IN THIS CHAPTER

- Discovering where templating for Visual Studio originated
- Installing the required workload
- Getting your environment ready for T4 use
- Using T4 like a language

Online Chapter 2
Transforming Text Templates

T4 (Text Template Transformation Toolkit) is a code-generation toolkit built into Visual Studio. It is a built-in language used to make the default ASP.NET and Windows Forms files in C# or VB (for example, when you use the Add New Item feature). Code generation increases developer productivity, decreases bugs in final versions, and makes people happy.

T4 is a starting point for a project, providing seamless redirection and logical formatting of what the user (in this case, the developer) actually sees. It is the template, just as its name suggests. T4 is also a big topic, like so many others in this book, but you should be able to get started creating code generation from the material here. Getting your environment ready and understanding the concepts are the most important steps.

Getting to Know T4

In this chapter, you discover T4 as a code-generation tool, but it can do more things that are beyond the scope of this book. Many things about computer science are elegant, and the current drive toward meta-programming and the use of higher-and-higher-level languages to solve tougher and tougher problems is one

of them. At its most complex, T4 can review information about the kind of projects you build and determine what kind of templates you need to get the job done. That's cool, but this chapter sticks with analysis and lets T4 do the dirty work.

Looking back at the DSL Tools

One of the first ways that Microsoft tried to implement code generation was through the Environment Design Time Environment (EnvDTE), which is still around, and by giving developers access to the code files themselves. The bare-bones approach provided by EnvDTE isn't really what Microsoft was after, though. The goal of templating was to provide a language of languages, which developers could use to model their own development style. The code generation is almost secondary.

Microsoft's next shot to solving the code-generation problem was the Domain-Specific Language (DSL) Tools. The idea here was to use the EnvDTE and other markup languages to describe a very high-level idea in a project — for example, a new ASPX page. Not any specific ASPX page: *any* ASPX page.

That description can then become the starter code that you see when you ask for a new ASPX page. When you describe a higher-level concept like requesting a new ASPX page, the DSL implements a concrete version, which you then can edit.

Looking ahead to what it became

Because T4 is designed to generate anything, it can emit any kind of file. If you think about it, it all makes sense. If an entity in your domain model is a product website, then T4 would have to emit HTML, CSS, and JavaScript when a new product is added to the database.

The same thing applies to the language that is emitted as part of the project. T4 doesn't care what code you are emitting. It just wants to stamp out what matches the model you define for it. If you set up the model to use C#, that's fine. VB? Fine. T4 is just generating and giving text an extension.

Additionally, you can write your code inside the T4 file in C# or VB.NET. This is the code that makes any decisions about the output, accesses data sources that might be needed to generate the code, and so on. It follows the usual .NET rules.


REMEMBER

Even though T4 was developed as an addition to Visual Studio 2008 and is baked into verstion 2012, you can use it in all previous versions back to 2005. o use T4 with Visual Studio 2017, you must install the correct workload, as described in the "Installing the Visual Studio Extension Development workload" section that follows.

Figuring Out When to Use T4

Now that it is more or less clear what T4 is designed to do, the next step is figuring out when to use it. There are two main applications, as well as many lesser ones out there; the following sections are not all-inclusive. The main ones are replacing repetitive coding and building code based on outside data. You can find additional information on this topic at `https://docs.microsoft.com/en-us/visualstudio/modeling/code-generation-and-t4-text-templates`.

Installing the Visual Studio Extension Development workload

Visual Studio 2017 can have built-in support for T4, but only if you include the required workload, which isn't installed by default. *Workloads* make it possible to keep Visual Studio small, yet provide a huge range of functionality. Earlier versions of Visual Studio just kept getting bigger and bigger — to the point where Visual Studio became unwieldy. The following steps show how to install the Visual Studio Extension Development workload. You can also use these steps when installing other workloads that Visual Studio supports.

1. **Open the Programs and Features window of the Windows Control Panel.**

   You see a list of applications installed on your system.

2. **Select the Visual Studio 2017 entry and click Change.**

   The Visual Studio installer starts, and you see a window telling you which version of Visual Studio you have installed.

3. **Click Modify for your installed version.**

   This chapter assumes that you have the Visual Studio Community 2017 version installed. The window shown in Figure OC2-1 appears. Note that the Workloads entry is underlined so that you see the available workloads.

4. **Check the Visual Studio Extension Development workload (or whatever workload you need to install) and click Modify in the lower-right corner of the window.**

   The installer begins the installation process. Progress bars show how the installation is progressing. When the process is complete, you can click Launch to start Visual Studio 2017 with the new feature installed.

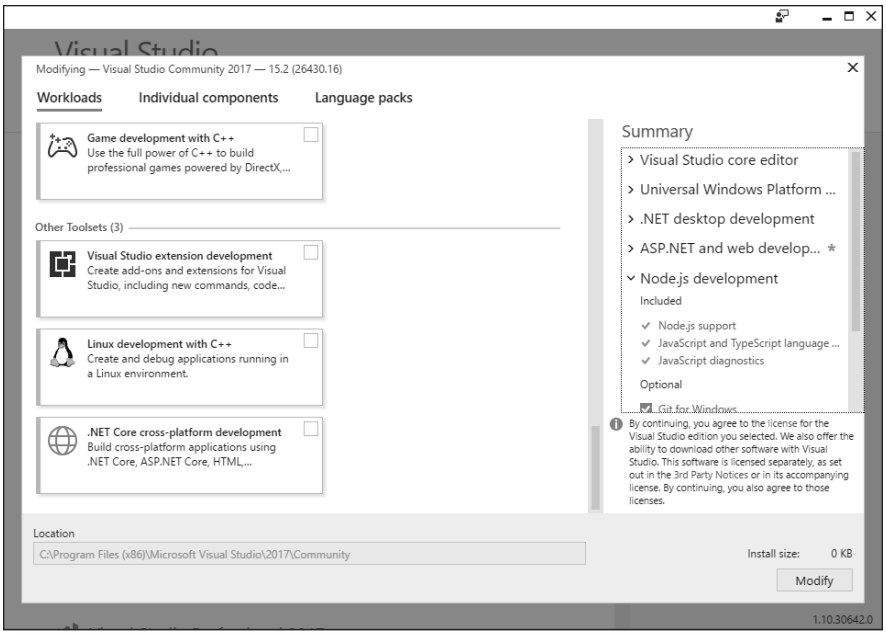

FIGURE OC2-1: Modify your Visual Studio setup to include new workloads as needed.

Replacing repetitive coding

As snippets do at the class level, T4 can replace repetitive coding at the file level. If you think about it, you realize that there is a whole host of code that you shouldn't have to write over and over and over, but nonetheless you do. Take something simple, such as an HTML page. Every page must have certain elements, by the HTML 4.0 Transitional standard:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title></title>
</head>
<body>

</body>
</html>
```

Those are required elements! Can't they just be . . . *assumed*? As it turns out, no, they can't, because some of the elements must have data inserted inside them. This is just the starting point — exactly what T4 is good at providing.

Setting Up the Environment

You need to know a few details relating to the environment.

Changing the security settings

You can relax the security requirements for T4. If you write T4 documents and don't sign them (which is often the case), you will receive a dialog box warning you that you might be generating some dangerous code. You can change the Show Security Message option to False in the Text Templating section to avoid this message, as shown in Figure OC2-2.

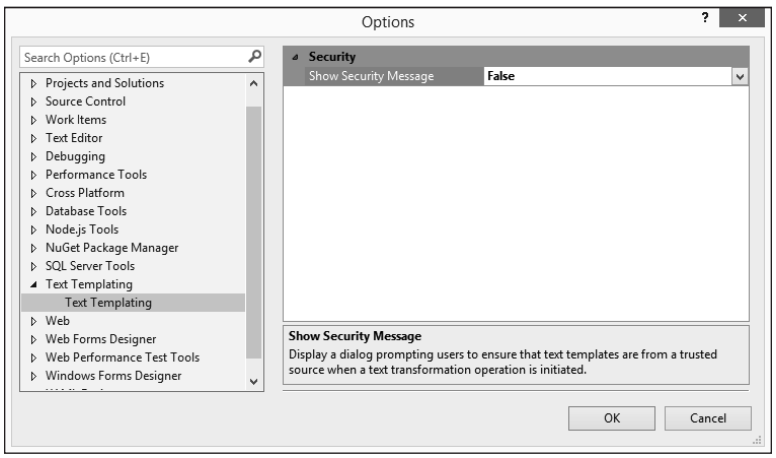

FIGURE OC2-2: The security setting in the Tools/Options box.

REMEMBER

Because T4 files are essentially macros, this warning makes sense. It is possible for someone to write malicious code and execute it on your development workstation. If you turn off the security during development of the template, be sure to turn it back on when you are back to your daily scheduled programming.

Creating a template from a text file

The simplest TT file is just a text file, edited in Notepad, with some basic commands in it called Directives. Follow these steps to create a template from that basic text file:

1. **Open Notepad.**
2. **In the new file, add the following lines of code.**

```
<#@ output extension=".cs" #>

public class TestClass
{
}
```

   The lines are cryptic, but they will make sense in a second.

3. **Save the file in your Documents folder as** `GenerateClass.tt`**.**

   Make certain you set the Save as Type field to All Files (*.*) in Notepad's Save As dialog box. It should look like Figure OC2-3.

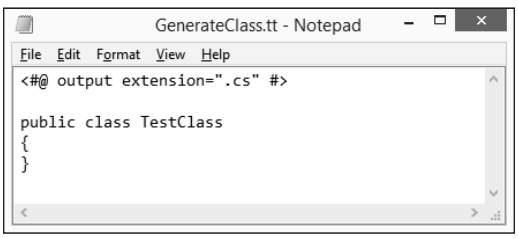

FIGURE OC2-3: The world's simplest TT file.

4. **Create a new class library named** TestTT **in Visual Studio.**
5. **Right-click the project file, select Add, and then select Existing Item.**

   You see a file selection dialog box.

6. **Select the** `GenerateClass.tt` **file that you created in Step 3.**

   You might have to select All Files from the Files of Type drop-down menu if `.tt` isn't one of the file types available.

7. **Click OK.**

   If you didn't set your security appropriately, you get the security warning discussed in the section "Changing the security settings," earlier in this chapter.

   Notice that in the file tree in Solution Explorer, under the TT file, there is now a CS file with your code in it, as shown in Figure OC2-4.

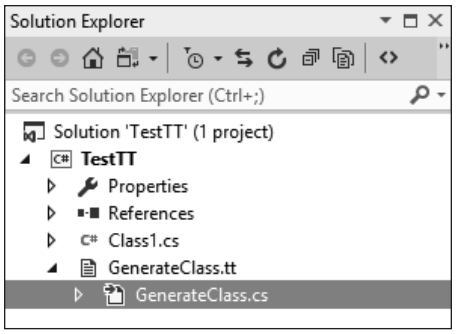

FIGURE OC2-4: The generated code file.

Using Some of the T4 Directives

Clearly, this is a quintessential Hello World example. The magic of T4 is complex.

One of the things that makes T4 work well are the directives built into the language. These commands are just like shell commands. They have parameters and perform certain tasks on the file.

Setting the output

The first directive you see in the example in the earlier section "Creating a template from a text file" is the `output` directive. To create a C# file, you use the `.cs` extension. You could emit anything, though — text, HTML, VB, or even a custom format that you need for your project, such as

```
<#@ output extension=".bill" #>
```

Another parameter for output is Encoding, which is exactly what it sounds like. If you need to represent special characters in your files, you need to specify your encoding. Options include

- Default
- ASCII
- BigEndianUnicode
- Unicode
- UTF32
- UTF7
- UTF8

Configuring a template

`Template` is a directive that specifies the various properties of the T4 text itself. The goal is to affect how the parsing engine interprets the code in the template itself. The options include

- **language:** This is the language for the code that does the work. You can emit anything, but the code in the template has to be VB or C#.
- **inherits:** This is a class that derives from `TextTransformation` to be used as the base class.
- **culture:** This sets the `System.Globalization` culture. You know, like en-US or en-GB.
- **debug:** Just like the ASP.NET debugger, this sets the return of debug symbols.
- **hostspecific:** This is for use with custom hosts. You'll run into it if you are writing for custom hosts.

Including includes

If you want to include the contents of a file somewhere in your template, just drop the `include` directive in there and the name of the file in a file attribute.

```
<#@ include file="c:\specialsource.cs" #>
```

Importing items and assemblies

The `import` and `assembly` directives assist with writing code in a template. The example in the previous section emits text, but in the real examples that you find in MSDN, you write C# or VB code to modify the text.

If you want that code to use .NET Framework constructs, you need to reference the `assembly` and `import` the namespace. For instance, if you are going to get values from a file in your template, you need the `System.IO` library.

First you need to reference the `assembly`, unless you are certain the `assembly` will be referenced in the project. Then you should `import` the namespace so that you don't have to reference items via fully qualified names, like this:

```
<#@ assembly name="System.IO.DLL" #>
<#@ import namespace="System.IO" #>
```

Then you can reference the file maintenance classes inside the code of the template. You reference code using the `<#` and `#>` statements.