Summer 2010

Leveraging Virtualization for Performance Driven Development

Matthew Sullivan
Regis University

Regis University
College for Professional Studies Graduate Programs
Final Project/Thesis

Disclaimer

LEVERAGING VIRTUALIZATION FOR PERFORMANCE DRIVEN DEVELOPMENT

A THESIS SUBMITTED ON MONDAY, APRIL 26, 2010

TO THE DEPARTMENT OF INFORMATION TECHNOLOGY

OF THE SCHOOL OF COMPUTER & INFORMATION SCIENCES

OF REGIS UNIVERSITY

IN FULFILLMENT OF THE REQUIREMENTS OF MASTER OF SCIENCE IN SOFTWARE

ENGINEERING
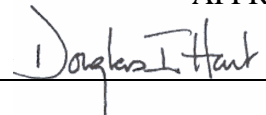
BY

Matthew Sullivan

APPROVALS

Douglas I. Hart

Daniel M. Likarish

Stephen D. Barnes

Abstract

This thesis contains the research component of a software engineering study to create a .NET application performance testing lab, and several guided learning activities intended to teach the fundamentals of how to use it. In arriving upon the research which serves as the groundwork for this project, an introduction to the concepts of software performance, the risks associated with performances, and an approach to mitigating this risks called "performance driven development" is presented. This introduction is expanded by an overview of how performance is affected from application, network, database and presentation aspects.

To address problems associated with performance in .NET web applications, a virtual test lab has been created on the software engineering lab server at Regis University's Academic Research Network (ARNe), and this paper documents the architecture of that test lab. In order to demonstrate how it can be used students, developers or others previously unfamiliar with performance testing, a series of presentations has been composed, and this paper represents the research conducted in composing them. This research includes a basic level understanding of Visual Studio Team System 2008's test tools, and virtualization with VMWare.

This paper ends after a self-evaluation of the constructed deliverables, a conclusion, and several appendices from published sources and references.

Table of Contents

Introduction

Software performance testing

In software engineering, processes and practices revolving around the discovery, analysis, specification and verification of functional requirements can be considered relatively mature in many enterprises. Similarly, institutions of advanced learning in software engineering such as Regis University often integrate functional requirement analysis in courseware and projects as a lifecycle approach where students begin development with use case analysis and proceed through the development process with this analysis as its bedrock. This approach is thought to result in software which assured to meet the varied business needs of its users, but may not result in overall user satisfaction if there is insufficient attention to software performance (Lewis, 2009).

For the purpose of this thesis, performance testing is an umbrella term for numerous types of non-functional testing intended to assure that software is adequate for its expected level of usage. In the discipline of software testing, a distinction is often made between functional and non-functional testing. While the definitions and meanings of these terms may vary among practitioners, functional testing can be thought to include the verification and validation that software meets the functional and system requirements specified in early stages of the project and development lifecycles (System testing, 2009). Non-functional testing can be thought to include the verification and validation of qualities not specified or documented in the analysis of business requirements, often revolving around quality related aspects like scalability and reliability (Non-functional testing, 2009). There are a wide variety of frequently described types of non-functional testing often associated with performance testing, such as load testing, resilience testing, stress testing, and volume testing, and there are also numerous types of non-functional testing not typically included in the scope of performance testing, such as security testing, compatibility testing, and usability testing (Non-functional testing, 2009).

In this thesis paper, the author will synthesize a body of knowledge supportive of an iterative approach to performance testing which will be referred to as "performance driven development". This body of knowledge will provide the basis for a short series of self-guided activities that can introduce a student or software engineer to this approach and to performance testing in general. This thesis will also discuss the potential role of virtual machines in performance driven development.

This chapter will introduce associated concepts, provide a justification for this research through an elucidation and documentation of risks associated with software performance, distinguish between a performance driven approach and one which treats performance as an acceptance criterion, describe challenges to the implementation of a performance driven approach, establish the role that virtualization can play in overcoming these challenges, and clarify the scope of this paper.

Risk factors associated with software performance

As previously stated, the concern of functional or system testing is the validation and verification that the software produced in a development effort meets the objectives and needs of its users. Non-functional software testing typically determines whether the software adequately addresses a set of known risk factors. This section will describe these risk factors and provide documentation of the negative impact these risks can present on software and the enterprises which they serve.

As web-based applications become more prevalent and integrated into the users' computing environments, users increasingly expect the application to perform at a comparable level of quality performance as a corresponding desktop application (Microsoft Application Consulting and Engineering Team, 2003). Poor performance in software can also cost organizations time in completing critical tasks, revenue by resulting in lost customers, sales, and strategic positioning by failing to provide information at the time when it is most useful (Subraya, 2006). A study published by Zona Research in 2001 indicated that inadequate performance for low to mid-band (dial-up, DSL or Cable) users accounted for as much as $25 billion in lost revenue annually for online retailers (M2 Presswire, 2001). As far back as 1998, a survey of

web consumers determined that some users will stop trying to complete a transaction if the system has not responded within 10 seconds, and that only 5 percent continue to wait after 30 seconds (Subraya, 2006). It is a reasonable presumption that expectations have risen considerably since then.

When customers encounter websites which do not respond within their subjective expectations, they may stop trying to use it that day, for several days, stop using it altogether, or even discourage other users from using the website (Subraya, 2006). Other stakeholders for an application may expect continuous availability, that transactions are completed within an expected amount of time, or that the software utilizes little enough shared system resources to minimize impact on other applications (Subraya, 2006). There are also costs associated with software which requires an inordinate amount of human knowledge and effort for the maintenance and management of the application in production environments (Subraya, 2006).

Aside from considering whether software meets the expectations of its users and stakeholders, it is important to know whether it is prepared for unexpected situations. For example, after the terrorist attacks of September 11, 2009, increased levels of use for MSN news websites exposed a memory leak which caused repeated outages, diminished perception among customers, and may have even resulted in increased public anxiety during an already difficult period (Microsoft Application Consulting and Engineering Team, 2003). All of these are concerns which represent risks to software development projects.

In *Performance Testing Guidance for Web Applications—Patterns & Practices*, the authors sort the risks addressed by performance testing into three categories: speed, scalability, and stability (Meiers, Farre, Bansode, Barber, & Rea, 2007). Of these three categories, speed is said to be typically be a concern of end-users, scalability is said to be a concern of the business or enterprise, and stability is said to be a concern for technical or mainentance related stake-holders (Meiers, Farre, Bansode, Barber, & Rea, 2007). Nevertheless, addressing these risks and establishing strategies for mitigating them presents value by saving time for all stakeholders, and more effectively using technical resources (Meiers, Farre, Bansode, Barber, & Rea, 2007).

Among the speed related risks addressed by software performance testing are the risk that the application is not "fast enough" to satisfy end users, that the system cannot process data into information while it is still relevant, and whether the system can complete a task before exceeding the maximum

response time preferences, thus avoiding errors and exceptions (Meiers, Farre, Bansode, Barber, & Rea, 2007). These risks can mitigated by a variety of strategies including the establishment of realistic expectations and service level agreements which meet or exceed the needs and desires of users, benchmarking or comparing the results of performance tests to prior versions or comparable applications, executing tests which reproduce typical and peak workloads for the application, referring to performance test results when making architecture and business decisions, and considering the time-critical transactions when designing tests (Meiers, Farre, Bansode, Barber, & Rea, 2007). Another approach recommended is the development of a questionnaire about performance for users asking whether there is a perceived problem with slow response times, unavailability, frequent time outs, also whether these problems affect all users or specific users, and whether these problems are consistent and affect the entire web site (Subraya, 2006).

The risks associated with the category of scalability involve the concurrency level or number of users that can be handled by an application, the amount of data that can be processed efficiently by the application, and the determination of limitations of the capacity of the application (Meiers, Farre, Bansode, Barber, & Rea, 2007). Among these risks are that the application will provide an inconsistent level of performance, will be unable to store or efficiently use the entirety of the data it is expected to collect over its lifecycle, will be unstable or non-functional under anticipated or unanticipated peak usage, or that the application may exceed its capacity without adequate warning (Meiers, Farre, Bansode, Barber, & Rea, 2007). Performance testing may help mitigate these risks by providing a comparison of how fast transactions are completed at differing levels of load, replicating or simulating realistic and extreme workloads during tests, utilizing test data which is similar to that which is expected to be encountered, informing the architectural and business decisions with performance test results, and determining the point at which systems reach their capacity to devise countermeasures and contingency plans (Meiers, Farre, Bansode, Barber, & Rea, 2007).

The risk category of stability includes those that concern reliability, recoverability, and the amount of predictable downtime (Meiers, Farre, Bansode, Barber, & Rea, 2007). Risks in this category include that the application will be unable to run for long durations without resulting in corruption, degradation, or the

need to maintain or reboot systems, that there will be undetected data loss in the event of unexpected system outages, that the application will encounter functional lapses after recovering from outages, that the application cannot be patched or updated without a lapse in service, that segments of load balanced systems can have deleterious effects on parallel segments, or that specific transactions may cause negative impacts to the entire system (Meiers, Farre, Bansode, Barber, & Rea, 2007). Performance testing can determine the exposure to these risks through stress tests, capacity tests, and endurance tests, but sometimes stability issues can be indicated in benchmarking and other moderate to low level performance tests (Meiers, Farre, Bansode, Barber, & Rea, 2007). Performance testing can help by determining the point at which the performance may degrade, the responsible component for this degradation, and the impact on company sales and technical support costs (Subraya, 2006). Performance testing can help mitigate these risks by simulating these scenarios and assessing their effects, analyzing the integrity of the system after prompting unexpected outages or failures, and performing basic maintenance activities such as backups or virus definition updates while under load (Meiers, Farre, Bansode, Barber, & Rea, 2007).

Performance as an acceptance criterion

In this section, problems with an approach to performance testing placed late in the development lifecycle will be cited from available literature. This will be expanded upon through an anecdotal description of the approach to performance testing observed in my workplace, and this will be provided as an example of addressing performance as an acceptance criterion rather than as a development driver. The drawbacks to this approach will be assessed.

As previously discussed in the section on risk factors associated with software performance, there is an increasing level of demand from businesses for performance of internet and web based applications (Subraya, 2006). Failure to meet these demands as well as insufficient performance in unexpected situations can result in loss of time, revenue, and business opportunities (Subraya, 2006). The field of performance testing is newer and less mature than other areas of testing, and the activities involved are not well

understood by many practitioners and stakeholders (Meiers, Farre, Bansode, Barber, & Rea, 2007). This condition coupled with the perception of challenges in setting up and integrating performance testing often results in the placement of performance testing late in the development lifecycle (Meiers, Farre, Bansode, Barber, & Rea, 2007). As a result, problems are detected when it is expensive or impractical to address the underlying causes (Meiers, Farre, Bansode, Barber, & Rea, 2007).

I am a software tester for a major accounting firm, and specialize in performance testing. The software development lifecycle at this firm is divided into four distinct phases with some overlap. The names of these phases are "define" (in which the scope and business justification is established), "discover" (in which the business and technical requirements for the application are determined and documented), "develop" (in which the application is built and tested) and "deploy" (in which the application is delivered to end users in the production environment).

In the context of this lifecycle, the test lead for a project is not involved in the define phase at all. The test lead will begin creating a test plan at the end of the "discover" phase when the initial drafts of the requirements specifications are available. At that time, the test lead will determine (often subjectively) whether performance testing is required at all. If performance testing is required, a performance tester (or group of testers led by a performance test lead) will be assigned. Often, this assignment is made near the end of the develop phase, and testing activities commence on a final version of the application in an integrated staging environment just prior to user acceptance testing. At the end of the prescribed set of performance tests, the test lead or performance test lead will issue a test results report with a recommendation on whether the application performs adequately, and whether the project can proceed to user acceptance testing and the deployment phase. It is because the objective of the testing is primarily this approval and recommendation absent of established performance objectives rather than guidance during the analysis, design and development process that I describe this approach as being "acceptance criterion" driven rather than as a "development driver".

In my opinion, the drawbacks to this approach are as follows. As previous stated by cited sources, when problems are detected during this approach, they are very difficult and expensive to trace to a root

cause (Meiers, Farre, Bansode, Barber, & Rea, 2007). Developers have frequently become largely disengaged from the project at this point, so there is a tendency to address the issue by increasing available system resources (such as RAM, bandwidth or dedicated CPU), which in turn increases the cost to deploy the system. There are also residual costs associated in the form of increased staffing requirements for the operation of the data center (because more machines and network infrastructure are needed), and in licensing because applications cannot share the same server. Additionally, because the root cause of a performance problem is not discovered, future development efforts do not benefit from this knowledge, and the costs are further compounded.

Performance as a development driver

In this section, an alternative to the previously described approach is presented which could be described as "performance driven development". It will begin with an examination of such approaches as recommended in available literature. From these sources a broadly defined understanding of performance driven development will be synthesized as supportive to the objectives of this thesis.

In *Performance Testing Guidance for Web Applications—Patterns & Practices*, an approach to performance testing is presented with nine activities which occur over each stage of the development lifecycle (Meiers, Farre, Bansode, Barber, & Rea, 2007). The objectives of the approach include identification of bottlenecks, establishment of a baseline for testing in the future, support tuning activities for the application and dependant systems, determine whether the application meets performance requirements and objectives, collecting information to assist with architectural and business decisions, and determination of hardware configuration required when the application is transitioned to production environments (Meiers, Farre, Bansode, Barber, & Rea, 2007). These objectives indicate that the purpose of testing is not just to determine whether acceptance criteria have been met, but to assist with the development efforts (Meiers, Farre, Bansode, Barber, & Rea, 2007). The approach contains nine activities (Meiers, Farre, Bansode, Barber, & Rea, 2007).

The first activity is "understand the project vision and context", and during this activity the performance requirements and ramifications of the system are understood from the inception of the project, and numerous questions to consider at this stage are listed (Meiers, Farre, Bansode, Barber, & Rea, 2007). The second activity is "identify reasons for testing performance", and at this point the risks indicated by the project are mapped to performance testing activities which can address and assess them (Meiers, Farre, Bansode, Barber, & Rea, 2007). The third activity is "identify the value performance testing adds to the project", and at this point the stakeholders should understand and buy into the specific project and business objectives that performance testing can provide (Meiers, Farre, Bansode, Barber, & Rea, 2007). The fourth activity is "configure the test environment", and at this point the performance testing tools and the system which will be tested are built and configured (Meiers, Farre, Bansode, Barber, & Rea, 2007). The fifth activity is "identify and coordinate tasks", and at this point the planning, scheduling and allocation of resources are made (Meiers, Farre, Bansode, Barber, & Rea, 2007). The sixth activity is "execute tasks", and at this point the tests are constructed and executed, and any other work surrounding the current iteration of testing is performed (Meiers, Farre, Bansode, Barber, & Rea, 2007). The seventh activity is "analyze results and report" in which the project team is made aware of the results of the prior activity (Meiers, Farre, Bansode, Barber, & Rea, 2007). The eighth activity is "revisit activities 1-3 and consider performance acceptance criteria", and at this point the objectives and value of performance testing in the project context is reassessed based on the findings of the iteration which just completed (Meiers, Farre, Bansode, Barber, & Rea, 2007). The nineth activity is "reprioritize task" in which planning for another iteration of testing is made based on the determinations of the prior activity (Meiers, Farre, Bansode, Barber, & Rea, 2007). At this point, the process resumes at the fifth activity in another iteration (Meiers, Farre, Bansode, Barber, & Rea, 2007). When placed in the context of the software development lifecycle described in the previous section, one can see that these activities commence during the define stage, that significant planning occurs during the discover stage, and that testing is executed iteratively throughout the develop stage, and thus has the potential of discovering performance issues early when the developer can adequately investigate the root cause and address them in a cost effective manner.

In *Integrated Approach to Web Performance Testing: A Practitioner's Guide*, another iterative approach is described which the author refers to as the "Performance Testing Lifecycle" (Subraya, 2006). In this book, the author refers to a commonly presented development lifecycle which includes the following phases: planning, analysis, design, implementation, testing, and maintenance (Subraya, 2006). It is stated that a common and inadequate approach is to place performance testing in the testing phase, which is late in the development lifecycle (Subraya, 2006). As an alternative, his performance testing lifecycle contains the following six stages: analysis of service level agreement, performance test definition, performance test design, performance test build, performance test execution, and performance test result analysis (Subraya, 2006). Similar to the nine activities in the previously described approach, the author recommends that these activities occur throughout the development lifecycle rather than being confined to a testing phase near the end, and states that increased value can be presented if executed and refined iteratively (Subraya, 2006).

Iterative lifecycle approaches are not unique to the domain of testing Microsoft .NET application. In *Using Rational Performance Tester Version 7*, the authors recommend an approach that contains ten steps (Patel, et al., 2008). The first step is to determine the system performance testing "questions" that need to be answered (Patel, et al., 2008). The second step is to characterize the workload that should be simulated in testing (Patel, et al., 2008). The third step is to identify what should be measured during the test executions (Patel, et al., 2008). The fourth step is to determine what the criteria are to consider a test successful (Patel, et al., 2008). In the fifth step, the workload is designed and modeled, including what variations should be part of the test (Patel, et al., 2008). The sixth step involves building the workload elements (such as test data) that was modeled and designed in the previous step (Patel, et al., 2008). The seventh step is to construct definitions for each of the tests for the purpose of collecting measurements (Patel, et al., 2008). The eighth step is to execute the test while monitoring the system activities to assure that a valid test is taking place (Patel, et al., 2008). The nineth step is to analyze the results, tuning where appropriate and repeat tests where necessary (Patel, et al., 2008). The tenth step is to gather the results of all executed tests and determine how they answer the questions documented in step one (Patel, et al., 2008).

In this section, we have reviewed three approaches to performance testing that are substantially different. The first approach gives much consideration and reconsideration on performance testing objectives and how they align to project vision. The second approach reimagines the performance testing process as a miniature lifecycle that spans the phases of the development lifecycle it supports. The third approach is more of a checklist that is specific to the workflow supported by the Rational Performance Tester tool. Nevertheless, these approaches share characteristics that indicate a greater trend. They are all approaches which put more emphasis on determining the performance required for the system under test. They are all approaches which span into areas of the development lifecycle where software is still being designed and constructed. Finally, they are all approaches which support iterative execution. It is these three shared qualities which indicate a trend which I am referring to as "performance driven development".

Challenges presented by performance driven development

Thus far in this introduction, we have reviewed available literature on the risks associated with inadequate software performance, and differentiated between approaches which treat performance testing as an acceptance criterion and those in which performance testing informs and helps drive performance efforts. In this section, some apparent challenges to the adoption of performance driven development will be discussed.

The first challenge is the availability of servers, networks, and other required components which make up the system that is being tested (Subraya, 2006). Many software applications reside on multiple tiers, and include clients, application servers, database servers, web servers, enterprise service buses and other components. Frequently, it is determined early in the process what are the characteristics of the network, and these should also be considered in performance testing efforts. Additionally, when determining optimal hardware and software configurations, preferences may need to be adjusted between test executions, RAM may need to be increased or decreased. Implementing these components and configurations can be an arduous task requiring coordination between team members across functional

areas, and this can make it that much more tempting to wait until the end of the development process, as inadequate as that approach may be.

The second challenge is the inexistence of knowledge and skills related to performance testing among developers, business analysts, and other stakeholders who are more active in the early part of the development lifecycle (Subraya, 2006). Organizations may not be prepared to dedicate testers (who have traditionally been trained and tasked with performance testing) to a larger portion of the project lifecycle, and while developers may the logical people to execute early performance tests, they may be less familiar with the tools and related knowledge. A related challenge is presenting the necessary information and skills in a format which can be quickly and easily by those whose primary function is not performance testing, and who may be impatient if handed a large textbook on the subject.

One more apparent challenge presented by the adoption of performance driven development is the expense associated with many of the available performance testing tools (Subraya, 2006). Purchasing additional licenses for this software may be cost prohibitive for many companies, and may also be impossible for software engineering students on a tight budget.

Solutions proposed in this research

This paper explores potential solutions to address each of these challenges faced when adopting performance driven development. First, it discusses how virtualization technology such as VMWare can be leveraged to provide a flexible, dynamic, and safe environment in which to test software performance. Virtual environments can be made available to testers and developers who can in turn build virtual machines, configure aspects such as CPU, RAM, network and disk structures, and adjust system preferences while minimizing risks and consequences. In addition, a functioning performance testing environment with Microsoft servers and testing tools has been constructed for these purposes, and information is included on its architecture and usage.

Second, a basic body of information and skills required to conduct basic performance tests are assembled in the form of this paper. While this information is not an exhaustive examination of the subject,

and is a synthesis and distillation of information contained in numerous prominent resources, it shall provide an overview and introduction that will be sufficient for a student or software engineer unfamiliar with the subject.

Finally, selected portions of the information in this paper are included in a set of self-guided learning activities which demonstrate performance testing concepts and techniques. This format is more digestible for those who are peripherally concerned with performance testing such as developers, and should help students and engineers quickly learn how to use the performance testing environment constructed as part of this research to enhance their skill set and improve the quality of the software they produce.

Conclusion

The thesis of this paper is that coupled with an adequate presentation of necessary skills and knowledge, virtualization can be leveraged for the adoption or instruction of performance driven development. The purpose of this research paper is to synthesize the body of the knowledge necessary for this instructional material, and supportive of the construction of an adequate virtual performance testing environment.

It is not within the scope of this research to develop a specific testing approach or methodology, as it is assumed from the review of literature that numerous approaches already exist that are performance driven. It is not within the scope of this research to conduct research on the efficacy or superiority of performance driven approaches to alternative approaches. This research is specific to performance considerations for Microsoft .NET web applications, and performance testing tools included in Visual Studio 2008 Team System. This research is specific to VMWare virtualization technology in the context of Regis University, but does not include the mechanisms or governance policies required to seamlessly provide access to users, or configuration

Aside from this introductory chapter, this research paper will include chapters on basic principles of software performance and testing, basic principles of virtualization, performance testing in Visual Studio 2008, and the software performance testing lab. Following these chapters will be a self assessment of the

efficacy of this research in addressing the previously listed challenges and validating the thesis, indications of further research that may be conducted, and a conclusion.

Basic principles of software performance and testing

In the previous chapter, risk factors associated with software performance were explored, establishing the importance of effective performance testing. Differentiation was made between approaches which treat performance testing as a milestone to be passed prior to deployment, and those which treat performance testing as a development driver. Apparent challenges to the adoption or instruction of performance driven development were listed, along with potential contributions that this research can make to overcome these challenges.

Before one can begin performance testing, it is worthwhile to understand the various aspects of software that can affect its overall performance. For the purpose of this paper, a distinction is made between a software application which is the individual logic and functionality performing the primary business functions of the software, and the software system which includes the application server, database, network, and other services which work together to perform all software functions. In this chapter, several of these aspects will be discussed, namely application performance, network performance, database performance, and presentation performance. Following these sections will be a discussion of tools and terminology commonly used in performance testing.

Application performance

The first software aspect that can affect the overall performance is the application itself. By this is meant the core logic and code which performs the fundamental business purposes of the software. This may localized to a client tier as in a desktop application, located in a server tier as in a client server application, or shared between tiers as in a distributed application (Multitier architecture, 2009). Frequently with Microsoft web applications, application logic is located in the server tier and is handled by Microsoft Internet Information Services (Internet Information Services, 2009).

Microsoft .NET web applications contain numerous types of executable code, files, and handler classes which together form the application (Hasan & Tu, 2003). Some of these are code modules which operate independently and contain functionality scoped within the application (Hasan & Tu, 2003). These can be written in any .NET compatible language such as C#.NET or VB.NET, and can be compiled either into an applications executable file, or as standalone .NET components (Hasan & Tu, 2003). .NET applications can also contain logic in web services which are invoked either at the client or application tier, and communicate via HTTP using XML formatted SOAP packets (Hasan & Tu, 2003). Some application functionality may reside in client side scripts using scripting languages such as Javascript (Hasan & Tu, 2003). Additionally, configuration files such as Web.config support the code modules, web services and other application logic (Hasan & Tu, 2003).

Application code in .NET can negatively impact performance in a variety of ways. For example, the core libraries of .NET contain numerous specialized collections such as arrays and dictionaries that are intended to offer optimal performance for specific purposes, and if used inappropriately can result in improper memory allocation or inefficient access to collection members (Meier, Vasireddy, Babbar, & Mackman, 2004). Looping can be performed in a variety of ways, and using the wrong type of loop can result in unnecessary extraneous executions, or compounding of inefficient code (Meier, Vasireddy, Babbar, & Mackman, 2004). Implicit conversion of data types can result in unnecessary boxing and unboxing as well as inefficient use of memory resources (Meier, Vasireddy, Babbar, & Mackman, 2004).

Within the .NET framework code modules, web services, configuration files, as well as web or windows forms are contained and modified within class libraries written in one of various .NET compatible languages depending on the developer's preference (Microsoft Application Consulting and Engineering Team, 2003). Prior to execution they are translated into a language called Microsoft Intermediate Language (MSIL) where they can run on the Common Language Runtime (CLR) (Microsoft Application Consulting and Engineering Team, 2003). The CLR operates as a stand-in for the Windows kernel, and handles activities such as memory management, exception handling, garbage collection, memory type safety, and compiles the MSIL code into native machine instructions using a Just-In-Time (JIT) compiler (Microsoft

Application Consulting and Engineering Team, 2003). The CLR enables applications to be coded once, and automatically run on any platform which supports the .NET framework, providing a certain level of platform independence (Microsoft Application Consulting and Engineering Team, 2003). But it also provides a performance advantage because it limits the need for interpretation required in classic ASP (Microsoft Application Consulting and Engineering Team, 2003). Multiple components of the CLR may affect the applications performance including the JIT compiler, the garbage collector, the structured exception handler, threading, security, the loader, metadata, remoting (which supports calls between application domains), debugger, and Interop (which supports calls to various types of unmanaged code such as COM or DLLs that were written outside of .NET) (Microsoft Application Consulting and Engineering Team, 2003).

There are numerous ways in which an application can perform poorly at the CLR level. For example, if too many objects are created, memory management can suffer (Meier, Vasireddy, Babbar, & Mackman, 2004). Resources can be managed poorly by failing use dispose methods, which results unnecessary finalization of resources (Meier, Vasireddy, Babbar, & Mackman, 2004). Unmanaged resources which are not released can result in delays in reclaiming these resources, or create what is known as resource leaks (Meier, Vasireddy, Babbar, & Mackman, 2004). Failing to use the CLR's self-tuning thread pools by creating new threads on each request can also result in bottlenecks (Meier, Vasireddy, Babbar, & Mackman, 2004).

When running performance tests, there are numerous metrics that can be used to evaluate performance which are known in the Windows / .NET environment as counters. For example, counters which provide valuable information about CLR garbage collection performance include "# GC Handles" which indicates the current number of garbage collection handles, "# Total Committed Bytes" which indicates the amount of memory committed to the application, and "% Time in GC" which indicates how much time the garbage collector is spending collecting and compacting memory (Microsoft Application Consulting and Engineering Team, 2003). Counters which are useful in examining threading include ".NET CLR LocksAndThreads\# of current physical Threads" which indicates the number of threads the application is using, "Thread\% Processor Time" which helps determine which thread is actually using the

processor at a significant level and which threads waiting or deadlocked, "Thread\Context Switches/sec" which indicates which threads are causing high context switching rates, and "Thread\Thread State " which also indicates whether a particular thread is consuming a disproportionate level of resources (Meier, Vasireddy, Babbar, & Mackman, 2004). CLR loading can be measured with counters such as "Total AppDomains" which indicates how many application domains are currently loaded (in general, it is better to have fewer domains loaded to minimize context switches), "Total Assemblies" which can indicate overworked system resources when the assemblies are created or destroyed, and "Total Classes Loaded" which may indicate an unnecessary number of resource intensive instantiations (Microsoft Application Consulting and Engineering Team, 2003).

This is not an exhaustive list of useful performance counters; there are hundreds more which may be useful depending on the circumstance. It is important to be aware of them when planning performance tests since some measurements may be particularly relevant to the project's concerns, and others may be helpful when tuning a specific portion of an application to help determine the cause of memory leaks, bottlenecks and resource drains.

Server performance

The application portion of a software system resides on one or more physical or virtual machines. For a desktop application this machine would be the desktop client, but for web-based and other client-server software, much of the application resides on one or more servers (Multitier architecture, 2009). As mentioned previously, for .NET applications this typically means Internet Information Services on a machine running one of Microsoft's server operating systems, such as Windows Server 2003 (Internet Information Services, 2009). Numerous applications may be hosted on a single server or set of servers, and if an application uses a disproportionate amount of resources on this server(s), it can have a negative impact on other applications which rely upon them (Meier, Vasireddy, Babbar, & Mackman, 2004).

Applications make use of hardware resources on the server such as the CPU, memory, and devices which are connected through an I/O bus such as the physical disk drive (Meier, Vasireddy, Babbar, &

Mackman, 2004). Applications may also make use of operating system resources such as the Win32 subsystem, encryption subsystem, or security subsystem (Friedman, 2005). While calls to the operating system and hardware from the application are often abstracted by .NET class libraries provided by Microsoft, there are numerous operating system preferences which can impact software performance (Microsoft Application Consulting and Engineering Team, 2003). Additionally, problems at the server level such as reaching the maximum amount of virtual memory may indicate memory leaks, and can be addressed at the application responsible (Friedman, 2005).

There are also steps which can be taken to improve performance at the server level. For example, if a specific application is found to use a disproportionate amount of processor resources, the server could be upgraded, or alternatively the application could be moved to another server or cluster of load balanced servers (Friedman, 2005). Excessive amounts of disk activity can be caused by increased amounts of memory paging, and can potentially be remedied with increased RAM (Friedman, 2005).

The performance counters discussed in the previous section are handled by the Windows operating system, so it is no surprise that there are numerous counters which can help measure server performance. For example, the processor can be monitored using the "Processor\% Processor Time" counter, which indicates the percentage that the processor is in use at a given point in time (Meier, Vasireddy, Babbar, & Mackman, 2004). The counter "Processor\% Privileged Time" indicates the amount of time that the processor is spending in privileged mode calling operating system functions, and a steady level of 75% or higher would merit investigation (Meier, Vasireddy, Babbar, & Mackman, 2004). The counters "% Interrupt Time" and "Interrupts/sec" indicates how much of the time normal thread execution is halted to service hardware interrupts from devices such as the system clock, the mouse, disk drivers, network interface cards and other peripheral devices (Microsoft Application Consulting and Engineering Team, 2003). Counters which can help monitor memory usage include "Memory\Available Mbytes" which indicates the amount of memory available for allocation, "Memory\Cache Faults/sec" which can indicate issues with caching, and "Memory\Pages/sec" which can indicate excessive paging (Volodarsky, et al., 2008). Counters that are useful for determining physical disk performance include "PhysicalDisk\Avg. Disk Queue Length" which

indicates the number of queued read or write requests, and "PhysicalDisk\Avg. Disk sec/Transfer" which can indicate slowness, disk fragmentation, or disk failures (Meier, Vasireddy, Babbar, & Mackman, 2004).

Network performance

Web applications rely upon networks to transmit information between the client, application server, database server and web server (Multitier architecture, 2009). Networks allow connected computers to perform numerous tasks such as share files and printers, connect to remote applications, and remotely administer systems (Tulloch & Tulloch, 2002). Software performance can be impacted by the network types such as internet, local area networks (LANs), metropolitan area networks (MANs) and wide area networks (WANs) (Tulloch & Tulloch, 2002). They can also be impacted by numerous types of topologies that may be used such as Ethernet, Token Ring, and Fiber Distributed Data Interface (FDDI), or the various types of hardware used such as bridges, hubs, switches and routers (Tulloch & Tulloch, 2002). Even the type of cabling used in network segments can have a large impact on software performance (Tulloch & Tulloch, 2002).

When considering the effect of networks on software performance, one should begin by considering the various types of network connections that may be used by clients, and in connecting the different servers that make up the software system (Subraya, 2006). Connection types vary widely in bandwidth, which is the amount of information that can be carried by a signal or technology (Tulloch & Tulloch, 2002). While the connection type and bandwidth of clients is not under the control of developers, it should be considered along with the expectations of users (Hasan & Tu, 2003). Latency can be caused when two nodes on a network are significantly separated geographically, or when packets have to travel a large number of hops to reach their destination (Microsoft Application Consulting and Engineering Team, 2003). Similarly, the number of times that data must travel back and forth across the network (referred to as round trips) can impact the overall performance of an application, and can often be controlled by developers as well as network and system administrators (Microsoft Application Consulting and Engineering Team, 2003).

There are numerous counters that can be used in performance tests to determine the level of network activity. Some of these include "Network Interface\Bytes Total/sec" which indicates the amount of data being transmitted and received per second by the network interface (referred to as throughput), "Network Interface\Bytes Received/sec" which could indicate the need for optimization or additional network interfaces to handle client requests, and protocol specific counters such as "Protocol_Object \Segments Received/sec" or "Protocol_Object \Segments Sent/sec" if multiple protocols are in use by your application (Meier, Vasireddy, Babbar, & Mackman, 2004).

Database performance

Microsoft .NET applications can use a variety of database services including ODBC compatible data sources, Oracle, MySQL, and others, but in this paper, we will focus on the Microsoft SQL Server family of DBMSs. Software performance can be impacted by the way applications connect to databases, and configuration and tuning within the database and database server.

Access to databases such as those stored in Microsoft SQL Server 2005 may be accomplished through the built-in class libraries of ADO.NET, and LINQ. ADO.NET is a set of core class libraries included in every version of the .NET framework commonly used to access, query and store information in relational database systems (ADO.NET, 2009). LINQ is a newer technology provided in .NET 3.5 which allows for native querying of data providers such as SQL, XML, and others (Language Integrated Query, 2009). Performance may compare favorably or unfavorably between ADO.NET and LINQ depending on use case, so it may be worthwhile to consider this when developing data access code (Language Integrated Query, 2009). For the purposes of this paper we will focus on ADO.NET

When designing data access code, it is important to take into consideration the software architecture (whether it is a desktop, web based, or mobile application), how the users expect to receive data (such as instantly when the data is generated or through synchronization), and how the users expect to update data (including whether they may need to edit the data offline) (Hasan & Tu, 2003). To effectively test the

performance of ADO.NET, it is important to consider how much connection pooling is occurring, the response time and efficiency of queries, how quickly and effectively indexes can be searched, cache utilization levels, and the impact of locking at the table and row levels (Meier, Vasireddy, Babbar, & Mackman, 2004). Generally it is a good practice to use the appropriate data access object and data types specific to the RDBMS system, take advantage of connection pooling, helper technologies and centralized data access functions, whenever possible make use of complex stored procedures without multiple retrievals, and properly handle data access related exceptions (Hasan & Tu, 2003).

In addition to considerations at the application level, tuning and optimizing the database itself or even the database server can have a significant performance impact. For example, assuring that the server's processor is adjusted for the best performance of background services rather than programs can contribute positively to SQL Server's performance, and the application overall (Wort, et al., 2008). Performance may also improve or be degraded by enabling hyper-threading on the servers processor. Additionally, the maximum degree of parallelism settings can be adjusted if the default configuration is causing excessive wait (Wort, et al., 2008). Database performance can be impacted by normalization of the schema, or in cases where high degrees of reporting and data access are occurring de-normalization (Wort, et al., 2008). Database performance can also be improved by assuring that the appropriate data types are specified in the schema (Wort, et al., 2008).

There are a number of performance counters which can be gathered to analyze database performance. The level at which database locks are impacting performance can be determined by examining the "Lock Requests/sec", "Lock Timeouts/sec", "Lock Waits/sec", and "Number of Deadlocks/sec" counters (Microsoft Application Consulting and Engineering Team, 2003). Connection pooling factors can be analyzed using the ".NET CLR Data\SqlClient: Current # connection pools", ".NET CLR Data\SqlClient: Current # pooled connections", ".NET CLR Data\SqlClient: Peak # pooled connections", and ".NET CLR Data\SqlClient: Total # failed connects" counters (Meier, Vasireddy, Babbar, & Mackman, 2004). Problems with SQL server can also be analyzed with many of the server counters involving CPU, memory and physical disk access (Wort, et al., 2008).

Presentation performance

Depending on the type of application, the presentation layer can also impact software performance. For .NET web applications, the presentation is handled by ASP.NET components such as web forms, user controls, code-behind objects and web services that are called by the web forms and controls, and client side scripts (Hasan & Tu, 2003). These components work together to provide the user with a dynamic user interface that can rival many desktop software interfaces (frequently referred to as 'thick clients') (Microsoft Application Consulting and Engineering Team, 2003).

There are numerous ways in which performance can be improved at the presentation layer in an ASP.NET application. For example, the following settings in the machine.config file can impact how the ASP.NET worker process handles timeouts, request queues, threads and memory allocation: Timeout, IdleTimeout, ShutdownTimeout, RequestLimit, RequestQueueLimit, RestartQueueLimit, MemoryLimit, ClientConnectedCheck, ResponseDeadlockInterval, responseRestartDeadlockInterval, MaxWorkerThreads, and MaxIOThreads (Hasan & Tu, 2003). Performance of state management functions in ASP.NET applications can also be improved by storing state on the client using cookies, query strings and hidden controls where possible, and by optimizing the serialization process when storing state remotely (Meier, Vasireddy, Babbar, & Mackman, 2004). Other common presentation bottlenecks can be avoided by limiting page size and images, disabling SSL wherever possible, and using .NET classes such as StringBuilder for common concatenation functions (Microsoft Application Consulting and Engineering Team, 2003).

There are numerous performance counters that can assist with analyzing ASP.NET requests such as "Anonymous Requests", "Request Bytes In Total", Request Bytes Out Total", "Requests Executing", "Requests Timed Out", and "Requests/Sec", (Hasan & Tu, 2003). The performance of the worker process can be effectively measured with the "Worker Process Restarts" counter, and the cache can be measured with counters such as "Cache Total Entries", "Cache Total Hit Ratio", "Cache Total Turnover Rate", "Cache API Hit Ratio", "Cache API Turnover Rate", "Output Cache Entries", "Output Cache Hit Ratio" and "Output Cache Turnover Rate" (Meier, Vasireddy, Babbar, & Mackman, 2004).

Basic principles of virtualization

In the first chapter, some risk factors associated with software performance were listed, and prominent literature was examined that indicated a trend among recommended approaches from performance testing experts that favors a performance driven approach to development to better address those risks. Some apparent challenges to the instruction and adoption of performance driven development approaches were considered, and it was hypothesized that the development of a virtual testing laboratory accompanied by some instructional material could help overcome these challenges. In the second chapter, an overview was given of the common concerns in the domain of performance testing, specifically Microsoft ASP.NET web applications. This chapter will explore virtualization technology, and how it can help enable development projects to commence with performance testing earlier in the development lifecycle, and support iterative execution of performance tests to inform the development and tuning processes.

Introduction to virtualization

A simple definition of virtualization is the abstraction of any technology from its physical form or boundaries (Wolf & Halter, 2008). By this definition, a voice mail system can be considered a virtualized answering machine but in IT, it frequently refers to software which reproduces the functionality and structure of desktop computers or servers, but it can also refer to technology used to reproduce networks, storage structures and other technology (Buytaert, et al., 2007)

In a virtual machine infrastructure, one or more virtual machines (referred to as "guests") run independently and concurrently on another machine (referred to as a "host"), essentially like a "computer within a computer" (Campbell & Jeronimo, 2006). Conversely, more than one physical machine can be used as a cluster to host one or more virtual machines (Wolf & Halter, 2008). Virtualization functions by either emulating the various physical aspects of a computer (CPU, memory, disk drives, and I/O ports such as the LAN interface) or by listening for instructions which reference a physical device and re-directing these

instructions to the underlying hardware on the host machine (Wolf & Halter, 2008). Virtual machines are isolated from one another similarly to two computers on the same network, and therefore can communicate between each other as though they weren't sharing the same hardware resources (Campbell & Jeronimo, 2006). In some forms of virtualization, the minimal amount of software is installed to emulate a particular hardware structure, often to facilitate running a desired application or set of applications on a hardware platform or chipset that it wasn't intended for (Golden, 2008). In others, the entire operating system is virtualized to allow the user to perform any function or adjust almost any setting supported by the operating system in its native environment. (Golden, 2008).

A brief history of virtualization

Modern virtualization technology has its roots as far back as 1950s during the mainframe era when the University of Manchester's Atlas system had used virtualization to handle activities which exceed the physical limitations of installed memory (Campbell & Jeronimo, 2006). This led to IBM's M44/44X Project which contained an IBM 7044 (M44) scientific computer and several simulated 7044 virtual machines (Buytaert, et al., 2007). While this architecture did not fully simulate the underlying hardware for the virtual machines, it proposed the notion (which was proven in later projects) that the virtual machines were as efficient as their hardware counterpart, and for the first time the term "virtual machine" was coined in describing this architecture (Buytaert, et al., 2007).

IBM and MIT researchers used the architecture developed in this project to create timesharing systems to allow more than one mainframe user to simultaneously operate the computer without impacting one another, and in the late 1960s successfully built the first virtual machine operating system on fully virtualized hardware (Buytaert, et al., 2007).

While virtualization became well accepted in the 1970s as a way for multiple users to share expensive resources, hardware level virtualization experienced a decline in importance in the 1980s and 1990s as personal computers and workstations became more inexpensive (Goldworm & Skamarock, 2007). Nevertheless, virtualization of another sort emerged in the middle of the 1990s with the Java Virtual

Machine (Campbell & Jeronimo, 2006). In order to facilitate platform independence, the JVM redirects application instructions for physical devices to the associated native operating system and hardware devices. Later, Microsoft.NET's common language runtime performed similar activities (Campbell & Jeronimo, 2006)

In the 2000s, organizations began to reconsider virtualization for datacenter consolidation (Goldworm & Skamarock, 2007). As servers began to accumulate, consolidation presented value by reducing the amount of electricity consumed, and the reducing the need for cooling (Campbell & Jeronimo, 2006). Also, it was determined that while applications may not be suitable for sharing servers, there is not a consistent level of usage of RAM, CPU or physical disks, and resources were underutilized (Goldworm & Skamarock, 2007).

Common business cases for virtualization

The technical benefits to virtualization are commonly understood in numerous areas such consolidation, reliability, and security (Buytaert, et al., 2007). In data center consolidation efforts, organizations can run numerous virtual machines on a single physical server, thereby increasing server utilization, more efficiently using electricity and cooling systems, and reducing the amount of idle CPU and unused RAM (Golden, 2008). One evaluation determined that by virtualizing five two-way application servers, an organization could realize over $58 thousand over three years (Buytaert, et al., 2007). Similarly, the same server can host virtual machines of various operating systems, reducing the need to deploy a new physical machine to run applications in their targeted platform (Buytaert, et al., 2007). Consolidation also improves IT operations by reducing the number of staff required to maintain physical servers, and making IT more responsive to business needs (Golden, 2008). Reliability is improved through the isolation of software faults, and by allowing for failover partitions on a dedicated or as-needed basis (Campbell & Jeronimo, 2006). Security is improved through virtualization because digital attacks are contained through fault isolation, and because security settings such as super-user accounts can be applied specific to the virtual machine rather than to the entire host (Buytaert, et al., 2007).

Uses for virtualization in performance tests

While all of the business justifications in the prior section are valid, they are all on the assumption that virtualization will be used to create persistent resources. Performance testing can benefit more by creating transient resources. For example, if we know that the software is going to be run on a Windows 2008 server with 4 processors and 8 GB of RAM, we don't need to have such a machine built. We can log on to a host and create this machine, and run the tests. When the machine is no longer needed, it can easily be re-purposed if that is desired, or delete it to free up system resources. Also, because aspects such as RAM and dedicated CPU are software settings rather than installed hardware, it is easy to run a performance test on that system, and then try the same test with a machine with 2 processors and 4 GB of RAM to see how they compare.

Secondly, because the systems are virtualized, there are fewer consequences to worry about. While a group of developers may all have virtual servers they are testing, if one developer or tester accidentally puts their machine into an endless loop or crashes it, it won't affect the other developers. If one developer needs three servers to run a few tests on, the virtual and transient nature simplifies the procurement process, and when those machines are not running the other developers and testers can utilize the unused RAM, CPU and other hardware resources.

Another advantage is aptly referred to by one writer as "Rinse and Repeat" virtualization (Campbell & Jeronimo, 2006). In this technique, machines are reverted after test execution into the state they were in prior to executing the test (Campbell & Jeronimo, 2006). This permits the developer or tester to experiment with different settings or modifications to test data without concern to consequences, or needing to rebuild test machines (Campbell & Jeronimo, 2006). This reversion to prior state is automatically supported in numerous virtualization software systems (Campbell & Jeronimo, 2006) . In VMWare, a user may take a "snapshot" of a machine's current state and revert at any point (Campbell & Jeronimo, 2006). In Microsoft's virtualization products, these are called "undo files" (Campbell & Jeronimo, 2006).

Considerations and challenges

While the value of virtualization both in consolidating servers and in building a test lab are clear, there are some important considerations to make when building a virtual infrastructure. The first consideration is in licensing (Golden, 2008). Some vendors such as Oracle charge a license fee for each processor on the server, and if the software is installed on a host with 50 processors but only two processors are being used by Oracle, in theory all 50 processors require licenses (Golden, 2008). Other software limits the number of concurrent users, so having this software running on VMs that are running but not in use would be inadvisable.

Another consideration is in the allocation of resources. Users familiar with physical machines may have a "more is better" mentality, and thus feel that when they build a machine they should allocate 4 processors "in case it's needed", but over-allocating RAM, CPU or disk space can actually decrease the performance of a VM (Petri, 2009). On the reverse end, one may think that since they are trying to run numerous machines concurrently that RAM should be minimized on each machine, but this can result in increased physical disk activity which is a more severe bottleneck (Petri, 2009)

Significant vendors

While there are a large number of vendors of virtualization solutions, perhaps the three most prominent product lines are from Microsoft, VMWare, and XenWorks. In this section, an appropriate product from each vendor will be discussed, followed by a discussion of which product will be used in the performance test lab built in support of this research.

Microsoft offers several virtualization products, but for running a lab environment the most likely choice would be Microsoft Virtual Server 2005. This was created as a key part of Microsoft's Dynamics Systems Initiative to reduce costs and streamline IT operations (Buytaert, et al., 2007). Microsoft believes that decoupling application workload from hardware allows for the rapid deployment of new systems and migration between physical servers when workload needs change (Buytaert, et al., 2007). This is further enhanced by numerous available tools which simplify surrounding processes, such as Microsoft's migration

toolkit which simplifies the conversion of physical servers to virtual (Dittner, Green, Grotenhuis, Majors, Rule Jr., & ten Seldam, 2006).

VMWare was founded in 1998, and released its first product VMWare Workstation in 1999 (Muller, Wilson, Happe, & Humphrey, 2005). This product is intended to run virtual machines on a desktop system, and is more useful for occasional (typically non-concurrent) use of an alternative operating system or machine configuration (Muller, Wilson, Happe, & Humphrey, 2005). For running a test lab, the more likely solution would be either its GSX Server or ESX Server software (Muller, Wilson, Happe, & Humphrey, 2005). These run much faster and are suitable for running numerous machines concurrently. VMWare is a widely adopted virtualization solution with more than 3 million registered users, and VMWare as of 2005 claimed to already be running 5% of the world's servers on the VMWare platform (Muller, Wilson, Happe, & Humphrey, 2005).

Xen is a virtualization system offered commercially by Xenworks, and as open source from the Xen community (Chaganti, 2007). It is different from VMWare and Microsoft's solutions because it uses a technique called paravirtualization, which provides virtualization as a modified version of the host's operating system without hardware emulation (Chaganti, 2007). The advantage to this is that the VMs do not incur the performance overhead required for hardware emulation (Chaganti, 2007). The disadvantage is that without hardware emulation, VMs are limited to being of the same operating system as their host (Chaganti, 2007).

For the purpose of this research paper, VMWare's ESX Server software will be used as the host. There are two reasons for this decision. First, ESX Server is currently widely adopted, and can be installed free on a system of up to 4 processor cores. This makes it a good choice for students. Second, ESX Server is what is currently installed in Regis University's SCIS lab, as well as in my workplace, making it much more simple to get a lab set up and running. Nevertheless, it is worthwhile to be aware of these alternative products because Microsoft's test tools are tightly integrated with their virtualization solution, and because Xen's paravirtualization technology may be useful in a more homogeneous environment.

Performance testing in Visual Studio 2008

In the first chapter, some risk factors associated with software performance were listed, and prominent literature was examined that indicated a trend among recommended approaches from performance testing experts that favors a performance driven approach to development to better address those risks. Some apparent challenges to the instruction and adoption of performance driven development approaches were considered, and it was hypothesized that the development of a virtual testing laboratory accompanied by some instructional material could help overcome these challenges. In the second chapter, an overview was given of the common concerns in the domain of performance testing, specifically Microsoft ASP.NET web applications. The third chapter explored virtualization technology, and how it can help enable development projects to commence with performance testing earlier in the development lifecycle, and support iterative execution of performance tests to inform the development and tuning processes.

This chapter will provide an overview of a prominent toolset for performance testing Microsoft .NET applications, namely Microsoft's own Visual Studio 2008 Team System (VSTS) Tester Edition. In this chapter, the three types of automated test classes in VSTS 2008 (unit tests, web tests, and load tests) will be discussed along with how they can be used to execute performance tests

Unit testing

Unit testing should be a familiar concept to any developer, particularly one familiar with test-driven development. Test-driven development aids programmers, analysts, testers and other participants in the development process by expressing requirements unambiguously in the form of tests which are automated, and then executed iteratively to assure that the requirements are met initially, and that regression errors have not been introduced with subsequent modifications and integrations (Newkirk & Vorontsov, 2004). Testing performed by programmers to validate a module of code is a unit test (Kumar & Kumar, 2008). Unit tests in

.NET are classes created using a framework such as NUnit, and may be written in a variety of languages, but it is often useful to write them in the same language as the code being tested (Newkirk & Vorontsov, 2004).

VSTS 2008 Tester Edition contains a built-in framework for unit testing (Randolph & Gardner, 2008). Unit tests can be created from scratch in VSTS 2008 Test Edition by selecting "New Test" from the "Test" menu, and then selecting "Unit Test" from the subsequent dialog, as illustrated in the figures below.
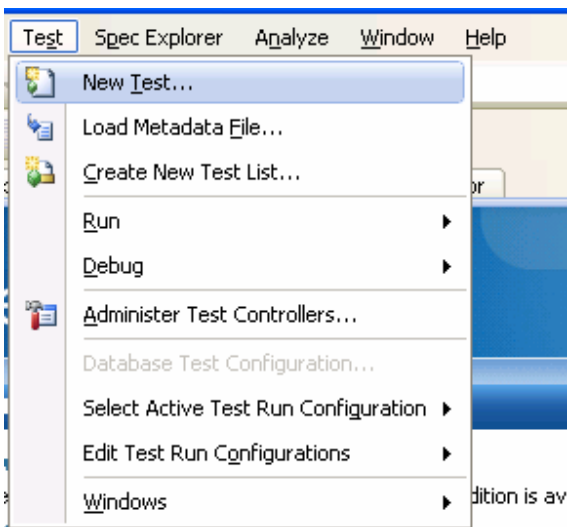
Figure 1
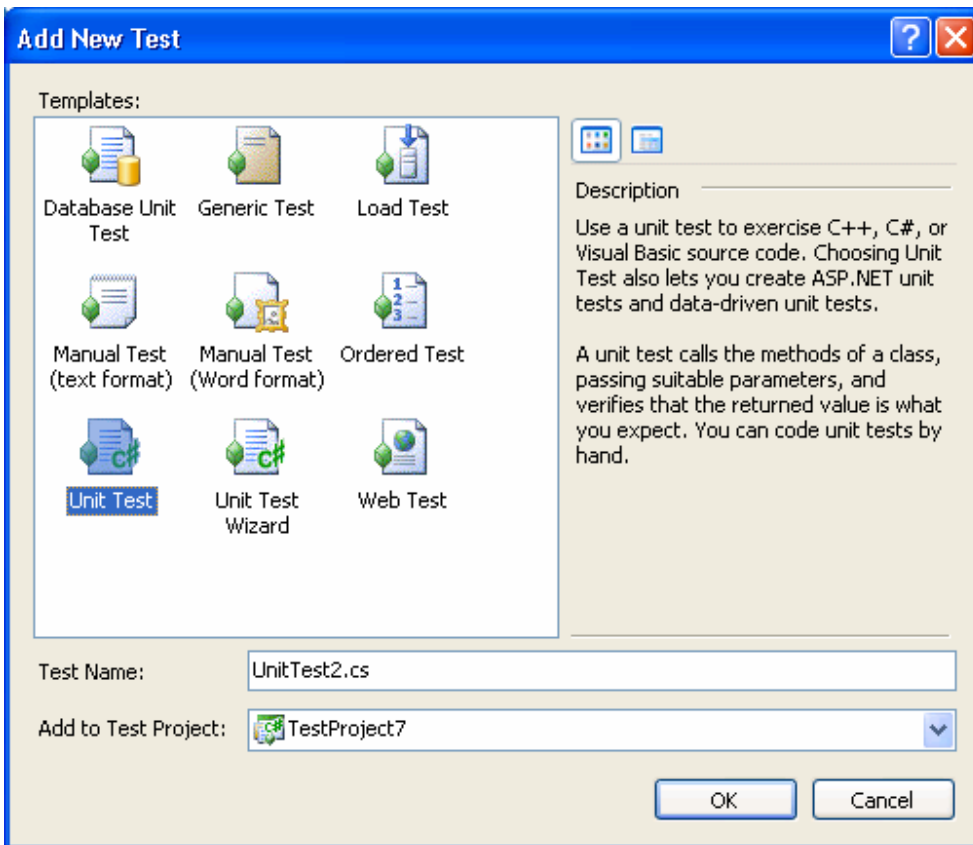
Figure 2

Alternatively, unit tests can be created for a specific class, method or for all classes in a project by right-clicking in the desired context, and selecting "Generate Tests". This brings up a subsequent dialog from which the user can select which classes or methods to generate unit tests for. Whichever method is used to generate the unit test, it is automatically coded to result as inconclusive (and thus failed) when executed (Randolph & Gardner, 2008). The tests reside within a TestClass, and individual test cases are contained in a TestMethod, and have a number of configurable attributes including "Description", "Owner", "Priority" which determines the order in which the test is executed when all tests are run, "WorkItem" which maps the unit test to tracking systems like Team Foundation Server, and "Timeout" which sets the amount of time in which a unit test must complete before failing (Randolph & Gardner, 2008).

In a unit test, code is written that asserts whether the test code operates as required (Kumar & Kumar, 2008). The unit testing framework facilitates this through a set of assertion functions, and in VSTS, these functions are as follows: "AreEqual()" which determines whether a set of values are equal,

"AreNotEqual()" which determines whether they are not equal, "AreNotSame()" and "AreSame()" which verify referential equality as opposed to value equality, "IsTrue()" and "IsFalse()" which verify the result of a boolean comparison, "IsNull()" and "IsNotNull()" which verify the instantiation state, and "IsInstanceOfType()" and "IsNotInstanceOfType()" which determines whether an object type is correct (Bunn & Plenderleith, 2009). Additionally, there are two functions that over-ride assertion without condition, and these are "Fail()", and "Inconclusive()" (Bunn & Plenderleith, 2009). Unit tests can be used to verify that a method's return value falls within specified boundaries, are equal in some way, are in the proper format or culture, and that a method's exception path is properly followed (Levinson & Nelson, 2006).

Because unit tests are simply sequences of code which assert conditions based on return values, they can be leveraged to do much more than validate a code unit against requirements. For example, unit tests can be written which interact with databases through ADO.NET, consume web services, or interact with outside systems or unmanaged codes through interop DLLs. As a performance tester, I have used unit tests to initiate and evaluate transactions using Microsoft's Sync Framework, test Microsft SQL databases and stored procedures, and even performance test Lotus Notes databases through a reference to a DLL called Domino.Interop.

As discussed in later sections, unit tests can leveraged in load tests for performance testing purposes. But there are two obvious ways in which unit testing functionality can support performance driven development. First, the timeout property can provide a mechanism (albeit crude) to assure a level of performance before a test passes (Randolph & Gardner, 2008). Additionally, unit tests can be specified as a build condition, so if code is modified which compiles successfully but causes a unit test to fail or exceed the specified timeout, the solution will not build (Meier, Taylor, Mackman, Bansode, & Jones, 2008).

Web testing

The second type of automated testing tool provided by VSTS 2008 Test Edition is the web test. Web tests allow testers and developers to record test cases using a web interface in a web browser (Levinson & Nelson, 2006). Web tests are created using the same "Test\New Test" function detailed in the prior section,

and when a new test is created a web browser window is opened with a panel used to control recording, as illustrated below.
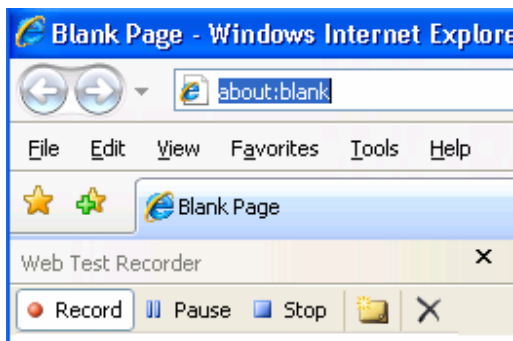
Figure 3

The "Record" button starts a recording, and once this is clicked each action performed in the web browser is listed below the control as a step. The recording is paused using the "Pause" button, and can be resumed by clicking on the "Record" button again. Clicking the "Stop" button terminates the recording session and returns the user to the Visual Studio IDE. The button next to "Stop" is used to add a comment to the test, and the button next to that can be used to delete the steps that have been previously recorded. To record a test case, the browser can be used in the same way that an end user would use the application. The valid HTTP and client activity performed during a recording, such as retrieving a web page, entering values into fields, or interacting with AJAX controls are all captured and included in the test (Randolph & Gardner, 2008).

One a test has been recorded and the "Stop" button has been clicked, the test is opened in the Web Test Editor (Kumar & Kumar, 2008). This shows all of the recorded steps in a tree view, and exposes properties and parameters associated with each step (Kumar & Kumar, 2008). In this view, extraction rules can be created which capture session variables to assure that the test can be executed again (Kumar & Kumar, 2008). Extraction can be accomplished from text including that matching a regular expression on the web page, HTML form fields, attributes, or header contents, and even hidden fields (Levinson & Nelson, 2006). Extraction rules can also be used to bind the field values on a form submitted in that step to a data source (Randolph & Gardner, 2008). Validation rules can also be created for a step to determine whether the response page is as expected (Kumar & Kumar, 2008). These rules can be based on a form field, text

contained on the page, the amount of time the request is completed within or the existence of a required attribute value or HTML tag (Levinson & Nelson, 2006).

While much of the necessary actions can be accomplished using the Web Test Editor, the test can be converted into code by using the "Generate Code" function in the context menu, and web tests can even be created entirely from code rather than being recorded (Levinson & Nelson, 2006). Using coded web tests can facilitate test in which an action should be performed in a loop, or should respond to a particular event or condition (Levinson & Nelson, 2006).

Web tests can provide a number of useful purposes to a development effort. They can be used to validate and verify a functional requirement for an application, test the application's usability, determine that the security model is properly configured for a set of sample users, verify the application's compatibility with a number of browsers and versions, and determine how well a web application functions for users of various network configurations (Kumar & Kumar, 2008).

Load testing

The third type of automated test in VSTS 2008 Test Edition is the load test. Load tests execute existing tests such as unit tests or web tests iteratively or concurrently to determine how well the application will perform under heavy use (Kumar & Kumar, 2008). The concurrent users can be executing the same test or different tests, and users can be set to different network settings (such as LAN, dial-up and DSL), and different browsers (Kumar & Kumar, 2008). Load tests are created using the same "Test\New Test" function detailed in the prior section, and when a new test is created, a wizard appears which guides through setting up the test (Randolph & Gardner, 2008).

The first section of the wizard determines scenario details, which specify the aspects of the actual user tests being simulated (Kumar & Kumar, 2008). The wizard asks whether recorded think time (in the case of a web test, the amount of time between steps when the test was recorded) should be used, a random think time normalized to the recorded think time, or no think times should be used (Randolph & Gardner,

2008).   The wizard then asks about the load pattern, which is how many concurrent virtual users are to be simulated, and whether they are to be applied at constant level, or whether they should be added in gradually (Kumar & Kumar, 2008). The next part of the scenario section is the test mix, and here the wizard determines which tests are to be executed, and how they are to be distributed (Levinson & Nelson, 2006). The last parts of the scenario section determine the distribution among virtual users of the browser being used, and the network connections (Randolph & Gardner, 2008).

The next section of the wizard is the counter sets, and here the machines which should be monitored are added, and the performance counters that should be tracked in each of these machines are specified (Kumar & Kumar, 2008). By default, the controller and agents (machines used to generate load) are added, but if the proper access has been granted, VSTS can also automatically grab performance counters for application servers, web servers, database servers, and other parts of the software system being tested (Levinson & Nelson, 2006).

The third section of the new load test wizard determines the run settings (Randolph & Gardner, 2008). These settings determine the maximum duration of the test, and how often performance counters are gathered (Kumar & Kumar, 2008). One important setting in this section is the validation level, which determines whether the load test will invoke validation rules that were specified in a web test (Levinson & Nelson, 2006).

Once this wizard has completed, the load test has been created, and can be either modified or executed. The load test appears in the Load Test Designer, and from here run settings can be added or modified, counters can be added or deleted, and tests can be added or removed from the test mix, among other modifications (Randolph & Gardner, 2008). To begin execution of a load test, the user clicks the button circled in the figure below.
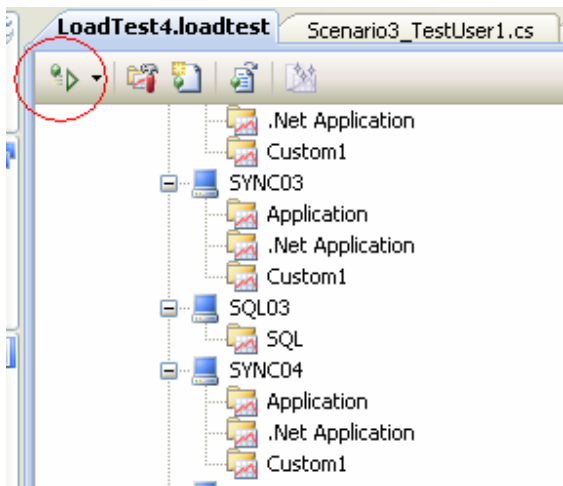
Figure 4

When a test has begun executing, the view switches to the Load Test Monitor in Graphs view (Randolph & Gardner, 2008). The graphs show the results of samples from the counters previously specified, and can be modified through options such as "Select Graph", "Show Legend", "Show Plot Points", "Show Horizontal Grid Lines", "Show Min/Max Lines", "Show Threshold Violations", "Display data for the entire run or recent data only" (which is available only while the run is in progress) (Levinson & Nelson, 2006). Load test results can also be loaded from the load test repository, which is a SQL database which stores the run details and performance counter samples from completed tests (Levinson & Nelson, 2006). The results repository database is configured through the menu command "Test\Administer Test Controllers", and it is in this dialog box where load controllers and load agents are configured and administered (Kumar & Kumar, 2008).
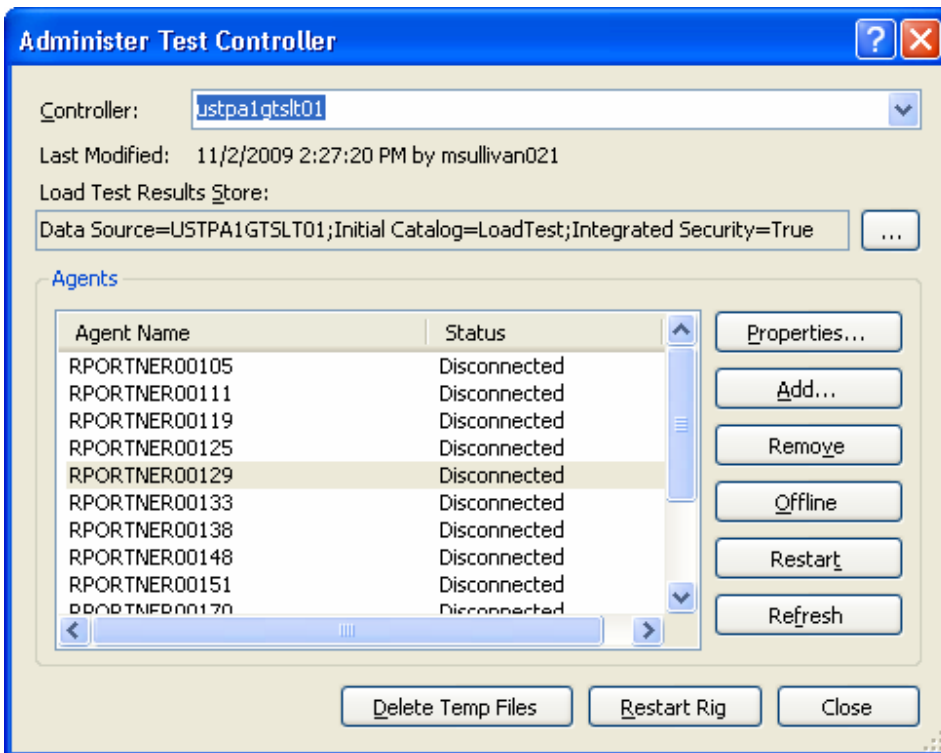
Figure 5

In order to set a rig (which is a load controller and collection of associated load agents), the LocalTestRun.testrunconfig must be edited (Levinson & Nelson, 2006). This is a file which is automatically added when load tests are created, and can be found in the Solution Explorer. Aside from allowing for remote test execution, this file also can be used for advanced configurations and functions, such as code coverage analysis, and setup and cleanup scripts (Levinson & Nelson, 2006).

Performance testing

In the context of VSTS 2008, a load test is used for performance testing purposes, and in this way the two terms are distinct. In the field of performance testing, the two terms are often used interchangeably, along with other terms. In this section, different types of performance tests will be listed, along with a brief discussion of how they may be conducted using the three types of automated tests described in previous sections.

In *Performance Testing Guidance for Web Applications: Patterns & Practices* the authors included a useful table of types of performance tests (see Appendix C: Summary Matrix of Performance Testing

Types). The authors also included a table of terms that they do not consider to be types of performance tests, but are frequently associated with the field (see Appendix D: Other Performance Related Tests and Activities). Finally, they included these very useful tables which map the common risk factors associated with software performance to performance test types which help identify the application's vulnerability and exposure (see Appendix D: Summary Matrix of Performance Testing Types by Risks Addressed, and Appendix F: Summary Matrix of Risks Addressed by Performance Testing Types).

With this information in hand, the question becomes how to create or conduct each of these types of performance tests in Visual Studio. Using one of the previously included tables as a guide, what follows is a description of how one might accomplish them.

Test Type	How to conduct in Visual Studio Team System 2008 Test Edition
Performance test	Once the necessary performance characteristics are modeled, create unit and web tests to validate each of these characteristics. Include the unit and web tests in a load test, and execute the tests starting at the baseline number of virtual users, and gradually incrementing the number of users every five minutes. When the maximum number of users has been reached, continue the execution for one hour, and then gradually decrease the number of users to zero.
Load test	This is the basic type of performance test. Simply create the required unit or web tests, and include them in a load test at whatever desired duration and concurrency level.
Endurance test	Once the necessary performance characteristics are modeled, create the required unit or web tests to validate the function(s) where endurance is a concern. Include them in a load test that is set up to increase the number of virtual users to the target level over a 30 minute timeframe. Once the target number of users has been reached, continue to execute the tests for an extended period of time, such as two days.
Stress test	Once the necessary performance characteristics are modeled, create the required unit or web tests to validate the function(s) that should be stressed. Include them in a load test that is set up to increase the number of virtual users to the target level over a 30 minute timeframe and set the maximum number of users to a number which far exceeds the expected level of concurrent users. Setup the test to continue running for an hour when it has reached the maximum, and then gradually decrease the number of users to zero.
Spike test	Once the necessary performance characteristics are modeled, create the required unit or web tests to validate the function(s) that should be stressed. Include them in seven load tests and set the odd numbered tests up to run at the baseline level of load for a moderate amount of time (perhaps 45 minutes) and set the even numbered tests up to quickly increase the virtual users to an extreme level of load for a short period of time (perhaps 15 minutes). Run all seven tests in order.
Capacity test	Once the necessary performance characteristics are modeled, create the required unit or web

Test Type	How to conduct in Visual Studio Team System 2008 Test Edition
	tests to validate the function(s) that should be stressed. Include them in a load test that is set up to increment the number of virtual users by 1 per minute, and set the maximum number of users to a number which is certain to exceed the capacity of the system. Setup the test to continue running for an hour when it has reached the maximum, and then gradually decrease the number of users to zero. While executing the test, watch the performance counters for indications that the system has locked up or failed, and then terminate the test.

In this chapter, we examined the three basic types of automated tests in Visual Studio Team System 2008 Test Edition. We gave a brief introduction to how these tests are created, and what they are used for. We reviewed material from *Performance Testing Guidance for Web Applications: Patterns & Practices* which differentiated between types of performance tests and the risks they address. Finally, instructions were provided on how one could accomplish each of these types of tests using VSTS 2008 Test Edition.

The software performance testing lab

The objective of this thesis is to create a virtual environment where developers, testers, and software engineering students can learn how to create and execute performance tests, and use this knowledge to conduct experiments on their own software and systems. In this section, three implementations of such a virtual environment will be presented and compared. These implementations include one that was built for Regis University's Academic Research Network (ARNe), one that was constructed on a personal desktop server, and one that was constructed for a large corporation.

The Software Engineering Lab at Regis University's Academic Research Network (ARNe)

The first implementation of the Software Performance Testing Lab was at the software engineering lab at Regis University's Academic Research Network (ARNe). This lab host virtual machines for a variety of student projects and other purposes on a Linux server running VMWare's ESX software. The purpose in implementing this lab is to offer students and faculty an environment for learning and testing. In order to assure stability and security, the software engineering lab is offered and accessed through ARNe's Citrix portal. The architecture of the host and guest machines is illustrated in Figure 6.

Windows Development Performance Lab on a desktop server

The second implementation of the Software Performance Testing Lab was on a desktop server running VMWare's ESX operating system. This implementation is intended to demonstrate how an individual developer or tester can build this lab with minimal resources, and can be used to run experiments without worrying about impacting other users. The architecture of the host and guest machines is illustrated in Figure 7.

Windows Development Performance Lab in a large corporation's data center

The third implementation of the Software Performance Testing Lab was on a set of HP Proliant DL580 G5 servers running VMWare's ESX operating system. This implementation is intended to demonstrate how with additional hardware, SAN and network resources, a corporation can build this lab to run tests and experiments from multiple users. This implementation includes load controller and agents to increase the capacity for concurrent activity and load. The architecture of the host and guest machines is illustrated in Figure 8.
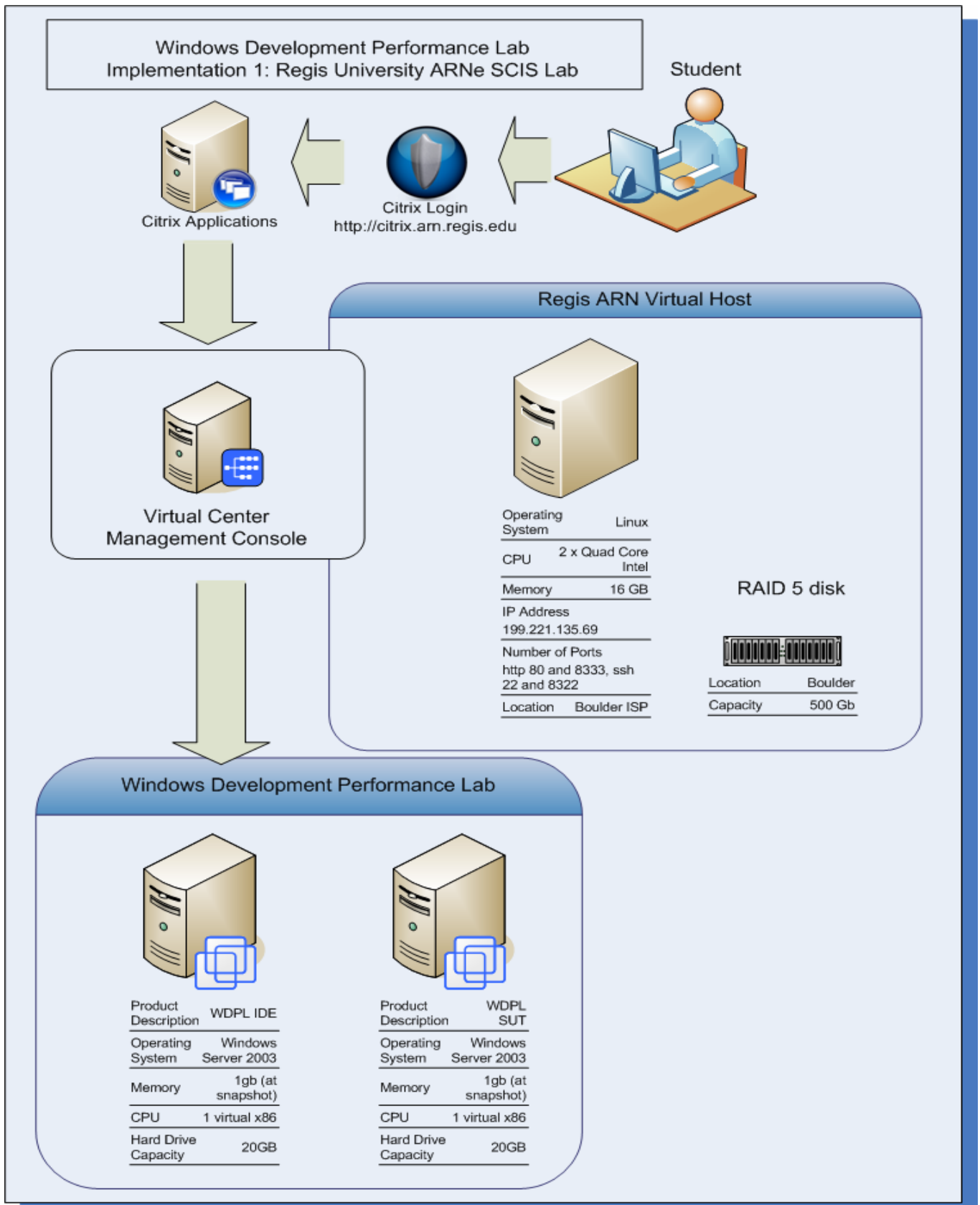
Figure 6
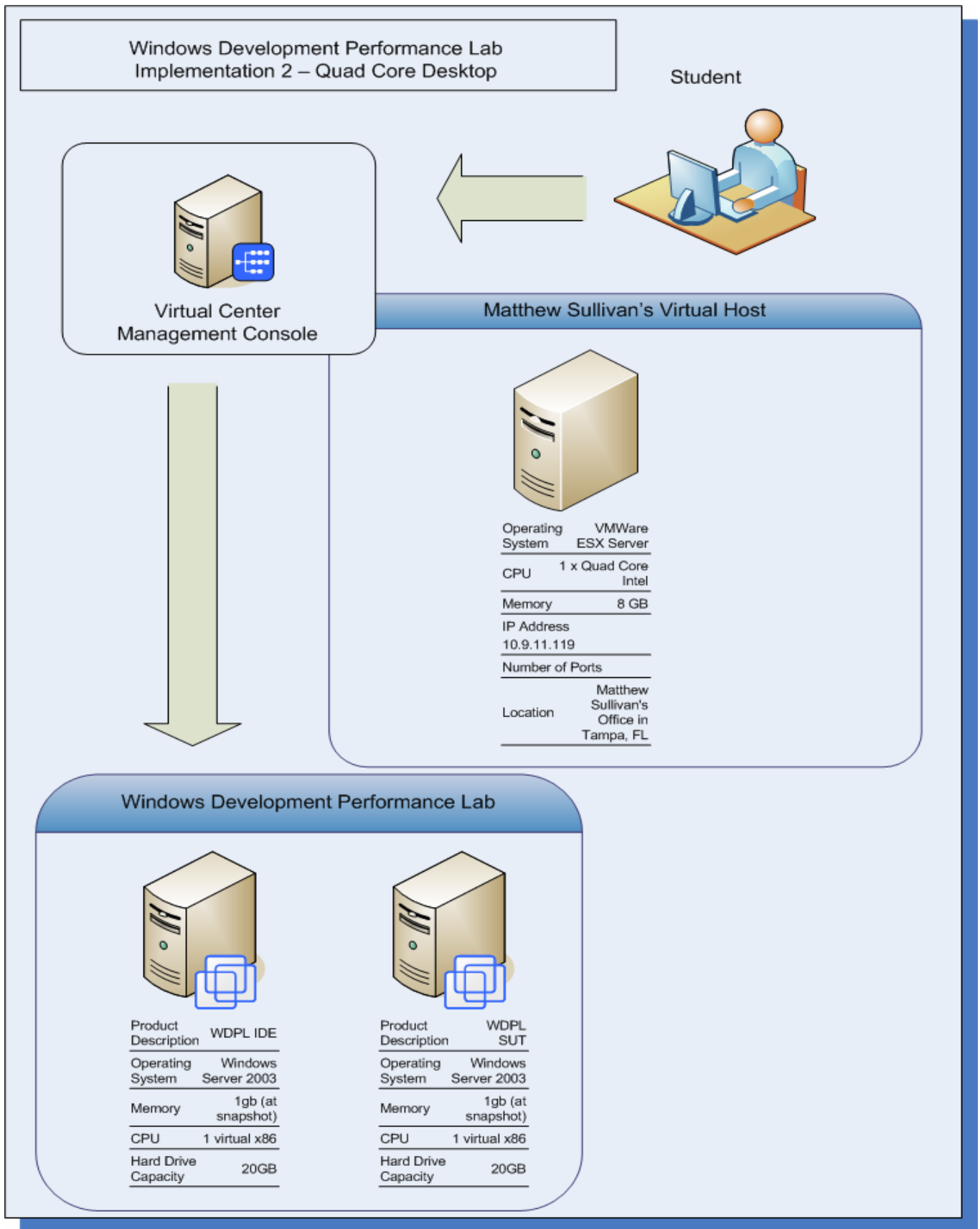
Figure 7
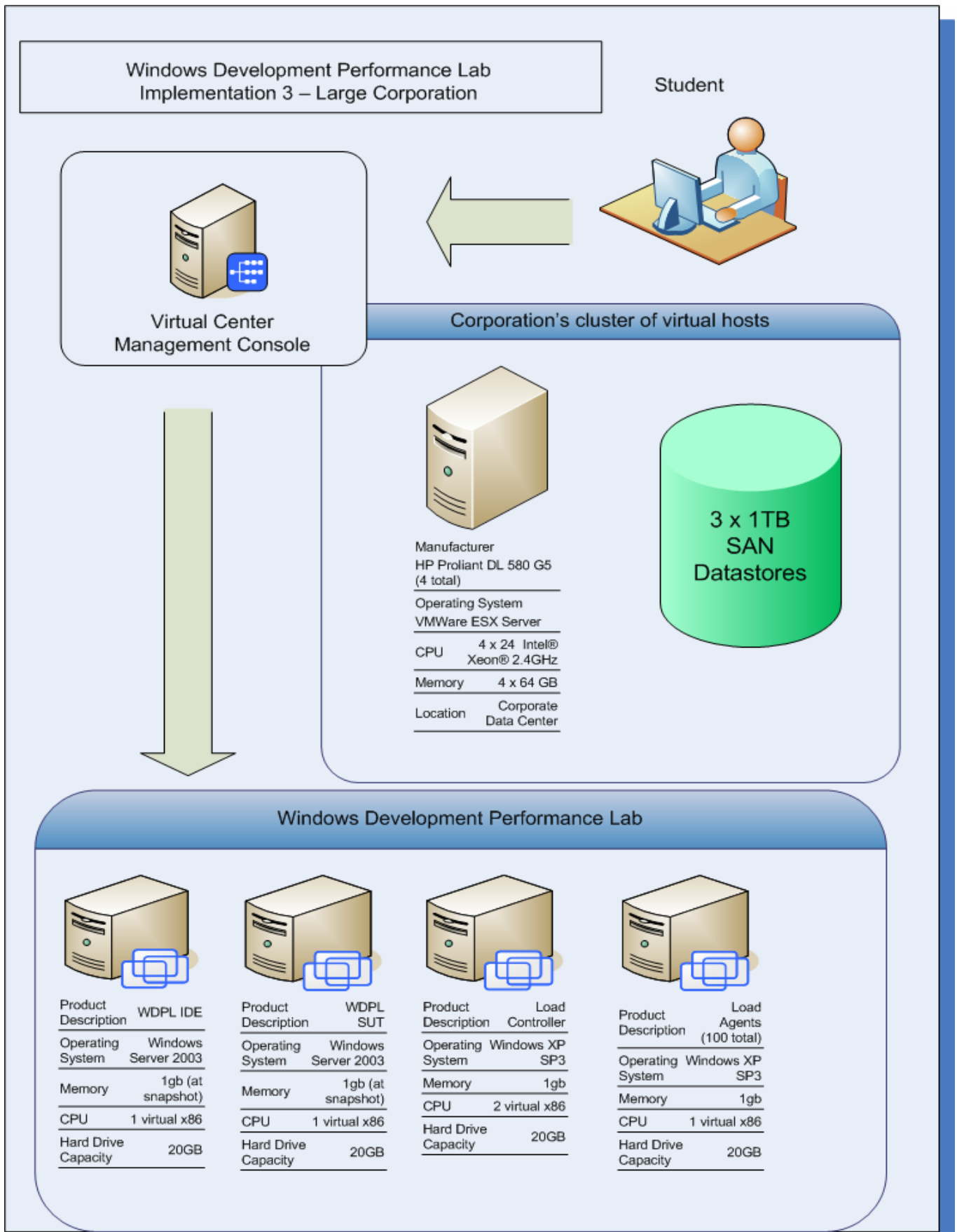
Figure 8

Guest architecture

In all three implementations, the Windows Development Performance Lab itself is accessed on two or more virtual machines on the host. The first machine is where the tests are created and executed on the machine labeled "WDPL IDE". This machine runs Windows Server 2003 operating system, has 1 GB of RAM (which can be changed at the host level), and a 20GB virtual hard drive. The additional software installed on this machine includes Microsoft Visual Studio Team System 2008 Test Edition, and Microsoft SQL Server 2008 Developer Edition. SQL Server is used for creating and manipulating test parameters and other database needs.

The second machine is "WDPL SUT", and this is only used for hosting a sample application which can be used for creating web tests. In the ARNe implementation, the sample application is Employee Information, which is a starter kit that was provided by Microsoft's ASP.NET web site. This application was not created or modified in the scope of this research, and any application or starter kit can be used in place of it. Aside from the sample application, the software installed on the machine includes IIS 6.0, ASP.NET 1 and 2, Microsoft Visual Studio Team System 2008 (which is only used to deploy the sample application), and Microsoft SQL Server 2008 Developer Edition, which is used to store sample data in the sample application.

In the large corporation's implementation, the lab also includes Windows XP virtual machines which are used as load controllers and load agents. These machines only require the Microsoft Visual Studio Team System 2008 Load Controller and Microsoft Visual Studio Team System 2008 Load Agent software respectively. The load controller distributes compiled test code to the agent machines for execution of tests and collects performance data from the agents for results reports, and the load agent executes the tests that are sent to it. These systems do not typically need to be directly accessed when executing tests, and are only there to add additional "horsepower" to the tests.

Self-evaluation of constructed solution

The intent of this research and project was not necessarily to discover any previously unknown body of knowledge about software performance, but rather to construct a forum for these discoveries. This forum consists of a simple but useful performance testing lab, along with material on how to use it. In this section, this solution is evaluated.

Research paper

The scope of this research could be considered a bit broad, as it covers all aspects of performance of .NET applications, Microsoft performance testing tools, and VMWare virtualization. Nevertheless, I believe it successfully provides for the fundamental needs to embark on a study of performance using the constructed lab. While any of the three subjects of performance, testing tools, and virtualization could be expanded much further, the essentials of each are collected and presented in such a way where the subject is cohesive and relevant.

Windows Development Performance Lab

The performance test lab created at Regis University is not particularly complex. It only contains two virtual machines, and the minimal software required for the performance testing conducted in the presentations and guided labs. This is by design because the SCIS VMWare server is fairly constricted in disk space, and other resources. The alternative implementations are offered to demonstrate what could be done with more or less resources. The virtual lab could be expanded to include other types of Microsoft servers, database servers, and other systems to be tested. The same is true of other platforms, and it could be useful to construct a parallel virtual lab for Java platform applications, and other platforms.

Another enhancement to this project could be made by the implementation of a more mature version of the web site created as a test case for the guided labs. While the web site is currently just a coarsely

constructed modification to the Club Site 2.0 starter kit from Microsoft's http://www.asp.net web site, a central portal where students and other parties could reserve access to the performance lab and share the results of their experiments could be worthwhile.

PowerPoint series

To help motivate the construction of the PowerPoint deliverables for this project, I arranged to have them presented at my workplace. I believe this was an effective strategy for getting them completed, and obtaining feedback to improve them. With these experiences and the feedback received from them in hand, a fair assessment could be as follows.

The presentations are cohesive and include the required information to embark upon performance testing and study. Originally, the first two presentations were combined, but I found it to be a bit too long, and a couple attendees agreed. I believe that splitting the presentation into one containing an overview on performance and one making the case for performance driven development made the subjects of both more cohesive and digestible.

The series could be improved by making the slides less "text heavy". Nevertheless, I believe that they achieve the objective of presenting the fundamental information, knowledge and skills required to perform the guided labs.

Guided labs included in the series

The biggest challenge in creating the guided labs that are included with the PowerPoint series was striking a balance between providing an overwhelming amount of detail, and providing too little to complete a meaningful activity. My approach was to be fine-grained when showing how to do something for the first time, and then coarse-grained when instructing the reader to repeat an activity. For example, in the web testing lab, I showed step by step how to create a test for one use case, but I left it to the student to figure out

the steps and procedures for the remaining use cases based on what we had already done. I believe that this strategy produced an effective but not overwhelming lab manual.

Overall assessment

In this research thesis, a strong case was made for improving performance in the pursuit of better software and more value for information technology. I maintain that this project represents an effective solution to numerous problems faced by organizations embarking on a study of software performance.

Conclusion

In this research paper, the fundamentals for a growing body of knowledge are presented. Although this research has produced this paper, a virtual lab, a series of six presentations and four guided labs, we have scarcely scratched the surface on the subject of software performance. To be truly effective in improving software performance in a substantive way, it would be necessary to be constantly researching, experimenting and learning. Hopefully this collection of knowledge and tools can start that process..

Appendices

Appendix A: Windows Development Performance Lab

Instructions

Accessing the Windows Development Performance Lab

1. Open Regis University's Academic Research Network Citrix portal and log in (http://citrix.arn.regis.edu/citrix/metaframexp/default/frameset.asp)

2. From the list of applications, select **VMWare Console.**

   (If you are unable to access the ARN Citrix portal, or do not have the VMWare Console in your list of applications, please go to http://help.arn.regis.edu).

3. When prompted to log in to the VMWare Console, enter **regisscis.net** for **Host name**, and your user name and password on the SCIS server.

4. In the **VMWare Console**, the lab can be accessed through the following virtual machines:

   Name: WDPL IDE
   Description: This machine is where tests can be constructed and executed
   Operating System: Windows2003
   User Name: Administrator
   Password: Adm1n123
   IDE: Visual Studio Team System 2008 Test Edition
   DB Engine: Microsoft SQL Server 2008 Developer Edition
   Lab Project: (Administrator)My Documents\Visual Studio 2008\Projects\PDDLabs
   Lab Database: (MSSQLSERVER)\dbo.PDDLabsTestParams

   Name: WDPL SUT
   Description: This machine acts as a web server hosting a sample application against which to test.
   Operating System: Windows2003
   User Name: Administrator
   Password: Adm1n123
   IP Address: 172.16.218.143
   (This is dynamically assigned, so if it doesn't work, use IPCONFIG to verify)
   IDE: Visual Studio Team System 2008 Basic Edition
   (Only used for deploying the application)
   DB Engine: Microsoft SQL Server 2008 Developer Edition

5. If the machines are not running, start them up to begin running the labs. If the machines are running, double check that another user is not using the lab.

6. When you are finished running the labs, please shut down the machine.

Note: These machines will be restored to snapshots periodically. If you need anything saved, please back it up to your local hard drive.

Accessing the Windows Development Performance Lab Presentations

1. Using the instructions above, access the Windows Development Performance Lab, and log on to the following virtual machine:

   Name: WDPL IDE
   User: Administrator
   Password: Adm1n123

2. On the desktop for that machine, there is a folder named Lab Presentations.

3. To view the presentation, you can launch any of the individual programs in PowerPoint Viewer, and you can print or view the presenter notes using Adobe Reader.

Lab 3.1 - Comparing specialized collections

Background

We are developing an application that performs a number of functions, one of which is to return the capital city for a given U.S. state. The architect would like to know which type of .NET specialized collection to use.

The following three types of collections are being considered:

```
System.Collections.Generic.Dictionary<TKey, TValue>
System.Collections.Specialized.ListDictionary
System.Collections.Hashtable
```

It is expected that there will be 200 concurrent users accessing the application.

Question

Which collection will perform fastest?

Approach

We will answer this question by creating three unit tests, one for each collection type. Each unit test will create the collection in an identical manner and will retrieve values in as similar a manner as possible, recognizing that collection members may vary slightly in implementation.

All three unit tests will be included in a load test which will use all three collections at the specified level of concurrency for a configured amount of time.

Instructions

1. Open up Microsoft Visual Studio Team System 2008 Test Edition.

2. Create a new project (see Figure 1).

   - For the project type, select **Visual C#\Test\Test Project**.
   - For the location, select **My Documents\Visual Studio 2008\Projects**.
   - For the name, specify **CollectionComparisonTest**_*[your name]*.
   - You may skip creating the project and tests by opening the "PDDLab" project at that location.
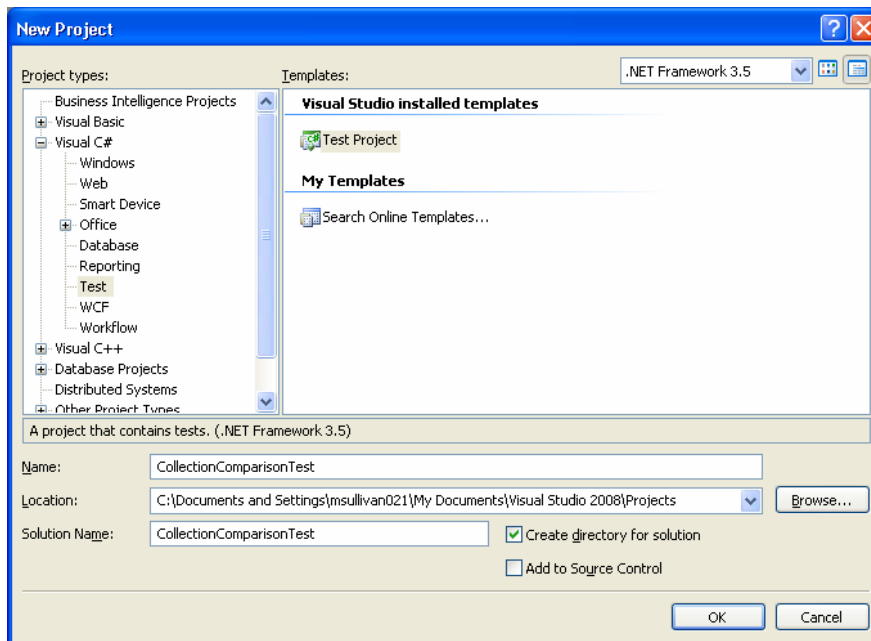
Figure 9

3. In your new solution, add a new project of type **Visual C#\Console Application**.
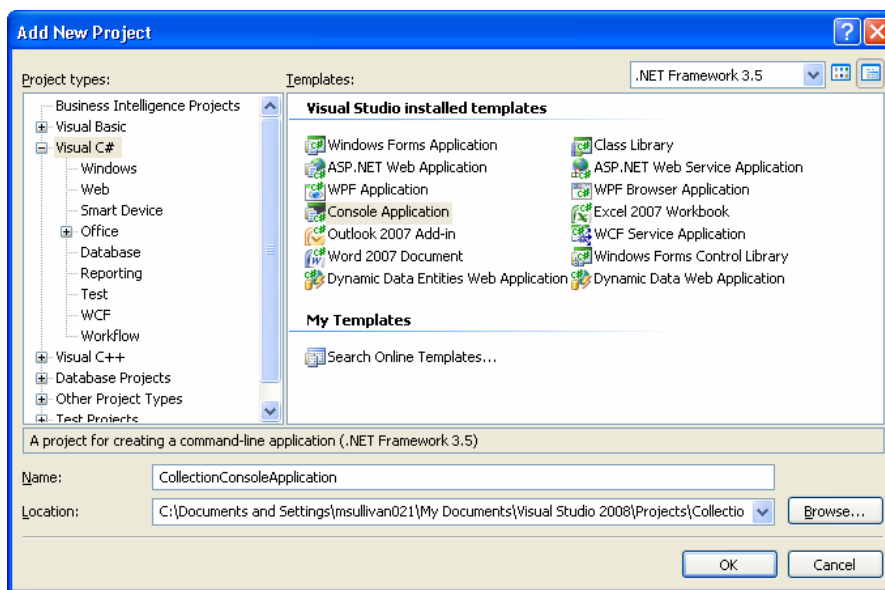
Figure 10

4. In the console application, add a new class called **StatesCapitalsHashTable.cs**, and input the following code:

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

namespace CollectionConsoleApplication
{
    class StatesCapitalsHashTable
    {

        Hashtable StateCapitals = new Hashtable();
```

```csharp
public StatesCapitalsHashTable()
{
    StateCapitals.Add("Alabama", "Montgomery");
    StateCapitals.Add("Alaska", "Juneau");
    StateCapitals.Add("Arizona", "Phoenix");
    StateCapitals.Add("Arkansas", "Little Rock");
    StateCapitals.Add("California", "Sacramento");
    StateCapitals.Add("Colorado", "Denver");
    StateCapitals.Add("Connecticut", "Hartford");
    StateCapitals.Add("Delaware", "Dover");
    StateCapitals.Add("Florida", "Tallahassee");
    StateCapitals.Add("Georgia", "Atlanta");
    StateCapitals.Add("Hawaii", "Honolulu");
    StateCapitals.Add("Idaho", "Boise");
    StateCapitals.Add("Illinois", "Springfield");
    StateCapitals.Add("Indiana", "Indianapolis");
    StateCapitals.Add("Iowa", "Des Moines");
    StateCapitals.Add("Kansas", "Topeka");
    StateCapitals.Add("Kentucky", "Frankfort");
    StateCapitals.Add("Louisiana", "Baton Rouge");
    StateCapitals.Add("Maine", "Augusta");
    StateCapitals.Add("Maryland", "Annapolis");
    StateCapitals.Add("Massachusetts", "Boston");
    StateCapitals.Add("Michigan", "Lansing");
    StateCapitals.Add("Minnesota", "Saint Paul");
    StateCapitals.Add("Mississippi", "Jackson");
    StateCapitals.Add("Missouri", "Jefferson City");
    StateCapitals.Add("Montana", "Helena");
    StateCapitals.Add("Nebraska", "Lincoln");
    StateCapitals.Add("Nevada", "Carson City");
    StateCapitals.Add("New Hampshire", "Concord");
    StateCapitals.Add("New Jersey", "Trenton");
    StateCapitals.Add("New Mexico", "Santa Fe");
    StateCapitals.Add("New York", "Albany");
    StateCapitals.Add("North Carolina", "Raleigh");
    StateCapitals.Add("North Dakota", "Bismarck");
    StateCapitals.Add("Ohio", "Columbus");
    StateCapitals.Add("Oklahoma", "Oklahoma City");
    StateCapitals.Add("Oregon", "Salem");
    StateCapitals.Add("Pennsylvania", "Harrisburg");
    StateCapitals.Add("Rhode Island", "Providence");
    StateCapitals.Add("South Carolina", "Columbia");
    StateCapitals.Add("South Dakota", "Pierre");
    StateCapitals.Add("Tennessee", "Nashville");
    StateCapitals.Add("Texas", "Austin");
    StateCapitals.Add("Utah", "Salt Lake City");
    StateCapitals.Add("Vermont", "Montpelier");
    StateCapitals.Add("Virginia", "Richmond");
    StateCapitals.Add("Washington", "Olympia");
    StateCapitals.Add("West Virginia", "Charleston");
    StateCapitals.Add("Wisconsin", "Madison");
    StateCapitals.Add("Wyoming", "Cheyenne");

}
public string GetCapital(string state)
{
    string retCapital = "";

    if (StateCapitals.ContainsKey(state))
    {
        retCapital = StateCapitals[state].ToString();
    }
    return retCapital;
}

}

}
```

5. In the console application, add a new class called **StateCapitalsListDictionary.cs**, and input the following code:

```csharp
using System;
using System.Collections;
using System.Collections.Specialized;
using System.Linq;
using System.Text;

namespace CollectionConsoleApplication
{
    class StateCapitalsListDictionary
    {
        ListDictionary StateCapitals = new ListDictionary();
        public StateCapitalsListDictionary()
        {
            StateCapitals.Add("Alabama", "Montgomery");
            StateCapitals.Add("Alaska", "Juneau");
            StateCapitals.Add("Arizona", "Phoenix");
            StateCapitals.Add("Arkansas", "Little Rock");
            StateCapitals.Add("California", "Sacramento");
            StateCapitals.Add("Colorado", "Denver");
            StateCapitals.Add("Connecticut", "Hartford");
            StateCapitals.Add("Delaware", "Dover");
            StateCapitals.Add("Florida", "Tallahassee");
            StateCapitals.Add("Georgia", "Atlanta");
            StateCapitals.Add("Hawaii", "Honolulu");
            StateCapitals.Add("Idaho", "Boise");
            StateCapitals.Add("Illinois", "Springfield");
            StateCapitals.Add("Indiana", "Indianapolis");
            StateCapitals.Add("Iowa", "Des Moines");
            StateCapitals.Add("Kansas", "Topeka");
            StateCapitals.Add("Kentucky", "Frankfort");
            StateCapitals.Add("Louisiana", "Baton Rouge");
            StateCapitals.Add("Maine", "Augusta");
            StateCapitals.Add("Maryland", "Annapolis");
            StateCapitals.Add("Massachusetts", "Boston");
            StateCapitals.Add("Michigan", "Lansing");
            StateCapitals.Add("Minnesota", "Saint Paul");
            StateCapitals.Add("Mississippi", "Jackson");
            StateCapitals.Add("Missouri", "Jefferson City");
            StateCapitals.Add("Montana", "Helena");
            StateCapitals.Add("Nebraska", "Lincoln");
            StateCapitals.Add("Nevada", "Carson City");
            StateCapitals.Add("New Hampshire", "Concord");
            StateCapitals.Add("New Jersey", "Trenton");
            StateCapitals.Add("New Mexico", "Santa Fe");
            StateCapitals.Add("New York", "Albany");
            StateCapitals.Add("North Carolina", "Raleigh");
            StateCapitals.Add("North Dakota", "Bismarck");
            StateCapitals.Add("Ohio", "Columbus");
            StateCapitals.Add("Oklahoma", "Oklahoma City");
            StateCapitals.Add("Oregon", "Salem");
            StateCapitals.Add("Pennsylvania", "Harrisburg");
            StateCapitals.Add("Rhode Island", "Providence");
            StateCapitals.Add("South Carolina", "Columbia");
            StateCapitals.Add("South Dakota", "Pierre");
            StateCapitals.Add("Tennessee", "Nashville");
            StateCapitals.Add("Texas", "Austin");
            StateCapitals.Add("Utah", "Salt Lake City");
            StateCapitals.Add("Vermont", "Montpelier");
            StateCapitals.Add("Virginia", "Richmond");
            StateCapitals.Add("Washington", "Olympia");
            StateCapitals.Add("West Virginia", "Charleston");
            StateCapitals.Add("Wisconsin", "Madison");
            StateCapitals.Add("Wyoming", "Cheyenne");
```

```
            }
            public string GetCapital(string state)
            {
                string retCapital = "";
                if (StateCapitals.Contains(state))
                {
                    retCapital = StateCapitals[state].ToString();
                }
                return retCapital;
            }

        }
    }
```

6. In the console application, add a new class called **StatesCapitalsDictionaryCollection.cs**, and input the following code:

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

namespace CollectionConsoleApplication
{
    class StatesCapitalsDictionaryCollection
    {
        Dictionary<String, String> StateCapitals = new
Dictionary<string, string>();
        public StatesCapitalsDictionaryCollection()
        {
            StateCapitals.Add("Alabama", "Montgomery");
            StateCapitals.Add("Alaska", "Juneau");
            StateCapitals.Add("Arizona", "Phoenix");
            StateCapitals.Add("Arkansas", "Little Rock");
            StateCapitals.Add("California", "Sacramento");
            StateCapitals.Add("Colorado", "Denver");
            StateCapitals.Add("Connecticut", "Hartford");
            StateCapitals.Add("Delaware", "Dover");
            StateCapitals.Add("Florida", "Tallahassee");
            StateCapitals.Add("Georgia", "Atlanta");
            StateCapitals.Add("Hawaii", "Honolulu");
            StateCapitals.Add("Idaho", "Boise");
            StateCapitals.Add("Illinois", "Springfield");
            StateCapitals.Add("Indiana", "Indianapolis");
            StateCapitals.Add("Iowa", "Des Moines");
            StateCapitals.Add("Kansas", "Topeka");
            StateCapitals.Add("Kentucky", "Frankfort");
            StateCapitals.Add("Louisiana", "Baton Rouge");
            StateCapitals.Add("Maine", "Augusta");
            StateCapitals.Add("Maryland", "Annapolis");
            StateCapitals.Add("Massachusetts", "Boston");
            StateCapitals.Add("Michigan", "Lansing");
            StateCapitals.Add("Minnesota", "Saint Paul");
            StateCapitals.Add("Mississippi", "Jackson");
            StateCapitals.Add("Missouri", "Jefferson City");
            StateCapitals.Add("Montana", "Helena");
            StateCapitals.Add("Nebraska", "Lincoln");
            StateCapitals.Add("Nevada", "Carson City");
            StateCapitals.Add("New Hampshire", "Concord");
            StateCapitals.Add("New Jersey", "Trenton");
            StateCapitals.Add("New Mexico", "Santa Fe");
            StateCapitals.Add("New York", "Albany");
            StateCapitals.Add("North Carolina", "Raleigh");
            StateCapitals.Add("North Dakota", "Bismarck");
            StateCapitals.Add("Ohio", "Columbus");
            StateCapitals.Add("Oklahoma", "Oklahoma City");
```

```
            StateCapitals.Add("Oregon", "Salem");
            StateCapitals.Add("Pennsylvania", "Harrisburg");
            StateCapitals.Add("Rhode Island", "Providence");
            StateCapitals.Add("South Carolina", "Columbia");
            StateCapitals.Add("South Dakota", "Pierre");
            StateCapitals.Add("Tennessee", "Nashville");
            StateCapitals.Add("Texas", "Austin");
            StateCapitals.Add("Utah", "Salt Lake City");
            StateCapitals.Add("Vermont", "Montpelier");
            StateCapitals.Add("Virginia", "Richmond");
            StateCapitals.Add("Washington", "Olympia");
            StateCapitals.Add("West Virginia", "Charleston");
            StateCapitals.Add("Wisconsin", "Madison");
            StateCapitals.Add("Wyoming", "Cheyenne");

        }
        public string GetCapital(string state)
        {
            string retCapital = "";
            if (StateCapitals.TryGetValue(state, out retCapital))
            {
                //do nothing
            }
            return retCapital;
        }

    }
}
```

7. At this point, you should have:

- Created a new Visual C# Test Project
- Added a console applications
- Added three classes to the console application, each of which perform the requested lookup function for one of the collections we are examining.

8. Open one of the classes you created and navigate to the **GetCapital()** method. Right click within that method, and select **Create Unit Tests**.
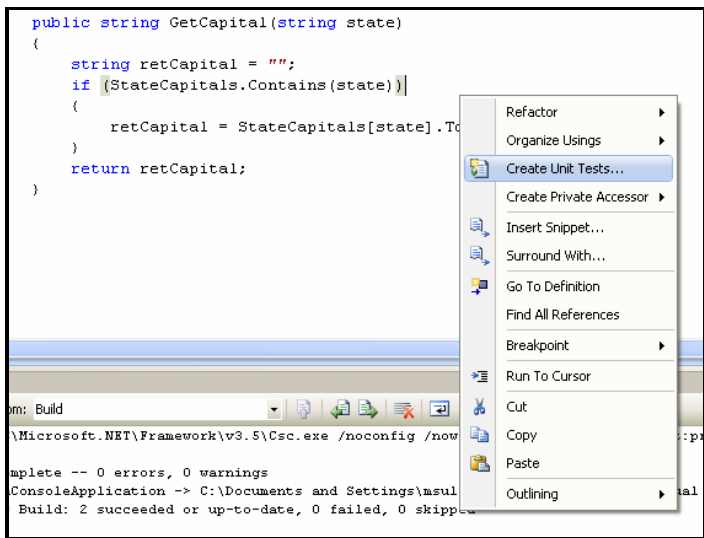
Figure 11

9. Select the **GetCapital()** method for each of the classes, and be sure to specify your test project as the **Output project**.
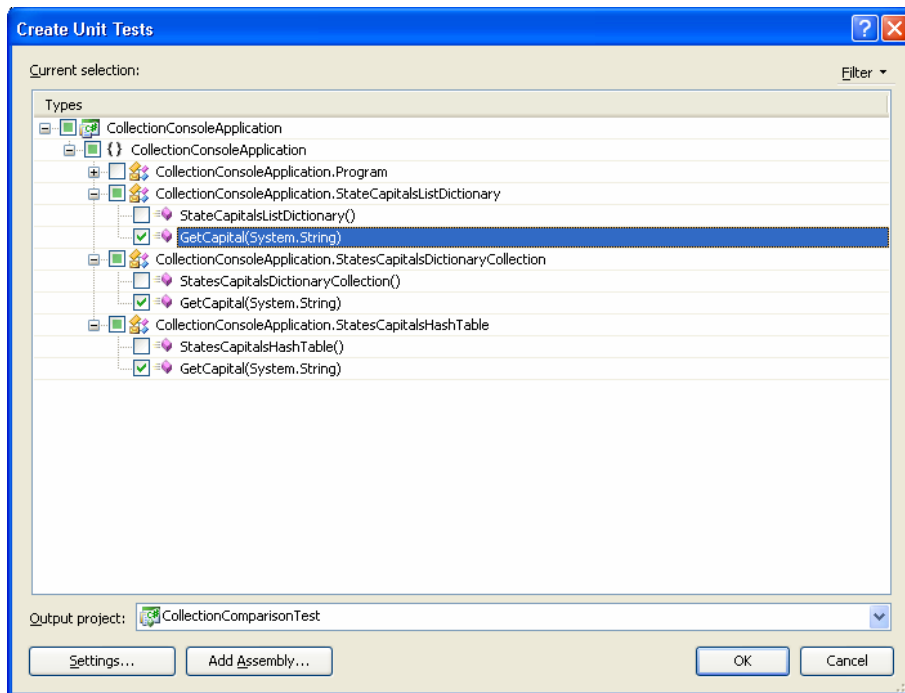
Figure 12

10. If you are prompted to add the **InternalsVisibleTo** attribute, click **Yes**.

11. At this point, you should have three new unit tests added to your project. Open **StatesCapitalsHashTableTest.cs** and navigate to the **GetCapitalTest()** method.

12. You will notice that the code is essentially a stub which needs to be replaced with valid test code. Replace that method code with the following:

```
[DataSource("System.Data.SqlClient", "Data Source=REGIS-FWVWSBM3F;Initial
Catalog=PDDLabsTestParams;Integrated Security=SSPI", "dbo.CollectionTest",
DataAccessMethod.Sequential), TestMethod()]
    public void GetCapitalTest()
    {
        TestContext.BeginTimer("Create Collection " + TestContext.TestName);
        StatesCapitalsHashTable target = new StatesCapitalsHashTable();
        TestContext.EndTimer("Create Collection " + TestContext.TestName);
        TestContext.BeginTimer("Get Test Data " + TestContext.TestName);
        string state = System.Convert.ToString(TestContext.DataRow["StateValue"]);
        string expected = System.Convert.ToString(TestContext.DataRow["CapitalValue"]);
        TestContext.EndTimer("Get Test Data " + TestContext.TestName);
        string actual;
        TestContext.BeginTimer("Get Value " + TestContext.TestName);
        actual = target.GetCapital(state);
        TestContext.EndTimer("Get Value " + TestContext.TestName);
        Assert.AreEqual(expected, actual);
    }
```

13. You may have noticed about this code:

- The test has been configured to use a data source. This allows the unit test to be data driven, which means that it will execute one time for every row of data available. If you are not running this test in the Windows Development Performance Lab, you will need to make sure that the data source is properly created, specified and configured.

- We have used the **TestContext** object to create timers around each significant activity. We named the timers by specifying the activity and the test being run. This allows us to differentiate between the amount of time generating the connection, pulling in test data, and retrieving data from the test database.

- We have renamed the test method to be specific about which collection is being tested. This will make it easier to differentiate between the tests when they are included in a load test.

14. Repeat this process for each of the other unit tests so that all three are pulling data from the data source, and the same timers have been established.

15. Now that we have created the classes that we are testing and unit tests for each of them, we will create a load test which includes these unit tests. Click on **Test-New Test** to open the **New Test** prompt, select **Load Test** as the test type, name the test **CollectionsLoadTest.loadtest** and click **OK**.
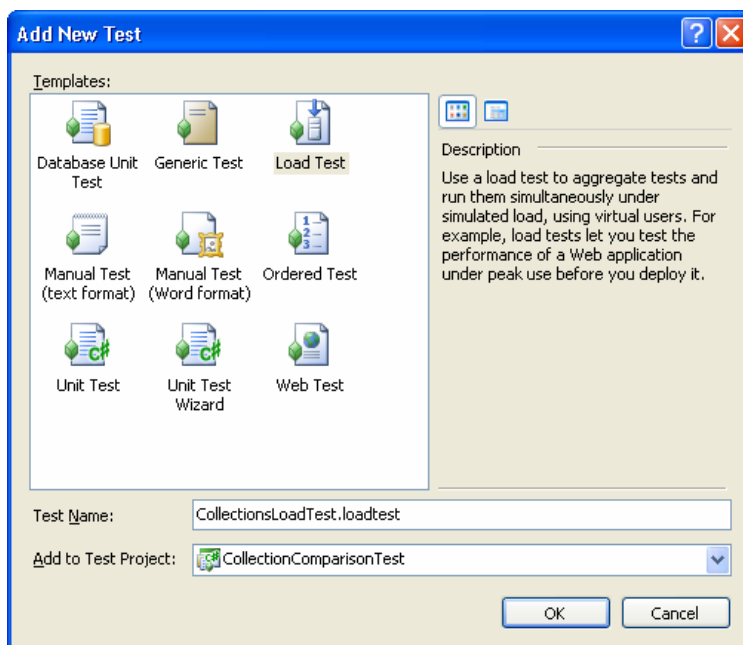
Figure 13

16. Click **Next** in the welcome screen, and specify 5 seconds as the **Think time between test iterations**.
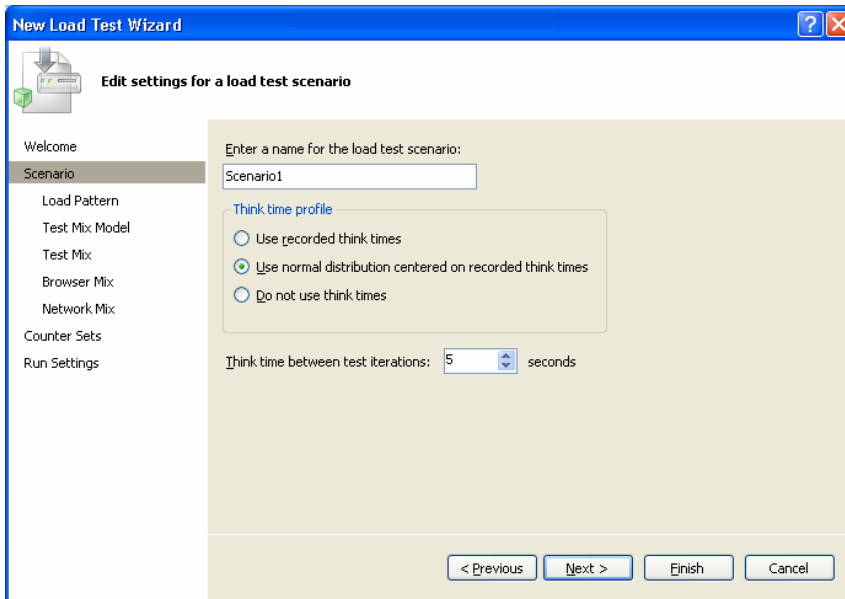
Figure 14

17. Specify a **Constant load** pattern of **300 users**.
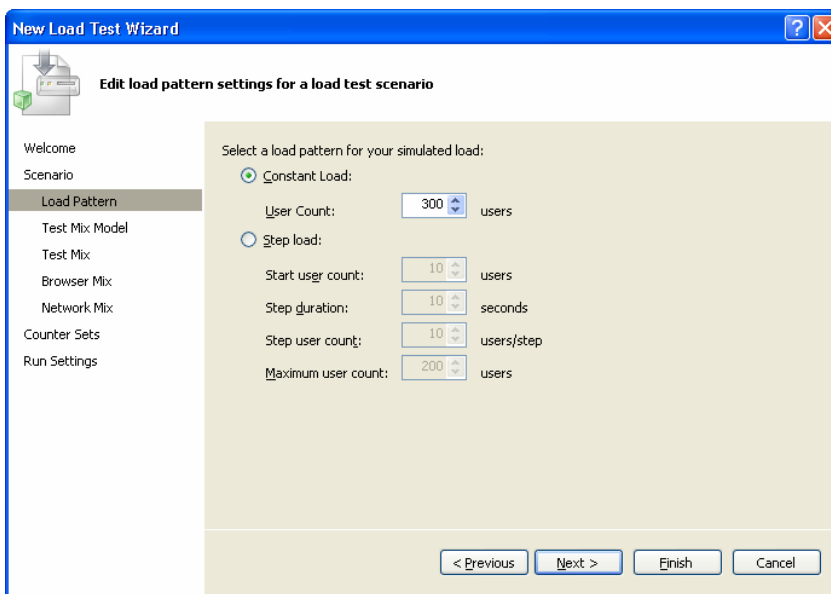
Figure 15

18. Model the test mix **Based on the total number of tests**.
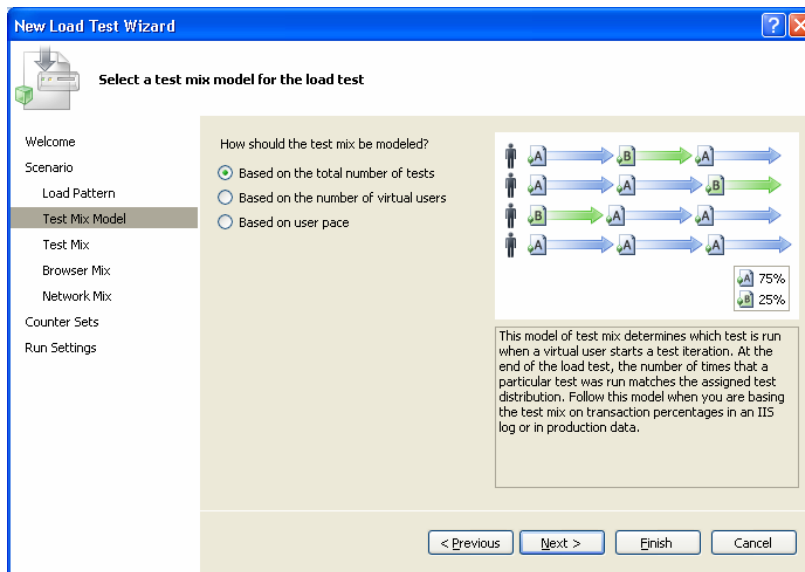
Figure 16

19. For the test mix, select the three unit tests you created.
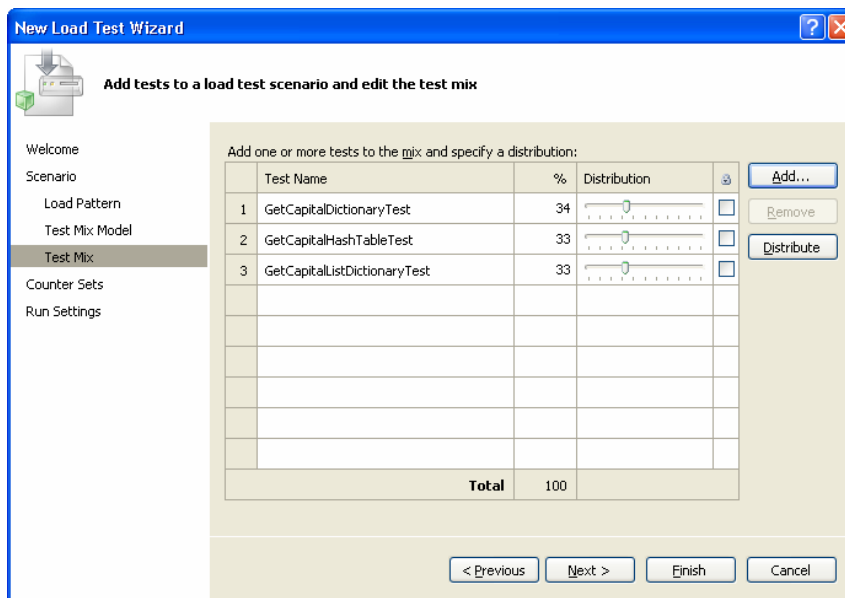
Figure 17

20. Click **Next** on the **Counter Sets** screen. Since we are executing this test locally, there are no other counters that we need to gather.

21. Specify 30 minutes for the **Run duration** in the Run Setting screen and click **Finish**.
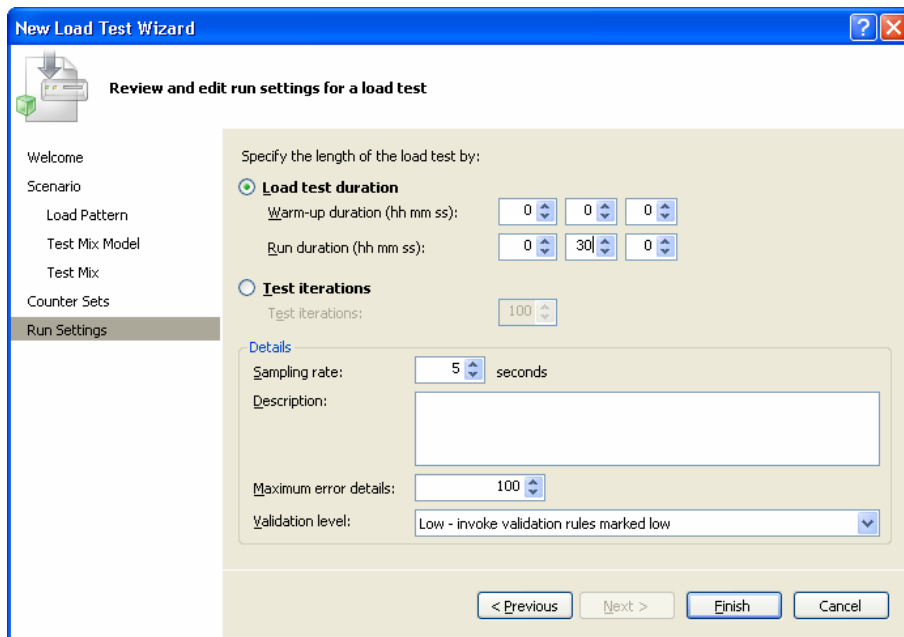
Figure 18

22. Now that you have created the classes we are testing, the unit tests, and the load test, execute the test and review the results. To begin the test, click the **Start** button (see Figure 11).
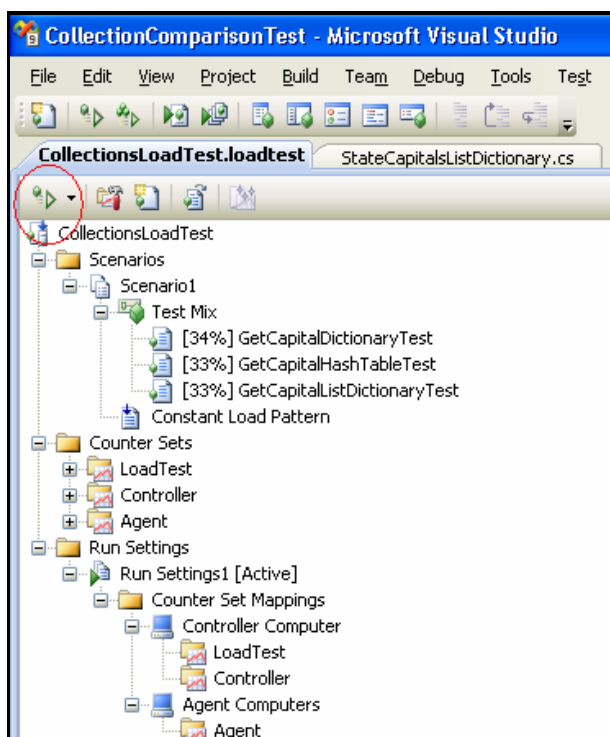
Figure 19

23. When the test run has completed, please refer to the results to complete the next section.

Results

Based upon your test results, which collection performed best?

Was there a difference in the amount of time it took to construct the lists?

Try re-running the test with more or less virtual users, and for longer or shorter durations. Does either of these conditions affect the results?

Were there any errors in the test execution? What is the cause?

Going Further

Based on what you have learned in this lab, you could enhance your learning experience by:

- Repeating this test approach to compare performance for different C# loop types or other class libraries
- Since the code in the unit class methods are almost identical, would it be possible to use abstraction, interfaces, or inheritance to simplify these methods?

Lab 4.1 - Creating web tests for simple use cases

We are developing an ASP.NET application for tracking reservations of the performance testing lab. The developer would like some basic web tests performed.

Currently, the following use cases are implemented:

- Register a user
- Login a user
- Create an announcement
- Read an announcement
- Modify a user account
- Create a reservation

We need a test created for each of these use cases. Each major activity should be wrapped in a transaction. There should be 15 seconds of "think time" before each transaction. Wherever it makes sense, the tests should be data driven.

Additionally, the development server which is currently hosting the site is likely to change, so the web tests should be set up to easily change the web server URL.

Question

What is the baseline time to execute each of the tests including think time?

How much time is spent in each activity?

Approach

We will answer this question by creating six web tests. Each test will have the distinct activities wrapped in transactions. Each test will have the web server parametized. We will edit the think time for each request, add a data source for test parameters, and then execute the tests, and determine the time required for execution.

Instructions

1. Open up Microsoft Visual Studio Team System 2008 Test Edition.

2. Create a new project.

- For the project type, select **Visual C#\Test\Test Project**.
- For the location, select **My Documents\Visual Studio 2008\Projects**.
- For the name, specify **WDPLWebTest_***[your name]*.
- You may skip creating the project and tests by opening the "WDPLWebTest" project at that location.
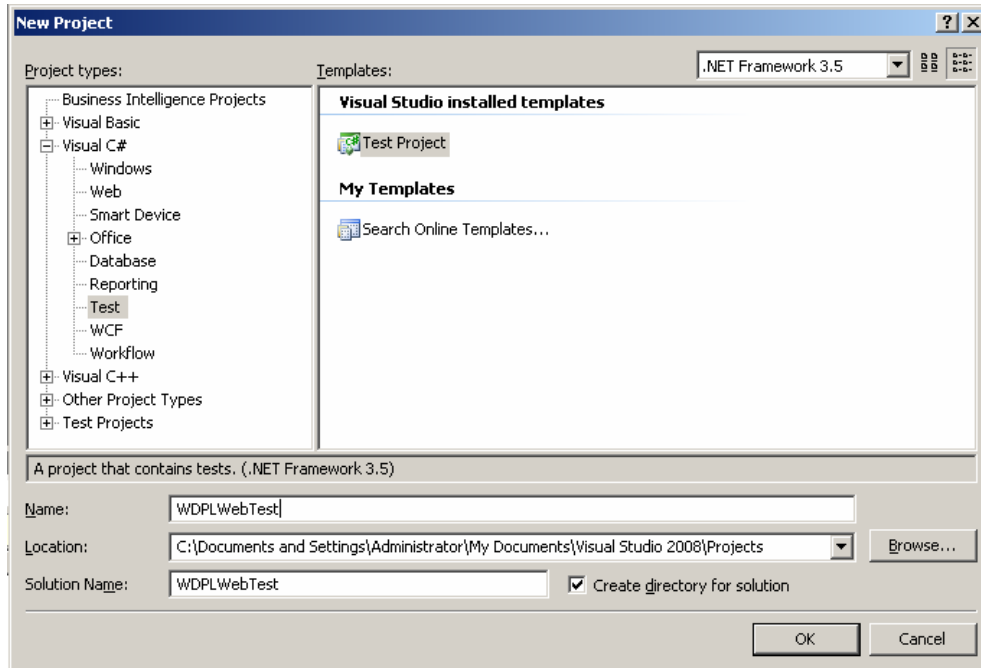
Figure 20

3. In your new project, go to **Test-New Test**. For the type of test select **Web Test**, for the test name specify **CreateUser** and click **OK**.
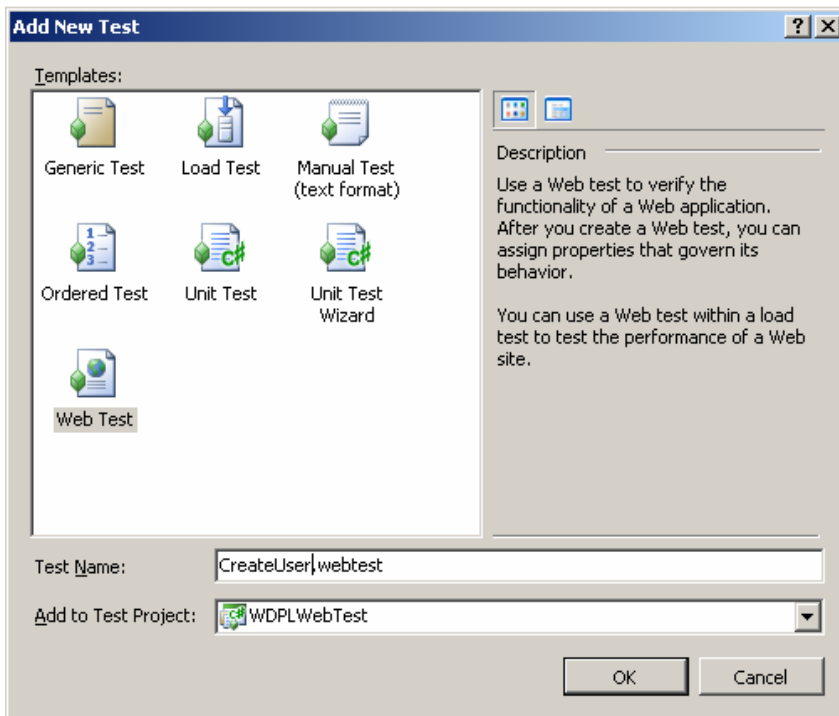
Figure 21

4. This will begin a web test recording session. Perform the following functions in order, and before executing each step, click the comment button to enter a brief comment about the step you are performing.
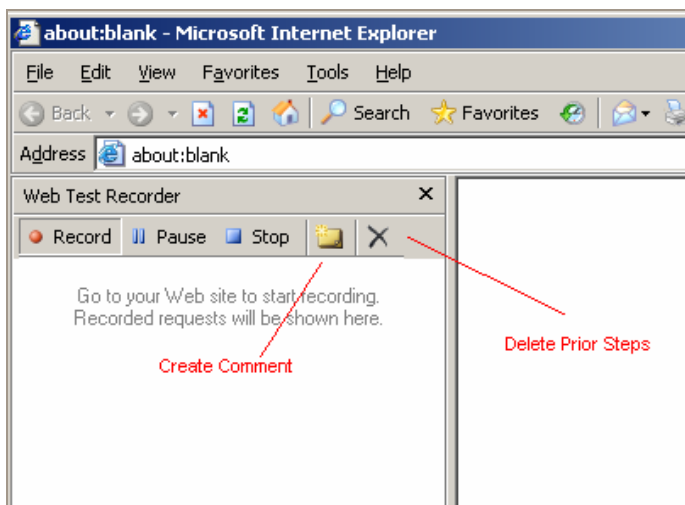
Figure 22

   a. After entering a comment, open the following URL: http:// regis-dufxwufuz /WDPL_Home

   b. After entering a comment, click on the **Testers** link in the upper navigation bar.

   c. After entering a comment, click on the **registrations** link in the **New Member Registration** section.

   d. After entering a comment:

       i. For the user name, type in **RECORD123**
     ii. For the password type in **test123**
   iii. For the email type in **test@test123.com**
    iv. For the security question and answer, type in **test**
     v. Once all of these fields are completed, click **Create User**.

e.   After entering a comment, in the next screen:

       i. For the first name type in **FirstName**
     ii. For the last name type in **LastName**
   iii. For the address type in **address123**
    iv. For the phone type in **555-555-5555**
     v. Deselect the box for **Receive Newsletter**
    vi. When you have completed this form, click **Finish**

f.   This should return you to the home page. Enter a comment, and then click the **Log Off** button. Once you have logged off, you can close the Internet Explorer window.

5. This should return you to Visual Studio and you should see the steps you recorded. If done properly, it should look something like this:
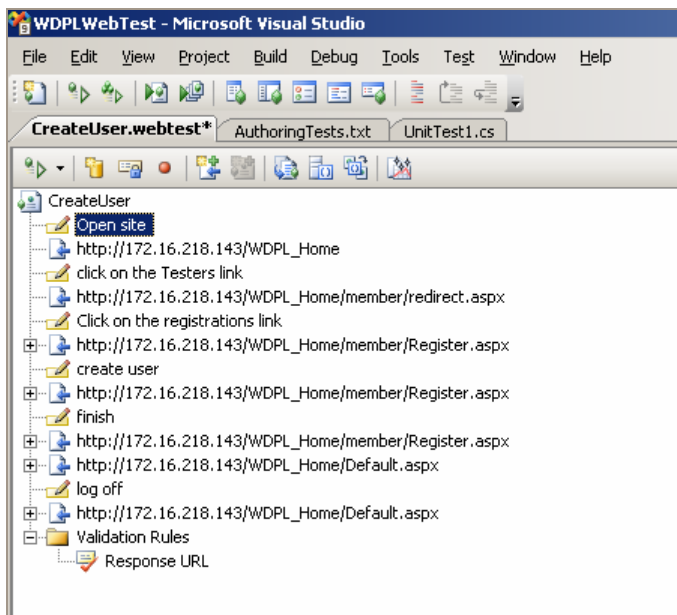
Figure 23

6. Once you have the web test properly recorded, the next step is to wrap the requests in transaction. Because we have commented before each step, it is easy to see which request belongs to which transaction. Right click on the request immediately below the first comment (in the diagram above, the comment "Open site"), and choose **Insert Transaction**.
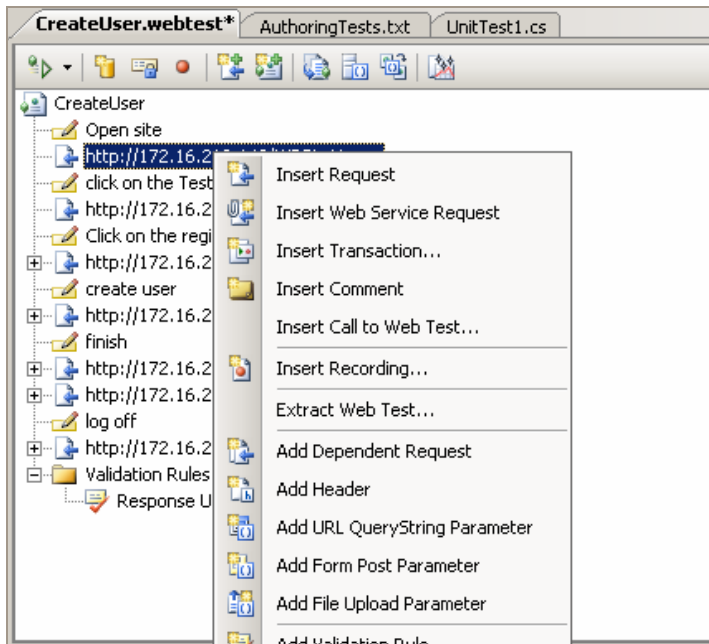
Figure 24

7. For the transaction name, enter **CreateUser_01_open_site**. This naming convention will allow us to determine which transaction belongs to a test, along with the step and what the recorded activity. Click **OK** when you are done.

8. Repeat this process for each request until your test looks like the figure below.
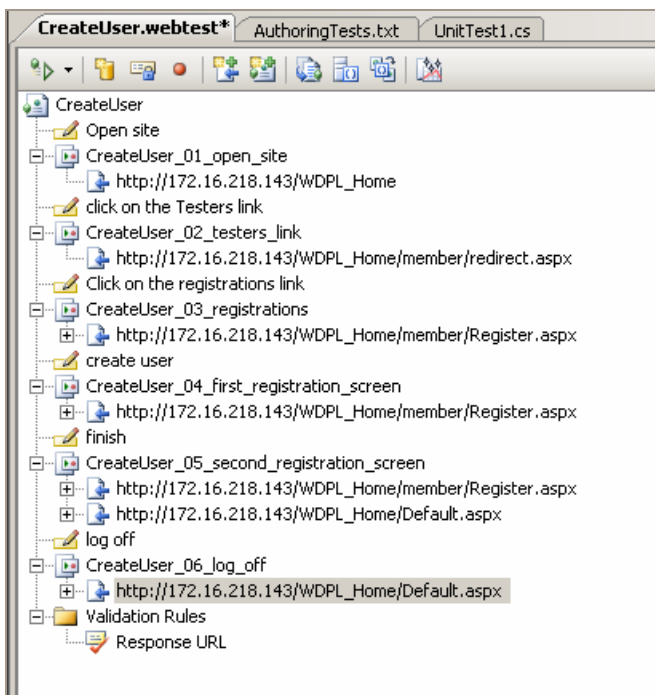

Figure 25

9. Now that all of the requests are contained in transactions, we would like to set the think time for each request. Right click on the first request (http://172.16.218.143/WDPL_Home) and select **Properties**. You should see a list of properties for that request. In that list is think time, which you should change to 15.
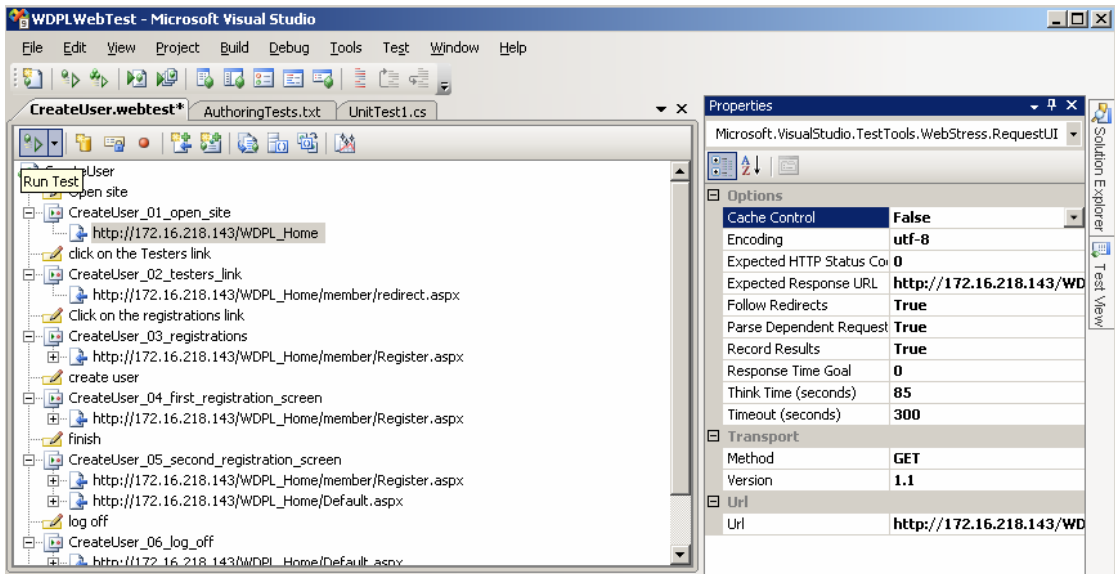
Figure 26

10. Repeat the above step for all of the other requests in the list.

11. The next thing we want to do is link up the fields from the registration to a data source. Expand the request in the **CreateUser_04_first_registration_screen** transaction so that you can see all the form post parameters, as demonstrated in the figure below.
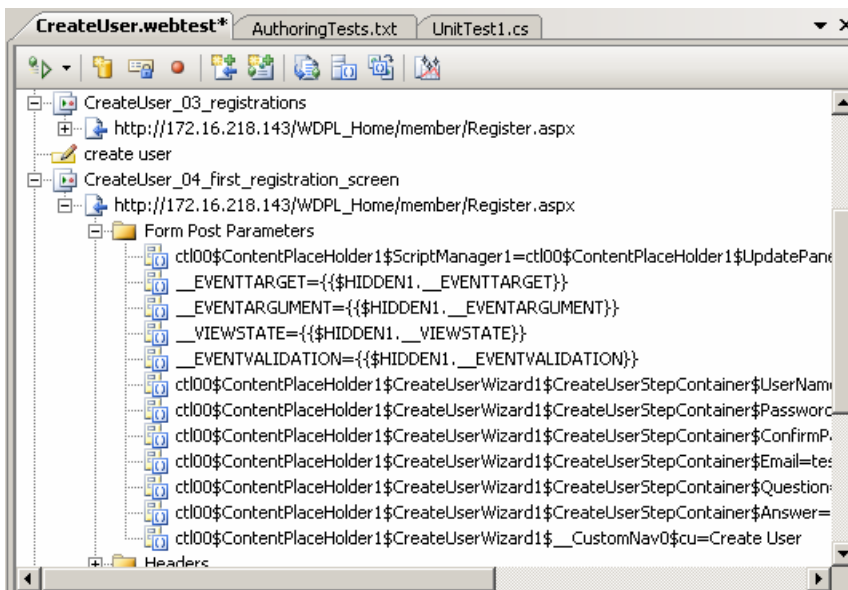
Figure 27

12. Right click on the entry with the following text **UserName=record123** and select **Properties**. That will open the properties for that Form Post Parameter, as shown in the figure below.
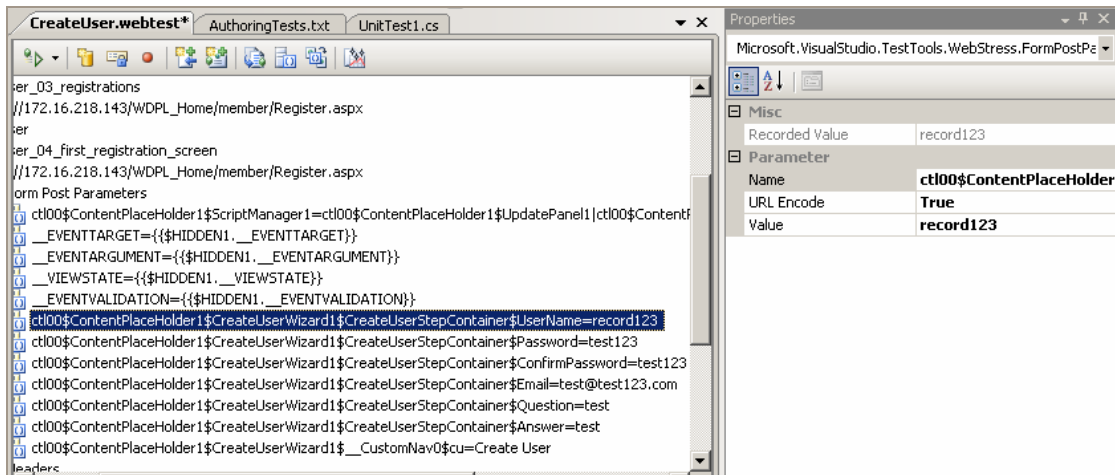
Figure 28

13. In the **Properties** screen, click on the **Value** of **record123**, and an arrow should appear allowing you to open a drop down menu. Click on **Add Data Source**.
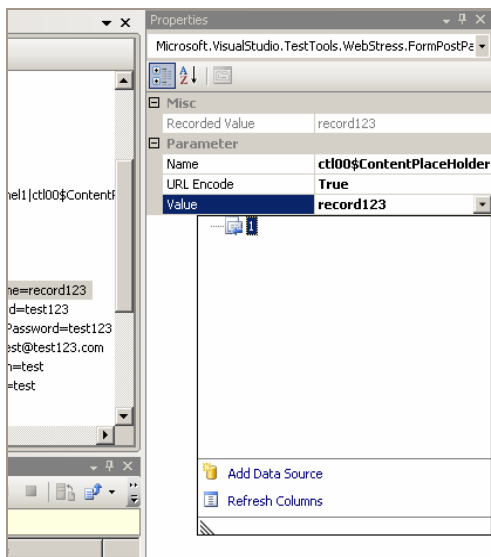
Figure 29

14. In the New Test Data Source Wizard, select **Database** and click **Next**.
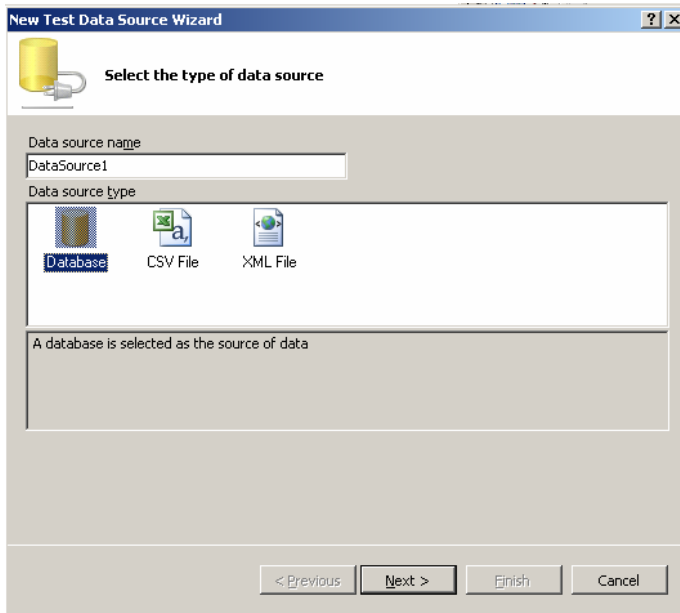
Figure 30

15. On the next screen, click the **New Connection** button. For the data source, **Microsoft SQL Server** should be selected. From the list of servers, select **REGIS-FWVWSBM3F**. For the database name, select **PDDLabsTestParams**. Once these are selected, click the **Test Connection** button to make sure you are able to connect, and then click **OK**.
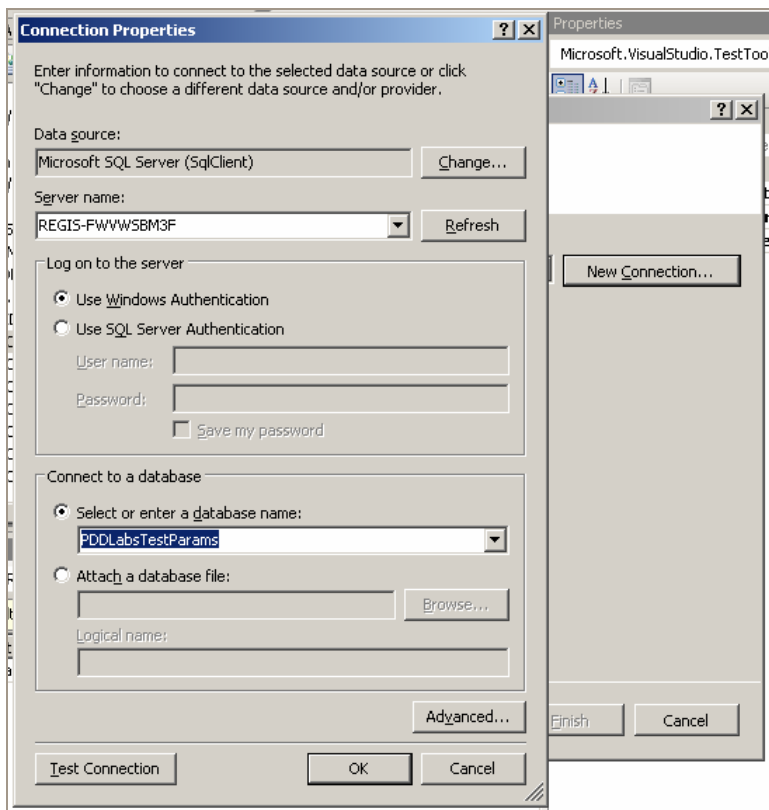
Figure 31

16. You should have been returned to the **Data connection selection** screen. Click **Next**, and then select **CreateUserTest** from the list of tables, and click **Finish**.
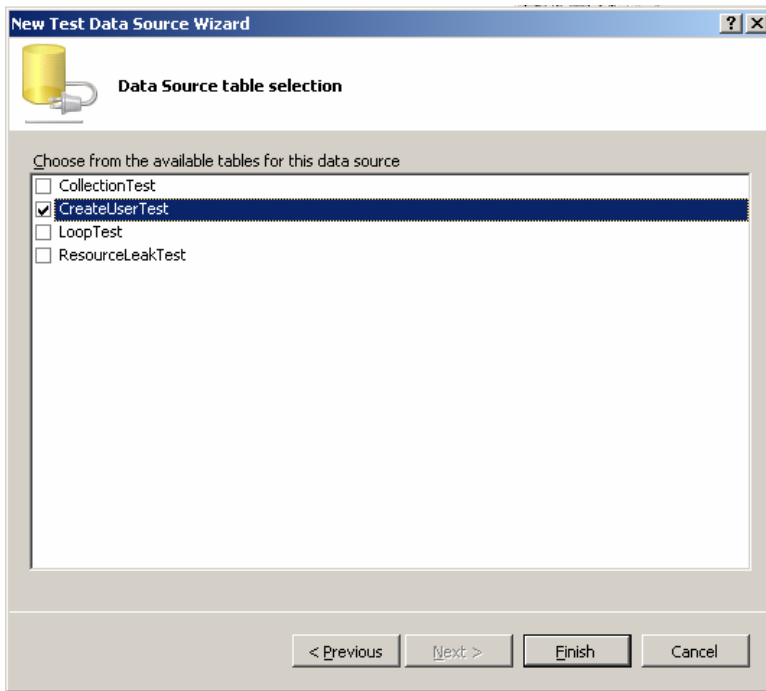
Figure 32

17. Now that you have created a data source, select the UserName=record123 post parameter again, and click the drop down menu in the **Properties** pane for the **Value** of **record123**. In the drop down menu, expand the data source and the table to locate the field you want to use (**UserName**) as a data source.
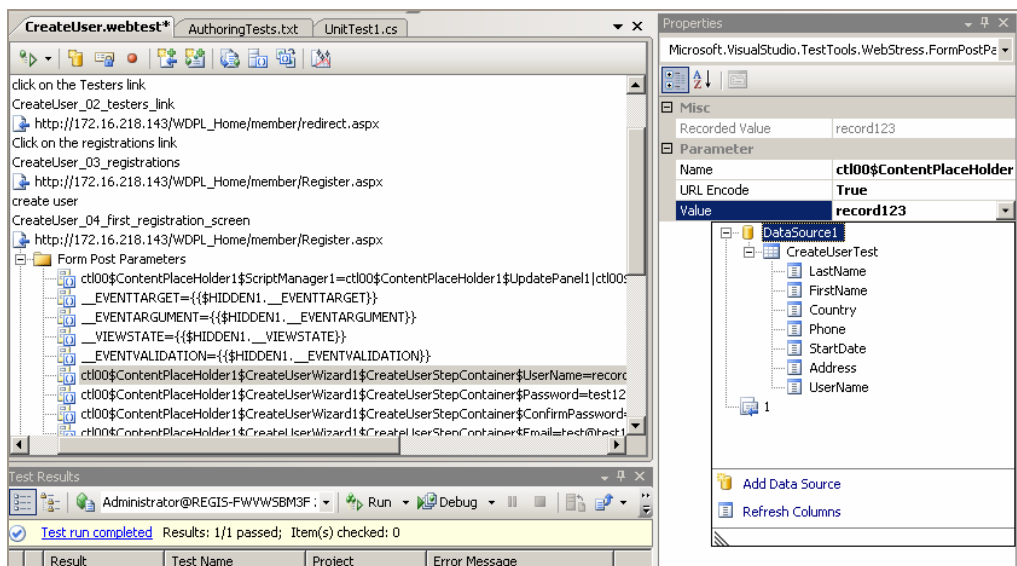

Figure 33

18. Repeat this process to map the Form Post parameters in the first and second registration screens for the first name, last name, address, and phone.

19. The last step for this web test is to parametize the web server, so that if the web site is moved to a new URL, you can still run the test. This is easily done by clicking the button shown in the figure below.
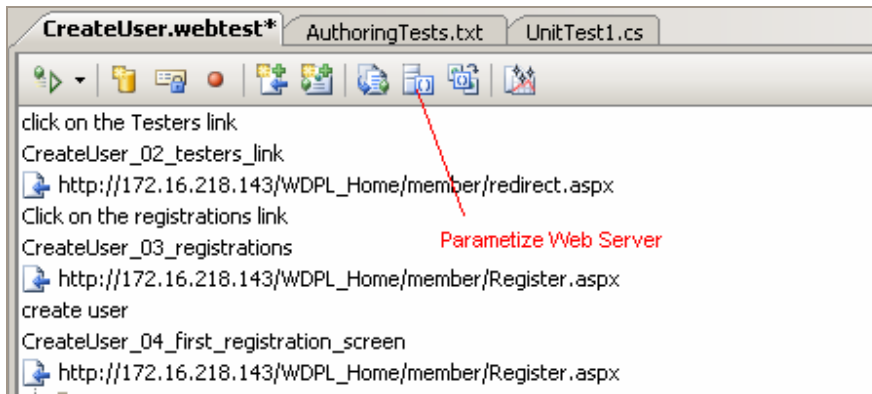
Figure 34

These instructions demonstrate how to record a web test, wrap the recorded requests in transactions, set the think time for each request, map post parameters to a data source, and parametize the web server. Use the same technique to create web tests for the remaining use cases, namely:

- Login a user
- Create an announcement
- Read an announcement
- Modify a user account
- Create a reservation

Results

Based upon your test results, how much time does it take to perform each web test?

How much time does it take to perform each transaction?

Were any errors or problems detected by the test? What is the cause?

Going Further

Based on what you have learned in this lab, you could enhance your learning experience by:

- Converting a web test to code and adding conditional flows or looping.
- Research web test plugins on the internet and imagine a way in which a plugin could enhance testing objectives.
- Use extraction rules to validate that the correct text is returned for the user name after logging in or registering a user.

Lab 5.1 - Load testing against known risks and usage patterns

Background

We are preparing to test a web application for a virtual performance testing lab. The web tests have already been created.

An analysis of web logs has determined:
- 50% of visitors use Internet Explorer 7
- 25% of visitors use Firefox 3
- 15% of visitors use Internet Explorer 6
- 5% of visitors use Firefox 2
- 5% of visitors use a phone web browser
- 65% of visitors are on a LAN connection
- 25% of visitors are using a remote broadband connection
- 10% of visitors are using a remote narrowband connection

The business analysis of the application determined that in an hour of peak usage:
- 10 new users will sign up
- 20 users will modify their account
- 40 users will create a reservation
- 10 users will create an announcement
- 40 users will read an announcement

A risk analysis of the application indicated the following concerns:

- If load exceeds the anticipated peak, there is a concern that the system may go down.
- In the above scenario, it would be good to know what indicators would alert operations engineers that a problem is occurring.
- In the event that the system can handle above peak load, there is a concern that performance would degrade over time.

In addition to these concerns, the infrastructure engineer is concerned about cache faults. She wants to know if during a long test, the cache faults/sec ever reaches above 500.

Question

Can this application be expected to handle the expected load and risks?

Approach

We will answer this question by creating three load tests.

The first load test will be a validation of the performance model at normal peak load. It will ramp up over 5 minutes to full load, and then run for 1 hour.

The second load test will be an endurance test. It will ramp up to double of the expected peak load over 30 minutes, and then sustain that level of load for 1 day.

The third load test will be a capacity test. It will ramp up gradually to quadruple of expected load over the span of two hours, and will be terminated when it reaches indications of system failure.

Before we can create the load tests, we need to determine how many concurrent users the baseline test will use, and how much time they will wait before each test execution. Here is how we can make our initial calculations.

1. Prior to executing this test, we determined that with think time included, the web tests take approximately this long in seconds per execution:

CreateAnnouncement	124
ReadAnnouncement	105
CreateReservation	93.7
ModifyAccount	92.7
CreateUser	78.3

2. With this measurement, we can safely say that all of the tests can be executed 10 times per hour.

3. Since we need to perform 120 total tests, this can be accomplished with 12 concurrent users.

4. Next, we averaged the execution time for all of the scripts, and determined that the average execution time is 98.74 seconds.

5. Figuring on ten executions per hour, each user would be spending 987.4 seconds executing tests.

6. An hour contains 3600 seconds, so we know that the users will need to be idle for 2612.6 seconds per hour.

7. If we divide that by 10 (the number of tests that each user will execute) we determine that each user should need to wait 261 seconds between test executions.

The reason that the wait time is important is because without it, all of the tests will be performed in the early part of the hour. Setting wait time allows for an even distribution of tests over the course of the hour.

This is a fairly crude method for determining the number of virtual users needed and the wait time between executions, so after performing the test one question will be whether we accomplished our objective and whether these numbers should be adjusted. If it takes more than an hour (factoring in ramp-up time) to complete the test, we will reduce the amount of time between tests.

1. Open up the test solution created in the prior lab. If you did not complete the prior test, you can use the solution WDPLWebTest

2. You should have six web tests in the solution for each of the six test cases from the previous lab. If you have not already done so, move each of these web tests into a new folder named **WebTests**.

3. Create a new folder in the solution called **LoadTests**. Right click on that folder and select **Add\New Test**. Select **Load test** as the type of test, and name it **BaselineTest.loadtest**.
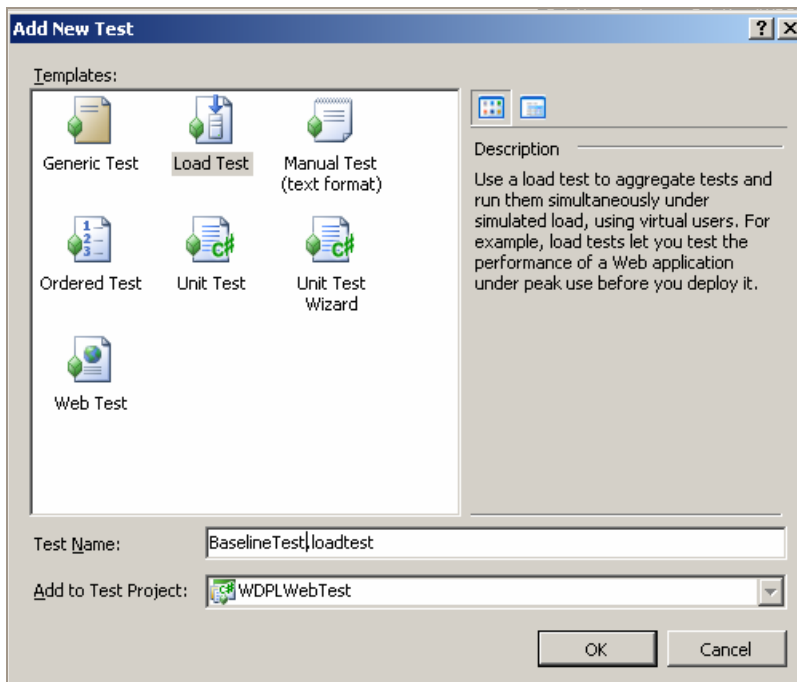
Figure 35

4. Click **Next** at the welcome screen of the **New Load Test Wizard**, and in the first scenario configuration screen:

   a. Specify the **Think time profile** to **Use recorded think times**
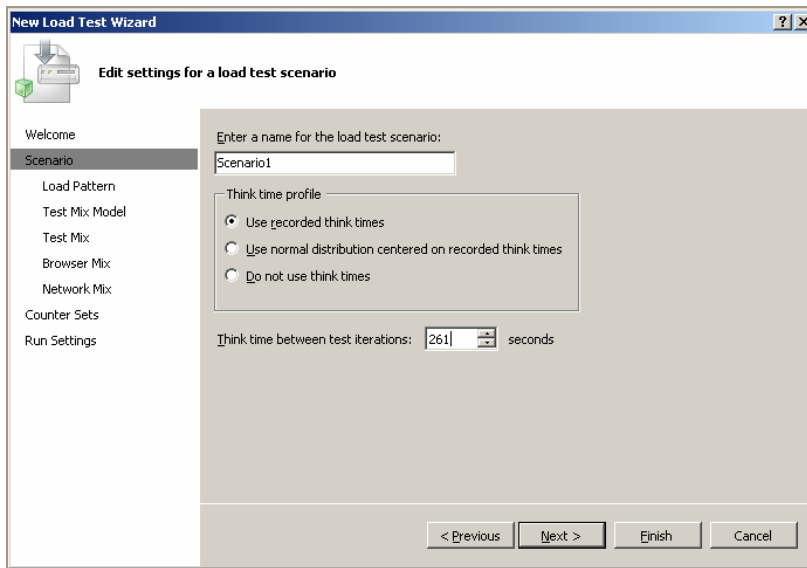   b. Specify the **Think time between test iterations** to **261 seconds**.

Figure 36

5. On the second scenario configuration screen for the load pattern:

    a. Select a **Step load** with the **Start user count** of **1**
    b. Specify the **Step duration** of **25 seconds**
    c. Specify the **Step user count** of **1 users/step**
    d. Specify the **Maximum user count** of **12 users**.

    The reason for these settings is we wanted the load to increase to full load over 5 minutes. There are 300 seconds in 5 minutes, which divided by 12 (the number of target users) is 25.
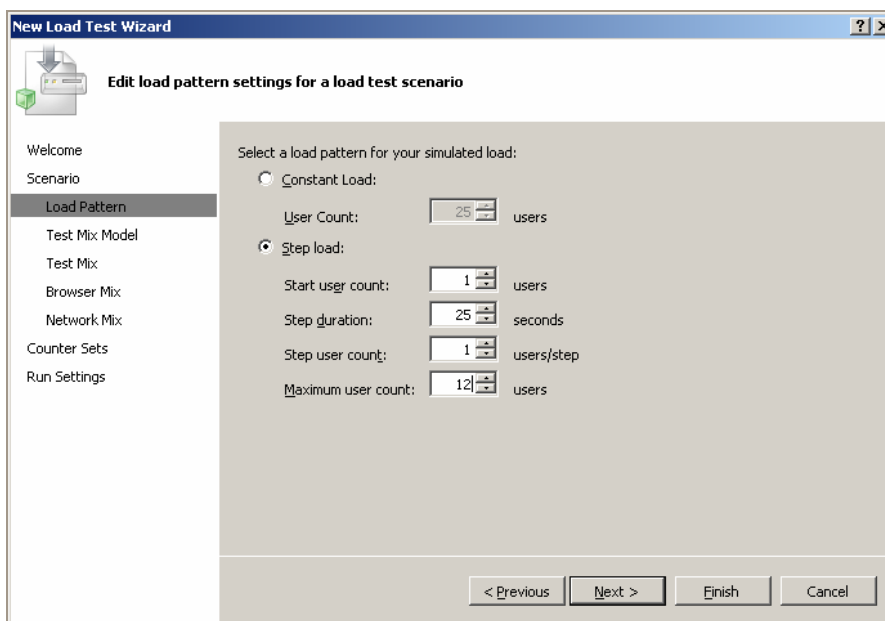

Figure 37

6. On the third scenario configuration screen, choose for the test mix to be modeled **Based on the total number of tests**.
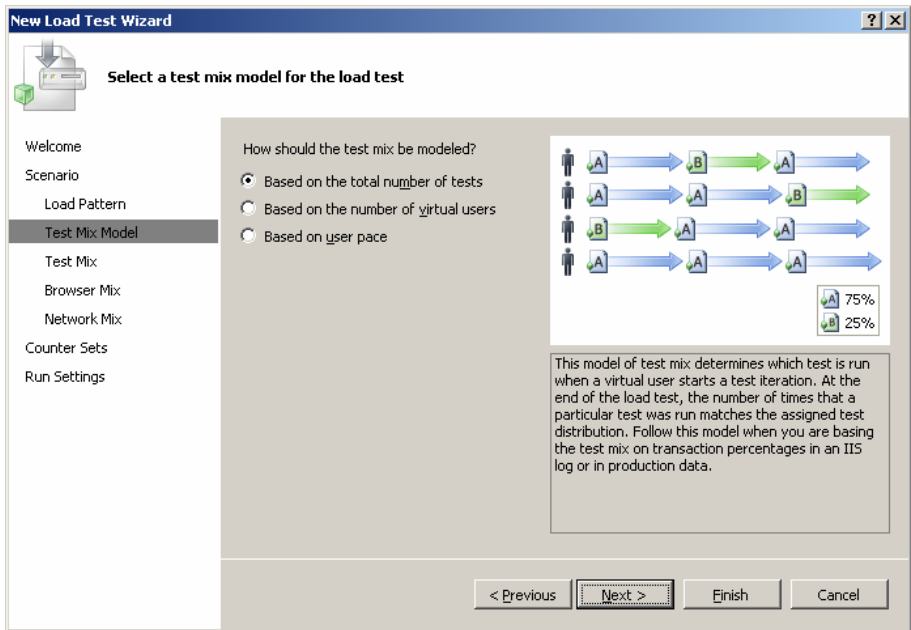
Figure 38

7. In the fourth scenario configuration screen, add the following tests and set their respective distribution percentage:

    a. CreateAnnouncement: 8.34%
    b. ReadAnnouncement: 33.33%
    c. CreateReservation: 33.33%
    d. ModifyAccount: 16.66%
    e. CreateUser: 8.34%

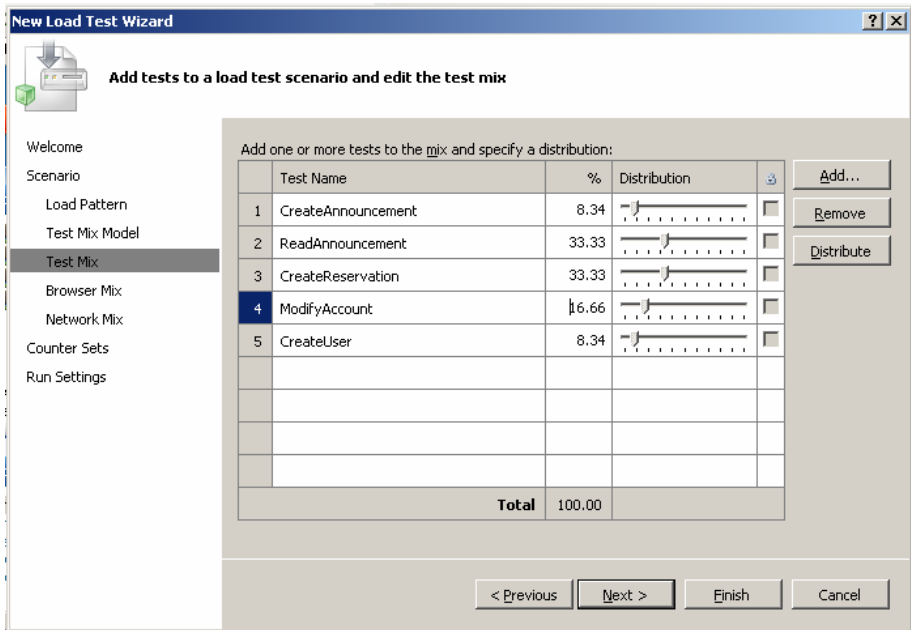These percentages we calculated as a simple percentage of the total tests.

Figure 39

8. For the **Browser mix**, input the specifications from Lab 5.1's Background section (as shown below).
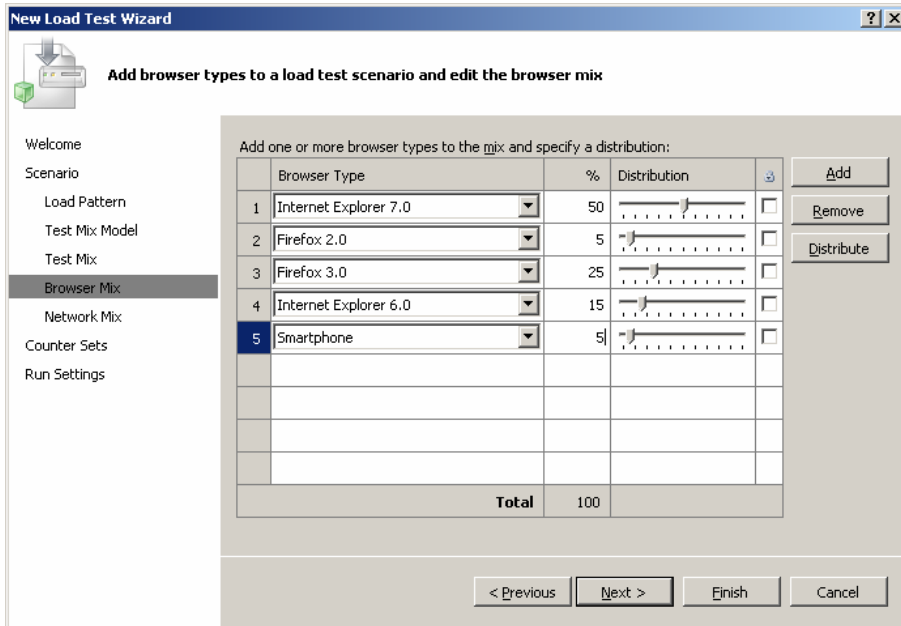
Figure 40

9. For the **Network mix**, input the specifications from Lab 5.1's Background section (as shown below).
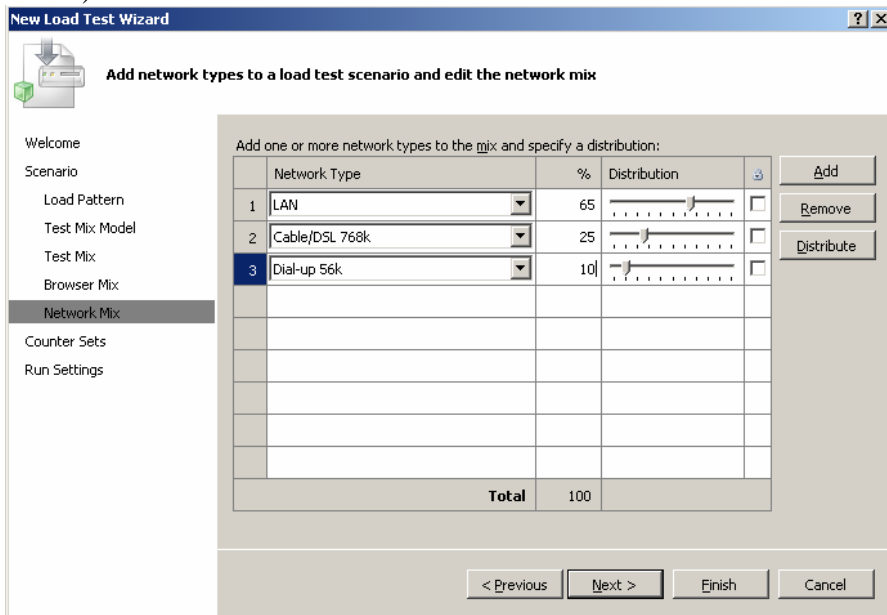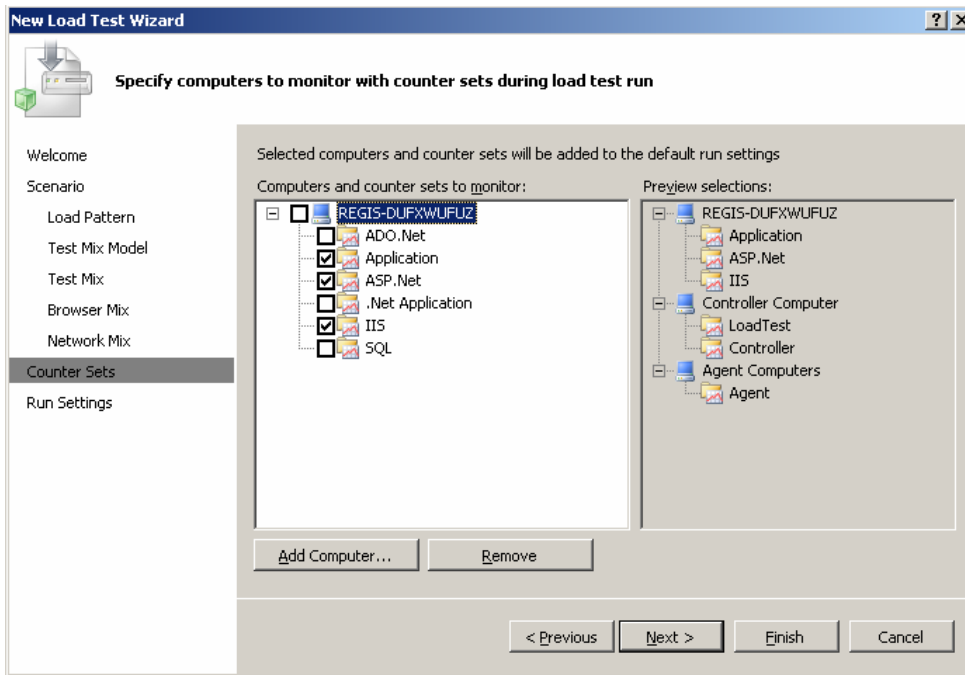

Figure 41

10. In the **Counter Sets** configuration screen, add the web server (**REGIS-DUFXWUFUZ**) and select the **Application**, **ASP.Net**, and **IIS** counter categories.

11. In the **Run Settings** configuration, **specify the length by Test iterations**, and set it to complete **120** iterations, and then click **Finish**.
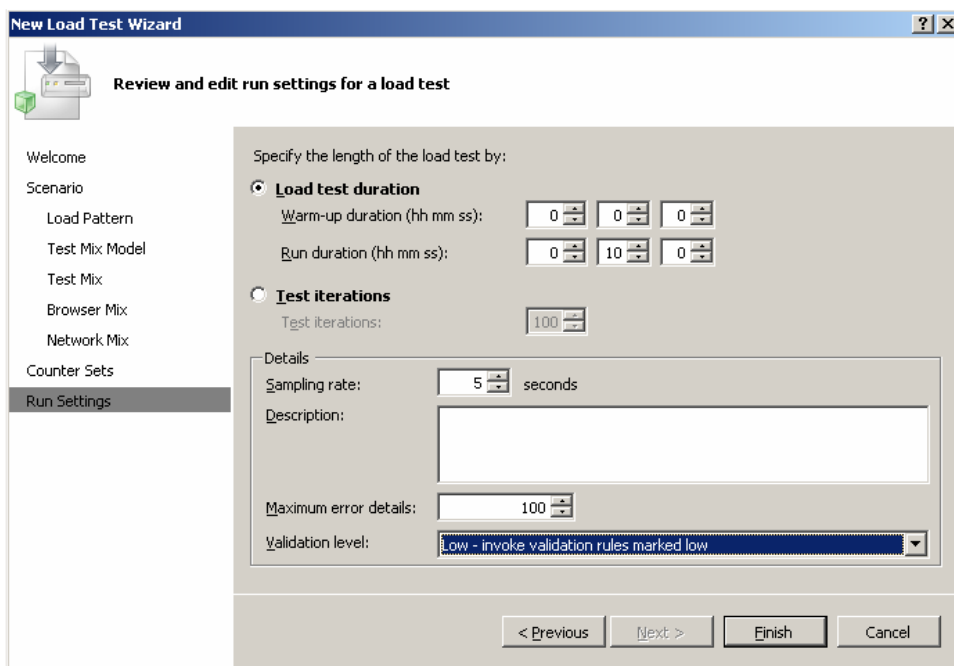

Figure 42

12. Now that you have created the baseline test, we need to create the endurance test. Follow the same procedures used to create the baseline test except:

    a. On the second scenario configuration screen for the load pattern:

        i. Select a **Step load** with the **Start user count of 1**,
       ii. The **Step duration** should be **75 seconds**
      iii. The **Step user count** should be **1 users/step**
      iv. The **Maximum user count** should be **24 users**.

The reason for these settings is we wanted the load to increase to twice the baseline load over 30 minutes. There are 1800 seconds in 30 minutes, which divided by 24 is 75.

    b. In the run setting configuration:

        i. Specify the length **by Load test duration**
        ii. Set the **Run duration** for **24:30:00**
        iii. Click Finish.

13. We also need to create the capacity test. Follow the same procedures used to create the baseline test except:

    a. On the second scenario configuration screen for the load pattern:

        i. Select a **Step load** with the **Start user count of 1**,
        ii. The **Step duration** should be **150 seconds**
        iii. The **Step user count** should be **1 users/step**
        iv. The **Maximum user count** should be **48 users**.

The reason for these settings is we wanted the load to increase to quadruple the baseline load over 120 minutes.

    b. In the run setting configuration:

        i. Specify the length **by Load test duration**
        ii. Set the **Run duration** for **2:00:00**
        iii. Click Finish.

14. The infrastructure engineer also wanted to know if during any of the tests the cache faults/sec ever reaches 500. In order to do this, open the baseline test, right click on the **Application** category in **Counter Sets**, and choose **Add Counters**.
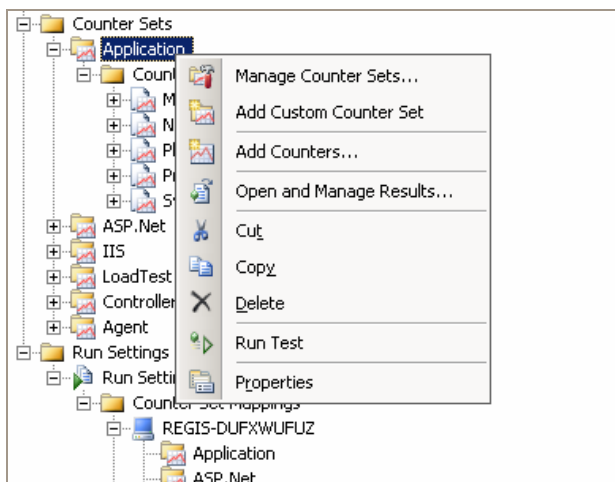

Figure 43

15. Select the web server (**REGIS-DUFXWUFUZ**) as the **Computer**, **Memory** as the **Performance Category**, and **Cache Faults/sec** in the list of counters.

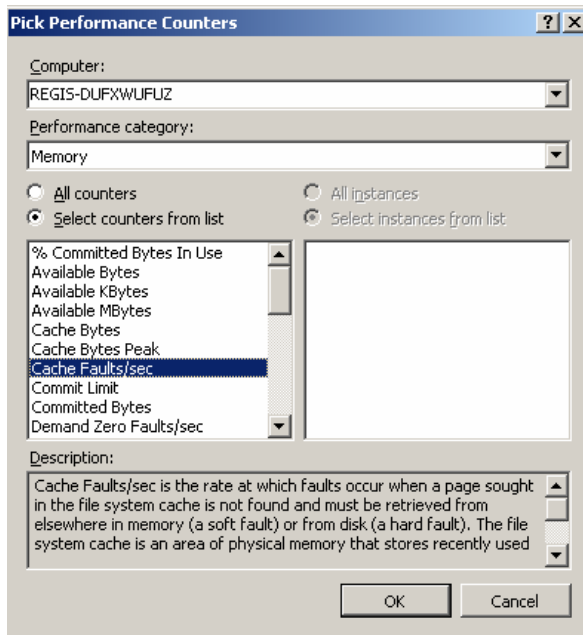**Figure 44**

16. Locate that new counter in the **Application\Memory** counters category, right click, and select **Add Threshold Rule**
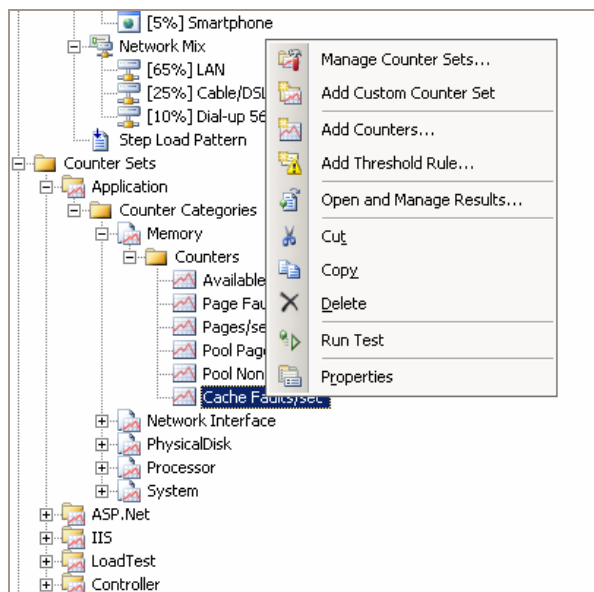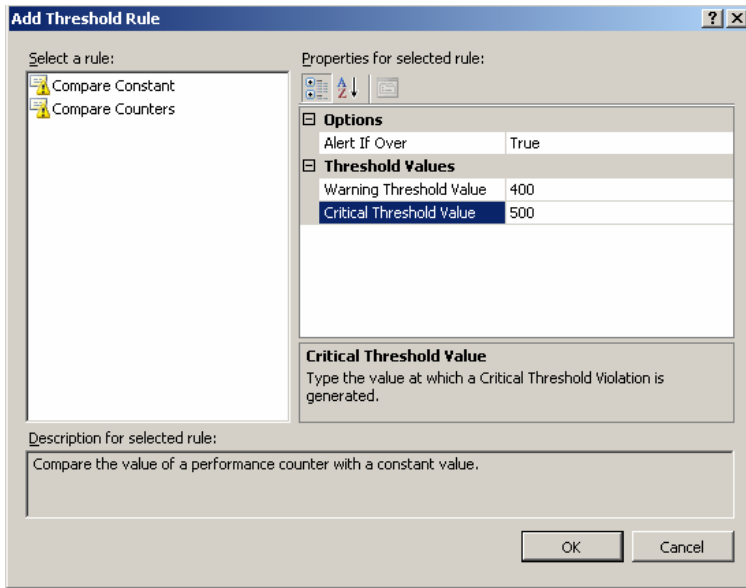

Figure 45

17. Set the threshold rule to **Alert if over** = **True**, with the **Warning Threshold Value** of **400**, and the **Critical Threshold Value** of **500**.

Figure 46

18. Repeat this process for the endurance and capacity tests.

That completes the instructions, so you should be able to run each of the load tests at this point.

Results

Were you able to run the baseline test within 1 hour, factoring in ramp up time?

Based upon your test results and analysis, does the application perform satisfactorily at baseline load?

Did you try running the endurance test? What were the results?

After running the capacity test, at what point did the server seem to fail?

Were there any errors in the test execution? What is the cause?

Going further

Based on what you have learned in this lab, you could enhance your learning experience by:

- Creating a spike test by developing a custom plugin
- Investigate and use VSTS's advanced code analysis and performance session functions.

Lab 6.1 - Using Virtualization for 'Wash and Rinse' Tests

Background

We are continuing with our testing of a web application for a virtual performance testing lab. The web tests and load tests have already been created.

At this point, the project sponsors would like some sizing guidance for web server.

Question

Should the web server be configured with 1 or 2 gigs of RAM?

Should the web server have 1 or 2 dedicated processors?

Approach

We will execute the baseline and capacity tests on the following configurations:

1G Ram, 1 Processor
1G Ram, 2 Processors
2G Ram, 1 Processor
2G Ram, 2 Processors

Between executions, we will revert the web server to a saved snapshot in order to automatically clear out any temporary files, and in order to remove the possibility that temporary files or other state-related conditions could be skewing the results.

Instructions

Prior to beginning your labs, a snapshot of the web server has been created. This spares you the effort required to log in to the web site and clear out all of the temporary data generated in prior load tests. If you are going to use this snapshot, you can skip to step 14.

If you want to create a snapshot anyway, here are the steps.

1. Log on to the **WDPL SUT** computer through the **VM Console**.

2. Launch the http://localhost/WDPL_Home web site and log in with these credentials:

   User Name: admin
   Password: admin

3. Click on the **Manage Users** button.

4. Delete all of the users EXCEPT:

   admin
   permuser001-permuser010

5. Click **Reservations** from the top navigation bar

6. Click **List view**

7. Remove any reservations that are currently in the system.

8. Click **Announcements** from the top navigation bar

9. Remove any announcements EXCEPT:

   "Welcome to the Windows Development Performance Lab"

10. Log out

11. Shut down **WDPL SUT**

12. Right click on the VM's listing in the **Inventory**, and select **Remove Snapshot**.
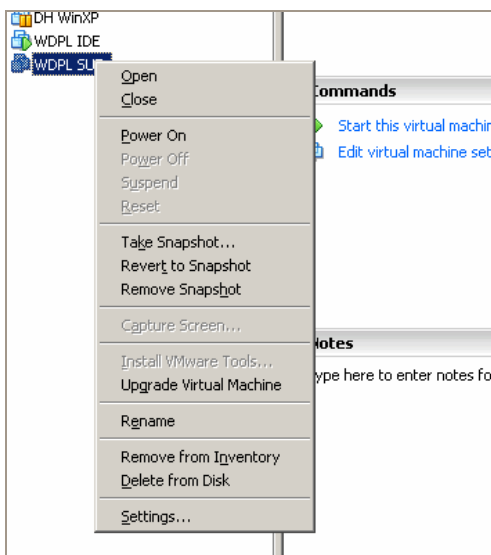
Figure 47

13. Right click on the VM from the **Inventory** list again, and select **Take Snapshot**.

14. Whether you are using the existing snapshot, or whether you just created a new snapshot using the instructions above, at this point we execute the first set of load tests. Be sure that both **WDPL IDE** and **WDPL SUT** are running and logged in, and then run the baseline and capacity tests on **WDPL IDE**.

15. Once they have run, shut down **WDPL SUT**.

16. Right click on **WDPL SUT** listing in the inventory, and select **Revert to Snapshot**.
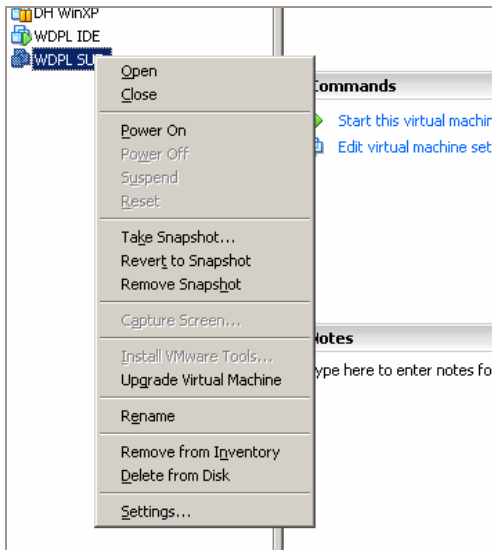
Figure 48

17. Next, right click on the **WDPL SUT** listing in the **Inventory**, and select **Settings**. In the **Settings** screen, change the **Number of processors** to **Two**.
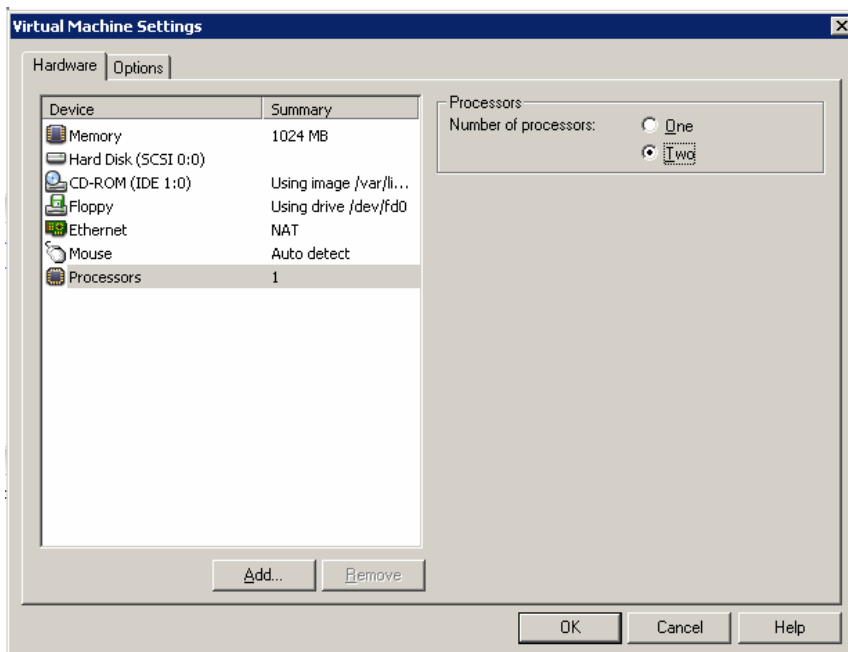
Figure 49

18. Start the virtual machine, log in, and execute the two load tests again.

19. Repeat steps 15-18, this time setting the **Number of processors** to **One**, and setting the **Memory** to **2048 MB**.

20. Repeat steps 15-18 again, this time setting the **Number of processors** to **Two**, and setting the **Memory** to **2048**.

21. Now that you have run the four sets of performance tests, review each result in Visual Studio by clicking the drop down list to the left of the **Run** button in the **Test Results** pane, and selecting **Manage test runs…**
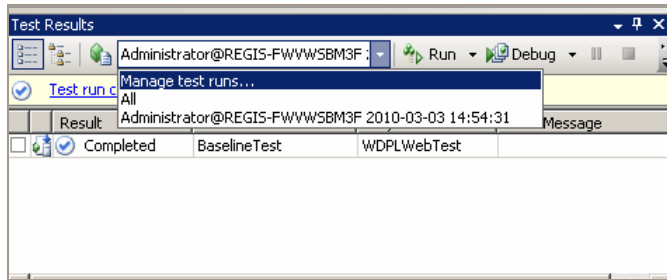
Figure 50

22. That will open the **Test Runs** window. Under **Completed**, you should find your prior test results.
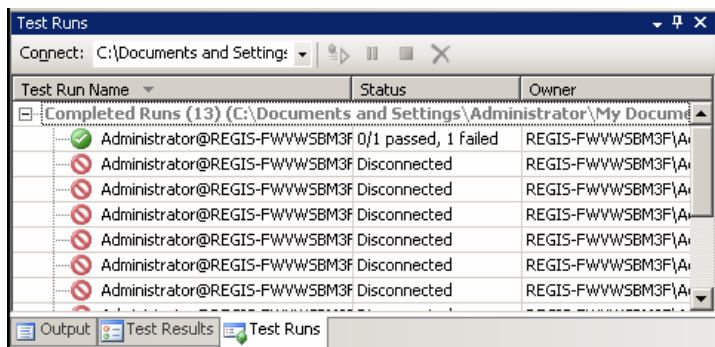
Figure 51

Results

Did performance improve or diminish with the additional processor?

Did performance improve or diminish with the additional RAM?

Was there a noticeable difference in the breaking points in the capacity tests?

What would you recommend for the configuration?

Going further

Based on what you have learned in this lab, you could enhance your learning experience and the performance testing lab by:

- Establishing an isolated network for the two VMs to assure that tests will not affect external systems, and vice versa.
- Configuring **WDPL_SUT** as a domain controller, and **WDPL_IDE** as a member of that domain to establish centralized security.

Appendix B: Performance testing terminology

The following table contains a useful set of performance testing terminology from *Performance Testing Guidance for Web Applications: Patterns & Practices*.

Term / Concept	Description
Capacity	The *capacity* of a system is the total workload it can handle without violating predetermined key performance acceptance criteria.
Capacity test	A *capacity test* complements load testing by determining your server's ultimate failure point, whereas load testing monitors results at various levels of load and traffic patterns. You perform capacity testing in conjunction with capacity planning, which you use to plan for future growth, such as an increased user base or increased volume of data. For example, to accommodate future loads, you need to know how many additional resources (such as processor capacity, memory usage, disk capacity, or network bandwidth) are necessary to support future usage levels. Capacity testing helps you to identify a scaling strategy in order to determine whether you should scale up or scale out.
Component test	A *component test* is any performance test that targets an architectural component of the application. Commonly tested components include servers, databases, networks, firewalls, and storage devices.
Endurance test	An *endurance test* is a type of performance test focused on determining or validating performance characteristics of the product under test when subjected to workload models and load volumes anticipated during production operations over an extended period of time. Endurance testing is a subset of load testing.
Investigation	*Investigation* is an activity based on collecting information related to the speed, scalability, and/or stability characteristics of the product under test that may have value in determining or improving product quality. Investigation is frequently employed to prove or disprove hypotheses regarding the root cause of one or more observed performance issues.
Latency	*Latency* is a measure of responsiveness that represents the time it *y* takes to complete the execution of a request. Latency may also represent the sum of several latencies or subtasks.
Metrics	*Metrics* are measurements obtained by running performance tests as expressed on a commonly understood scale. Some metrics commonly obtained through performance tests include processor utilization over time and memory usage by load.
Performance	*Performance* refers to information regarding your application's response times, throughput, and resource utilization levels.
Performance test	A *performance test* is a technical investigation done to determine or validate the speed, scalability, and/or stability characteristics of the product under test. Performance testing is the superset containing all other subcategories of performance testing described in this chapter.
Performance budgets or	*Performance budgets* (or *allocations*) are constraints placed on developers regarding allowable resource consumption for their component.

Term / Concept	Description
allocations	
Performance goals	*Performance goals* are the criteria that your team wants to meet before product release, although these criteria may be negotiable under certain circumstances. For example, if a response time goal of three seconds is set for a particular transaction but the actual response time is 3.3 seconds, it is likely that the stakeholders will choose to release the application and defer performance tuning of that transaction for a future release.
Performance objectives	*Performance objectives* are usually specified in terms of response times, throughput (transactions per second), and resource-utilization levels and typically focus on metrics that can be directly related to user satisfaction.
Performance requirements	*Performance requirements* are those criteria that are absolutely non-negotiable due to contractual obligations, service level agreements (SLAs), or fixed business needs. Any performance criterion that will not unquestionably lead to a decision to delay a release until the criterion passes is not absolutely required and therefore, not a requirement.
Performance targets	*Performance targets* are the desired values for the metrics identified for your project under a particular set of conditions, usually specified in terms of response time, throughput, and resource-utilization levels. Resource-utilization levels include the amount of processor capacity, memory, disk I/O, and network I/O that your application consumes. Performance targets typically equate to project goals.
Performance testing objectives	*Performance testing objectives* refer to data collected through the performance-testing process that is anticipated to have value in determining or improving product quality. However, these objectives are not necessarily quantitative or directly related to a performance requirement, goal, or stated quality of service (QoS) specification.
Performance thresholds	*Performance thresholds* are the maximum acceptable values for the metrics identified for your project, usually specified in terms of response time, throughput (transactions per second), and resource-utilization levels. Resource-utilization levels include the amount of processor capacity, memory, disk I/O, and network I/O that your application consumes. Performance thresholds typically equate to requirements.
Resource utilization	*Resource utilization* is the cost of the project in terms of system resources. The primary resources are processor, memory, disk I/O, and network I/O.
Response time	*Response time* is a measure of how responsive an application or sub-system is to a client request.
Saturation	*Saturation* refers to the point at which a resource has reached full utilization.
Scalability	*Scalability* refers to an application's ability to handle additional workload, without adversely affecting performance, by adding resources such as processor, memory, and storage capacity.
Scenarios	In the context of performance testing, a *scenario* is a sequence of steps in your application. A scenario can represent a use case or a business function such as searching a product catalog, adding an item to a shopping cart, or placing an order.
Smoke test	A *smoke test* is the initial run of a performance test to see if your application can perform its operations under a normal load.
Spike test	A *spike test* is a type of performance test focused on determining or validating performance characteristics of the product under test when subjected to workload models and load volumes that repeatedly increase beyond anticipated production operations for short periods of time. Spike testing is a subset of stress testing.
Stability	In the context of performance testing, *stability* refers to the overall *y* reliability,

Term / Concept	Description
	robustness, functional and data integrity, availability, and/or consistency of responsiveness for your system under a variety conditions.
Stress test	A *stress test* is a type of performance test designed to evaluate an application's behavior when it is pushed beyond normal or peak load conditions. The goal of stress testing is to reveal application bugs that surface only under high load conditions. These bugs can include such things as synchronization issues, race conditions, and memory leaks. Stress testing enables you to identify your application's weak points, and shows how the application behaves under extreme load conditions.
Throughput	*Throughput* is the number of units of work that can be handled per unit *t* of time; for instance, requests per second, calls per day, hits per second, reports per year, etc.
Unit test	In the context of performance testing, a *unit test* is any test that targets a module of code where that module is any logical subset of the entire existing code base of the application, with a focus on performance characteristics. Commonly tested modules include functions, procedures, routines, objects, methods, and classes. Performance unit tests are frequently created and conducted by the developer who wrote the module of code being tested.
Utilization	In the context of performance testing, *utilization* is the percentage of time that a resource is busy servicing user requests. The remaining percentage of time is considered idle time.
Validation test	A *validation test* compares the speed, scalability, and/or stability characteristics of the product under test against the expectations that have been set or presumed for that product.
Workload	*Workload* is the stimulus applied to a system, application, or component to simulate a usage pattern, in regard to concurrency and/or data inputs. The workload includes the total number of users, concurrent active users, data volumes, and transaction volumes, along with the transaction mix. For performance modeling, you associate a workload with an individual scenario.

(Meiers, Farre, Bansode, Barber, & Rea, 2007)

Appendix C: Performance Testing Types

Term	Purpose	Notes
Performance test	To determine or validate speed, scalability, and/or stability.	A performance test is a technical investigation done to determine or validate the responsiveness, speed, scalability, and/or stability characteristics of the product under test.
Load test	To verify application behavior under normal and peak load conditions.	Load testing is conducted to verify that your application can meet your desired performance objectives; these performance objectives are often specified in a service level agreement (SLA). A load test enables you to measure response times, throughput rates, and resource-utilization levels, and to identify your application's breaking point, assuming that the breaking point occurs below the peak load condition.
Endurance test		Endurance testing is a subset of load testing. An *endurance test* is a type of performance test focused on determining or validating the performance characteristics of the product under test when subjected to workload models and load volumes anticipated during production operations over an extended period of time. Endurance testing may be used to calculate Mean Time Between Failure (MTBF), Mean Time To Failure (MTTF), and similar metrics.
Stress test	To determine or validate an application's behavior when it is pushed beyond normal or peak load conditions.	The goal of stress testing is to reveal application bugs that surface only under high load conditions. These bugs can include such things as synchronization issues, race conditions, and memory leaks. Stress testing enables you to identify your application's weak points, and shows how the application behaves under extreme load conditions.
Spike test		Spike testing is a subset of stress testing. A *spike test* is a type of performance test focused on determining or validating the performance characteristics of the product under test when subjected to workload models and load volumes that repeatedly increase beyond anticipated production operations for short periods of time.
Capacity test	To determine how many users and/or transactions a given system will support and still meet performance goals.	Capacity testing is conducted in conjunction with capacity planning, which you use to plan for future growth, such as an increased user base or increased volume of data. For example, to accommodate future loads, you need to know how many additional resources (such as processor capacity, memory usage, disk capacity, or network bandwidth) are necessary to support future usage levels. Capacity testing helps you to identify a scaling strategy in order to determine whether you should scale up or scale out.

(Meiers, Farre, Bansode, Barber, & Rea, 2007)

Appendix D: Other Performance Related Tests and Activities

Term	Notes
Component test	A *component test* is any performance test that targets an architectural component of the application. Commonly tested components include servers, databases, networks, firewalls, clients, and storage devices.
Investigation	*Investigation* is an activity based on collecting information related to the speed, scalability, and/or stability characteristics of the product under test that may have value in determining or improving product quality. Investigation is frequently employed to prove or disprove hypotheses regarding the root cause of one or more observed performance issues.
Smoke test	A *smoke test* is the initial run of a performance test to see if your application can perform its operations under a normal load.
Unit test	In the context of performance testing, a *unit test* is any test that targets a module of code where that module is any logical subset of the entire existing code base of the application, with a focus on performance characteristics. Commonly tested modules include functions, procedures, routines, objects, methods, and classes. Performance unit tests are frequently created and conducted by the developer who wrote the module of code being tested.
Validation test	A *validation test* compares the speed, scalability, and/or stability characteristics of the product under test against the expectations that have been set or presumed for that product.

(Meiers, Farre, Bansode, Barber, & Rea, 2007)

Appendix E: Summary Matrix of Performance Testing Types by Risks Addressed

Performance test type	Risk(s) addressed
Capacity	• Is system capacity meeting business volume under both normal and peak load conditions?
Component	• Is this component meeting expectations? • Is this component reasonably well optimized? • Is the observed performance issue caused by this component?
Endurance	• Will performance be consistent over time? • Are there slowly growing problems that have not yet been detected? • Is there external interference that was not accounted for?
Investigation	• Which way is performance trending over time? • To what should I compare future tests?
Load	• How many users can the application handle before undesirable behavior occurs when the application is subjected to a particular workload? • How much data can my database/file server handle? • Are the network components adequate?
Smoke	• Is this build/configuration ready for additional performance testing? • What type of performance testing should I conduct next? • Does this build exhibit better or worse performance than the last one?
Spike	• What happens if the production load exceeds the anticipated peak load? • What kinds of failures should we plan for? • What indicators should we look for?
Stress	• What happens if the production load exceeds the anticipated load? • What kinds of failures should we plan for? • What indicators should we look for in order to intervene prior to failure?
Unit	• Is this segment of code reasonably efficient? • Did I stay within my performance budgets? • Is this code performing as anticipated under load?
Validation	• Does the application meet the goals and requirements? • Is this version faster or slower than the last one? • Will I be in violation of my contract/Service Level Agreement (SLA) if I release?

(Meiers, Farre, Bansode, Barber, & Rea, 2007)

Appendix F: Summary Matrix of Risks Addressed by Performance Testing Types

Risks	Performance test types									
	Capacity	Component	Endurance	Investigation	Load	Smoke	Spike	Stress	Unit	Validation
Speed-related risks										
User satisfaction			X	X	X			X		X
Synchronicity		X	X	X	X		X	X	X	
Service Level Agreement (SLA) violation			X	X	X					X
Response time trend		X	X	X	X	X			X	
Configuration			X	X	X	X		X		X
Consistency		X	X	X	X				X	X
Scalabilfty-related risks										
Capacity	X	X	X	X	X					X
Volume	X	X	X	X	X					X
SLA violation			X	X	X					X
Optimization	X	X		X					X	
Efficiency	X	X		X					X	
Future growth	X	X		X	X					X
Resource consumption	X	X	X	X	X	X	X	X	X	X
Hardware / environment	X	X	X	X	X		X	X		X
Service Level Agreement (SLA) violation	X	X	X	X	X					X
Stability-related risks										
Reliability		X	X	X	X		X	X	X	
Robustness		X	X	X	X		X	X	X	
Hardware / environment			X	X	X		X	X		X
Failure mode		X	X	X	X		X	X	X	X
Slow leak		X	X	X	X				X	
Service Level Agreement (SLA) violation		X	X	X	X		X	X	X	X
Recovery		X		X			X	X	X	X
Data accuracy and security		X	X	X	X		X	X	X	X

(Meiers, Farre, Bansode, Barber, & Rea, 2007)

References

ADO.NET. (2009, December 4). Retrieved December 28, 2009, from Wikipedia, The Free Encyclopedia:

    http://en.wikipedia.org/w/index.php?title=ADO.NET&oldid=329627377

Bunn, S., & Plenderleith, J. (2009). *Microsoft Visual Studio 2008 Programming.* New York: McGraw-Hill.

Buytaert, K., Dittner, R., Garcia, J. R., Grotenhuis, T., Hart, D. E., Jones, A., et al. (2007). *The Best Damn Server*

    Virtualization Book Period: Including Vmware, Xen, and Microsoft Virtual Server. Burlington, MA:

    Syngress Publishing.

Campbell, S., & Jeronimo, M. (2006). *Applied Virtualization Technology: Usage Models for IT Professionals and*

    Software Developers. Hillsboro, OR: Intel Press.

Chaganti, P. (2007). *Xen Virtualization: A Fast and Practical Guide to Supporting Multiple Operating Systems with*

    the Xen Hypervisor. Birmingham, UK: Packt Publishing.

Dittner, R., Green, G., Grotenhuis, T., Majors, K., Rule Jr., D., & ten Seldam, M. (2006). *Virtualization with*

    Microsoft Virtual Server 2005. Rockland, MD: Syngress Publishing.

Friedman, M. (2005). *Microsoft Windows Server 2003 Performance Guide.* Redmond: Microsoft Press.

Golden, B. (2008). *Virtualization For Dummies.* Hoboken: Wiley Publishing.

Goldworm, B., & Skamarock, A. (2007). *Blade Servers and Virtualization: Transforming Enterprise Computing*

    While Cutting Costs. Indianapolis: Wiley Publishing.

Hasan, J., & Tu, K. (2003). *Performance Tuning and Optimizing ASP.NET Applications.* New York: Apress.

Internet Information Services. (2009, December 26). Retrieved December 28, 2009, from Wikipedia, The Free

    Encyclopedia: http://en.wikipedia.org/w/index.php?title=Internet_Information_Services&oldid=334129117

Kumar, S. S., & Kumar, N. S. (2008). *Software Testing with Visual Studio Team System 2008—A Comprehensive*

    and Concise Guide to Testing Your Software Applications with Visual Studio Team System 2008 .

    Birmingham, UK: Packt.

Language Integrated Query. (2009, December 26). Retrieved December 28, 2009, from Wikipedia, The Free

    Encyclopedia: http://en.wikipedia.org/w/index.php?title=Language_Integrated_Query&oldid=334107132

Levinson, J., & Nelson, D. (2006). *Pro Visual Studio 2005 Team System.* Berkeley: Apress.

Lewis, W. (2009). *Software Testing and Continuous Quality Improvement, Third Edition.* Boca Raton: Auerback

    Publications.

M2 Presswire. (2001, May). *$25 billion in web business potentially lost due to poor web performance*. Retrieved

    December 27, 2009, from Bnet:

    http://findarticles.com/p/articles/mi_hb5243/is_200105/ai_n20457563/?tag=content;col1

Meier, J., Taylor, J., Mackman, A., Bansode, P., & Jones, K. (2008). *Team Development with Visual Studio Team

    Foundation Server: Patterns & Practices.* Redmond: Microsoft Press.

Meier, J., Vasireddy, S., Babbar, A., & Mackman, A. (2004). *Improving .NET Application Performance and

    Scalability: Patterns & Practices.* Redmond: Microsoft Press.

Meiers, J., Farre, C., Bansode, P., Barber, S., & Rea, D. (2007). *Performance Testing Guidance for Web

    Applications—Patterns & Practices.* Redmond: Microsoft Press.

Microsoft Application Consulting and Engineering Team. (2003). *Performance Testing Microsoft .NET Web

    Applications.* Redmond: Microsoft Press.

Muller, A., Wilson, S., Happe, D., & Humphrey, G. J. (2005). *Virtualization with VMware ESX Server.* Rockland,

    MD: Syngress Publishing.

Multitier architecture. (2009, December 3). Retrieved December 28, 2009, from Wikipedia, The Free

    Encyclopedia: http://en.wikipedia.org/w/index.php?title=Client-server&oldid=330098673

Newkirk, J. W., & Vorontsov, A. A. (2004). *Test-Driven Development in Microsoft .NET.* Redmond: Microsoft

    Press.

Non-functional testing. (2009, December 17). Retrieved December 27, 2009, from Wikipedia, The Free

    Encyclopedia: http://en.wikipedia.org/w/index.php?title=Non-functional_testing&oldid=332173398

Patel, A., Reinstrom, J., Siefkes, K., Silva, P., Ulrich, S., Yeung, W., et al. (2008). *Using Rational Performance

    Tester Version 7 .* IBM Redbooks.

Petri, D. (2009, January 8). *Improve Virtual Machine Performance.* Retrieved December 30, 2009, from Petri IT

    Knowledgebase: http://www.petri.co.il/improve-virtual-machine-performance.htm

Randolph, N., & Gardner, D. (2008). *Professional Visual Studio 2008.* Indianapolis: Wiley.

Subraya, B. M. (2006). *Integrated Approach to Web Performance Testing: A Practitioner's Guide.* Hershey, PA:

    IRM Press.

System testing. (2009, May 5). Retrieved December 27, 2009, from Wikipedia, The Free Encyclopedia:

    http://en.wikipedia.org/w/index.php?title=System_testing&oldid=329864583

Tulloch, M., & Tulloch, I. (2002). *Microsoft Encyclopedia of Networking, 2nd Ed.* Redmond: Microsoft Press.

Volodarsky, M., Londer, O., Hill, B., Cheah, B., Schofield, S., Aguilar Mares, C., et al. (2008). *Internet Information*

    Services (IIS) 7.0 Resource Kit. Redmond: Microsoft Press.

Wolf, C., & Halter, E. M. (2008). *Virtualization: From the Desktop to the Enterprise.* Berkeley: Apress.

Wort, S., Bolton, C., Langford, J., Cape, M., Jin, J. J., Hinson, D., et al. (2008). *Professional SQL Server 2005*

    Performance Tuning. Indianapolis: Wiley Publishing.