))
            parameterString = " string url";
        else
            parameterString += ", string url";
        this.WriteLine("(" + parameterString + ")");
        this.WriteLine(CurrentIndent + "{");
#>
  var client = new HttpClient();
  client.DefaultRequestHeaders.Accept.Add(
   new MediaTypeWithQualityHeaderValue("application/json"));
  var response = client.PutAsJsonAsync(url,
<#=action.GetParameters()[0].Name #>).Result;
  return response;
<#+
        this.WriteLine(CurrentIndent + "}");
    }
 }
#>
```

Code Listing 80: The Supporting Process Methods for the Services template

Note: This is not intended to be a guide to coding Web API calls. For best practices and better understanding the Web API, please refer to the Succinctly Web API book.

An important takeaway from this example should be how easy it would be to modify the template to change the generated code as your best practices improve.

Summary

In this chapter we have covered most topics related to accessing data through reflection. We saw how to load an assembly without locking it, and we explored how to get and filter the types contained in the assembly. We have seen some simple ways to format the name of some complex types so that they look nice in our generated code.

We explored retrieving and filtering the basic components of type that would commonly be used in code generation. There are similar methods available to retrieve the fields and events from a type.

Throughout this chapter we have seen that .NET keeps a wealth of information for us about the code being run. The only drawback is that the code has to be compiled before any of this information is available through reflection.

In the next chapter we will explore ways to retrieve similar information directly from the source code without having to wait for the code to be compiled.

Chapter 8 Working with Code Model

Introduction

The code model is an API provided by Visual Studio that allows us to essentially get much of the same data at design time that reflection provides at run time. This opens up some intriguing possibilities for code generation without having to wait for compiled code.

The code model has a reputation for being difficult to work with. In part this is because it's not very well documented. It also is generally only used when creating plug-ins and extensions for Visual Studio.

We will keep things simple here, following the same path we followed when exploring reflection. Drawing parallels between the two APIs will help us keep things straight.

In this chapter we will work our way through accessing the various pieces of metadata made available. We will look at getting the active project, getting a list of types, and finding all of the details about the types discovered.

Once we have finished this survey of the code model, we will pull all of the pieces together by using the code model to scan the active project, looking for any MVC controllers. For each controller found, we will search for the containing actions and generate JavaScript code to simplify calling the various actions that we find.

Throughout this chapter we will add functionality to a new class called **`CodeModelHelper`** that we can use to simplify the process of accessing code model data. Go ahead and create a new class called **`CodeModelHelper`** in the T4Utilities project that we created in Chapter 4.

Finding the Current Project

Let's start with the constructor for the **`CodeModelHelper`**. To do anything with the code model, we will need the DTE that we looked at in Chapter 4.

To simplify access to this object, we will create a constructor that will require that the host be provided when the helper is created.

```
public DTE VisualStudio { get; set; }
public Project ActiveProject { get; set; }

public CodeModelHelper (ITextTemplatingEngineHost host )
{
   IServiceProvider serviceProvider = (IServiceProvider)host ;
   VisualStudio = serviceProvider
     .GetService(typeof(EnvDTE.DTE)) as DTE;
   ActiveProject = VisualStudio.ActiveDocument
```

```
        .ProjectItem.ContainingProject;
}
```

Code Listing 81: Constructor for the CodeModelHelper call

The last line is a subtle way to get the active project. In this case the active document will be the template itself. If you need to work with a different project, you can use the methods outlined in Chapter 4 for navigating a solution.

Get a List of Types

Getting a list of types is a more involved process than with reflection. One complication is that getting a list of classes is completely separate from getting a list of interfaces. Let's start with a list of classes.

```
public IList<CodeClass> GetTypes ()
{
   return GetCodeItemsInProject<CodeClass>
            (ActiveProject.ProjectItems);
}
```

Code Listing 82: Get a List of Types from the Code Model

This works very similar to the **`GetTypes`** that we defined in **`ReflectionHelper`**. We can filter by any property in the **`CodeClass`** interface.

```
public IList<CodeClass> GetTypesInNamespace(string nameSpace)
{
   return GetCodeItemsInProject<CodeClass>
             (ActiveProject.ProjectItems)
        .Where(t=>t.Namespace.Name == nameSpace).ToList();
}
```

Code Listing 83: Filtering Types by NameSpace

Unfortunately, **`IsInterface`** is not one of the properties.

We will need a new method to get the interfaces. Fortunately, it should look familiar:

```
public IList<CodeInterface> GetInterfaces()
{
   return GetCodeItemsInProject<CodeInterface>
             (ActiveProject.ProjectItems);
}
```

Code Listing 84: Get a List of Interfaces from the Code Model

The only difference between **`GetTypes`** and **`GetInterfaces`** is the generic parameter passed to the **`GetCodeItemsInProject`** method.

Let's delve into the implementation of our **`GetCodeItemsInProject`** method to see how this is used.

```
private List<T> GetCodeItemsInProject<T>(ProjectItems items)
                 where T : class
{
   var classes = new List<T>();
   foreach (ProjectItem item in items)
   {
      if (item.ProjectItems != null
            && item.ProjectItems.Count > 0)
      {
         classes.AddRange(GetCodeItemsInProject<T>
                (item.ProjectItems));
      }
      if (item.FileCodeModel != null)
      {
         classes.AddRange(GetCodeItemsInCodeModel<T>
             (item.FileCodeModel.CodeElements));
      }
   }
   return classes;
}
```

Code Listing 85: Get Code Items in Project

This method is not really as complicated as it initially looks. The recursive call to **`GetCodeItemsInProject`** is to handle cases where a project item may have nested projected items. This happens with the code-behind for a web form or the output of a template. This also happens when we add a folder to the project.

The true magic happens in the **`GetCodeItemsInCodeModel`** method.

```
private List<T> GetCodeItemsInCodeModel<T>(CodeElements elements)
           where T : class
{
   var classes = new List<T>();
   foreach (CodeElement element in elements)
   {
      if (typeof(T).Name == "CodeClass")
         if (element.Kind == vsCMElement.vsCMElementClass)
            classes.Add(element as T);
         if (typeof(T).Name == "CodeInterface")
         {
            if (element.Kind == vsCMElement.vsCMElementInterface)
               classes.Add(element as T);
         }
         var members = GetCodeElementMembers(element);
         if (members != null)
            classes.AddRange(GetCodeItemsInCodeModel<T>(members));
```

```
        }
        return classes;
    }
```

Code Listing 86: GetCodeItemsInCodeModel

The **if** statements in the middle can be expanded if you are searching for anything other than a **CodeInterface** or **CodeClass**. As written, these are the only type generic arguments suitable for the call to **GetCodeItemsInProject**.

We won't have to worry so much about recursion and looping to find all the places where code might hide in the project beyond this point. From here on out we will be dealing in the context of a **CodeInterface** or a **CodeClass**. Unfortunately, in the code model API, there is not a common base class for these two entities, even though their interfaces are nearly identical. So we will need overloads for each of the methods we are about to discuss. One for **CodeClass**, and one for **CodeInterface**. Through the remainder of this chapter we will talk about types. This should not be confused with **System.Type**, which we used in reflection, but is used instead to collectively refer to interfaces and classes in **CodeModel**.

Get the Properties of a Type

Getting the properties of a type is very straightforward.

```
public IList<CodeProperty> GetProperties(CodeClass name)
{
    return name.Members.OfType<CodeProperty>().ToList();
}
```

Code Listing 87: Code 1: Getting the Properties of a Type

Once we get the properties, we have access to some standard properties that we will need:

- **Name**
- **Type**
- **Access**
- **Attributes**

Name and **Type** will work pretty much as you would expect. **Access** is based on an **enum** that you may need a special decoder ring to make sense out of.

Member name	Description
vsCMAccessAssemblyOrFamily	Element has assembly or family access; proposed new language feature

Member name	Description
vsCMAccessDefault	Won't show up as a code reference; only provided to support adding code to a project
vsCMAccessPrivate	Regular private
vsCMAccessProject	Regular internal
vsCMAccessProjectOrProtected	Combination of protected and internal
vsCMAccessProtected	Regular protected
vsCMAccessPublic	Regular public
vsCMAcessWithEvents	Indicates that a variable declaration will respond to raised events

We may also often be interested in whether or not the property has a getter or setter. To check this, we check to see if the **Getter** or **Setter** properties are **null**. The actual value for these properties will be a **CodeFunction** that implements the get or set. We aren't really interested in how these functions are implemented, only whether or not they exist.

Get the Methods of a Type

The **GetMethods** method is very similar to the **GetPropeties** method.

```
public IList<CodeFunction> GetMethods(CodeClass name)
{
   return name.Members.OfType<CodeFunction>().ToList();
}
```

Code Listing 88: Get the Methods of a Type

We have access to the same data about methods that we could get about properties. We can also get the parameters and a list of overloaded functions.

The **Parameters** property exposes a **CodeElements** object holding all of the associated parameters.

We can easily get the parameters:

```
public IList<CodeParameter> GetParameters (CodeFunction name)
{
   return name.Parameters.OfType<CodeParameter>().ToList();
}
```

Code Listing 89: Get the Parameters to a Method

Or, we can format the list of parameters as a string.

```
public string GetParameterString(CodeFunction action)
{
   var paramList = "";
   var parameters = GetParameters(action);
   foreach (var paramater in parameters)
   {
      paramList += paramater.Name;
      paramList += ", ";
   }
   char[] charsToTrim = { ',' };
   paramList.TrimEnd(charsToTrim);
   return paramList;
}
```

Code Listing 90: Get the Parameters to a Method Formatted to be Inserted to Generated Code

Get the Attributes of a Type, Method, or Property

We can easily determine if any of these code structures has a particular attribute.

```
public bool HasAttribute(CodeElements attributes, string attr)
{
    return attributes.OfType<CodeAttribute>()
          .Any(attribute => attribute.Name == attr);
}
```

Code Listing 91: Determine whether or not a Group of CodeElements has a Given Attribute

This code can be called like this:

```
var getActions = actions.Where(
  a => code.HasAttribute(a.Attributes, "HttpGet"));
```

Note: We are not passing the actual name of the attribute—just what would be added in code to add the attribute.

The only problem here is that this does not easily allow us to retrieve attributes that were added at higher levels in the inheritance hierarchy.

If your design relies on inheriting attributes from base classes, you will need to extend this method to walk the inheritance tree.

We can get a list of all of the base classes like this:

```
public List<CodeClass> GetBaseClasses(CodeElements bases)
{
   var allBases = new List<CodeClass>();
   foreach (var b in bases.OfType<CodeClass>())
   {
      allBases.Add(b as CodeClass);
      allBases.AddRange(GetBaseClasses
         ((b as CodeClass).Bases));
   }
   return allBases;
}
```

Code Listing 92: Get all of the Bases Class for a Type

Once we have the list of base classes, we will need to know where we are getting the attributes from to know what to check at each level.

The easiest case is to check to see if a class has a particular attribute.

```
public bool ClassHasAttribute (CodeClass currentClass,
                  string attr)
{
   var baseClasses = GetBaseClasses(currentClass.Bases);
   baseClasses.Add(currentClass);
   var result = baseClasses.Any
           (s => HasAttribute(s.Attributes, attr));
   return result;
}
```

Code Listing 93: Check to See if a Class has an Attribute

We can craft **`PropertyHasAttribute`** and **`MethodHasAttribute`** following the same pattern, except these methods will also need to know which property or method to check:

```
public bool PropertyHasAttribute (CodeClass currentClass,
          CodeProperty currentProperty, string attr)
{
   var baseClasses = GetBaseClasses(currentClass.Bases);
   var properties = new List<CodeProperty>();
   baseClasses.ForEach(b => GetProperties(b)
          .Where(p => p.Name == currentProperty.Name));
   return properties.Any(p =>
```

```
            HasAttribute(p.Attributes, attr));
    }
```

Code Listing 94: Check to See if a Property has an Attribute

And:

```
    public bool MethodHasAttribute(CodeClass currentClass,
        CodeFunction currentProperty, string attr)
    {
        var baseClasses = GetBaseClasses(currentClass.Bases);
        var methods = new List<CodeFunction>();
        baseClasses.ForEach(b => GetMethods(b)
            .Where(p => p.Name == currentProperty.Name));
        return methods.Any(p =>
               HasAttribute(p.Attributes, attr));
    }
```

Code Listing 95: Check to See if a Method has an Attribute

In addition to walking the inheritance tree, we may often find that we need to simply know if a given class has a particular base class. We can write a **`HasBaseClass`** method like this:

```
    public bool HasBaseClass(CodeElements bases,
                string baseClass)
    {
        return GetBaseClasses(bases)
              .Any (b => b.Name == baseClass);
    }
```

Code Listing 96: Determine if a Class has a Particular Base Class

Pulling it All Together

Let's go back to the MVC API controller that we created in Chapter 7. You may recall that we used the scaffolding to create an **`EmployeeController`** using the template to create read/write actions with Entity Framework.

You will often need to call these actions through JavaScript from the client side. Often this will be done using jQuery. While this is not an overly complex task, it is one where the best practices can often change. Since we can easily discover all the potential methods that can be called, this makes it a good candidate for code generation.

Let's start by adding a new template to the **book.MVC.Api** project that we created in Chapter 7. Name this template **controllerscript.tt**.

We start by adding the supporting directives.

```
<#@ template debug="true" hostSpecific="true" #>
<#@ output extension=".js" #>
<#@ assembly Name="System.Core.dll" #>
<#@ assembly name="System.Web.Mvc" #>
<#@ assembly name="EnvDTE" #>
<#@ assembly name="EnvDTE100" #>
<#@ import namespace="EnvDTE" #>
<#@ import namespace="EnvDTE100" #>
<#@ import namespace="System.Linq" #>
<#@ import namespace="System.Collections.Generic" #>
<#@ import namespace="System.Web.Mvc" #>
<#@ assembly name="$(SolutionDir)T4Utilities\bin\Debug\T4Utilities.dll" #>
<#@ import namespace = "T4Utilities" #>
```

Code Listing 97: Assembly and Import Directives for the controllerscript Template

The assembly and import directives for **EnvDTE** are key to being able to use the code model. Also note that we added the T4Utilities to get the **CodeModelHelper** that we have been working on.

A good deal of this template will be static boiler plate code with some dynamic code in the middle. Let's add the static header code next.

```
var Controllers = {
 relativePath: "",
 SetRelativePath : function(relPath){
  Controllers.relativePath = relPath;
 }
```

Code Listing 98: Static Header Code

Now we need to initialize the **CodeModelHelper** and add the calls to generate the dynamic code.

```
<#
 code = new CodeModelHelper(this.Host);
 CallActions();
#>
```

The code property will be defined a little later in the feature block, along with the **CallActions** method.

The **CallActions** method will generate code that will be added to the **Controllers** object that we are building up by looping through the controllers and their associated actions, and providing a handy method to call the discovered method.

After we have declared the **Controllers** object, we have some more static JavaScript code to define.

```
(function (AjaxRequest, undefined) {
 AjaxRequest.Post = function (jsonData, url, successCallback,
errorCallback) {
  $.ajax({
  url: url,
  type: "POST",
  data: jsonData,
  datatype: "json",
  contentType: "application/json charset=utf-8",
  success: function (data) {
   successCallback(data);
  },
  error: function (jqXHR, textStatus, errorThrown) {
   if (errorCallback) {
    errorCallback();
   }
  }
  });
 };


 AjaxRequest.Get = function (jsonData, url, successCallback, errorCallback)
{
  $.ajax({
  url: url,
  type: "GET",
  data: jsonData,
  datatype: "json",
  contentType: "application/json charset=utf-8",
  success: function (data) {
    successCallback(data);
  },
  error: function (jqXHR, textStatus, errorThrown) {
   if (errorCallback) {
    errorCallback();
   }
  }

  });
 };

}(window.AjaxRequest = window.AjaxRequest || {}));
```

Code Listing 99: More Definition

Note: A full explanation of what this JavaScript code does is outside the scope of this book, but you can get all the details you need by downloading and reading the JavaScript Succinctly and jQuery Succinctly books.

Now we are ready to add the feature block. We will start by declaring the code property and the **`CallActions`** method.

```
public CodeModelHelper code{get ; set;}
public void CallActions()
{
   PushIndent(" ");
   var classes = code.GetTypes()
         .Where (c=> code.HasBaseClass(c.Bases, "ApiController"));
   if(classes.Count() > 0) WriteLine(CurrentIndent+ ",");
   foreach (var currentClass in classes)
   {
      var controllerName =
          currentClass.Name.Replace("Controller","");
      WriteLine(CurrentIndent+ "'" + controllerName +"' :");
      WriteLine(CurrentIndent+ "{");
      PushIndent(" ");
      var actions = code.GetMethods(currentClass)
                    .Where(func =>
                        func.Type.CodeType.Name == "ActionResult"
                        || code.HasBaseClass(func.Type.CodeType.Bases,
                          "ActionResult")).ToList();;
      foreach (var action in actions)
      {
         Write(CurrentIndent+ "'{1}URL' : '{0}/{1}'",
               controllerName, action.Name);
         if(action != actions.Last())
            WriteLine(CurrentIndent+ ",");
      }
      ProcessActions (actions, currentClass, "Get");
      ProcessActions(actions, currentClass, "Post");
      PopIndent();
      WriteLine("\r\n"+CurrentIndent+ "}");
    }
    PopIndent();
    WriteLine(CurrentIndent+ "};");
}
```

Code Listing 100: Add Feature Block

In the **`CallActions`** method, we start by getting a list of all of the classes that are derived from controller. We ignore all other classes in the project. We strip off the word “Controller” from the name of the class, because this will not actually be included in references to the class.

Now that we have our list of controllers, we loop them, looking for actions. In this case, we are looking for the methods that are return types of any type derived from **`ActionResult`**.

We will process this list of actions in two steps. In the **`foreach`** loop, we build up a list of URLs to the various actions.

After the loop, the two calls to the **`ProcessActions`** method handle generating the method calls.

```
public void ProcessActions (IList<CodeFunction> actions,
     CodeClass controller, string verb)
{
    var filteredActions = actions.Where(
        a => code.HasAttribute(a.Attributes, "Http" + verb));
    if(filteredActions.Any())
       WriteLine(CurrentIndent+ ",");
    foreach (var action in filteredActions)
    {
       WriteAjaxWrapper(verb,action, controller);
       if(action != filteredActions.Last())
          WriteLine(CurrentIndent+ ",");
    }
}
```

Code Listing 101: Calling the ProcessActions Method

Here we filter based on which methods have the appropriate attribute for the requested verb. Each action should have either an **`HttpGet`** or an **`HttpPost`** attribute. The **`WriteAjaxWrapper`** method makes the call to **`Ajax`**:

```
public void WriteAjaxWrapper(string method, CodeFunction action,
         CodeClass controller)
{
   var paramList = code.GetParameterString(action);
   var parameters = code.GetParameters (action);
   WriteLine(CurrentIndent+ "'{0}{1}' : "
          +"function({2}successCallback, errorCallback) {{",
           action.Name, method,paramList);
   PushIndent(" ");
   WriteLine(CurrentIndent+ "AjaxRequest.{0}({1}"
          +"Controllers.relativePath + Controllers.{2}.{3}URL, "
          +"successCallback, errorCallback);",
          method, string.IsNullOrEmpty(paramList) ? "undefined,":
          paramList, controller.Name.Replace("Controller",""),
          action.Name);
    PopIndent();
    Write(CurrentIndent+ "}");
}
```

Code Listing 102: The WriteAjaxWrapper Method

Given the **`EmployeeController`** that we generated in Chapter 7, this template output will look like this:

```
var Controllers = {
 relativePath: "",
 SetRelativePath : function(relPath){
  Controllers.relativePath = relPath;
 }
 ,
 'Employee' :
 {
  'IndexURL' : 'Employee/Index' ,
  'DetailsURL' : 'Employee/Details' ,
  'CreateURL' : 'Employee/Create' ,
  'CreateURL' : 'Employee/Create' ,
  'EditURL' : 'Employee/Edit' ,
  'EditURL' : 'Employee/Edit' ,
  'DeleteURL' : 'Employee/Delete' ,
  'DeleteConfirmedURL' : 'Employee/DeleteConfirmed' ,
  'IndexGet' : function(successCallback, errorCallback) {
   AjaxRequest.Get(undefined, Controllers.relativePath +
    Controllers.Employee.IndexURL, successCallback,
    errorCallback);
  } ,
  'DetailsGet' : function(id, successCallback, errorCallback) {
   AjaxRequest.Get(id, Controllers.relativePath +
    Controllers.Employee.DetailsURL, successCallback,
    errorCallback);
  } ,
  'CreateGet' : function(successCallback, errorCallback) {
   AjaxRequest.Get(undefined, Controllers.relativePath +
    Controllers.Employee.CreateURL, successCallback,
    errorCallback);
  } ,
  'EditGet' : function(id, successCallback, errorCallback) {
   AjaxRequest.Get(id, Controllers.relativePath +
    Controllers.Employee.EditURL, successCallback,
    errorCallback);
  } ,
  'CreatePost' : function(employeeModel, successCallback,
      errorCallback) {
   AjaxRequest.Post(employeeModel, Controllers.relativePath
    + Controllers.Employee.CreateURL, successCallback,
    errorCallback);
  } ,
  'EditPost' : function(employeeModel, successCallback,
      errorCallback) {
   AjaxRequest.Post(employeeModel, Controllers.relativePath
```

```
    + Controllers.Employee.EditURL, successCallback,
    errorCallback);
  } ,
  'DeletePost' : function(id, successCallback, errorCallback) {
   AjaxRequest.Post(id, Controllers.relativePath +
     Controllers.Employee.DeleteURL, successCallback,
     errorCallback);
  } ,
  'DeleteConfirmedPost' : function(id, successCallback,
      errorCallback) {
   AjaxRequest.Post(id, Controllers.relativePath +
     Controllers.Employee.DeleteConfirmedURL,
     successCallback, errorCallback);
  }
 }
};

(function (AjaxRequest, undefined) {
 AjaxRequest.Post = function (jsonData, url, successCallback,
      errorCallback) {
  $.ajax({
  url: url,
  type: "POST",
  data: jsonData,
  datatype: "json",
  contentType: "application/json charset=utf-8",
  success: function (data) {
   successCallback(data);
  },
  error: function (jqXHR, textStatus, errorThrown) {
   if (errorCallback) {
    errorCallback();
   }
  }
  });
 };


 AjaxRequest.Get = function (jsonData, url, successCallback,
     errorCallback) {
  $.ajax({
  url: url,
  type: "GET",
  data: jsonData,
  datatype: "json",
  contentType: "application/json charset=utf-8",
  success: function (data) {
    successCallback(data);
  },
  error: function (jqXHR, textStatus, errorThrown) {
```

```
    if (errorCallback) {
     errorCallback();
    }
   }
   });
 };

}(window.AjaxRequest = window.AjaxRequest || {}));
```

Code Listing 103: Template Output

Having such a script can be very helpful with projects with a large collection of controllers. It is also much easier to change the template generating this code than it is to change the generated code as best practices change.

Summary

In this chapter we have covered most topics related to accessing data through the code model. We saw an easy way to get to the project hosting the template. We explored how to get and filter the types contained in this project.

We explored retrieving and filtering the basic components for these types that would commonly be used in code generation. Visual Studio provides a wealth of information for us about the code being written. While the code model is more commonly used to manipulate existing code in a project, and is intended to be used for plug-ins and extensions, we have seen how it can be used to provide the data we need to generate code.