



.NET Performance Testing and Optimization

Part 1: Building your test environment

Paul Glavich with Chris Farrell



.NET Performance Testing and Optimization

Part 1: Building your test environment

By Paul Glavich

With Chris Farrell

First published by Simple Talk Publishing 2010

Copyright Paul Glavich and Chris Farrell 2010

ISBN 978-1-906434-41-0

The right of Paul Glavich and Chris Farrell to be identified as the authors of this work has been asserted by them in accordance with the Copyright, Designs and Patents Act 1988.

All rights reserved. No part of this publication may be reproduced, stored or introduced into a retrieval system, or transmitted, in any form, or by any means (electronic, mechanical, photocopying, recording or otherwise) without the prior written consent of the publisher. Any person who does any unauthorized act in relation to this publication may be liable to criminal prosecution and civil claims for damages.

This book is sold subject to the condition that it shall not, by way of trade or otherwise, be lent, re-sold, hired out, or otherwise circulated without the publisher's prior consent in any form other than that in which it is published and without a similar condition including this condition being imposed on the subsequent publisher.

Technical Review by Alex Davies, Jeff McWherter, and Corneliu Tusnea

Cover Image by Paul Vlaar

Edited by Chris Massey

Typeset & Designed by Matthew Tye & Gower Associates

Table of Contents

- About the Authors..... ix
- Chapter 1: Introduction - The What and the Why.....12**
 - Performance testing..... 12
 - Load testing..... 13
 - Stress testing 13
 - Profiling.....14
 - Cost benefits of performance and load testing.....14
 - Example scenarios 15
 - Sometimes, what seems right can be wrong. 18
 - Conclusion 18
- Chapter 2: Understanding Performance Targets 20**
 - Identifying performance targets 20
 - Structuring test breakdowns 21
 - Determining what load to target 22
 - Contingency in your estimations 24
 - Estimate the mix of browsers for your web application 26
 - What data do we measure? 26
 - Time to First Byte 27
 - Total page response time..... 28
 - What about average response time? 29
 - Sweet spots and operational ceilings 30

Conclusion	32
Chapter 3: Performance and Load Test Metrics	33
What metrics do we need?	33
Basic metrics.....	33
Web application basic metrics	36
What to look for	40
CPU utilization	40
Memory utilization.....	41
Response time	43
Creating a baseline	44
Using Visual Studio to analyze the results	44
Using the Test Results management window	45
Using the Open and Manage Test Results dialog	45
Filtering performance test result selection.....	46
Sweet spots and operational ceilings	50
Detailed performance metrics.....	52
Performance metrics	53
What do I do with all this information?	60
Conclusion	62
Chapter 4: Implementing Your Test Rig	64
Creating the performance test rig	64
Architecture and structure of a performance test rig.....	64
Role breakdown.....	65

Setting up and configuration.....	67
Port setup and firewall considerations.....	67
Network segmentation/isolation.....	69
Controller setup.....	73
Creating the load test database.....	75
Guest policy on Windows XP in workgroup mode.....	76
Agent setup.....	76
Workstation setup.....	77
Troubleshooting the controller and agents.....	78
Setting up performance counter collection.....	82
Conclusion.....	87
Chapter 5: Creating Performance Tests.....	88
Basic solution structure.....	88
Recording the web tests.....	91
Test replay.....	99
Data Binding Web Tests.....	106
Creating a data source for data binding.....	106
Test deployment considerations.....	113
Web test code generation.....	115
Extensibility through plug-ins.....	117
Alternative ways of recording web tests.....	120
Considerations for load balancing / load balanced hardware.....	123
Test automation.....	125

Creating a performance test scenario.....	125
Putting automation in place.....	136
Executing the load test.....	136
Collecting performance monitor data.....	137
Collecting SQL Server usage statistics	140
Clean up tasks	143
Conclusion	144
Chapter 6: Application Profiling.....	145
Types of profiling	145
Performance profiling	147
Memory profiling.....	149
When to start profiling.....	151
Reactive debugging.....	152
Proactive analysis.....	153
Technique validation.....	153
Tools used for profiling.....	154
CLR profiler	154
Red Gate's ANTS Memory and Performance Profilers	155
Microfocus DevPartner Studio Professional 9.1.....	159
Microsoft Visual Studio 2008 profiling tools	163
What to look for	167
Performance analysis.....	167
Memory analysis	169

Production / load test clues	171
General performance counters	171
Managing profiling results.....	172
Comparing analysis runs	172
Pre-check-in requirements.....	172
Continuous integrated testing	172
Summary.....	173

About the Authors

Paul Glavich

Paul has been an ASP.NET MVP for the last six years, and works as a solution architect for Datacom. Past roles have included technical architect for EDS Australia, and senior consultant for Readify. He has accumulated 20 years of industry experience ranging all the way from from PICK, C, C++, Delphi, and Visual Basic 3/4/5/6 to his current speciality in .NET with ASP.NET.

Paul has been developing in .NET technologies since .NET was first in Beta. He was technical architect for one of the world's first Internet Banking solutions *using* .NET technology.

He can be found on various .NET-related newsgroups, and has presented at the Sydney .NET user group ([WWW.SDNUG.ORG](http://www.sdnuug.org)) and TechEd Australia on numerous occasions. He is also a member of ASPInsiders ([WWW.ASPINSIDERS.COM](http://www.aspinsiders.com)) with direct lines of communication to the ASP.NET team. He has co-authored two books on ASP.NET Ajax, has written technical articles which can be seen on community sites such as ASPAlliance.com ([WWW.ASPALLIANCE.COM](http://www.aspalliance.com)), and also has his own blog at [HTTP://WEBLOGS.ASP.NET/PGLAVICH](http://weblogs.asp.net/pglavich). On top of all this, he also runs the Sydney Architecture User group ([HTTP://THESAUG.COM](http://thesaug.com)).

Paul's current interests in technology revolve around anything in ASP.NET, but he also has a strong interest in Windows Communication Foundation, on which he is a Microsoft Certified Technical Specialist. Naturally, performance testing and optimisation have both been a major focus throughout this entire period.

On a more personal note, he is married, with three children and two grandkids, and he holds a 5th Degree black belt in a form of martial arts known as Budo-Jitsu, a free-style eclectic method of close quarter combat.

Acknowledgements

We are living in an age where information has never been so accessible, nor the available content so huge in scope. Much of what we do and say is the result of influences from a wide variety of factors, be they technical, social, or emotional, and this book is no exception. One cannot go through life without the help and assistance of others, and it is such a small thing to acknowledge these factors, though they exert such a huge influence on a person's ability to succeed. To that end, it would have been impossible to write this book without the support and love of my family. My wife Michele shows never-ending tolerance to my late

About the Authors

nights writing, researching, and technical tinkering (and sometimes gaming). My children Kristy, Marc, and Elizabeth are a constant blessing to me, and also put up with many late nights, and with me just being a geek. My two lovely grandchildren infuse me with some of their boundless energy, which I desperately need sometimes. My parents bought me my first computer, a Vic 20, so long ago, and watched as I spent hours in front of a seemingly uninteresting screen, and it all began from there. Without all this, I would not have even started down the technical path, and I am constantly amazed at how lucky I am.

Having been in the computing industry for approximately 20 years tends to bring about a real appreciation for where the industry is at today. Developing complex applications was exponentially harder even 5 to 10 years ago, let alone 20 years ago. Distributed, transactional, high-performing applications are easier than ever to develop. To make this process even easier, abstractions on complex technologies are developed, and Services, Frameworks, components, libraries, and runtimes are all mingled together to create the applications of today. Measuring the performance of these applications is still somewhat of a "black art." It is, without question, easier than ever, but if you have ever tried writing your own performance test tools, or using early versions of performance test tools, you will understand how tedious and time consuming it can be. Even with a wealth of information available to us, it is still quite difficult and complex to research the proper steps to setting up a performance rig, what metrics to analyse, how to record, execute, and analyse performance tests, and what to do when problems arise.

During my career I have performed a number of performance testing engagements. Without exception, this involved bringing together substantial amounts of information from various sources such as blogs, articles, books, product documentation, fellow colleagues, and anything else I could find. There was no single cohesive source for the information I needed, and that is how the idea to create this single gospel of performance testing and optimisation information was born. As a result, if you're doing .NET performance testing, you can save yourself countless hours of effort by using this book

A good friend of mine, Wallace B. McClure (also a book author), started me on the path to book writing, and I am indebted to him for providing me with that initial opportunity. It was from that experience that I was able to form a plan for this book and present it to Red Gate. Red Gate has been a great company in terms of technology, and produces an excellent set of both SQL and profiling tools. They wasted no time in moving the idea forward, for which I am deeply thankful. It would be remiss of me not to mention my editor, Chris Massey who has been extremely helpful and responsive throughout the entire book's progress. In addition, my co-author, Chris Farrell, has made this book what it is by not only contributing quality content, but by taking on additional content above and beyond any initial agreements, and allowing this book to be delivered in a timely manner.

Finally, my thanks go out to you, the reader, for taking the time to read this book. I believe it will prove extremely valuable to you, and I look forward to using more high performing applications in the years to come.

Chris Farrell

Chris Farrell has over 18 years of development experience, and has spent the last seven as a .NET consultant and trainer. For the last three years, his focus has shifted to application performance assurance and the use of tools to identify performance problems in complex .NET applications. Working with many of the world's largest corporations, he has helped development teams find and fix performance, stability and scalability problems with an emphasis on training developers to find problems independently in the future.

In 2009, after working at Compuware as a consultant for two years, Chris joined the independent consultancy CodeAssure UK (WWW.CODEASSURE.CO.UK) as their lead performance consultant.

When not analyzing underperforming websites, Chris loves to spend time with his wife and young son swimming, bike riding, and playing tennis. His dream is to encourage his son to play tennis to a standard good enough to reach a Wimbledon final, although a semi would also be fine.

Acknowledgements

I would like to thank Paul Glavich for his brilliant focus, and editor, Chris Massey, for his help and support. Thanks also to my wife and son, Gail and Daniel, the sources of my happiness and inspiration.

Chapter 1: Introduction – The What and the Why

Performance in web applications is clearly very important. The web potentially allows millions of users to access your application simultaneously. How is your application going to cope with such a load? How much hardware do you need to ensure it can handle the required number of users? What happens to your application when its peak capacity is exceeded? These are questions that really need to be answered.

As a business, I want to know that the applications supporting my commercial endeavors can cope with the size and usage patterns of my customer base. I also want to make accurate estimations around the amount of infrastructure required to support my current customer base, and what infrastructure is going to be required to support my *future* customers, based on a projected growth factor. All this can apply to both intranet and broader Internet applications.

As solution developers, we try to write software that is fast, responsive and scalable; but until we can measure and quantify these factors, we can really only make guesses based on the technology and techniques used, and the environment that the application must operate in.

Effective performance, load, and stress testing can be used to answer all these questions. It removes the vague assumptions around an application's performance, and provides a level of confidence about what the application can do in a given scenario. But what do we mean by performance testing, load testing, and stress testing? Often these terms will be used interchangeably, but they actually refer to slightly different aspects of an overall process.

Performance testing

This involves testing the application at increasing levels of concurrent users, and measuring how the system reacts under the increasing load (not to be confused with load testing, which I'll come to in a moment). The concept of "concurrent usage levels" is one which gets thrown around a lot in testing and profiling, and it refers to the number of users that are accessing your web application at the same time.

Typically, a performance test may start out by simulating a low level of concurrent users, say ten, and then increase the number of concurrent users on the system at defined intervals, and measure the effects.

This type of testing is used to examine response times and system usage patterns such as CPU usage, memory usage, request execution time, and a host of other factors which will be discussed in detail later in this book. These attributes are used to characterize the system at various points, and to define patterns in the application's behavior when operating at various levels of concurrent users.

Performance testing is used, not only to determine patterns or characteristics of an application at various levels of concurrent users, but also to determine bottlenecks and operational capacity. This all contributes to the overall capability of an application.

Load testing

Load testing involves executing tests against the system at a consistently high load level or, typically, a high number of concurrent users. The number of concurrent users is naturally relative to each application, but is high enough to present a large load to the system being tested. While load testing is technically a separate, but related, aspect of performance, it can be combined with general performance testing. The primary reason to keep them separate is that you don't really know what a high load is for your application until you begin actually testing it and analyzing the metrics.

This type of testing is often referred to as "volume testing," as the application is tested against high volumes of load for extended periods of time to determine its reliability, robustness, and availability characteristics. In other words, how does your application perform when it needs to handle nearly its maximum capacity for hours, or even days, at a time?

Stress testing

Stress testing is very similar to load testing, except that the focus is on continuous stress being applied to the system being tested. The goal is to examine the system as it is being overwhelmed with applied load. As you can imagine, this can cause the tested system to operate at levels beyond what it's capable of, and the net result is usually some form of failure, ranging from requests being rejected to complete system failure. The primary question behind stress testing is "What is the recoverability of the application being tested?" which also contributes towards the overall availability of the application.

This may seem somewhat unrealistic, but you are trying to determine how the application functions when placed under extreme conditions. Does the application refuse requests and then recover gracefully after a short period, or does the application never recover at all?

This information can be used to confidently answer questions of the "What if..." variety. To take an obvious example, a project stakeholder may ask "What happens if the system gets overloaded with requests? Will it stop functioning?" Stress testing allows you to answer these kinds of questions with confidence and a good degree of accuracy.

Profiling

Performance, load, and stress testing all represent a broad, general approach to determining the performance characteristics of your application. In order to improve application performance, you need to determine what specific aspects of the application need improving. You also need to be able to quantify the performance of isolated parts of the application, so that you can accurately determine when you have improved them.

This is what profiling is all about – getting a performance profile of your application or, more specifically, quantifying the performance characteristics of a "slice" of it.

Whereas the broad-based performance testing will identify slow pages or how the application copes with a high load, profiling will highlight, at a granular level, what methods take a long time to execute, what objects are utilizing excessive amounts of memory, and so on.

With that in mind, there are generally two types of profiling when it comes to .NET applications: performance-based and memory-based. Performance profiling measures how long a method or function may take to run, and memory profiling measures how much memory certain aspects of the application (or even individual objects) use.

Profiling is a crucial part of the overall performance testing process. Performance testing can provide the broad metrics and characteristics required to determine where changes need to be made, and profiling can pinpoint the exact areas that need those changes.

Cost benefits of performance and load testing

Performance testing is a comprehensive and expensive operation. It takes a lot of time to design tests, execute them, and then gather, manage, and analyze the data. Once you have the results and have drawn some conclusions, you often need to make some changes to improve the overall performance of your application. This typically involves changing and refactoring it to improve upon the areas that are performing badly.

In addition, the hardware and infrastructure costs can also be prohibitive, depending on how hard you would like to push your performance tests. If you want to simulate a large number of users, you are going to need an isolated network and enough machines to simulate that load.

This whole process can take a significant amount of time, and means that resources are diverted away from enhancing the functional aspects of the application. It's easy to see why many organizations shy away from performance testing, or only partially address the situation.

This is exactly why it is so important to do it properly. Making sure that your tests are effective and require minimal effort to execute is important to the success of your performance-testing regime. Automation of tests is crucial in this regard, and the collection of results should be as painless as possible. You want to be able to get the results of your tests easily, analyze them quickly, know exactly where to make changes, and demonstrate the benefits to the business.

There are really two major beneficiaries of performance testing. The first is the business. The business, which is typically sponsoring your application, not only has a vested interest in having it perform well; it also needs the metrics you provide through performance testing to ensure infrastructure, budgets, and projected growth requirements are all taken into account.

The second set of beneficiaries of performance testing are the application developers themselves. The metrics can be used to ensure that the development process is not itself generating performance issues. You can ensure that developers are not writing inefficient code, and that the architecture of the application is not an impediment to performance. Regular or periodic testing can ensure that development always stays on track from a performance perspective which, in turn, will cause less stress all round.

So far we have discussed the individual aspects of performance testing and what they mean. This does not mean that we should necessarily execute them in isolation. Given the related nature of performance, load and stress testing, you can run all these types of tests together, provided you carefully manage their execution and the subsequent collection of metric data.

The following chapters in this book will demonstrate how to do exactly that: to provide the most value, for as little effort as possible.

Example scenarios

What value does performance testing really offer to the business? I can most easily describe this by providing two comparative scenarios. One where performance testing was not done, and one where it was. Consider the following two scenes.

Scenario 1

Business Stakeholder

"We currently have a user base of approximately 15,000 users. We expect about 5,000 of these users on the system at any one time. Can the system handle that?"

Solution Architect

"Well, I cannot be exactly sure, but we have used best practices for the system architecture and coding techniques, so it should be able to handle a reasonable number of users."

Business Stakeholder

"What exactly does this mean? I have a budget for three web servers, maybe four, but I am unsure how much we need. How many users can a single web server sustain?"

Solution Architect

"Again, I cannot give you an accurate estimate, but I think three web servers should be enough. I think that one web server may be able to handle around 2,000 concurrent users, so three should be sufficient. If you have the budget for four servers, then that's probably a wise decision to go with, just in case."

Business Stakeholder

"What about our usage peaks, as well as our projected growth? During certain peak usage times, we could experience up to 10,000 concurrent users. We also expect to grow our customer base by approximately 1,000 users per year. At what point should we be purchasing extra infrastructure?"

Solution Architect

"Hmm, I'm not really sure right now. I'll need to perform some investigations and get back to you."

Scenario 2

Business Stakeholder

"We currently have a user base of approximately 15,000 users. We expect about 5,000 of these users on the system at any one time. Can the system handle that?"

Solution Architect

"We have measured the application on a single web server of a slightly lower specification than what is in production. On this machine, we could achieve approximately 2,000 concurrent users with a response time less than five seconds. A system of three web servers could handle this required load with good response times."

Business Stakeholder

"What exactly does this mean? I have a budget for three web servers, maybe four, but I am unsure how much we need. How many users can a single web server sustain?"

Solution Architect

"As I mentioned, one web server, of lower specification than production, can handle approximately 2,000 concurrent users at any one time. A higher specification machine, such as the one in production could handle a little more, perhaps up to 2,100. Three web servers in a load-balanced scenario will easily cope with the desired load. I would recommend that the fourth web server be utilized to ensure adequate breathing space and allow for some growth."

Business Stakeholder

"What about our usage peaks, as well as our projected growth? During certain peak usage times, we could experience up to 10,000 concurrent users. We also expect to grow our customer base by approximately 1,000 users per year. At what point should we be purchasing extra infrastructure?"

Solution Architect

"Our current test metrics show that, while the system could sustain that load, response times may degrade to approximately 10–15 seconds per page. If this is acceptable by the business, then the four web servers you've mentioned will be sufficient. If you want to maintain the 'five seconds or less' response time, I would recommend having two extra web servers on hand to share the load at expected peak times or, if budget permits, online constantly. At your projected growth rate, the system will easily cope for approximately two years, so I would suggest you look at provisioning a new server every eighteen months. You would be wise to assess the customer growth every year to ensure that it has not massively exceeded expectation. If that occurs, the system may become unacceptably slow, and cause the load to exceed acceptable operational capacity."

In these two scenes, it is obvious that the second is the situation that a business stakeholder wants to be in – provided with the answers to all their questions, and armed with enough information to be able to make accurate estimations for budget and infrastructure. Ultimately, being able to properly service the current and future customer base is a very attractive end result for any business. Performance testing provides known quantities for characteristics of your application that, *without* performance testing, are unknown and, at best, rough guesswork.

Sometimes, what seems right can be wrong.

As solution architects, we are presented with numerous theories on best practices, a constantly changing technology landscape, and widely varying opinions from highly rated industry experts. This, on top of a host of other variables, makes definitive statements about the performance capability of our applications very difficult.

There are many good practices to utilize when it comes to flexible, easy-to-test solutions including inversion of control (IoC), design patterns, and tiered architecture, not to mention using the latest and greatest technology. Yet, quite often, the most flexible and loosely coupled architecture comes at the expense of performance, and the allure of utilizing the latest technology can actually be a big performance risk.

Many development-related technologies are aimed at making developers' lives easier and more productive, meaning that a runtime or technology stack does a lot of the work for the developers. But how is this implemented in the runtime? It might be easy to implement and have taken an incredibly short time to develop, but has it cost the application a lot in terms of performance?

Without testing, we can never really be sure. Technologies such as LINQ (.NET Language Integrated Query) enable fantastic productivity gains, but the danger is that they can make complex things too easy. Later in this book, we will have a look at some of these technologies that make developers' lives easier, but which can come at the cost of performance if not used with caution.

Conclusion

As the title implied, this chapter has just been an introduction to what performance testing really means, and what the different aspects of performance testing are. Performance testing, stress testing, load testing and profiling are all singular measures of an application's performance, which contribute to the overall understanding of its usability experience. Quite often, these measures are intangible until your application is deployed and being used by thousands of users, by which time it's really too late to be worrying about how it's going to cope.

The wonderful functionality or fantastic user interface features of your applications will all count for nothing if the application takes over 20 seconds to load a single page.

Clearly, it is extremely important to be able to quantify the performance aspects of your application, not just for the business (although this is one of the most crucial reasons) but also to validate the architectural and technological decisions made about the application itself.

Currently, there is a vast amount of vague, high-level information describing how someone might go about achieving the goals I've mentioned. In the chapters that follow, you will find detailed instructions on how you can achieve these goals, and effectively demonstrate the performance characteristics of your applications.

No longer will performance testing be a mystical "black art," dominated by the all-knowing few, but rather a regular part of your application life cycle, integrated into the development and deployment plan, and producing tangible value.

Chapter 2: Understanding Performance Targets

Identifying performance targets

Naturally, in order to achieve a goal, you first need to understand what that goal *is*. So, before you can determine whether your application performs well, you need to understand what that means in terms of the metrics your application needs to produce.

Whether or not your application performs well is a relative target; not all applications are the same, so it stands to reason that the measure by which an application's performance is tracked changes, based on its requirements. This is where the business side of things comes in.

It is easy to say that a given business has a customer base of 5,000 concurrent users, but what does that really mean? It means you need to ask yourself questions like those below.

- If your application is being used by the entire customer base, what is the typical usage pattern?
- What percentage of users are performing searches?
- What percentage of the users are buying goods?
- What percentage of users are simply browsing around?

Having an accurate determination of user behavior is absolutely critical to determining whether your application can meet the performance needs of your customer base, and this is what the business needs to decide. This task is made a lot easier if there is existing behavior that can be used as an example. If there is no existing data on "typical" user behavior, then an educated guess obviously needs to be made. Also, given that technical staff usually have a biased view of the application usage, it is probably best if the business can provide some metrics around what users are doing on the site (by that, I mean what percentages of users are performing what actions on the site).

Structuring test breakdowns

An application can have many functional paths, and the amount that a functional path is exercised is based upon typical business scenarios and user behaviors. These assessments are made far more accurate if based on statistical evidence, such as evidence based on past activity or, perhaps, analytical statistics gathered from companies specifically engaged to measure usage patterns of the site. This kind of data will provide a perfect template from which to draw test breakdowns. If no such analytical evidence is available, then the business must provide as accurate an estimation as possible around usage patterns and functional paths to the site.

Quite often, a site map will be produced as part of the functional or technical specification of a web application, and this can be used as the basis from which to ascribe weighted percentages to customer use (although they can often contain too much detail). An example of a usage diagram might look something like Figure 2.1.

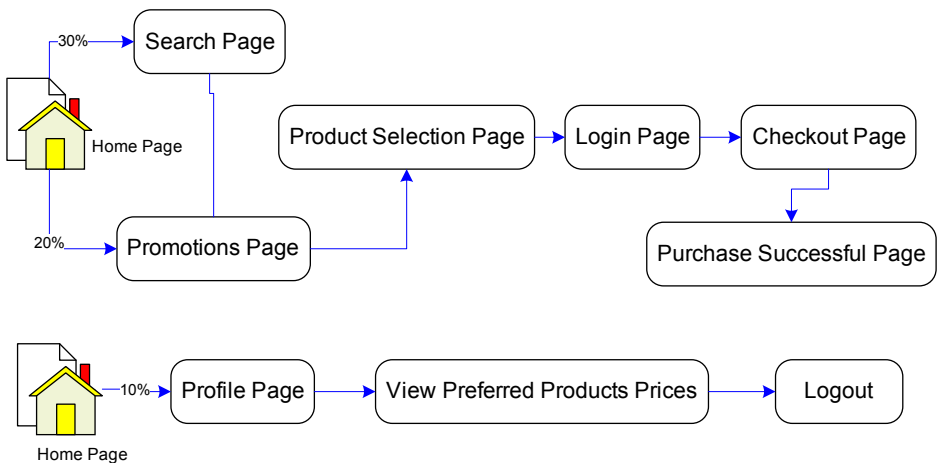


Figure 2.1: Example Usage Patterns.

While this example is somewhat simplistic, it is indicative of the type of diagram or "breakdown" required when structuring your performance tests. It is important to ensure that you exercise any aspects of the application which are deemed "heavy" in terms of performance (i.e. performing a lot of work) to gain metrics on just how much of an impact they are having. However, it is also important to note that performance tests are not like unit or integration tests, as it is not vitally important to cover every functional path and achieve high code coverage when running performance tests.

For example, consider the fact that different functional paths and usage scenarios quite often exercise similar code paths. In the same way, from a performance perspective, the same or

very similar performance can also be achieved from different usage patterns. Obviously, it is necessary to model the different usage patterns that the business has identified, but trying to exercise every single aspect in fine detail will often end up causing a lot more work than is necessary, while achieving comparatively little extra value from a performance-measurement perspective. Generally speaking, performance testing is a broader approach than unit, integration, and functional testing.

Determining what load to target

By this point, we have a good understanding of what tests we need to run, and how these tests are going to be distributed across a performance run to accurately simulate the target application's performance requirements.

What we still need to determine is how many concurrent users the application needs to be able to handle at various times. This is not a simple static number, though, as a typical application needs to be able to deal with concurrent user load in a number of ways. Specifically, it needs to be able to:

- remain responsive during a normal day's use, with a "typical" amount of customer or client concurrent usage
- remain responsive during the high peak times, where almost the entire user base might be using the application concurrently
- be resilient enough to sustain extreme loads without complete failure, and return to normal operation when stress or load levels return to normal.

There are a few points here which are open to interpretation. For example, when the application needs to "remain responsive," exactly what does this mean? Unfortunately, this is something that can only be answered by the business in consultation with technical staff. Quite often, if you ask the business how responsive each page in the application should be, they respond with "sub-second" as a default answer for all pages. While this *might* be achievable, there may often be some aspects of the web application that require serious computation, and are thus very hard to deliver in sub-second time-frames.

It is also important to allow for some variance in these response figures at different load levels. The response times during normal daily use may differ quite significantly compared with times when the server is under extreme loads. In an ideal world it would be preferable to have a consistent response time across all load levels, but this can be costly and sometimes extremely difficult to achieve. If it is achievable (as with the ideal sub-second response time), it might require significant additional computational power, and this cost will need to be quantified and justified to the business. Indeed, the business needs to be involved more or

less throughout this entire stage of the profiling process, as you will need to be able to make several informed judgments before you can proceed with the actual testing.

Note

It is important to specify a frame of reference regarding what are acceptable response times for your pages. This is also why it is important to involve technical staff in your consultations, so that the mechanics behind each page and function can be given due consideration when determining how they should perform.

So, what are the metrics we actually need to identify in terms of concurrent users? There are no hard-and-fast rules here, as this is often dictated by the application and what is important to the business. Generally speaking, you will almost certainly be looking at:

- the number of typical concurrent users that represents average usage
 - This might be expressed by number of page views per hour, or number of total users visiting the site per day (assuming an even number of page views per user). As long as a metric-over-time is provided by the business, the average typical concurrent user rate can be calculated from this.
- the number of concurrent users during peak times
 - This figure represents the peak usage times of the application, and is an estimation of what the absolute peak number of concurrent users is. It is obviously important that the application can handle these peaks as well as handle the normal day's usage.
- the project growth of the user-base over time.
 - This is important for the application to accommodate future growth and not need sudden provisioning of extra resources soon after implementation.

In addition to specifying the accepted response times during these various concurrent use cases, it is also worth considering differences in these times across the different pages on the site. In most applications, there is a small subset of pages which have exceptionally high traffic (such as login pages, home pages, etc.) and these pages should often be considered individually for testing purposes. In any case, different pages will frequently have a different computational cost, so it may be acceptable to have longer response times on some pages.

This may not be the case for most of the pages so, where certain pages involve more resources or computational cost, special consideration must be given to achieving the desired response times. At the very least, the cost of achieving the response time target should be known, so the business can properly make a value decision.

There does not need to be a formal document listing these considerations, but some common location for reference and comparison purposes is a good idea. This might be a wiki page, a Word or Excel document, or some other listing of the agreed and expected targets and response times. A simple example of this is shown in Table 2.1.

Scenario	Concurrent users	Accepted response times	Pages applicable to	Projected growth in concurrent users	
				Next year	Next 2 years
Typical usage	1,000	< 5 seconds	Reporting.aspx	1,500	2,500
Typical usage	1,000	1-3 seconds	All (Except reporting.aspx)	1,500	2,500
Peak	2,500-3,000	< 8 seconds	Reporting.aspx	1,500	2,500
Peak	2,500-3,000	3-5 seconds	All (Except reporting.aspx)	3,500-4,000	4,000-5,000

Table 2.1: Example of response time expectations.

This table show a very simplistic example of what may be produced by discussions between the business and technical staff around expected response times and concurrent user load. The goal of this is to produce explicit guidelines of what is acceptable by the business at various points of the application's usage, within the context of the current user base. Without explicit guidelines in these areas, all the relevant test conclusions are subjective and open to interpretation. Bearing that in mind, it is once again up to the business, in conjunction with consultation from appropriate technical staff, to come up with realistic and achievable goals that meet business needs.

Now that we have clear expectations around application performance, you would think we have set our targets to achieve when doing performance testing, right? Not quite.

Contingency in your estimations

As part of a general rule, when the table of estimations has been produced around concurrency expectations and response times, it is important to emphasize that these are just estimations. They have plenty of potential to be incorrect. In fact they probably *are* incorrect but, because we have used whatever metric data we can, as well as knowledge of usage patterns and systems, they should be reasonably close to reality. As a result of all

this, even using these estimations as our limits for performance testing is not going to be accurate, either. I generally recommend that you double the concurrent load targets for any given scenario.

Why should you double your estimates, you ask? This can be thought of as our contingency component. In many estimation processes, such as when you judge the time and effort taken to complete a project, some level of contingency is usually introduced to cater for errors or unknowns in the estimation process itself. If we double our performance targets, it is reasonable to assume that, if we can hit them, then even if our initial estimations were incorrect by a small margin, those errors are accounted for. We can then be confident, not only that the application does meet the required business goals, but also that it can handle more load than anticipated, and performs well within our set bounds.

The performance targets in Table 2.1 may seem excessive once doubled, but remember that part of the purpose of performance testing is to give the business a relatively accurate determination of hardware and infrastructure requirements for current and future use of the application. Using doubled performance targets clearly ensures that we cover current requirements, future requirements, and also any contingency that either the business or technical estimations may have failed to address. It provides a safeguard in the sometimes tenuous game of estimation.

As a bonus, this will obviously also guarantee responsiveness under normal circumstances; if the system maintains acceptable responsiveness under double the expected load, then it will be even more responsive under the originally estimated load.

The previously shown table, and its double, are by no means the only way to express estimated loads, current needs, projected growth, and performance targets. These are provided simply as examples, and you can use them as they are, or find different ways, that suit your individual needs, to express your targets. The main point here is that it is absolutely essential to ascertain these targets before any testing is performed. If you don't, then there will only be a vague understanding of what needs to be achieved in the performance testing process. As I said, performance testing is an expensive process, and the need to gain valuable results, as opposed to ambiguous results which do not allow proper conclusions, is of paramount importance.

One final thing to consider is the percentage of new users accessing the site, compared to returning users. This will have implications in terms of browser caching efficiency, and will affect how many requests are issued against the server. The more returning users visit the site, the more data will be cached by the clients' browsers, and so fewer requests are likely to be issued against the server for resources within a particular page.

This will also be dependent on the type of application; this metric is quite important for public-facing web applications, but intranet-based applications may place less significance on it. Often, to present a worst case scenario, the amount of new users will be assumed to be 100%.

This means that each test will consistently request all resources for a page whereas all common web browsers *do* cache resources.

Estimate the mix of browsers for your web application

Finally, in a web application, it is also important to be able to estimate the percentage of different browsers that will be used to access the website. Different browsers from different vendors naturally all have different performance characteristics, and therefore impose different performance factors on the site.

If the website is public-facing, generally the percentage of different browsers can be gleaned by the respective market share of each browser. Quite often, the business will dictate which browsers should be used, and even which version will be supported.

If the website is intranet-based or has its visibility limited to within certain units of the organization, then the organization in question will often have set standards about what browsers are permitted as part of the standard operating environment.

The final outcome of all these estimations is that you will be able to record tests that exercise accurate, or at least realistic, usage patterns of the site. Once these are recorded, you can then weight the results by applying percentages according to how often each test is executed as part of the entire load test, which will be essential in establishing which result-sets are the most relevant for your purposes. In addition, you can also specify how much each browser is simulated within the overall load test. Within Visual Studio Team Test, you can end up with a load test, specifying a test and browser mix, looking something like Figure 2.2.

We will discuss in detail how to set up the test percentage and browser mix in a later chapter.

What data do we measure?

We have now identified our performance targets across a number of scenarios, and we have also identified the response times required across them by the business. Now we need to establish what metrics we use to compare against our targets.

A huge variety of metrics are measured and analyzed as part of performance and load testing. For comparative purposes against the targets that were identified earlier, we are primarily concerned with a few key metrics which will give us an immediate idea of the application's performance. These are **Time to First Byte** and **total page response time**.

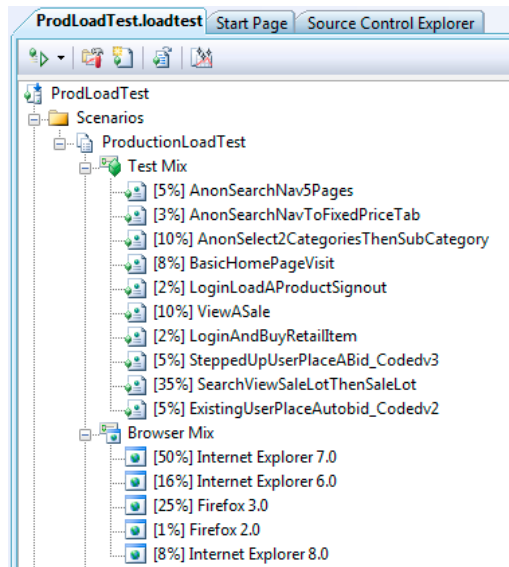


Figure 2.2: Sample test and browser distribution.

Time to First Byte

Time to First Byte (TTFB) represents the time it takes a server to issue the first byte of information, typically in response to a request from a web browser. This time covers the socket connection time, the time taken to send the HTTP request, and the time taken to get the first byte of the page. It is a good indicator of the responsiveness of the web application, as the server must receive the request, interpret it, execute the ASP.NET pipeline to process the request, and produce a response.

This is one of the primary metrics to use when determining how responsive a site or web application is. A large TTFB value means that a typical user will see no activity in their browser (apart from whatever "waiting" indicator the browser uses) for a long time, until that first byte of information is received from the server and the browser can start parsing.

This is also typically a good indicator of how fast the web application can process the requests made against it, as no response will be issued until the web server/ASP.NET has finished processing a given request. There are caveats to this, but I'll cover them alongside analysis in later chapters.

Total page response time

The total page response time is often referred to as simply "response time." This metric includes, not only the TTFB time described previously, but also all the dependent requests required to completely load and display all aspects of a web page. This can include items such as images, JavaScript files, and Cascading Style Sheet (CSS) files.

In contrast to the TTFB measurement, the total page response time measures the time it takes for a page to completely finish loading all resources required to present the page to the user. This may include non-functional aspects as well, such as tracking images hosted on external sites. It is important to quantify the effect that external tracking mechanisms can impose upon the site. Once this is done, it is valuable to remove this component during performance testing, to get a more accurate view of the site's performance. External tracking mechanisms are normally beyond the control of the application and cannot, therefore, be modified or improved.

To further illustrate these points, the following diagrams show some TTFB and total page response time breakdowns. The first diagram represents a personal, hobby site, [HTTP://WWW.THEGLAVS.COM](http://www.theglavs.com) and the second site represents the Microsoft main site at [HTTP://WWW.MICROSOFT.COM](http://www.microsoft.com).

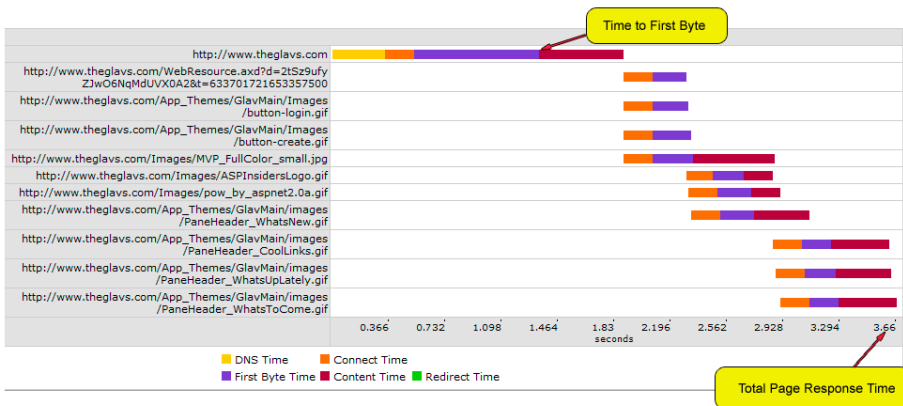


Figure 2.3: www.theglavs.com response times.

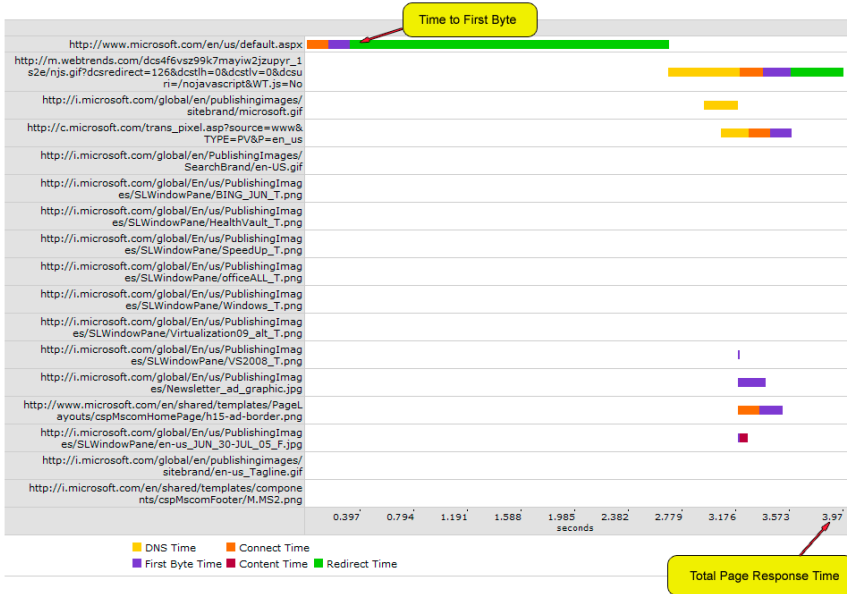


Figure 2.4: www.microsoft.com response times.

From the previous two diagrams, it is evident that the Time to First Byte and the total page response times can vary quite considerably. This will be dependent upon the number of other resources and artifacts that are present on the particular page being measured. It is important to be able to quantify these differences, as your web application may respond very fast on its own, but the dependent requests and resources in the page may be degrading the performance considerably.

What about average response time?

The average response time for a web application is often a misleading metric, as there is ambiguity around what the average time actually means.

- Does it refer to the average TTFB?
- Does it refer to the average total page response time?
- Does it include static resources such as CSS and image files?
- Does it include the low end figures at low levels of load or at other different times?

It is worth bearing in mind that serving static files will be much faster than processing a request through the full request execution pipeline so, if static files are included in this metric, then average response times will appear faster than they really are. The result is obviously more ambiguity, and the metric will provide no real correlation to page responsiveness and overall application performance.

There is also the question of the point at which the metric samples are taken to determine the average. A common practice is to use percentile brackets to determine the average response time. For example, if a 90th percentile was used to determine the average response time, this would mean that, out of 100 requests, ordered from best to worst times, the requests in the last 10% of requests (that is, the 10 worst-performing requests) are used to find the average.

Because of this ambiguity, the average response time is generally best used to compare against previous averages for the same time period, but only for the purposes of determining if the latest performance run has shown improvement or degradation compared to the last run (in other words, it's used as a relative measure).

This can be a useful metric, though I always recommend that this percentile bracket should be used in conjunction with examining the more specific TTFB and total page response times discussed previously. Equally, you don't necessarily have to use the 90th percentile – the 85th percentile could just as easily be chosen to determine averages. That being said, I recommend that you use the 85th, 90th and 95th percentile brackets, as these provide a valuable insight into the worst case scenario of response times for whatever is being tested, in terms of both Time to First Byte and total page response time.

Sweet spots and operational ceilings

We've seen that the performance targets which have been identified by the business represent acceptable response times under various load conditions. Irrespective of the current system load, these response times are often referred to as the "sweet spot," and are the response times and usable states that best serve the users of the application.

The sweet spot may initially be nowhere near what the business requires from the application. The response times expected of the application may initially be at concurrent user levels far below what is deemed necessary to serve the customer base. It is crucial to identify the sweet spot, and how far from that target the application currently is, as you'll need to make those two states match before business requirements can be met.

However, the sweet spot is just one aspect of the application. It is also important to know what the limit of the application is, and whether it is resilient enough to cope with extremely large user loads.

This is the stress-testing aspect of performance testing and analysis, and requires you to ask the questions below.

- How long can the application cope with relatively high concurrent user loads before it becomes totally unresponsive?
- In addition, what characteristics does the application exhibit at these high concurrent user loads?

These are important questions in determining the operational characteristics of the application. The number of concurrent users (or "load") that the application can withstand before becoming totally unresponsive is referred to as its "operational ceiling." That is, the ceiling or limit at which the application can operate before failure. This limit will typically involve excessive response times that make the website practically unusable, but this metric still serves as a good comparative indicator against previous performance tests. It also provides valuable evidence as to what will happen when the application experiences a larger load than it can handle.

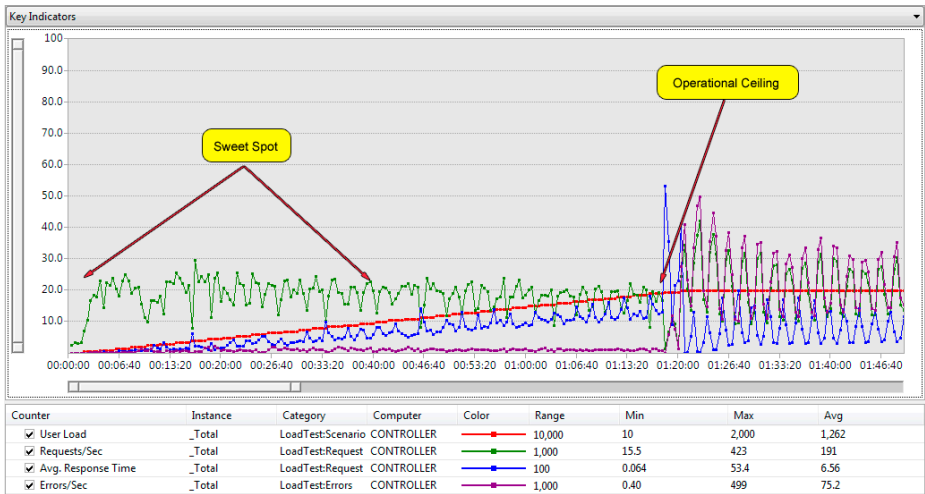


Figure 2.5: Example performance run graph.

Figure 2.5 shows an example performance run. The red line represents concurrent user load, the blue line represents response time, the purple line represents errors per second, and the X-axis represents the time component of the performance test. It is clearly apparent where the operational ceiling is, as errors-per-second and response time make a sharp change at approximately one hour and twenty minutes into the run.

In addition, we can discern that the sweet spot for this application (in this case, a Time to First Byte response time of less than five seconds) is between the start of the performance run and approximately forty minutes into the test.

Using Visual Studio Team Test we can drill into these results to determine the concurrent user load at the sweet spot as well as the operational ceiling. This process will be described later in the book.

Conclusion

The objective of this chapter is to provide a understanding and a framework around determining what targets need to be identified for a web application to be classified as a well-performing application. This is a relative term that needs to factor in the user base and the business which the application serves. There are no "right" answers, but the more experience a developer gains, the more honed their instincts will become.

Having these targets identified means that the relatively expensive exercise of performance testing has a well-defined set of goals which can be measured and tested against. In addition, the business has a clear set of measurements from which to determine whether the application meets its requirements. These measurements provide a degree of confidence in a technical procedure that is typically unfamiliar territory for businesses.

The sweet spot and the operational ceiling aspects of the application provide valuable evidence into how the application performs at various load levels. With this evidence, as solution architects, we can provide informed conclusions around the application capabilities and also how estimate much time and effort is needed to achieve the goals of the business.

The following chapters will take you through the technical steps required to ensure that the business and technical staff alike can ensure that the application performs as required.

Chapter 3: Performance and Load Test Metrics

What metrics do we need?

The purpose of running performance, load, and stress testing is to gather metrics about a system so that you can determine which parts of it are performing well, and which are not.

This sounds simple enough, but the myriad combination of data-types to record can make choosing which to use difficult. For example, if the wrong set of metrics were recorded for a performance test run, the system might appear to be able to cope with a given load relatively easily. However, in reality the system may have been experiencing severe problems in areas that simply weren't measured.

Equally, one of the most frustrating things is to have just enough data to show that a problem exists, and vaguely where it is, but not enough to provide accurate information as to why.

The short answer as to what metrics to record would be that recording everything possible is ideal. Indeed, if at all possible, then this is a fail-safe approach to ensuring you have all the data necessary for analysis. However, as you can imagine, this is often just not practical for any one of a variety of reasons. When discussing performance and load test metrics, the data gathered is quite different from data gathered during application profiling. This chapter will deal primarily with the former. I'll discuss profiling data in the context of profiling itself (and the associated toolset) in Chapter 6.

To start with, we'll deal with the most basic metrics that will provide the quickest indications of application performance in most typical scenarios, before moving on to more specialized metrics for given scenarios.

Basic metrics

So what is required to meet the most diverse set of needs? There are some basic metrics which are important at all levels of testing, regardless of whether data is being collected from a database server, application server, web server or even a workstation running a web browser to access a web application. Most of the data is gathered via the performance counters that will be discussed in Chapter 5.

These are typically accessed using the PerfMon tool (**perfmon.exe**) or via Visual Studio Team Test. Visual Studio Team Test also collects additional metric data specific to web applications.

The most common (and mandatory) counters required at any level of testing are CPU utilization and memory usage. Both are early indicators of problems in an application.

High CPU utilization *can* indicate that an application is performing tasks very inefficiently, or is perhaps running computationally intensive tasks in unexpected ways; as a benchmark, an application that constantly maintains more than 90% processor utilization would be considered to have high CPU utilization. Although today's high-level languages and frameworks provide constructs that are easy to implement, it is often not apparent what processing is required to achieve the desired functionality.

High memory utilization can indicate that an application is not using memory efficiently, or is perhaps not releasing resources appropriately. There are obviously instances where using a lot of memory is required, but not releasing that memory as soon as possible is a serious issue, and this is how memory leaks can manifest. As an application is used over time, a memory leak causes memory usage to increase steadily until the available resources are exhausted, and since running low on memory obviously has a big impact on system performance, it is imperative that memory be managed correctly. It is a common misconception that because of the .NET garbage collector silently operating in the background, cleaning up memory, memory leaks cannot occur. This is far from the truth. Items such as static objects, event handlers referencing shared data and many other things are ways in which the .NET garbage collector can interpret an object as in use, when in fact it is not. This can build up over time and cause memory issues. Later in this book, we will look at the common mistakes with respect to memory and performance issues, what to watch out for and how to overcome them.

CPU and memory usage are also applicable at all levels of an application, regardless of physical topology. Even a user's system, accessing an application via a web browser, is a good candidate to record CPU and memory usage. High indicators in this situation could indicate inefficient JavaScript being executed in the browser on that machine, for example.

CPU utilization and memory usage are basic indicators that should form part of every metric set recorded, regardless of system role. System performance problems will almost always manifest via one of these broad metric counters, indicating the need to investigate further. PerfMon can be used to gather this data and, in fact, defaults to capturing processor utilization (amongst other counters).

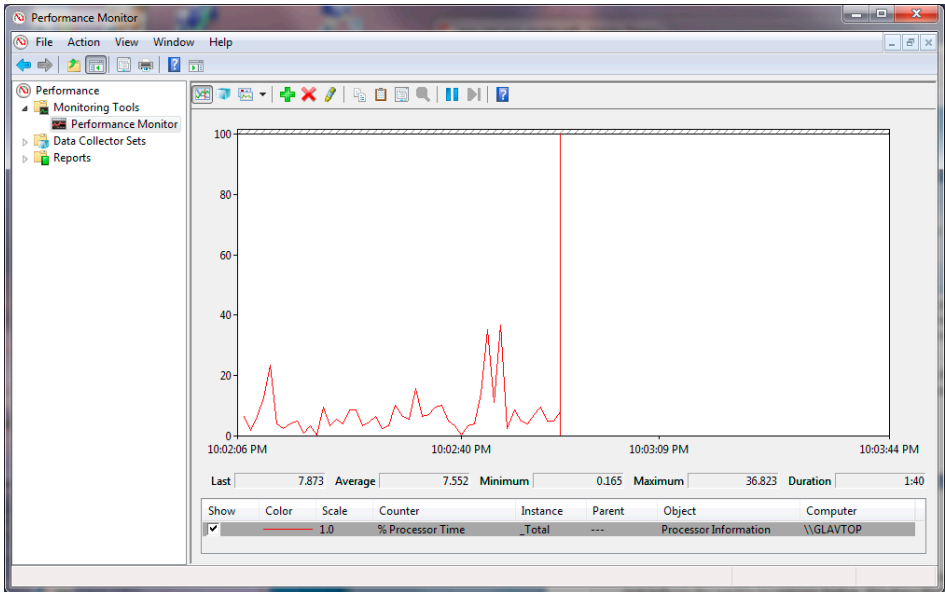


Figure 3.1: Default PerfMon Counters – Windows 7 / Server 2008.

For the basic performance counters, **% Processor Time** in the processor category and **% Committed bytes in use** (for a high-level view), **Committed bytes**, or **Available Mbytes** in the memory category are sufficient for initial analysis.

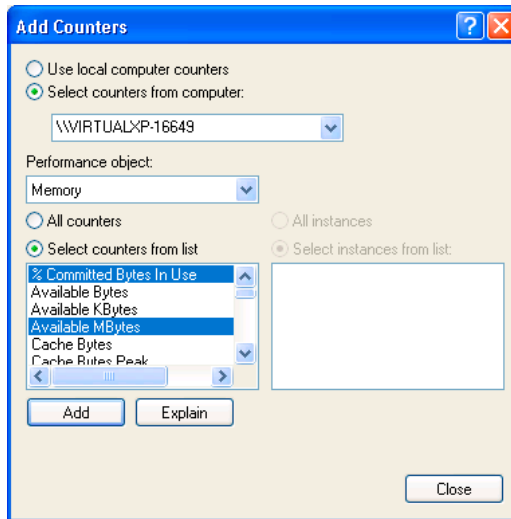


Figure 3.2: PerfMon memory performance counters.

Much like PerfMon, Visual Studio Team Test will capture CPU utilization and memory usage in every performance test run by default. However, remember that this relies on remote WMI (Windows Management Instrumentation) communication, and sometimes this cannot work. Also, performance metrics may need to be analyzed by teams or individuals who do not have Visual Studio Team Test or access to the results of the performance tests; hence the need for separate PerfMon-recorded data.

Web application basic metrics

In addition to the basic CPU and memory counters, web applications are often broadly measured by two other characteristics, response time and requests per second. Response time refers to how quickly the server can provide a response to the browser, and requests per second indicates the throughput of the server, and shows how many requests it can handle every second. Both metrics provide an indication of how efficient an application and server are at processing requests and providing responses. Low CPU utilization and low memory usage will not mean much if a web page takes a long time to load.

Quite often when designing web applications, a business will specify that a particular page is considered to be performing well if the response time is less than a certain amount, usually measured in seconds. For example, the business can specify that the home page of the web application must load in less than five seconds.

At this point, it is important to note that "response time" is a broad term that can be interpreted in a few ways. Since part of the purpose of performance testing is to remove ambiguity about an application's performance and the business's expectations, it is necessary to clarify exactly what response time means.

A web page is typically made of many assets, including such things as the HTML itself, images, Cascading Style Sheets (CSS), JavaScript, and many others. A web browser will not load all these assets as one sequential block and subsequently display them on screen, but will rather load the page in parts. The HTML is downloaded first, and then any referenced assets are requested. These secondary assets are often referred to as "dependent requests," and are usually accessing static resources such as CSS files, image files, and so on.

Static resources are typically delivered much faster than the HTML markup itself, since no real processing is required to serve them to the browser; they are simply loaded from disk as requested. In addition, Internet Information Server has extensive support for caching these requests, making accessing them even quicker.

By contrast, the application is typically required to perform some processing before sending the resulting HTML to the browser. In ASP.NET webforms applications, the application will go through the full page event life cycle (in simplified terms, consisting of 18 individual steps) before finally rendering some content. Just as an example of just how much is involved in this,

when an ASP.NET webforms page is requested, the following (albeit simplified) page life cycle is executed:

- `ProcessRequest`
 - `DeterminePostBackMode`
 - `PerformPreInit` and `OnPreInit`
 - `InitialiseThemes` and `ApplyMasterPage`
 - `ApplyControlSkin`
 - `OnInit`
 - `TrackViewState`
 - `LoadControlState` and `LoadViewState`
 - `OnPreLoad`
 - `OnLoad`
 - `RaiseChangedEvents` and `RaisePostBackEvent`
 - `OnLoadComplete`
 - `EnsureChildControls` and `CreateChildControls`
 - `OnPreRender`
 - `SaveControlState` and `SaveViewState`
 - `RenderControl`
 - `BeginRender` and `Render`
- `EndRender`

Clearly, a lot of processing occurs before the actual rendering of any content to the browser, so the efficiency of the application code executing during this life cycle is what will determine the overall response time of the page.

While the life cycle of an ASP.NET MVC request is not as substantial as a webforms request, there are still significant processing steps required to generate the HTML response.

Once the browser receives the first byte of HTML, the client can be assured that the server processing is complete and the results of processing are being sent to the browser. This time is referred to as Time to First Byte (TTFB) and is one of the primary measures of response time in web applications.

Given the amount of processing required for ASP.NET to send a response to the browser, it is easy to see how static resource requests can be much quicker than any page. This is important to note as, when response times are averaged across a performance run, only page response times should be factored into the calculation. Including static resource requests will cause the average figures to look much better than they really are!

Static resources will be addressed later in this book when dealing with Content Delivery Networks (CDN) and other mechanisms to help static resource load times.

To illustrate the point, this effect is highlighted in the following graph of a website response time report, issued by a service called "Gomez," a paid-for service for measuring response times, provided by Telstra. Many companies provide similar services and associated reports for monitoring websites.

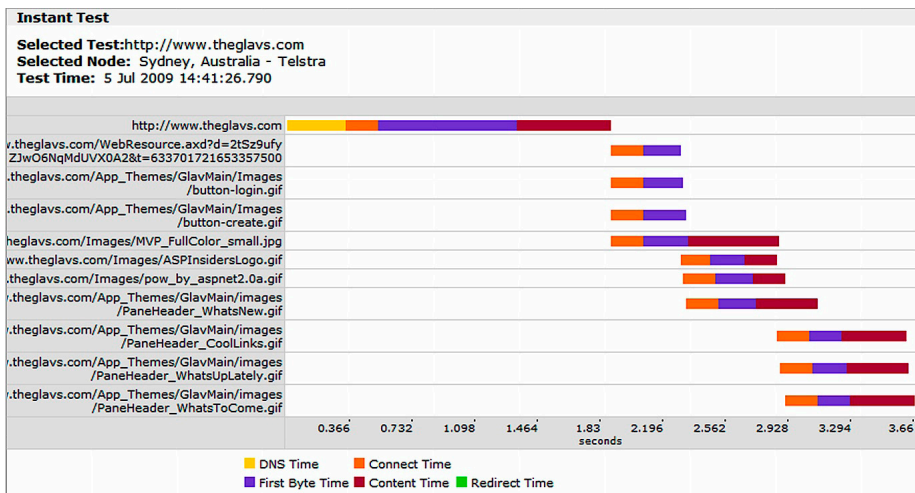


Figure 3.3: Response time report.

The end of the purple bars indicate the TTFB response time, and you can see that the initial request to <http://www.theglavs.com> takes considerably longer than the dependent requests that comprise the rest of the page.

This is why it is important to only factor in the main request TTFB time. The dependent request response times should not be ignored, as they contribute to overall page load time. Some dependent requests may actually be the cause of long overall page load times but, to improve the performance of the application, only the page itself should be considered. Later chapters in this book will deal with the issue of dependent requests and how to improve load times for these artifacts.

Visual Studio Team Test provides a convenient way to analyze these key metrics. When a performance run is executed and the test run data loaded into Visual Studio, the results are displayed in a series of graphs for easy analysis, as you can see below.

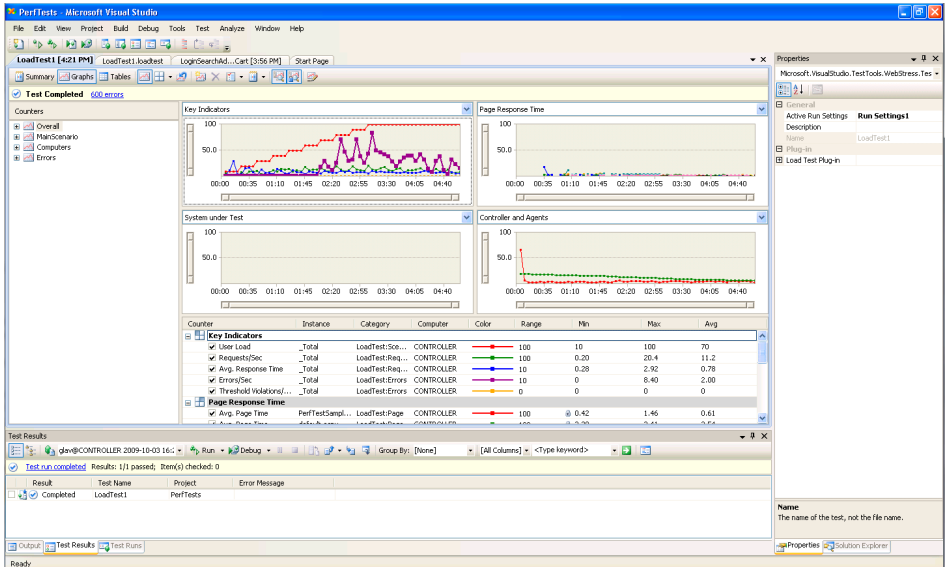


Figure 3.4: Performance run initial result display.

The areas of concern for this example are the **Key Indicators** and **Page Response Time** graphs, which can be easily focused on using the **2 Horizontal Panels** option from the **View Graph** button.

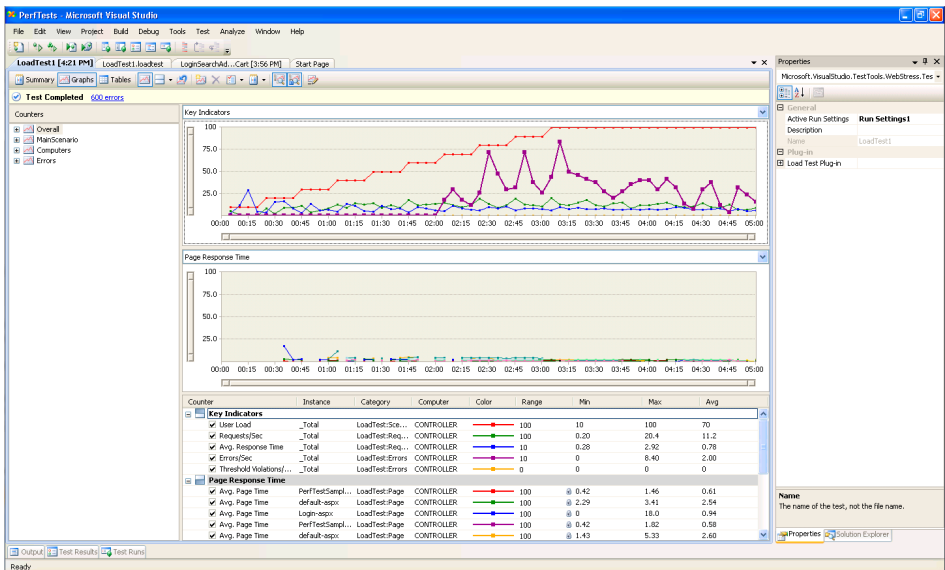


Figure 3.5: Dual graph result view.

The available data summaries will automatically adjust to only display results from whichever graphs you happen to be focusing on at any given time.

What to look for

The basic metrics that have been discussed can provide a good indicator of an application's performance at a quick glance. There are other metrics to factor in, which will be discussed later in this chapter, but these key metrics are a good start for determining if an application will meet its performance criteria. In combination, they are the best way to quickly assess if an application performs acceptably, without going through the time consuming task of analyzing all the possible metric data in detail.

CPU utilization

Naturally, the CPU utilization should ideally be as low as possible. If the CPU is being measured on a database server, then the usage should remain low at all times; an average of 20–30% is generally acceptable. Anything over this could indicate that the database server will begin to be overloaded and exhibit slower than usual performance.

An average CPU utilization of 20–30% on a web or application server is excellent, and 50–70% is a well utilized system. An average CPU utilization above 75% indicates that a system is reaching its computational capacity; however there's usually no need to panic if you have an average of 90–95%, as you may be able to horizontally scale out by adding an additional server.

Obviously, in this last scenario, adding extra load may mean the system consistently achieves 100% CPU utilization. For a web or application server, this is actually not that uncommon, and the ideal is to have a stateless system which can be horizontally scaled as required. As a matter of fact, if a web server is averaging 100% CPU utilization, but the database server's load is only 30% CPU utilization, this is actually a good scenario to be in. This means that the database server has capacity to serve more load, and that the web server is being well-utilized. Simply adding an extra web server into the farm at this point would be an easy, relatively predictable way to address the web tier's capacity to handle extra load.

Some tasks, such as cryptographic functions, are computationally intensive and will cause CPU "spikes" even at a relatively low load. In these instances, it is important to design the system in such a way that it can be easily scaled out appropriately, and not be tied to a single server (which is known as having Server Affinity.)

Memory utilization

A system with no available memory will not be able to cope with any more work to do, and will potentially be unstable, so it's important to ensure available memory is monitored, and remains at acceptable levels.

As I mentioned in passing earlier, it is important to ensure that memory consumption does not steadily increase over long periods of time until there is none available, as this usually indicates a memory leak.

Memory utilization is obviously relative to the amount of memory on the system in question, and as long as the available memory remains above approximately 25%, then the system should have enough "head room" to operate efficiently. Again, this is not an exact figure and a system can operate with less than that, but this can indicate that memory thresholds are being reached and any spike in activity or load could cause unexpected (and often undesirable) results. At best, paging will occur, wherein memory will be read and written from disk, causing the system to operate very slowly. At worst, further load or requests will be unable to be serviced, connections will be refused, memory exceptions will occur, and the system's reliability will be compromised.

In .NET, memory usage should ideally follow a predictable "saw-tooth" pattern. This is because memory is allocated during normal program execution and then, at certain points determined by the .NET garbage collector, the memory is reclaimed. When objects are no longer in use, or are out of scope, they are removed from memory, and the memory is returned to the system for use. The following screen shot shows a typical graph of memory usage for an application, made using a tool called CLRProfiler, a memory profiling tool freely available from Microsoft at [HTTP://TINYURL.COM/CLRProfiler](http://tinyurl.com/CLRProfiler), which will be discussed in Chapter 6.

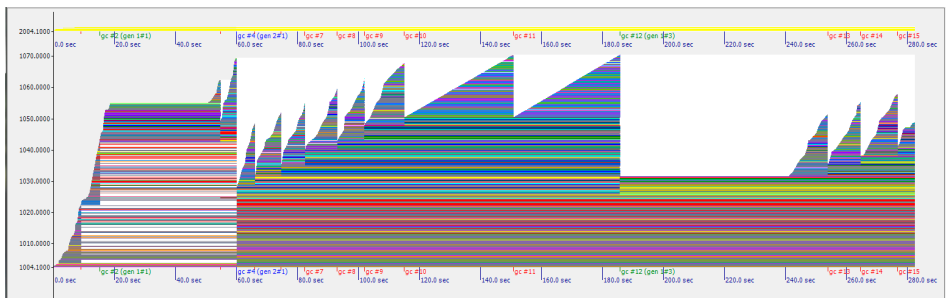


Figure 3.6: "Saw-tooth" memory usage pattern.

The saw-tooth pattern represents memory being allocated, peaking, and then being reclaimed by the Garbage Collector. Memory usage then climbs again as objects are allocated, the garbage collector initiates another collection, and the cycle continues.

What we *don't* want to see is a saw-tooth pattern that is ever increasing:

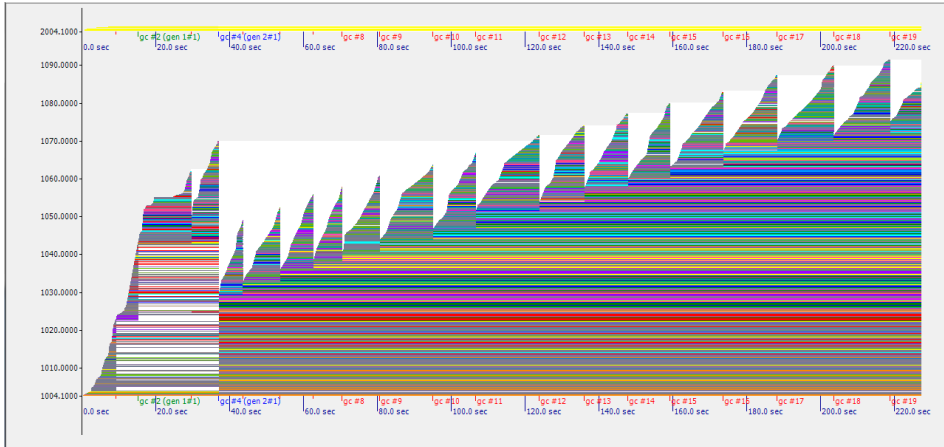


Figure 3.7: "Saw-tooth" memory usage pattern – potential memory leak.

Figure 3.7 shows a classic indication of a memory leak in an application, but this can occur in both web applications and desktop or service applications.

Both of the previous graphs present a low-level, detailed view of memory usage. If memory usage was viewed at a higher level using a tools such as PerfMon or even Visual Studio Team Test, the saw-tooth pattern would not be as evident, but the general pattern would remain the same – a relatively flat horizontal line for good and predictable memory usage by an application, and a line trending upwards for a memory leak-type situation.

A database server should typically remain at a relatively constant level of memory usage without too much variation. Again, this is dependent on other system activities such as scheduled tasks, but memory usage should, in general, remain even. The average amount of memory used will naturally depend on how much work the database has to do, and the nature of the queries themselves. Memory will occasionally dip and spike but should always return to the normal operating level.

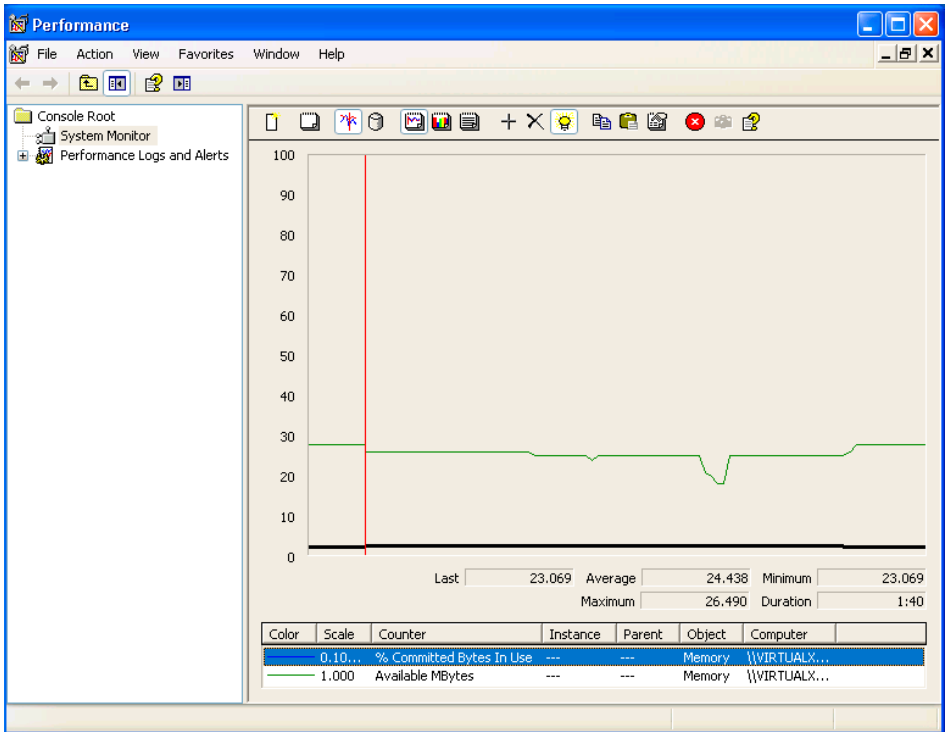


Figure 3.8: PerfMon – Even memory usage on a database server.

Response time

Response time (as measured in TTFB) is relatively easy to assess. Generally, any response time over 3–4 seconds is perceived as slow, although this obviously varies for each individual case.

When developing business applications, it is best to let the business decide what is an appropriate response time, although this metric must be determined in conjunction with the development team. It would be potentially unrealistic if a business were to stipulate that every page must respond in less than one second (although this is possible, and some businesses manage it!). Latency and the operations a page performs play a huge part in these decisions. If a page needed to produce a complex and computationally heavy report, then this response goal would be very hard to achieve without some high-end servers and computational equipment. Business value and realistic response times in these types of scenario are clearly matters for negotiation.

Creating a baseline

Before performance tests can be properly analyzed, it is essential to establish a good way of comparing performance data to determine if improvements have been made from one test run to another.

The first step is to establish a "baseline" performance run against the application. A baseline performance run is a performance test run executed against the application for the very first time, without specific performance modifications (aside from normal development). In short, the application as it currently stands.

Always keep track of your baseline run data, as this will allow future performance runs to be compared to it to determine if performance has altered anywhere. The performance run can be named as a baseline within Visual Studio Team Test; alternatively, a separate list or spreadsheet can be used to catalog performance runs against their purpose or context.

Utilizing a spreadsheet, with each run, its date/time, and any details such as performance modifications made, is an extremely valuable way to collate and manage performance test run data. It then becomes easy to quickly glance over the list and view what changes were made to achieve the performance aspects of a particular test run.

However, without a baseline run, each subsequent performance test has nothing to compare against. Modifications made at some point in the development process could have seriously hampered performance, rather than increasing it, and this would not be apparent without a baseline run.

Using Visual Studio to analyze the results

Visual Studio Team Test provides excellent tools to interactively analyze performance test results and investigate the large amount of metric data in tabular or visual form.

Firstly, we need to load in the results of a performance test run. If a test run has just been executed, then the results will be loaded immediately afterwards. However, if we need to load in a previous set of results we can do so using the following two main methods.

Using the Test Results management window

- Select the **Test > Windows > Test Results** menu option to activate the **Test Runs** window.
- Open the **Connect** drop-down and select a controller.
- Completed performance runs will be listed and can be selected to load in the performance test results.

Using the Open and Manage Test Results dialog

To use this option, a performance test solution needs to be open, and a load test must be loaded into the Visual Studio Editor.

- Click on the **Open and Manage Test Results** button to open the dialog window.
- Select a Controller to use, and then select a load test whose results you wish to load. A list of performance test results will be shown. Select a performance test result and click the **Open** button.

Note

*Occasionally, for whatever reason, Visual Studio may fail to list the results in the **Test Runs** window. This rarely happens, but it can sometimes be caused by the correlating result metadata files not being present or being corrupt. This means you cannot load the performance test results via the **Test Runs** window as they will not be shown. Using the **Open and Manage Test Results** dialog will allow you to get around this issue.*

Now that we have some performance test results to analyze, let's start off by looking at response times. Initially, Visual Studio presents four graphs of key results. **Key Indicators** is one of these graphs, and has an average response time metric listed in the table of results shown below the graph in Figure 3.9.

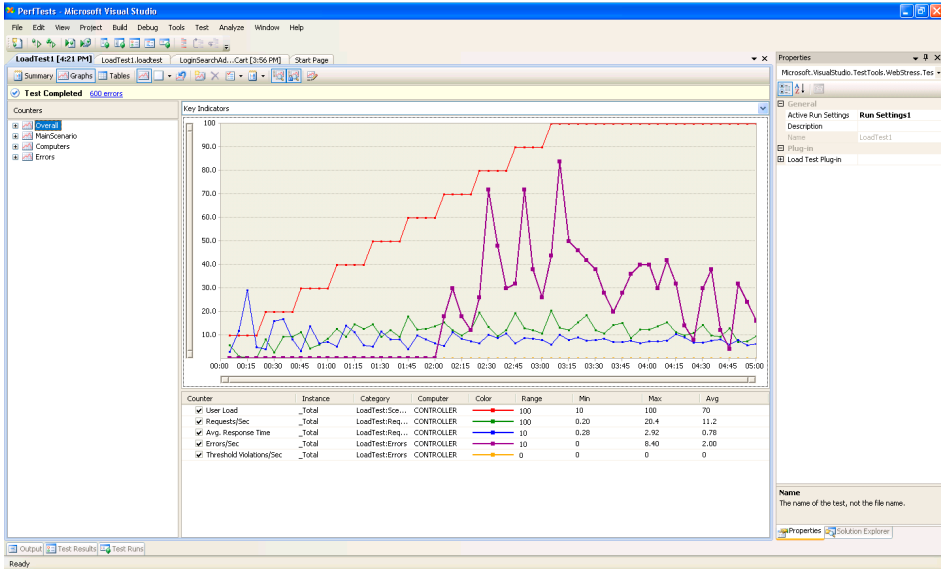


Figure 3.9: Key Indicators graph.

The average response time metric is only a general indication. Dependent requests can factor into this figure, in addition to the response time for low concurrent users, which skews the result to look more favorable than it should.

Filtering performance test result selection

In order to get a better average response time figure for a particular number of concurrent users (or, in fact, any metric data) we can use the dynamic filtering and selection feature of Visual Studio Team Test.

By using the timeline grab handles, it is possible to constrain the result set to a specified time window. For example, we may wish to see the average response time when the concurrent user count is between 50 and 80 concurrent users, To do this, drag the start and end timeline grab handles until the **Min User Load** column equals 50, and the **Max User Load** column equals 80. The grab handles are shown in red circles in Figure 3.10.

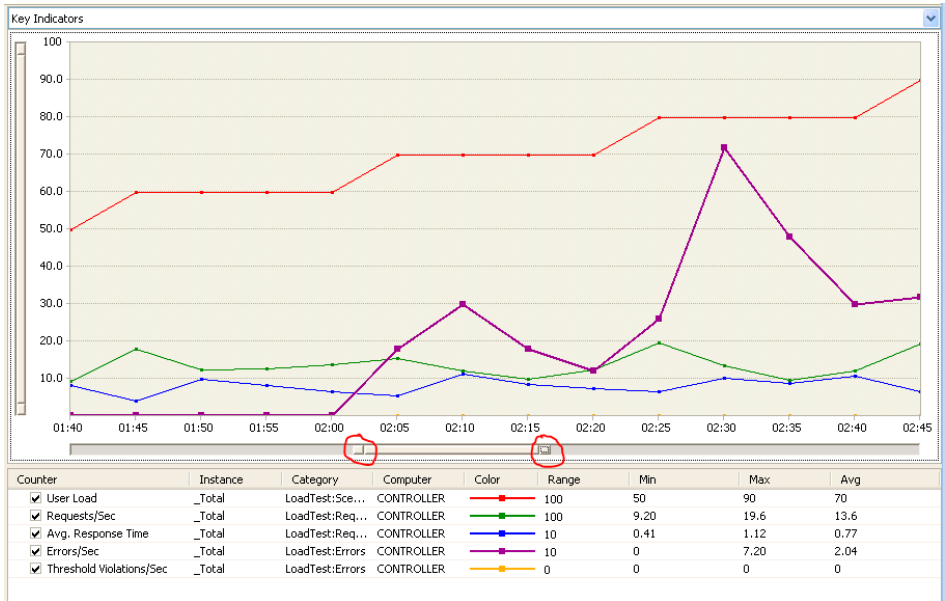


Figure 3.10: Timeline grab handles.

Note that the timeline grab handles can also be moved using the keyboard for finer adjustment, although the handle must be selected with the mouse first. There are also grab handles on the vertical axis to change the scale upon which the graph is shown, and these operate in exactly the same way as the timeline grab handles.

An alternative way of selecting a portion of the results is by selecting an area on the graph itself, although this is a little less accurate, and fine-grained control is not as easy. Click and drag the mouse to select an area. Once the area is selected, only that set of metrics will be displayed on the graph and in the tabular results below the graph.

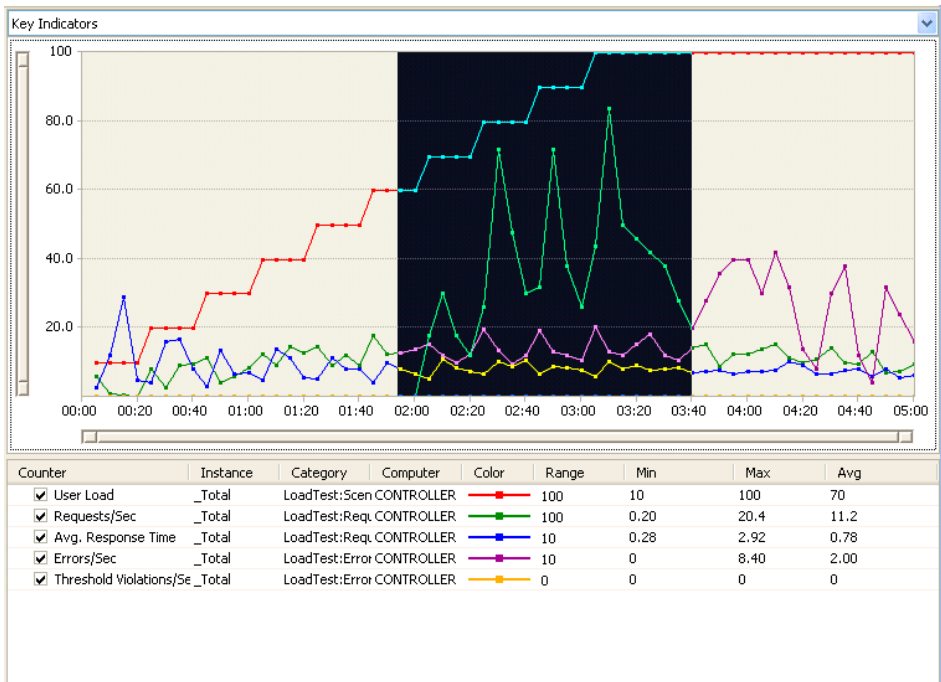


Figure 3.11: Selecting or filtering results via mouse selection on the graph.

This concept can be taken a little further. One of the first items to analyze in a performance test run is the response time at a certain concurrent user level. For example, let's say we would like to look at the response times when there are between 80 and 100 concurrent users. We need to ensure that the test run's user load is set to **Step Up** at periodic levels of the performance test run, which can be set in the properties for the load test scenario. The pattern must be set to either **Step Up** or **Goal Based** with user load increment values and time periods for the increments set to your desired values.

With a graph selection on and the key indicators graph selected, adjust the timeline grab handles so that the **User Load** has a minimum value of 80 and a maximum value of 100. Now select the drop-down box where **Key Indicators** is shown and select **Page Response Time** as shown in Figure 3.12.

The page response times will be shown, but will be constrained to the same time period and concurrent load that was selected while the **Key Indicators** graph was selected (see Figure 3.13).

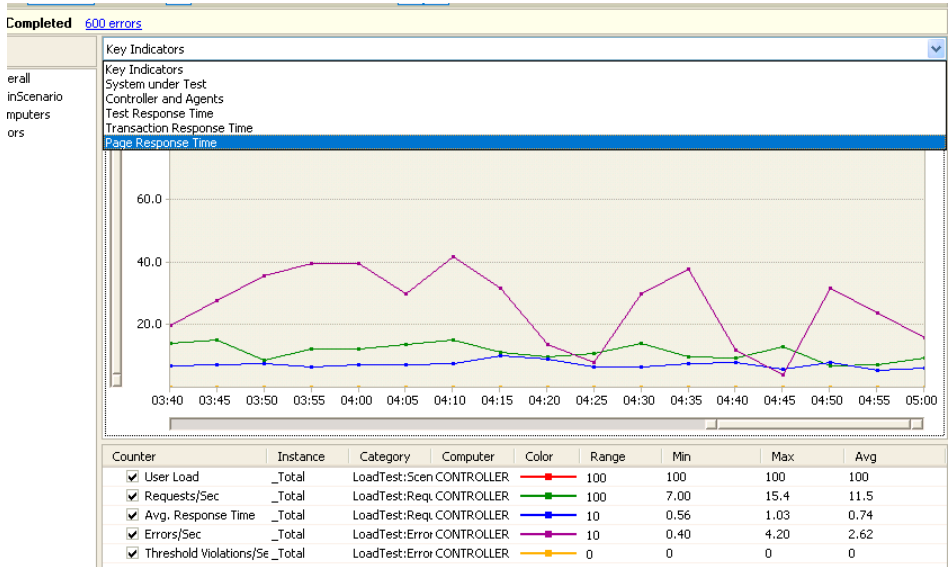


Figure 3.12: Selecting the Page Response Time graph.

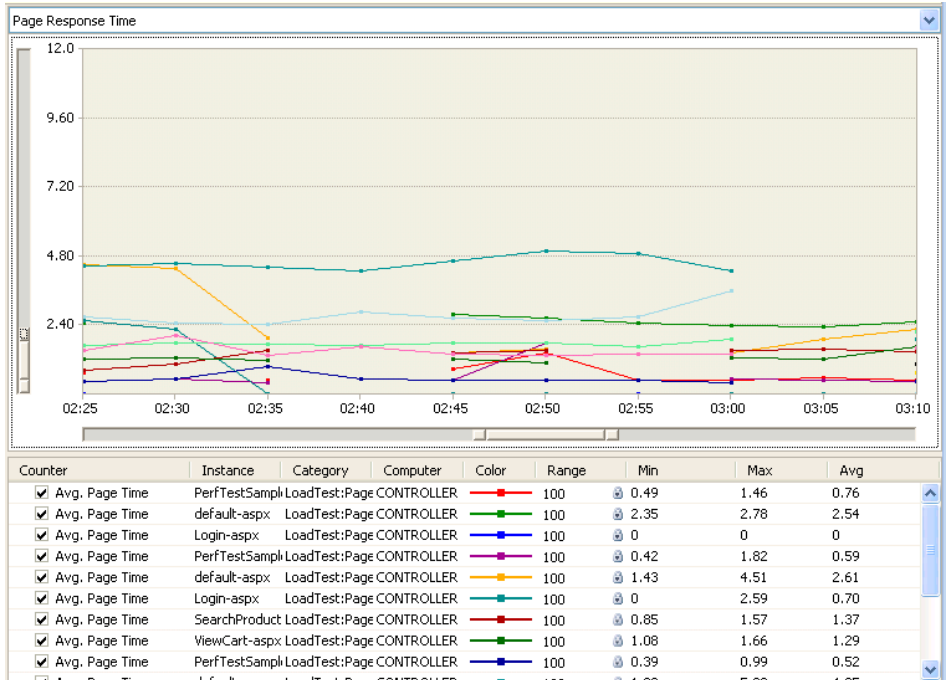


Figure 3.13: Page Response Times filtered by 80–100 concurrent users.

This is an extremely easy way to visualize each page's response time at a particular concurrent user level. If we take a step back and examine part of the business objectives of performance testing that were mentioned earlier, we can see that the goal was to validate that particular pages can respond in a set time (TTFB) at a particular concurrent user load.

Using this technique, it is easy to examine any page response time at any particular point in time or concurrent user level. If the response time for a page exceeds the agreed business requirements, then some performance modifications need to be made. However, even if the pages meet the required response times, it is still important to gauge what happens beyond the required concurrent user load.

Sweet spots and operational ceilings

Using the techniques discussed previously, it is easy to plot the response time characteristic of the application as concurrent user load is increased. In web applications, there is a typical pattern of response time progression as the concurrent user load is increased. This means that during the low concurrent user load stages of the performance test run, response time is excellent.

There comes a point where response time is still good and within acceptable bounds, but beyond this point response times start to increase sharply and keep increasing until timeouts begin to occur. The period before the sharp increase is what is referred to as the "optimal operating period" or "sweet spot." These terms refer to the time where the application can service the largest possible number of users without incurring a large or unacceptable response time. This is best shown in an example (see Figure 3.14).

Figure 3.14 shows a typical performance run. The optimal operating period, or sweet spot, was achieved relatively early in the run. The concurrent user load was relatively low at 200–300 concurrent users, with page response times of 3 seconds or less. This is the time when the application is performing at its most efficient and maintaining good response times. After this, the application still services requests, but sharp spikes in response time start to appear (represented by the blue line in the graph). Beyond this period, the response time continues to increase until erratic measurements occur. This is the operational ceiling, where the application begins to refuse requests and return **Service Unavailable**-type errors.

It is important to note that, while the business objectives have been exceeded before the application hits its operational ceiling, the behavior of the application can still be observed. This is important as it shows the resilience of the application and what could potentially happen if a sharp spike in load occurs that exceeds estimates.

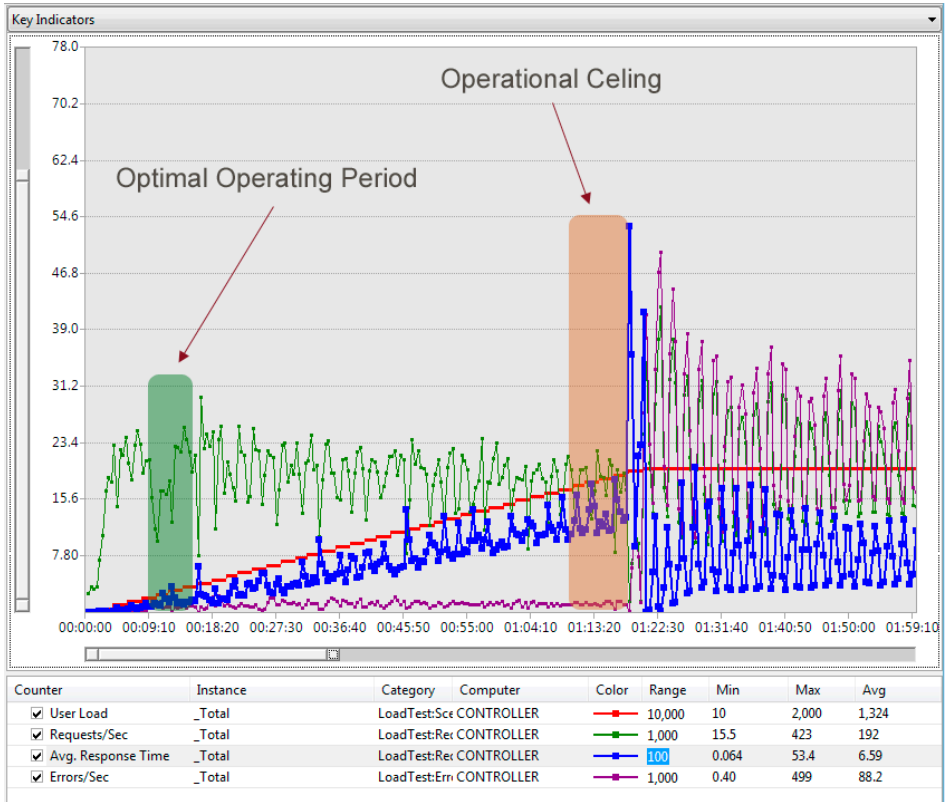


Figure 3.14: Sweet spot and operational ceiling.

In addition, during times of high stress, the functional aspects of the application will be exercised at maximum capacity and it will be easier to see what components of the application are performing more work than others. This will be relatively easy to see through the use of detailed performance metrics which are discussed in the next section.

While performance graphs and PerfMon metrics are extremely valuable in determining the sweet spot of an application, there is nothing quite like practical usage. In order to truly verify that the application is responsive and functional at the estimated sweet spot, it is best to actually perform a performance test run and simulate the number of concurrent users that is being applied at the estimated sweet spot. During this time, enlist the business users or stakeholders to use the application and report on its responsiveness. It will become quickly apparent whether the application is indeed responsive during the given load, and thus whether the optimal operating period is the one that has been estimated.

Detailed performance metrics

With an indication of system performance in hand (using the key metric values), as well as the ability to filter and isolate portions of a performance test run, it is now important to examine performance metrics in detail.

In particular, it is important to understand what the various metrics that are available mean in the context of a performance run. Whilst the key metrics discussed previously usually show whether there is a potential performance issue or not, they do not provide much insight into what the problem is.

This is where a detailed investigation and analysis needs to be performed on all available performance metric data to ascertain the nature of the performance issue. It is also important to understand what areas of the application can be generally improved. The immediate performance issue is an obvious choice, but many other areas may exist that could also be improved and contribute to an overall performance gain.

We'll start by first looking at the general performance metrics which can be relevant to almost all applications, whether on a server or desktop machine, and then the web application specific metrics will be discussed. It is important to note that not every single performance counter will be listed here as there are a huge number of them, details of which can be found in the Microsoft reference documentation. Here, we will be concentrating on the performance counters that aid in the detailed analysis of a test run – in other words, the counters that deliver real value in analyzing performance test run data.

Almost all metrics and counters are normally available within the PerfMon tool. The web specific metrics, such as response time, are only available via the Visual Studio Team Test tool and will be listed as such. It may seem obvious, but note that ASP.NET performance counters are typically only available on machines with the .NET runtime installed, such as web or application servers. Database servers would not typically have the runtime installed. Finally, rather than simply provide the detailed set of counters and their definitions, I'll also provide a discussion on typical analysis paths using these counters. Having the counters and their definition is not usually enough, as it is equally important to determine what counters are worth examining in certain situations, and what indicators to look for. This is where many people can become confused as, after determining that there is a problem, it is hard to choose what metrics and indicators use in the various categories of further investigation.

Performance metrics

For the following counter sets, it is generally recommended to monitor only the specific process in question. For web applications, this is typically the worker process (**aspnet_wp** in Windows XP and **W3WP** in Windows Server) that IIS uses to host them. For desktop and other applications, such as services, it will be necessary to monitor the specific application or host process. This is to minimize the performance counter variance that other, unrelated processes may introduce into the measurements.

General

Category: Processor

- **% Processor Time**

The main counter is the **% Processor Time**, which shows the total percentage of processor utilization across all processes. This provides a good general indication of system utilization and is the best starting point when looking at system performance. Visual Studio Team Test has predefined threshold levels for CPU utilization and will provide visual warnings when this counter goes beyond them.

Category: Process

- **% Processor Time**

The **% Processor Time** counter is exactly the same as the previously discussed processor category counter, but the processor utilization can be measured specific to a single process. For example, the **W3WP.exe** process (the web server host process) can be specifically measured for processor utilization in order to exclude any other process activity. This allows a good segregation of data and makes it possible to potentially pinpoint CPU intensive processes outside of the application itself.

- **Working Set**

The **Working Set** counter shows the amount of memory pages in use by all threads within the process, and is listed in bytes. This is a good way to examine the memory usage of a specific process.

Category: Memory

- **Available Mbytes**

This represents the amount of available physical memory in the system. Ongoing monitoring of this counter can show if an application has a memory leak. This condition can actually be mitigated somewhat in a web application using IIS health monitoring. This is where IIS will recycle or restart an application pool (and thus the processes running within it) when a memory limit has been reached. While this can alleviate the consumption of all available memory by the process, it is a defensive measure only, and the root cause of the memory leak should be investigated.
- **Pages/Sec**

This counter effectively represents the number of hard faults per second. A hard fault is when pages of memory are read from, or written to, disk; since disk operations are relatively slow compared to memory operations, hard faults are quite expensive in terms of system performance. The larger this counter, the worse the overall system performance will be. This counter should ideally be zero or at least very low. If it's high, this can indicate serious memory issues and that physical memory is either near limits or not used effectively.
- **Page Faults/Sec**

This counter should be used in conjunction with the previously mentioned **Pages/Sec** counter, and represents the number of hard and soft faults per second. A soft fault is where a page of memory was elsewhere in physical memory, and needed to be swapped into the process address space. Since memory operations are very fast, having a high number of soft faults is generally OK, as most systems can cope with this. Monitoring this counter can help provide the tipping point where hard faults begin to occur and where soft faults become excessive.

Category: .NET CLR Memory

- **Gen 0 heap size, Gen 1 heap size, Gen 2 heap size**
- **#Gen 0 Collections, #Gen 1 Collections, #Gen 2 Collections**

Both the heap size set of counters and the collection counters should show similar patterns of behavior. The .NET CLR garbage collector is a "mark and sweep" collection mechanism that partitions objects into different generations, Generation 0 (Gen0) being the shortest lived, most often collected and least expensive to collect. Generation 2 contains the longest-living objects, is collected the least often, and is the most expensive in terms of performance to collect. The #Gen 0, #Gen 1, and #Gen 2 collection counters represent the number of times each generation had a garbage collection performed, whereas the Gen 0 heap size, Gen 1 heap size, and Gen 2 heap size represent the memory heap size of each respective generation. While not an unbreakable rule, both sets of

counters should show approximately a 1:10 ratio between each generation. That is, the #Gen 0, #Gen 1, and #Gen 2 collections should follow a 100:10:1 pattern, and the heap size counters should show approximately a 1:10:100 pattern. This ratio of garbage collection statistics shows a healthy and normal memory usage by the application. Metrics that are largely different from this ratio can indicate erratic and inefficient memory usage or use of the garbage collector itself. Note that, for web applications, measuring only the W3WP process is preferable to looking at the total memory pattern and, for a desktop application, monitoring the application itself is preferable. For those of you uncomfortable with firm assertions, these ratios are supported by Microsoft performance documentation, mentioned by Rico Mariani (a Microsoft performance specialist), and are something I've often encountered myself. Whilst deviation from these ratios does not prove that there is an issue, it can often provide a strong indication.

Category: .NET CLR Exceptions

- **# of Exceps Thrown / sec**

This counter represents the number of exceptions being thrown per second by the application, and should be very low. Throwing exceptions is a relatively expensive operation and should be performed only in exceptional circumstances (i.e. actual, legitimate exceptions) not for control flow. Again, in web applications it is best to monitor only the W3WP process specific to IIS web hosting process. The exception to this rule is if a web application utilizes a lot of `Response.Redirect` calls because they generate a `thread aborted` exception. If this figure is high and there are a lot of `Response.Redirect` calls in the web application, then the figure may be representative of this, and it may be worthwhile trying to replace the calls with ones to the overload of `Response.Redirect`, which also takes a bool as the second parameter, and set that bool to false. This causes the request to not immediately terminate processing of the current page, (which is what causes the `thread aborted` exception).

Category: .NET CLR Jit

- **% Time in Jit**

This counter shows the percentage of elapsed time the CLR spent in a Just in Time (JIT) compilation phase. This figure should be relatively low, ideally below 20%. Figures above this level can indicate that perhaps some code is being emitted and dynamically compiled by the application. Once a code path is JIT compiled, it should not need to be compiled again. Using the `NGEN` command-line tool against your application assemblies to create a native, pre-JIT compiled image for the target platform can reduce this figure. Too much time spent in JIT compilation can cause CPU spikes and seriously hamper the overall system performance. Visual Studio Team Test provides a threshold warning when this counter has gone beyond a predefined acceptance level, which is 20% by default.

Category: .NET CLR Security

- **% Time in RT Checks**

This counter represents the percentage of time spent performing Code Access Security (CAS) checks. CAS checks are expensive from a performance perspective and cause the runtime to traverse the current stack to compare security context and code identity for evaluation purposes. Ideally, this should be very low, preferably zero. An excessive figure here, and by that I mean a figure exceeding 20%, can hamper system performance and cause excessive CPU utilization. This can often be caused by accessing resources across a network share or SAN where network credentials and security contexts need to be evaluated to gain access to the resource.

Category: .NET CLR Locks and Threads

- **Total # of Contentions**
- **Contention Rate / Sec**

These counters represent the number of unsuccessful managed-lock acquisitions, the **Total # of Contentions** being the total amount of unsuccessful lock acquisition attempts by threads managed by the CLR. The **Contention Rate / Sec** represents the same metric but expressed as a rate per second. Locks can be acquired in the CLR by using such constructs as the `lock` statement, `System.Monitor.Enter` statement, and the `MethodImplOptions.Synchronized` attribute. When a lock acquisition is attempted, this causes contention between the threads attempting to acquire the same lock, and blocks the thread until the lock is released. Unsuccessful locks can cause serious performance issues when the rate is high, as the threads are not only synchronized but ultimately unsuccessful, potentially throwing exceptions and waiting excessively. This rate should be very low, ideally zero.

Web / ASP.NET specific

Category: ASP.NET

- **Application Restarts**

This counter represents the number of times that the ASP.NET worker process has been restarted. Ideally, this should be zero. IIS has features to detect problems and restart worker processes, but this is a defensive measure for problem applications. Enabling these features for performance testing will detract from the value of collecting ongoing performance metrics for a test run. Ideally, the application should coexist with the infrastructure well enough to not require restarts.

The restarts of the worker process usually indicate that IIS has detected a memory condition, CPU condition, or unresponsive worker process, and forced the process to restart. The memory and CPU thresholds before IIS restarts a worker process can be configured within the IIS management tool. In addition, the amount of time to wait before a health check request is returned from the worker process can also be defined in the IIS management tool, although this is usually performed within the specific application pool that the application belongs to within IIS. The options for application pool health monitoring are shown in Figure 3.15.

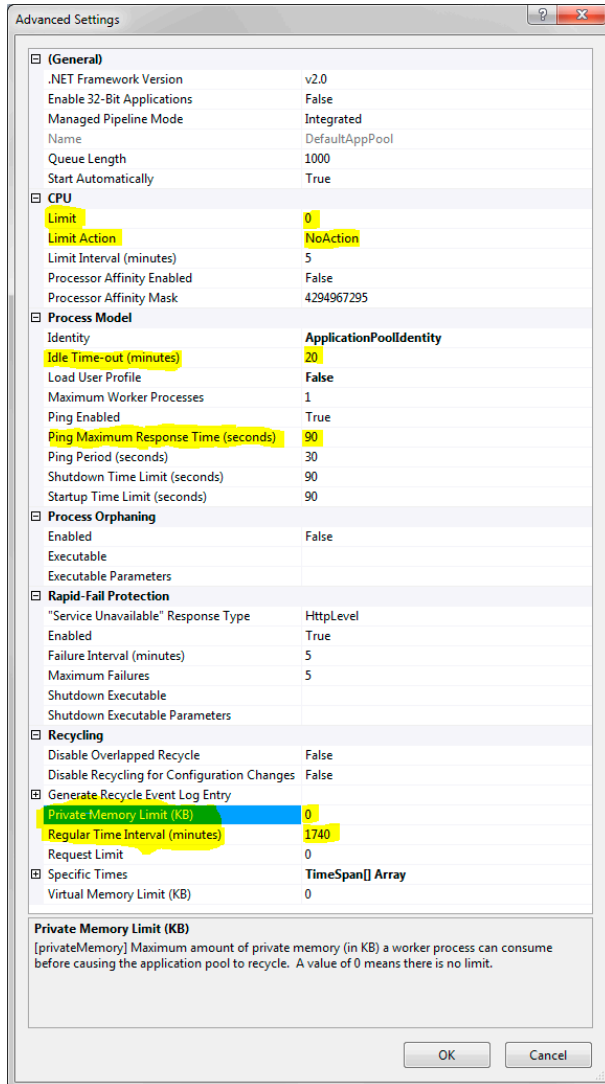


Figure 3.15: Options for application pool health monitoring.

Category: ASP.NET Applications

- **Pipeline Instance Count**

This counter represents the number of concurrent or active requests currently in the ASP.NET pipeline. Ideally (in case of very fast requests) there should be a very low number of concurrent requests, but this is not always possible. After reaching ASP.NET concurrency limits, requests begin to be queued. If requests are not executed quickly enough, more and more requests will be added to the queue until it becomes full and no more requests can be serviced. While this counter in itself does not indicate poorly performing requests in the pipeline, it can show a pattern in behavior at various load levels. In conjunction with the **Requests in Application Queue** counter (discussed next) this can indicate at what point the system experiences too much load to efficiently handle all requests.

- **Requests in Application Queue**

This counter represents the number of requests waiting to be added to the pipeline for processing. This counter should remain at 0 most of the time, otherwise the web server is not capable of processing requests as fast as possible. Occasional spikes are acceptable, but ongoing system usage with requests being added to the queue will eventually exhaust the web server's ability to process request in a timely fashion. Long response times will result, eventually resulting in timeouts or **Service Unavailable** type errors.

- **Request Execution Time**

This represents the number of milliseconds it took to execute the most recent request. The lower this figure, the faster ASP.NET is processing requests. This figure should be compared against a baseline figure when attempting to improve the performance of the application.

- **Requests/Second**

This is the number of requests executing concurrently per second, and is effectively the throughput of the application. This counter is closely tied with the **Request Execution Time** and **Pipeline Instance Count** counters. The higher this figure, the better, as it indicates that more requests can be serviced by the application. Visual Studio Team Test provides the requests-per-second figure in the metrics in the Key Indicators graph.

Database

It should be noted that, in a very general sense, if CPU utilization and memory utilization are within acceptable bounds, then a database server is able to perform optimally.

CPU utilization should ideally be as low as possible and not exceed 50–60%. An average of 15–20% is a good optimum operating value.

Memory utilization should also remain as low as possible, ideally averaging below 50%. An average of 30% is a good figure. Memory is key for a database server's fast performance, since a database engine will generally use as much memory as possible for caching execution plans and similar items to achieve high throughput.

While the above figures are gross generalizations, many performance issues will manifest on the database server as high CPU or Memory utilization. Obviously, this does not include more subtle problems such as deadlocks, transactions and disk I/O, which are covered later.

Remember, these counters are just good indicators. Further analysis using tracing and profiling tools may be required, and is covered in later chapters in this book along with typical scenarios in which performance issues can arise.

Category: Physical Disk

- **Avg. Disk Queue Length**

The physical disk subsystem on a database server is extremely important due to the I/O intensive operations that a database server performs. This counter represents the average number of read and write requests that have been queued and are yet to be fulfilled. As the number of simultaneous requests for data from the disk increases, the disk subsystem can become overloaded and unable to fulfill the requests as fast as required. The requests then become queued until the system can service the request. If the queue continues to grow, then the database server may be experiencing performance issues. Fewer requests, more efficient requests, and a faster disk subsystem can alleviate this issue.

Category: SQL Server: SQL Statistics

- **Batch Requests/Sec**

This is the amount of effective work the database server must perform, and can roughly equate to CPU utilization. This figure is dependent on the hardware specifications of the database server. However, 1,000+ requests per second can indicate potential issues and that the server may soon begin to experience stress.

Category: SQL Server: Databases

- **Transactions/Sec**

This counter simply represents the number of transactions the server is processing per second, and can be thought of as a submeasurement of the previously discussed **Batch Requests/Sec**. While not an accurate measure of the total work the server has to perform, this counter can provide an indication of how much relative transactional work is being performed when compared to the **Batch Requests/Sec** counter. Transactions are expensive from a performance perspective, and a high relative value may indicate a need to re-evaluate the isolation level and transaction policy of the application.

What do I do with all this information?

Performance testing is a very complex process, with the analysis and investigation of metric data extremely dependent on your application's specifics and the surrounding environment.

The large variances that can occur are the reason why specific guidelines around what to do in a given situation are very difficult to provide. In addition, it could be regarded as irresponsible or misleading to provide specific guidance and problem resolution to a general performance issue. This is not the goal here; my goal is to help you know where to start the process, which can sometimes be the catalyst to finding answers. This section will therefore attempt to provide some general guidance or clues to kick-start the analysis of performance issues when dealing with common problems. Once experience is gained in this process, individuals develop a general feel for the analysis process, and can begin the investigative process with great efficiency.

- Slow response times (TTFB) for a web application
 - Examine the request execution time. If the request execution time is long, with high CPU utilization, then look at optimizing the code itself. Profiling can provide insight here, and is discussed in the next few chapters.
 - If request execution time is long but CPU utilization is low, look at external systems such as database servers and/or web service calls. The system can be executing a database request or a web service and spending its time waiting for a response.
 - Examine the HTTP modules and handlers loaded for each request. Sometimes unnecessary handlers and/or modules can be configured for all requests, and will perform unnecessary processing as part of their default pipeline.

- High CPU utilization
 - This can occur for a large number of reasons and sometimes at very low load on the system. Areas to begin looking at can be:
 - CLR exceptions thrown/sec: lots of exceptions thrown can seriously hamper system performance and place extra load on the CPU.
 - % time in Jit: the Jit compilation phase can be computationally expensive. If the application is emitting any code or utilizing XML serialization assemblies, then this may be an issue. Optionally, test the code with all possible assemblies having native images generated via NGEN. Note that this counter may simply be a byproduct of the application and environment, and as such cannot be alleviated. If attempts at alleviating this figure prove unsuccessful early in the process, then it is generally best to concentrate on other aspects.
 - Consider utilizing caching where possible, regardless of the issue. Caching is one of the single most important performance optimizations for all applications. The most efficient database query is the one that doesn't occur or use the database at all. Techniques such as Output Caching for web applications, and caching within the application itself can help CPU utilization, response time, and database performance.

Granted, there are situations where it may not be able to be used (such as highly dynamic data) but that does not detract from its positive performance effects. Here caching refers to either browser based, proxy caching, output caching, application level caching, or even SQL caching. This is the reason that Microsoft can support millions of users through ASP.NET site with relatively little hardware, and also the reason communities like Facebook can accommodate 350 million users. It is also the reason why systems such as memcached and Project Velocity by MSFT are so high on the priority list. (More or less as an aside, Rico Mariani and Microsoft's official best practices also support this kind of behavior.)

- Aborted Requests
 - In a web application this can manifest as HTTP 500 errors, and as exceptions in a desktop or service application. This can be for any number of reasons but things to look at can be:
 - SQL Transactions and deadlocks: a deadlock can cause the victim query to be rejected and the request which instigated it to throw an error.
 - Downstream systems unable to handle the load: it is essential to have a good exception management policy in the application that will record external system activity and log all errors. Looking at request execution time and pipeline instance count metrics for web applications, and thread counts for service or desktop applications, can provide clues here. High values here can point to problems in this area.

- CLR Locks / Contention Rate/sec: this can indicate excessive locking in application code as threads of execution fight for resources, and often threads may abort after not acquiring those locks. At the very least, performance and throughput will be reduced.
- Exceptions in general: these should be caught and reported by the application; however, the exceptions/sec counter can provide clues if the figure is very high.

While this section has provided some clues as to what to begin investigating when performance issues are identified, there is simply nothing like deep knowledge of the application and infrastructure.

Often, developers or application architects will have a reasonable idea as to what might be causing the performance issues. Backing this up with metrics from various performance tests will enable quick and efficient identification of potential issues, and ultimately resolution.

Conclusion

This chapter has looked at a wide range of counter and metric data related to application performance and performance tests. Initially, a set of basic metrics and indicators were examined to provide quick and immediate insight into the performance of an application, These were:

- CPU utilization
- Memory utilization
- Response time / Time to First Byte (for web applications)

Web application and database specific counters were also addressed to cover more detailed, but also indicative, counters that will provide relatively quick insights into performance issues on both web and database servers.

While far from comprehensive, these counters can provide the "at-a-glance" view of your application's performance. Once a general idea of application performance is established, the process of investigation and analyzing performance results can occur, as shown using the excellent tools available within Visual Studio.

Using Visual Studio, it is possible to discern how an application performs over time, at various load levels, utilizing a broad set of performance metrics.

The detailed look at performance counters and metrics does not cover every performance counter and metric available, and yet shows the vast possibilities and variances that can affect an application's performance. This huge number of variables is what can take an enormous amount of time in the investigation and analysis of performance issues. The detailed view, trigger points, and potential courses of action that have been discussed in this chapter should significantly reduce that investigative time.

Now that we know what we're looking for, we can get a detailed view of performance testing and metric collection. After that (Chapter 6 onwards), we'll look at more isolated forms of performance testing, such as profiling.

Following from that will be practical advice on typical performance traps in applications, and how to overcome them. Integrating this process into the development process of software will complete the entire performance testing and analysis picture.

Chapter 4: Implementing Your Test Rig

Creating the performance test rig

So far, we have discussed the "why" and the "what" of performance testing. That is, why we do performance testing, and what metrics we can use to determine the performance of an application. This chapter will focus on the "how." Specifically, how is a performance test environment constructed so that we can record and perform performance tests?

Here, the architecture and construction of the performance rig will be discussed in detail, ranging from the test controller and test agents to the ideal network configuration to best support high volume performance testing. We will also cover performance metrics setup, collection and automation to ensure that the metric data will be collected reliably and automatically, with the minimum of effort. This data is the most valuable output of performance testing as, without it, we cannot make any assertions and must instead resort to guesswork. Finally, we will discuss ideal environments for application profiling, and the implications that must be considered when using load balancing, i.e. whether to test first in a load-balanced environment, or to begin performance testing in a single server scenario.

It is important to note that while profiling is an important part of determining the performance capability of an application, it is typically executed on a single workstation – more often than not, the developer's. Profiling will be discussed in greater detail later in this book but, for the most part, setting up for profiling is as simple as installing the necessary profiling tools on the workstation itself. However, the majority of *this* chapter will discuss the specifics of setting up a performance test rig.

Architecture and structure of a performance test rig

Being able to run high volume, effective performance tests requires more than a single workstation connected to a web server, simply executing multiple requests concurrently. When dealing with high loads, one workstation exercising a server is pretty soon going to run out of resources, whether memory, processor power, or network throughput. In addition, how are the user loads defined, what distribution of tests are run, how do we achieve high concurrent user loads for a sustained time, and how do we ensure that the network connection itself does not limit the amount of load generated against the server?

To help achieve these goals, a distributed performance test rig architecture is required. To that end, Visual Studio Team Test enables a remote workstation to connect to a dedicated controller machine. The controller manages test runs and coordinates the activities of one or more agents. The agents are actually responsible for running the tests and generating load against the desired server or servers, and they also collect data and communicate the test results back to the controller for storage and management.

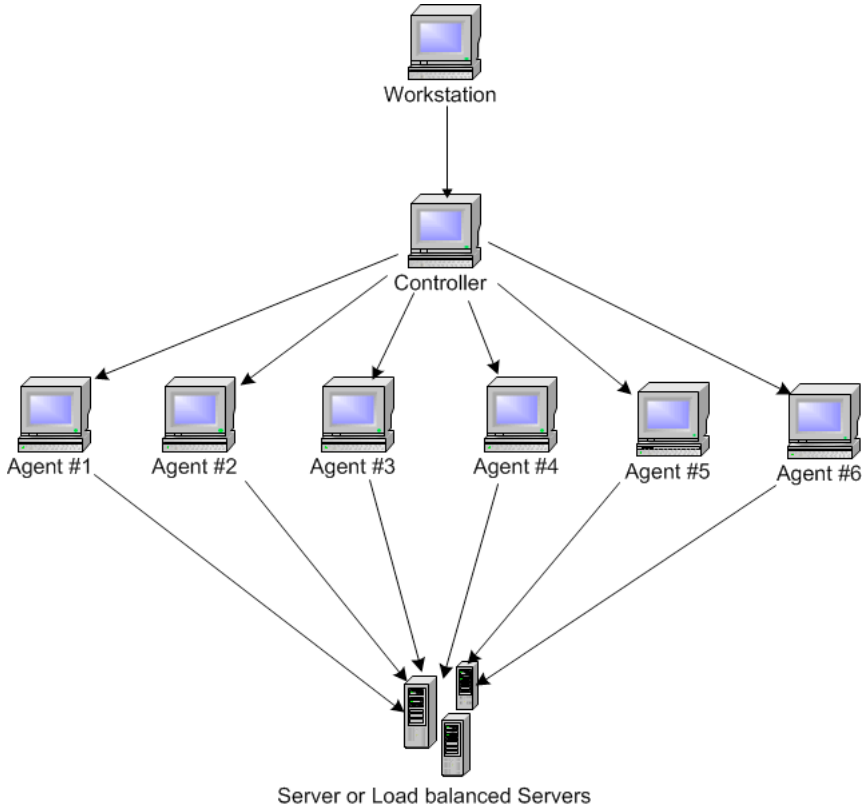


Figure 4.1: Performance test rig architecture.

Role breakdown

For now, I'll just give you a brief description of the role that each component plays in this architecture. A detailed discussion of installing, configuration, and management of each role within the system will follow later in this chapter.

Workstation

The workstation machine can be any machine with Visual Studio Team Test installed on it. Using Visual Studio Team Test, you can access a local or remote controller via the **Test** menu option, as shown in Figure 4.2.

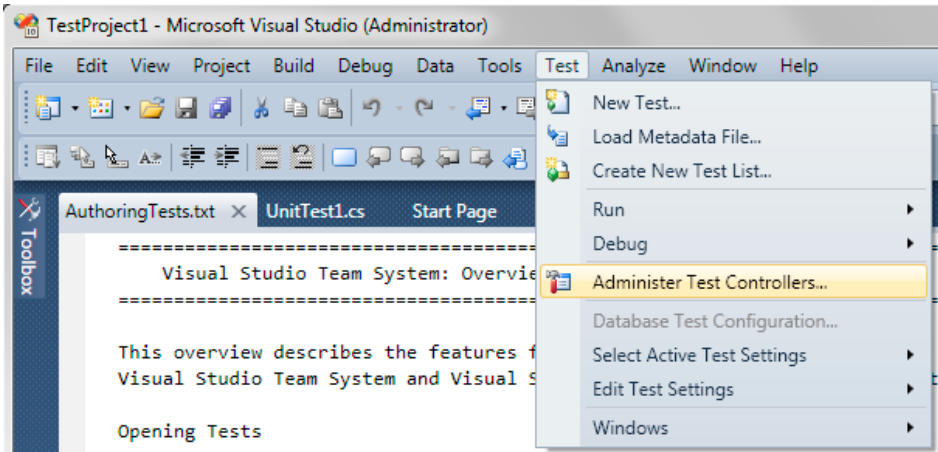


Figure 4.2: VSTS Test menu option.

Controller

The controller's role in the test rig is to coordinate the execution of the tests across multiple test agent machines, and manage the collection of results from all the test agents.

For larger scale systems, a workstation with Visual Studio Team Test can be used to connect to a separate machine which acts as the controller; however, the workstation can also act as the controller itself. Setting up the controller involves a simple software install which will be discussed later in this chapter.

Whichever machine acts as the controller, it *must* have access to a SQL database. By default, SQL Express is used to store and manage test results, but a traditional SQL Server database can also be used. SQL Express has a size limit of only 4 GB, so if you anticipate going over this limit, it is obviously best to use a full-sized SQL Server.

The physical specifications of the controller machine should include a minimum of a 1.4 GHz processor and 1 GB of memory. A 2 GHz or greater processor and 2 GB or greater of memory is relatively standard for today's workstations and is a recommended specification.

Test agent

Each test agent machine is responsible for executing the performance tests against the server, collecting the metric data for those tests, and then reporting those results back to the controller for storage and management.

When tests are scheduled for execution, the controller compiles them into assemblies and distributes these to the agents for execution. The controller manages the executing tests, ensuring that the appropriate number of concurrent users are simulated, as well as other factors and distribution details. Each agent, like the controller, requires a simple software installation, and the physical specifications of an agent machine should include a minimum of a 2 GHz processor and 1 GB of memory. Much like the controller, a 2 GHz or greater processor and 2 GB or greater of memory is relatively standard for today's workstations and is a recommended specification, although it's worth bearing in mind that memory is used heavily in agent machines, so the more the better.

Profiling system

As already mentioned, profiling an application is also an important part of assessing an application's performance and involves investigating said application at a finer-grained level than the broad approach of load testing. Because of this, profiling is the logical next step to load testing. However, it can be an intrusive operation, significantly affecting the performance of the application while it is being profiled. It can also, amongst other things, restart Internet Information Services in order to attach to profiling events to gain measurement data at a very low level.

Profiling can also be very memory- and processor-intensive, so the more memory and the better the processor, the better the profiling experience. Specific minimum requirements will depend on the profiling tool being used. For these various reasons, profiling is typically performed on a developer's workstation, as these are generally high-specification machines.

Setting up and configuration

Port setup and firewall considerations

In order to correctly install the controller and agent software on machines, certain criteria should be observed. It is important to remember that the controller and the agent are not normal user machines, and so should not contain all the security restrictions that regular

organizational workstations may have. Imposing such restrictions typically restricts the type of operations that can be performed, as well as such things as what ports are open for communication. This can seriously impact the ability of the controller and agent software to install or operate correctly.

This is not to say that the coexistence of the test rig and any security restrictions cannot be achieved, but rather that it simply requires more work. That being said, sometimes diagnosing errors in this coexisting system is not trivial and impedes the ability to even start *running* tests, let alone reliably execute them and receive results.

For these reasons, I recommend disabling firewall software on the controller and agent machines to ease setup and operational issues. Having no firewall restrictions means no possible port blockages at all, but it does also mean that these machines have no protection. This may not be an issue if they are on a separate network that is well-isolated from any public networks, public access or other potential security risks, but this is not always possible. Should you decide to keep a firewall active on these machines, and selectively enable the ports required to allow communication between workstations, controllers and agents, the following list shows the default ports and protocols that need to be allowed to ensure correct setup and operation.

- **Workstation used to connect to Controller**
 - File and printer sharing protocol
 - Port: 6901 (for test coordination)
- **Controller**
 - Port: 6901 (for test result collection)
- **Test Agent**
 - Port: 6910 (for test distribution)
 - Ports: 137, 138, 139 (for performance counter collection).

Note

In case you're wondering, these details come from digging deep into blog posts by Ed Glass (a VSTS team member who has great, detailed content) and then verifying them experimentally.

Network segmentation/isolation

To get meaningful results from performance tests, it is important to make the metrics that you record as clear and unambiguous as possible. In order to ensure metrics are valid and unskewed, all unknown quantities and variables need to be removed from the tests.

It is often hard to determine what traffic is traversing a network, and this can affect test results. While a network may seem responsive enough to perform load testing on, when someone decides to download gigabytes-worth of data across this network, congestion can occur. Because of this, an important factor when setting up a performance rig is ensuring a clean and direct path between the agents (which execute the tests) and the server (or servers) which are the target of the performance tests.

In order to conserve hardware costs, it's often tempting to provide a single machine with a lot of memory as a controller-cum-agent machine, and to connect to the server (or servers) being tested through the regular public network, or even the corporate intranet. The problems with this approach are outlined below.

- Network throughput of a single machine could be a limiting factor when generating extremely large loads. The amount of users being simulated might be 1,000 (for example), but the network interface may be saturated at the 500-user point, meaning that a true load is not being applied to the server.
- Latency, other traffic, and multiple network hops on the network path from the agent to the server may impede the speed at which data can be delivered to the server. Again, this may mean that the intended simulated load is not what is actually being delivered to the server. This may also mean that errors are generated in the tests which are not a direct effect of the load, and thus the results are colored. Latency and general traffic are a major impediment to the accurate application of load when you're attempting to generate it over a public Internet.

Note

Some organizations do offer performance testing services utilizing the general Internet and simulated browsers. They offer high load with attempts to mitigate the latency effect of the public Internet. The effectiveness of the tests themselves can only really be measured at a server level, and although the required load may simulated, this kind of testing is not as easily controlled, and a sustained high load cannot be easily guaranteed as the amount of "interference" on the Internet may vary. This does not mean that this type of testing is ineffective, but just that repeatable and sustained testing can be difficult. Whatever your decision, the recording and analyzing of metric data recorded during these kinds of tests is the same, whichever method is employed.

The ideal scenario in which to execute tests is to have a completely separate and isolated network, as this means that there is no network interference from the corporate infrastructure or the general Internet. The amount of traffic can be strictly controlled, and the load simulated by the agents has a direct route to the servers, and, thus, a direct effect. In short, no factors outside your control can affect the simulated load and skew the results.

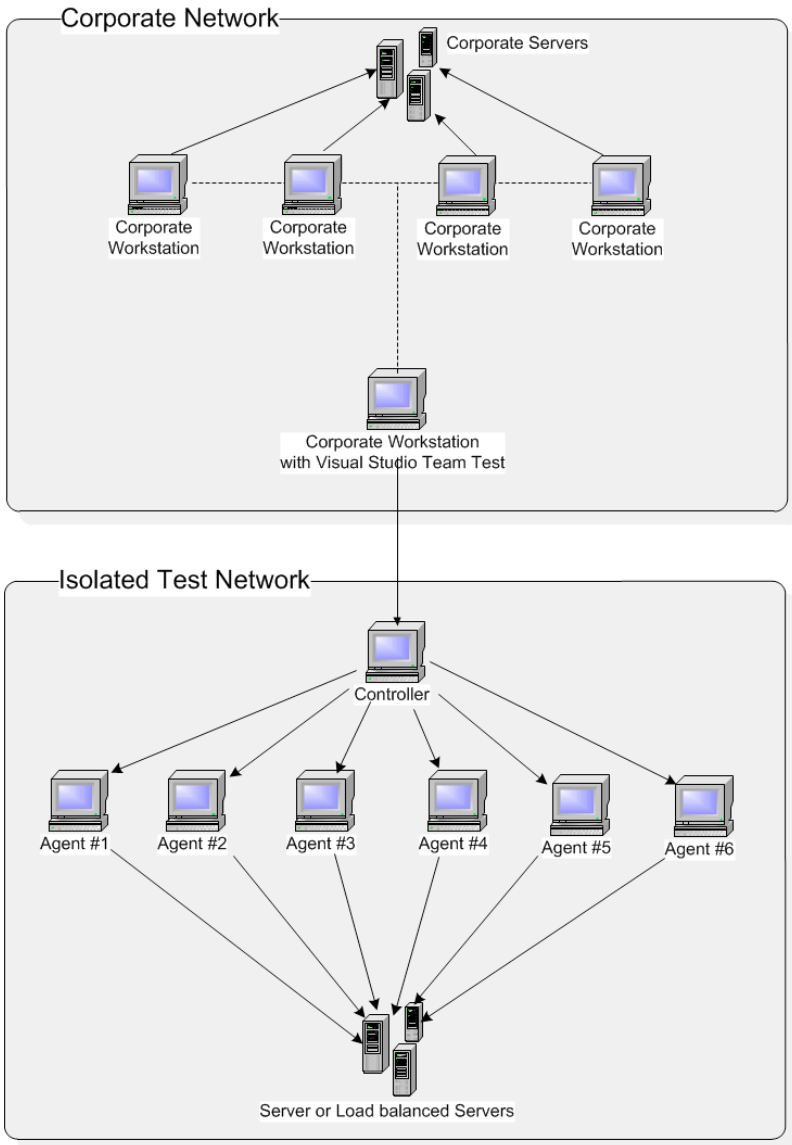


Figure 4.3: Isolated network test rig setup.

As shown in Figure 4.3, the workstation that exists in the corporate network has a direct connection to the controller machine, and so the controller is the only machine that has a path between the intranet/Internet/other network and the test network. The best way to achieve this is by using dual network interface cards (NIC); one with a direct route to the test network, and the other with a route to the intranet/Internet on which your workstation exists.

However, completely isolating a segment of your organization's network solely for performance testing is not always feasible, possible due to lack of time, money, or other resources. Remember that the goal is ultimately just to ensure a clean path from the test agents to your server or servers which are going to be tested, so that there can be no unknown elements introduced in load generation. Often, all the machines to be utilized as test agents are on the same network. Test agent machines are sometimes simply other users' workstations! To fit with this kind of infrastructure, it is common practice to install dual NICs in the machines that will act as test agents. Additionally, a simple network switch that supports the required number of ports for the test agents and server(s) can be used to create a separate network on which to run the performance tests. Figure 4.4 illustrates this:

In a dual NIC configuration as has been described, the default configuration of agents may not work. I'll discuss this issue in detail in the Agent Setup section later in this chapter.

Test agents and controllers can be installed on almost any machine. Some of those machines can be fellow co-workers' workstations, rarely-used machines acting as file servers, etc., although these will most probably not be on an isolated network. Generating extremely high loads can require many agents, so any spare machines may be enlisted to assist. If this is your situation, then you simply need to work with what you have. The effects of a mixed test environment can be mitigated by recording results directly on the server and ensuring that the requisite load is being applied, or at least measuring the difference between simulated load at the agents and actual load at the server. I touched upon this towards the end of Chapter 3 – it simply requires a little more analysis work.

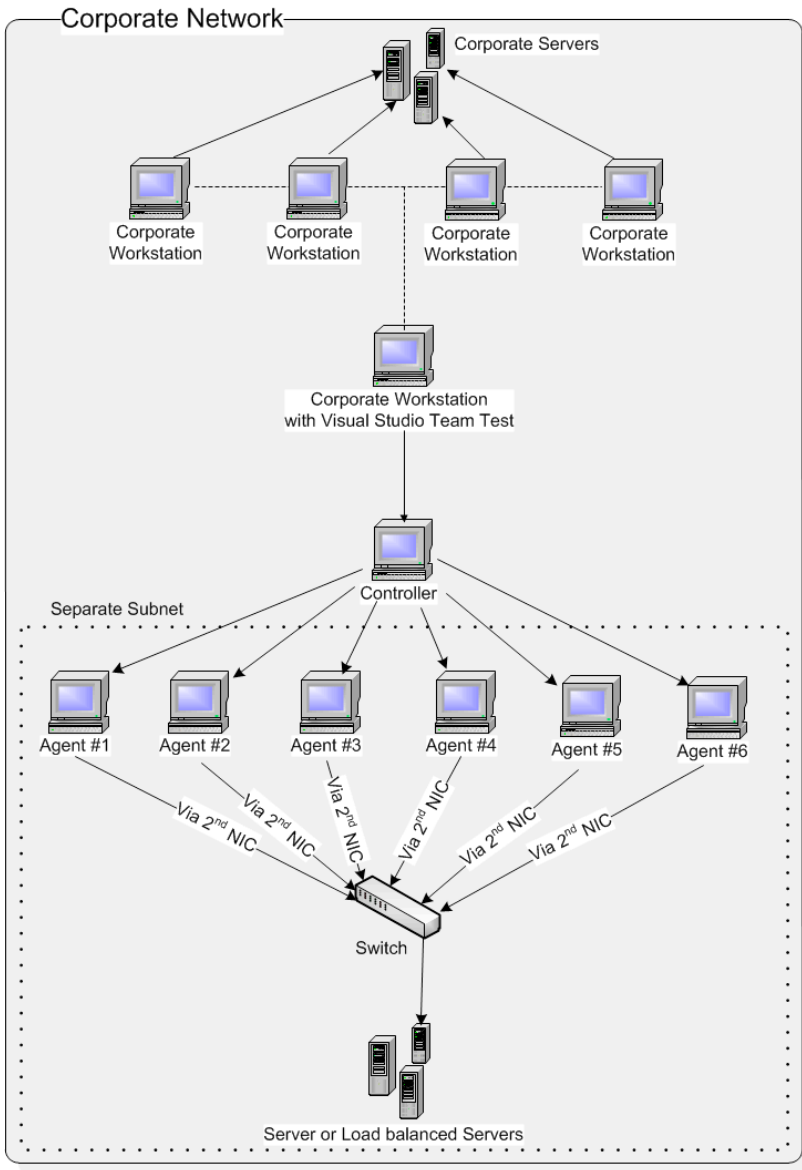


Figure 4.4: Typical isolated test segment, dual NIC setup for agents.

Controller setup

Both the controller and agent software are relatively easy to set up. It is important to install the controller software first, as the load agent software needs to be able to connect to the controller as part of the installation process.

Note

A trial version of the controller and load agent can be downloaded from [MICROSOFT'S WEBSITE](#). The trial version can be used for 90 days from the installation date or 25 test executions for the Load Agent.

Starting the controller installation process is a matter of executing the **setup.exe** application located on the install media. As is normal for Microsoft installations, you'll need to click **Next** through a series of screens, including a **User Experience Improvement Program** opt-in, the license agreement and specifying the destination folder for installation.

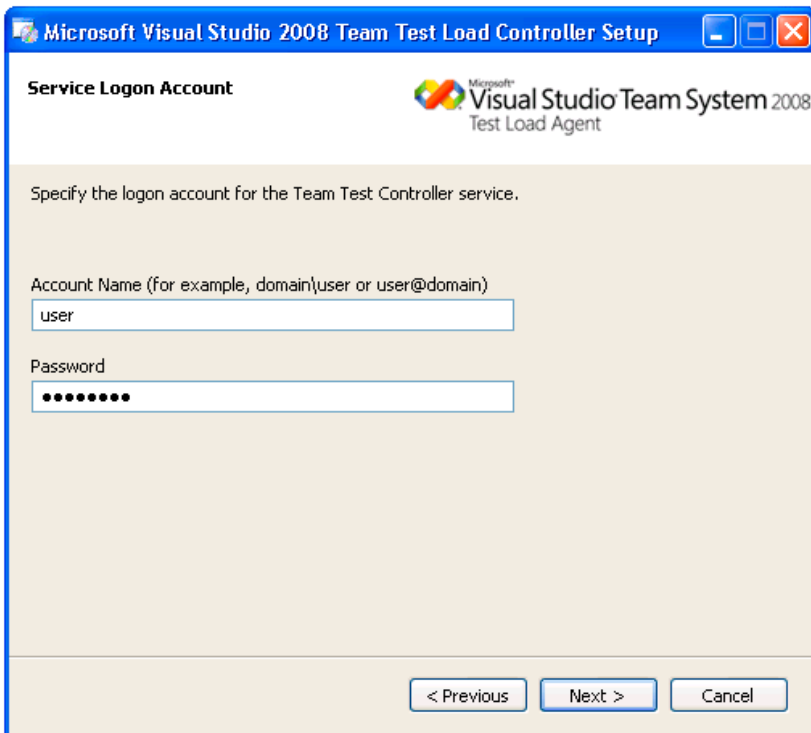


Figure 4.5: Controller installation – controller service user.

The final step prior to actually beginning the installation requires the specification of a user account which will be used to run the controller service. This can be a local user account or a domain account. It is best to ensure that this account does not have regular password expiration periods. While this is not best practice with respect to security, and most domain accounts would have this enabled, it *does* prevent having to re-enter the user's credentials each time the password expires.

This user must also have sufficient access privileges to be able to read performance counters from the computers under test – typically the server(s) being load tested – as that is the entire purpose of the controller.

A controller can also be running in workgroup mode, meaning that a non-domain-user is used for the controller and subsequent agents. If running in workgroup mode, there must be a local computer account on the controller which also exists on all the agents. When the agents are set up, this common user is specified as the agent service account, and so can connect to the controller successfully. However, for the security negotiation to occur successfully, the accounts must have the same username and password.

Once you've moved past all these dialogs, the actual installation should only take five to ten minutes.

Further notes on controller installation

In most typical client engagements I have been involved in, the workstation and the controller have been the same machine. This is generally due to the cost of an extra controller-specific machine and, most importantly, the extra effort it takes to connect to the controller from a workstation machine. The tangle of issues surrounding user access privileges, or matching up users on the controller and agents, means that making the controller and workstation the same machine is a common scenario. This setup is made more attractive by the fact that the controller only coordinates tests and does not need excessive resources to run.

Once the controller setup is complete, there are three new groups created on the controller machine. These are:

- TeamTestControllerAdmins
- TeamTestControllerUsers
- TeamTestAgentService.

If a workstation machine needs to connect to the controller, the user context being used must be a member of the TeamTestControllerUsers group.

Note

Any workstation, controller or agents that are participating in the test rig must all be on the same version of Visual Studio, right down to the Service Packs.

After the controller is installed, access to it is limited to members of the TeamTestControllerUsers and TeamTestControllerAdmins groups that were created during setup, and to the Administrators group. Add appropriate users and/or groups to one of these groups to allow them to access the controller. Members of the TeamTestControllerAdmins group or the Administrators group can administer the controller by clicking the Test menu in Visual Studio, and then choosing **Administer Test Controller**. Bear in mind that members of the TeamTestControllerAdmins group must also be power users or administrators on the controller computer.

In order for agents to connect to the controller, they must be a member of the TeamTestControllerUsers group at the very least. Normally the user is added to this group during the agent installation process. However, there may be instances where you change users on the agent manually, thus you need to ensure this alternate user is also in the appropriate group on the controller.

Creating the load test database

When the controller is installed, a database is created to hold all the performance metrics and recorded results. Wherever this database is located, be it on another database server, or on some other instance other than the default, it *must* have the correct schema. To that end, when the controller software is installed, a SQL script file is also installed which can recreate the load test database with the correct schema and everything required to hold the performance test results. By default, this script is located at: **C:\Program Files (x86)\Microsoft Visual Studio 9.0 Team Test Load Agent\LoadTest\loadtestresultsrepository.sql**

By executing this file against a database, typically using a tool such as SQL Management Studio, a new database called **LoadTest** is created and is ready to be used as the repository for performance test results.

Guest policy on Windows XP in workgroup mode

Finally, if the controller software has been installed on a Windows XP machine in a workgroup environment, then Windows XP will have the **ForceGuest** policy setting enabled by default. This means that any time a remote user wishes to connect to this machine, it will only be allowed to connect as the **Guest** user. So, no matter which user the agent is configured to use when connecting to this controller, it will be forced to connect as the **Guest** user, which has very minimal security privileges.

The fix for this is not entirely straightforward, but not very difficult either. To disable the **ForceGuest** policy in Windows XP:

- Run the **Registry Editor** (open the **Run** dialog, type `RegEdit` and press **Enter**).
- Navigate to the key: `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Lsa`.
- Double-click on the **ForceGuest** item and change the value in the presented dialog from **1** to **0**.
- Click **OK**, and you're done.

Note

This process should be performed on the controller as well as all agents, otherwise you may find your agents listed in the controller agent list, but they remain disconnected or offline. Any error mentioning that the server rejected the client's credentials is indicative of this problem.

Agent setup

Starting the agent installation process is also just a matter of executing the **setup.exe** application located on the install media. You will be presented with the same screen as shown in the controller setup, albeit with some different text. As with the controller setup, you will just need to move through the various screens until you reach the point at which you need to specify a user account under which to run the **Test Agent Service** process.

This is where it is important to have defined an appropriate test agent user on the controller machine so that, when the installation process executes, the test agent machine can connect successfully with the controller.

The next step in the installation is different from the controller setup, in that the user is asked which controller to connect to. Completing this process adds the user to the appropriate groups on the controller (if required) as well as setting up the user on the local agent machine.

Once the controller is specified, the installation process proceeds in exactly the same fashion as the controller setup.

Workstation setup

The workstation that is used to connect to the controller can, in fact, be the controller machine itself. Having separate machines for the controller and the connecting workstation is preferable so that when the controller is busy coordinating tests and collecting data, the responsiveness and performance of the workstation remains unaffected.

Whether the controller is on the same machine or a separate machine, to connect to it from within Visual Studio, simply select the **Test** menu option, and then select the + menu option.

This will present the dialog in Figure 4.6.

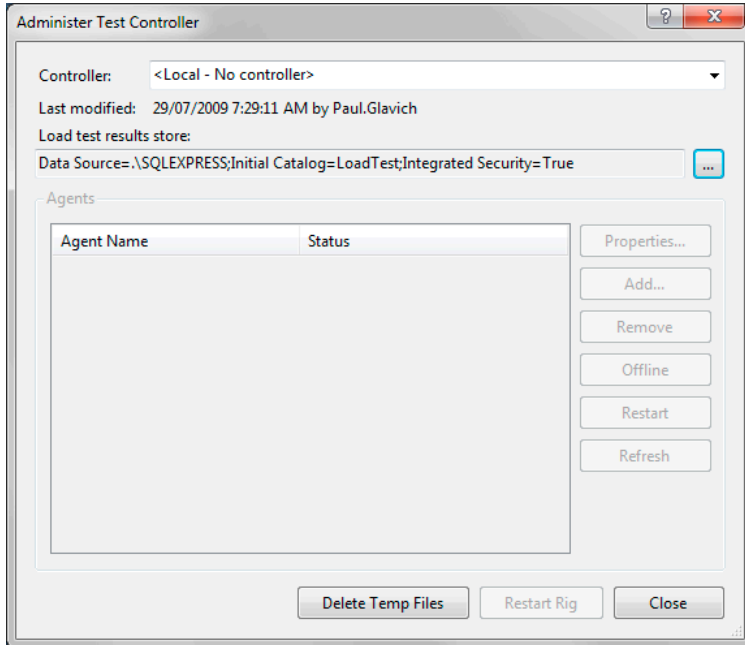


Figure 4.6: Administer Test Controller dialog.

As you can see in Figure 4.6, the default controller is listed as <local – No controller>. The local machine is the default controller, but the controller software has not been installed. Entering the correct machine name or IP address in this text field will connect to the controller and list any test agents registered with that controller. The dialog should then update to list the number of agents installed and connected to the controller, and their current status.

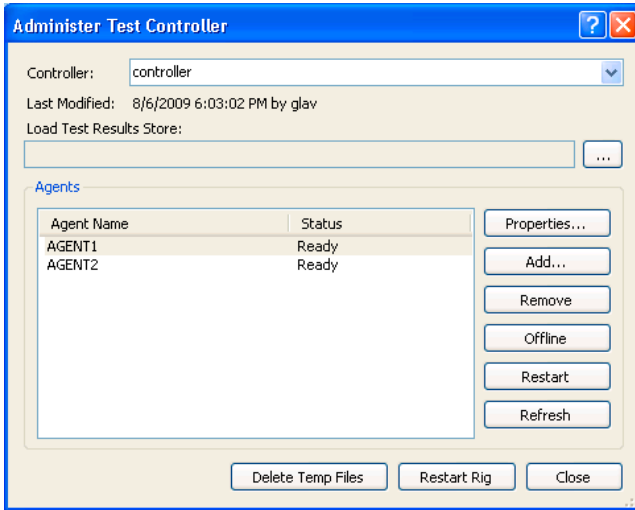


Figure 4.7: Administer Test Controller showing multiple agents connected.

Note that if you have installed the controller software on your workstation, then invoking the dialog shown in Figure 4.7 will automatically connect to the local controller and display any agents already connected.

Troubleshooting the controller and agents

Getting all the components in a test rig to talk to each other nicely is often not an easy task, and things will sometimes not work for seemingly non-existent reasons.

When both the controller and agents are installed, there are certain settings within each configuration that can help in diagnosing issues. Since the issues can be numerous and very environment-specific it would be impossible to list them all; However, I can provide some knowledge on the techniques to find out what the issues are.

Both the controller and agents utilize settings from their respective configuration files, which reside in the same directory as the controller and load agent respectively.

By default, the **Load Test Agent** configuration file is located in a directory such as:

For VSTS 2005: <Program Files>\Microsoft Visual Studio 2005 Team Test Load Agent\
LoadTest\QTAgentService.exe.config

For VSTS 2008: <Program Files>\Microsoft Visual Studio 9.0 Team Test Load Agent\
LoadTest\QTAgentService.exe.config

The `QTAgentService.exe.config` file contains the configuration of the agent.

Similarly, for the controller, the default location of the configuration file is:

For VSTS 2005: <Program Files>\Microsoft Visual Studio 2005 Team Test Load Agent\
LoadTest\QTController.exe.config

For VSTS 2008: <Program Files>\Microsoft Visual Studio 9.0 Team Test Load Agent\
LoadTest\QTController.exe.config

And the `QTController.exe.config` file contains the configuration of the controller.

The controller and agent configuration files are almost identical in their settings. Both contain `appSettings` and `system.diagnostics` sections that define the specific settings, and on default installation look similar to this:

```
<system.diagnostics>
  <switches>
    <!-- You must use integral values for "value".
         Use 0 for off, 1 for error, 2 for warn, 3 for info,
and 4 for verbose. -->
    <add name="EqTraceLevel" value="3" />
  </switches>
</system.diagnostics> <appSettings>
  <add key="LogSizeLimitInMega" value="20"/>
  <add key="AgentConnectionTimeoutInSeconds" value="120"/>
  <add key="AgentSyncTimeoutInSeconds" value="300"/>
  <add key="ControllerServicePort" value="6901"/>
  <add key="ControllerUsersGroup" value="TeamTestControllerUse
rs"/>
  <add key="ControllerAdminsGroup" value="TeamTestControllerAdm
ins"/>
  <add key="CreateTraceListener" value="no"/>
</appSettings>
```

In order to enable trace logging for either the controller or the agent, change the following settings in the configuration file:

- In the `appSettings` section, set the `CreateTraceListener` value to `yes`.
- In the `system.diagnostics` section, set the `EqtTraceLevel` to `4`.

The respective services will need to be restarted before any changes will take effect. A log file will then be produced in the same directory as the configuration file, and will be named `VSTTAgent.log` for the test agent and `VSTTController.log` for the controller.

With logging enabled, exhaustive detail will be written to the log file, providing very fine-grained insight into what is happening and why a problem may be occurring. By default, the `LogSizeLimitInMegs` setting limits the log file size to 20 megabytes, which is sufficient for most purposes. If this limit is reached, a new log file will be created and named in numerical sequence. Setting the `LogSizeLimitInMegs` value to 0 allows unbounded log file size, limited only by the available disk space.

Note

*A tool called **DebugView** can be used to show logged data without actually writing to a file, or having to monitor and refresh the log file's content. The tool is a free download from [HTTP://TINYURL.COM/MSDEBUGVIEW](http://tinyurl.com/MSDEBUGVIEW). By simply setting the `EqtTraceLevel` to `4` in the `appSettings` section in the configuration file, log information will be captured by **DebugView** and displayed immediately in a separate window.*

As already discussed earlier, often either controllers or test agents will be set up in a dual NIC configuration (dual Network Interface cards). In this instance, the controller/agent needs to know which network card to use to communicate with the rest of the test rig. In order to specify this, you can add the `BindTo` configuration value to the configuration file, and specify the IP address of the network card to use. For example, adding the following line to the `appSettings` section of a configuration file will tell the controller/agent to communicate on the network interface card with the IP address, `192.168.1.10`:

```
<add key="BindTo" value="192.168.1.10" />
```

If this value is omitted, then this could quickly result in problems with the agent connecting to the controller or vice versa.

When dealing with issues specific to the test agents, you can also use the provided command-line tool to configure various aspects of the agents. The command-line tool is named `AgentConfigUtil.exe`, and exists in the same directory as the agent executable and configuration file mentioned earlier. To use this tool, open a command prompt and navigate to the load agent installation directory (by default this is `C:\Program Files\Microsoft Visual Studio 9.0 Team Test Load Agent\LoadTest`) Type `AgentConfigUtil` and press `Enter`, and you will see a display of available commands.


```

C:\Program Files\Microsoft Visual Studio 9.0 Team Test Load
Agent\LoadTest>AgentConfigUtil.exe

Microsoft (R) Visual Studio Test Rig Command Line Tool Version
9.0.21022.8
Copyright (c) Microsoft Corporation. All rights reserved.

Usage:                               AgentConfigUtil [options]
Description:                           Used to perform test rig
                                        configuration operations.

Options:
/help                                  Displays this usage
                                        message
                                        (short form: /? or /h).

/nologo                                Do not display the startup
                                        banner and copyright
                                        message.

/nolog                                  Do not create setup log.

/unregister                             Removes the agent's
                                        registration from the
                                        specified controller.

/controller:controllername[:port]      Displays the test
                                        controller name and port
                                        number [optional].

/controllerusername:[domain\]username  Domain and user name
                                        for connecting to the
                                        controller.

/controlleruserpassword:password       Password for connecting to
                                        the controller.

/agentserviceusername:[domain\]username Domain and user name
                                        for the AgentServiceUser
                                        account.

/agentserviceuserpassword:password     Password for the
                                        AgentServiceUser account.

```

For example, using this tool, you can instruct the agent to re-register with the controller, specifying the controller machine name, port number, username and password to use. Alternatively, you can unregister an agent from the controller. For example, to remove an agent from a controller called **TestController**, you would enter this command:

```
AgentConfigUtil /controller:TestController /unregister
```

Alternatively, to add an agent to a controller named **TestController** using port 6901, the command would be:

```
AgentConfigUtil /controller:TestController:6901
```

Setting up performance counter collection

When Visual Studio Team Test executes a load test, there are a number of metrics that are collected by default from test agents and the server(s) being tested. These are usually sufficient for most general purposes. What counters to record and their respective meaning was covered in Chapter 3. For now, we know that we can collect performance data from the agents and servers, and have those metrics recorded within the database that the controller is configured to use as a repository.

However, this only provides one location where data can be recorded. It is also important to record performance data directly onto the servers being tested where possible.

Before we discuss *how* to do this, let's discuss *why* we should. There are a few important reasons why you would also want to record performance metrics on each server being tested, even though this may seem somewhat redundant. The reasons include those below.

- In a few rare circumstances, performance data is not recorded to the controller's data store, possibly because of system problems on the controller, disk space, etc. While in some circumstances, the data is recoverable (this will be shown later), often it is not. Not being able to get the recorded data is the same as not running the test at all. As previously mentioned, performance testing is a relatively expensive operation, and having data recorded on each server ensures you have an alternate copy of this pricy data.
- If there are multiple servers being used in the load test, you can determine if certain servers are experiencing more stress than others. This could be for a variety of reasons, including load balancing configuration, and system specification. Either way, ensuring that load is evenly distributed is important. If one server has to handle substantially more load than others, then the ability of the entire system to handle the load will be determined by this particular server. Additionally, being able to measure the

performance on individual servers means that tuning the configuration of a load balancer and observing the effects becomes a lot easier.

- Occasionally, a test agent may not send performance data to the controller for recording. When a test agent is under stress (due to lack of memory or processor capacity for example), its data may not be able to be collected by the controller. This may appear as gaps in the visual graph that Visual Studio presents for visualizing the performance data. To be able to verify that load was still being generated during this period, or to validate other metrics not apparent in the Visual Studio visualization, the secondary performance data recorded on the servers can be used.
- Many individuals or teams may wish to analyze the performance data. This data may need to be sent to external parties for analysis. Other interested parties may not have access to the visualizations and analytical facilities provided by Visual Studio Team Test. Recording data at the server level, using commonly available tools ensures that performance data can be viewed and analyzed by anyone who requires it.

It is not strictly necessary to record performance data at the server in addition to using Visual Studio Team Test, but the cost of doing so is quite low. Given that performance testing is an expensive process, it is a worthwhile investment to be able to record the performance metrics on the server(s) as an alternate location for data storage.

One important component that needs to be looked at more closely in load testing is the database. Using Perfmon to record performance data on the database is extremely important, as the database plays such a crucial role in the majority of applications today. Having a set of recorded performance data on the database machine itself will allow individuals such as dedicated database administrators to examine said data and provide valuable insights into the performance of the database. Even if no other data is recorded via Perfmon on the web or application servers, then it is recommended that the database have Perfmon recording SQL-specific performance metrics (along with standard counters such as CPU utilization).

You can set up recording performance metrics on the server(s) themselves using a tool called *Performance Monitor* which is available on all versions of Windows from XP to Server 2008. Performance Monitor will allow you to specify and record WMI counters, either to a file or to the database.

Note

Visual Studio uses a mechanism called WMI – Windows Management Instrumentation Counters to query and collect data.

To use this tool, select the **Start** menu, go to **Administrative Tools**, and select **Performance Monitor**. Alternatively, open the **Run** dialog and type *PerfMon*. The user interface looks a little different on Vista / Windows 7 / Server 2008 from how it does on older operating systems, but the functionality is very similar. You will be presented with a screen similar to that shown in Figure 4.8.

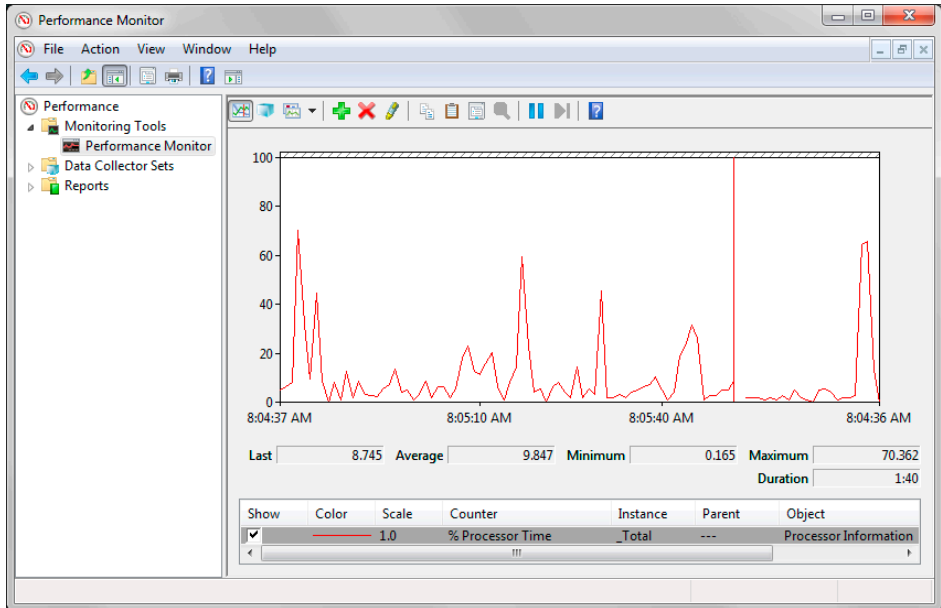


Figure 4.8: Performance Monitor on Vista / Windows 7 / Server 2008.

This initial view presents a real-time view of the currently selected performance metrics. By default, these metrics are **% of processor time**, **available memory** and **average disk queue length**. There are a huge number of performance counters that are available to monitor, and many products add extra counters (specific to their respective technology) to the list when they are installed. SQL Server or Windows Communication Foundation are examples of such products.

Adding counters to the monitoring instance is a simple process. Clicking the **Add** icon will display a dialog of counter categories and their associated counters that can be added from there.

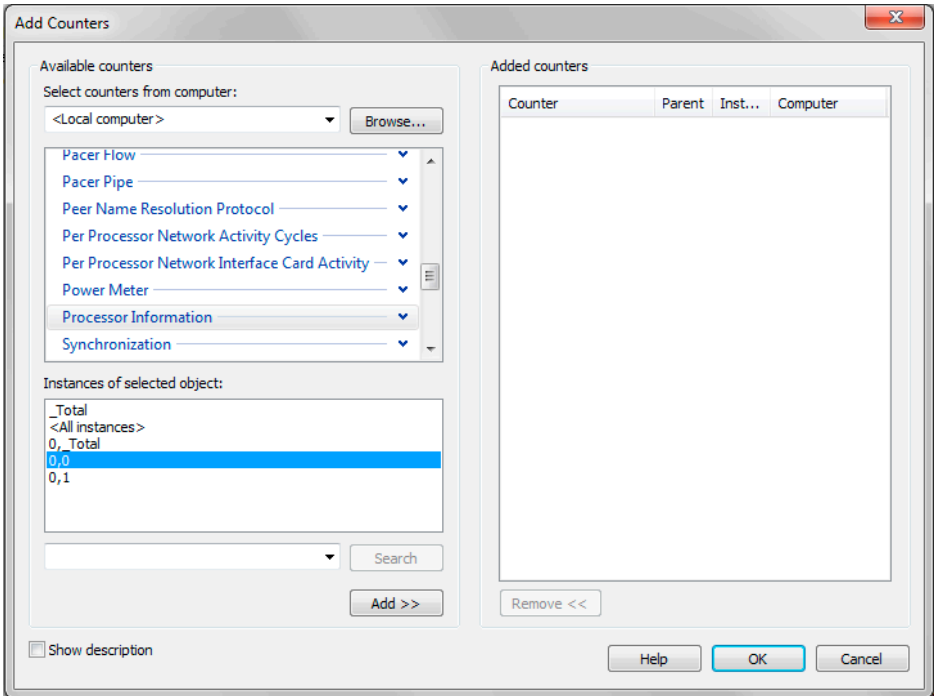


Figure 4.9: Add Counters dialog on Vista / Windows 7 / Server 2008.

Selecting a counter category will show the individual counters that can be selected and added. Multiple individual counters can be selected and added and, if you like, an entire category can be selected and added, with all the counters in that category added to the display.

You can obtain a brief description about each counter before adding the counter by selecting the **Show Description** option (**Explain** in Windows XP/2000/2003). The dialog should look similar to the one in Figure 4.10.

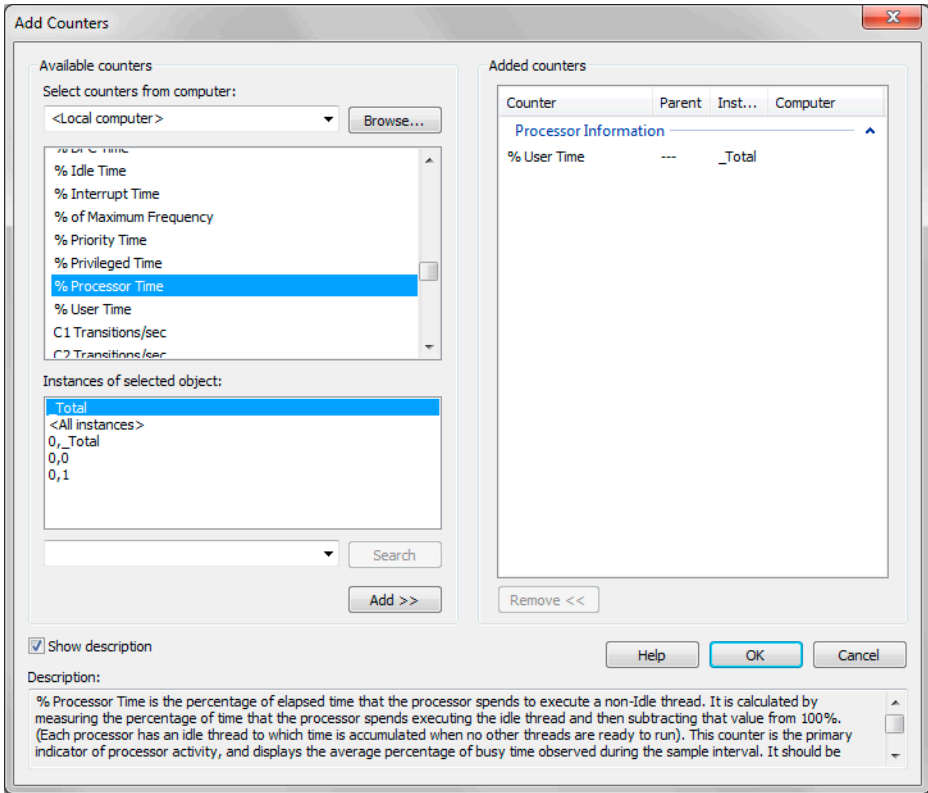


Figure 4.10: Add Counter dialog – "Show Description" check box is selected in the lower left of the window.

The initial display of the performance monitor shows a real-time view of the system with the counters being measured shown in the display. Data can be recorded to either disk or a database, and recorded performance data can be reloaded, viewed, and analyzed using this tool. This process, in addition to the range of counters and their meaning, was detailed in Chapter 3.

Conclusion

In this chapter, we looked at how to set up a performance testing rig. It is essential that this fundamental piece of infrastructure be set up correctly, otherwise we risk invalidating test results, and wasting considerable time and money in the process.

With a fully functional test rig, we are now able to record, execute, and analyze our performance tests. The test rig, once set up, can form an important facet of an organization's overall infrastructure. Setting up the rig is typically a one-time process that can be utilized for multiple projects, and which provides ongoing benefits.

The next chapter will focus on the recording, creation and automation of performance tests, as well as defining the load tests themselves. Later chapters will deal with the execution and analysis of the tests.

Now that we have built our new toy, it's time to play with it.

Chapter 5: Creating Performance Tests

Having a performance test rig is not very useful if you don't have any tests to execute with it. Creating tests is a relatively easy task, but it is important to have the functional path breakdown that was mentioned in Chapter 2. This way, there is a defined path to execute when recording the tests.

Basic solution structure

To start with, we need to create a separate project to house the performance tests. This can be added to the same solution that houses the main project and source code of your application, but it is best to place the performance test project in a project outside the main source code branch. This will prevent the main source code tree from being affected by the extra build time it takes to compile the performance test project, and will also keep the test outside the view of the main development team. It is a completely independent project that has no dependencies on the main project or solution being tested.

With this in mind, create a new Test project in Visual Studio by opening the **File** menu, selecting the **New Project** menu option, then the **Test** project type, then selecting **Test Project** in the project template window. Name the project *PerfTests*, select a directory location and click **OK**.

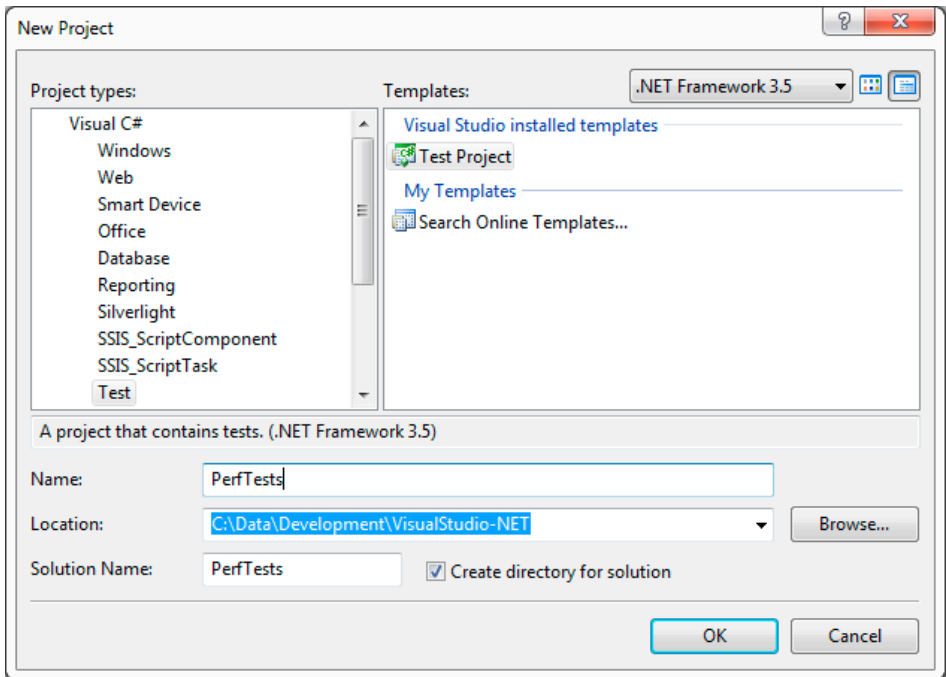


Figure 5.1: Creating a new performance test project.

Once you have performed this step, you should end up with a solution looking similar to Figure 5.2.

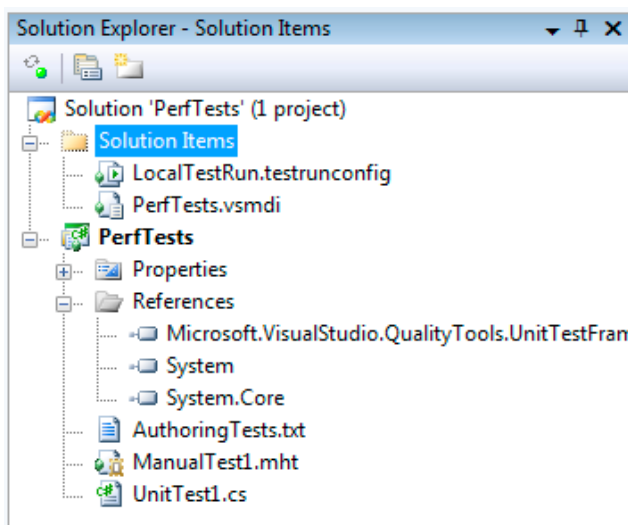


Figure 5.2: New performance test project – Solution Explorer view.

Note the presence of the **AuthoringTests.txt** and **ManualTest1.mht** files. The former provides general information around testing, and the latter provides a template for manual tests to be written. For the purposes of performance testing, these files can be safely deleted from the project. We can also remove the **UnitTest1.cs** file, as it does not apply to load tests.

In the solution items folder, the **LocalTestRun.testrunconfig** file holds general settings for the test run, such as which controller to use, test run naming schemes, deployment requirements and so on. These items can be edited by opening the **Test** menu, selecting the **Edit Test Run Configurations** option, and then selecting the test run configuration file.

There is currently only one configuration, but you can have several. Selecting this option displays a configuration dialog.

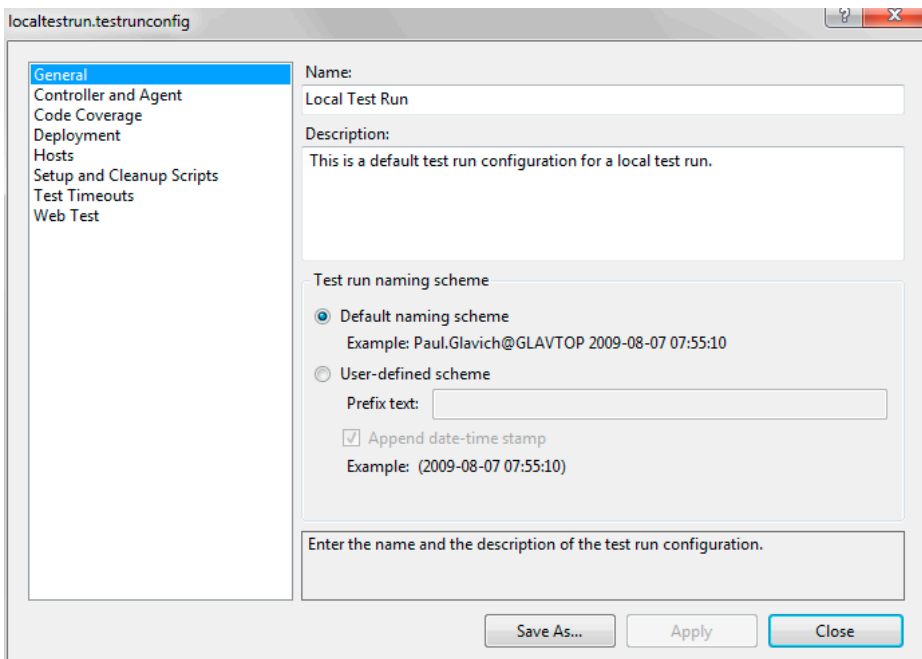


Figure 5.3: Test run configuration dialog.

For basic tests, these options can be left at defaults; but I'll cover these in more detail a little later.

Recording the web tests

The functional test breakdown and how it should be structured were discussed in previous chapters. When recording tests, the functional breakdown is used to determine what tests to record and what functions to exercise when recording them.

In order to be able to record a test, the machine used to do the recording must clearly be able to access the web application being tested. When a test is recorded, an instance of Internet Explorer is automatically launched by Visual Studio. While browsing using the newly launched instance of Internet Explorer, all web activity is recorded. This includes all browsing activity, not just those requests targeted at your application. This is why it is important to have a clear functional path, and only exercise singular aspects of the application at any given time. This way, the individual tests can be executed, recorded, and later attributed to the appropriate aspects of the application. When the tests are replayed, they are applied to the overall run according to what percentage of activity is set up within the test run (this will be detailed in later in this chapter. If you record too much activity in a single test, it becomes very hard to accurately allocate functional activity within the run. It then becomes very difficult to simulate the desired user behavior (and therefore the expected load) when spreading tests across a load test run.

To start recording a test, right-click the test project and select either the **Add > New Test** or the **Test > New Test** menu option (see Figure 5.4).

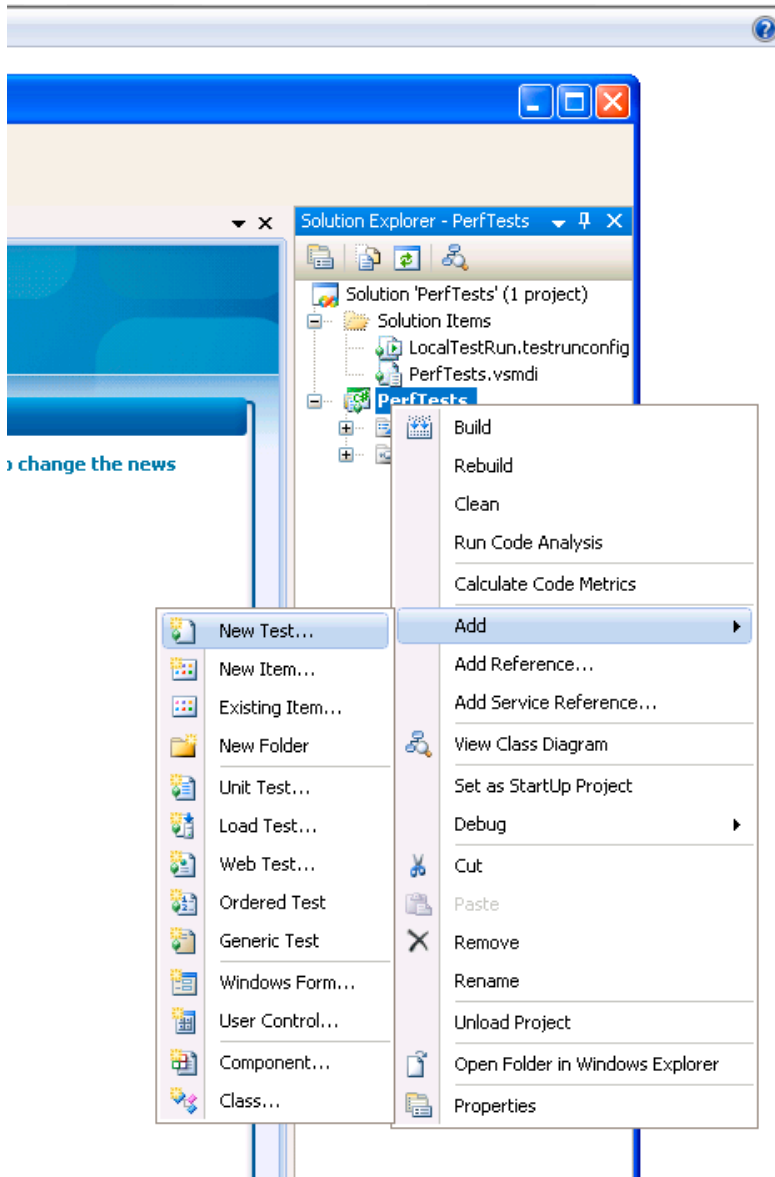


Figure 5.4: Add New Test option.

Bear in mind that Visual Studio Team Test is the minimum requirement installation in order for the Web Test option to be available. Once the **New Test** option is selected, a dialog allowing the user to select what type of test to add is presented. Selecting **Web Test** (Figure 5.5) will launch an instance of Internet Explorer and invite the user to begin navigating the site.

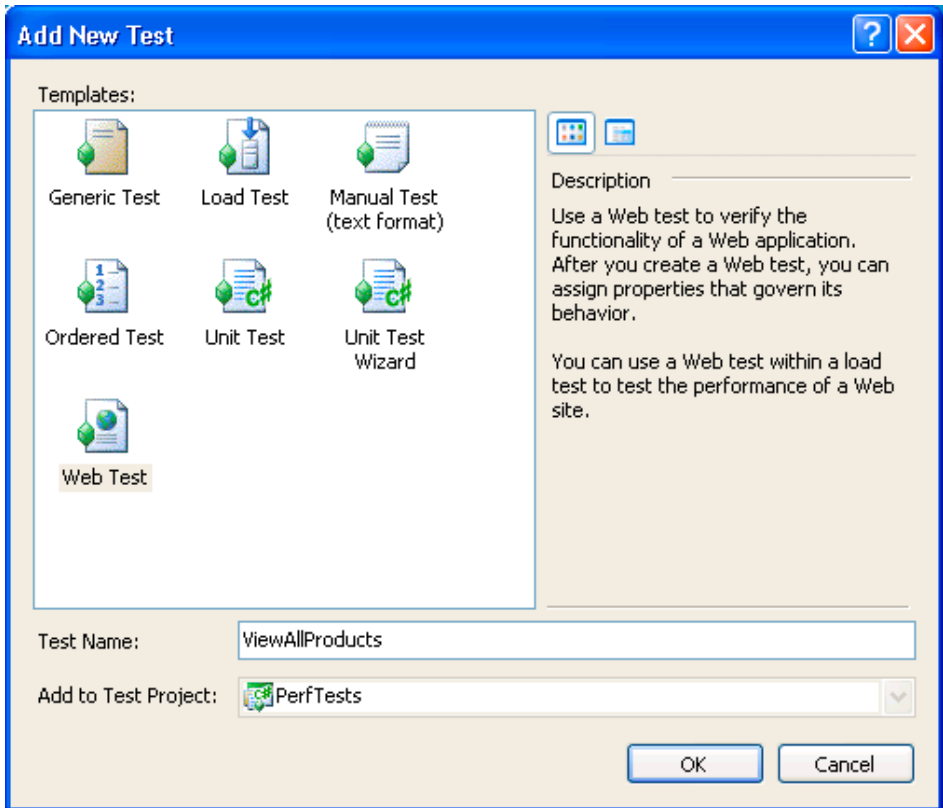


Figure 5.5: Selecting a Web Test dialog.

Once the test name is entered and you've clicked **OK**, the Internet Explorer instance is opened, and all actions are recorded as part of the web test. It is important to note that, if a particular home page is set, then accessing this home page will also be recorded, even if it has nothing to do with testing the application itself. It is best to set the Internet Explorer home page to a blank page so that no requests are recorded that do not pertain to the application being tested.

Once Internet Explorer is launched, start navigating the site in line with the functional area being tested. A web test of my own sample application can be seen in Figure 5.6.

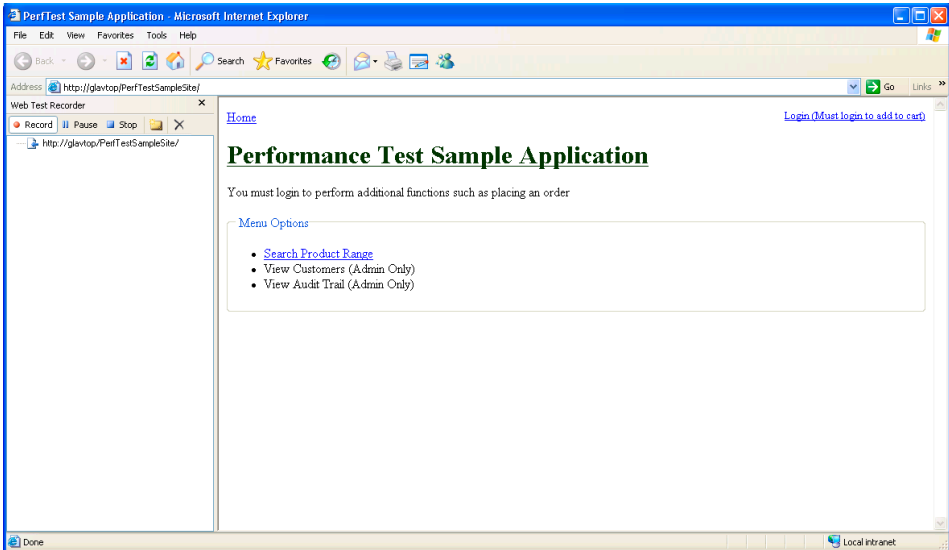


Figure 5.6: Recording a web test.

On the left side of the Internet Explorer window is a pane showing the currently recorded web test actions. As you navigate the site, each request or post will be listed in this pane.

Continue using the site according to the functional aspect being exercised and, once you have completed recording the necessary series of actions, close down the instance of Internet Explorer. Visual Studio will add the recorded **get** and **post** actions as part of the web test as shown in Figure 5.7.

Note

*Be sure to name your tests appropriately. Good organization will make it easy to set up the appropriate weighting for the tests once recorded. Having tests named **Web test1**, **Web test2**, etc., means you'll need to actually go into the test, and perhaps run it, to find out what aspect of functionality the test exercises. Instead, name your tests verbosely, such as **LoginAndViewAllProducts**, or **LoginViewCartThenLogout**.*

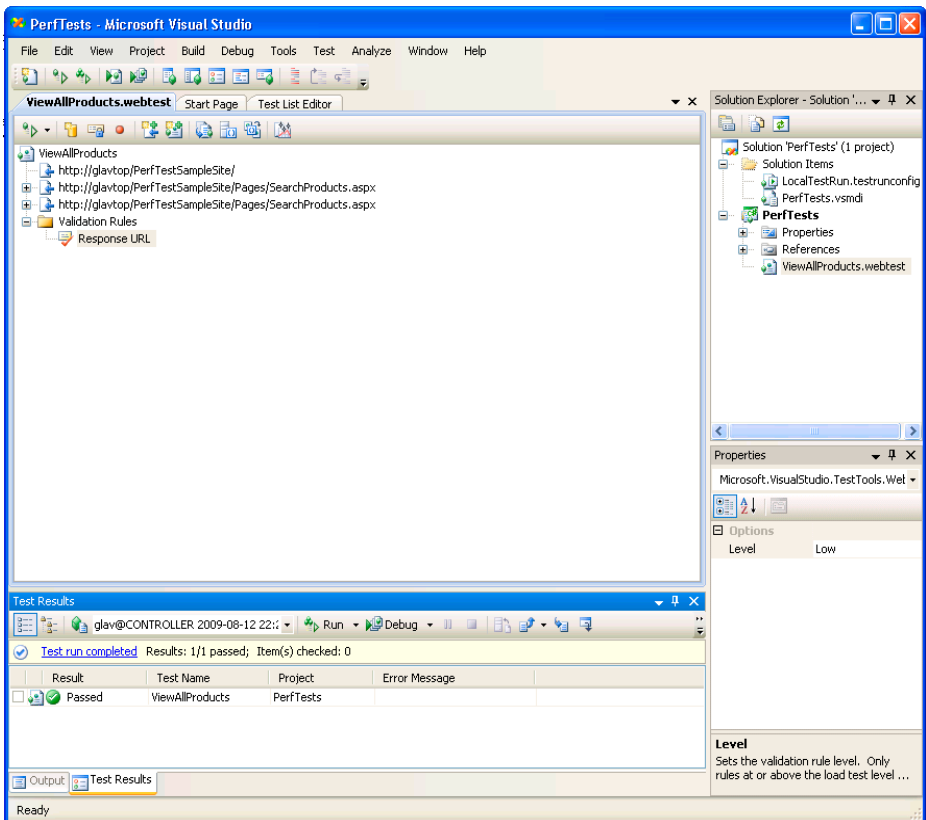


Figure 5.7: Recorded web test.

The web test actions are listed sequentially according to the URL accessed for that particular request. If a request has other elements to it, such as query string parameters, post parameters, or hidden fields, then these are associated with that request, and can be seen by expanding the request tree view. Clicking or selecting on the requests will display their properties in Visual Studio's properties windows.

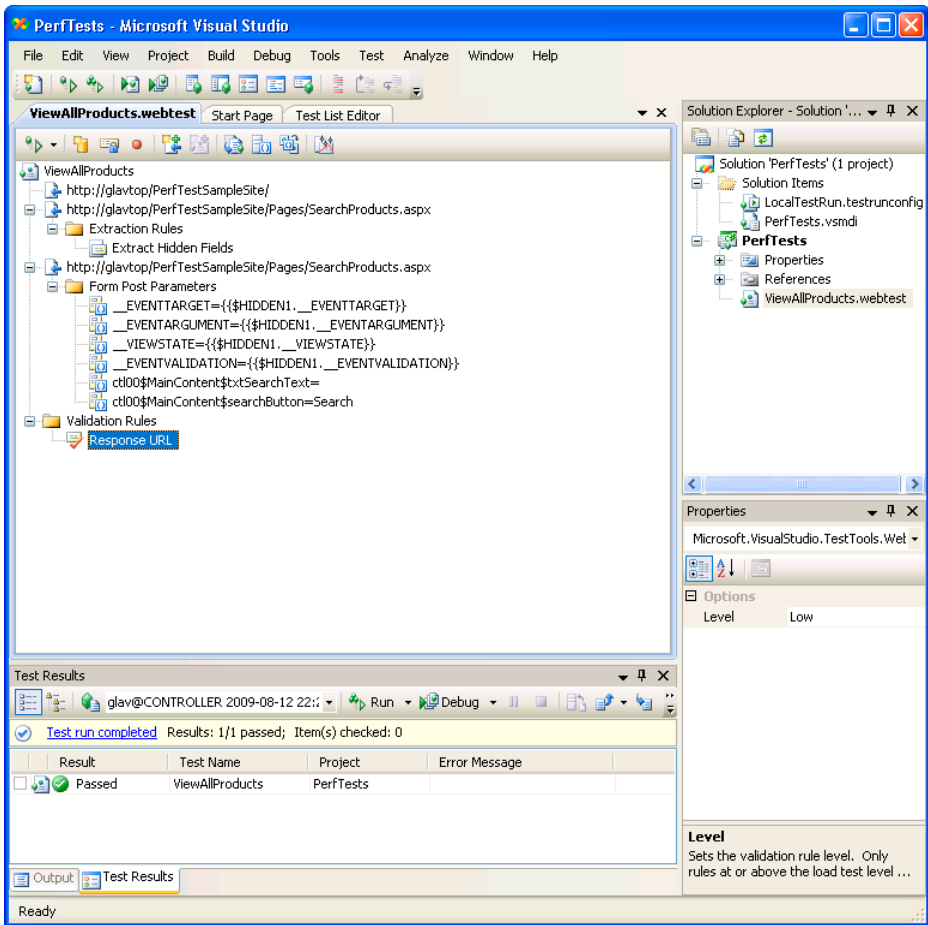


Figure 5.8: Expanded web test request.

Figure 5.8 shows various form post parameters such as an ASP.NET viewstate and other parameter values forming part of the POST payload of the request, all of which are easily visible

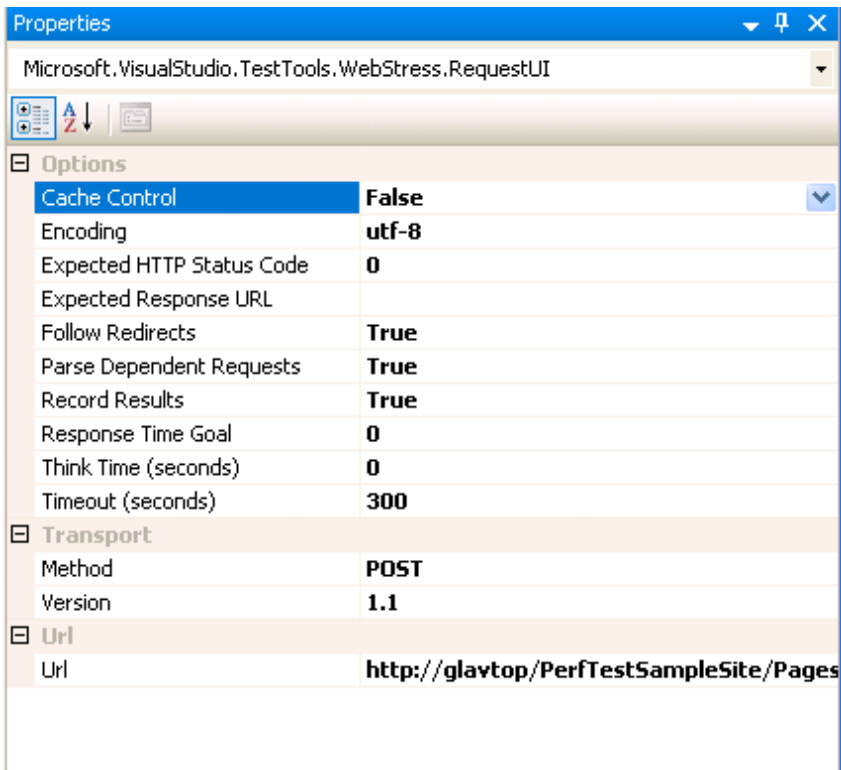


Figure 5.9: Web Test Request properties.

Figure 5.9 shows some common properties of all web test requests, together with values specific to this example request. Each request can have these properties modified to achieve different effects within the overall test, and potentially pass or fail a particular test.

- **Cache-Control** and **Encoding** are relatively self-explanatory. Cache-Control determines whether the request can be cached, and Encoding refers to the character encoding used for the request itself.
- **Method** and **Version** refer to the specific HTTP properties of the request.

On the other hand, the properties that affect the pass or fail status of a single request are:

- **Expected HTTP Status Code**
Determines what the expected status code of a result of issuing this request should be. This is a standard HTTP status code, such as 404 for "Not Found." Leaving this value as 0 means default browsing behavior will be used which, in turn, means that any 200 or 300 level code indicates a successful request, but any 400 or 500 level code indicates a failure.

- **Expected Response URL**
Indicates what the response or returned URL is after issuing this request. A blank value will not expect any particular URL but, if one is entered and a different URL is returned, this indicates a test failure.
- **Response Time Goal**
Indicates the maximum time (in seconds) that this request should take to execute. If the request exceeds this time, then the test is deemed failed. No value for this indicates no expected or maximum response time.
- **Timeout (seconds)**
Indicates the maximum amount of time (in seconds) that this request can take to execute.

Note

Test failures do not stop performance tests from running, but simply add to the metric or total data for failed tests.

The **Parse Dependent Requests** property is interesting, because it determines whether any further requests which would be typically required to satisfy this request in a real world scenario are made as a result of issuing the web test request. For example, stylesheets, images and script files are often requested by a browser after an initial request to the resource is made and the HTML has been parsed. Web Tests will simulate this behavior by default as these are considered dependent requests.

However, in some cases you may want to disable the parsing of dependent requests to enable you to test only the processing efficiency of the web application, and not rely on possible latent connections for resources not directly affecting this performance. For example, if a page makes requests to external parties, such as Google, to request JavaScript files, or to a marketing company providing analytics for the site, then you may want to remove these requests from performance testing and only concentrate on your application. Obviously, these requests still affect the overall perceived performance of the request itself, but you may have little control over them, and not want to skew the measurements of your application's performance with these figures. If you're feeling really fine-grained, it may be useful to isolate each aspect of the request to further analyze what are the limiting factors.

Record Results indicates whether results for this request are recorded in the database. If this request is of no interest to you, then perhaps you may not wish to record any data about it, thus minimizing noise within the results.

Overall, the default settings for web test requests attempt to mimic the default browser behavior. Initially at least, it is best to leave these settings as is, though during the course of

performance test analysis, further investigation may require experimentation with them. This will be discussed later in the book when we look at performance load test metrics and iterative testing approaches.

Test replay

So now we have some tests. We can easily replay these by double-clicking them in the solution explorer to display them in the main window, and then selecting a **Run** option from the **Run** menu. The **Run/Play** menu is located in the top left of the main display window.

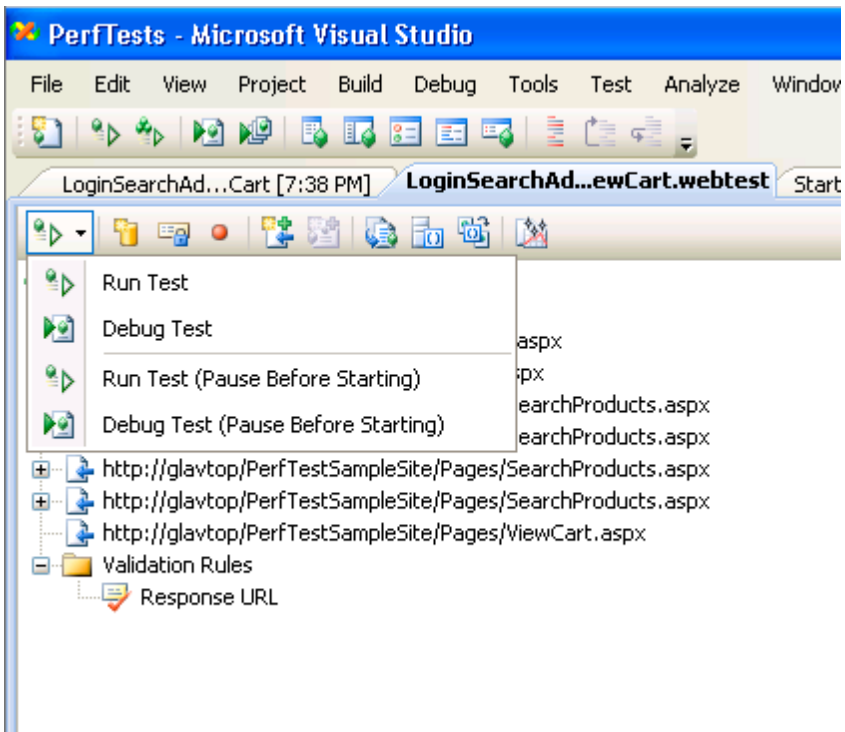


Figure 5.10: Run/Play test menu.

The menu shown in Figure 5.10, shows both the **Run** and **Debug** execution modes for replaying a test. **Debug** mode provides the ability to step through each request, examining the value of variables just as you would a normal .NET application. The options with **(Pause Before Starting)** against them will prepare to run the test, but will pause just before beginning execution, allowing you to step through each request using the **Step** menu option located to the right of the **Run** button. Also to the right, in order, are the **Pause** and **Stop** test options.

Once a test has been run, the status will be displayed in the **Test Results** window, as well as in the top left window of the test results main window.

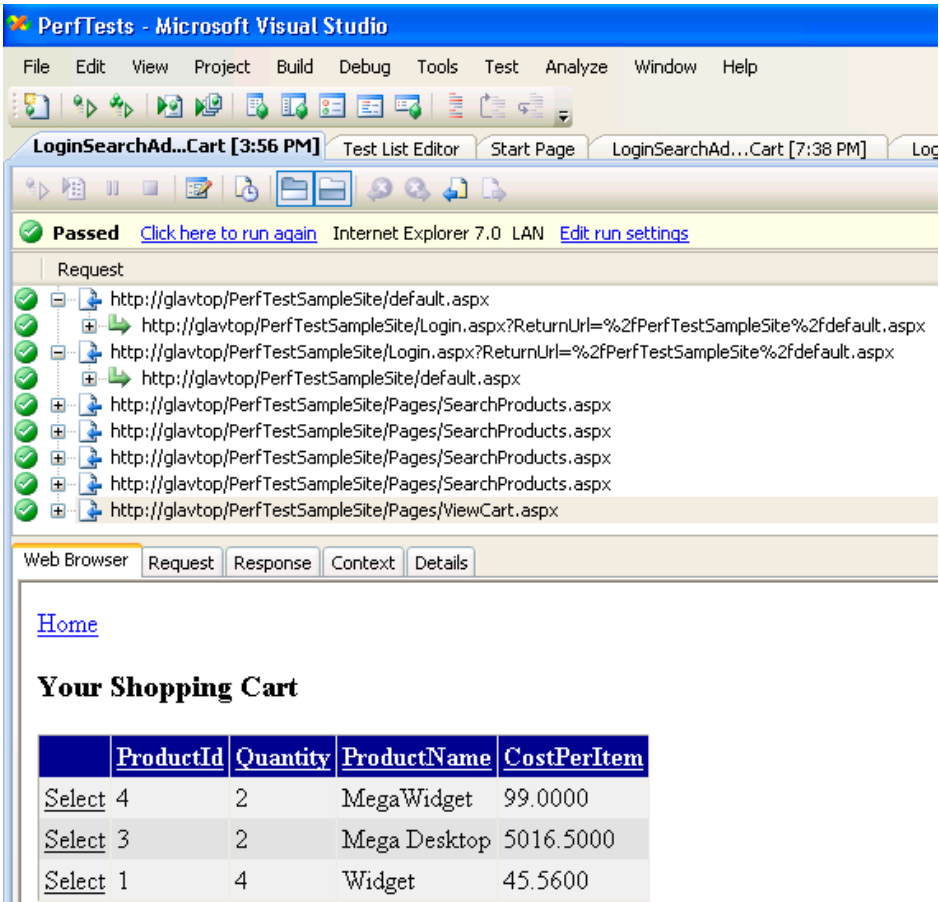


Figure 5.11: Successful test run.

Whether the test has run successfully or not, you can now examine each request in detail. Selecting a request will display its particular results in the **Web Browser** tab shown just below the test results in Figure 5.11. In addition, you can expand the main request to show any dependent requests, and selecting a dependent request also shows its results.

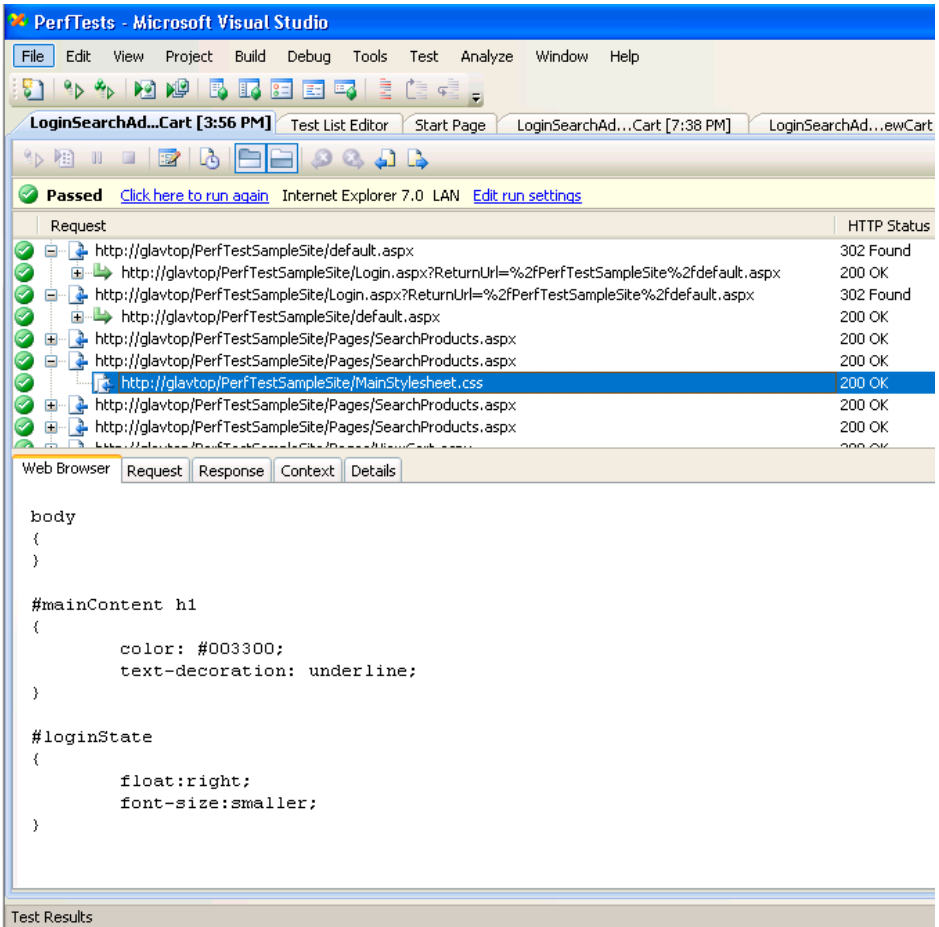


Figure 5.12: Selecting a dependent request.

In the example in Figure 5.12, we can see that the dependent request was a CSS file. To the right of the **Web Browser** tab are the **Request** and **Response** tabs.

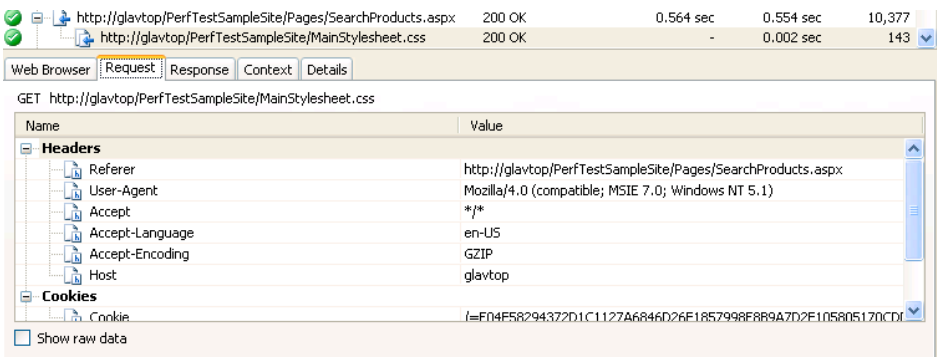


Figure 5.13: Web Test Request tab.

The **Request** tab (Figure 5.13) allows you to examine individual header elements in a formatted, table-like display, or you can view the raw request by selecting the **Show raw data** check box in the bottom left-hand corner.

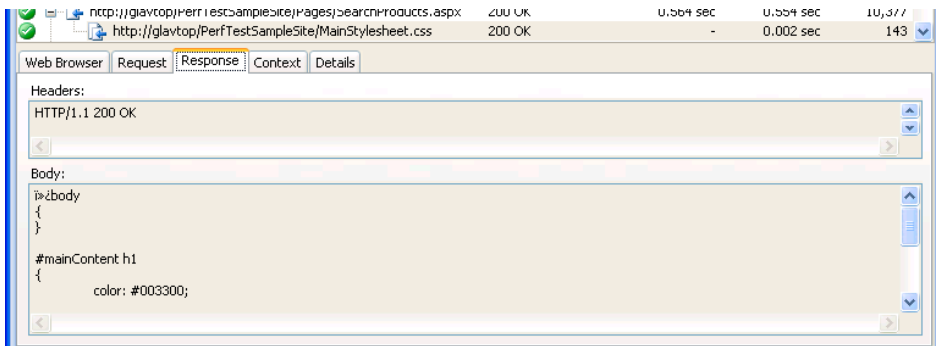


Figure 5.14: Web Test Response tab.

The **Response** tab (Figure 5.14) shows the response to the selected request, with the information segregated to display the Headers and Body responses in separate fields.

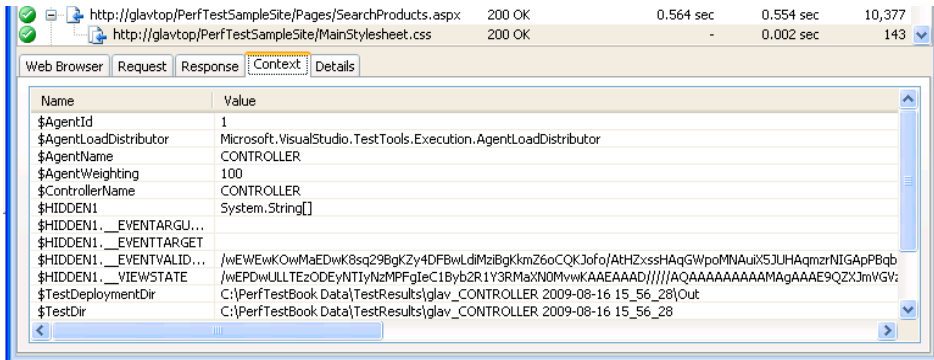


Figure 5.15: Web Test Context tab.

The **Context** tab shows any contextual information related to the current request. This typically involves hidden fields in the request, test agent id, environment variables such as test directory, deployment directory, and other elements depending on the request itself.

The **Details** tab lists any validation or extraction rules that have been applied to the test. Extraction rules are custom actions designed to extract data from a test response and assign it to a variable. Many of these are added by default to requests by the Visual Studio test engine when the test is recorded.

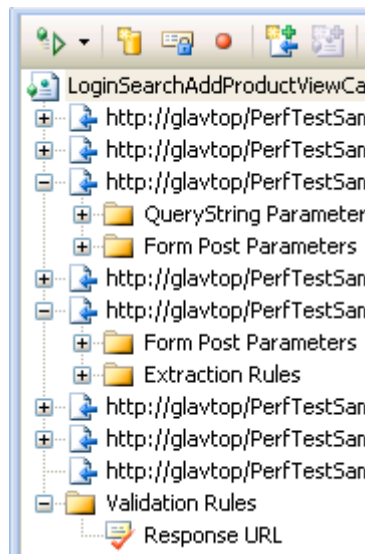


Figure 5.16: Web Test Validation rule.

Validation rules will cause a test to pass or fail based on an expected value within the test. By default, a **Response URL** validation rule is added to the last request of each test as shown in Figure 5.16.

Selecting the last request and then selecting the **Context** tab shows the result of the validation rule.

When a test is recorded, the explicit URL is recorded along with it. Sometimes, you may be recording the test against a development server, but wish to replay the test against a different server, perhaps a UAT (User Acceptance Test) server. It would be very time consuming to have to alter the URL of each request to reflect this. It would be nice to have a way of assigning a variable to the URL and have Visual Studio use this when replaying tests.

Fortunately, there is direct support for this via the **Parameterize Web Servers** option, which is found in the web test toolbar.

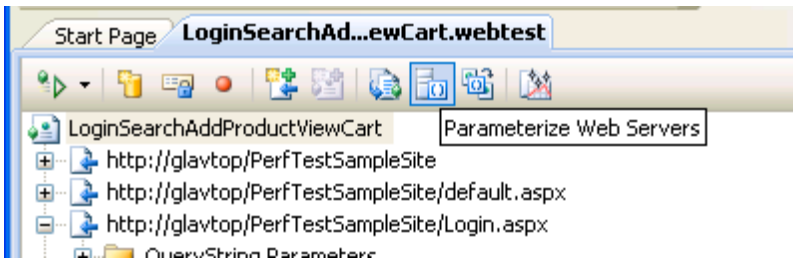


Figure 5.17: Parameterize Web Servers option.

Clicking this button will display a dialog allowing you to specify the variable name assigned to the web server address. By default, this is **WebServer1**.

Clicking the **Change** button will allow you to alter the name of the variable. If there were more than one server detected as part of the whole test, for example, if an HTTPS address were also used, and any other addresses as part of the test, they would be listed here.

Enter a name for the server you are targeting. This will then replace all the explicit URLs with a dynamic variable name according to what you have entered.

Once the explicit server address has been substituted with a context variable, changing the server address for all requests is then just a matter of altering the value of that context variable. Clicking on the same **Parameterize Web Servers** button will allow you to specify a different server.

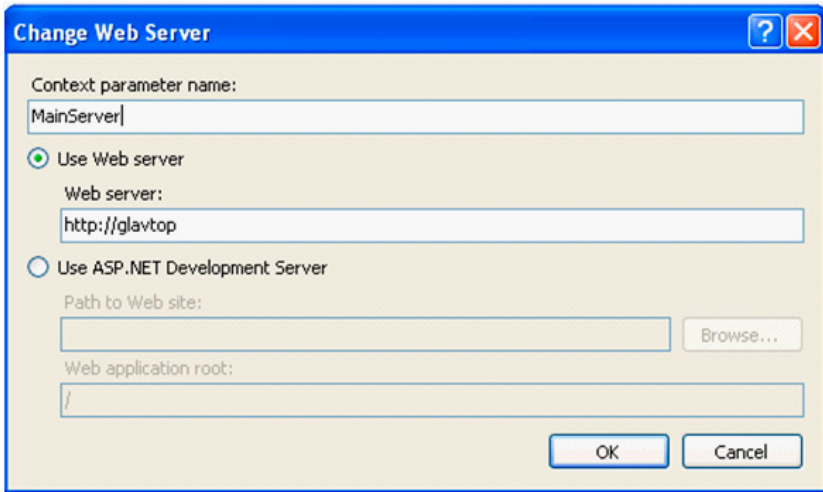


Figure 5.18: Assigning a web server a variable name in a web test.

Once you have assigned a name to the server address, all explicit references to that address will be replaced by a context variable with the name you have specified. The test will be altered accordingly.

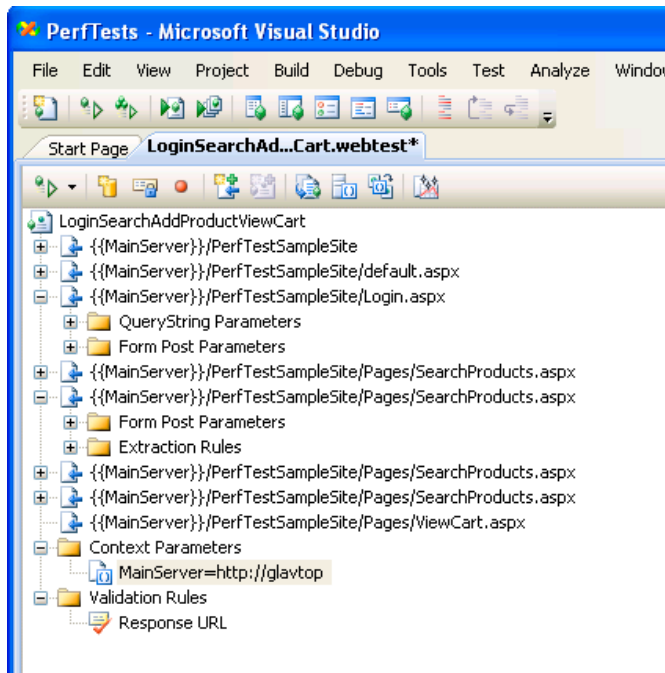


Figure 5.19: Web Test after server parameterization.

Data Binding Web Tests

Now that we have recorded our web tests, we would like to make them resilient enough to be played over and over again. In addition, we would like to introduce a random factor that ensures a different element is introduced for each web tests, just like in real life.

A good example of this is a user login. We don't want one single user acting as the basis for all of the web tests. Ideally, a wide range of users is required to better simulate the expected user activity on a real site.

In order to do this, we first need a data source to be used as the source for our users within the test. Visual Studio supports a database, CSV file, or XML file as the basis for a data source. It expects a simplistic structure for the data source and simply extracts the data within either the database table, CSV file, or XML file as it is presented. This means that all columns in the table and CSV file are used, and all elements within the XML file are used.

When creating the data source, it is best to keep things relatively simple. For our username and password data source, we will utilize a CSV file, which can be created with Microsoft Excel or any text editor.

Creating a data source for data binding

Let's start by creating a simple CSV file to provide a number of usernames and passwords to log in to our site. In Visual Studio, open the **File** menu, click on the **New File** menu option, select **Text File** from the dialog displayed, and select **Open**.

Enter the column headings *Username* and *Password*, separated by a comma. Next, enter a separate username and password, separated by a comma, on each line of the file. The file should look similar to this:

```
Username, Password
```

```
admin, password
```

```
test, password
```

```
viewer, password
```

Save this file within the same directory as your performance test project.

Note

To better organize your performance test project, it is good practice to create a data sources folder if using CSV or XML files for your data sources, and place the files in that directory. This is entirely down to personal preference, though, and you can organize the data source files as you see fit. Having them in a separate folder reduces clutter in the general project and keeps tests, code, and data source files separate.

You will probably want the file to be included as part of your test project so that it is included with source control and forms a component of the project as a whole. This is not strictly necessary, but makes sense and allows better organization of your test projects.

In your test project, ensure the web test that you would like to add a data source to is opened in the main window, and click the **Add Data Source** toolbox button.

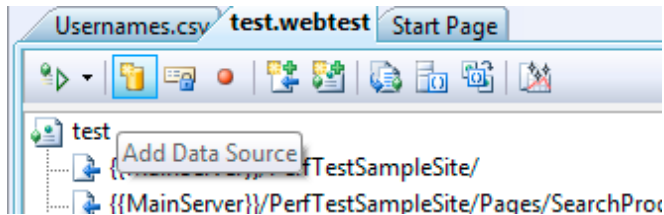


Figure 5.20: Add Data Source button.

A dialog is presented where you can specify the data source type and name. For our example, we will select the **CSV** file option and name the data source, appropriately enough, *UsernamesDataSource*.

Next, we choose the file that will actually be used as our data source. Either by browsing or by specifying the file path, select the data-source file you have created.

Visual Studio will parse the file, determine what data is present, and display it in the dialog window. Click the **Finish** button to add the data source to your web test. You should now see a **Data Sources** node within your web test, listing the data source you have just added, as in Figure 5.21.

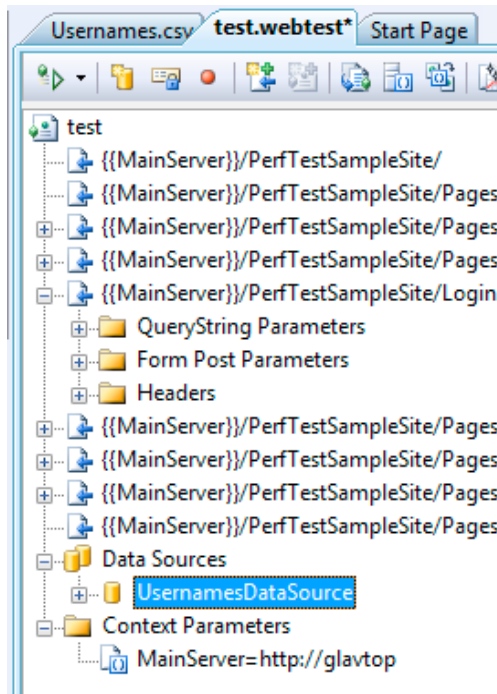


Figure 5.21: Web test with a data source added.

This data source can now be utilized within the test itself. In this case, we will use the usernames and passwords contained in the file to feed into the web test. First, expand the test action that requests the **login.aspx** page.

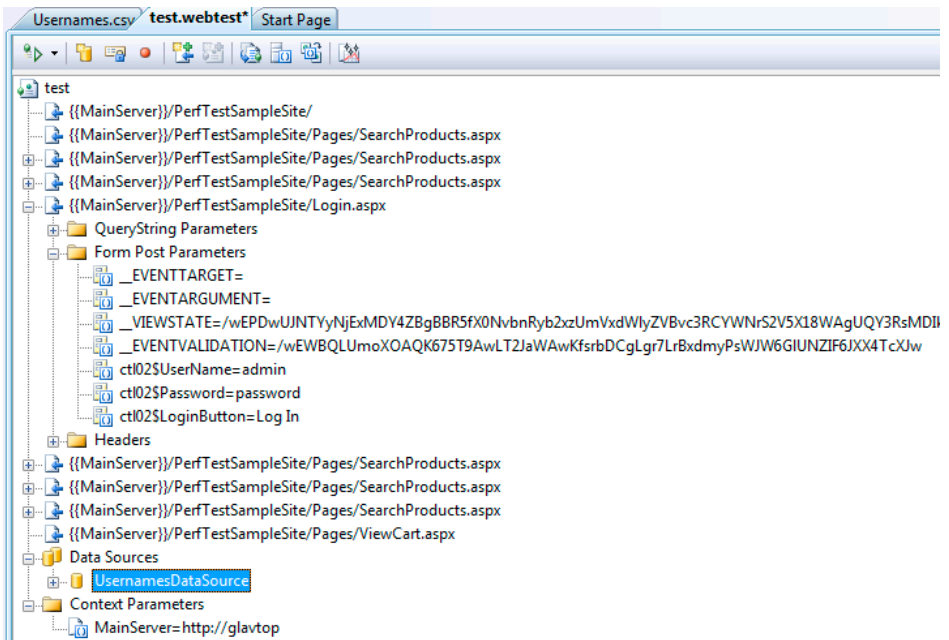


Figure 5.22: Expanded login test action showing form parameters.

This request will contain some form post parameters which the page will post to the server to log in. Currently, the values are assigned to exactly what was typed when the web test was recorded. Select the **Username** parameter and switch to the **Properties** window.

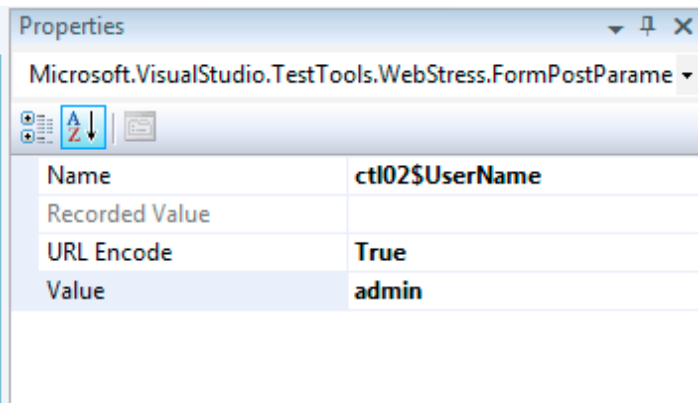


Figure 5.23: Properties window displayed for Username parameter.

The **Value** property contains the value of the form parameter. Clicking in the **Value** property field will present a drop-down box, which you should expand to show the values which are options for this parameter. One of the options is the data source that was previously added.

Expand this node, and also expand the data source that we added. This should display the fields within that data source, which we can use to data bind to the form parameter .

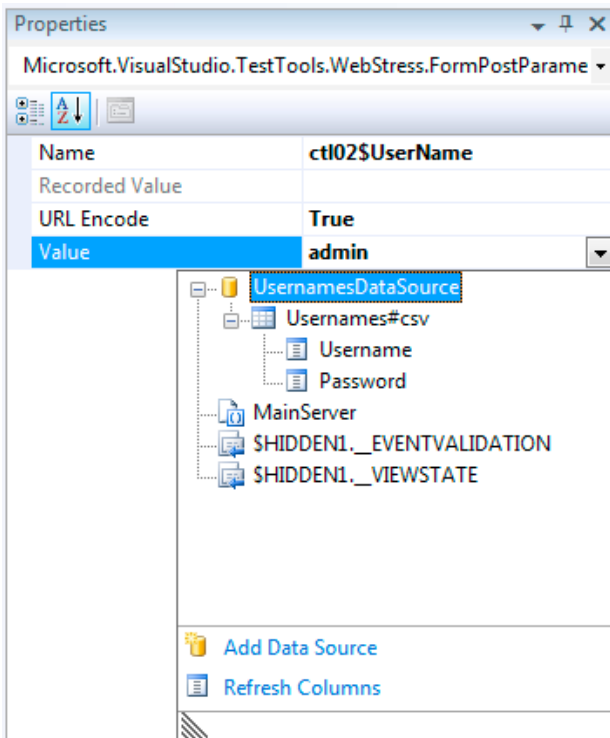


Figure 5.24: Select a field from the data source to bind to.

Select the appropriate field to data bind to the form parameter; in this example, the **Username** field. This binding will be reflected in the Properties window as well as in the web test action parameter.

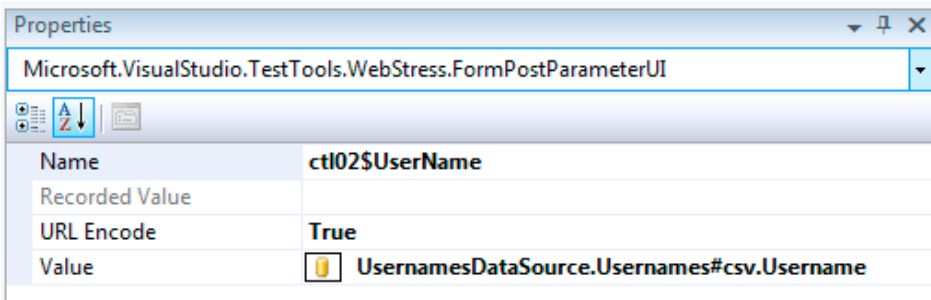


Figure 5.25: Data bound form parameter Properties window.

Now, each time this test is executed, the **Username** form parameter will be populated from the data source. By default, Visual Studio will start at the beginning and sequentially iterate through the data source each time the test is run.

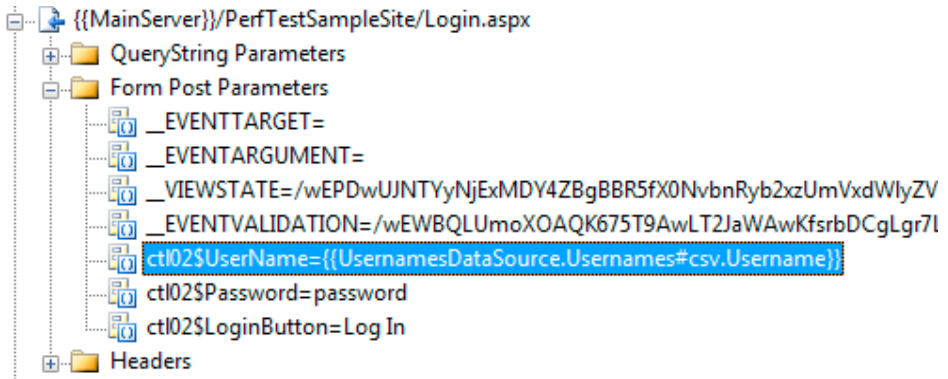


Figure 5.26: Data bound form parameter request action parameter window.

If you like, this behavior can be changed by expanding the **Data Sources** node in the main window, selecting the relevant data source sub-node, and viewing its properties in the properties window. Expand the drop-down in the **Access Method** property, and as the property values suggest, the method of access for a data source can be changed from the default sequential access to random order, or ensuring that each selected value is unique and no duplicates occur (see Figure 5.27).

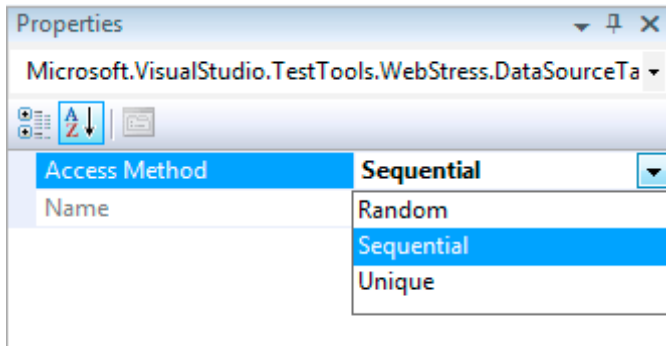


Figure 5.27: Access Method selection for a data source.

Since the data source has already been defined, the same technique can be used to assign a value to the password form element from the data source. Using a database is almost identical to using a CSV file, except that the connection to the database is specified, instead of the path to a CSV file.

Once the connection to the database is selected, a check box dialog is presented, allowing the user to select all the tables to be used as a data source. Any number of tables can be selected and added to the web test.

You may prefer to manage your performance test input data by having a single, separate database containing all the necessary tables for use as a data source. If you use this system, or indeed any databases as a data source, you must obviously ensure that all test agents can easily access the database during test runs.

Finally, XML data sources are something of a mix between CSVs and databases. Like a CSV file, the XML will contain the performance data input in text format, but it can also contain multiple tables. Take the following XML file as an example:

```
<?xml version="1.0" encoding="utf-8" ?>
<SampleData>
  <Users>
    <UserName>admin</UserName>
    <Password>password</Password>
  </Users>
  <Users>
    <UserName>test</UserName>
    <Password>password</Password>
  </Users>
  <Users>
    <UserName>viewer</UserName>
    <Password>password</Password>
  </Users>
  <Products>
    <Product>Widget</Product>
    <Quantity>3</Quantity>
  </Products>
  <Products>
    <Product>MegaWidget</Product>
    <Quantity>1</Quantity>
  </Products>
  <Products>
    <Product>Recombobulator</Product>
    <Quantity>16</Quantity>
  </Products>
</SampleData>
```

The XML file contains an arbitrary root node of <SampleData> and then subsequent child root nodes of <Users> and <Products> respectively. These nodes represent the tabular structure. The child elements of <Users> and <Products> represent the columns, extracted and used as the data source field. When adding the XML data source, the user specifies the location of the XML file and then selects a table to use from that data.

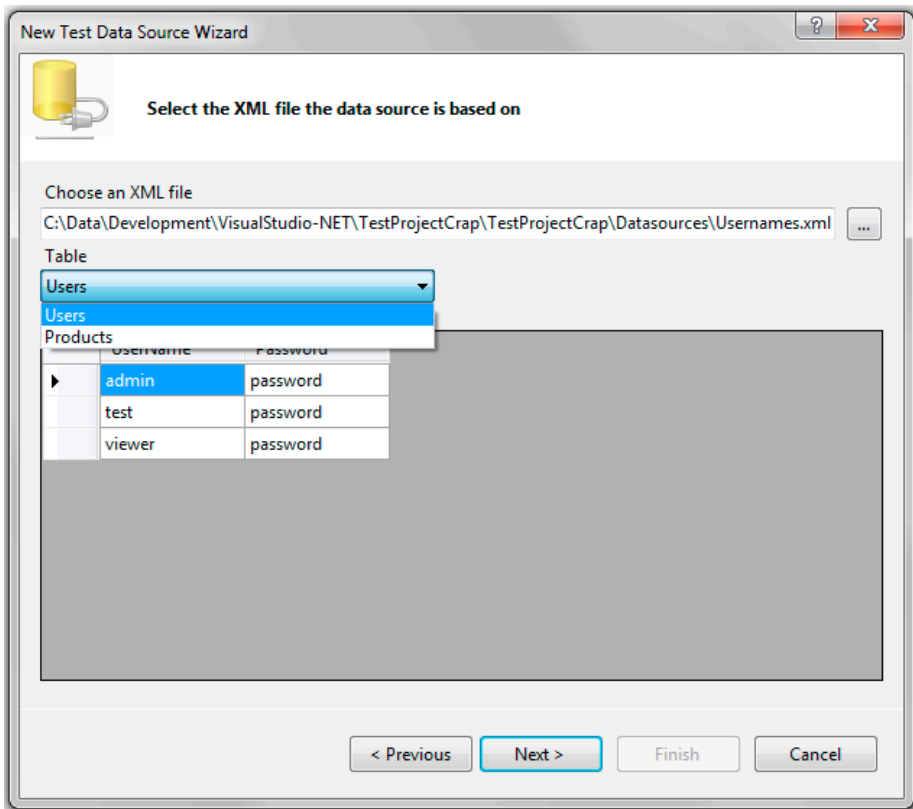


Figure 5.28: Selecting an XML data source and associated table.

While an XML file is more verbose, it does provide the advantage of being able to host multiple tables in one file.

Test deployment considerations

When a data source is used within a web test, that data source must obviously be made available to the test agents, so that the data can be extracted and used as context items during the test execution.

For a database, this means ensuring that the agents can connect to the database machine, and have the appropriate access to do so.

CSV and XML files must be deployed along with the tests in the compiled assemblies, otherwise the tests will not be able to locate the files. Indicating that the files need to be

deployed as part of the test execution requires the test run configuration file to be edited. To do this, open the **Test** menu, then select the **Edit Test Run Configuration** option, and then the **Local Test Run** (`localtestrun.testrunconfig`) option.

Note

The name of your test run configuration may differ, or you may have multiple configurations. Ensure you apply the changes to the configuration that you will be using.

A dialog is displayed listing all the configuration categories. Select the **Deployment** option from the left pane. In the right pane, multiple files or entire directories can be added, and these items will all be included when tests are deployed to each agent during a performance test run.

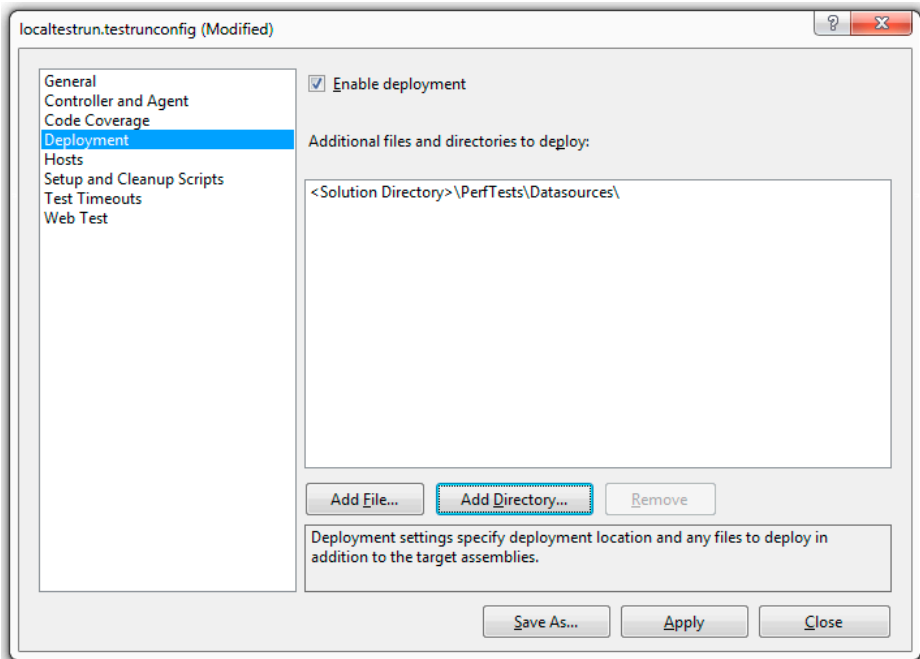


Figure 5.29: Specifying files or directories for deployment.

Having a single directory containing all the data sources means that the directory only needs to be defined once in the deployment configuration, and all files within that directory will be included in the deployment. If the director has been defined in this configuration, then any new data source files added to it later on will automatically be deployed. This is a really convenient deployment method.

Specifying singular files is certainly acceptable, as long as you ensure that each new data source file is added to the deployment configuration for all test run configurations that require it.

Web test code generation

Web tests provide a great deal of flexibility and a wide range of possibilities for customization and generating effective load. However, there may be instances where the supported customization methods are simply not enough, and more specific customization is required to create a valid test against the target system.

It is possible to generate code from web tests, and then simply treat that code like any other .NET development library component. You can then write whatever customization code is required. When code is generated for a web test, you will then have a separate copy of the web test, but in pure code. This will include any references to data sources, or other specific customization made to the test while it was still a web test.

Note

While relatively minor changes are often added to the generated code to achieve any desired effects, specifying data sources and modifying request properties is usually easier in web tests. For this reason, it's best to retain the original web test from which the coded test was generated.

To generate the code from a web test, select the **Generate Code** button from the web test toolbar in the main window.



Figure 5.30: Generate Code button for a web test.

A dialog will be presented, allowing you to specify a name for the generated coded test; once you've supplied that information, selecting **OK** will generate the code and add the file to the test project.

The code will contain the required attributes to define a data source (if one is used) and any other contextual items. The `GetRequestEnumerator` method will be overridden and will contain the execution path of the tests, as shown in the following code snippet:

```
public class MyTest_CodeGenerated : Web test
{
    public MyTest_CodeGenerated()
    {
        this.Context.Add("MainServer", "http://glavtop");
        this.PreAuthenticate = true;
    }

    public override IEnumerable<Web testRequest>
    GetRequestEnumerator()
    {
        Web testRequest request1 = new Web testRequest((this.
        Context["MainServer"].ToString() + "/PerfTestSampleSite/"));
        request1.ThinkTime = 8;
        request1.Timeout = 60;
        yield return request1;
        request1 = null;

        Web testRequest request2 = new Web testRequest((this.
        Context["MainServer"].ToString() + "/PerfTestSampleSite/Pages/
        SearchProducts.aspx"));
        request2.ThinkTime = 2;
        request2.Timeout = 60;
        yield return request2;
        request2 = null;
    }
}
```

This effectively shows one big method that yields requests for each web test action that was recorded. The `Web testRequest` object represents the request being executed, and contains properties as you would see in the Visual Studio user interface through the Properties window.

It quickly becomes apparent that, with a large web test recording, the generated source file can become quite large. Performing modifications and maintenance to just one source file can be quite time consuming, as there is a tendency to refactor and clean up the code.

In general, the best practice is to leave the generated code as untouched as possible. Any customized actions should be factored out into separate classes and assemblies, with the generated code simply calling into the customized methods or classes. This way, the originally recorded web test can be retained, and code regenerated if required, with only minor changes being needed to call into the customized code.

Extensibility through plug-ins

Previously, we discussed the ability to modify the properties of a request within a web test. An example property was the **ParseDependentRequests** property, which determined if items such as style sheets and images could also be requested by the original test action. It would obviously be time consuming and inconvenient to have to do this for each request in a web test if you wanted to disable all dependent requests.

Extensibility is made possible in Visual Studio Team Test through plug-ins. A custom plug-in can be created quite easily and applied to the web test. In order to create a custom plug-in, the required steps are listed below.

- Create a new class in the test project, or in a referenced project.
- Ensure the class inherits from the `Microsoft.VisualStudio.TestTools.Web testing.Web testPlugin` class.
- Override the appropriate method for your needs.

By way of example, let's create a plug-in to disable dependent requests for all requests in a web test.

- In your test project, add a new class to the project.
- Ensure the class is public, and that it inherits from the `Microsoft.VisualStudio.TestTools.Web testing.Web testPlugin` class.

The class should look similar to this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.VisualStudio.TestTools.Web testing;

namespace TestProjectCrap
{
    public class TestPlugin : Web testPlugin
    {
    }
}
```

- Override the `PreRequest` method.
- The `PreRequestEventArgs` parameter contains references to context elements such as the current request. In the implementation of the method, have the following code:
`e.Request.ParseDependentRequests = false;`
- Compile the code.

The completed plug-in code should look something like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace TestProjectCrap
{
    public class TestPlugin : WebTestPlugin
    {
        public override void PreRequest(object sender,
        PreRequestEventArgs e)
        {
            e.Request.ParseDependentRequests = false;
            base.PreRequest(sender, e);
        }
    }
}
```

This class is now a web test plug-in that can be added into any web test. To do this, click the **Add Web Test Plug-in** button, located in the web test toolbar above the test display window.



Figure 5.31: The Add Web Test Plug-in button.

A dialog will be presented showing a selection of available plug-ins to choose from.

Select the plug-in that was just created and click **OK**. The plug-in will now appear in the web test in a **Web Test Plug-ins** node as shown in Figure 5.32.

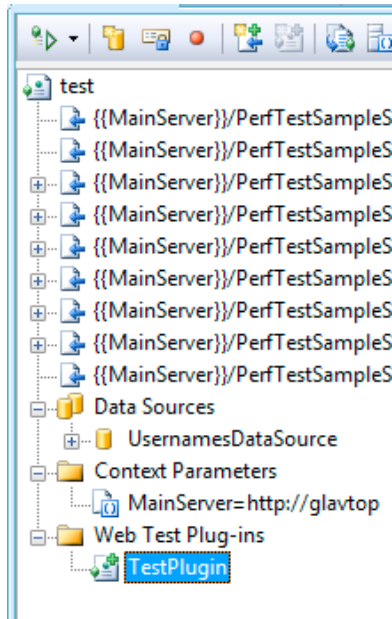
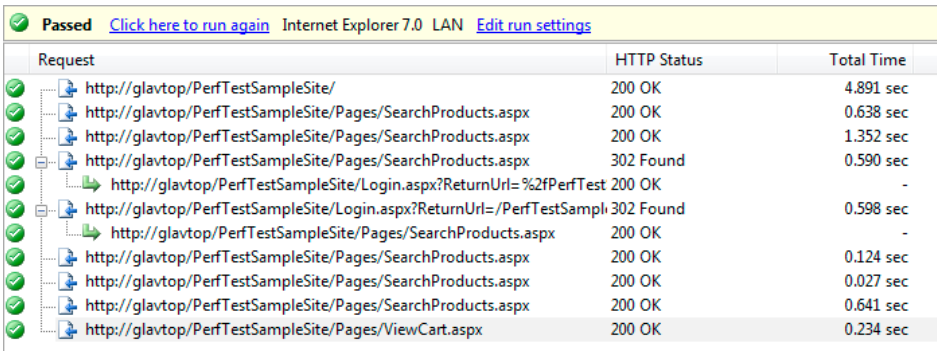


Figure 5.32: Web test plug-in added into the web test.

Now, each time the test is run, the plug-in will also take effect and set the parsing of dependent requests to "false" for every request. The changes are easy to see in these two screenshots showing a web test being run: Figure 5.33 without the plug-in added, and Figure 5.34 with the plug-in added.

Request	HTTP Status	Total Time
http://glavtop/PerfTestSampleSite/	200 OK	0.096 sec
http://glavtop/PerfTestSampleSite/MainStylesheet.css	200 OK	-
http://glavtop/PerfTestSampleSite/Pages/SearchProducts.aspx	200 OK	0.039 sec
http://glavtop/PerfTestSampleSite/MainStylesheet.css	200 OK	-
http://glavtop/PerfTestSampleSite/Pages/SearchProducts.aspx	200 OK	0.045 sec
http://glavtop/PerfTestSampleSite/MainStylesheet.css	200 OK	-
http://glavtop/PerfTestSampleSite/Pages/SearchProducts.aspx	302 Found	0.200 sec
http://glavtop/PerfTestSampleSite/Login.aspx?ReturnUrl=%2fPerfTest	200 OK	-
http://glavtop/PerfTestSampleSite/Login.aspx?ReturnUrl=/PerfTestSampli	302 Found	0.049 sec
http://glavtop/PerfTestSampleSite/Pages/SearchProducts.aspx	200 OK	-
http://glavtop/PerfTestSampleSite/Pages/SearchProducts.aspx	200 OK	0.038 sec
http://glavtop/PerfTestSampleSite/MainStylesheet.css	200 OK	-
http://glavtop/PerfTestSampleSite/Pages/SearchProducts.aspx	200 OK	0.030 sec
http://glavtop/PerfTestSampleSite/MainStylesheet.css	200 OK	-
http://glavtop/PerfTestSampleSite/Pages/SearchProducts.aspx	200 OK	0.065 sec
http://glavtop/PerfTestSampleSite/Pages/ViewCart.aspx	200 OK	0.041 sec

Figure 5.33: Web test run without the plug-in.



Request	HTTP Status	Total Time
http://glavtop/PerfTestSampleSite/	200 OK	4.891 sec
http://glavtop/PerfTestSampleSite/Pages/SearchProducts.aspx	200 OK	0.638 sec
http://glavtop/PerfTestSampleSite/Pages/SearchProducts.aspx	200 OK	1.352 sec
http://glavtop/PerfTestSampleSite/Pages/SearchProducts.aspx	302 Found	0.590 sec
http://glavtop/PerfTestSampleSite/Login.aspx?ReturnUrl=%2fPerfTest	200 OK	-
http://glavtop/PerfTestSampleSite/Login.aspx?ReturnUrl=/PerfTestSampl	302 Found	0.598 sec
http://glavtop/PerfTestSampleSite/Pages/SearchProducts.aspx	200 OK	-
http://glavtop/PerfTestSampleSite/Pages/SearchProducts.aspx	200 OK	0.124 sec
http://glavtop/PerfTestSampleSite/Pages/SearchProducts.aspx	200 OK	0.027 sec
http://glavtop/PerfTestSampleSite/Pages/SearchProducts.aspx	200 OK	0.641 sec
http://glavtop/PerfTestSampleSite/Pages/ViewCart.aspx	200 OK	0.234 sec

Figure 5.34: Web test run with the plug-in.

As you can see, the web test with the plug-in added does not make any dependent requests for CSS resources.

There are numerous other methods in the `Web testPlugin` class that can be overridden, and each participates in a different part of the request life cycle. All life cycle events that can be overridden follow the same pre- and post-condition pattern. Below is a list of those methods.

```
PostPage(object sender, PostPageEventArgs e)
PostRequest(object sender, PostRequestEventArgs e)
PostTransaction(object sender, PostTransactionEventArgs e)
PostWeb test(object sender, PostWeb testEventArgs e)
PrePage(object sender, PrePageEventArgs e)
PreTransaction(object sender, PreTransactionEventArgs e)
PreWeb test(object sender, PreWeb testEventArgs e)
PreRequest(object sender, PreRequestEventArgs e)
```

The naming of the methods makes it easy to understand at what point of the request life cycle they each take part, and this will factor in to how to implement the required plug-in to perform any desired custom actions. As you run more web tests, you will develop your own library of custom plug-ins to suit your personal needs or those of your organization.

Alternative ways of recording web tests

You don't necessarily have to use Visual Studio to record web tests. Having to install Visual Studio Team Test just for someone to record a web test may seem quite a high cost.

Luckily there is an alternative; you can also use a tool called Fiddler to record web tests. Fiddler is an HTTP proxy which allows you to capture all incoming and outgoing HTTP traffic for analysis. It was written by Eric Lawrence, of Microsoft, and it is a very powerful tool indeed. What is even better is that it's freely downloadable from:

[HTTP://WWW.FIDDLER2.COM/FIDDLER2/VERSION.ASP](http://www.fiddler2.com/fiddler2/version.asp)

After downloading and installing Fiddler, recording a web test is very simple. Start the Fiddler application, load Internet Explorer and perform the usual navigational steps to simulate the test you are recording. So far, this is no different from using Visual Studio to record test actions.

Once you have completed your actions, switch back to the Fiddler application, and you should have a screen with some recorded requests looking something like the screen in Figure 5.35.

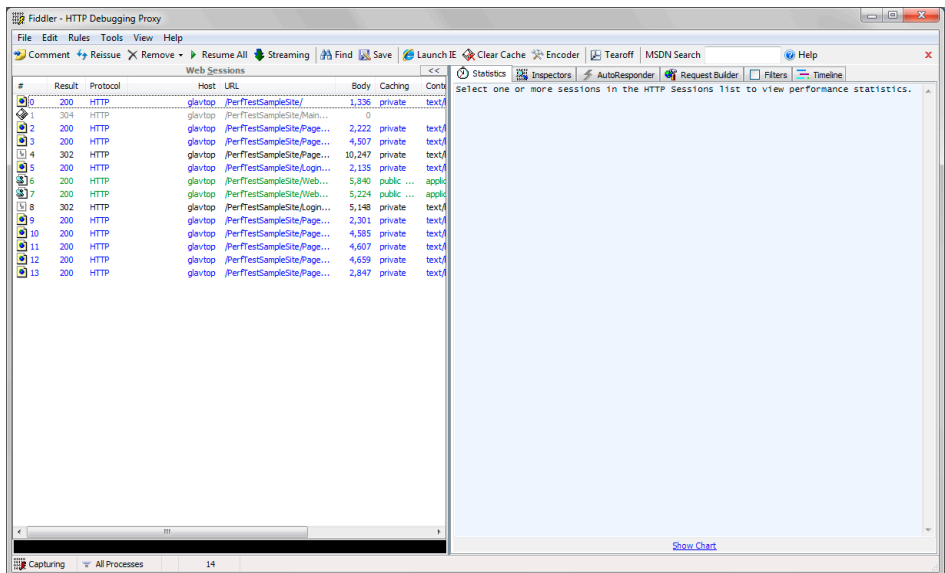


Figure 5.35: Fiddler showing captured requests.

Note

Once you have finished capturing requests for your test, it is best to either shut down Internet Explorer or stop Fiddler from capturing requests by deselecting the **Capture Traffic** option from the **File** menu, or alternatively by pressing **F12**. This is advisable because sometimes toolbars and plug-ins in the web browser can make requests which have nothing at all to do with the site or the test.

You will notice that selecting a request in the left pane of Fiddler shows the request details in the right pane. This is similar to Visual Studio, although the latter can show the request in a lot more detail.

To save the captured requests as a Visual Studio web test, select them all by clicking the **Edit > Select All** menu option, and then open the **File** menu, and select the **Save > Selected Sessions > as Visual Studio Web Test...** menu option.

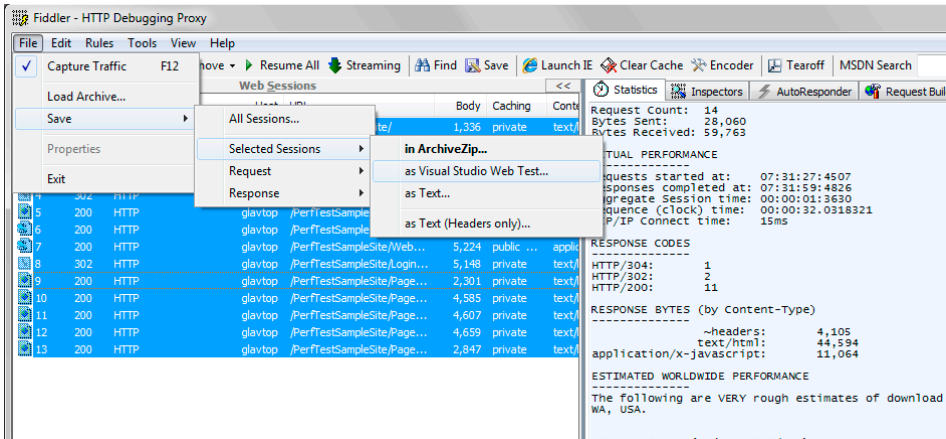


Figure 5.36: Fiddler – saving requests as a Visual Studio Web Test.

A dialog is then presented allowing the user to specify a name and location for the saved test. Before the test is saved, yet another dialog is presented, this time asking the user to specify the plug-ins used to execute against the recorded requests when saving. Simply accept the defaults and click **OK** to save the test.

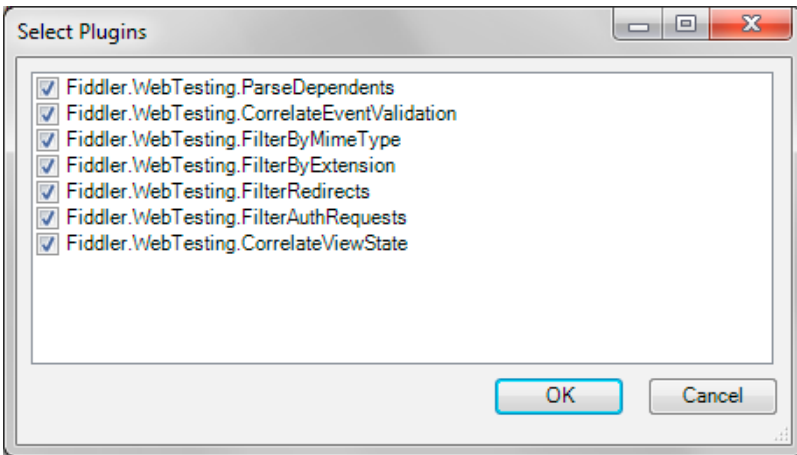


Figure 5.37: Fiddler plug-in selection when saving recorded requests.

The test is now saved as a normal Visual Studio Web Test that can be included in any Visual Studio Test project. To do this, simply use Windows Explorer to copy the saved web test file and paste it into the test project within Visual Studio.

These tests are exactly the same as those recorded within Visual Studio. Using Fiddler just provides a convenient and low cost (in terms of both price and installation effort) way of recording web tests to use for performance testing.

Considerations for load balancing / load balanced hardware

Production systems will often employ a technique called load balancing or load distribution. This is typically where more than one server is used to handle the load, or concurrent users, being applied to an application. This set of servers is often called web farm, or a farm of servers. In order to achieve this, load balancing software or hardware is employed to take the incoming requests, and send them to one of the servers in the farm – typically the server that is experiencing the least amount of load, or doing the least amount of work at the time.

So the question is: When setting up the performance test rig, should the production scenario be replicated exactly, with load balancing in place while performing tests?

The answer to this question is "yes and no." It is important to test this scenario, but it is important to first test against a single server without load balancing in place. The reason for this is that a single server will produce a set of results which can be considered, if you like, as the "single measure of performance." That is to say, a single server is easier to identify as a known quantity because you are isolating results to within the specifications of that machine only, and that is useful information to have. Adding more machines via load balancing will typically produce better overall performance, but that performance is still essentially based on this single measure of performance, as well as the load balancing solution itself. Having this measurement based on a single server also provides an easy set of metrics for subsequent analysis when changes are made. Having load balancing in place introduces another variable into the environment in which changes are applied, and thus increases the "surface area" of change and effect – which is a rather grand-sounding way of saying that you'll have more things to account for when quantifying effects if you try and factor in load balancing.

Having said that, when a good idea of performance capability is ascertained from a single server, introducing a load balanced scenario is also important to gauge the effect of horizontal scalability. This will determine how much of an effect an extra server provides. An often incorrect assumption is that, if one server can easily handle, say, 1,000 concurrent users, then two servers will be able to easily handle 2,000 users. Unfortunately, load balancing doesn't usually provide a direct linear increase in the capability of a system to bear load.

The amount of extra capacity that load balancing will provide depends, for starters, upon the load balancing solution itself. Software-based mechanisms, such as Windows load balancing software (WLBS) are usually not as effective as hardware-based ones, although they are often a lot cheaper. Software-based mechanisms are often suitable for smaller-scale web farms, though.

Also bear in mind that the method of load balancing is important. How does the load balancer distribute requests and load to other servers? Various methods are employed to do this, such as:

- **Round robin style**
This involves simply alternating between the available servers for each subsequent request coming in.
- **Connection based**
An incoming request is forwarded to the server that has the least number of open connections servicing requests.
- **Load based**
The server experiencing the least load will receive the next request. This brings up other questions of how the load balancer determines this information, and there are multiple ways to achieve that, as well.

Various load balancers will support at least one or more of the methods described above, some more efficiently than others. These variables make sure that the effects of load balancing are not as straightforward as expected.

Finally, the ability of the application to exist on multiple servers, with no affinity to any one server is also an important factor. This is referred to as being "stateless." Some load balancers can accommodate an application that requires "stateful" behavior, although I'll talk about this in greater detail in later chapters specifically covering optimization and load balancing.

This is why it's important to measure the effect of introducing load balancing. It will provide a more accurate gauge of expected performance, and allow better quantification of infrastructure requirements and system capability.

It is important to factor this consideration into your performance testing plan early on, so that the appropriate infrastructure tasks can be put into place to ensure testing can be performed against load balanced servers.

If you want to know more about load balancing, I'll cover it in the context of performance testing and application considerations later in the book.

Test automation

As the final part of creating a performance test rig, automation of test execution and collection of performance test results should be considered.

However, in order to automate a performance test, we first need a performance test to automate and, up until now, we have only discussed how to create and customize web tests. Web tests are the singular test items that will ultimately comprise a performance test using the functional scenario breakdowns mentioned earlier.

Now we need to create a performance test scenario and assign some web tests to it. For our purposes, we will create a basic performance test scenario to execute a series of web tests, and then automate their execution as well as the collection of the resulting performance data.

Creating a performance test scenario

Ensure you have a test project open in Visual Studio, containing some web tests that have already been recorded. Select the **Add Load Test** option from the **Project** menu, and you will be taken through the **New Load Test Wizard**.

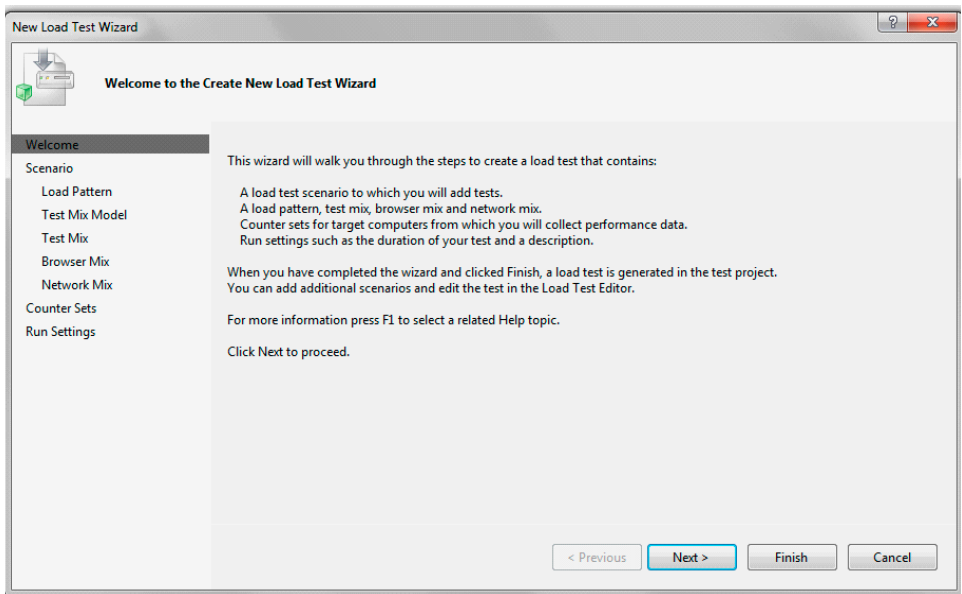


Figure 5.38: New Load Test Wizard – Welcome.

You will then be presented with a dialog around scenario settings. Enter a name for this load test scenario that will be simulated.

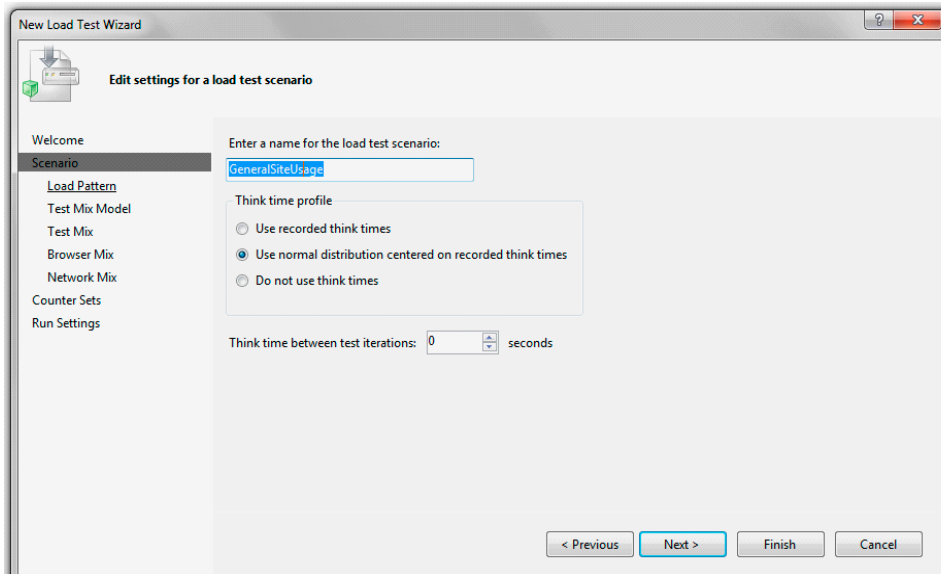


Figure 5.39: New Load Test Wizard – Scenario.

Leave the other settings at their defaults. The **Think times** settings determine whether the test will replay the idle periods where a user is thinking about what action to perform next; the default is to utilize an average distribution of think times based on the think times as they were recorded. Choosing the **Do not use think times** option incurs extra stress on the server and is useful for pure stress testing, but is not indicative of real world usage.

Next, you have the option to specify the initial settings for the concurrent user load to be simulated (see Figure 5.40).

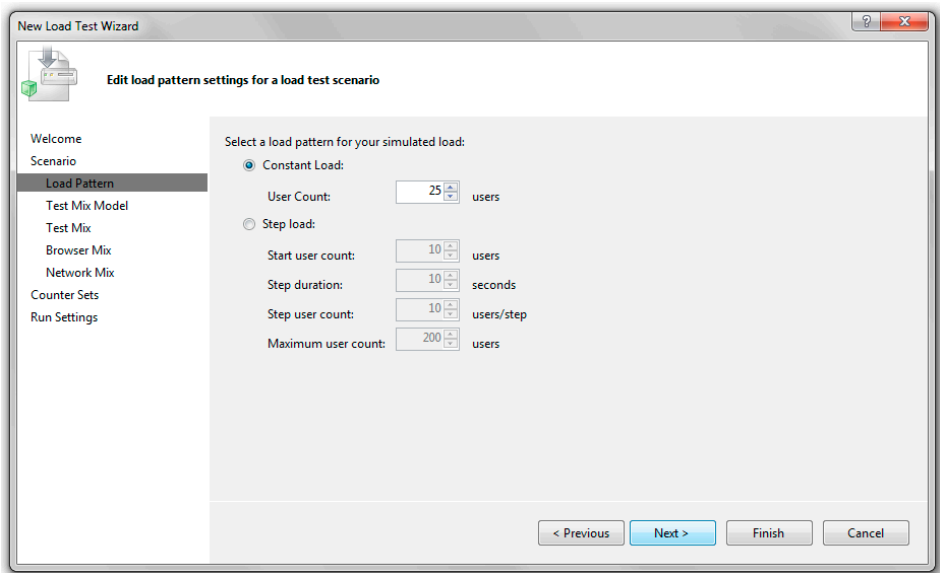


Figure 5.40: New Load Test Wizard – Load Pattern.

If you are unsure of the concurrent user load to specify at this point, simply accept the default setting of a constant 25 concurrent users, as this can always be edited later on.

The **Step load** option allows for a steadily increasing load to be applied, starting at a set user level (**Start user count**), then progressively adding a number of users (**Step user count**). You can also control the length of time for which to execute each progressive step (**Step duration**), as well as the maximum number of users to simulate (**Maximum user count**).

Moving to the next screen (Figure 5.41) shows the **Test Mix Model** dialog, which (unsurprisingly) allows you to specify what kind of test mix is required. For most requirements, sticking to the default of **Based on the total number of tests** is best suited to most organizations, and will allow you to best model the needs of the business and assign percentage weightings of test based on functional paths and use cases.

If you want to investigate other available test mixes then, as each method is selected, an information box to the right will describe the option in detail.

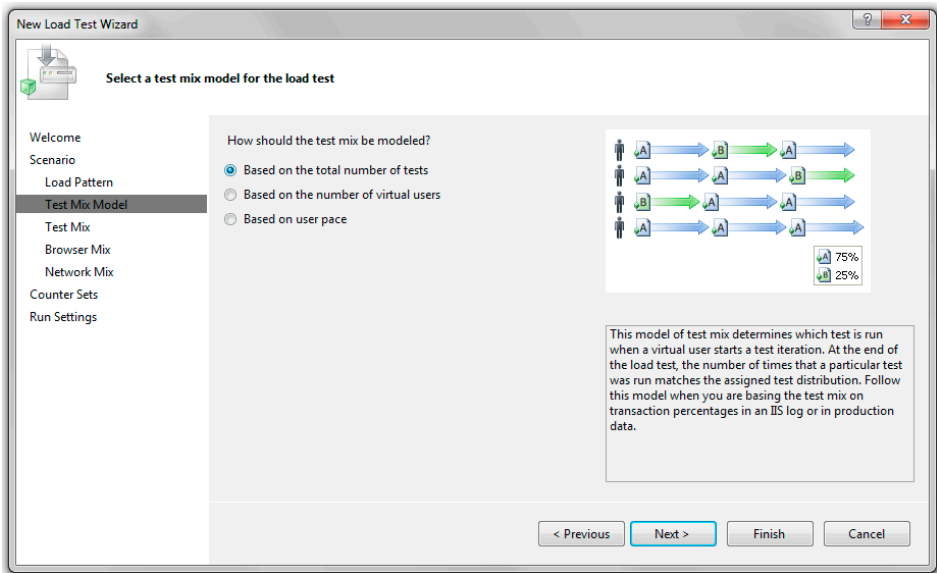


Figure 5.41: New Load Test Wizard – Test Mix Model.

On the next screen (Figure 5.42) you can select the tests that the load test will execute (known as the **Test Mix**), and assign percentage weightings to those tests to determine the ratio of their execution.

Basically, this means that a test with a weighting of 50% will execute twice as many times as a test with a weighting of 25%. Selecting the **Add** button will display a dialog where tests can be selected (or deselected) to participate in this load test using the arrow buttons between the two panes of the dialog.

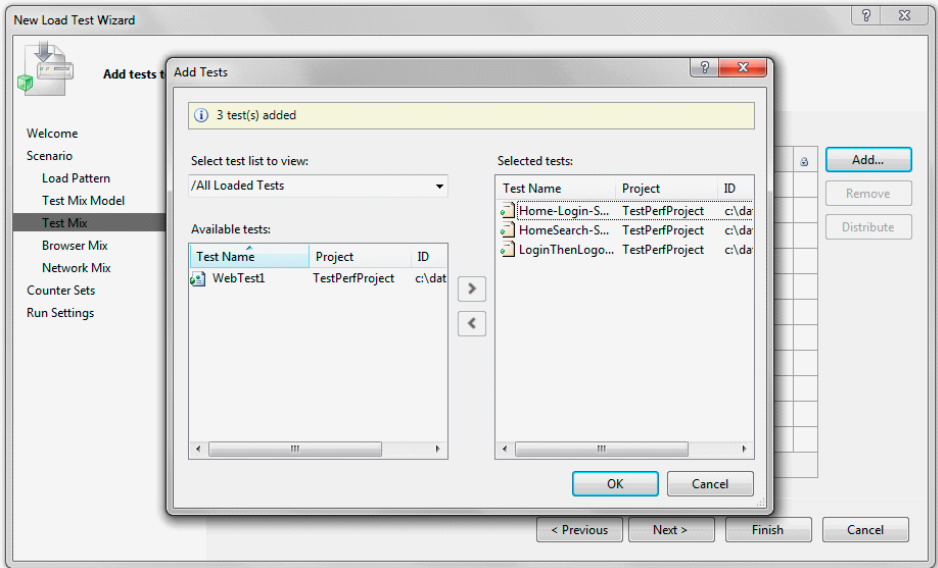


Figure 5.42: New Load Test Wizard – tests added.

Obviously, in order to run a performance or load test, there will need to be at least some tests selected to execute.

Once the tests are selected, clicking **OK** will show the tests added to the load test scenario with a default distribution (Figure 5.43). The distribution of the tests can be set to the required ratio required by the functional paths and use cases decided by the business, and they can also be easily modified later.

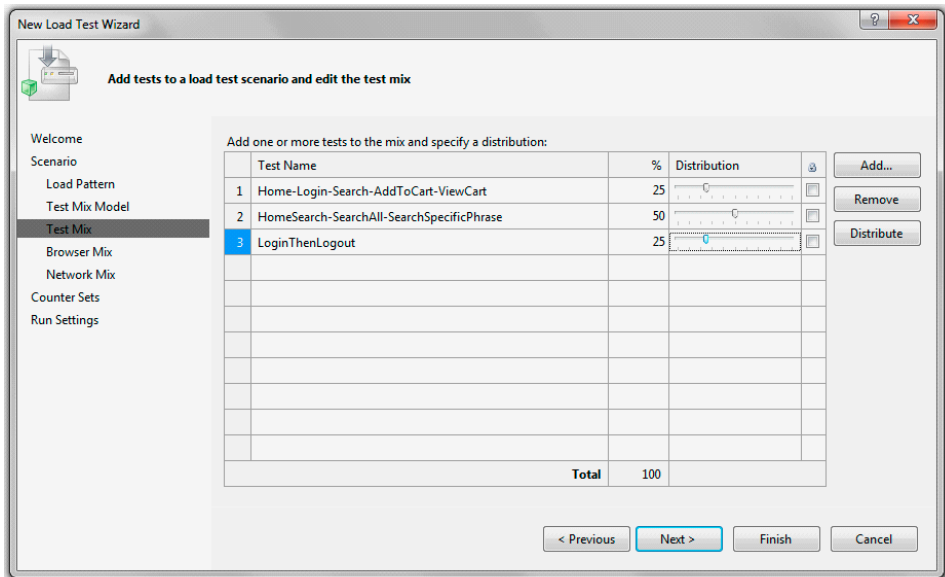


Figure 5.43: New Load Test Wizard – Test Mix defined.

At this point, you can click the **Finish** button to add the load test to the project with defaults attributed to the rest of the options. However, it's best to at least go through and confirm the default settings are appropriate for your needs.

The next step (Figure 5.44) involves adding browsers to simulate during the load test. If this is not a concern, then simply skip to the next step. Otherwise, the dialog will allow you to add a simulated browser by selecting the **Add** button and then distribute the weighting of browsers in the same way as was done for test weightings.

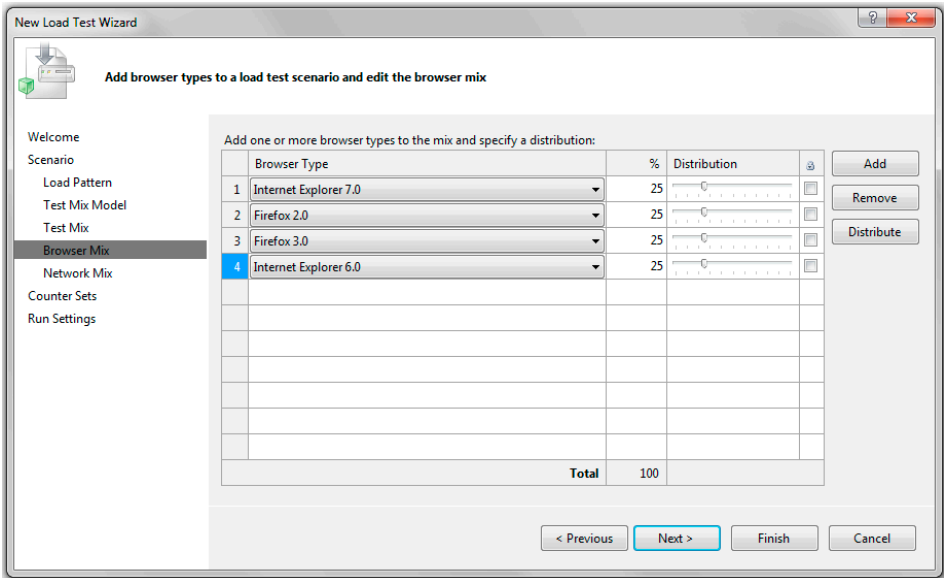


Figure 5.44: New Load Test Wizard – Browser Mix.

The next step (shown in Figure 5.45) involves adding simulated networks to the mix, with the now familiar weighting process. For most scenarios, leaving a single selection of LAN is best, as this will not reduce any of the simulated traffic to the server.

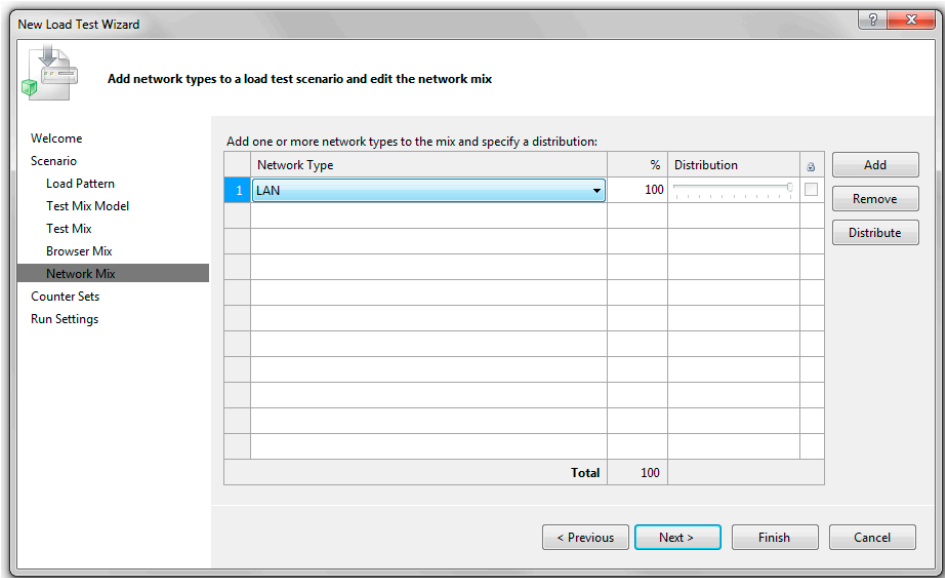


Figure 5.45: New Load Test Wizard – Network Mix.

Initially at least, this is best left at the default setting, with some tweaking performed at later stages when a good idea of raw performance is obtained.

The next, penultimate step (Figure 5.46) involves defining additional computers and associated counter sets for Visual Studio to monitor and collect during the performance test. For now, we can accept the defaults Visual Studio provides, as these cover the main metric points, such as the CPU, and memory for the test servers, load agents, and controllers.

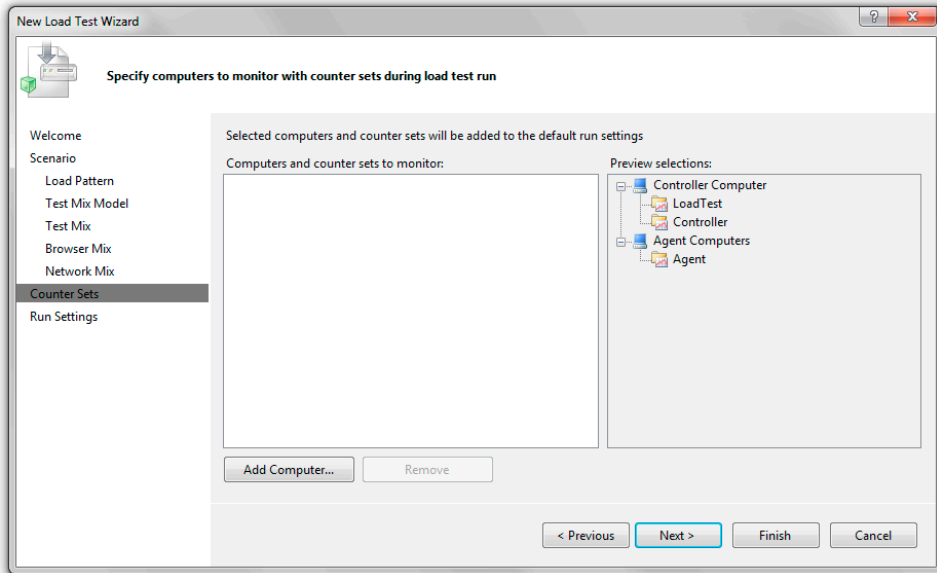


Figure 5.46: New Load Test Wizard – Counter Sets.

The final step in the process involves defining the run settings (see Figure 5.47). These are comprised of things like the time period for the performance test to run, including warm-up time, or the number of test iterations to execute. Additionally, the sample rate and the validation level are also specified. The validation level determines whether low validation rules, low and medium ones, or all validation rules are executed against the tests. A validation level of Low indicates only low validation rules are executed, whereas a validation level of **High** indicates all validation rules are executed. Initially, leave the run duration at a default, low time period of ten minutes, as this will allow us to perform a series of small tests to ensure everything is working as expected.

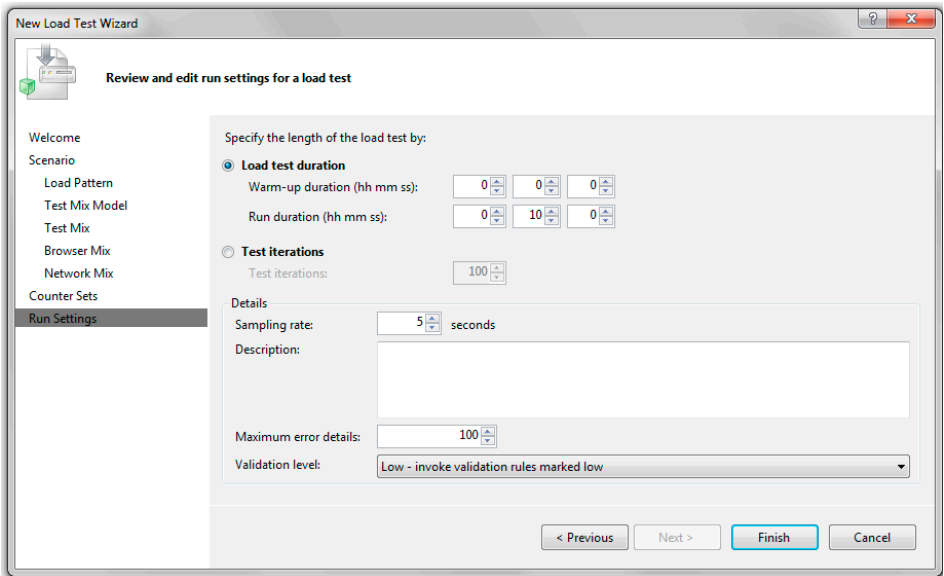


Figure 5.47: New Load Test Wizard – Run Settings.

Click on **Finish**, and the load test has now been added to the project.

Items can be selected in the main window and their properties edited using the property window, in the same way as any object in the test project.

To run the load test, the controller needs to be running as an absolute minimum. Agents are not strictly required to run the tests at low load levels, and are mainly used to distribute test generation to simulate high load levels.

With the controller running, the **Run Test** button can be pressed (Figure 5.49) and the load test will commence running, based on the run settings. If the defaults are used, then this will be for ten minutes.

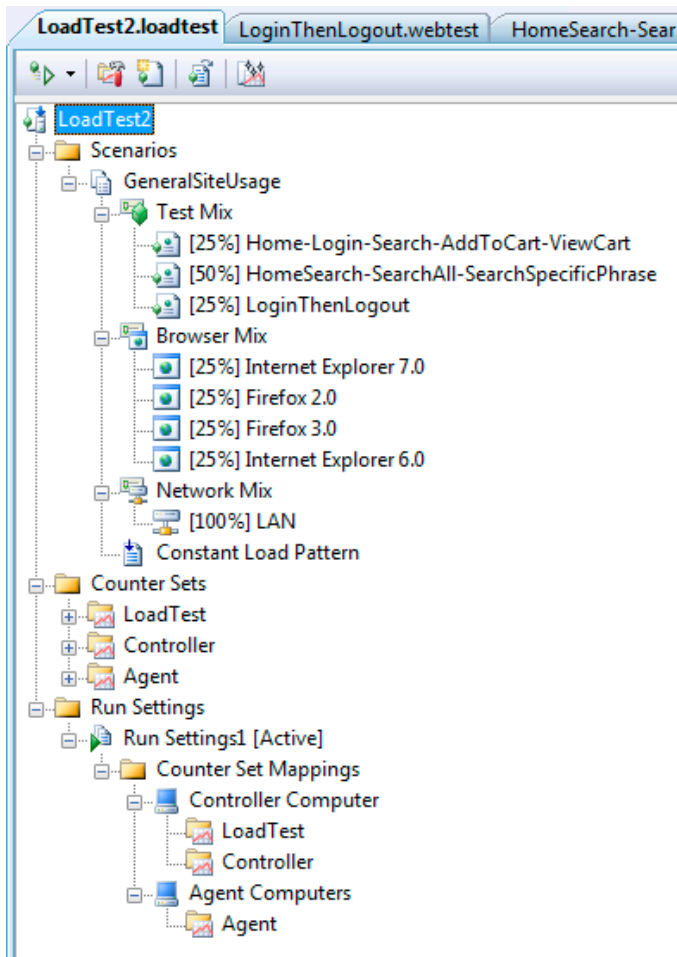


Figure 5.48: Load Test in the project.

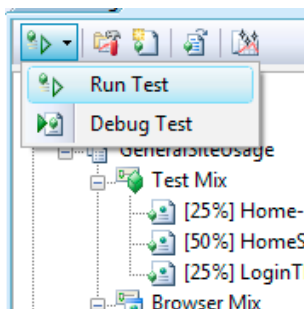


Figure 5.49: Load Test – Run Test button.

When the load test is executing, a series of windows will be presented which show the current progress of the load test, together with some key counters and measurements. The display should look similar to that in Figure 5.50.

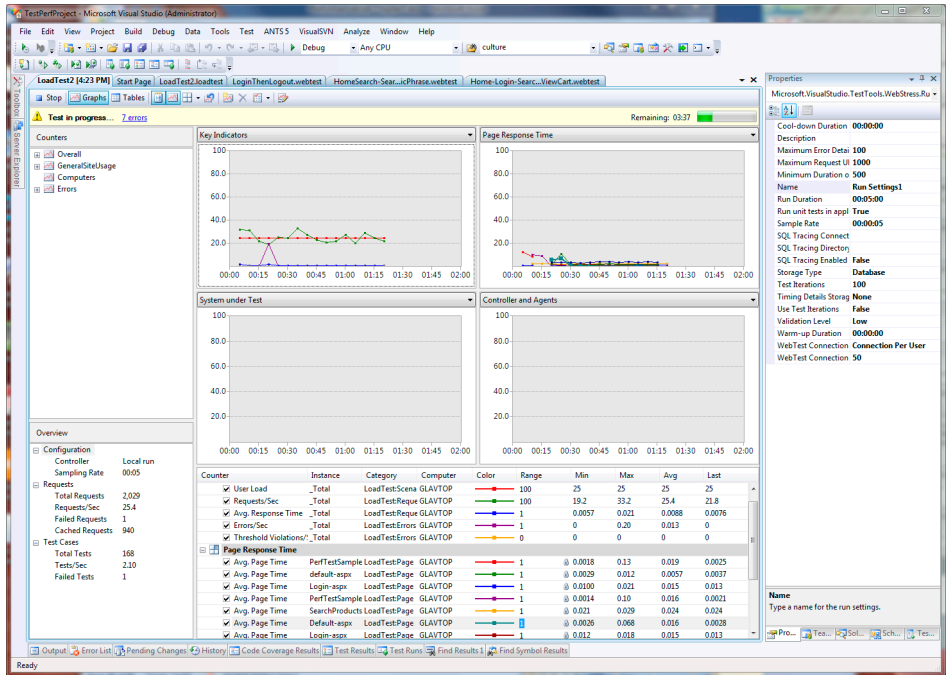


Figure 5.50: Load test currently executing.

While the test is executing, additional performance metrics can be viewed by simply expanding and locating the desired metric in the left-hand **counters** pane, and then either double-clicking or dragging the metric into the desired display window.

Once the load test has completed executing, a dialog is shown (Figure 5.51), asking if you'd like to view the detailed results.

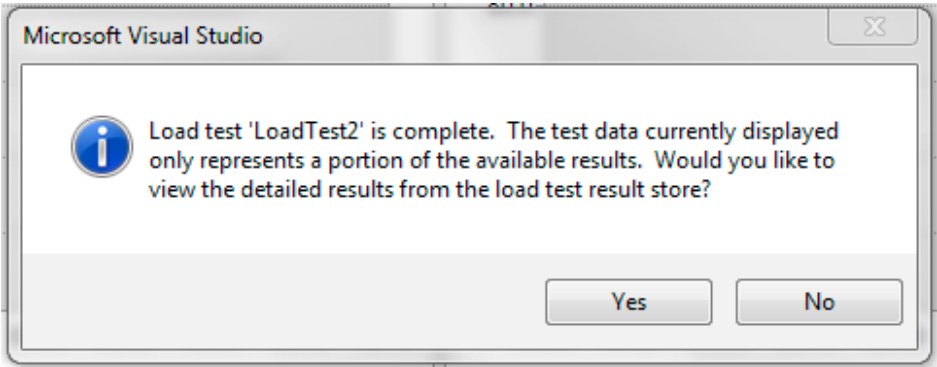


Figure 5.51: Load Test Complete dialog.

The dialog allows the user to load in all the performance metric data for evaluation if desired, as only a small portion is calculated and presented during the execution of the test. To get a full view of the metric data, more data must be loaded and analyzed.

Putting automation in place

Executing a load test and collecting results is a relatively trivial task when all the infrastructure is in place. Ideally, automating this manual task can allow tests to be run without manual intervention and/or during non-business hours. Cleaning up after a test run is another manual task that is a good candidate for automation.

All of this automation allows effort to be concentrated on the areas that provide value, such as result analysis, rather than on manual, mundane tasks like initiating execution and clean up. Furthermore, the cost of performance testing is substantially reduced as a result of automating as much of the process as possible.

Executing the load test

Automating the execution of load tests is actually quite easy. Typically, a load test would be scheduled to run overnight, over a weekend, or even over the course of a few weeks. For these scenarios, it's useful to have the performance test execute without human interaction, so that regular performance runs can be performed and analyzed.

The easiest way to accomplish this is to simply have a batch file that executes the load test, and to use the NT Scheduler that comes with all current versions of windows to schedule when this batch file is executed.

The batch file itself only needs to execute the standard Microsoft test application, MSTest executable (**mstest.exe**), and pass in arguments to allow MSTest to properly execute the tests. An example load test start script may look something like this:

```
"C:\Program Files\Microsoft Visual Studio 9.0\Common7\IDE\mstest.exe" /TestContainer:ProdLoadTest.loadtest /RunConfig:"C:\Source Code\LoadTest\PerfTestRun.testrunconfig"
```

In this example, the arguments below are passed to MSTest.

- **TestContainer**
Represents the load test that should be executed. This contains all the details, such as Network Mix, run details, and so on, required to execute the test.
- **RunConfig**
Represents the current test configuration which lists the controller to use, deployment options, test naming standards, etc., discussed earlier.

That is sufficient to start the load test executing. The Scheduler can be located in the Administrative Tools section of the Windows Control Panel.

Collecting performance monitor data

Previously, the Perfmon tool was discussed as a great way of collecting server performance metrics to validate and, indeed, of backing up the metric data collected by Visual Studio Team Test. However, if the load test execution is being automated, it makes sense that the beginning and end of the collection of performance data via Perfmon should also be automated.

The Perfmon tool already contains functionality to schedule starting and stopping the collection of performance data. Each version of Perfmon has a slightly different user interface, although the overall functionality is the same.

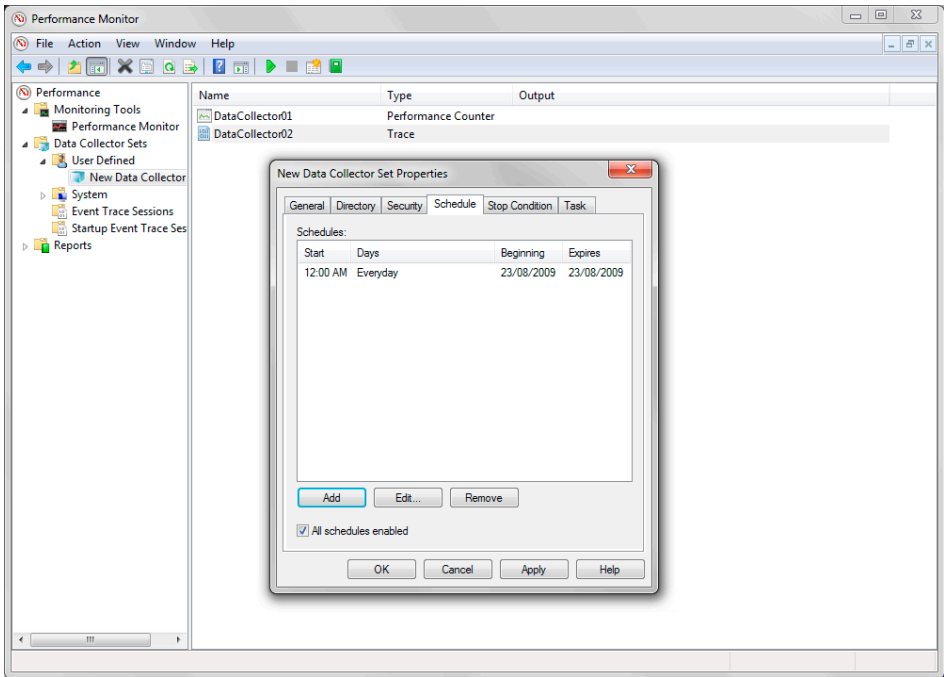


Figure 5.52: Windows 7 / Server 2008 Performance Monitor – Schedule dialog.

In both versions of Perfmon, you'll need to create a new counter set, select the properties of that counter set and then select the appropriate scheduling options. Alternatively, select the **Data Collector** set and then select its properties. From the displayed dialog, select the **Schedule** tab to add, edit or remove a schedule, and define when it should start collecting, as shown in Figure 5.53.

In Windows XP, Server 2000, and Server 2003, the terminology is a little different, but the concepts are the same. Selecting the properties of an existing counter log, then selecting the **Schedule** tab, allows a user to define the start and stop times of the counter recording.

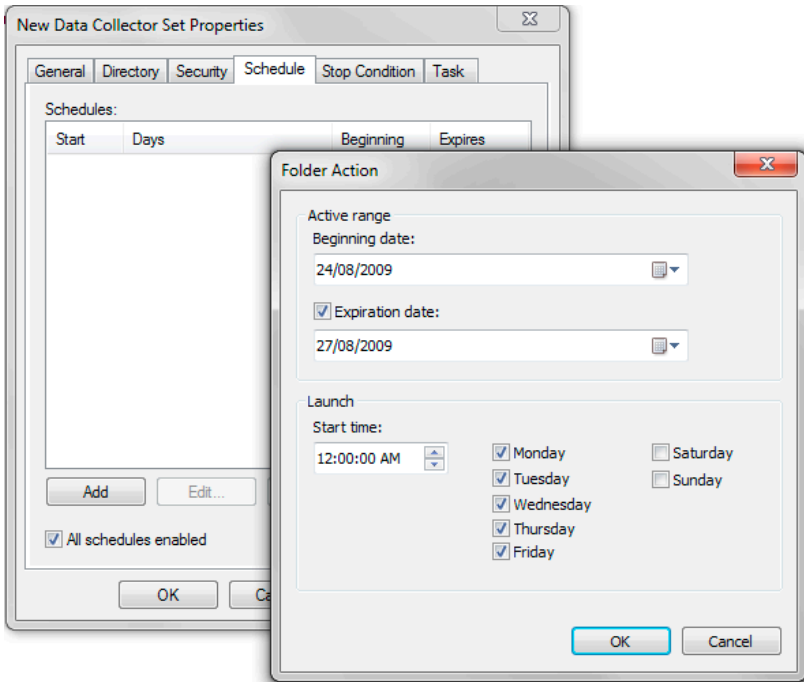


Figure 5.53: Windows Vista, Windows 7, Server 2008 Perfmon Schedule dialog.

Additionally, in more recent versions of Windows and Windows Server, selecting the **Stop Condition** tab provides options as to when the data collection should terminate.

When scheduling performance monitor collection and termination, it is best to generally start the collection a short period before the performance run commences, and then to terminate it a short period after the performance run has ended. This ensures that all data for the performance run is collected, but it will also clearly indicate what the idle or quiet times of activity on the system look like from a performance metric point of view.

It is important to note that, under Windows XP, Server 2000 and Server 2003, the scheduling features of performance monitor were somewhat unreliable. Occasionally, when scheduled to start, they simply wouldn't, and I have yet to find a good reason for this. Terminating collection has never seemed to be an issue. To prepare against this eventuality, you can use a command-line tool called **logman** to instruct PerfMon to start collection, and let the scheduling definition in PerfMon terminate the collection. **logman** is available on all Windows server operating systems, and to use it to start a PerfMon collection, use the syntax:

```
logman start MyPerfCounters
```

where *MyPerfCounters* represents the counter log set name. This tool can also be used to stop counter collection.

Collecting SQL Server usage statistics

Since the database plays such an important part in today's applications, it is useful to determine, not only how hard the database server is working, but ideally what are the most expensive queries the database is running. This kind of information can be extremely useful in determining how the database is performing, and how the application is making use of the database.

SQL Server provides a convenient way to schedule the execution of scripts via the SQL Server agent. Other database vendors offer similar methods. However, with SQL Server it is possible to define scheduled jobs in SQL Management studio that execute scripts at specific times and output the result to a file or database table. There are, of course, many ways to schedule such an activity, but SQL Server Agent (managed via SQL Management Studio) provides one of the most convenient.

In order to determine what the most expensive queries are, the system tables in SQL Server need to be queried, as this is where SQL Server records its statistical data. An example SQL script to retrieve the most expensive queries would be this:

```
set nocount on;

select
    qs.total_worker_time,
    qs.execution_count,
    SUBSTRING(st.text, (qs.statement_start_offset/2)+1,
        ((CASE qs.statement_end_offset
            WHEN -1 THEN DATALENGTH(st.text)
            ELSE qs.statement_end_offset
            END - qs.statement_start_offset)/2) + 1
    ) AS statement_text
from
    (select top 100
        qs.plan_handle,
        qs.total_worker_time,
        qs.execution_count,
        qs.statement_start_offset,
        qs.statement_end_offset
    from
        sys.dm_exec_query_stats qs
    order by qs.total_worker_time desc) as qs
    cross apply sys.dm_exec_sql_text(plan_handle) as st
order by qs.total_worker_time desc;
```

This query will return results that look similar to those in Figure 5.54.

	total_worker_time	execution_count	statement_text
1	2411138	15812	INSERT INTO LoadTestPerformanceCounterSample (LoadTestRunId, T...
2	2134122	3	SELECT 'Server[@Name=' + quotename(CAST(serverproperty(N'Serve
3	1402079	2435	SELECT [t0].[Id], [t0].[ShortDescription], [t0].[LongDescription], [t0].[Category
4	1396079	114	INSERT INTO [LoadTestPerformanceCounter]([LoadTestRunId],[CounterCat
5	1374078	1	SELECT SCHEMA_NAME(udf.schema_id) AS [Schema], udf.name AS [Nam
6	904051	1	insert into LoadTestRun (LoadTestName,RunId,Description,Comment,IsLoca
7	804045	600	INSERT INTO [LoadTestPerformanceCounterInstance]([LoadTestRunId],[Cc
8	774044	3	SELECT se.is_admin_endpoint AS N'AdminConnection', (SELEC
9	701039	1	SFI FCT SCHEMA_NAME(udf.schema_id) AS [Schema] udf.name AS INam

Figure 5.54: SQL Server most expensive query results.

The results of the query show the total worker time, execution count, and even the actual text of the query. The first two results are related to the performance tests and SQL server respectively, so these are not of immediate concern. The highlighted row (Row 3) represents a query from the sample application, and Rows 12 and 13 also represent queries generated from the application. This is determined by looking at the tables being used in the query text. Given the relatively few times that these queries are executed, the fact that they are appearing near the top of the result list may indicate an opportunity for optimization.

Results of this nature, when used in conjunction with performance tests, can quickly show less than optimal parts of the application from a database perspective. This is in contrast to the top-down perspective that Visual Studio adopts in reporting performance metrics. Using these two techniques can provide enormous insight into potential performance gains for your application.

Ideally, these scripts should be used after a performance run and then used comparatively as further runs are executed, in order to ensure that any applied performance changes are effective.

Another useful script to execute is one to determine the general index usage of the target database:

```
set nocount on;

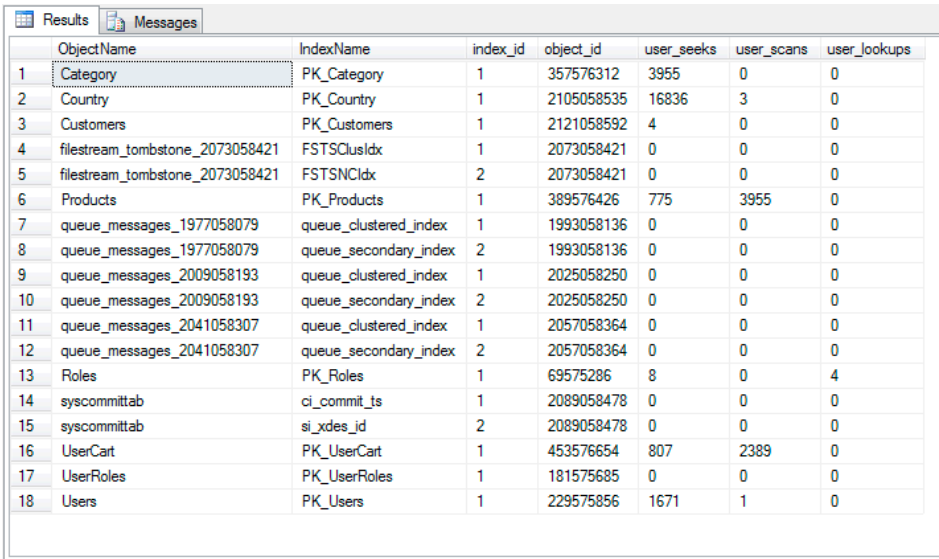
use PerfTestSampleDB;
select
    obj.Name as ObjectName,
    ind.name as IndexName,
    ind.index_id,
    ind.object_id,
    isnull(user_seeks, 0) as user_seeks,
    isnull(user_scans, 0) as user_scans,
    isnull(user_lookups, 0) as user_lookups
```

```
from sys.indexes ind
    join sys.objects obj on (ind.object_id = obj.object_id)
    left join sys.dm_db_index_usage_stats st on (st.index_id =
ind.index_id and st.object_id = ind.object_id)
where obj.Type_Desc <> 'SYSTEM_TABLE'
order by obj.Name, ind.Name;
```

Note

If using this script, replace the **PerfTestSampleDB** database name with the name of the target database to be analyzed.

Executing this script produces an output similar to that in Figure 5.55.



Object Name	Index Name	index_id	object_id	user_seeks	user_scans	user_lookups
1 Category	PK_Category	1	357576312	3955	0	0
2 Country	PK_Country	1	2105058535	16836	3	0
3 Customers	PK_Customers	1	2121058592	4	0	0
4 filestream_tombstone_2073058421	FSTSCLusIdx	1	2073058421	0	0	0
5 filestream_tombstone_2073058421	FSTSNCLdx	2	2073058421	0	0	0
6 Products	PK_Products	1	389576426	775	3955	0
7 queue_messages_1977058079	queue_clustered_index	1	1993058136	0	0	0
8 queue_messages_1977058079	queue_secondary_index	2	1993058136	0	0	0
9 queue_messages_2009058193	queue_clustered_index	1	2025058250	0	0	0
10 queue_messages_2009058193	queue_secondary_index	2	2025058250	0	0	0
11 queue_messages_2041058307	queue_clustered_index	1	2057058364	0	0	0
12 queue_messages_2041058307	queue_secondary_index	2	2057058364	0	0	0
13 Roles	PK_Roles	1	69575286	8	0	4
14 syscommittab	ci_commit_ts	1	2089058478	0	0	0
15 syscommittab	si_xdes_id	2	2089058478	0	0	0
16 UserCart	PK_UserCart	1	453576654	807	2389	0
17 UserRoles	PK_UserRoles	1	181575685	0	0	0
18 Users	PK_Users	1	229575856	1671	1	0

Figure 5.55: SQL Server database index usage statistics.

The results of the query show the general usage patterns of indexes within the sample application, with metrics around the number of scans, seeks and lookups for each index. Optimization of indexes is a great way to reduce query execution times, and the metrics provided with this query can help to ensure that the index usage is always kept efficient.

Clean up tasks

When a performance run is executing, sometimes the test agents can experience low memory conditions, high CPU utilization and other resource issues. This is particularly likely at very high loads where the expectations placed on the test agents were not in line with their specifications. This usually happens early in the performance test phase, when the exact characteristics of test agents are only estimated.

Additionally, be aware that, after a performance test run has completed execution, even with the best intention and attempts by software, some memory may not be properly released. Eventually this can lead to unstable test agents and, ultimately, failed or less-than-effective performance test runs.

For this reason, I recommend that you restart or reboot the test agent machines after each run. Thankfully, this can also be automated, using the included scheduling ability of Windows as described previously. We need to instruct each of the agent machines to restart, and this can be done using a command-line tool called **Shutdown** which is available on all versions of Windows from Windows XP to Server 2008. To restart a remote machine, for example, the following syntax is used:

```
shutdown -r -t 0 -f -m \\testagent
```

Where:

- **-r** : instructs the machine to restart, as opposed to simply shut down
- **-t 0** : instructs the shutdown/restart process to happen after a timeout of 0 seconds, that is, immediately.
- **-f** : instructs the machine to force all running applications to close without prompting any warning.
- **-m \\testagent**: represents the machine to shut down/restart. In this case, the machine is named *testagent*.

Typically a batch file is created that restarts all agent machines that you may have in your performance test rig. This script could either be called directly after the scheduled performance test execution script, or it could be scheduled to execute at regular times during the day when it is known that the performance run has completed.

This way, the test agent machines are always in a known, clean state before each performance test run, which better ensures a successful performance test.

Note

This technique could also be applied to the controller if desired.

Conclusion

This chapter has provided an extensive walk-through of the following aspects of performance testing:

- architecture of a test rig
- setting up the various components of the performance test rig such as controller and agents
- troubleshooting test rig setup
- creating web tests and load tests, and parameterization of the tests
- automation of the execution of load tests, collection of performance data, and clean-up tasks.

The amount of setup and effort required to have a performance test rig running and automated is not small, and it is therefore important to have a good understanding of how a performance test rig operates and, more importantly, how to debug and diagnose it if, or when, errors occur.

Once the rig is running and automated, the cost of performance testing then just comes down to analyzing the results and any subsequent changes this may generate. This is exactly the situation you need to be in to effectively and continuously monitor your applications for performance issues. More importantly, this will allow constant metrics to be fed back to interested parties to ensure application development is going according to plan.

Chapter 6: Application Profiling

If you talk to teams of developers about performance profiling, someone will usually say something like, "We don't have time to profile our code, that's why we have load testers" or "If it runs slowly we just throw another server into the farm." Many developers see performance profiling as an extra piece of work to add to their existing workload, and yet another steep learning curve to climb.

Many developers enter the world of performance and memory profiling only when something has gone badly wrong. This usually means during system testing, load testing, and often (sadly) in production. Developers will download an evaluation copy of a profiler and try to isolate why the application is running slowly or keeps crashing. The pressure is on, and it's now the worst possible time to learn the skills required to be an effective application profiler.

Using profiling tools to look for potential bottlenecks *during* development can significantly reduce the number of problems that show up later. With the right tools and training, it can become a regular part of the development process without adding too much overhead.

Development profiling will never uncover all of the issues that a comprehensive load test would, but it can highlight parts of the code that have the potential to become a bottleneck when the application is stressed. Finding and fixing them early can make a big difference overall, especially if all the developers are testing the code they write.

This chapter, and the next two, are all about the tools and techniques that you can quickly master and then use as part of your development process. Remember, it costs between 15 and 75 times more to find and fix an issue found during test than if that same issue was found during development (Boehm, 1981).

Types of profiling

Application profiling goes beyond the raw performance statistics obtained from system performance monitoring tools, and looks directly at the functions and allocated objects inside the executing application.

When profiling a .NET application, the execution speeds of its internal functions and the resources they use are recorded for a specific set of test transactions. The recorded data will give insight into where there may be performance bottlenecks and possible memory problems (such as memory leaks).

Profilers retrieve performance and memory information from .NET applications in one of three ways:

- **Sample based**
The application function call stack is periodically recorded to give a low overhead but equally low resolution analysis.
- **Events based**
The Common Language Runtime can be configured to send notifications to specific profiler DLLs. Key information on function execution, CPU, memory, and garbage collection can be collected using this mechanism.
- **Instrumentation**
Instrumentation code that measures the application is added to it at runtime, which can give very detailed and accurate results,, but also comes with a high overhead.

A word about profiler overhead

Whichever profiler you use will add some overhead to the executing application it's measuring, and to the machine it is running on. The amount of overhead depends on the type of profiler.

In the case of a performance profiler, the act of measurement may itself impact the performance being measured. This is particularly true for an instrumenting profiler, which has to modify the application binary to insert its own timing probes to every function. As a result, there is more code to execute, requiring additional CPU and memory, causing increased overhead. Most profilers try to compensate by deducting the overhead of the instrumentation from the results.

The profiler also has to deal with the torrent of data it receives and, for a detailed analysis, it may require a lot of memory and processor time just to cope.

If your application is already memory and processor intensive, things are unfortunately only going to get worse, and it could be that it's just not possible to analyse the entire application. Thankfully, most tools allow you to limit the scope and depth of the analysis, which can help. In some situations, the only way to get results may be by writing test harnesses to exercise portions of the application in ways analogous to the full application.

Performance profiling

Performance profiling is all about discovering which parts of your application consume a disproportionate amount of time or system resource. For example, if a single function takes up 80% of the execution time, it's usually worth investigating.

Profiling will highlight small areas of code that would never otherwise be looked at again, and it makes developers ask some interesting questions. To be fair, most of the time the answer will be, "It has to do that and that's as fast as we can make it." The rest of the time, a potential bottleneck will have been uncovered.

What to profile

Profiling a multilayered networked application can be really difficult simply because of the number of possible variables involved. The question that's difficult to answer is, "Which bit is slow?" Is it the client, the web server, the application server, the database server, or even one of the network links in between?

The first stage in profiling performance is to identify the "slow bit." Application server monitoring can help isolate the guilty layer, and will often help you determine if it is an application or a database problem. Sometimes the problem is even more complex, and a network monitoring tool will be required. These tools analyze packet journey times between the layers, and break down application transactions into server processing time and network time. They can help identify the layer responsible for slow-down, and determine if the problem is to do with a network issue, such as congestion, bandwidth or latency. Chapter 7 discusses this topic in more detail.

Once you have identified the layer responsible (or, if you like, the slow bit) that will give a clue as to the kind of profiler to use. Obviously, if it's a database problem, then use one of the profiling tools available for the products of the database vendor, or simply add another index (just kidding!). If it's a .NET application problem, then there are a whole host of profilers available, and we will be discussing some of the best ones later in this chapter, and when we look at more specific types of profiling later on.

Function analysis

To measure the performance of an application, you need to know how long specific test transactions take to execute. You then need to be able to break those results down in a number of ways. Specifically, function call and function call tree (the sequence of calls created when one function calls another, and so on).

This breakdown identifies the slowest function and the slowest execution path, which is useful because a single function could be slow, or a set of functions *called together* could be slow. Many tools create elaborate hierarchical diagrams which allow the developer to explore the call trees, and this can really help when trying to identify a bottleneck.

Line level analysis

Profilers can accurately time individual code lines, allowing you to identify the slowest line within a slow function. For me, this is an essential feature because it gives you the best chance of coming up with workable optimizations.

However, line-level analysis does add a greater overhead to the profiling session and can normally be switched off, leaving the profiler to analyze at the function level only.

Wall-clock (elapsed) vs. CPU time

Most profilers measure wall-clock time and CPU time. The ability to distinguish between the two is important because CPU time is pure processing and excludes any waiting time. By contrast, wall-clock time is the total time taken to process a function, including any wait time.

A function may take a long time to execute, but use comparatively little CPU time because it is actually waiting for a database / web service call to return or for a thread synchronization lock to free up. Identifying wait time can help you identify where your application may benefit from asynchronous processing.

At the same time, a CPU-intensive function is usually a good candidate for optimization, because the CPU is a finite resource and a potential bottleneck.

Resource bottlenecks

Resources such as disk space, network bandwidth, server availability, graphics cards and shared threads can all create bottlenecks in an application. Identifying functions causing high levels of resource activity and contention is a key goal in profiling. This kind of activity, when scaled, could quickly become a problem and reduce the scalability of the application.

Call count

Function call count is the easiest statistic to look at first, because a non-trivial function with a high call count often indicates an immediate problem. It's *always* worth validating the origins of the high call count.

Small optimizations add up and scale

The great thing about performance profiling an application *during* development is that a developer can immediately see where the main processing hotspots/bottlenecks in the code are. Optimizing the hotspots and asking intelligent questions about call counts can give small but significant improvements in performance and, if the whole team adopts this strategy, the gain can be significant.

With so much code executing on servers, small performance gains become significant because they quickly scale according to the number of users and the number of locations they affect. More to the point, identifying and eliminating potential bottlenecks will prevent them from ever becoming problems during load testing or in production.

Memory profiling

The way you write your code directly impacts how and when the objects you create are allocated and destroyed. Get it right, and your application will use memory efficiently as needed, with minimal performance impact. Get it wrong, however, and your application could use more memory than necessary, which will cause the memory manager to work harder than it needs to, and this will directly impact performance.

Even worse than that, your application could just keep allocating memory until no more is left, causing the application or the machine to crash. This is the Memory Leak which every developer fears.

The good news is that there are plenty of tools out there which you can use to find and fix memory problems before they actually become problems. All you need is some background knowledge and a few basic techniques, and it will become second nature.

Checking that an application doesn't have memory leaks and efficiently uses memory, together with fixing any issues found, will improve its overall stability and performance.

Garbage collection

The .NET memory management model ensures that any allocated objects which are no longer in use by the application will be reclaimed automatically. This relieves developers of the responsibility of having to free memory explicitly, which is something that was often omitted in native C/C++ applications, leading to memory leaks.

Garbage collection was invented by John McCarthy et al. in 1959 as part of the Lisp language, but gained most prominence when it was adopted as the memory management model for Java in 1995.

Instead of depending on the developer to manually de-allocate objects, garbage collection adopts an automatic model in which objects are monitored to determine if they are still in use. Those no longer used will have their memory reclaimed automatically. The automatic memory management model, of which garbage collection is a part, was adopted by Microsoft as the model for .NET. I will cover .NET's memory management model and how it works in detail in Chapter 7 but, for now, here is a brief overview.

The .NET CLR allocates objects (less than 85K) onto a managed memory heap, and ensures they are placed consecutively in memory with no gaps in between objects. The garbage collector then periodically determines which objects are still in use by looking to see if they are referenced by other objects, or from the stack, globals, statics, or even CPU registers. If no references are found, it concludes that the object isn't in use and can be "garbage collected."

When an object is garbage collected, it is simply overwritten by the objects above which are moved down in memory – a process known as compaction. This makes sure there are no gaps left in the heap. In truth, it's actually a bit more complicated than this, as objects are grouped into generations depending on how recently they were allocated. (For performance reasons the garbage collector always tries to collect the youngest objects first.)

Anything that keeps hold of a reference to an object will keep it alive indefinitely, and that can be the cause of a leak if it repeats continually. Memory profiling is all about finding suspiciously persistent objects, and tracing back to find the references in code that are keeping them in memory.

Using memory profiling techniques and tools, you can identify large objects that cause the application to have a larger memory footprint than necessary. You can also look for objects that are continually created and never garbage collected, causing memory leaks. I'll cover the garbage collector and associated concepts in much more detail in Chapter 8.

Profiler approaches

All memory profilers will track instances of allocated classes. Some will also track the allocation call stack, which means that they can report on a function's allocation profile and identify function "hotspots."

The ability to view allocations, in terms of both classes and functions, can be really useful. However, recording allocation call stacks can be very memory intensive and can sometimes limit the size and complexity of application that can be profiled.

Symptoms of memory problems

- **Memory leak**
 - Memory usage slowly increases over time
 - Performance degrades
 - Application will freeze/crash requiring a restart
 - After restart it's OK again, and the cycle repeats.
- **Excessive memory footprint**
 - Application is slow to load
 - After load, other application runs slower than expected.
- **Inefficient allocation**
 - Application performance suddenly degrades and then recovers quickly
 - % Time in GC Statistic in PerfMon is greater than 20–30%.

I will go through memory profiling in a lot more detail in Chapter 7.

When to start profiling

In my view, the answer to this question is profile when you feel you have achieved functional stability in your coding task. That means, after you have completed the requirement and your testing confirms it works as specified.

Profiling at this stage will highlight potential bottlenecks that should be investigated. Profile any earlier and you could be optimizing code that may significantly change.

Profiling usually occurs in one of three ways: reactive debugging, proactive analysis, or technique validation.

Reactive debugging

Reactive debugging happens when a problem has been found, typically during a load test or in a live system, and the developers have to react to this unwelcome news and fix the problem.

With load test debugging, you have a lot more data to work with because the results will describe the failing transactions in fine detail and give many detailed server statistics, which will help in isolating exactly where the problems is.

Production debugging is much more difficult, because really all you will get is some performance monitoring statistics and, if you are lucky, some anecdotal evidence about what might have been running when the slowdown occurred.

If you carry out load testing late in the life cycle, or if it's a production problem, a number of things now have to happen.

Developers have to:

- isolate the tests to run to reproduce the problem
- understand how to use the profiler
- interpret the results
- get familiar with the code again
- fix the code
- confirm the fix.

Production problems are inevitably accompanied by pressure from management to fix the issue. Developers are also usually caught off guard and are ill prepared for an in-depth analysis of a system they may have last worked with many months earlier.

This is all just an inefficient waste of time and resources, and it doesn't even include the time that would then be required for system testing in a production environment.

The earlier you start your application profiling and load testing, the better. Which is why my general recommendations are:

- Test your application transactions under load as soon as you can during development, and test regularly (as soon as you have something testable). Ensure issues are found and fixed early.
- Encourage a culture where developers proactively look for potential bottlenecks in their code using profiling tools (see next section).

You don't have to wait until the load test phase to begin load testing (although often the expense is too great to use these facilities too early). There are lots of tools out there that you can use to place stress/load on your application, and doing this as early as possible will highlight issues that single test profiling won't find. If you can, automate the stress testing and run it regularly, so that any code changes that impact performance are picked up quickly.

Proactive analysis

Proactive analysis, as the term implies, is all about the developer actively looking for performance and memory issues during the development cycle. It has the major advantage of being by far the quickest and cheapest type of analysis, because the developer already knows the code well, and is able to quickly make knowledgeable optimization decisions.

Proactive analysis takes place as part of the developer testing process, and should be an essential requirement before source code is checked back into the repository. It takes 15–75 times longer to fix an issue in development, than if it was found in later testing (Boehm, 1981).

The proactive approach does require an investment in tools and training, but it also results in more highly skilled development teams who are actively looking for problems in applications, and who are empowered with the skills necessary to find and fix these problems when they occur.

Technique validation

Profilers can really help developers choose the optimal algorithm to achieve a specific processing task. Questions such as, "Should I process it all on the server, or on the client in batches?" can be answered quickly and easily by running a few quick tests.

Finding the most efficient technique to process data can also be very difficult without a profiler. Searching online merely opens up a huge debate, and the only way to be sure is to write some test harnesses, and run a profile.

Tools used for profiling

Many of the available profiling tools combine both performance and memory profiling in one package. I will summarize the features of some of the main tools on the market and, in later chapters, I'll describe how to use them to carry out both performance and memory analysis.

CLR profiler

The CLR Profiler is, at first glance, quite a basic memory profiling tool. On closer analysis it's actually extremely powerful once you get the hang of it. Whilst it isn't the most intuitive or easy-to-use profiler you will find, it is certainly very detailed and comprehensive in the information that can be retrieved.

It can profile applications up to and including .NET Framework 3.5, although it only officially supports up to Framework 2.0.

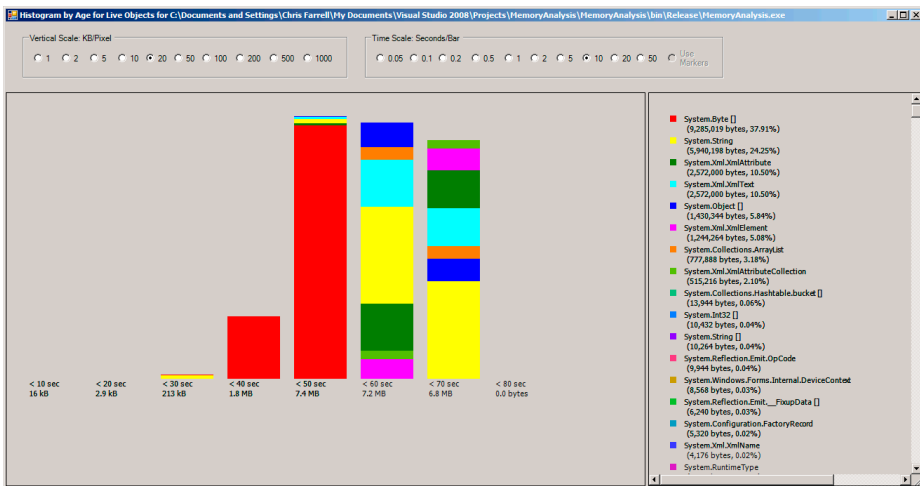


Figure 6.1: CLR Profiler Histogram by age.

CLR Profiler will monitor the executing application (Exes, services and web applications) and then provides a number of histograms and call graphs. These can be used to track memory leaks, excessive memory usage, Large Object Heap issues and excessive garbage collection overhead; it's also possible to analyze Finalizer issues.

Unfortunately, CLR Profiler is one of those tools most developers have downloaded and tried out, but given up on after about twenty minutes because it is quite difficult to use.

It's free to download and, once you have mastered its quirky interface, and adopted a technique that works for you, it's possible to gain real insight into the memory state of the application.

CLR Profiler gets complicated really quickly and for that reason I will cover it in more detail in Chapter 7.

Red Gate's ANTS Memory and Performance Profilers

The Red Gate .NET Developer Bundle v5 works with .NET framework 1.1, 2.0, 3.0, 3.5, and 4.0, integrates into Visual Studio 2005, 2008, and 2010 at March 2010 release, and supports both 32-bit and 64-bit profiling.

The .NET Developer Bundle includes ANTS Performance Profiler Pro and ANTS Memory Profiler and, at the time of writing, costs \$795 for a single user license (*Red Gate, 2010*).

ANTS Memory Profiler

ANTS Memory Profiler (Figure 6.2) captures class instance allocation and has a low overall overhead. It provides an easy-to-use and flexible user interface.

ANTS also provides graphical insight into the heap with the memory timeline, which is a graphical representation of various performance counters including bytes on all heaps, private bytes and Large Object Heap size (other counters can be added in the options). The primary technique for using this tool involves the developer taking memory snapshots at relevant times. Snapshots can then be compared against each other and used to find classes requiring further investigation.

Filters allow the developer to filter out application noise and to focus on specific problems. Application noise refers to any object allocations that are irrelevant to our analysis but whose presence on the heap we may misinterpret. There are standard filters to eliminate general application noise, and specific filters that can be used to find common causes of memory leaks.

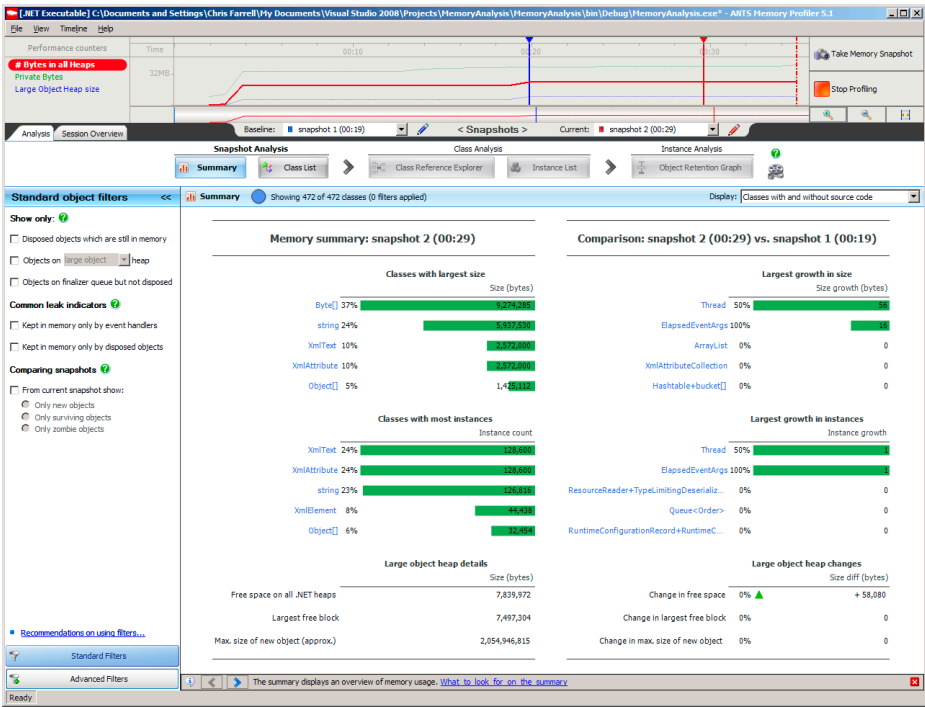


Figure 6.2: ANTS Memory Profiler.

Once a set of suspicious class instances has been identified, the Class Reference Explorer allows the developer to trace back into the tree of object references to find the exact references in the code which are causing the problem.

It's also possible to view a session overview of the snapshot, which gives insight into the state of both the Small Object Heap (including Gen 1 and 2) and the Large Object Heap.

ANTS Performance Profiler

When performance profiling an application, ANTS Performance Profiler (Figure 6.3) presents a performance graph with percentage processor time, plus a number of other performance counters which can be selected.

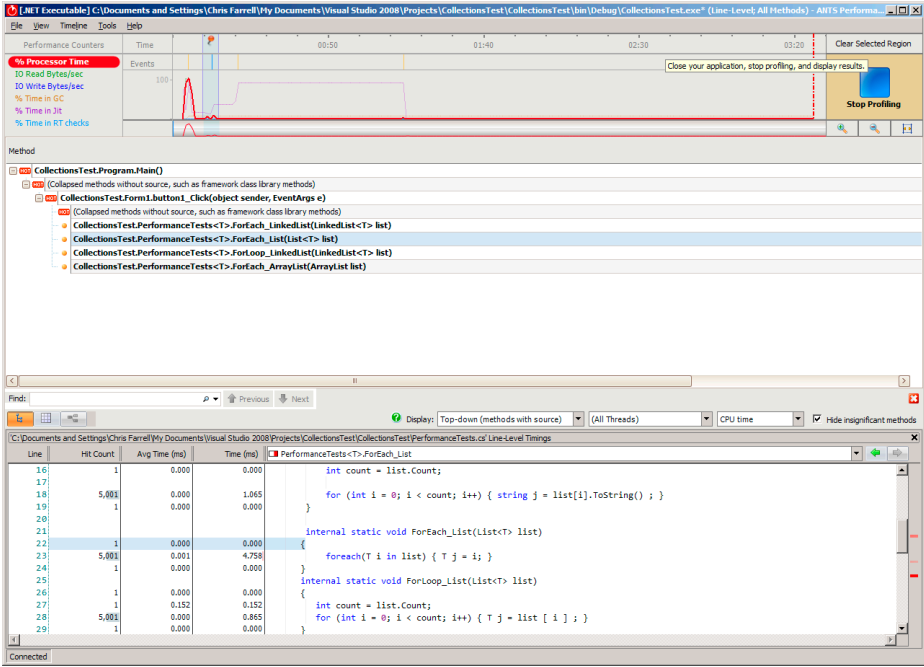


Figure 6.3: ANTS Performance Profiler.

Results can be viewed for the entire analysis or for just a small portion using the trace graph and selecting an area of interest using the mouse (see Figure 6.4).

This can be really useful if you notice part of your trace for example with high CPU activity, and it allows you to focus on what was happening just for that tightly defined period. The profile results for the trace can be viewed in Call Tree, Grid or Call graph modes.

The Call Tree displays a hierarchical list of the slowest call trees for each execution path in the selected period, and highlights as the "hottest" the path that is most likely to be the bottleneck.

The grid displays the results in a classic grid format, giving:

- Time (CPU or wall clock)
- Time with children (CPU or wall clock)
- Hit count (number of times called).

A call graph can also be generated for every function, allowing the sequence of calls to and from a function to be traced.

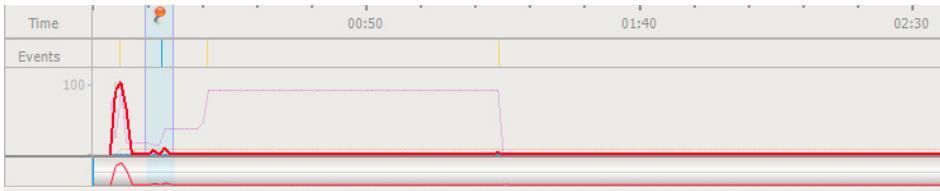


Figure 6.4: Selecting a portion of the analysis trace.

Performance results can be viewed by CPU time or wall-clock time, which is a useful feature as it can help to quickly identify where the application may benefit from asynchronous processing.

ANTS also provides a number of analysis modes which change the amount of overhead that is added to the executing application:

- method level – lower overhead but less detailed
- line level – higher overhead, more detailed.

It is also possible to further reduce the overhead of both modes by selecting to profile only methods that have source code. That is often the most sensible course of action, since you can only optimize where you have source code to change.

Microfocus DevPartner Studio Professional 9.1

MicroFocus's DevPartner 9 is a suite of tools for .NET framework 2.0, 3.0, and 3.5, and Visual Studio 2005 and 2008. DevPartner pricing depends on the licensing model, but if you are buying from ComponentSource a single-user standalone license is \$2834.67 at the time of writing (*ComponentSource.com, 2010*).

The suite is a developer tool that integrates into Visual Studio 2005 and 2008, and can optionally also be run from the command line. It supports 32-bit profiling on both x86 and x64 systems, and includes a range of tools.

Memory Profiler

The Memory Profiler (Figure 6.5) can perform three types of analysis:

- RAM footprint analysis
- memory leak detection
- temporary object analysis.

DevPartner captures both the class instance allocation and the allocation call stack, so it's possible to view the results in terms of class hotspots *and* function hotspots.

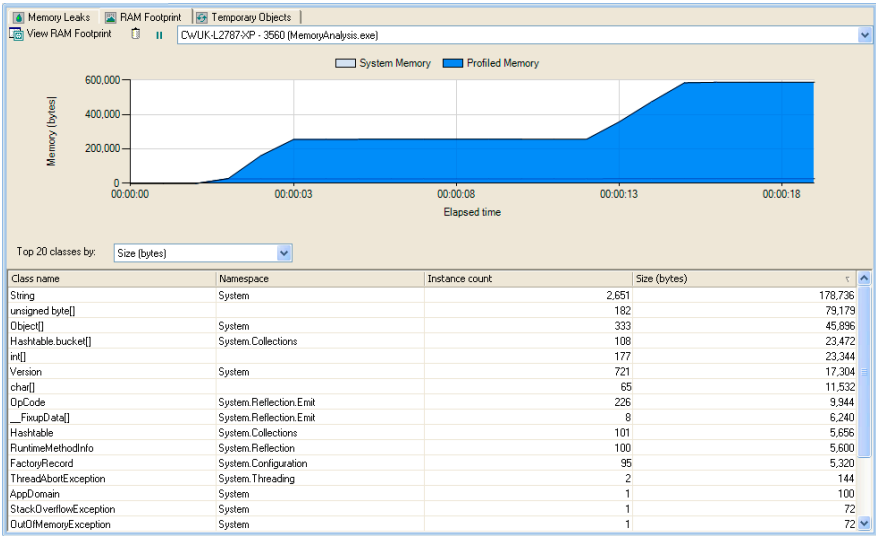


Figure 6.5: DevPartner memory analysis.

RAM footprint

RAM footprint looks at both the largest allocated objects and the methods and call trees responsible for allocating large amounts of memory. With these types of analysis it is possible to identify parts of the application that are causing its overall memory footprint to be larger than may be necessary. This often occurs when an application loads data and keeps it cached in memory for later use.

Temporary object analysis

Temporary Object Analysis looks at the allocation of relatively short-lived objects. These are objects that manage to stay around long enough to survive a couple of garbage collections, but then lose their references and become available for collection. These types of objects make full garbage collections run more frequently, which is inefficient. Having insight into where temporary objects are allocated can help a developer reduce object longevity and improve overall memory usage and performance.

Memory leak detection

The memory leak detection tool allows the developer to navigate through the application until they get to the point where they want to start tracking the leak. At that point, they press a button and all objects allocated *from* that point are recorded. When the developer has finished their test, they press a **View Memory Leaks** button, and the tool forces a full garbage collection before presenting the results of all of the classes allocated since tracking began and which survived collection.

The results can be viewed by class/object instance and also by function, as the call stack for each allocation is also recorded.

Performance Profiler

There are two separate performance profilers in DevPartner; one for function timing analysis the other, called Performance Expert, for function resource analysis, looking at the CPU, disk and network activity.

The timing analyser can profile both managed and native instrumented applications, though the resource analyser is a pure .NET tool only.

Performance timing analysis

The timing analyzer (Figure 6.4) can profile at the function and the code line level. It's also possible to profile both managed and native code at the same time, although the native code must be built with instrumentation.

Running a test is a simple matter of starting the application using the DevPartner Integration toolbar button within Visual Studio.



Figure 6.6: DevPartner toolbar buttons.

A couple of further buttons will appear on the toolbar, which will allow you to take performance snapshots, and the application will start.

The snapshots results are displayed within Visual Studio in a standard grid which can be sorted and filtered. It's also possible to reduce the scope of results by selecting to only view specific source or system modules. The main statistics provided per function (see Figure 6.7) include:

- Called (number of times the function was called)
- % in Method (% of time spent in function excluding time spent in calls to non-framework functions)
- % with Children (% of time spent in function including time spent in calls to non-framework functions)
- Average time (total time in function/number of calls).

The user can view the source for each function listed, giving them a timed line-by-line breakdown of the source code. Alternatively, it's possible to view the call tree for the function, and from here you can track backwards or forwards through the call tree to investigate the function's critical path (more on this later).

Method List	Source [Form1.cs]	Session Summary			
Method Name	% in Method	% with Children	Called	Average (us)	
MemoryAnalysis.Form1..ctor(void)	0.6	22.3	1	159,031.6	
MemoryAnalysis.Form1.InitializeComponent(v...	0.1	7.0	1	26,297.2	
MemoryAnalysis.CurrencyManager.Start(void)	0.0	0.2	1	6,891.6	
MemoryAnalysis.Form1.Form1_Load(Object, ...	0.0	0.3	1	5,208.0	
MemoryAnalysis.Form1.AddOrder_Click(Object...	0.0	0.2	5	626.2	
MemoryAnalysis.Form1.button2_Click(Object,...	0.0	1.4	1	2,174.8	
MemoryAnalysis.Form1.button1_Click(Object,...	0.0	0.8	1	1,735.8	
MemoryAnalysis.Form1.AllocateLOH(Int32)	0.0	1.4	1	1,517.9	
MemoryAnalysis.Form1.UpdateQueueLength(...	0.0	0.1	5	182.7	
MemoryAnalysis.CurrencyManager..ctor(void)	0.0	0.0	1	394.5	
MemoryAnalysis.CurrencyManager.add_OnPri...	0.0	0.0	6	4.2	
MemoryAnalysis.Order..ctor(void)	0.0	0.0	5	4.3	
MemoryAnalysis.CurrencyManager.Stop(void)	0.0	0.0	1	8.9	

Figure 6.7: DevPartner performance analysis.

Performance Expert Analysis

As with performance timing analysis, Performance Expert Analysis (Figure 6.5) is started from within Visual Studio, and additional buttons appear which allow you to take performance snapshots. This time, the application is being measured for CPU, disk, network activity and wait time, which are all potential bottlenecks.

When a snapshot is taken, the most CPU-intensive execution paths and functions are displayed, and various forms of analysis are available. Execution path analysis allows you to perform a call graph analysis on the execution path. The functions analysis displays a grid of resource statistics for each function, allowing you to sort by each column. From this view, you can quickly determine the most CPU/disk, etc. intensive functions.

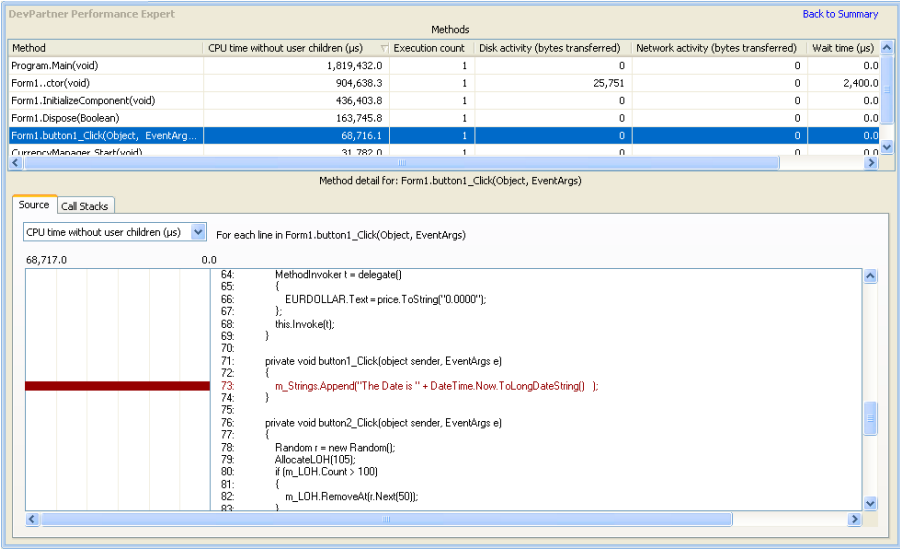


Figure 6.8: DevPartner Performance Expert Analysis.

Other tools

In addition to the profiling tools, DevPartner also has:

- **Code review tool**
Code quality, standards, security analysis
- **Code coverage analyzer**
Determines how much of an application has been tested, and what hasn't been tested.

Microsoft Visual Studio 2008 profiling tools

Visual Studio 2008 Team Edition has a built-in performance and memory profiler, and you can choose to use either sampling or instrumentation methodologies. As well as the usual performance timing and memory allocation information, it is also possible to collect additional CPU counters, Windows events and Windows counters with these tools.

A profile report can be produced at any time, at which point a summary report is displayed from where you can drill into the data or select more detailed reports from a drop-down list. The report view provides a filter mechanism which allows for the creation of sophisticated queries on the available data. It's also possible to compare reports, which is useful, for example, to check that an optimization has been successful.

Performance Explorer

Visual Studio's profiler settings have a lot of options available, and so multiple performance analysis configurations can be set up for the same application in a Performance Explorer window. You may, for example, have separate configurations for Sampling and Instrumentation, and for Memory and Performance profiling. All of the reports for each configuration are stored together.

Performance Analyzer

On completion of a test, the performance analyzer will give a summary of the worst functions, as well as reports on most called functions, functions with the most individual work, and functions taking the longest.

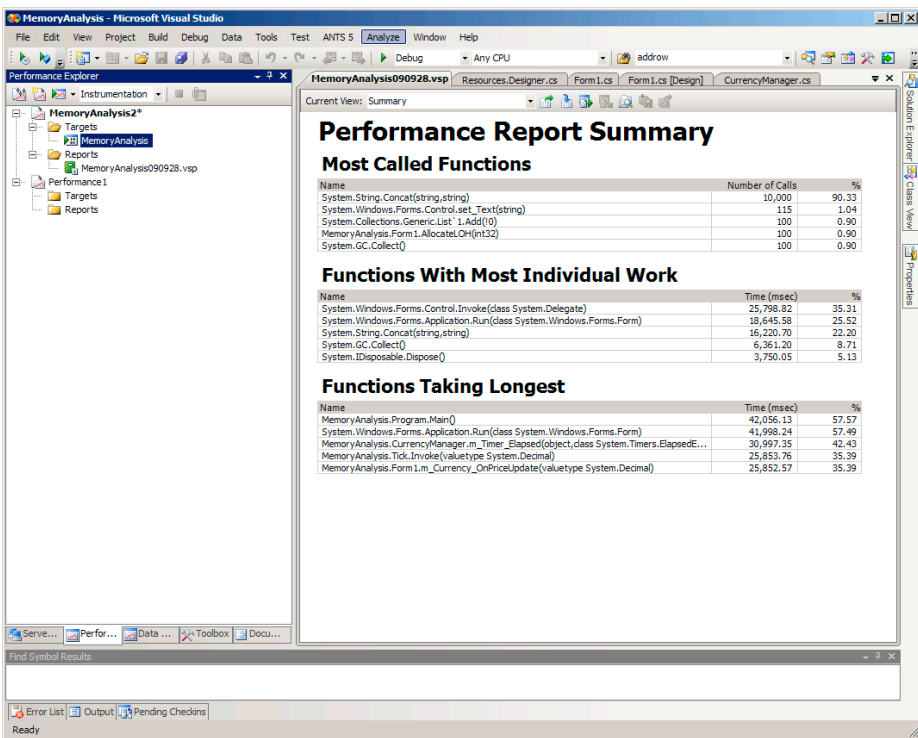


Figure 6.9: Visual Studio 2008 Performance Profiler.

From here, you can choose a function from the list and view its callers, its child calls, or you can view the source, if available. The profiler works well at the function level, but has only a crude reporting mechanism to give code-line level statistics. The next version of Visual Studio 2010 will address this issue, giving full line-level timings and source code visibility. Many other reports can be selected, including a function grid to determine the slowest functions, and a call tree to identify the slowest execution paths. It is possible, using a combination of the reports, to find function bottlenecks, which is naturally a good starting point to being able to correct these issues.

Memory Analyzer

To carry out a memory analysis (see Figure 6.10) you need to make sure that the Performance Explorer configuration you are using has the following .NET memory profiling options set:

- Collect .NET object allocation information
 - helps identify expensive allocated classes and functions.
- Also collect .NET object lifetime information
 - memory leaks
 - mid-life crisis detection
 - Large Object Heap issues.

The application can now be launched from within the Performance Explorer window.

Run your test transaction, then either press the **Stop** button on the Performance Explorer toolbar, or close your application.

The memory analyzer (see Figure 6.11) reports on:

- functions allocating the most memory
- types with the most memory allocated
- types with the most instances.

From the summary, you can view reports that detail:

- object allocation (largest allocated classes and the methods that allocated them)
- object lifetime (when objects are de-allocated)
- call tree (most memory expensive function call trees).

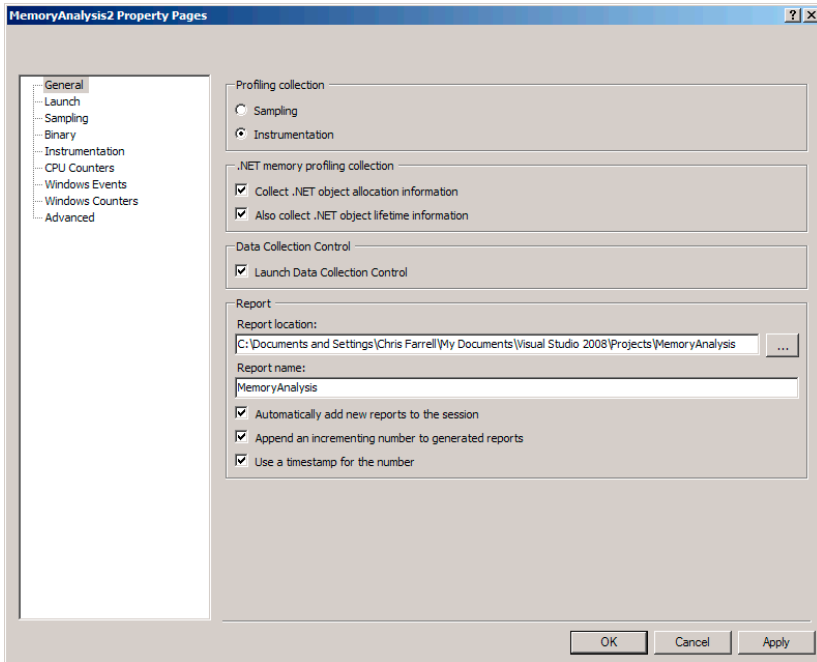


Figure 6.10: Visual Studio 2008 memory analysis configuration.

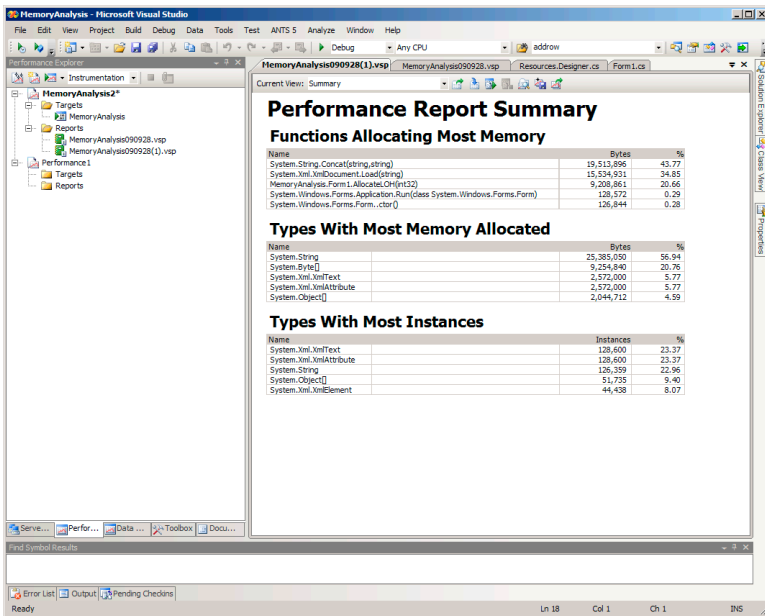


Figure 6.11: Visual Studio 2008 memory analysis.

What to look for

Let's now look briefly at some of the main types of problem that can be uncovered using the tools described above. As this chapter is just an introduction to the noble art of profiling, all of the techniques mentioned will be described in more detailed in subsequent chapters.

Performance analysis

The following key indicators can be used to identify potential bottlenecks and problems in your code. We will cover performance profiling in Chapter 7.

High call count

Functions with very high call counts should be treated with suspicion and investigated. Often the high call count is valid, but sometimes it's due to an error in event handling, and can be a major source of unintended processing.

Resolution

Using the call graphing facility of your performance tool, it should be possible to trace back to where the calls to the function originate, and decide if it is acceptable behaviour. It's a very quick and easy check, and a very quick optimization if a problem is found.

I have actually lost count of the number of times I have found this issue in live code!

Slowest function excluding child calls

This is the slowest function where the body of the function itself is responsible for the time. It includes time spent calling .NET framework functions, but excludes time spent calling other source code functions. In other words, it's answering the question, "What's the slowest function we have written?"

Resolution

Identify the slowest functions excluding child calls and then, if available, look for the slowest code lines and determine if they are optimizable. You will often see slow lines waiting for database and web service calls to return.

Slowest function including child calls

This is the slowest function where the total cost of the functions, *including* time spent into calls to child functions (we have written), is accounted for.

Resolution

Use your tool's call graph facility to explore the slowest part of the call tree.

Functions with high CPU utilization

Any function with high CPU utilization is a prime candidate for optimization, as high resource demands can be a key bottleneck.

Resolution

Identify the most CPU-intensive lines of code within the function and determine if there are workable optimizations that may apply.

Functions with wait time

Functions with wait time can indicate performance problems in other application layers, or problems with thread locking (I'll discuss thread locking in Chapter 7, where it'll be more relevant).

Resolution

Identify which resource the function is waiting for, e.g. database or web service, then investigate the cause of the contention on that layer.

Functions generating disk activity

A function generating disk activity needs to be investigated further, as it is demanding resources and so is a potential bottleneck.

Resolution

Make sure the disk activity is necessary, particularly if this is a server application. Even if it is necessary, try to find an alternative if possible.

Functions generating network activity

A function generating network activity needs to be investigated further as another potential bottleneck.

Resolution

Make sure the network activity is valid and not an artifact left behind from prototyping or developer testing. Ensure that the number of times this network activity occurs is as low as possible, to reduce the effect of latency. If possible, get more data in one hit.

Memory analysis

When and where you create objects in your code has far-reaching consequences for the application as a whole. Allocating too early and for too long will increase the application's memory footprint. Leave references to objects in collections or from event listeners, for example, and they will stay in memory indefinitely.

We're going to look at Memory Analysis in a lot more detail in Chapter 8 but, for now, let's look at some of the key types of analysis that can help improve your application's memory profile.

Memory leak detection

Finding memory leaks is all about identifying objects that are allocated but never garbage collected. Memory leaks always get worse so, in theory, the longer the application runs, the bigger the leak will get, and the easier it will be to see. That doesn't really help when profiling, though, because you need to be able to identify a leak quickly.

Profiling tools help leak detection by allowing you to take memory snapshots. A snapshot usually involves forcing a garbage collection and then recording all of the objects that are left behind in memory. Objects that repeatedly survive garbage collection should be investigated further.

If objects of the same type continually survive garbage collection and keep building up in memory, you need to investigate the references that are keeping those objects in memory. Tracking object references back to source code allows you to find the cause of the leak in your own code, which means you can fix it.

Some profilers track memory allocation by function calls, which allows you to see the functions that are potentially leaking memory. This can also be a highly effective technique for finding a memory leak.

Excessive memory usage

Reducing the overall memory footprint can help an application to coexist with other applications on the desktop or server. It's always worth checking where your application is allocating and retaining large chunks of memory, just to ensure that this behaviour really is necessary. Often it's done for performance reasons and is perfectly valid, as the memory overhead is worth the performance gain. Unfortunately, I have analyzed many applications where large amounts of data are held but then never used again, and this is the kind of behaviour you need to be on the lookout for.

Inefficient allocation and retention

Certain programming techniques, such as string concatenation, for example, can create large numbers of intermediate objects on the heap, which makes the garbage collector work harder than it needs to. The harder the garbage collector works, the greater the performance impact on the application.

Detecting when your application is allocating inefficiently will allow you correct the issue.

Large Object Heap fragmentation

The Large Object Heap is used to store objects that are greater than 85K in size. The trouble is, it can become fragmented, which can lead to the heap size expanding to larger than it needs to be. In severe cases, this can eventually lead to Out of Memory issues. See Chapter 8 for more detail on this.

Production / load test clues

Problems uncovered during load test or in production will usually be accompanied by a wide variety of performance metrics collected from multiple servers. Below are some of the most useful statistics that are widely regarded as key indicators of performance issues (Meier, Vasireddy, Babbar, Mariani, Mackman, and Microsoft, 2004). They are, at the very least, a starting point, and will help you identify where to begin your analysis and which tools to employ. For this article, go to [HTTP://MSDN.MICROSOFT.COM/EN-US/LIBRARY/MS998579.ASPX](http://msdn.microsoft.com/en-us/library/ms998579.aspx).

General performance counters

The following performance counters can act as general guidelines for different performance problems. Please refer to Chapter 3 for a more detailed breakdown.

- Processor\% Processor Time
- Memory\% Committed Bytes in Use
- PhysicalDisk\%Idle Time
- Network Interface\Output Queue Length
- .NET CLR Memory\% Time in GC
- .NET CLR Memory\# Gen 0,1,2 Collections
- .NET CLR Memory\# of Pinned Objects
- .NET CLR Memory\Large Object Heap Size
- .NET CLR LocksAndThreads\Contention Rate/sec
- ASP.NET\Requests Queued
- ASP.NET\Requests Rejected

Managing profiling results

Each of the profiling tools stores the profile data in a proprietary flat file format, although some of the tools allow the data to be exported to XML or CSV files.

The main benefit to XML or CSV export is that you can use the data to generate your own reports and publish that data to other systems. This becomes more important when you begin automating your unit testing because you can also analyze the executing tests using a profiler. Instead of just getting Pass and Fail for your tests, you could also collect performance and stability metrics. By comparing these metrics with previous test runs, it's then possible to identify problems as they occur.

Comparing analysis runs

Applications such as Visual Studio 2008 Profiler and DevPartner Professional have tools which allow various profiling results to be compared, and ANTS Memory Profiler allows for the comparison of memory profiling snapshots. This feature can help to quickly identify where there has been a performance improvement or degradation.

Pre-check-in requirements

In support of proactive analysis, it's a good idea to require developers to include evidence of performance and memory analysis results as part of a source code check-in procedure at the end of a unit of work.

This could be as simple as a manual procedure in which all of the profiler results files are zipped together and added (suitably labelled) to the project office. Alternatively, the source control system itself could be used to define a pre-check-in process requiring the addition of profile results files. This largely depends on how extensible the source control system is. Microsoft Team Foundation Server 2005 and 2008 allow custom check-in policies to be defined, allowing more complex check-in procedures.

Continuous integrated testing

Tools which support command-line execution and XML export can be incorporated into an automated testing framework, in which the automated tests are run and the executing process is profiled using performance or memory analysis.

The results are then extracted to XML and uploaded to a results server, along with the results for the control cases.

To make life even easier, an automated testing framework can be set up to identify when the performance of a test transaction has degraded, and report it to the development team.

Summary

Knowing how to profile an application, and understanding what the potential issues are, will help you write better code. Routinely testing the functionality you have written using a profiler, and looking for the common bottlenecks and problems will allow you to find and fix many minor issues that would otherwise become bigger problems later on.

Load testing as early as possible during development, as well as adding to these tests and running them regularly with the latest builds, will identify problems almost as soon as they occur. It will also highlight when a change has *introduced* a problem.

In the next two chapters, I will go through the performance and memory issues you might encounter, and techniques you can use to deal with them. I will also highlight how to use some of the most common tools to find and fix problems in your code.

.NET and SQL Server Tools from Red Gate Software

Pricing and information about Red Gate tools are correct at the time of going to print. For the latest information and pricing on all Red Gate's tools, visit www.red-gate.com

redgate[®]
ingeniously simple tools

ANTS Memory Profiler™

\$495

Profile the memory usage of your C# and VB.NET applications

- Locate memory leaks within minutes
- Optimize applications with high memory usage
- Get clear, meaningful profiling results, for easy interpretation of your data
- Profile any .NET application, including ASP.NET web applications

"Freaking sweet! We have a known memory leak that took me about four hours to find using our current tool, so I fired up ANTS Memory Profiler and went at it like I didn't know the leak existed. Not only did I come to the conclusion much faster, but I found another one!"

Aaron Smith IT Manager, R.C. Systems Inc.

ANTS Performance Profiler™

from **\$395**

Profile and boost the performance of your .NET code

- Speed up the performance of your .NET applications
- Identify performance bottlenecks in minutes
- Drill down to slow lines of code, thanks to line-level code timings
- Profile any .NET application, including ASP.NET web applications

"Thanks to ANTS Performance Profiler, we were able to discover a performance hit in our serialization of XML that was fixed for a 10x performance increase."

Garret Spargo Product Manager, AFHCAN

"ANTS Performance Profiler took us straight to the specific areas of our code which were the cause of our performance issues."

Terry Phillips Sr Developer, Harley-Davidson Dealer Systems

Visit www.red-gate.com for a 14-day, free trial

Exception Hunter™

\$295

Analyze your .NET assembly for possible unhandled exceptions

- Analyze your .NET assemblies
- Locate unhandled exceptions that can be thrown by a particular method – before you even ship
- Find out where those exceptions originate (down to a single line of code)
- Decide which exceptions need to be handled (with some exception-handling code) before you release

.NET Reflector®

Explore, browse, and analyze .NET assemblies

- View, navigate, and search through the class hierarchies of .NET assemblies, even if you don't have the source code for them
- Decompile and analyze .NET assemblies in C#, Visual Basic and IL
- Understand the relationships between classes and methods
- Check that your code has been correctly obfuscated before release

SmartAssembly™

from \$499

Protect your .NET code, your Intellectual Property, and your business

- Obfuscate and secure your .NET application
- Optimize your .NET assemblies (remove non-useful code and metadata and perform other code optimization) and simplify the deployment of your application
- Save countless hours of debugging and diagnostics
- Build a bullet-proof application

"It is the most effective obfuscation, optimization, and all-round compilation improvement tool we've come across to date."

John Cioni Fabsoft

Visit www.red-gate.com for a 14-day, free trial

SQL Compare Pro®

\$595

Compare and synchronize SQL Server database schemas

- Automate database comparisons, and synchronize your databases
- Simple, easy to use, 100% accurate
- Save hours of tedious work, and eliminate manual scripting errors
- Work with live databases, snapshots, script files, or backups

"SQL Compare and SQL Data Compare are the best purchases we've made in the .NET/SQL environment. They've saved us hours of development time, and the fast, easy-to-use database comparison gives us maximum confidence that our migration scripts are correct. We rely on these products for every deployment."

Paul Tebbutt Technical Lead, Universal Music Group

SQL Data Compare Pro™

\$595

Compare and synchronize SQL Server database contents

- Compare your database contents
- Automatically synchronize your data
- Row-level data restore
- Compare to scripts, backups, or live databases

"We use SQL Data Compare daily and it has become an indispensable part of delivering our service to our customers. It has also streamlined our daily update process and cut back literally a good solid hour per day."

George Pantela GPAnalysis.com

Visit www.red-gate.com for a 14-day, free trial

SQL Prompt Pro™

\$295

The fastest way to work with SQL

- Code-completion for SQL Server, including suggestions for complete join conditions
- Automated SQL reformatting with extensive flexibility to match your preferred style
- Rapid access to your database schema information through schema panes and tooltips
- Snippets let you insert common SQL fragments with just a few keystrokes

"With over 2,000 objects in one database alone, SQL Prompt is a lifesaver! Sure, with a few mouse clicks I can get to the column or stored procedure name I am looking for, but with SQL Prompt it is always right in front of me. SQL Prompt is easy to install, fast, and easy to use. I hate to think of working without it!"

Michael Weiss VP Information Technology, LTCPCMS, Inc.

SQL Search™

Free

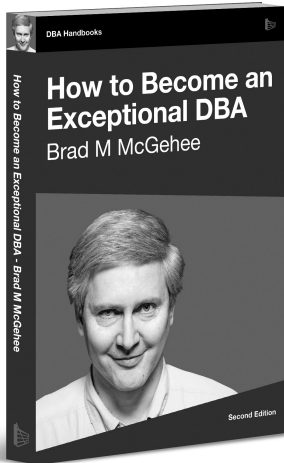
A free Management Studio add-in to search for SQL in your databases

- Find fragments of SQL text within stored procedures, functions, views, and more
- Quickly navigate to objects wherever they happen to be on your servers
- Find all references to an object
- No need to use a separate tool

Visit www.red-gate.com for a 14-day, free trial

How to Become an Exceptional DBA

Brad McGehee

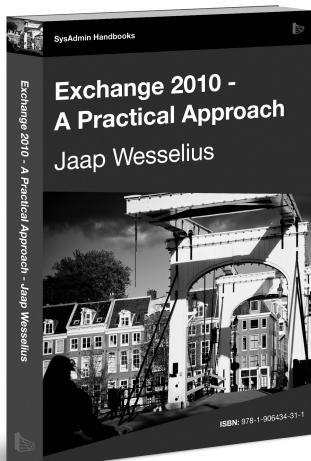


A career guide that will show you, step by step, exactly what you can do to differentiate yourself from the crowd so that you can be an Exceptional DBA. While Brad focuses on how to become an Exceptional SQL Server DBA, the advice in this book applies to any DBA, no matter what database software they use. If you are considering becoming a DBA, or you are a DBA and want to be more than an average DBA, this is the book to get you started.

ISBN: 978-1-906434-05-2
Published: July 2008

Exchange 2010 – A Practical Approach

Jaap Wesselius

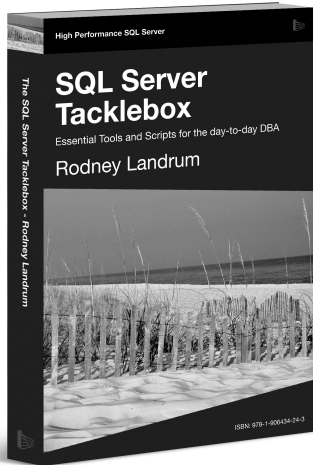


As a practical field-guide to Exchange Server 2010, this book will tell you exactly what you need to know to get started with upgrading, installing, configuring, and managing your new Exchange Server. If you need to get to grips with Exchange Server 2010 fast, or you want a short, to-the-point, practical guide to Microsoft's latest offering, then you should read this book.

ISBN: 978-1-906434-31-1
Published: December 2009

SQL Server Tacklebox

Rodney Landrum



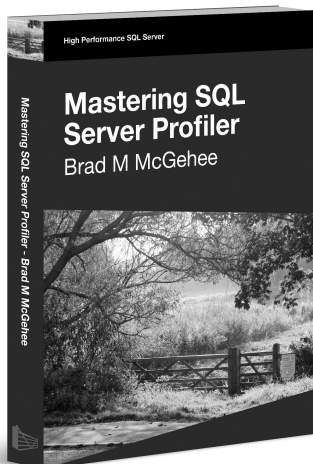
As a DBA, how well prepared are you to tackle "monsters" such as backup failure due to lack of disk space, or locking and blocking that is preventing critical business processes from running, or data corruption due to a power failure in the disk subsystem? If you have any hesitation in your answers to these questions, then Rodney Landrum's SQL Server Tacklebox is a must-read.

ISBN: 978-1-906434-25-0

Published: August 2009

Mastering SQL Server Profiler

Brad McGehee



For such a potentially powerful tool, Profiler is surprisingly underused; unless you have a lot of experience as a DBA, it is often hard to analyze the data you capture. As such, many DBAs tend to ignore it and this is distressing, because Profiler has so much potential to make a DBA's life more productive. SQL Server Profiler records data about various SQL Server events, and this data can be used to troubleshoot a wide range of SQL Server issues, such as poorly-performing queries, locking and blocking, excessive table/index scanning, and a lot more.

ISBN: 978-1-906434-16-8

Published: January 2009