Testing in the .NET Maze

Tom Arnold, tom@xtenddev.com
Presented at STAR East 2002
Wednesday, May 15th, 2002

Introduction

Microsoft has created a new environment that promises to ease the process of software development. As test engineers, it is up to us to figure out how the new .NET Framework applies to us, and our efforts, in testing the resulting applications.

Because of the enormity of the .NET Framework this paper moves from the large, high-level view of .NET down to specific examples in ASP.NET. That is, *WebForms* are explored instead of *WinForms*, two pieces of the puzzle that will become clearer as you read on.

This paper expands on my May 15th presentation at the 2002 STAR East ("Software Test, Analysis and Review") conference held in Orlando, Florida. For more information about STAR visit www.sqe.com.

Topics

In this paper I will introduce you to a high-level view of what Microsoft .NET is, issues – such as migrating to the .NET solution – that could result in bugs, inherent challenges for software test engineers, approaches to testing deployed projects, and where to find more information to continue to learn about testing .NET applications.

Author/Speaker Background

My background is in software development and automated testing. I started my professional career in the software industry as a test engineer in the Seattle, Washington, area in 1990. Since that time I've continued to be involved in software testing (focusing mostly on test automation), development (C, C++, VB, and most recently Java), and managing software development projects.

I started using Microsoft .NET in August 2001 and have found it to bring some very exciting things to the table for developers. I was happy to see that Microsoft kept software test engineers in mind as they created this new solution, as you'll see.

Why .NET

Why Microsoft .NET? Sun Microsystems is one reason. Sun has been building on its Java solutions since 1995 when Java was first released, and they've been running hard for these past 7 years. Their solution is J2EE (Java 2 Enterprise Edition) that allows the Java language to work within multiple operating systems as well as with many databases. Sun has also come out with J2SE (Java 2 Standard Edition) and J2ME (Java 2 Micro Edition). These solutions allow Java users to work in simple web environments (J2SE), Enterprise (J2EE), and with handheld devices (J2ME). Very exciting, and all bundled up in a very nice package with many developers excited about the prospects.

Enter Microsoft, a company that has long dominated the software development industry, suddenly seeing some of its development supporters casting their gazes upon Sun Microsystems' solutions to Internet applications and multiple platform support. Sun, with its popular Java programming language that is secure and easy to use compared to C++.

Microsoft took the next logical step in the evolution of their development approaches and brought together many of its development solutions to be placed into a bucket named ".NET." Does that mean .NET is entirely new? No. Microsoft has taken all of its existing functionality, added in some additional bits (albeit some rather large and important bits), and pulled it into a solution that will compete (very well) with Sun Microsystems.

This is a good thing, why? Because Microsoft .NET brings a new focus on how to approach Windows and Internet development, an approach that not only opens up the architecture to allow a host of new languages to be supported in the .NET development environment, but operating systems as well. The .NET Framework is setting the scene to allow applications to be developed and deployed in many environments, and additional support for such deployment created by third party vendors.

.NET Framework

This framework, at first glance, seems more like a huge puzzle or maze. Just when we're getting things figured out, yet another enigma in the software industry presents itself, this time in the form of Microsoft .NET.

Fear not, .NET is not so overwhelming after all. Remember that it's an encapsulation of a number of pre-existing Microsoft technologies with a few new ones thrown in for good measure. This, as well as a common thread – or framework – that pulls it all together, is what makes up Microsoft .NET.

Common Language Specification

The top layer shown in **Figure 1** (on the following page) illustrates the default languages already supported by .NET: Visual Basic, C++, JavaScript (known at Microsoft as "JScript"), and Microsoft's new C# (pronounced, "C Sharp").

Microsoft Visual Basic .NET

Visual Basic now offers full object-oriented language features, including implementation inheritance. It also allows developers to create highly scalable code with explicit free threading and highly maintainable code with the addition of modernized language constructs like structured exception handling.

Microsoft Visual C# .NET

Microsoft also created its own Java-like language called C#. C# is very much like Java in that it handles all garbage collection, provides security, and is fairly easy to use compared to C++. It was built from the ground up with the .NET Framework in mind and is a modern, object-oriented, type-safe language. C# "is designed to bring rapid development to the C++ programmer without sacrificing the power and control that have been a hallmark of C and C++."

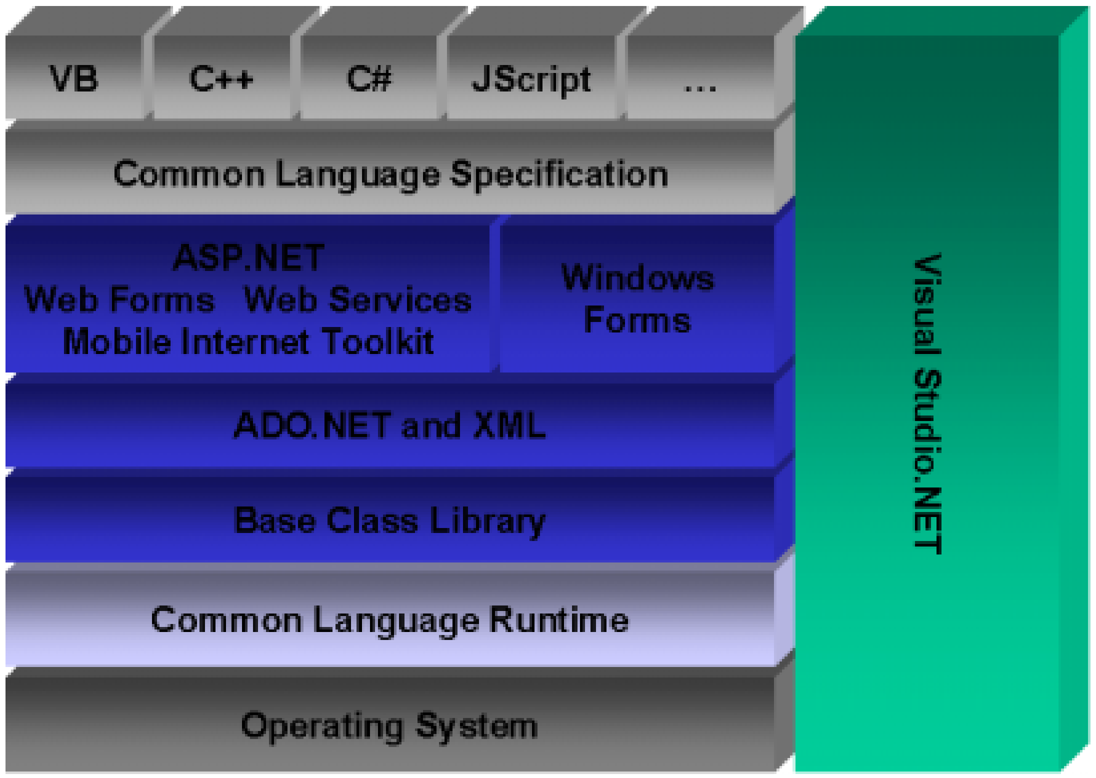


Figure 1 **-** *Microsoft .NET Framework*

Microsoft Visual C++ .NET

Traditional *unmanaged* (outside of .NET) C++ and new *managed* (within .NET's structure) C++ code can be mixed freely within the same application. Existing components can be wrapped as .NET components by using the managed extensions. Most importantly, providing support for C++ preserves investment in existing code while integrating with the .NET Framework.

JavaScript / JScript

JavaScript is the common language used for web development. This is because older versions of Netscape Navigator supported only JavaScript, while Microsoft Internet Explorer supported JavaScript and VBScript. If a website is being created with client-side scripting, maximum compatibility can be maintained by using JavaScript. It's no wonder the popular language is supported in the .NET Framework.

Other Languages

This is not a complete list of the languages supported by .NET, however, not by a long shot. By creating and publishing a *Common Language Specification* (CLS), third party vendors can take new or existing languages and fit them into the .NET puzzle. This means COBOL, FORTRAN, Java, and many other languages can now be used to program Windows (WinForms) and Web (ASP.NET) applications and services. Therefore the list will never be complete as vendors continue to add to the roster of supported languages. And, if these languages follow the rules laid down by the CLS, they can then access the libraries provided with .NET, as shown in Figure 1's middle layer.

The Middle Layer

The middle layer looks nice and neat the way it's divided up in Figure 1, but really it all exists in a grouping of over 90 collections, each one referred to as a *namespace*. All languages that follow the CLS can work with these namespaces and thereby use ADO.NET (database support) and the other libraries necessary to create Windows and Web applications. It is also possible to create namespaces outside of what Microsoft has already provided.

ASP.NET

The rewrite of ASP (Active Server Pages) – called ASPX for no other reason than they were focused more on creating something cool than trying to figure out a catchy name – became known as ASP+, and later renamed to ASP.NET. It addressed many shortcomings of ASP. While ASP pages (.asp) are still supported by the ASP.NET server, they must use the new file extension (.aspx). ASP.NET focuses more on separating the HTML from the code. By using a *code behind* approach, the HTML is kept in an .aspx file, and the new general practice is to place the blocks of code into separate files to be included by the .aspx files (such as pagename.aspx.vb for VB.NET code, or pagename.aspx.cs for C# code, for example).

In addition, ASP.NET works closely with Microsoft NT/Win2K servers' security settings. It works within those policies to make changes to permissions, rights, and more, much easier. It also allows a user session to be shared over multiple servers so that load balancing is easier and more effective. And, should one of the servers go down, the user remains blissfully unaware and is able to carry out his transaction because his session does not live on any one server.

And last, ASP.NET compiles its pages *just in time* so that execution is much faster than ASP. The first time an .aspx page is accessed after being saved to the web server, the page is compiled into a pseudo code form. This doesn't increase actual execution speed of the code (that is, it's not compiled into machine language), but it does allow the ASP.NET Server to avoid the compilation step for each and every user before spitting out the generated HTML. Execution speed seems faster to the end-user since the compilation step occurs only when the page is modified.

ADO.NET

Open Data Base Connectivity (ODBC) is an old tried and true standard for accessing data. It was designed to provide a common set of routines to programmers. These routines remained unchanged regardless of the type of database being accessed (e.g. Access, SQL, Oracle). The next step in the evolution of the anonymous data store was OLE-DB that not only supports ODBC, as well as its own methods for working with Access/SQL/Oracle, it also works with Exchange, Excel, and other applications (no, they don't have to be Microsoft applications, just support OLE-DB).

ADO.NET (ActiveX Data Objects) is a friendly interface to OLE-DB. It provides a set of objects to the languages working within the guidelines of the CLS. It's yet another level of abstraction to keep things simple and common to the programmer, and allows ADO.NET to deal with the bit twiddling behind the scenes.

Base Class Libraries

ASP.NET and ADO.NET are part of the base class libraries provided in the .NET framework. These namespaces are what provide common objects and methods used by all CLS-compliant languages in the .NET framework.

Common Language Runtime

The final layer is the Common Language Runtime, or CLR. This piece sits on top of the operating system and executes the compiled code. So here's where it gets really cool. Because all languages that want to work with .NET must conform to the CLS, and these *managed* languages all use the base class libraries (including ASP.NET and ADO.NET), everything can be compiled down to a common set of metadata or an *Intermediate Language*. This is the most basic level of data and at this point it doesn't matter what language the instructions were written in. VB, C#, C++, Java, COBOL, whatever, it all looks the same at this intermediate language level.

This is a wonderful thing because this means that all languages can (and do) share the same class definitions and objects defined further up the ladder. A namespace can be created and used by all of the languages because they all eventually end up at this very basic level so that the runtime engine can interpret them.

It gets better. Because the programming languages use the objects created via the .NET namespaces for file manipulation, and therefore the CLR separates the operating system from those languages, different versions of the CLR can be written for the Macintosh, Linux, and so on. When OS-specific versions of the CLR are rolled out, it will be possible to write your program once and have it deployed on multiple operating systems without any extra work. (In theory). Sound familiar? (Hint: Sun Microsystems' goal with Java).

Challenges

As with any new approach, there is always a price of entry, whether it's the learning curve or bringing your now-Legacy-code along into the new system. In the case of the .NET Framework there are two obvious challenges from the start: Migration of old code into the new environment and understanding how much control .NET wants to exercise in an attempt to hold our hands and make things easier.

Migration Issues

The Common Language Specification requires all languages to follow specific guidelines to be allowed to participate in the .NET Framework. This applied to creating a .NET version of Visual Basic as well. The result is changes that are easy to accept when creating new applications, but can be more involved when migrating code.

In the case of Visual Basic (VB.NET), for example, migration issues exist in regards to the introduction and handling of new data types, renaming/moving of functions, and the discontinuation of keywords.

What this has to do with software testing is that the code base that once worked "good enough" to share with the user-base gets touched, and in an invasive way. Opening a code base after Testing has blessed it is already a tricky business, but to modify code so that it can work within the new framework – replacing type declarations, using new functions, and more – will require a full test pass to verify nothing breaks in the process. (Software test automation that's already in place will come in handy).

New Types & Keywords

With the creation of a more generalized approach allowing many languages to work together comes the need to tighten and redefine past approaches. **Table 1** reflects just a few of the differences between VB6 and VB.NET.

Visual Basic 6	**VB.NET**
Fixed length strings were declared as: Dim Name as String * 30	Fixed-length strings are not allowed
An Integer type is 16-bits	A Short type is 16-bits
A Long type is 32-bits	An Integer type is 32-bits
No support for a 64-bits integer type	A Long type is 64-bits
Any data can be set to a Variant variable	The Variant type is unsupported
"Dim X, Y as Long" results in X declared as a Variant and Y as a Long	"Dim X, Y as Long" results in X and Y declared as a Long
Keyword "Empty" indicates an un-initialized Variant variable. "Null" indicates that a variable contains no valid data.	"Null" and "Empty" have been replaced by the keyword "Nothing"

Table 1 *- Visual Basic 6 & VB.NET differences (Types and Keywords)*

Moved Functions

To follow the new object-oriented approach that .NET utilizes through its libraries – known as *namespaces* – functions have been relocated. Let's take Visual Basic's Math functions for example. They have all been moved into the `System.Math` group, so now:

```
X = Cos(Y)
```

Becomes:

```
X = System.Math.Cos(Y)
```

A tool does exist for migrating Visual Basic 6 projects over to VB.NET. Before you breathe a sigh of relief, however, know that most people who have used this tool say that it is not that helpful on large conversion efforts. If you have a simple application to convert, it will provide you with some assistance. However, the changes between version 6 and VB.NET are great enough to make a re-write of the application worth considering, depending on the type of application and its features. For more details about upgrading your Visual Basic 6 applications to VB.NET, I recommend the following MSDN article as a very good starting point:

http://msdn.microsoft.com/vbasic/techinfo/articles/upgrade/vbupgrade.asp

No Direct Mapping of ASP.NET to HTML Controls

One of the exciting features of ASP.NET is its ability to spare the developer from the hassles of tracking which browser a visitor is using during a web session and providing different HTML based on the visitor's browser's capabilities. ASP.NET will issue the HTML it thinks best suits the client. This is a great concept, but in practice the results are not yet clearly known. It remains to be seen how well this type of handholding will work and if it results in workarounds being that much more challenging.

An example is the text box placed on a web page. In ASP.NET it looks like this:

```
<asp:TextBox id="SearchTextBox" runat="server"
MaxLength="25"></asp:textbox>
```

It generates the following HTML:

```
<input name="ModuleSearch:SearchTextBox" type="text"
maxlength="25" id="ModuleSearch_SearchTextBox" />
```

However, depending on the browser, it could also generate this HTML:

```
<textarea name="ModuleSearch:SearchTextBox" rows=1
maxlength="25" id="ModuleSearch_SearchTextBox"><textarea>
```

You will note that these are two different control types, yet they can resemble each other depending on the browser being used. The theory is that ASP.NET knows best, and this remains to be seen. To be sure, as feedback comes in ASP.NET will become much more robust as Microsoft builds on its goal of helping testers and developers alike worry less about browser compatibility.

Testing in .NET

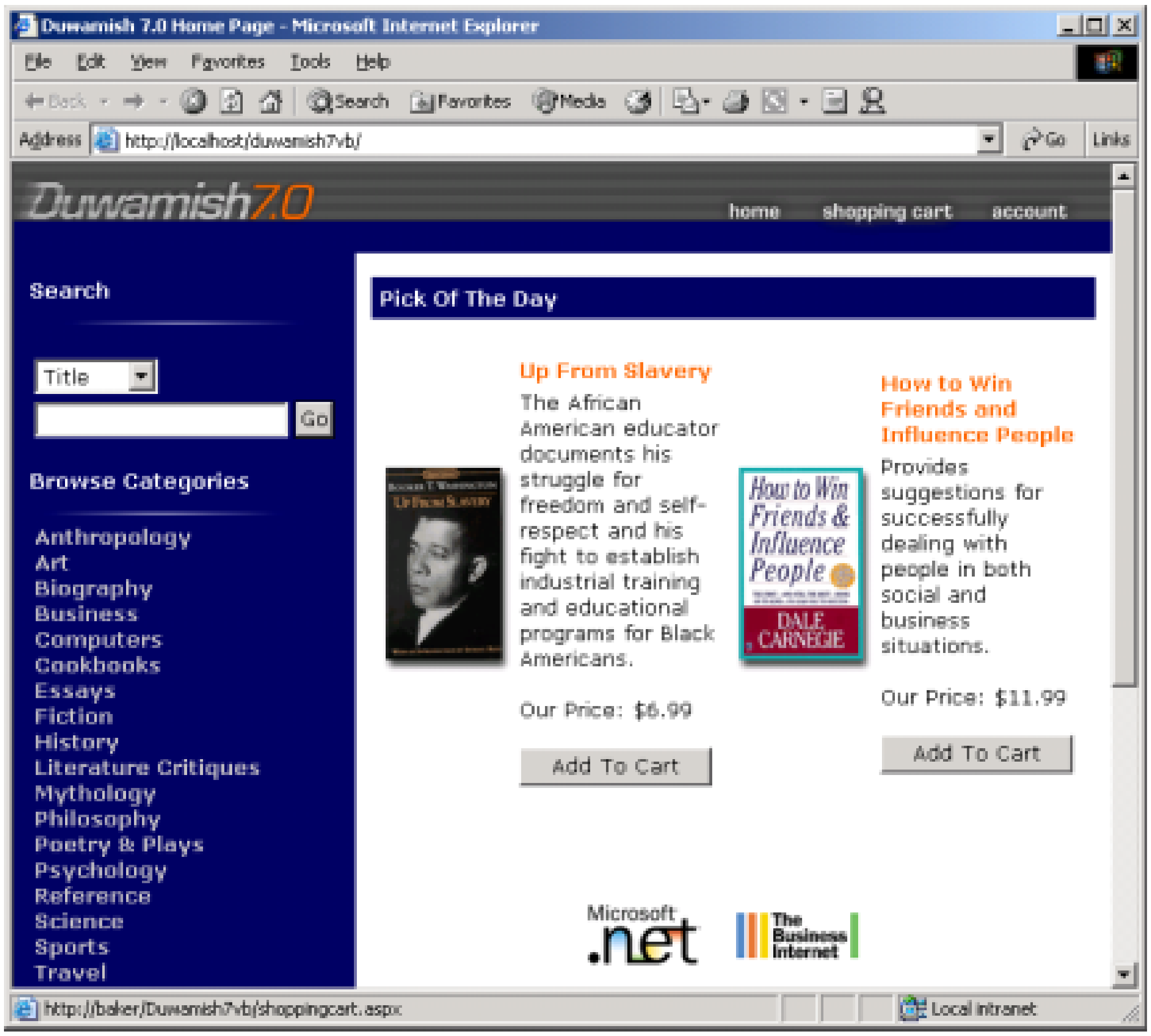
Now that we have a high-level view of what Microsoft .NET is, some of the challenges that programmers face, and some of the issues testers need to be aware of, let's look at a sample ASP.NET application and some of the things we should consider in its testing.

The web application we'll use in this example comes with Microsoft Visual Studio .NET and is called "Duwamish 7.0." Its home page is shown in **Figure 2** on the following page. This application is for a fictional on-line bookseller and demonstrates the concepts of modular development, working with controls, searching, an e-commerce shopping cart, and working with a Microsoft SQL Server database. (Microsoft was kind enough to fill the database for us with sample data).

Black Box Testing Remains Crucial

Although .NET makes technical testing more accessible to software test engineers, the non-programming aspects for software testing remains key. Usability issues remain important, of course, as does verifying that an application behaves, as an end-user would expect. There is nothing new to be introduced to software test engineers in the realm of black box testing in regards to an ASP.NET web application. Browser compatibility testing remains an important part of the process, especially since ASP.NET generates HTML unique to a user's operating system, browser and browser version.

In regards to taking more technical approaches, there are many opportunities available to test engineers, which the remainder of this document will explore.



The African American educator documents his struggle for freedom and self-respect and his fight to establish industrial training and educational programs for Black Americans.

Provides suggestions for successfully dealing with people in both social and business situations.

Figure 2 *– Sample application that invokes many of ASP.NET's features.*

Technical Testing

Microsoft has provided the tools necessary to test and debug ASP.NET applications that are not only useful for developers but testers as well. Some of these tools move into the realm of gray and white box testing, however, which some testing organizations are against. The concern of these organizations is that it steps too far away from what the end-user will experience. It also requires a more technical (and hence, typically more costly) test engineer. I am of the opinion that while black box testing is extremely important, the more technical a tester can be in their efforts, the more effective they can be in diagnosing and tracking down issues and bugs and communicating those problems to their programmer counterparts.

In this section we will introduce the `<trace>` setting that can be added to an application's `web.config` file and how it is used in debugging. We will also look at some of the tools available for automating the testing on ASP.NET applications, both in functionality and load/stress testing.

Web.config

ASP.NET has what Microsoft refers to as a *configuration system*. This system is an extensible infrastructure that enables all ASP.NET applications' configuration settings to be defined when an application is first deployed, and modified any time thereafter. The root configuration file is `machine.config` and configures the entire web server. Another file – `web.config` – can appear in multiple directories throughout the ASP.NET web application server. The `web.config` file affects the directory it is in, as well as its directory's sub-directories. In addition, a `web.config` file in a lower child directory can override or modify those settings of its parent.

Each `web.config` file contains a nested hierarchy of XML tags and sub-tags. These tags have attributes that specify the configuration settings. There are over 60 elements that make up the configuration schema that controls how ASP.NET web applications behave. You can even add your own, if you like. **Listing 1** shows an example of a `web.config` file.

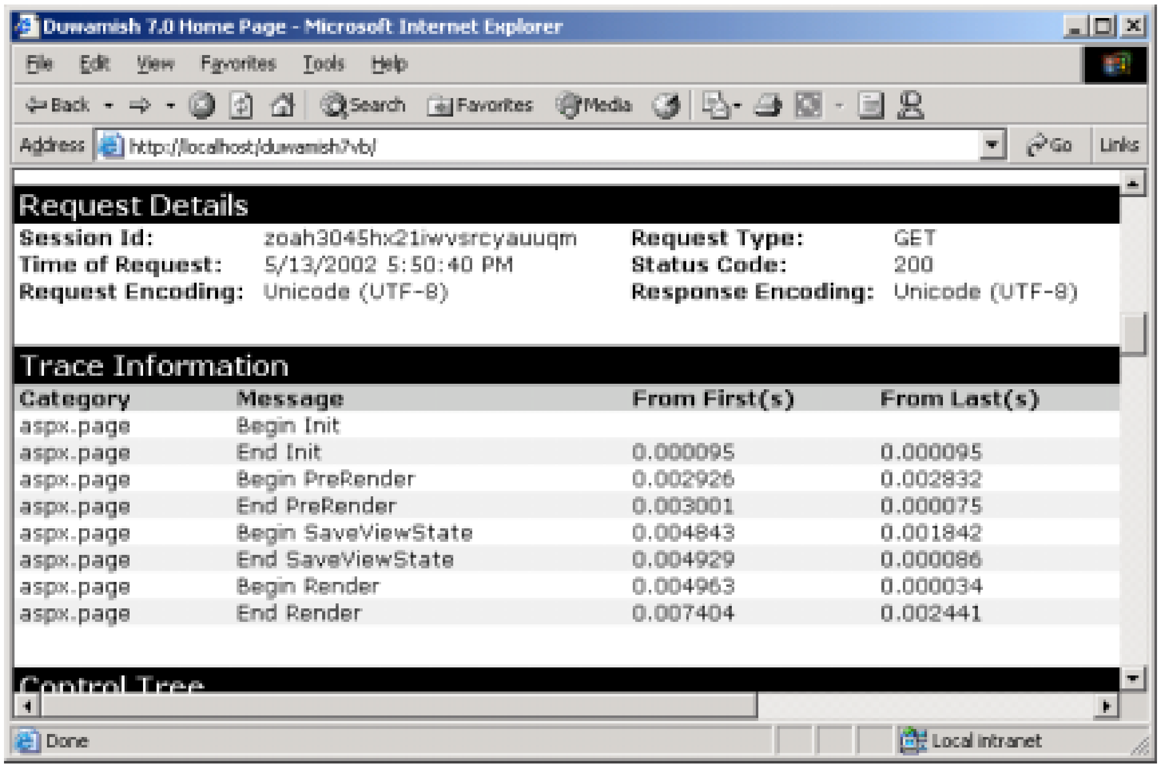
```xml
<?xml version="1.0" encoding="utf-8"?>
<configuration>
 <configSections>
  <section name="ApplicationConfiguration"
   type="Duwamish7.SystemFramework.ApplicationConfiguration,
   Duwamish7.SystemFramework" />
  <section name="DuwamishConfiguration"
   type="Duwamish7.Common.DuwamishConfiguration,
   Duwamish7.Common" />
  <section name="SourceViewer"
   type="System.Configuration.NameValueSectionHandler, System,
   Version=1.0.3300.0, Culture=neutral,
   PublicKeyToken=b77a5c561934e089" />
 </configSections>
 <system.web>
  <customErrors defaultRedirect="errorpage.aspx" mode="On" />
  <compilation debug="true" />
  <sessionState cookieless="false" timeout="20" mode="InProc"
   stateConnectionString="tcpip=127.0.0.1:42424"
   sqlConnectionString="data source=127.0.0.1;
   user id=sa;password=" />
  <globalization responseEncoding="utf-8"
   requestEncoding="utf-8" />
  <!-- security -->
  <authentication mode="Forms">
    <forms name=".ADUAUTH" loginUrl="secure\logon.aspx"
     protection="All">
    </forms>
  </authentication>
  <authorization>
    <allow users="*" />
  </authorization>
 </system.web>
</configuration>
```

***Listing 1**: Example of a web.config file.*

First and foremost, don't sweat it. This isn't as scary as it looks, and not only that, we only want to work with a very small section of the file. Specifically, we will insert a `<trace>` tag directly under the `<system.web>` tag and place it into the directory containing the pages we want to work with. In this case, to keep it simple, and to avoid multiple copies of `web.config` that could go forgotten, it will be the root version of `web.config` that we modify. The following line is added:

```
<trace enabled="true" pageOutput="true" requestLimit="15"
 traceMode="SortByCategory" />
```

Inserting this simple XML tag and its properties has dramatic effects on your ASP.NET web application. In standard ASP pages it was necessary to insert statements to print out the current status of variables or track what branches of code were executed. This was done by strategically placing `Response.Write()` functions throughout the .asp files. The problem was that these statements had to be removed later (and not forgotten). ASP.NET's solution to this is the *trace* functionality. With the above line added to `web.config` in the Duwamish 7.0 sample application, navigating to its home page (Figure 1) tacks on the additional information shown in **Figure 2**.


Duwamish 7.0 Home Page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost/duwamish7vb/
Go Links
Done
Local intranet

Request Details

Session Id:	zoah3045hx21iwvsrcyauuqm	**Request Type:**	GET
Time of Request:	5/13/2002 5:50:40 PM	**Status Code:**	200
Request Encoding:	Unicode (UTF-8)	**Response Encoding:**	Unicode (UTF-8)

Trace Information

Category	Message	From First(s)	From Last(s)
aspx.page	Begin Init		
aspx.page	End Init	0.000095	0.000095
aspx.page	Begin PreRender	0.002926	0.002832
aspx.page	End PreRender	0.003001	0.000075
aspx.page	Begin SaveViewState	0.004843	0.001842
aspx.page	End SaveViewState	0.004929	0.000086
aspx.page	Begin Render	0.004963	0.000034
aspx.page	End Render	0.007404	0.002441

Control Tree

***Figure 2** – The <trace> tag is added to web.config in the Duwamish 7.0 web directory.*

As you can see, ASP.NET's new tracing functionality allows us to view verbose information about an application with minimal intrusiveness. In the past it was necessary to sprinkle `Response.Write()` routines throughout the code. Now, only one file needs to be modified lowering the likelihood this setting will be accidentally left enabled. In addition, `Trace.Write()` routines may be used throughout the code where `Response.Write()` methods might have been used for debugging purposes in the past. Because `Trace.Write()` is only invoked when the `<trace>` tag is in place, those statements may remain in place without ill effect.

The output provided when using `<trace>` is provided in six sections:

- **Request Details:** This provides such basic information as the visitor's unique session ID, the time & date the request came into the server, HTTP request type and its status code.
- **Trace Information:** In addition to ASP.NET-generated information about the execution of the application, this is where `Trace.Write()` values are printed. The two parameters taken by `Trace.Write()` are displayed here as Category and Message values.
- **Control Tree:** This section provides information about the controls used within the page. ID, type, render size, and view state information is provided.
- **Cookies Collection:** Any cookies sent by the client in its request to the server are displayed.
- **Headers Collection:** HTTP header values sent by the client to the server are displayed here in a simple Name/Value pairing.
- **Querystring Collection:** This shows up only when the `GET` method is used to submit a form. It shows the variables and values sent with the request. (These same values can be found in their raw form in the `QUERY_STRING` entry of the Server variables section).
- **Server variables:** The Name and Value of server side variables are displayed in this table. This section includes a lot of the same information listed in the other sections, just not as nicely formatted.

This information is helpful to test engineers in a number of ways. The *Request Details* section, for example, is important in showing how the server responded back to the client. Specifically, showing a status code of 200 means that no errors were encountered. An error of 400 is a *Bad Request*, 401 is *Required Authorization*, 403 is *Forbidden Directory*, 404 is *Page Not Found*, and 500 is *Internal Server Error* (nothing new). Being able to print a page that shows the request string that resulted in an Internal Server Error can be very helpful to development. The *Trace Information* section can provide helpful information about which branches of code were executed if the `Trace.Write()` method was used by the programmers. The *Headers Collection* shows the type of browser and operating system used to access the page, which can be helpful in verifying the browsers your group identified as important are actually used in testing. The *QueryString Collection* makes it easy to see what values a form sent to the page currently being displayed.

<u>Microsoft ACT 1.0</u>

Purpose of Microsoft Application Center Test 1.0 – or ACT – is to stress test web servers and analyze performance and scalability problems with web applications. This includes ASP.NET applications and their components. This type of testing is accomplished by opening multiple connections to the server and rapidly sending HTTP requests, thereby simulating a large group of users. Although high-load stress testing over long periods of time is ACT's main purpose, it can also be used for functionality testing. Lastly, Application Center Test will work with any web server or web application that adheres to the HTTP protocol.

Putting Application Center Test into use on a testing project allows you to see how your web server reacts when several hundred users access your application at the same time. This simulates peak periods and not only provides performance and scalability information, but also tests databases in regards to such issues as concurrency, transactions, number of users supported, locks, pooling, and so on.

This tool comes with Visual Studio .NET Enterprise Developer and can be used within the Visual Studio .NET's Integrated Development Environment (IDE), but more options are available when the stand-alone ACT program is used.

```
Sub SendRequest1()
    Dim oConnection, oRequest, oResponse, oHeaders, strStatusCode
    If fEnableDelays = True then Test.Sleep (0)
    Set oConnection = Test.CreateConnection("localhost", 80, false)
    If (oConnection is Nothing) Then
        Test.Trace "Error: Unable to create connection to localhost"
    Else
        Set oRequest = Test.CreateRequest
        oRequest.Path = "/duwamish7vb"
        oRequest.Verb = "GET"
        oRequest.HTTPVersion = "HTTP/1.0"
        set oHeaders = oRequest.Headers
        oHeaders.RemoveAll
        oHeaders.Add "Accept", "image/gif, image/x-xbitmap, " + _
             "image/jpeg, image/pjpeg, application/msword, " + _
             "application/vnd.ms-powerpoint, application/vnd.ms-excel, */*"
        oHeaders.Add "Accept-Language", "en-us"
        oHeaders.Add "User-Agent", "Mozilla/4.0 (compatible; " + _
             "MSIE 6.0; Windows NT 5.0; .NET CLR 1.0.3705)"
        'oHeaders.Add "Host", "localhost"
        oHeaders.Add "Host", "(automatic)"
        oHeaders.Add "Cookie", "(automatic)"
        Set oResponse = oConnection.Send(oRequest)
        If (oResponse is Nothing) Then
            Test.Trace "Error: Failed to receive response for URL to " + _
             "/duwamish7vb"
        Else
            strStatusCode = oResponse.ResultCode
        End If
        oConnection.Close
    End If
End Sub
```

Listing 2 *– Example of a single request generated by ACT 1.0's recorder.*

Listing 2 shows an example of a subroutine generated by ACT's recorder. When a script is generated, requests are broken up into individual subroutines that are then called one-by-one by a `Main()` subroutine.

Just like the rest of .NET, this application relies on an object-oriented approach and provides a number of objects that you can use. In Listing 2, note that we're looking at the `SendRequest1()` subroutine. This is the first routine generated when the webapp-under-test was navigated to. The line of interest is the one that says, `oRequest.Path = "/duwamish7vb"`. This is the root of the web directory or site we've selected for testing. The other pieces specify the type of request coming in (in this case "GET" instead of "POST" or "HEAD), the MIME types your browser declares that it will accept and understand, your browser information, and so on. (You'll be happy to know that cookie handling is built in to ACT 1.0). After completing the `Request` information, it is dispatched to the web site with a call to the `Connection.Send()` method (`oConnection` is a Connection object, which was created when a call was made to `Test.CreateConnection` at the front part of the subroutine). The `oConnection.Send(oRequest)` call sends off the Request information that was filled out in the middle of the script. (Lots of stuff just to generate a single request, eh?)

Subsequent `SendRequest`*N*`()` routines are created (where *N* equals the next number in the sequence) based on what the displayed page relies upon. This includes a request for the cascading style sheet (/duwamish7vb/css/duwamish.css), images (bannerlogo.gif, bannerhome.gif, bannercart.gif, banneraccount.gif, line.gif, etc.), and the page itself (Default.aspx).

Finally, down around the 15th request, in the `SendRequest15()` routine, we get to the request that was generated when typing in a search string and clicking the submit button. The resulting request path looks something like this:

```
oRequest.Path = "/Duwamish7vb/searchresults.aspx" + _
"?type=0&fullType=Title&text=how+to+win+friends"
```

This is a direct call to the .aspx (ASP.NET) file using the `GET` method (the `GET` method causes the submitted form's values to be part of the URL, as opposed to `POST` which embeds the variables and their paired values into the HTTP header and goes unseen by the user). This call goes through the whole process previously described, sending off the above request to `searchresults.aspx` with the query string shown above, hoping for a reply in HTML by the server. The CSS file is downloaded as are the GIFs and JPEGs. Whew! Lot's of traffic going on just for a single request!

When all is said and done, 34 separate requests are generated, and all we did was:

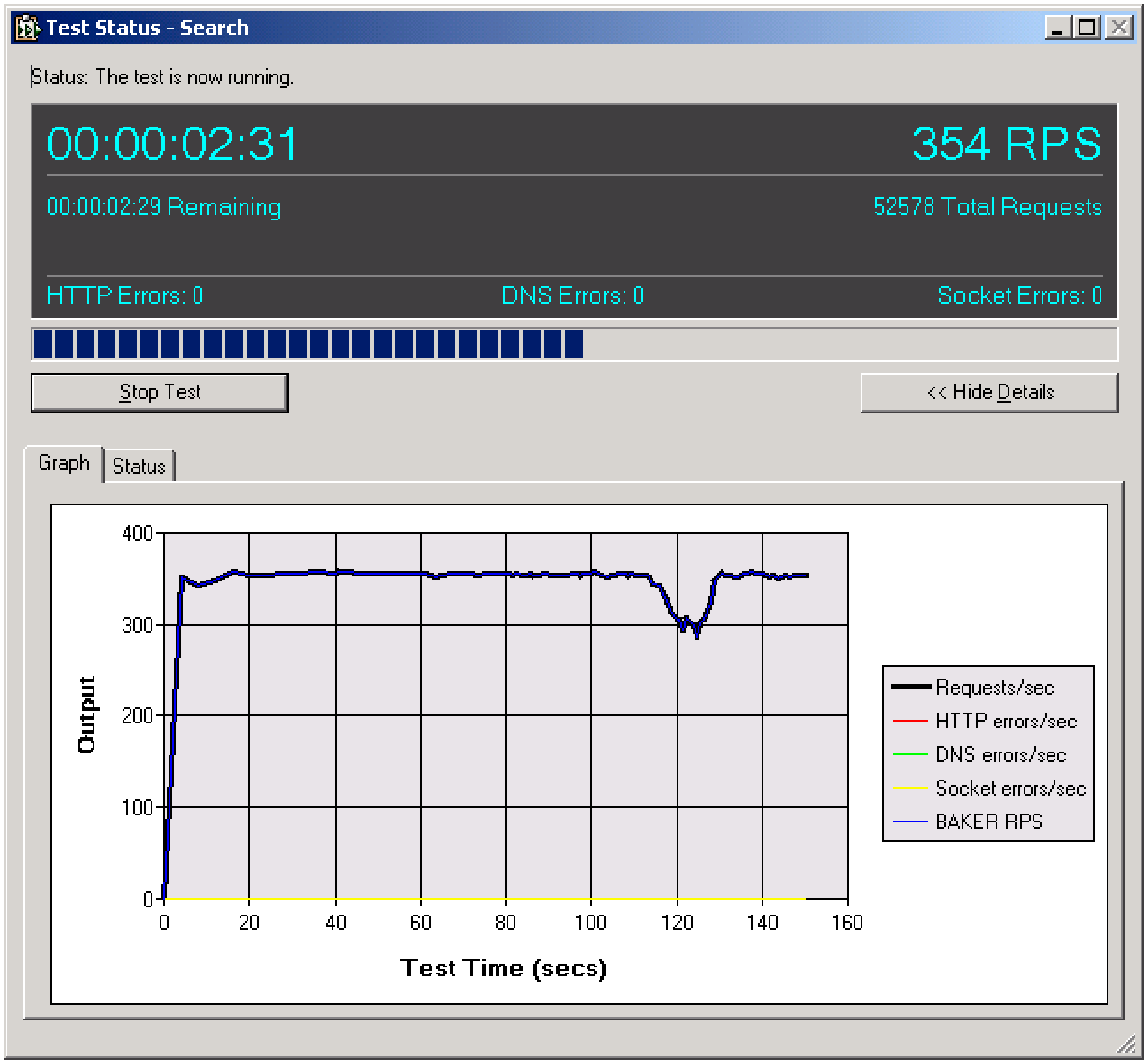
1. Navigate to http://localhost/duwamish7vb
2. Type "how to win friends" into the search box
3. Clicked the "Go" button to submit the form
4. Clicked on the link of the book found by the query

The main routine that fires off each of these requests is simple enough, and shown (with some abbreviation) in **Listing 3**.

```
Sub Main()
    call SendRequest1()
    call SendRequest2()
    ': (3-33 removed for brevity)
    call SendRequest34()
End Sub
Main
```

Listing 3 *– Each Request is sent in turn by calls to their corresponding subroutines.*

Running the script is as simple as clicking a *Play* button found on the toolbar. As the test executes, the status can be viewed and looks similar to **Figure 3**. This status box communicates the current state (for example, "The test is now running"), time elapsed, time remaining, average requests per second, and total requests made so far. It also lists the number of errors currently encountered allowing you to decide if the test needs to be aborted or allowed to continue.


Test Status - Search

Status: The test is now running.

00:00:02:31 354 RPS

00:00:02:29 Remaining 52578 Total Requests

HTTP Errors: 0 DNS Errors: 0 Socket Errors: 0

Figure 3 *– Execution of the ACT script halfway through its test run.*

At approximately 2 minutes and 5 seconds into the test run, I jumped to my browser and tried 3 searches, resulting in the dip in Requests Per Second (RPS) generated by the load test, shown in Figure 3. (Hey, the straight line was looking boring). This allowed me to see how peppy the site was even under load.

When a script completes, its results can be viewed by clicking on the *Results* object in the test project. The results are displayed in a list based on the name of the script being executed, and the time/date of its execution. Results show the total number of requests generated during the run (in this case, it was 5 minutes and generated 104,227 requests, an average of 347.42 requests per second). It also gives the test engineer an idea of how responsive the system was for those requests. Responsiveness is measured by tracking the "Average Time to First Byte" (how long until the server started sending back a response) and "Average Time to Last Byte" (total time to send the requested item). Errors are also listed (HTTP, DNS and Socket), as are network statistics.

Considering my test example ran on the same computer as the web server, bandwidth was pretty good (1,227,864.16 bytes/second) on my 1.2ghz laptop. And last, response codes are tracked allowing us to understand the quality of the requests and responses. The return result of "200" communicates the request was received and a full response was returned. Using multiple computers on the network is one way to raise the number of requests per second experienced by the server. Getting a faster test machine with a big fat pipe direct to the server is another.

There is much more to discover about this program, including tracking server performance, test machine performance, and more. I've used it more in a recorder/playback capacity, but the language is based on VB Script and isn't difficult to master. There are only about 5 objects (with many methods each) that comprise the model: `Test`, `Request`, `Connection`, User and the Response object. Real-world use of this tool is likely to be: Record a few actions to generate a script, move the common bits into separate routines, and use constants based on the type of machine being used and browser being emulated. There is definitely room for structured programming to be put into practice to make it less bulky and more responsive to the evolution of a web site.

All in all, it's a great tool that provides a lot of capability, and it's "free" (as long as you have the full-blown version of Visual Studio .NET).

Other Considerations

There are other tools out on the market worth considering. These include *Red-Gate*'s new ANTS ("Advanced .NET Testing System") product, *Rational Software*'s new Java-based tool for testing Java and HTML applications, and *PushToTest*'s free open-source load & monitoring tools. These are new or less known, so I won't bother to list the other tool vendors and tools of which you are probably already aware.

- **ANTS** appears to be very similar to Microsoft Application Center Test 1.0 in what it provides. It can be used to generate HTTP requests as well as test Web Services through HTTP requests enriched with the SOAP protocol. They have a 14-day free trial that's worth a look (only allows 10 simultaneous connections in the demo, however). At present they're pricing their tool at around $2,000+.
- **Rational Software**'s new Java-based testing tool is very intriguing. Look for my white paper about it on the AutomationJunkies.com site in July when it releases. By taking advantage of an object-oriented language like Java, this new tool provides flexibility by treating each item on a web page or Java applet as an object. This allows the object to be modified in the script later if it is modified on the website or in the applet. Broken scripts due to UI changes are fixed more quickly (a common automation nightmare). In addition, this new tool has a level of intelligence built in allowing it to weigh the likelihood that a modified control is still the control it wants to interact with, further improving maintainability.

- **PushToTest** has an automation tool whose UI is in Java – allowing it to work anywhere – and the scripting language is based on *Python*, a language that's easier to learn than C or C++. Using the Java framework allows test engineers to deploy their Python test scripts and generate loads against their server under test. Best of all, it's open source, so you have the code base and are free to modify its functionality and capabilities to your heart's content.

Summary

This is a lot of information to absorb, to be sure. We started by looking at the .NET Framework and getting a 30,000-foot view of what it is and some of the things it has to offer. To continue learning about .NET I suggest visiting such sites as **GotDotNet.com** and **DotNetJunkies.com**. These are two great sites for tutorials and other information about working in .NET. Unfortunately, they're mostly programmer-centric, so you should also check out StickyMinds.com and **AutomationJunkies.com**. Be sure to also visit **QAForums.com** for a great list of discussion groups.

The next thing we looked at were some of the challenges development teams face – concerns to programmers and testers alike – when moving into the .NET realm. These include the modification of variable types between VB6 and VB.NET, the placement of functions within the .NET object model, and how .NET does some handholding, which could prove to be problematic when it comes to tweaking how a page is displayed in different browsers. (This could be especially problematic to automation tools that rely on specific controls to be in place and don't tie themselves to a single browser for testing).

We then looked at an ASP.NET example and a tool that comes with Microsoft Visual Studio .NET (Application Center Test 1.0), as well as intrinsic support for ASP.NET deployed applications (the `<trace>` tag in `web.config`). Let's not forget the other new tools that are coming to market to help support .NET testing, as well as new tools that weren't necessarily targeting .NET, but can be used regardless.

This paper scratches the surface, but still provides you with a strong starting point with some of the options available to you when testing in the .NET maze. Good luck and have fun!

#

If you have questions or comments about this paper, please address them to tom@xtenddev.com. If you find your team is undertaking automation approaches, be sure to also visit this new test automation site: www.AutomationJunkies.com