@Bench Web Services
Performance Benchmark Study

April 2003
# Table of Contents

**Executive Summary** .................................................................................................................. 3
**Test Results Summary** ................................................................................................................ 5

**Overview of Web Services** ........................................................................................................ 7
**The Need for Testing Web Services Performance** ...................................................................... 8

**@Bench Web Services Performance Test Methodology** ............................................................ 9
**Test Application** .................................................................................................................................. 9
**Test Environment** ................................................................................................................................ 13
**Test Automation Approach** ........................................................................................................ 17
**Test Process** ..................................................................................................................................... 18

**Application Server Platforms Tested** .......................................................................................... 22
**Implementation Architecture** ........................................................................................................ 22
**Code Review** ..................................................................................................................................... 25
**Test Configurations and Tuning Optimizations Considered** ...................................................... 26

**Findings and Analysis** .................................................................................................................... 28
**Transaction and User Load Performance Results** ....................................................................... 28
**Scalability Results** ........................................................................................................................ 33
**Interoperability Results** ................................................................................................................. 35
**Summary Results: Application Server X on Windows 2000 and Oracle9i** ................................. 40
**Summary Results: Application Server Y on Windows 2000 and Oracle9i** ................................. 42
**Summary Results: JBoss on Red Hat Linux 8.0 and Oracle9i** ...................................................... 43
**Summary Results: .NET 1.1 on Windows 2000 and Oracle9i** ....................................................... 45
**Summary Results: .NET 1.1 on Windows 2003 and Oracle9i** ....................................................... 46
**Summary Results: .NET 1.1 on Windows 2003 and SQL Server 2000** ......................................... 47

**Conclusion** ...................................................................................................................................... 48

**Appendices** .................................................................................................................................... 50
**Appendix A: Database Schema** .................................................................................................. 50
**Appendix B: Sample SOAP Responses** ..................................................................................... 52
**Appendix C: Web Client HTML Files** ........................................................................................ 55
**Appendix D: Code Reviews** ......................................................................................................... 56
**Appendix E: Changes based on Code Reviews** ........................................................................ 62
**Appendix F: Detailed Results** ..................................................................................................... 64
**Appendix G: Test Application Code – J2EE Implementation** .................................................. 79
**Appendix H: Test Application Code – .NET Implementation** .................................................. 99
**Appendix I: About Doculabs** ..................................................................................................... 120
Executive Summary

Web Services are quickly becoming a strategic technology in organizations of all sizes, thanks to Web Services’ promise of interoperability between technologies and platforms, and ease of development and deployment. As organizations spend more time and money investigating how best to leverage Web Services and its enabling technologies, they should be aware of the strengths and limitations of the technology – specifically, those related to performance and scalability, interoperability, and security.

To help organizations better understand the first two aspects of a Web Services implementation (performance and scalability, and interoperability), Doculabs developed a performance benchmark program called @Bench Web Services. Doculabs’ goal for this study was to compare the Web Services performance characteristics of commonly used application server platforms, and to communicate the resulting analysis to the market to aid end-user organizations in making informed decisions about technology selection and capacity planning.

The evaluation was designed to be highly demanding on application servers and their associated infrastructure technologies. A number of vendors were contacted and invited to participate in this paid evaluation, including BEA, IBM, JBoss, Microsoft, Oracle, and others. Specifically, these vendors were invited to contribute to the specification, provide reference implementation of the specification, and be present for installation and tuning of their application server environments.

Microsoft was the only vendor that accepted the invitation to participate. To provide a useful comparison between J2EE-based application servers and .NET, however, three other J2EE-based application servers were also evaluated. JBoss was one of the three and two others were selected at random from a larger set of industry leaders. These J2EE-based application servers will be identified as Application Server X and Application Server Y due to benchmark disclosure restrictions present in their end-user licensing agreements.

Doculabs conducted testing at a lab facility made available by Microsoft, with server, data storage, and network equipment donated by Hewlett Packard for use in the tests. For the duration of the testing, Doculabs had full control of this lab facility and certifies all results as valid and fair to all tested products.
All implementations, including .NET, were tested against a backend Oracle 9i database for the core comparisons. .NET was additionally tested against SQL Server 2000 at the request of Microsoft.

The evaluation consisted of a performance and scalability test, as well as an interoperability test. The methodology called for the development of a simple set of Web Services. To ensure a completely balanced and fair evaluation, an independent third-party development firm was used to develop a reference implementation of the @Bench Web Services specification for both J2EE and .NET. In addition, both of the reference implementations were reviewed by two independent J2EE-certified development firms to determine if the code was skewed in favor of any one vendor based on standard development practices. These companies were BrowserMedia of Baltimore, Maryland and UrbanCode of Cleveland, Ohio.

The results of their reviews are available in the Appendix of this report, as well as a list of changes made to code based on their reviews. Based on their reviews and incorporation of certain J2EE-optimizations, the development firms determined that both the .NET and J2EE implementations adhered to the specification, and represented best practice development techniques for each platform.

Mercury Interactive LoadRunner 7.5 was chosen as the testing software, given its stature as the industry-leading test tool, and its ability to directly test Web Services with its SOAP client driver.

All testing was conducted on industry-leading Intel-based Hewlett Packard ProLiant enterprise servers, including both DL760 and DL580 ProLiant Servers. Full details on the hardware used in testing are available in the Appendix.

This document provides the results from the performance evaluation, as well as a general overview and specification of the study’s test application, test environment (including hardware, software, and database), and test process (including details on the test methodology, the variables measured, and the testing approach).
Test Results Summary

The following are the significant highlights and conclusions from the performance evaluation:

- Microsoft .NET provided the best performance among the products evaluated in all tests conducted.
  - It is believed that the performance results for .NET are due to the fact that Web Services were part of the core .NET framework from the start, and not added features on top of an existing framework (as is the case with J2EE).

- JBoss performed better than the two leading commercially available, J2EE-certified application servers.

- There was wide variation in performance between all three J2EE-based application server products.

- The results from the interoperability tests prove that with Web Services, interoperability between .NET and J2EE platforms is a reality, and performance is not a blocking issue in such cases.

- Against a backend Oracle 9i database, .NET on Windows Server 2003 (8 CPU) achieved 110 percent better throughput than the leading J2EE product against the same Oracle 9i database.

- The Java Bean implementation of the test application yielded only marginal performance gain compared to the EJB implementation.
  - The performance of the EJB implementation (using a stateless session bean with local interfaces) is possibly better in this application because of the relatively low number of objects and the stateless nature of the Web Services.

- While the application servers did demonstrate better performance when scaling from four processors to eight processors (both for Windows and Linux), the additional performance gained when moving from four processors to eight processors was not as significant as expected.
  - This is likely due to the fact that Web Services are network-intensive and such applications likely benefit more from clustering multiple machines (as opposed to adding many processors to a single machine).
Organizations should understand that a combination of scaling up (adding more processors to a single system) and scaling out (adding more computers to a cluster) should be used to provide the necessary capacity to host a Web Service, while reducing the overall number of servers required in the cluster (in order to reduce hardware costs and ongoing management costs).

Due to the stateless nature of Web Services, application servers operating in a clustered mode can be expected to deliver near linear scalability. So when it comes to hosting Web Services, it may be beneficial for organizations to consider scaling out two-and four-processor systems (adding more servers) rather than scaling up (adding more than four processors).

- While JBoss delivered the best J2EE-based Web Services host results, Application Server Y delivered the best J2EE-based Web Services client results. Overall, .NET provided significantly better performance results versus the J2EE products acting as both a Web Service client and a Web Service host.

- .NET 1.1 framework running on Windows 2003 with SQL Server 2000 database achieved the best throughput results for direct Web Service invocation test on an 8 CPU host, at 1488.53 transactions per second (1072.90 transactions per second on a 4 CPU host), followed by .NET 1.1 running on Windows 2003 and Windows 2000 with Oracle9i as the backend database. The fastest J2EE application server performance was achieved by JBoss running on Red Hat Linux 8.0 at 513.33 transactions per second (354.25 transactions per second on a 4 CPU host) and followed by Application Server Y at 411.87 transactions per second (312.70 transactions per second on a 4 CPU host) and Application Server X at 288.26 transactions per second (213.37 transactions per second on a 4 CPU host).

- This evaluation showed that organizations could expect ultra-high-performance (millions of transactions per day) related to serving Web Services using an Intel-based 4-CPU server from Hewlett-Packard. Combined with the near linear scalability when adding new machines in parallel, organizations should be able to handle most of their Web Services needs with ease.

- Though reliability and failover were not tested directly in this evaluation, all of the evaluated applications performed well and had little or no problems during testing. The overall reliability features of the major application server products in the market have improved dramatically.
Overview of Web Services

Many organizations are realizing that there are solid business reasons to invest in Web Services-enabled technologies to solve their integration problems. Many companies are already benefiting from implementing Web Services and are able to reduce development and maintenance efforts, and respond more quickly to organizational demands for new applications and services.

Most companies maintain silos of information, distributed across applications at divisional or departmental levels that typically represent sizable investments. For an organization to realize enterprise-wide benefits, it must aggregate these disparate systems. By simply adding a Web Services interface to existing systems, organizations are finding that Web Services offer faster and more cost-effective solutions to interoperability and integration problems, while also enabling it to leverage its existing infrastructure and reuse technology components.

The Web Services approach also eliminates the complexity and expense of traditional application integration methods. Vendors in virtually all technology areas are continuing to develop support for Web Services, and making Web Services part of their core offerings. Moreover, Web Services are already becoming the primary method of communication between portal servers and remote services, and are creating greater interoperability between portal services, content management systems, business process management systems, and a variety of other enterprise applications.

Implementing Web Services should result in the following benefits:

- Reduced development and deployment costs
- Seamless interoperability between computing frameworks such as Microsoft .NET and Java-based solutions
- Seamless interoperability between computing platforms such as Microsoft Windows, Sun Solaris, Linux, IBM AIX and zOS, HP-UX, and others
- Faster and cheaper integration with third-party applications
- Greater leverage in technology put in place to secure TCP/IP networks, particularly those using HTTP and SSL
- Increased application re-use throughout an organization
The Need for Testing Web Services Performance

Performance and scalability can be impacted by any number of factors, including the configuration of systems running the application server software, network configurations, maximum data transfer rates associated with the different devices, other enterprise applications participating in integration processes, and the back-end database systems.

The only way to determine the limits of Web Services performance and scalability is to run real-life tests that reflect the multitude of factors that can affect those attributes. With this in mind, the specific goals of this benchmark were to:

- Demonstrate that the leading application servers in the market have the ability to process a large number of Web Service transactions per second, in an environment representing a real customer scenario involving multiple systems and high user loads.
- Demonstrate that the application server systems hosting the Web Services can support high user loads, while maintaining a stable CPU utilization and acceptable response times.
- Demonstrate that interoperability of Web Services is a reality between platforms and to understand performance implications of calling Web Services that reside on different platforms like J2EE and .NET.
- Understand the performance implications of invoking Web Services directly from many distributed clients versus invoking them in a typical enterprise application integration scenario using an application server as a Web Service client, providing an integration portal to backend Web Service hosts.
- Understand the scalability characteristics of various application servers running on different platforms, to determine if end-user organizations can achieve performance gains through adding processors.
- Compare the Web Services performance characteristics of commonly used application servers in the market, and to communicate Doculabs’ analysis to the market to aid end-user organizations in making informed decisions for technology selection and capacity planning.
@Bench Web Services Performance Test Methodology

This section describes the @Bench Web Services Performance Test methodology.

Test Application

This subsection describes the test application used in the benchmark program. The pseudo-code descriptions of each Web Services interface for the application and sample SOAP responses are included in the Appendix.

Test Application Overview and Functional Description

For this testing program, all vendors used the same application based on an underlying customer scenario. This scenario involves taking existing business logic and exposing it as a callable Web Service (that is, SOAP-based accessible objects) such that business logic can be easily integrated with other enterprise applications, or potentially called by and integrated into business partner applications across the Internet. Thus, the focus of the evaluation is not on the productivity of the application servers at implementing the core business logic; rather, the focus is on the following:

- How easy each application server and corresponding tools make it to expose existing business logic as a Web Service. This is captured using developer productivity metrics.
- How well the Web Services perform/scale under a very high number of concurrent users. This is captured through an extensive series of scalability and performance tests.

The test application was designed to provide the simplest possible configuration to isolate the real Web Services performance and scalability characteristics of the underlying application server that is running them. The application runs against a simple database, which consists of several million rows of data. The large volume of data ensures that the effect of excessive positive cache hits is minimized, a factor that would normally skew performance results significantly.
Among the key design choices and assumptions made to keep the test straightforward, fair, and easy to implement are the following:

- Each Web Service acts independently, and does not rely on another Web Service except in the case of the specialized interoperability test.
- Only basic error catching is necessary since the test scripts will not make “bad requests”.
- Web Services will be written using tools such as integrated development environments (IDEs) that are commonly bundled or provided with a particular product.
- Web Services will be generated using standard, out-of-the-box functionality wherever possible, and custom optimizations will be limited to fixing obvious problems generated by poorly written wizards or tools.

Doculabs audited all source code to ensure that it meets the test specification. In addition, Doculabs reviewed any customizations and optimizations to determine whether to accept code for use in the testing. Finally, Doculabs additionally contracted two independent J2EE-based development organizations to audit both the .NET and J2EE implementations to ensure equivalence in implementation architectures (to extent possible given platform differences), and adherence to best-practice J2EE development techniques for high performance scenarios. The results of these audits are available in the Appendices.

**User Actions**

The load-test software will make requests to the application servers just as a user may in a real-world application. The actions that can be performed are purposely very limited in this benchmark. The actions are implemented as SOAP methods generated by the application server that expose existing business logic (also running within the application server) as SOAP objects. Note that while the focus of the study is on exposing existing application logic as Web Services, the project necessitates that the application logic itself be implemented in both .NET and J2EE and configured/tuned correctly for the various application servers in the test.
The application logic was exposed as three Web Service (SOAP) methods as follows:

• **Request for Order (GetOrderDetails)**
  This transaction returns a complex data type of type Order. The Order class has both simple and complex data types. Simple data types include strings, dates, and integers. Complex data types will be objects including LineItems (repeating element for each item in an order), and a Customer object representing the customer information for the order. The input from the client was a random valid order number. Note that the data load contained from 1 to 10 line items (randomly created) for each order, evenly distributed through 1 million total orders. This tested the ability of the application server to serialize objects and transmit XML/SOAP messages of varying sizes over the network.

• **Retrieve a Customer Object (GetCustomer)**
  This transaction returns the Customer object that represents a customer record. The Customer object contained simple and complex types, including simple strings and integers, as well as two Address structures that contain billing and shipping information. The Customer object is read from the database based on an input customer ID, and formatted into a Customer object instance returned by the Web Service as a SOAP object.

• **Insert a New Customer**
  This transaction will insert a new customer record to the customer table in the database. The input will be a valid Customer object of well-formed customer data. The output will be a customer ID returned as an identity seed/sequence number from the database:

  o The new customer ID will be returned for a successful insert
  o Negative 1 will be returned if the insert fails for some reason

These actions include a mix of simple queries, as well as actual transactions that write to a database. This mix of actions is typical of web-based applications that may use Web Services.
**Database**

The database to be used for this application contained several million rows of data. The database contains generated data, which makes it possible for the test scripts to request very random data. An example is the descriptions of products in the product table may look like “Product Description 1,” “Product Description 2,” “Product Description 93939,” etc.

The database contained the following tables:

- Orders
- LineItems
- Products
- Customers

**Database Record Caching/Data Staleness**

All data records were read from the database on every request. No data caching was allowed, and the implementation for both J2EE and .NET required that each select be executed on every request such that the data was guaranteed to be up to date based on what was stored in the database. In addition, all Web Service requests were made in a synchronous fashion. No stored procedures were allowed for either the Oracle or SQL Server implementations, so both J2EE and .NET used dynamic SQL in all cases from the data access layer.
Test Environment

This subsection provides relevant details about the environment and configuration that was used in the performance benchmark testing program. This includes details on:

- Hewlett Packard ProLiant Server hardware
- Network configuration

Server Hardware

One of the design goals of this benchmark was to use a fixed set of hardware for all vendors. However, vendors had some flexibility to configure the hardware in a fashion that made the most sense for their particular product architecture or application deployment.

The primary hardware server components that were used in the testing were as follows:

- Two (2) quad-processor HP Proliant DL580 servers (with 1.4 GHz Intel Pentium XEON processors) that can be used in any combination for the web server and application server. These machines have standard (unchangeable) configurations for memory, disk space, processor speed, and network interface cards.

- One (1) eight-processor HP Proliant DL760 (with 900 MHz processors) server that was used strictly for hosting the Web Services application server testing.

- One (1) eight-processor Intel server that acted as the database server. This machine had to be used as the database server, and was not allowed to be swapped with one of the other machines.

The following table summarizes the server hardware used in the benchmark testing.

<table>
<thead>
<tr>
<th>Server</th>
<th>CPU</th>
<th>Memory</th>
<th>Disk</th>
<th>NIC(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two (2) Web/Application Servers (Hewlett Packard DL580)</td>
<td>4 x 1.4 GHz</td>
<td>2 GB</td>
<td>1 x 18.2 GB Ultra SCSI 3</td>
<td>2 @ 1,000 mbps</td>
</tr>
<tr>
<td>One (1) Web Services Application Server (Hewlett Packard DL760)</td>
<td>8 x 900 MHz</td>
<td>4 GB</td>
<td>1 x 18.2 GB Ultra SCSI 3</td>
<td>2 @ 1,000 mbps</td>
</tr>
<tr>
<td>One (1) Database Server (Hewlett Packard DL760)</td>
<td>8 x 900 MHz</td>
<td>4 GB</td>
<td>1 x 18.2 GB Ultra SCSI 3</td>
<td>1 @ 1,000 mbps</td>
</tr>
</tbody>
</table>

Table 1 – Server Hardware for @Bench Test Environment
**Network Configuration**

A consistent network configuration was maintained across all testing. A three-tier architecture was utilized for the web, application, and database servers on a flat network topology, with the only changes to test configurations being the addition of servers for the web and application server layers of the three tiers.

One hundred (100) client machines were used for the testing, with each running the Mercury Interactive LoadRunner client software to execute test scripts, submit requests, and monitor results.

The lab configuration was based on a flat network topology. All one hundred (100) clients were connected to Cisco 3548G network switches that in turn had two Gigabit uplink ports grouped together. This insures that there is 2 GB worth of bandwidth between the clients and the aggregator at any given time.

The Aggregator was a 3508G 8 port Cisco Gigabit Switch. This aggregator also provided for direct connection of the Mercury Controller, as well as port grouping of the dedicated gigabit switch that the servers are connected to. The servers share a single Cisco 3508G such that the maximum bandwidth was available (10GB on the switch’s backplane) and that there was no more than 1 network hop for any server-to-server network request. The server side gigabit switch was port grouped to provide 2GB worth of cross-linked bandwidth to the aggregator.
The following figure shows the topology of the @Bench test environment where clients make SOAP/HTTP Web Services requests directly to the Web Services application server.

![Figure 1 – Test Environment for Invoking Web Services Directly (Source: CN2 Technology)](image-url)
The following figure shows the topology of the @Bench test environment where clients make SOAP/HTTP Web Services requests to the Web Service client application server.

Figure 2 – Test Environment for Invoking Web Services Via Client Application Server (Source: CN2 Technology)
The following table provides detailed information on the network configuration used for testing.

<table>
<thead>
<tr>
<th>Component</th>
<th>Details</th>
</tr>
</thead>
</table>
| Clients                          | • One hundred (100) 1.8 GHz Intel Celeron machines (each with 256 MB RAM, 4 GB HDD)  
  • Operating System: Windows XP Professional  
  • Browser: Microsoft Internet Explorer 5.5 or higher  
  • NIC: one integrated 10/100 NIC  
  • Clients divided into five groups of 20 machines |
| Testing automation software      | • Mercury Interactive LoadRunner 7.5  
  • Software was used to submit web requests (both HTTP and SOAP) and parse returned pages |

*Table 2 – Test Environment Components*

**Test Automation Approach**

The benchmark testing was conducted using the Mercury Interactive LoadRunner 7.5 test suite. This software can simulate tens of thousands of users using commonly available Windows-based client computers.

Doculabs ran a mix of three scripts, with each script executing one of the methods of the Web Service object. Each script ran in a 1/3 mix. Doculabs also performed manual transactions during testing using a Web browser.

The client-interface to test the Web Service was:

- Web Service Direct Invocation Tests: LoadRunner’s SOAP client making direct SOAP over HTTP calls to the Web Service from the load generators.
- LoadRunner’s HTML client making HTTP requests to a JSP/ASPX page running on the application server tier that, in turn, made remote SOAP calls to the Web Service Host computer.
- There was extensive manual auditing of returned pages for page size, content, database row insertion, as well as source code review and collection. These measures are designed to ensure the accuracy and validity of the testing across a variety of vendor solutions.
Test Process

This section explains how the various tests were executed, with descriptions of the testing, the variables, and the approach. The types of testing conducted as part of this benchmark included:

- Performance tests
- Scalability tests
- Interoperability tests

The following table provides a high-level summary of the three types of testing that were conducted as part of this benchmarking effort.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| Performance tests   | • Designed to measure how many dynamic requests can be processed per second over a fixed amount of time (throughput), as well as the average response time for each client. Both measurements were calculated under varying user loads.  
  • The first set of tests were setup so that the clients were making direct SOAP requests to the Web Services application server, measuring the performance characteristics of the application server hosting the Web Service.  
  • The second set of tests were configured so that a pool of clients were submitting requests to a cluster of web/application servers that in turn made remote SOAP/Web Service calls to a second application server hosting the Web Service. This tests the SOAP client, as well as the SOAP server stack of the application server.  
  • The number of concurrent users (user load) was steadily increased to see when each of the application server response times begins to degrade, and how long users have to wait for each page as the load increases (up to a maximum acceptable time of 3 seconds for each transaction).  
  • Generally speaking, response times do not significantly increase as user load is added until a bottleneck is reached. In this testing, the network and database were scaled to ensure the tests accurately measured application tier capacity. Hence, once the application tier became saturated, response times would start to rapidly rise. The user load at which this change in slope was observed is the “elbow” in the performance curve, and was used to calculate the maximum number of users the application server can be expected to support. This elbow typically occurs at the point maximum throughput is observed.  
  • In addition, CPU utilization on the various servers in the testing environment was monitored to understand the effects of load on various parts of the infrastructure. |
| Scalability test     | • This test is designed to evaluate the ability to scale an application across many CPUs (to determine the capacity and/or performance gained through adding processors).  
  • This is accomplished by running the basic performance tests on an application server configured with 4 and 8 CPUs (Invoking the Web Services directly) and comparing throughput curves, as well as comparing maximum number of users supported at peak throughput (transactions per second). Similarly the scalability tests were run with 2 and 4 CPUs on the client application servers that in turn called Web Services hosted on an application server running 8 CPUs. |
| Interoperability test| • This test was designed to determine the performance implications of calling Web Services that reside on different platforms.  
  • Two tests were conducted with:  
    o Web Services hosted on the fastest J2EE application server being called from the fastest .NET client (8 CPU host, 4 CPU client)  
    o Web Services hosted on the fastest .NET application server being called from the fastest J2EE client (8 CPU host, 4 CPU client) |

Table 3 – @Bench Test Suite Descriptions
**Performance Testing**

“Performance” refers to the capacity of the server to generate dynamic pages (throughput), as well as the average response time for each client. Both throughput and response time will be impacted as the number of requests increases and as user load increases.

The performance metrics captured as part of the performance testing include:

- Throughput (transactions per second at a given user load, e.g. each getCustomer, getOrderDetails, and insertCustomer call is considered a transaction)
- Response time (the time required to process each client request)
- Total system CPU utilization

**Performance Test Configurations**

The vendors were restricted to using the servers as designated in this document. The network topology was not modified in a way that removed components such as the firewall or routers.

Throughput, response time, error rates, and total CPU utilization were measured as the number of requests and the number of concurrent users increased (through the execution of test scripts and requests submitted from client machines).

**Performance Test Details**

The performance testing was executed as follows:

- In a first set of tests (**Scenario A – Direct Invocation**), the clients were configured to make SOAP/HTTP Web Services requests directly to the Web Services application server.
- In a second set of tests (**Scenario B – via Web Service Client**), hardware/software was configured so that the load generating clients were submitting requests to an application server (acting as an integration server) that then made remote Web Service requests to the application server hosting the Web Service itself. A simple HTML page was returned to the load-generating clients consisting of data requested/returned from the remote Web Service.
In each test suite, the number of concurrent users (user load) was steadily increased (typically 250 users every 15 minutes) to see when each of the application server response times started to degrade, and how long users had to wait for each page as the load increases. Doculabs used a random client “think time” of between 2 and 8 seconds (average: 5-second think time).

Doculabs defined the maximum acceptable response time to be three (3) seconds. (NOTE: Doculabs realizes that in a real-world scenario, network latency can also add significantly to the user-perceived response time – thus making this three-second requirement equivalent to 5 to 10 seconds of round-trip time, depending on the speed of the user’s Internet connection.)

Throughout the course of the testing, CPU utilization on all tiers (web/application server; Web Service application server; and database) was logged and recorded for future use.

**Scalability Testing**

Many enterprise-level applications require high performance – and the quickest way to boost throughput is by adding more hardware. One way to scale an environment is to add processors to existing servers (“scaling up”). This approach allows organizations to reduce the maintenance and operational management overhead of running clusters with a very large numbers of servers.

Ideally, each processor added to a system would increase peak bandwidth by a consistent amount, thus making the scaling linear or near-linear. For example, a Web Services server that can service 200 requests per second with a single processor in the ideal case would be able to process 400 requests per second with two processors, and 800 requests per second with four processors (representing a linear scaling factor of 1.00.) In reality, in any system there is overhead for SMP processing, and linear scaling of 1.00 is not observed. Moreover, as more and more processors are added to a system, typically the slope of the scaling begins to degrade. For the scalability testing, Doculabs measured throughput, response time, and CPU utilization. The overall test environment consisted of one (1) web/application server, one (1) Web Services Host application server, and one (1) database server.
To measure scalability, four separate tests were run, each with a different number of processors running the application server software. Specifically, we collected test data for the following application server configurations:

- Direct Invocation – Running application server software on four (4) processors and eight (8) processors
- Via Web Services Client – Running application server software on two (2) processors and four (4) processors, calling Web Services hosted on another application server always configured with 8 processors.

**Interoperability Testing**

Prospective buyers of Web Services technology often ask whether Web Services are really interoperable between .NET and J2EE application platforms. While Web Services are interoperable in concept, there may be differences in the way they execute when accessing services running on different platforms, or when being accessed by different platforms. This test was designed to capture empirical data about the characteristics of Web Services when operating in a mixed .NET and J2EE environment.

Doculabs conducted two tests to determine each technology’s interoperability characteristics, including:

- Web Services hosted on the fastest J2EE application server being called from the .NET Web Services client application (8 CPU J2EE application server hosting the Web Service, 4 CPU .NET application server making requests to the remote Web Service Host)
- Web Services hosted on the .NET application server (Windows 2003/.NET 1.1/to Oracle 9i) being called from the fastest J2EE client application (8 CPU .NET application server hosting the Web Service, 4 CPU J2EE client application making requests to the remote Web Service host)
Application Server Platforms Tested

This section describes the architectures for the J2EE and .NET implementations of the test application along with information on the code reviews performed by two independent development organizations. It also includes the final list of server platforms tested and the tuning optimizations considered to determine the final test configurations.

Implementation Architecture

In general, the implementation architecture was left to the contract development firm for both .NET and J2EE, which were given the directive to use “best practice development guidelines” for each platform. One key rule that was adhered to, however, was that the application implementation logic to be exposed via the Web Service façade be implemented in fully encapsulated middle tier components (namely, EJBs or .NET assemblies with public methods exposed as Web Service methods), and that these components be invoked by a separate Web Service Class (the Web Service façade) that makes calls into these application logic classes. The Web Service façade and application logic classes should be deployable to different physical machines, or they may reside on the same physical machine depending on deployment considerations (in the case of our testing, they resided on the same physical machine).

This architecture was chosen since it represents the common case of the customer taking existing application logic, and exposing it via a Web Service façade to other external applications for the purpose of integration.

The same contract development firms developed both the .NET and J2EE reference implementations based on vendor-recommended best practices, in consultation with vendor documentation/sample applications/product support services. The contract development firm used their experience building high-scale systems to extensively tune their .NET and J2EE reference implementations to ensure they were optimized for performance. In addition, both implementations were provided to two external J2EE development firms for audit/review. Feedback received was incorporated into the applications (or tested in addition to base architecture to show how results differed).
.NET Implementation

For the .NET implementation C# was the language chosen for all tiers of the application; however, VB.NET could have been used for any or all of the tiers in the application. In .NET, Web Service façades are typically built using an asmx page, allowing the exposure of methods using tags in the application code. By simply adding [WebMethod] tag in front of a method, a business function can be exposed as a Web Service and .NET automatically generates the appropriate WSDL and provided automatic serialization/mapping of native .NET objects/types to SOAP objects/types via XML. IIS (Internet Information Services) is typically used to host the Web Service, requiring only the creation of a virtual directory in IIS based on the folder where the asmx file resides on the server’s file system.

The business logic for the application was written in C# and was used to access data access classes also written in C#. The data access classes made calls to the ADO.NET framework libraries. To access the database tables, prepared statements were used as they provide a secure mechanism to make database calls while offering similar performance to stored procedures. Both the business logic classes and the data access classes were compiled into a single .NET assembly that was then copied to the bin directory in the Web Services virtual directory folder.

The .NET implementation used the built in .NET OracleClient classes to access the Oracle database. This is a new high-performance managed-code provider introduced with .NET 1.1. In the case of a SQL Server backend, the .NET application used the built in SQLClient classes that similarly is a managed-code interface to SQL Server databases.

The relationship between the application logic (as implemented in business objects) and the Web Services façade to be implemented on top of these objects is depicted below.

![Figure 3 – Microsoft .NET Architecture for Web Services](image-url)
**J2EE Implementation**

While Java was used as the development language, a very similar architecture was used for the J2EE implementation. Most J2EE application server vendors have implemented Web Services in a slightly different manner; for example, some use either the Apache SOAP or Apache AXIS Web Serviced tool kits, while some have proprietary solutions. Each vendor’s toolset was used to create any code required to expose objects as Web Services.

The Java implementations used Stateless Session EJBs to implement the business logic. Stateless Session EJBs were used because they provide a mechanism by which the objects can be accessed remotely, and they allow the business components to have transactional contexts so that they can perform multiple database calls as atomic operations. Entity beans could also have been used but they are generally recognized to offer less performance in return for having to write less database access logic.

To access the database the test used the latest JDBC drivers, which were pre-configured in the application servers to be obtainable through the application server’s data sources. As with the .NET implementation, prepared statements were used to access the database. For each application server, all the Java code was pre-compiled and collected into war and ear files for deployment on the server.

![Diagram](image.png)

*Figure 4 – J2EE Web Services Implementation using EJBs*
Code Review

Two development organizations agreed to review the code and determine if the code was skewed in favor of any one vendor based on standard development practices. These companies were BrowserMedia of Baltimore, Maryland and UrbanCode of Cleveland, Ohio. Both companies reviewed the code developed for the 3 J2EE application servers and Microsoft .NET implementations. A number of factors were considered while reviewing the code for various implementations. The detailed results of these reviews and a list of changes made to code based on these reviews are available in the Appendix of this report.

Changes Based on Code Reviews

The following are some of the changes made to the various implementations based on recommendations made by the reviewing companies.

- Both the review companies pointed out that the initial implementation of wsFacade.java makes JNDI calls and that this should be cached. The application was modified to use static member variables in the wsFacade file to cache the EJB home object.

- Due to concerns about the use and performance overhead of EJBs, a second Java implementation was created in which the EJBs were replaced by simple Java Beans. The implementation code was identical between the two versions except that the first implementation had the extra code required for the EJB interface classes for the business logic components. The Java Bean implementation was tested for each application server on a 4 CPU host.

A detailed list of changes made to the code and responses to the comments in the code reviews are included in the Appendix.
Test Configurations and Tuning Optimizations Considered

Each J2EE application server platform was tested on Windows 2000, Windows 2003, and Linux, and the fastest operating system for that application server was chosen for the final testing. A number of initial tests for each J2EE application server platform were conducted to determine the combination of the OS platform, JVM, and tuning parameters to use in the final tests.

For each combination of operating system platform and J2EE application server, the following performance optimizations were considered.

- Varying thread pool size in EJB container
- Varying thread pool size in web container
- Changing the database connection pool size, and JDBC drivers to use
- Testing various JVMs and JVM settings to use
- Changing Garbage collection methodology
- Use of JVM clones

The following configuration changes were tested during tunings.

- Changing the web container thread pool size. (min and max set to the same)
- Setting processor affinity for individual VMs
- Having keep-alive connections enabled/disabled
- Trying different numbers of application server clones
- TCP configuration (max ports, time wait delay)

The following variables were monitored during tunings.

- CPU load
- DB load
- Thread context switches
- TCP connection status
- EJB Method invocation times
- Servlet utilization
- Thread pool utilization
- Verbose Garbage Collection output
After the above tuning optimization (two weeks of testing time), the following configurations were used for final testing of direct web services performance as they yielded the best performance:

- J2EE Application Server X running on Windows 2000 with Oracle9i R2 Enterprise Edition backend database
- J2EE Application Server Y running on Windows 2000 with Oracle9i R2 Enterprise Edition backend database
- JBoss 3.2 RC1 Application Server running on Red Hat Linux 8.0 with Oracle9i R2 Enterprise Edition backend database
- Microsoft .NET 1.1 framework running on Windows 2000 with Oracle9i R2 Enterprise Edition backend database
- Microsoft .NET 1.1 framework running on Windows 2003 with Oracle9i R2 Enterprise Edition backend database

For the Web Services client tests on the 4 CPU and 2 CPU servers:

- Application Servers X and Y, and .NET 1.1 frameworks were tested on the same platforms as the direct tests calling the corresponding Web Services hosts as mentioned above.
- JBoss Web Services clients were tested on Windows 2000 instead of Red Hat Linux as the Windows 2000 yielded better performance than Linux for the client case.

Oracle9i Enterprise Edition was chosen as the database to keep the backend database consistent for all application server platforms. For the last configuration running .NET 1.1 on Windows 2003, the backend database was replaced with SQL Server 2000 to determine performance gains in using SQL Server 2000 as the backend database on Windows 2003.
Findings and Analysis

This section summarizes the results of the performance tests. Throughput (transactions per second at a given user load), response time (the time required to process each client request) and total system CPU utilization on the application server were the measured parameters in all cases. This section also discusses Doculabs’ analysis of the testing results in terms of performance and scalability.

Transaction and User Load Performance Results

![Web Services Direct Performance on an Eight-CPU Host](image)

Figure 5 – Web Services Direct Performance on an 8 CPU Host

Findings

As the figure above shows, the .NET 1.1 framework running on Windows 2003 with SQL Server 2000 database achieved the best throughput results for direct Web Service invocation on a 8 CPU host, at 1488.53 transactions per second, followed by .NET 1.1 running on Windows 2003 and Windows 2000 with Oracle9i as the backend database. The fastest J2EE application server performance was achieved by JBoss running on Red Hat Linux 8.0 at 513.33 transactions per second and followed by Application Server Y at 411.87 transactions per second and Application Server X at 288.26 transactions per second.
Doculabs believes that the high performance results for .NET are because Web Services were part of the core .NET framework from the start and not added features on top of an existing framework, as is the case with J2EE. Given the dramatic difference in the core implementation of Web Services into .NET and J2EE products, it was very promising to see the performance results of J2EE products was respectable. Furthermore, the .NET Web Services stack was built on IIS, which is optimized to perform better on Windows 2003. Microsoft .NET’s packaged approach to Web Services compared to J2EE’s approach helped it to achieve better throughput results.

JBoss performed better than the two leading commercially available J2EE certified application servers, which was an interesting result given the relative price differential between JBoss and commercial J2EE application servers which typically cost $10,000 per CPU or more. .NET does not have such a separate licensing cost, it is provided as an integrated feature with Windows Server 2003.

.NET 1.1 framework running on Windows 2003 was able to process 38 percent more transactions per second with SQL Server 2000 compared to Oracle9i database. This was attributed by Microsoft to be due mostly to performance differences in the .NET database providers. Microsoft conveyed that .NET performance against Oracle back-ends is now significantly better than COM/ASP performance was against SQL Server on Windows 2000, although it still falls short of .NET performance on SQL Server. At the same time, our testing observed that .NET performance against Oracle was better than J2EE performance against Oracle for this suite of Web Service tests.
The following figure shows the Web Services direct performance on a 4 CPU host. The figure also compares the performance of EJB and non-EJB implementations for the J2EE application servers on a 4 CPU host.

**Figure 6 – Web Services Direct Performance on a 4 CPU Host Comparing EJB/Non EJB and .NET 1.1 Implementations**

**Findings**

As the figure above shows, the .NET 1.1 framework running on Windows 2003 with SQL Server 2000 database achieved the best throughput results for direct Web Service invocation on a 4 CPU host, at 1072.90 transactions per second, followed by .NET 1.1 running on Windows 2003 and Windows 2000 with Oracle9i as the backend database. The fastest J2EE application server performance was achieved by JBoss running on Red Hat Linux 8.0 at 354.25 transactions per second and followed by Application Server Y at 312.70 transactions per second and Application Server X at 213.37 transactions per second.

The Java Bean implementation of the test application yielded only marginal performance gain compared to the EJB implementation on the 4 CPU host. The performance of EJB implementation is good in this application due to the relatively low number of objects and the stateless nature of the application.
The following figure shows the user loads supported at the time the maximum transactions per second were achieved on the 4 CPU and 8 CPU direct Web Services tests.

![Web Services Direct Performance - User Loads Supported At Peak TPS](image)

**Figure 7 – Web Services Direct Performance: User Loads Supported at Peak TPS**

**Findings**

Figure 5 above shows the number of users supported for the 4 CPU and 8 CPU direct tests when the maximum throughput rate was achieved. The overall user load performance leader was .NET 1.1 on Windows 2003 with SQL Server 2000 backend at 8500, while JBoss running on Linux achieved the best results among the J2EE platforms at 3750 users for the 8 CPU direct test.
The following figure shows the user loads supported at the time the maximum transactions per second were achieved on the 2 CPU and 4 CPU Web Service client remote invocation tests. The databases shown in the figure were the back-end databases used by the remote eight-CPU Web services host.

![Web Services Client Performance - User Loads Supported At Peak TPS](image)

**Figure 8 – Web Services Client Performance – User Loads Supported at Peak TPS**

**Findings**

Application Servers X and Y performed better acting as Web Service clients than as direct Web Services hosts, which demonstrates good management of incoming Web Service requests and efficiently relaying the requests to the remote Web Service host. The rest of the application server platforms exhibited better relative performance acting as direct hosts for Web Services on similar hardware and software configurations.
Scalability Results

The following figure compares the throughput results obtained by the various application server platforms for the 4 CPU and 8 CPU direct Web Services tests.

Figure 9 – Web Services Direct Scalability from 4 CPU Host to 8 CPU Host

Findings

None of the application server platforms demonstrated enough additional throughputs to justify upgrading the server hardware from a 4 CPUs to 8 CPUs for using the servers as Web Service hosts. Although, JBoss running on Linux did not achieve the throughput rates achieved by .NET 1.1 on Windows 2003, it attained a scalability factor of .44 (1.00 is ideal) compared to 0.38 achieved by .NET 1.1 on Windows 2003. The scalability factors for these two platforms in terms of number of users supported at peak throughput were 0.66 and 0.48 respectively.
It is possible that Servers acting as Web Service hosts demonstrate better scalability from dual processor servers to 4 processor servers, but those tests were not performed due to time constraints. However, due to stateless nature of Web Services, since no state has to be managed, Application servers operating in a clustered mode can be expected to deliver near linear scalability. So, it might be beneficial for end-user organizations to consider scaling out (adding more 2 and/or 4 CPU servers) rather than just scaling up (add more processors) when it comes to hosting Web Services. Scale out has the additional benefit of providing failover and higher availability for the hosted Web Service.

The following figure compares the throughput results obtained by the various application server platforms for the 2 CPU and 4 CPU Web Services client tests.

![Figure 10 – Web Services Client Scalability from 2 CPU Client to 4 CPU Client](image)

**Findings**

For the Web Services client test case, most application server platforms exhibited a scalability of close to 60 percent or better. For practical real-world applications, 60 percent scalability is considered an acceptable justification for upgrading from a 2 CPU server to a 4 CPU server.
Interoperability Results

The interoperability tests were designed to capture empirical data about the characteristics of Web Services when operating in both .NET and J2EE environments.

Two tests were conducted. The first test environment consisted of JBoss running on Linux as the 8 CPU Web Services host (since it was the fastest J2EE Web Services host) and .NET 1.1 running on Windows 2003 as the 4 CPU client (since it produced the fastest Web Services client in the .NET environment).

The second test environment consisted of .NET running on Windows 2003 with against an Oracle 9i database as the 8 CPU Web Services host (since Windows Server 2003 was the fastest in the .NET environment) and Application Server Y running Windows 2000 as the 4 CPU client (since it produced the fastest Web Services client in the J2EE environment).

The following figure shows the throughput results from the interoperability tests.

![Web Services Interoperability Performance](image)

*Figure 11 – Web Services Interoperability Performance*
Interoperability test with JBoss / RH Linux 8.0 / Oracle9i Database as Host and .NET 1.1 / Windows 2003 as Web Service Client

.NET 1.1 running on Windows 2003 as the Web Service client application server against an 8 CPU Web Services host running JBoss on Linux 8.0 with Oracle9i achieved a throughput rate of 392.78 transactions per second while supporting a user load of 2250 users. The JBoss application server acting as a Web Service client to the same 8 CPU Web Services Host (J2EE client to J2EE Host) achieved a throughput rate of 187.5 transactions per second. The peak CPU utilizations on the 8 CPU server hosting the Web Services and the 4 CPU client during this test were 85% and 80% respectively.

Interoperability test with .NET 1.1 / Windows 2003 / SQL Server 2000 as Host and App Server Y / Windows 2000 as Web Service Client

Application Server Y running on Windows 2000 as the Web Service client against an 8 CPU Web Services host running .NET 1.1 on Windows Server 2003 with Oracle 9i database achieved a throughput rate of 405.02 transactions per second while supporting a user load of 2500 users. .NET 1.1 on Windows 2003 application server acting as a Web Service client to the same 8 CPU Web Services Host (.NET client to .NET Host) achieved a throughput rate of 793.18 transactions per second.

The peak CPU utilizations on the 8 CPU server hosting the Web Services and the 4 CPU Web Service client during this test were 42% and 66% respectively.
The following table summarizes throughput, User load and CPU utilization results derived from the transaction and user load performance tests.

<table>
<thead>
<tr>
<th>Application Server Platform</th>
<th>Direct to 8 CPU Host</th>
<th>Direct to 4 CPU Host</th>
<th>Direct to 4 CPU Host No EJB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Peak Users</td>
<td>TPS</td>
<td>Avg Resp Time</td>
</tr>
<tr>
<td>App Server X / Windows 2000 / Oracle9i Database</td>
<td>1750</td>
<td>296.25</td>
<td>2.73</td>
</tr>
<tr>
<td>App Server Y / Windows 2000 / Oracle9i Database</td>
<td>2500</td>
<td>411.87</td>
<td>3.20</td>
</tr>
<tr>
<td>JBoss / Redhat Linux 8.0 / Oracle9i Database</td>
<td>3750</td>
<td>513.33</td>
<td>5.22</td>
</tr>
<tr>
<td>.NET 1.1 / Windows 2000 / Oracle9i Database</td>
<td>4570</td>
<td>840.52</td>
<td>1.27</td>
</tr>
<tr>
<td>.NET 1.1 / Windows 2003 / Oracle9i Database</td>
<td>5500</td>
<td>1076.95</td>
<td>0.34</td>
</tr>
<tr>
<td>.NET 1.1 / Windows 2003 / SQL Server 2000</td>
<td>8500</td>
<td>1486.53</td>
<td>2.08</td>
</tr>
</tbody>
</table>

**Table 4 – Throughput, User Load, and CPU Usage Results: Direct Web Services Tests**

*Note: Peak Users shown above is the user load supported at the time peak TPS was achieved. Average Response Time shown above is the average response time at the time peak TPS was achieved. CPU utilizations shown above are the maximum sustained CPU usage recorded during the test.*
The following table summarizes throughput and CPU utilization results derived from the transaction and user load performance tests for the 4 CPU and 2 CPU Web Services client tests.

<table>
<thead>
<tr>
<th>Application Server Platform**</th>
<th>Via 4 CPU Client</th>
<th>Via 2 CPU Client</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Peak Users</td>
<td>TPS</td>
</tr>
<tr>
<td>App Server X / Windows 2000 / Oracle9i Database</td>
<td>1250</td>
<td>225.42</td>
</tr>
<tr>
<td>App Server Y / Windows 2000 / Oracle9i Database</td>
<td>2250</td>
<td>382.20</td>
</tr>
<tr>
<td>JBoss / Windows 2000 / Oracle9i Database</td>
<td>1875</td>
<td>342.30</td>
</tr>
<tr>
<td>.NET 1.1 / Windows 2000 / Oracle9i Database</td>
<td>3000</td>
<td>558.25</td>
</tr>
<tr>
<td>.NET 1.1 / Windows 2003 / Oracle9i Database</td>
<td>4342</td>
<td>787.07</td>
</tr>
<tr>
<td>.NET 1.1 / Windows 2003 / SQL Server 2000</td>
<td>4765</td>
<td>793.18</td>
</tr>
</tbody>
</table>

Table 5 – Throughput, User Load, and CPU Usage Results: Web Services Client Tests

** JBoss was tested on Windows 2000 as it performed better than on Red Hat Linux 8.0. The databases shown were the backend databases used by the remote eight-CPU Web Services host.
The following table summarizes throughput and CPU utilization results derived from the interoperability tests.

<table>
<thead>
<tr>
<th>Interoperability Tests</th>
<th>Peak Users</th>
<th>TPS</th>
<th>Avg. Resp. Time</th>
<th>CPU Usage (Server)</th>
<th>CPU Usage (Client)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interoperability Test With JBoss / RH Linux 8.0 / Oracle9i Database (HOST) .NET 1.1 / Windows 2003 (CLIENT)</td>
<td>2250</td>
<td>392.78</td>
<td>2.15</td>
<td>85%</td>
<td>80%</td>
</tr>
<tr>
<td>Interoperability Test With .NET 1.1 / Windows 2003 / SQL Server 2000 (HOST) App Server Y / Windows 2000 (CLIENT)</td>
<td>2500</td>
<td>405.02</td>
<td>3.48</td>
<td>42%</td>
<td>66%</td>
</tr>
</tbody>
</table>

*Table 6 – Throughput, User Load, and CPU Usage Results: Interoperability Tests*

Note that higher CPU utilization at peak throughput means that the application server is able to run efficiently in that it is able to take advantage of all available CPU processing horsepower. Readers should not interpret the data as lower CPU utilization as equivalent to more efficiency since lower CPU utilization means that the application server was bottlenecking in some way (for example, context switching across CPUs) and hence not able to achieve additional throughout at higher user loads even though the CPUs were not at maximum. Given no other bottleneck in a system, it is expected that CPU saturation will occur at peak throughput for a system as the system is able to utilize all available CPU cycles.

Note that extensive use of JVM cloning and extensive thread tuning was used to get the J2EE application servers to the highest level of throughput possible as user loads increased, taking advantage of as much of the available CPU processing power as was possible. Likewise, for .NET on Windows Server 2003, multiple .NET worker processes were configured to get the best possible throughput rates. .NET on Windows Server 2003 in all cases was able to take advantage of all available (close to 100%) of the CPU processing cycles; whereas, J2EE, in most cases, was not able to take full advantage of all CPU processing power, likely due to additional context switching incurring CPU wait times for the JVMs running on SMP machines.
Summary Results: Application Server X on Windows 2000 and Oracle9i

Application Server X was tested on Windows 2000 because it exhibited better results on Windows 2000 than on Linux during pre-testing. The latest release of this J2EE application server was not yet supporting Windows Server 2003, and hence Windows Server 2000 was chosen as its base OS.

Application Server X achieved a maximum throughput rate of 294.25 transactions per second while supporting a user load of 1,750 users on an 8 CPU host for the direct Web Services test. For this test, the maximum CPU utilization on the server was only 84 percent. Application Server X was not able to take complete advantage of the server’s processing power and run it at full capacity despite extensive use of JVM cloning via multiple application server instances, and extensive thread tuning. The response times and CPU utilization were stable for up to 1.15 minutes into the test at 1,250 users and then the response times started to increase as user loads increased beyond peak capacity further into the test.

The peak throughput rate was achieved at 1,750 users. The server was able to support another 500 users and sustain the same throughput rates, but the response times were beyond the predefined acceptable rate of 3 seconds for each transaction (total average response time of 9 seconds for three transactions). The total test ran for about 2.5 hours given a ramp up rate of 250 users every 15 minutes.

For the 4 CPU direct test, Application Server X achieved a peak throughput rate of 213.37 transactions per second while supporting a user load of 1,500 users with a maximum CPU utilization of 100 percent at peak user load. Application server X was able to drive the 4 CPU host to full capacity and run at 100 percent CPU utilization compared to the maximum utilization of only 84 percent on the 8 CPU host, resulting in a scalability factor of only 0.38 going from a 4 CPU host to an 8 CPU host. The inability to achieve maximum CPU utilization on the 8 CPU configuration is likely due to higher CPU context switching incurring blocking CPU wait times for the application server/JVM instances as processors are added. Running multiple application server instances allowed both the 4 and 8 CPU configurations to achieve higher CPU utilization and throughout rates than running a single instance.
The Java Bean implementation of the test application yielded only marginal performance gain at 211.38 transactions per second compared to 213.37 transactions per second of EJB implementation on the 4 CPU host. This could be attributed to the stateless nature of the application. The overhead of using EJBs for this application was negligible.

Application Server X acting as a Web Service client on a 4 CPU server achieved a throughput rate of 225.42 transactions per second while supporting a user load of 1,250 users. The peak CPU utilizations on the 8 CPU server hosting the Web Services and the 4 CPU client during this test were 73 percent and 76 percent respectively. Application Server X performed better acting as Web Service client than as a direct Web Services host, which demonstrates good management of incoming Web Service requests and efficiency in relaying the requests to the remote Web Service host and sending the responses from the Web Service host back to the clients.

The 2 CPU Web Service client test achieved throughput rate of 162.30 transactions per second while supporting a user load of around 1,200 users. The peak CPU utilizations on the 8 CPU server hosting the Web Services and the 2 CPU client during this test were 46 percent and 93 percent respectively. The throughput increased by only 38.89 percent when increasing the number of processors from 2 to 4.
Summary Results: Application Server Y on Windows 2000 and Oracle9i

Application Server Y was tested on Windows 2000 because it exhibited better results on Windows 2000 than on Linux during pre-testing. Application Server Y achieved a maximum throughput rate of 411.86 transactions per second while supporting a user load of 2,500 users on an 8 CPU host for the direct Web Services test. For this test, the maximum CPU utilization on the server was around 99 percent, taking complete advantage of the Server’s processing power and running it at full capacity. While extensive thread tuning was used to get this result, the use of application server cloning was not necessary for Application Server Y in the direct tests, even on the 8 CPU configuration.

The response times and CPU utilization were stable for up to 2 hours into the test at 2,250 users and then the response times started to increase. The peak throughput rate was achieved at 2,500 users, but the server was able to support another 500 users and sustain the same throughput rates. The total test ran for about 3 hours and 15 minutes given a ramp up of 250 users every 15 minutes.

For the 4 CPU direct test, Application Server Y achieved a peak throughput rate of 312.7 transactions per second while supporting a user load of 1,750 users with a maximum CPU utilization of 99 percent at peak user load. The throughput increased by 31.7 percent by increasing the number of processors from 4 to 8, while the number of users supported at peak throughput increased from 1,750 to 2,500.

The EJB implementation on Application Server Y performed better than the Java Bean implementation on a 4 CPU host. The throughput rates were 411.86 and 348.2 transactions per second respectively at a user load of 2,500 users.

For the 4 CPU client test, Application Server Y, acting as a Web Service client on a 4 CPU server, achieved a throughput rate of 382.5 transactions per second while supporting a user load of 2,250 users. The peak CPU utilizations on the 8 CPU server hosting the Web Services and the 4 CPU client during this test were 82 percent and 69 percent respectively. Application Server Y performed significantly better (22.2 percent) acting as a client than as a direct Web Services host. The use of thread tuning and JVM cloning was necessary to get the throughput and CPU utilization to the maximum possible on the Web Service client computer.
The 2 CPU client test achieved throughput rate of 304.8 transactions per second while supporting a user load of 1,750 users. The peak CPU utilizations on the 8 CPU server hosting the Web Services and the 2 CPU client during this test were 45 percent and 97 percent respectively. The throughput increased by only 25.4 percent by increasing the number of processors from 2 to 4. Thread tuning but no JVM cloning was necessary to achieve maximum possible throughput/CPU utilization at the higher user loads.

**Summary Results: JBoss on Red Hat Linux 8.0 and Oracle9i**

JBoss RC1 3.2 was tested on Linux because it exhibited better performance results on Linux than on Windows 2000 for the direct Web Services tests during pre testing. However, for the Web services client tests, JBoss was tested on Windows 2000 as it performed better than on Red hat Linux 8.0. JBoss RC1 3.2 application server running on Red Hat Linux 8.0 achieved better performance than the other two J2EE application servers tested in this benchmark. It achieved a maximum throughput rate of 513.30 transactions per second while supporting a user load of 3,750 users on an 8 CPU host for the direct Web Services test. However, the maximum CPU utilization on the server was only 76 percent despite the use of thread tuning and JVM cloning (running multiple JBoss instances).

Although the application server could not run at full CPU utilization on the 8 CPU configuration, it was able to support more users and achieve higher throughput rates than the leading commercial J2EE application servers. CPU utilization, and throughput rates increased proportionately as the user load increased. The peak throughput rate was achieved at 3,750 users, with the total average response time for all three transactions under 5.5 seconds at all times. The total test ran for about 3 hours and 45 minutes given a ramp up of 250 users every 15 minutes.

JBoss achieved a peak throughput rate of 354.25 transactions per second while supporting a user load of 2,250 users with a maximum CPU utilization of 92 percent at peak user load for the 4 CPU direct test. JBoss running on Linux demonstrated the best scalability among all the platforms tested with an increase of 44.9 percent in throughput rate by increasing the number of processors from 4 to 8.

The Java Bean implementation of the test application yielded a performance gain at 412.42 transactions per second compared to 354.35 transactions per second for the EJB implementation on the 4 CPU host, an increase of 16.4 percent.
The JBoss application server (running on Windows 2000), acting as a client for Web Services on a 4 CPU server, achieved a throughput rate of 342.30 transactions per second while supporting a user load of 1,875 users. The peak CPU utilizations on the 8 CPU server hosting the Web Services and the 4 CPU client during this test were 36 percent and 89 percent respectively.

The 2 CPU Web Services client test achieved a throughput rate of 200.08 transactions per second while supporting a user load of around 1,250 users. The peak CPU utilizations on the 8 CPU server hosting the Web Services and the 2 CPU client during this test were 20 percent and 94 percent respectively. The throughput increased by 71 percent by increasing the number of processors from 2 to 4, which was the best scalability demonstrated by any application server platform tested. This was followed by .NET 1.1 running on Windows 2003 with an SQL Server backend at 61.5 percent.
Summary Results: .NET 1.1 on Windows 2000 and Oracle9i

The .NET 1.1 framework running on an 8 CPU Windows 2000 Server achieved a maximum throughput rate of 840.51 transactions per second while supporting a user load of 4,570 users for the direct Web Services test. For this test, the maximum CPU utilization on the server was only 83 percent. The response times and CPU utilization were stable for up to 4:15 minutes into the test at 4,320 users and then the response times increased once the user load increased beyond maximum capacity later in the run. CPU utilization, and throughput rates increased proportionately as the user load increased. The peak throughput rate was achieved at 3,750 users, but the server was able to support another 500 users and sustain the same throughput rates. The total average response time for all three transactions was under 5 seconds at all times. The total test ran for about 6 hours given a ramp up of 250 users every 15 minutes.

.NET 1.1 running on an 4 CPU Server achieved a peak throughput rate of 689.88 transactions per second while supporting a user load of 3,574 users with a maximum CPU utilization of 95 percent at peak user load for the 4 CPU direct test. The increase in throughput rate of the server from a 4 CPU host to an 8 CPU host was 21.83 percent.

For the 4 CPU Web Service client test, .NET 1.1 on Windows 2000 achieved a throughput rate of 558.25 transactions per second while supporting a user load of 3,000 users. The peak CPU utilizations on the 8 CPU server hosting the Web Services and the 4 CPU client during this test were 52 percent and 84 percent respectively.

The 2 CPU client test achieved throughput rate of 354.83 transactions per second while supporting a user load of around 2,000 users. The peak CPU utilizations on the 8 CPU server hosting the Web Services and the 2 CPU client during this test were 28 percent and 81 percent respectively. The throughput increased by 57.3 percent by increasing the number of processors from 2 to 4.

Note that thread tuning was used to achieve better results, but multiple .NET worker processes are not supported on Windows 2000 (as opposed to Windows Server 2003 which does support this feature).
Summary Results: .NET 1.1 on Windows 2003 and Oracle9i

The .NET 1.1 framework running on an 8 CPU Windows 2003 Server achieved an impressive throughput rate of 1076.95 transactions per second while supporting a user load of 5,500 users for the direct Web Services test. For this test, the maximum CPU utilization on the server was 100%. The response times (less than 100 ms) and CPU utilization were impressively low and stable for up to 5 hours into the test at 5,250 users and then the response times increased after the user load was increased beyond maximum capacity later in the test run. Response times, CPU utilization, and throughput rates increased proportionately as the user load increased. The total average response time for all three transactions was under 2.5 seconds at all times. The total test ran for about 6 hours with a ramp of 250 users added every 15 minutes.

.NET 1.1 running on an 4 CPU Server achieved a peak throughput rate of 785.95 transactions per second while supporting a user load of 4,500 users with a maximum CPU utilization of 100 percent at peak user load for the 4 CPU direct test. The increase in throughput rate of the server from a 4 CPU host to an 8 CPU host was 37.03 percent.

For the 4 CPU client test, .NET 1.1 on Windows 2003 achieved a throughput rate of 787.07 transactions per second while supporting a user load of 4,342 users. The peak CPU utilizations on the 8 CPU server hosting the Web Services and the 4 CPU client during this test were 75 percent and 56 percent respectively.

The 2 CPU Web Services client test achieved throughput rate of 482.07 transactions per second while supporting a user load of around 2,750 users. The peak CPU utilizations on the 8 CPU server hosting the Web Services and the 2 CPU client during this test were 42 percent and 99 percent respectively. The throughput increased by 63.27 percent by increasing the number of processors from 2 to 4.

Thread tuning and the use of multiple worker processes was used to get the best possible results.
Summary Results: .NET 1.1 on Windows 2003 and SQL Server 2000

The .NET 1.1 framework running on an 8 CPU Windows 2003 Server with a SQL Server 2000 backend database achieved the best throughput rate among all platforms tested at 1486.53 transactions per second while supporting a user load of 8,500 users for the direct Web Services test. For this test, the maximum CPU utilization on the server was 100 percent. The response times and CPU utilization were stable for up to 7 hours and 15 minutes into the test at 7,250 users and then the response times increased. CPU utilization and throughput rates increased proportionately as the user load increased. The total test ran for about 8.5 hours with a ramp of 250 users every 15 minutes.

.NET 1.1 running on an 4 CPU Server achieved a peak throughput rate of 1072.90 transactions per second while supporting a user load of 5,750 users with a maximum CPU utilization of 100 percent at peak user load for the 4 CPU direct test. The increase in throughput rate of the server from a 4 CPU host to an 8 CPU host was 38.6 percent.

For the 4 CPU Web Service client test, .NET 1.1 on Windows 2003 achieved a throughput rate of 793.18 transactions per second while supporting a user load of 4,765 users. The peak CPU utilizations on the 8 CPU server hosting the Web Services and the 4 CPU client during this test were 54 percent and 99 percent respectively. There was a 26 percent drop in performance compared to the 4 CPU direct Web Services host.

The 2 CPU client test achieved throughput rate of 491.12 transactions per second while supporting a user load of around 2,666 users. The peak CPU utilizations on the 8 CPU server hosting the Web Services and the 2 CPU client during this test were 27 percent and 99 percent respectively. The throughput increased by 61.5 percent by increasing the number of processors from 2 to 4.

Thread tuning and multiple worker processes were used to get the best possible results.
Conclusion

Through the process of conducting this performance benchmark, we are more convinced than ever of the value of Web Services and that companies should recognize that Web Services will play an important role in the near future for e-business and application integration and interoperability. In most enterprises both application server platforms co-exist, and there is a business need to access existing applications developed in J2EE from .NET and vice versa and Web Services are already playing a key role in solving these problems.

The following are the significant highlights and conclusions from the performance evaluation:

- Microsoft .NET provided the best performance among the products evaluated in all tests conducted.
  - It is believed that the performance results for .NET are due to the fact that Web Services were part of the core .NET framework from the start, and not added features on top of an existing framework (as is the case with J2EE).

- JBoss performed better than the two leading commercially available, J2EE-certified application servers.

- There was wide variation in performance between all three J2EE-based application server products.

- The results from the interoperability tests prove that with Web Services, interoperability between .NET and J2EE platforms is a reality, and performance is not a blocking issue in such cases.

- Against a backend Oracle 9i database, .NET on Windows Server 2003 (8 CPU) achieved 110 percent better throughput than the leading J2EE product against the same Oracle 9i database.
  - This shows that .NET is a viable platform choice for organizations that have standardized on Oracle as their enterprise RDBMS.

- The Java Bean implementation of the test application yielded only marginal performance gain compared to the EJB implementation.
  - The performance of the EJB implementation (using a stateless session bean with local interfaces) is possibly better in this application because of the relatively low number of objects and the stateless nature of the Web Services.
• While the application servers demonstrated better performance when scaling from four processors to eight processors (both for Windows and Linux), the additional performance gained when moving from four processors to eight processors was not as significant as expected.
  o This is likely due to the fact that Web Services are network-intensive and such applications likely benefit more from clustering multiple machines (as opposed to adding processors to a machine).
  o Organizations should understand that a combination of scaling up (adding more processors to a single system) and scaling out (adding more computers to a cluster) should be used to provide the necessary capacity to host a Web Service, while reducing the overall number of servers required in the cluster (in order to reduce hardware costs and ongoing management costs).
  o Due to the stateless nature of Web Services, application servers operating in a clustered mode can be expected to deliver near linear scalability. So when it comes to hosting Web Services, it may be beneficial for organizations to consider scaling out two-and four-processor systems (adding more servers) rather than scaling up (adding more than four processors).

• While JBoss delivered the best J2EE-based Web Services host results, Application Server Y delivered the best J2EE-based Web Services client results. Overall, .NET provided significantly better performance results versus the J2EE products acting as both a Web Service client and a Web Service host.

• This evaluation showed that organizations could expect ultra-high-performance (millions of transactions per day) related to serving Web Services using an Intel-based 4-CPU server from Hewlett-Packard. Combined with the near linear scalability when adding new machines in parallel, organizations should be able to handle most of their Web Services needs with ease.

• Though reliability and failover were not tested in this evaluation, all of the evaluated applications performed well and had little or no problems during testing. The overall reliability features of the major application server products in the market have improved dramatically.

It should be noted that Web Services currently only offers loosely coupled integration between systems due to the simple request/response-type functionality. In addition, security is still a concern. However, with the evolution of this technology, and support for multiple transactions while maintaining state and the ability to orchestrate Web Services to build new applications/processes improves, organizations will soon start considering Web Services for mission-critical applications that involve integrations between business partners and suppliers across the Internet.
Appendices

Appendix A: Database Schema

The following tables illustrate the database schema. Columns in bold indicate primary keys.

**Table Orders**

<table>
<thead>
<tr>
<th>Column</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORDERID</td>
<td>NUMBER</td>
</tr>
<tr>
<td>CUSTOMERID</td>
<td>NUMBER</td>
</tr>
<tr>
<td>ORDERSTATUS</td>
<td>NUMBER</td>
</tr>
<tr>
<td>ORDERDATE</td>
<td>DATE</td>
</tr>
<tr>
<td>TOTALAMOUNT</td>
<td>LONG</td>
</tr>
</tbody>
</table>

**Table Line Items**

<table>
<thead>
<tr>
<th>Column</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORDERID</td>
<td>NUMBER</td>
</tr>
<tr>
<td>ITEMID</td>
<td>NUMBER</td>
</tr>
<tr>
<td>PRODUCTID</td>
<td>NUMBER</td>
</tr>
<tr>
<td>QUANTITY</td>
<td>NUMBER</td>
</tr>
<tr>
<td>AMOUNT</td>
<td>LONG</td>
</tr>
</tbody>
</table>

**Table Products**

<table>
<thead>
<tr>
<th>Column</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRODUCTID</td>
<td>NUMBER</td>
</tr>
<tr>
<td>PRODUCTDESCRIPTION</td>
<td>VARCHAR2(50)</td>
</tr>
<tr>
<td>PRICE</td>
<td>LONG</td>
</tr>
<tr>
<td>INVENTORY</td>
<td>NUMBER</td>
</tr>
</tbody>
</table>
## Table Customers

<table>
<thead>
<tr>
<th>Column</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUSTOMERID</td>
<td>NUMBER</td>
</tr>
<tr>
<td>CONTACTFIRSTNAME</td>
<td>VARCHAR2(50)</td>
</tr>
<tr>
<td>CONTACTLASTNAME</td>
<td>VARCHAR2(50)</td>
</tr>
<tr>
<td>CONTACTPHONE</td>
<td>VARCHAR2(50)</td>
</tr>
<tr>
<td>LASTACTIVITYDATE</td>
<td>DATE</td>
</tr>
<tr>
<td>BILLINGFIRSTNAME</td>
<td>VARCHAR2(50)</td>
</tr>
<tr>
<td>BILLINGLASTNAME</td>
<td>VARCHAR2(50)</td>
</tr>
<tr>
<td>BILLINGADDRESS1</td>
<td>VARCHAR2(50)</td>
</tr>
<tr>
<td>BILLINGADDRESS2</td>
<td>VARCHAR2(50)</td>
</tr>
<tr>
<td>BILLINGCITY</td>
<td>VARCHAR2(50)</td>
</tr>
<tr>
<td>BILLINGSTATE</td>
<td>VARCHAR2(50)</td>
</tr>
<tr>
<td>BILLINGZIP</td>
<td>VARCHAR2(50)</td>
</tr>
<tr>
<td>BILLINGCREDITCARDNUMBER</td>
<td>VARCHAR2(50)</td>
</tr>
<tr>
<td>BILLINGCREDITCARDEXPIRE</td>
<td>VARCHAR2(50)</td>
</tr>
<tr>
<td>SHIPPINGFIRSTNAME</td>
<td>VARCHAR2(50)</td>
</tr>
<tr>
<td>SHIPPINGLASTNAME</td>
<td>VARCHAR2(50)</td>
</tr>
<tr>
<td>SHIPPINGADDRESS1</td>
<td>VARCHAR2(50)</td>
</tr>
<tr>
<td>SHIPPINGADDRESS2</td>
<td>VARCHAR2(50)</td>
</tr>
<tr>
<td>SHIPPINGCITY</td>
<td>VARCHAR2(50)</td>
</tr>
<tr>
<td>SHIPPINGSTATE</td>
<td>VARCHAR2(50)</td>
</tr>
<tr>
<td>SHIPPINGZIP</td>
<td>VARCHAR2(50)</td>
</tr>
</tbody>
</table>
Appendix B: Sample SOAP Responses

The following are the sample XML SOAP returns for GetOrderDetails, GetCustomer, and insertCustomer Web Services calls.

GetOrderDetails XML SOAP Return:

```xml
<?xml version="1.0" encoding="utf-8" ?>
<Order xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns="http://tempuri.org/">
  <orderId>1</orderId>
  <orderStatus>1</orderStatus>
  <orderDate>2002-07-06T00:00:00.0000000-07:00</orderDate>
  <orderTotalAmount>78.7</orderTotalAmount>
  <customer>
    <customerId>1</customerId>
    <contactFirstName>John</contactFirstName>
    <contactLastName>Smith</contactLastName>
    <contactPhone>332-3233</contactPhone>
    <lastActivityDate>2002-06-03T00:00:00.0000000-07:00</lastActivityDate>
    <creditCardNumber>823323</creditCardNumber>
    <creditCardExpirationDate>4/2002</creditCardExpirationDate>
    <billingAddress>
      <firstName>John</firstName>
      <lastName>Smith</lastName>
      <address1>Billing Address1</address1>
      <address2>Billing Address 2</address2>
      <city>Chicago</city>
      <state>IL</state>
      <zip>60607</zip>
    </billingAddress>
    <shippingAddress>
      <firstName>John</firstName>
      <lastName>Smith</lastName>
      <address1>Shipping Addr1</address1>
      <address2>Shipping Addr2</address2>
      <city>Shipping City</city>
      <state>Shipping State</state>
      <zip>22232</zip>
    </shippingAddress>
  </customer>
  <lineItems>
    <item>
      <orderId>1</orderId>
      <itemId>1</itemId>
      <productId>2</productId>
    </item>
  </lineItems>
</Order>
```
<itemDescription>Product2</itemDescription>
<orderQuantity>4</orderQuantity>
<unitPrice>56.45</unitPrice>
</item>

- <item>
  <orderId>1</orderId>
  <itemId>2</itemId>
  <productId>4</productId>
  <itemDescription>Product4</itemDescription>
  <orderQuantity>67</orderQuantity>
  <unitPrice>2.23</unitPrice>
</item>
</lineItems>
</Order>

GetCustomer XML SOAP Return:

<?xml version="1.0" encoding="utf-8" ?>
<Customer xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://tempuri.org/"
  customerId="4"
  contactFirstName="Contact First Name"
  contactLastName="Contact Last Name"
  contactPhone="Contact Phone"
  lastActivityDate="2002-09-26T01:28:03.8307056-07:00"
  creditCardNumber="Credit Card Number"
  creditCardExpirationDate="12/05"
  billingAddress="BILLINGFIRSTNAME BILLINGLASTNAME BILLINGADDRESS1 BILLINGADDRESS2 BILLINGCITY BILLINGSTATE BILLINGZIP"
  shippingAddress="SHIPPINGFIRSTNAME SHIPPINGLASTNAME SHIPPINGADDRESS1 SHIPPINGADDRESS2 SHIPPINGCITY SHIPPINGSTATE SHIPPINGZIP"/>

Doculabs 53
InsertCustomer XML SOAP Return:

<?xml version="1.0" encoding="utf-8" ?>
<int xmlns="http://tempuri.org/">8</int>
Appendix C: Web Client HTML Files

Calls made to the Web Services by way of a web client required a standard syntax. The following is syntax for calling the Web Services through HTTP requests:

**For getOrderDetails:**

*Java:*


*Microsoft:*


**For getCustomer:**

*Java:*


*Microsoft:*


**For addCustomer**

For addCustomer, a simple HTTP Get is not sufficient; an HTTP POST is used through an HTML form. The HTML form contained input text boxes for all the required fields.

*Java:*


*Microsoft:*


Note: values in [] should be replaced with a dynamic value generated by the test script.
Appendix D: Code Reviews

The following is the code review performed by BrowserMedia.

---

**How we reviewed the code**

In general, guesses about perceived performance are just that, guesses, and really need to be backed up by actual hard data. Ultimately, performance optimization can best be done on simple, clean code. So in reviewing the source code we looked for the following items:

- Does the source code meet the rules laid out by the tests?
- How simple is the code?
- How much duplication is there?
- Does the projects contain the minimum possible number of classes/methods?
- Are there any obvious performance bottlenecks?

**Assumptions**

Based on our understanding of the test requirements, we had several assumptions about what the implementations needed to do:

- The web services code does not need to remotely access the business logic. They both could exist on the same computer. (All 4 implementation seem to assume this).
- The intent of the test is to compare web services implementations; therefore the implementations should be as identical as possible to eliminate unwanted variables in testing.

**Common Grounds**

All four implementations of the project are very similar. They all implement a clean data model with full-fledged objects for Customers, Orders, Line Items and Addresses. The object model is a little more finely grained than the table model. The Customer table gets split into a Customer object with 2 Addresses (billing/shipping), so 1 table maps to 3
objects in memory. This makes the object code easier to work with but increases the complexity of the object-relational mapping.

All three Java implementations use Ant (http://jakarta.apache.org/ant/index.html) an open source build tool to build and deploy their applications.

Very Important Objects

The core objects in each implementation are listed here, along with their roles.

- **wsFacade** – A wrapper around the business logic, this is what the web services on each platform will interface with. This class is basically identical in all four implementations, so the only variable is how complicated deploying it as web service will be on each app server.

- **Model Objects** – (i.e. Customer, Address) – Hold the persistent fields from the database and roughly correspond to tables.

- **Data Access Layer (DAL) objects** – One each for Customer and Order, these hold the sql, execute statements against the database, and map the database columns to object fields. Java uses JDBC and .NET uses Oracle commands.

- **Business Layer** – In java these take the flavor of Stateless Session Beans (CustomerSessionBean/OrderSessionBean) and just delegate to the DAL’s. In .NET these are ordinary objects (CustomerBL/OrderBL) that also delegate to the DAL’s. This is a fundamental different between the Java and .NET that we will discuss later.

Differences

The main difference between the implementations ultimately comes down to a few issues.

- How complicated is it to use the App Server’s web services layer?
- How do the Java implementations generate their Enterprise Java Bean (EJB) code (the deployment descriptors, EJB interfaces, etc)?
- Is EJB even necessary?
- How coupled is the code to the database?
- How coupled is the code to the App Server?

With these differences in mind let us take a look at each implementation.

**JBoss**

The distinguishing feature of the JBoss implementation is that it uses XDocelet, a free open source code generation tool (http://xdocelet.sourceforge.net) to actively generate the EJB interface and configuration files. These files get generated every time the project gets built/deployed.
So what?
The use of XDoclet means no tedious EJB hand coding, which should reduce the
development time and make maintenance much simpler during the application lifetime.
As a bonus, it makes understanding the code easier (therefore easier to modify) as there
are fewer files to look through. It also makes the JBoss implementation the most portable,
as it’s not coupled to the app server for code generation.

Application Server X / Application Server Y
The other two Java implementations appear to use their respective app server tools to edit
the EJB files. Without a detailed understanding of the EJB tools provided by
them, it’s hard to guess which files need to be hand edited and which are
wizard generated. Other than web service configuration and EJB generation, these two
are identical to the JBoss implementation.

.NET
The two main issues between the Java and .NET code are that the .NET code has:
• An oracle specific data access layer.
• No remote calls from the web service facade.

Oracle specific
While the .NET data model looks identical to the J2EE model, the Data Access Layer
(DAL) has some substantial differences. The .NET DAL code manually creates Oracle
parameters and runs OracleCommands against the database. The .NET code to insert a
customer is about twice as long as the corresponding JDBC code and looks more brittle.
I.e. changes would ripple to more places. For example, the .NET CustomerDAL
specifies the order of the columns and the data type of each column. The JDBC code
only specifies the column order. In addition, the entire .NET CustomerDAL would need
to be rewritten to change databases, whereas the Java code could just change a line in a
configuration file.

It’s tough to judge performance from just looking at the code. The .NET code appears
to be optimizing for an Oracle database. However, this means duplication of code and a
much higher degree of coupling to the database than the J2EE implementations.

No Remote Calls
Finally, the .NET code wSFacade appears to make simple local procedure calls to a
BusinessLayer object which delegates to the Data Access Layer objects. This local call
does not incur the overhead of Enterprise Java Beans, which all of the Java
implementations use.
What does EJB buy?

*My first law of Distributed Object Design: Don’t distribute your objects!*  
- Martin Fowler, Patterns of Enterprise Application Architecture

The decision to use a heavy weight (and complicating technology) like EJB should be firmly based on the business requirements for the application. The data model is very simple, and none of the functionality really requires the key benefits of EJB (truly ACID transactions, remote object access, etc).

In this case, the only thing it’s going to buy the Java implementations (as compared to the .NET implementation) is worse performance. This is particularly important to the getCustomer() method, whose intent is to test just the Web Services layer without the database access performance variable.

In the J2EE implementations, a call to wsFacade.getCustomer() results in the following operations:

1. a JNDI lookup is done for CustomerSessionBeanLocalHomeInterface
2. a call to CustomerSessionBeanLocalHomeInterface.create() which gets a CustomerSessionBean
3. a call via the CustomerSessionBean to CustomerDAL.getCustomer() – which returns a hard coded Customer object.

These all use the local EJB interfaces, which will certainly perform better than remote calls. But it will not be nearly as fast as making a direct call to the CustomerDAL.getCustomer() method from the wsFacade.

The multiple interfaces and configuration files that EJB needs increases the line count of the Java implementations as compared to the .NET. The JBoss implementation reduces this by using XDoclet, but it still needs to configure the Xdoclet in the ant build file.

**Conclusions**

Overall, all the implementations are extremely similar, reasonably well factored, obey the rules of the test and would be relatively easy to optimize based on further performance profiling. Both the .NET and Java implementations use hand written database access code that could probably be simplified (reducing the complexity and line count). Without hard tests numbers, its not really productive to guess how either would perform. Since all database code is encapsulated, all the implementations would be relatively easy to refactor. The database drivers can play a role in the performance as well. .NET uses drivers optimized for Oracle, where as the Java performance will depend on what driver is deployed along with the J2EE application server.

Finally, the other big wild card is the use of the EJB in the Java implementations. It’s our opinion that given the simple business logic, EJB is a poor choice that will unnecessarily handicap the Java implementations relative to the .NET implementation. Plain old java objects would work better in this case. As long as these issues don’t skew
the data, the performance of each app server will come down to their respective Web services implementations.
The following is the summary of review of code performed by UrbanCode.

**JBoss Implementation**

src/main/client/com/doculabs/webServices/wsFacade.java: `getCustomerSes()` and `getOrderSes()` make JNDI calls to lookup the Home interface and create a new session bean instance on every invocation. This is expensive and unnecessary. JNDI calls can be very time consuming.

A more efficient implementation would make the JNDI lookup once and cache the created session bean reference.

**Application Server X Implementation**

src/com/doculabs/webServices/wsFacade.java: same problem as with JBoss above. `getCustomerSes()` and `getOrderSes()` make JNDI calls to lookup the Home interface and create a new session bean instance on every invocation.

**DotNet Implementation**

DotNet implementation of database access logic is tied Oracle via a hard coded reference to `System.Data.OrcleClient` in DotNetDoculabsWebService/BusinessLogic/DatabaseOraCL.cs. This is in contrast to using a vendor neutral implementation based JDBC in the J2EE code. Using a vendor specific library/protocol may provide a performance advantage and definitely makes the resulting code less portable. Thus, as written, the DotNet code would require significant rewrites in order to be ported to another database, while the J2EE code would not require such rewrites.

A more apples to apples comparison of DotNet versus J2EE implementations would require the DotNet code to use a vendor agnostic database access strategy such as one based on the `System.Data.SqlClient` namespace.

**Interoperability Test**

We were unable to find any J2EE code for the interoperability test mentioned on page 22 of the Test Specification. We found DotNet interoperability code that calls each of the J2EE implementations but no code for each of the J2EE implementations that call the DotNet web service.

Overall, the J2EE web services code was pretty messy. The Doculabs Business Logic code was not that messy, but the web services implementations for Application Server X and JBoss were. Files were duplicated in several directory locations, old versions of files that were no longer being used were present in the source tree, etc.
Appendix E: Changes based on Code Reviews

The following are the issues identified by the two companies that performed the code reviews. This appendix summarizes the changes made to the code and responses to the comments in the code reviews.

1. **[Comment]** wsFacade.java makes JNDI calls and this should be cached.

   **[Reply]** The application now uses static member variables in the wsFacade file to cache the EJB home object.

2. **[Comment]** .NET implementation uses OracleClient reference.

   **[Reply]** This is the way .NET is designed and hence why a dedicated data access classes is good design pattern. The data access classes allow the data access logic to change without the need to modify the business logic code. It should also be noted that the SQL statement used in the java versions also need to be modified if the underlying database changes (for example, to use SQL Server identity seeds vs. Oracle Sequences). In our implementation we specifically optimized for performance, whilst trying to allow for easy code maintenance by separating the code into multiple tiers.

3. **[Comment]** Should have used the SqlClient namespace.

   **[Reply]** This only works with SQL Server. .NET uses SQLClient classes for SQL Server Access, and OracleClient classes for Oracle access to support native database features. Abstraction is provided at two levels. First, ADO.NET provides the same programming model against both sets of clients, so ultimately results sets are going to either be forward-only ADO.NET Data Readers or updatable ADO.NET Data Sets (using XML to represent data) in both cases. Also, the .NET implementation is designed to abstract data access into a Data Access Layer such that UI and middle-tier business logic does not change when changing backend databases. This is the design of .NET.

4. **[Comment]** Missing java clients for interop testing.

   **[Reply]** We were waiting to see which java client performed the fastest as the benchmark only required that we test .NET interoperability with the fastest J2EE client.
5. [Comment] The Doculabs businessLogic code was not that messy, but the Web Services implementations for Application Server X and JBoss were messy.

[Reply] This has been cleaned up. The zip file was checked for old versions of files. The JBoss Web Service façade file was in several places plus an implementation as a jws page. The reason for this was that there was little guidance on where this type of file should go. For the record, the master and final version is in ‘src\main\client\com\doculabs\webservices’, implemented as a java bean. This is not true of the Application Server X implementation and there is only one version of the file. The code for J2EE is clean at this point.

Also, it might have helped with the JBoss implementation if there had been an IDE to help manage the development process rather than just using a simple text editor.

6. [Comment] Data access layers were Oracle specific.

[Reply] Both implementations are Oracle specific in terms of the SQL statements being executed. An Oracle specific driver is used in the .NET implementation as this is the driver that is recommended by Microsoft. The database specific code has been abstracted into its own tier, so it is not necessary to use a generic database implementation to maintain one code base for business logic and UI logic. The question that perhaps should be posed, is, are there any additional Oracle optimizations that can be made to the Java code?

7. [Comment] No remote calls (in the direct case), so why use an EJB at all?

[Reply] We are trying to model what a customer might use in the real world and not what is the smallest foot print required to run this benchmark. In terms of the .NET implementation, if we wanted to make the object remote able, i.e. callable from another machine, then we would only need to create an xml file describing how the object should be accessed ‘remotely’ and we would not need to change the object code itself. In the java world the object needs to include RMI code or be created as an EJB and hence inherit RMI functionality. This is down to a difference in the way remoting functionality is implemented.

We now have simple plain old java objects instead of EJBs as one test case to show the performance tradeoff in the final results between using local EJBs vs. using no EJBs at all.
Appendix F: Detailed Results

This appendix contains graphs for all the tests conducted during this benchmark. The graphs display the number of completed transactions, the average time taken to perform the transactions and the number of users during the load test sampled at 60-second intervals. Each getCustomer, getOrderDetails, and insertCustomer call is considered a transaction. Transactions per second shown in the following graphs are the sum of getCustomer, getOrderDetails, and insertCustomer transactions. Similarly Average Response Time is the sum of response times for each of the three transactions.

Note: For the Web Services Client tests, the databases shown in the graphs were the backend databases used by the remote eight-CPU Web Services hosts.

Figure 12 – Detailed Results for Application Server X on Windows 2000 and Oracle9i Database – 8 CPU Direct Test
Figure 13 – Detailed Results for Application Server X on Windows 2000 and Oracle9i Database – 4 CPU Direct Test

Figure 14 – Detailed Results for Application Server X on Windows 2000 and Oracle9i Database – 4 CPU Direct Test – No EJB
Figure 15 – Detailed Results for Application Server X on Windows 2000 and Oracle9i Database – 4 CPU Client Test

Figure 16 – Detailed Results for Application Server X on Windows 2000 and Oracle9i Database – 2 CPU Client Test
Figure 17 - Detailed Results for Application Server Y on Windows 2000 and Oracle9i Database – 8 CPU Direct Test

Figure 18 – Detailed Results for Application Server Y on Windows 2000 and Oracle9i Database – 4 CPU Direct Test
Figure 19 – Detailed Results for Application Server Y on Windows 2000 and Oracle9i Database – 4 CPU Direct Test – No EJB

Figure 20 – Detailed Results for Application Server Y on Windows 2000 and Oracle9i Database – 4 CPU Client Test
Figure 21 – Detailed Results for Application Server Y on Windows 2000 and Oracle9i Database – 2 CPU Client Test

Figure 22 – Detailed Results for JBoss on Red Hat Linux 8.0 and Oracle9i Database – 8 CPU Direct Test
Figure 23 – Detailed Results for JBoss on Red Hat Linux 8.0 and Oracle9i Database – 4 CPU Direct Test

Figure 24 – Detailed Results for JBoss on Red Hat Linux 8.0 and Oracle9i Database – 4 CPU Direct Test – No EJB
Figure 25 – Detailed Results for JBoss on Windows 2000 and Oracle9i Database – 4 CPU Client Test

Figure 26 – Detailed Results for JBoss on Windows 2000 and Oracle9i Database – 2 CPU Client Test
Figure 27 – Detailed Results for .NET 1.1 on Windows 2000 and Oracle9i Database – 8 CPU Direct Test

Figure 28 – Detailed Results for .NET 1.1 on Windows 2000 and Oracle9i Database – 4 CPU Direct Test
Figure 29 – Detailed Results for .NET 1.1 on Windows 2000 and Oracle9i Database – 4 CPU Client Test

Figure 30 – Detailed Results for .NET 1.1 on Windows 2000 and Oracle9i Database – 2 CPU Client Test
Figure 31 – Detailed Results for .NET 1.1 on Windows 2003 and Oracle9i Database – 8 CPU Direct Test

Figure 32 – Detailed Results for .NET 1.1 on Windows 2003 and Oracle9i Database – 4 CPU Direct Test
Figure 33 – Detailed Results for .NET 1.1 on Windows 2003 and Oracle9i Database – 4 CPU Client Test

Figure 34 – Detailed Results for .NET 1.1 on Windows 2003 and Oracle9i Database – 2 CPU Client Test
Figure 35 – Detailed Results for .NET 1.1 on Windows 2003 and SQL Server 2000 Database – 8 CPU Direct Test

Figure 36 – Detailed Results for .NET 1.1 on Windows 2003 and SQL Server 2000 Database – 4 CPU Direct Test
Figure 37 – Detailed Results for .NET 1.1 on Windows 2003 and SQL Server 2000 Database – 4 CPU Client Test

Figure 38 – Detailed Results for .NET 1.1 on Windows 2003 and SQL Server 2000 Database – 2 CPU Client Test
Figure 39 – Detailed Results for Interoperability Test With JBoss on Red Hat Linux 8.0 and Oracle9i Database As Host and .NET 1.1 on Windows 2003 As Client

Figure 40 – Detailed Results for Interoperability Test With .NET 1.1 on Windows 2003 and SQL Server 2000 Database As Host and Application Server Y on Windows 2003 As Client
Appendix G: Test Application Code – J2EE Implementation

This section contains the application logic for the J2EE implementation.

Java Middle Tier

Thin Data Classes

Address.java

```java
package com.doculabs.businessLogic;
/**
 * A thin data class which contains information about an address.
 * <BR>Copyright: Copyright (c) 2003
 * <BR>Company: Doculabs
 * @author unascribed
 * @version 1.0
 */
public class Address implements java.io.Serializable {
    private String firstName;
    private String lastName;
    private String address1;
    private String address2;
    private String city;
    private String state;
    private String zip;

    /**
     * Default constructor
     */
    public Address() {
    }

    /**
     * Constructor which specifies a value for each field
     */
    public Address(String firstName, String lastName, String address1, String address2, String city, String state, String zip) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.address1 = address1;
        this.address2 = address2;
        this.city = city;
        this.state = state;
        this.zip = zip;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```
public String getAddress1() {
    return address1;
}

public void setAddress1(String address1) {
    this.address1 = address1;
}

public String getAddress2() {
    return address2;
}

public void setAddress2(String address2) {
    this.address2 = address2;
}

public String getCity() {
    return city;
}

public void setCity(String city) {
    this.city = city;
}

public String getState() {
    return state;
}

public void setState(String state) {
    this.state = state;
}

public String getZip() {
    return zip;
}

public void setZip(String zip) {
    this.zip = zip;
}
}

Customer.java

package com.doculabs.businessLogic;

/**
 * A thin data class which contains information about a customer.
 * <BR>Copyright: Copyright (c) 2003
 * <BR>Company: Doculabs
 * @author unascribed
 * @version 1.0
 */
public class Customer implements java.io.Serializable {

    private int customerId = -1;
    private String contactFirstName = null;
    private String contactLastName = null;
    private String contactPhone = null;
    private java.util.Date lastActivityDate = null;
    private String creditCardNumber = null;
    private String creditCardExpirationDate = null;
    private Address billingAddress = null;
    private Address shippingAddress = null;

    /**
     * Default constructor
     */
    public Customer() {
    }

    /**
     * Constructor which specifies a value for each field
     */
    public Customer(int customerId, String contactFirstName, String contactLastName, String contactPhone,
                    java.util.Date lastActivityDate, String creditCardNumber, String creditCardExpirationDate,
                    Address billingAddress, Address shippingAddress){
this.customerId = customerId;
this.contactFirstName = contactFirstName;
this.contactLastName = contactLastName;
this.contactPhone = contactPhone;
this.lastActivityDate = lastActivityDate;
this.creditCardNumber = creditCardNumber;
this.creditCardExpirationDate = creditCardExpirationDate;
this.billingAddress = billingAddress;
this.shippingAddress = shippingAddress;
}

public int getCustomerId() {
    return customerId;
}

class setCustomerId(int customerId) {
    this.customerId = customerId;
}

class getContactFirstName () {
    return contactFirstName;
}

class setContactFirstName(String contactFirstName) {
    this.contactFirstName = contactFirstName;
}

class getContactLastName () {
    return contactLastName;
}

class setContactLastName(String contactLastName) {
    this.contactLastName = contactLastName;
}

class getContactPhone () {
    return contactPhone;
}

class setContactPhone(String contactPhone) {
    this.contactPhone = contactPhone;
}

class getLastActivityDate () {
    return lastActivityDate;
}

class setLastActivityDate(java.util.Date lastActivityDate) {
    this.lastActivityDate = lastActivityDate;
}

class getCreditCardNumber () {
    return creditCardNumber;
}

class setCreditCardNumber(String creditCardNumber) {
    this.creditCardNumber = creditCardNumber;
}

class getCreditCardExpirationDate () {
    return creditCardExpirationDate;
}

class setCreditCardExpirationDate(String creditCardExpirationDate) {
    this.creditCardExpirationDate = creditCardExpirationDate;
}

class getBillingAddress () {
    return billingAddress;
}
public void setBillingAddress(Address billingAddress) {
    this.billingAddress = billingAddress;
}

public Address getShippingAddress() {
    return shippingAddress;
}

public void setShippingAddress(Address shippingAddress) {
    this.shippingAddress = shippingAddress;
}
}

### LineItem.java

```java
package com.doculabs.businessLogic;

/**
 * A thin data class which contains information about a line item for an order.
 * <BR>Copyright: Copyright (c) 2003
 * <BR>Company: Doculabs
 * @author unascribed
 * @version 1.0
 */

public class LineItem implements java.io.Serializable {

    private int orderId;
    private int itemId;
    private int productId;
    private String productDescription;
    private int orderQuantity;
    private float unitPrice;

    /**
     * Default constructor
     */
    public LineItem() {}

    /**
     * Constructor which specifies a value for each field
     */
    public LineItem(int orderId, int itemId, int productId, String productDescription, int orderQuantity, float unitPrice) {
        this.orderId = orderId;
        this.itemId =itemId;
        this.productId = productId;
        this.productDescription = productDescription;
        this.orderQuantity = orderQuantity;
        this.unitPrice = unitPrice;
    }

    public int getOrderId() {
        return orderId;
    }

    public void setOrderId(int orderId) {
        this.orderId = orderId;
    }

    public int getItemId() {
        return itemId;
    }

    public void setItemId(int itemId) {
        this.itemId = itemId;
    }

    public int getProductId() {
        return productId;
    }

    public void setProductId(int productId) {
        this.productId = productId;
    }

    public int getOrderQuantity() {
        return orderQuantity;
    }

    public void setOrderQuantity(int orderQuantity) {
        this.orderQuantity = orderQuantity;
    }

    public float getUnitPrice() {
        return unitPrice;
    }

    public void setUnitPrice(float unitPrice) {
        this.unitPrice = unitPrice;
    }

    public String getProductDescription() {
        return productDescription;
    }

    public void setProductDescription(String productDescription) {
        this.productDescription = productDescription;
    }
}
```
public void setProductId(int productId) {
    this.productId = productId;
}
public String getProductDescription() {
    return productDescription;
}
public void setProductDescription(String productDescription) {
    this.productDescription = productDescription;
}
public int getOrderQuantity() {
    return orderQuantity;
}
public void setOrderQuantity(int orderQuantity) {
    this.orderQuantity = orderQuantity;
}
public float getUnitPrice() {
    return unitPrice;
}
public void setUnitPrice(float unitPrice) {
    this.unitPrice = unitPrice;
}

Order.java

package com.doculabs.businessLogic;

/**
 * A thin data class which contains information about an order.
 * <BR>Copyright: Copyright (c) 2003
 * <BR>Company: Doculabs
 * @author unascribed
 * @version 1.0
 */
public class Order implements java.io.Serializable {
    private int orderId=-1;
    private int orderStatus=-1;
    private java.util.Date orderDate=null;
    private float orderTotalAmount=-1f;
    private Customer customer=null;
    private LineItem[] lineItems=null;

    /**
     * Default constructor
     */
    public Order() {
    }
    /**
     * Constructor which specifies a value for each field
     */
    public Order(int orderId, int orderStatus, java.util.Date orderDate, float totalAmount, Customer cust, LineItem[] lineItems) {
        this.orderId = orderId;
        this.orderStatus = orderStatus;
        this.orderDate = orderDate;
        this.orderTotalAmount = totalAmount;
        this.customer = cust;
        this.lineItems = lineItems;
    }
    public int getOrderId() {
        return orderId;
    }
    public void setOrderId(int orderId) {
        this.orderId = orderId;
    }
    public int getOrderStatus() {
        return orderStatus;
    }
    public void setOrderStatus(int orderStatus) {
        this.orderStatus = orderStatus;
    }
public void setOrderStatus(int orderStatus) {
    this.orderStatus = orderStatus;
}

public java.util.Date getOrderDate() {
    return orderDate;
}

public void setOrderDate(java.util.Date orderDate) {
    this.orderDate = orderDate;
}

public Customer getCustomer() {
    return customer;
}

public void setCustomer(Customer cust) {
    this.customer = cust;
}

public LineItem[] getLineItems() {
    return lineItems;
}

public void setLineItems(LineItem[] lineItems) {
    this.lineItems = lineItems;
}

public float getOrderTotalAmount() {
    return orderTotalAmount;
}

public void setOrderTotalAmount(float orderTotalAmount) {
    this.orderTotalAmount = orderTotalAmount;
}
}

Data Access Classes

CommonDAL.java

package com.doculabs.businessLogic;

import java.sql.*;
import javax.sql.*;
import javax.naming.*;

/**<br>* A base class containing all the infrastructure to connect to a database.<br>* &lt;BR&gt;Copyright: Copyright (c) 2003<br>* &lt;BR&gt;Company: Doculabs<br>* @author unascribed<br>* @version 1.0<br>* /

public class CommonDAL {
    private static DataSource dataSource = null;

    /**<br> * A method to clean up all database related objects<br> */
    protected static void cleanup(Connection conn, PreparedStatement stmt, ResultSet result) {
        // cleanup resultset
        try {
            if (result != null) {
                result.close();
            }
        } catch(SQLException e) {
            e.printStackTrace();
        }

        // clean up prepared statement
        try {
            if (stmt != null) {

            }
CustomerDAL.java

package com.doculabs.businessLogic;

import java.sql.*;
import java.util.*;

/**
 * An implementation of the CommonDAL specific to customer information.
 * @author unascribed
 * @version 1.0
 */
public class CustomerDAL extends CommonDAL {

    private static final String selectCustIdStmt = "SELECT CUSTOMER_SEQ.nextval FROM DUAL";
    private static final String insertCustStmt = "INSERT INTO CUSTOMERS VALUES(?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?)";

    protected static Connection getConnection() throws SQLException, Exception {
        if (dataSource != null) {
            return dataSource.getConnection();
        } else {
            InitialContext initCtx = null;
            try {
                initCtx = new InitialContext();
                dataSource = (javax.sql.DataSource) initCtx.lookup("jdbc/DoculabsDB");
                // Version below only works if using EJB
                // dataSource = (javax.sql.DataSource) initCtx.lookup("java:comp/env/jdbc/DoculabsDB");
                return dataSource.getConnection();
            } catch (NamingException ne) {
                System.out.println("Failed to lookup JDBC Datasource. Please double check that the name is correct in the config file");
                throw new Exception("Failed to lookup JDBC Datasource " + ne);
            } finally {
                try {
                    if (initCtx != null) initCtx.close();
                } catch (NamingException ne) {
                    System.out.println("Error closing context: " + ne);
                    throw new Exception("Error closing context: " + ne);
                }
            }
        }
    }
}
private static final String selectStmt = "SELECT CONTACTFIRSTNAME,CONTACTLASTNAME,CONTACTPHONE, LASTACTIVITYDATE, BILLINGFIRSTNAME," + "BILLINGLASTNAME, BILLINGADDRESS1, BILLINGADDRESS2, BILLINGCITY, BILLINGSTATE, BILLINGZIP," + "BILLINGCREDITCARDNUMBER, BILLINGCREDITCARDEXPIRE, SHIPPINGFIRSTNAME, SHIPPINGLASTNAME, SHIPPINGADDRESS1," + "SHIPPINGADDRESS2, SHIPPINGCITY, SHIPPINGSTATE, SHIPPINGZIP FROM CUSTOMERS WHERE CUSTOMERID=?";

/**
 * A method to insert customer data into a database
 * @param customer A thin data class containing all the customer information to insert
 */
public static int insertCustomer(Customer customer) {
    PreparedStatement stmt = null;
    Connection dbConnection = null;
    ResultSet result = null;
    int customerId = -1;
    try {
        Address billing = customer.getBillingAddress();
        Address shipping = customer.getShippingAddress();
        dbConnection = getConnection();
        stmt = dbConnection.prepareStatement(selectStmt);
        try {
            result = stmt.executeQuery();
            if (result.next()) {
                customerId = result.getInt(1);
            } else {
                throw new Exception("Could not determine next customer ID from Database");
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            cleanup(null, stmt, result);
        }
        stmt = dbConnection.prepareStatement(insertStmt);
        stmt.setInt(1, customerId);
        stmt.setString(2, customer.getContactFirstName());
        stmt.setString(3, customer.getContactLastName());
        stmt.setString(4, customer.getContactPhone());
        stmt.setDate(5, new java.sql.Date(System.currentTimeMillis()));
        stmt.setString(6, billing.getFirstName());
        stmt.setString(7, billing.getLastName());
        stmt.setString(8, billing.getAddress1());
        stmt.setString(9, billing.getAddress2());
        stmt.setString(10, billing.getCity());
        stmt.setString(11, billing.getState());
        stmt.setString(12, billing.getZip());
        stmt.setString(13, customer.getCreditCardNumber());
        stmt.setString(14, customer.getCreditCardExpirationDate());
        stmt.setString(15, shipping.getFirstName());
        stmt.setString(16, shipping.getLastName());
        stmt.setString(17, shipping.getAddress1());
        stmt.setString(18, shipping.getAddress2());
        stmt.setString(19, shipping.getCity());
        stmt.setString(20, shipping.getState());
        stmt.setString(21, shipping.getZip());
    } finally {
        cleanup(null, stmt, result);
    }
}
public static Customer getCustomer(int customerId) {

    Customer customer = null;
    Connection dbConnection = null;
    PreparedStatement stmt = null;
    ResultSet result = null;
    try {
        dbConnection = getConnection();
        stmt = dbConnection.prepareStatement(selectCustStmt);
        stmt.setInt(1, customerId);
        result = stmt.executeQuery();

        if (!result.next())
            throw new RecordNotFoundException("No Customer for primary key " + customerId);

        int i = 1;
        Calendar cal = Calendar.getInstance();
        String customerFirstName = result.getString(i++);
        String customerLastName = result.getString(i++);
        java.util.Date customerActivityDate = new java.util.Date(result.getDate(i++, cal).getTime());
        String customerBillingFirstName = result.getString(i++);
        String customerBillingLastName = result.getString(i++);
        String customerBillingAddr1 = result.getString(i++);
        String customerBillingAddr2 = result.getString(i++);
        String customerBillingCity = result.getString(i++);
        String customerBillingState = result.getString(i++);
        String customerBillingZip = result.getString(i++);
        String customerCardNum = result.getString(i++);
        String customerCardExpDate = result.getString(i++);
        String customerShippingFirstName = result.getString(i++);
        String customerShippingLastName = result.getString(i++);
        String customerShippingAddr1 = result.getString(i++);
        String customerShippingAddr2 = result.getString(i++);
        String customerShippingCity = result.getString(i++);
        String customerShippingState = result.getString(i++);
        String customerShippingZip = result.getString(i++);

        Address billing = new Address(customerBillingFirstName, customerBillingLastName,
            customerBillingAddr1, customerBillingAddr2, customerBillingCity, customerBillingState,
            customerBillingZip);
        Address shipping = new Address(customerShippingFirstName, customerShippingLastName,
            customerShippingAddr1, customerShippingAddr2, customerShippingCity, customerShippingState,
            customerShippingZip);

        customer = new Customer(customerId, customerFirstName, customerLastName, customerPhone,
            customerActivityDate, customerCardNum, customerCardExpDate, billing, shipping);
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        cleanup(dbConnection, stmt, result);
    }

    return customerId;
}
package com.doculabs.businessLogic;

import java.sql.*;
import java.util.*;

/**
 * An implementation of the CommonDAL specific to order information.
 * <BR>Copyright: Copyright (c) 2003
 * <BR>Company: Doculabs
 * @author unascribed
 * @version 1.0
 */
public class OrderDAL extends CommonDAL {

    private static final String orderSelectStmt = "SELECT ORD .CUSTOMERID, ORD.ORDERSTATUS,
    ORD.ORDERDATE, ORD.TOTALAMOUNT, CUST.CONTACTFIRSTNAME,
    CUST.CONTACTLASTNAME, " +
    "CUST.CONTACTPHONE, CUST.LASTACTIVITYDATE, CUST.BILLINGCREDITCARDNUMBER, CUST.BILLINGCREDITCARDEXPIRE, " +
    "CUST.BILLINGFIRSTNAME, CUST.BILLINGLASTNAME, CUST.BILLINGADDRESS1, CUST.BILLINGADDRESS2, CUST.BILLINGCITY, " +
    "CUST.BILLINGSTATE, CUST.BILLINGZIP, CUST.SHIPPINGFIRSTNAME, CUST.SHIPPINGLASTNAME, CUST.SHIPPINGADDRESS1, " +
    "CUST.SHIPPINGADDRESS2, CUST.SHIPPINGCITY, CUST.SHIPPINGSTATE, CUST.SHIPPINGZIP, LITM.ITEMID, LITM.PRODUCTID, PROD.PRODUCTDESCRIPTION, LITM.QUANTITY, LITM.AMOUNT " +
    "FROM ORDERS ORD, CUSTOMERS CUST, LINEITEMS LITM, PRODUCTS PROD " +
    "WHERE  ORD.ORDERID = ? AND CUST.CUSTOMERID=ORD.CUSTOMERID AND PROD.PRODUCTID=LITM.PRODUCTID AND LITM.ORDERID=ORD.ORDERID ";

    /**
     * A method to retrieve order information from database
     * @param orderId A unique identifier for an order
     * @return the method return an order thin data class
     */
    public static Order getOrderDetails(int orderId) {
        Order order = null;
        Connection dbConnection = null;
        PreparedStatement stmt = null;
        ResultSet result = null;
        try {
            //dbConnection = getDBConnection();
            dbConnection = getConnection();
            stmt = dbConnection.prepareStatement(orderSelectStmt);
            stmt.setInt(1, orderId);
            result = stmt.executeQuery();
            if ( !result.next() )
                throw new RecordNotFoundException("No Order for primary key " + orderId);

            int i = 1;
            Calendar cal = Calendar.getInstance();

            int custId = result.getInt(i++);
            int orderStatus = result.getInt(i++);
            java.util.Date orderDate = new java.util.Date(result.getDate(i++,cal).getTime());
            float orderAmt = result.getFloat(i++);
            String custContactFirstName = result.getString(i++);
            String custContactLastName = result.getString(i++);
            String custContactPhone = result.getString(i++);
            java.util.Date custActivityDate = new java.util.Date(result.getDate(i++,cal).getTime());

            Order order = new Order(custId, orderStatus, orderDate, orderAmt, custContactFirstName, custContactLastName, custContactPhone, custActivityDate);
            for ( ; i < result.getMetaData().getColumnCount(); i++) {
                String columnName = result.getMetaData().getColumnName(i);
                String columnValue = result.getString(i);
                order.addOrderDetails(columnName, columnValue);
            }

            return order;
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            try {
                if ( stmt != null )
                    stmt.close();
                if ( result != null )
                    result.close();
                if ( dbConnection != null )
                    dbConnection.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
        return null;
    }
}
String custCardNum = result.getString(i++);
String custCardExpDate = result.getString(i++);
String custBillingFirstName = result.getString(i++);
String custBillingLastName = result.getString(i++);
String custBillingAddr1 = result.getString(i++);
String custBillingAddr2 = result.getString(i++);
String custBillingCity = result.getString(i++);
String custBillingState = result.getString(i++);
String custBillingZip = result.getString(i++);
String custShippingFirstName = result.getString(i++);
String custShippingLastName = result.getString(i++);
String custShippingAddr1 = result.getString(i++);
String custShippingAddr2 = result.getString(i++);
String custShippingCity = result.getString(i++);
String custShippingState = result.getString(i++);
String custShippingZip = result.getString(i++);
int lineItemStrtColumn=i;

Address billing = new Address(custBillingFirstName, custBillingLastName, custBillingAddr1,
custBillingAddr2,
custBillingCity, custBillingState, custBillingZip);
Address shipping = new Address(custShippingFirstName, custShippingLastName, custShippingAddr1,
custShippingAddr2,
custShippingCity, custShippingState, custShippingZip);

Customer customer = new Customer(custId, custContactFirstName, custContactLastName,
custContactPhone,
custActivityDate, custCardNum, custCardExpDate, billing, shipping);

order = new Order(orderId, orderStatus, orderDate, orderAmt, customer, null);

ArrayList lineItems = new ArrayList();
do {
i=lineItemStrtColumn;
int itemId = result.getInt(i++);
int prodId = result.getInt(i++);
String prodDesc = result.getString(i++);
int qty = result.getInt(i++);
float amt = result.getFloat(i++);
lineItems.add(new LineItem(orderId, itemId, prodId, prodDesc, qty, amt));
} while(result.next());

order.setLineItems((LineItem[])lineItems.toArray(new LineItem[0]));

} catch(Exception e) {
e.printStackTrace();
} finally {
    cleanup(dbConnection, stmt, result);
}

return order;
Stateless Session EJB Classes

For simplicity only the local interfaces are listed but the application included definitions for remote interfaces.

For Application Servers X and Y an IDE was used to create the EJB classes and the required interfaces. With JBoss, XDoclet comments were placed in the code and the XDoclet parser was used to produce the relevant interfaces and deployment descriptors.

Customer Session Bean

CustomerSesBean.java

```java
package com.doculabs.businessLogic;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

/**
 * A session bean to access customer information
 * <BR>Copyright: Copyright (c) 2003
 * <BR>Company: Doculabs
 * @author unascribed
 * @version 1.0
 */
public class CustomerSesBean implements SessionBean {
    public void ejbCreate() {}
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbRemove() {}
    public Customer GetCustomer(int custId) {
        Customer cust = null;
        try {
            cust = CustomerDAL.getCustomer(custId);
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        return cust;
    }
    public int InsertCustomer(Customer cust) {
        int custId = -1;
        try {
            custId = CustomerDAL.insertCustomer(cust);
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        return custId;
    }
}
```
CustomerSesLocal.java

```java
package com.doculabs.businessLogic;
import javax.ejb.EJBLocalObject;
public interface CustomerSesLocal extends EJBLocalObject
{
    public Customer GetCustomer(int custId);
    public int InsertCustomer(Customer cust);
}
```

CustomerSesLocalHome.java

```java
package com.doculabs.businessLogic;
import javax.ejb.EJBLocalHome;
import javax.ejb.CreateException;
public interface CustomerSesLocalHome extends EJBLocalHome
{
    CustomerSesLocal create() throws CreateException;
}
```

Order Session Bean

OrderSesBean.java

```java
package com.doculabs.businessLogic;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
/**
 * A session bean to access order information
 * <BR>Copyright: Copyright (c) 2003
 * <BR>Company: Doculabs
 * @author unascribed
 * @version 1.0
 */
public class OrderSesBean implements SessionBean
{
    public void ejbCreate(){
    }
    public void ejbActivate(){
    }
    public void ejbPassivate(){
    }
    public void ejbRemove(){
    }
    public void setSessionContext(SessionContext ctx){
    }
    public Order GetOrderDetails(int orderId)
    {
        Order ord=null;
        try {
            ord = OrderDAL.getOrderDetails(orderId);
        } catch(Exception e) {
            e.printStackTrace();
        }
        return ord;
    }
}
```
**OrderSesLocal.java**

```java
package com.doculabs.businessLogic;
import javax.ejb.EJBLocalObject;

public interface OrderSesLocal extends EJBLocalObject
{
    public Order GetOrderDetails(int orderId);
}
```

**OrderSesLocalHome.java**

```java
package com.doculabs.businessLogic;
import javax.ejb.EJBLocalHome;
import javax.ejb.CreateException;

public interface OrderSesLocalHome extends EJBLocalHome
{
    OrderSesLocal create() throws CreateException;
}
```

**Non-EJB Implementation Classes**

Based on feedback in the code review, a non-EJB version of the application was implemented. The implementation logic is identical to the EJB versions except the EJB infrastructure code has been removed.

**Customer Plain Bean**

**CustomerPlainBean.java**

```java
package com.doculabs.businessLogic;

/**
 * A plain old java object java bean to access customer information
 * <BR>Copyright: Copyright (c) 2003
 * <BR>Company: Doculabs
 * <BR>@author unascribed
 * <BR>@version 1.0
 */
public class CustomerPlainBean
{
    public Customer GetCustomer(int custId)
    {
        Customer cust = null;
        try {
            cust = CustomerDAL.getCustomer(custId);
        } catch(Exception e) {
            e.printStackTrace();
        }
        return cust;
    }

    public Customer GetCustomerNoDB(int custId)
    {
        Customer cust = null;
        try {
            cust = CustomerDAL.getCustomerNoDB(custId);
        } catch(Exception e) {
            e.printStackTrace();
        }
        return cust;
    }
```
public int InsertCustomer(Customer cust) {
    int custId = -1;
    try {
        custId = CustomerDAL.insertCustomer(cust);
    } catch(Exception e) {
        e.printStackTrace();
    }
    return custId;
}

Order Plain Bean

OrderPlainBean.java

package com.doculabs.businessLogic;

/**
 * A plain old java object java bean to access order information
 *<BR>Copyright: Copyright (c) 2003
 *<BR>Company: Doculabs
 * @author unascribed
 * @version 1.0
 */
public class OrderPlainBean {
    public Order GetOrderDetails(int orderId) {
        Order ord = null;
        try {
            ord = OrderDAL.getOrderDetails(orderId);
        } catch(Exception e) {
            e.printStackTrace();
        }
        return ord;
    }
}

Miscellaneous Classes

RecordNotFoundException.java

package com.doculabs.businessLogic;

/**
 * An exception that is thrown if data is not found in the database.
 *
 */
public class RecordNotFoundException extends Exception implements java.io.Serializable {
    public RecordNotFoundException() {}
    public RecordNotFoundException(String msg) {
        super(msg);
    }
}
Java Web Service Facade Tier

Web Service

wsFacade.java

```java
package com.doculabs.webservices;

import com.doculabs.businessLogic.*;
import javax.ejb.CreateException;
import javax.naming.*;

public class wsFacade {
    private static CustomerSesLocalHome customerLocalHome = null;
    private static OrderSesLocalHome orderLocalHome = null;

    /*
     * EJB Implementation
     */
    public Customer getCustomer(int customerId) throws Exception{
        Customer customer = null;
        try{
            customer = getCustomerSes().getCustomer(customerId);
        }catch(Exception e){
            System.out.println("wsFacade getCustomer " + e.getMessage());
            e.printStackTrace();
            throw new Exception("Unable to process web service request");
        }
        return customer;
    }

    /*
     * EJB Implementation
     */
    public Customer getCustomerNoDB(int customerId) throws Exception{
        Customer customer = null;
        try{
            customer = getCustomerSes().getCustomerNoDB(customerId);
        }catch(Exception e){
            System.out.println("wsFacade getCustomerNoDB " + e.getMessage());
            e.printStackTrace();
            throw new Exception("Unable to process web service request");
        }
        return customer;
    }

    /*
     * EJB Implementation
     */
    public int insertCustomer(Customer customer) throws Exception{
        int customerId = -1;
        try{
            customerId = getCustomerSes().InsertCustomer(customer);
        }catch(Exception e){
            System.out.println("wsFacade insertCustomer " + e.getMessage());
            e.printStackTrace();
            throw new Exception("Unable to process web service request");
        }
        return customerId;
    }
}
```
public Order getOrderDetails(int orderId) throws Exception {
    Order order = null;
    try {
        order = getOrderSes().GetOrderDetails(orderId);
    } catch (Exception e) {
        System.out.println("wsFacade getorderdetails " + e.getMessage());
        e.printStackTrace();
        throw new Exception("Unable to process web service request");
    }
    return order;
}

/*
 * Non-EJB Implementation
 */
public Customer getCustomerPlain(int customerId) throws Exception {
    Customer customer = null;
    try {
        CustomerPlainBean customerBean = new CustomerPlainBean();
        customer = customerBean.GetCustomer(customerId);
    } catch (Exception e) {
        System.out.println("wsFacade getCustomer " + e.getMessage());
        e.printStackTrace();
        throw new Exception("Unable to process web service request");
    }
    return customer;
}

/*
 * Non-EJB Implementation
 */
public Customer getCustomerPlainNoDB(int customerId) throws Exception {
    Customer customer = null;
    try {
        CustomerPlainBean customerBean = new CustomerPlainBean();
        customer = customerBean.GetCustomerNoDB(customerId);
    } catch (Exception e) {
        System.out.println("wsFacade getCustomerNoDB " + e.getMessage());
        e.printStackTrace();
        throw new Exception("Unable to process web service request");
    }
    return customer;
}

/*
 * Non-EJB Implementation
 */
public int insertCustomerPlain(Customer customer) throws Exception {
    int customerId = -1;
    try {
        CustomerPlainBean customerBean = new CustomerPlainBean();
        customerId = customerBean.InsertCustomer(customer);
    } catch (Exception e) {
        System.out.println("wsFacade insertcustomer " + e.getMessage());
        e.printStackTrace();
    }
    return customerId;
}
throw new Exception("Unable to process web service request");
    }
    return customerId;
}
/*
* Non-EJB Implementation
*/
public Order getOrderDetailsPlain(int orderId) throws Exception{
    Order order = null;
    try{
        OrderPlainBean orderBean = new OrderPlainBean();
        order = orderBean.getOrderDetails(orderId);
    }catch(Exception e){
        System.out.println("wsFacade getorderdetails " + e.getMessage());
        e.printStackTrace();
        throw new Exception("Unable to process web service request");
    }
    return order;
}
private CustomerSesLocal getCustomerSes() throws CreateException, Exception{
    if (customerLocalHome != null){
        return customerLocalHome.create();
    } else{
        InitialContext initCtx = new InitialContext();
        try {
            initCtx = new InitialContext();
            customerLocalHome = (CustomerSesLocalHome) initCtx.lookup("local:ejb/ejb/com/doculabs/businessLogic/CustomerSesHome");
        return customerLocalHome.create();
    } catch(NamingException ne) {
        System.out.println("Failed to lookup Customer EJB. Please double check that the name is correct in the config file");
        throw new Exception("Failed to lookup Customer EJB "+ ne);
    } finally {
        try {
            if(initCtx != null) initCtx.close();
        } catch(NamingException ne) {
            System.out.println("Error closing context: "+ ne);
            throw new Exception("Error closing context: "+ ne);
        }
    }
}
private OrderSesLocal getOrderSes() throws CreateException, Exception{
    if (orderLocalHome != null){
        return orderLocalHome.create();
    } else{
        InitialContext initCtx = null;
        try {
            initCtx = new InitialContext();
            orderLocalHome = (OrderSesLocalHome) initCtx.lookup("local:ejb/ejb/com/doculabs/businessLogic/OrderSesHome");
        return orderLocalHome.create();
    } catch(NamingException ne) {
        System.out.println("Failed to lookup Order EJB. Please double check that the name is correct in the config file");
        throw new Exception("Failed to lookup Order EJB "+ ne);
    }
Java Web Service Client

The relevant tool for each application server was used to generate a web service client proxy. Typically the tool is given the URL for the web service wsdl file and it generates a java client file. This client file can then be called by a JSP page or middle tier logic file. For this test there was a client page for each method exposed as a web service, but for simplicity we have just listed the GetCustomer page.

Client_InsertCustomer.jsp

```html
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
  <%@ page language="java" import="java.net.*, java.text.*, com.doculabs.*" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1" %>
  
  <%! 
  WsFacade ws = null; 
  public void jspInit() { 
  if (ws == null){
      try{
          WsFacadeService service = new WsFacadeServiceLocator();
          ws = service.getwsFacade(new java.net.URL("http://192.168.1.206:8080/jboss-net/services/wsFacade"));
      }catch (Exception foo){
          System.out.println("The Server is not working");
      }
  }
  }
  public void jspDestroy() {
  }
  %>

  <title>Web Service Client Insert Customer</title>
  </head>
  
  <body>
  <%
  String randomString = "RANDOM DATA " + ((request.getParameter("randomNumber") != null) ? request.getParameter("randomNumber") : "1");
  %>
```
Customer customer = new Customer();
customer.setContactFirstName(randomString);
customer.setContactLastName(randomString);
customer.setContactPhoneNumber(randomString);
customer.setCreditCardNumber(randomString);
customer.setCreditCardExpirationDate(randomString);

Address address = new Address();
address.setFirstName(randomString);
address.setLastName(randomString);
address.setAddress1(randomString);
address.setAddress2(randomString);
address.setCity(randomString);
address.setState(randomString);
address.setZip(randomString);

customer.setShippingAddress(address);
customer.setBillingAddress(address);

customer.setCustomerId(0);
int customerId = 0;
synchronized(this){
    customerId = ws.insertCustomer(customer);
}

<form id="Form1" method="post">
  <table id="Table1" cellSpacing="1" cellPadding="1" width="400" border="1">
    <tr bgColor="yellow">
      <td><strong>Test</strong></td>
      <td><strong>Attribute</strong></td>
      <td><strong>Value</strong></td>
    </tr>
    <tr>
      <td>InsertCustomer</td>
      <td>Contact First Name</td>
      <td>% = customer.getContactFirstName()%</td>
    </tr>
    <tr>
      <td>InsertCustomer</td>
      <td>CustomerId</td>
      <td>% = customerId%</td>
    </tr>
    <tr>
      <td>InsertCustomer</td>
      <td>Contact First Name</td>
      <td>% = customer.getContactFirstName()%</td>
    </tr>
  </table>
</form>
Appendix H: Test Application Code – .NET Implementation

This section contains the application logic for the .NET implementation.

.NET Middle Tier

Thin Data Classes

Address.cs

using System;
namespace Doculabs.BusinessLogic {
    [Serializable]
    public class Address {
        private string firstName;
        private string lastName;
        private string address1;
        private string address2;
        private string city;
        private string state;
        private string zip;

        public Address()
        {
        }

        public Address(string firstName, string lastName, string address1, string address2, string city, string state, string zip)
        {
            this.firstName = firstName;
            this.lastName = lastName;
            this.address1 = address1;
            this.address2 = address2;
            this.city = city;
            this.state = state;
            this.zip = zip;
        }

        public string FirstName{
            get { return this.firstName; }
            set { this.firstName = value; }
        }

        public string LastName{
            get { return this.lastName; }
            set { this.lastName = value; }
        }

        public string Address1{
            get { return this.address1; }
            set { this.address1 = value; }
        }

        public string Address2{
            get { return this.address2; }
            set { this.address2 = value; }
        }

        public string City{
            get { return this.city; }
            set { this.city = value; }
        }
    }
}
public string State{
    get { return this.state; }
    set { this.state = value; }
}

public string Zip{
    get { return this.zip; }
    set { this.zip = value; }
}

Customer.cs

using System;
namespace Doculabs.BusinessLogic
{
    [Serializable]
    public class Customer
    {
        private int customerId;
        private string contactFirstName;
        private string contactLastName;
        private string contactPhone;
        private DateTime lastActivityDate;
        private string creditCardNumber;
        private string creditCardExpirationDate;
        private Address billingAddress;
        private Address shippingAddress;

        public Customer()
        {
        }

        public Customer(int customerId, string firstName, string lastName, string phone, DateTime lastActivityDate, string creditCardNumber, string creditCardExpirationDate, Address billingAddress, Address shippingAddress)
        {
            this.customerId = customerId;
            this.contactFirstName = firstName;
            this.contactLastName = lastName;
            this.contactPhone = phone;
            this.lastActivityDate = lastActivityDate;
            this.creditCardNumber = creditCardNumber;
            this.creditCardExpirationDate = creditCardExpirationDate;
            this.billingAddress = billingAddress;
            this.shippingAddress = shippingAddress;
        }

        public int CustomerId
        {
            get { return this.customerId; }
            set { this.customerId = value; }
        }

        public string ContactFirstName
        {
            get { return this.contactFirstName; }
            set { this.contactFirstName = value; }
        }

        public string ContactLastName
        {
            get { return this.contactLastName; }
            set { this.contactLastName = value; }
        }

        public string ContactPhone
        {
            get { return this.contactPhone; }
            set { this.contactPhone = value; }
        }

        public DateTime LastActivityDate
        {
            get { return this.lastActivityDate; }
            set { this.lastActivityDate = value; }
        }

        public string CreditCardNumber
        {
            get { return this.creditCardNumber; }
            set { this.creditCardNumber = value; }
        }

        public string CreditCardExpirationDate
        {
            get { return this.creditCardExpirationDate; }
            set { this.creditCardExpirationDate = value; }
        }
    }
}
LineItem.cs

using System;

namespace Doculabs.BusinessLogic{
    [Serializable]
    public class LineItem{
        private int orderId;
        private int itemId;
        private int productId;
        private string productDescription;
        private int orderQuantity;
        private float unitPrice;
        public LineItem(){
        }
        public LineItem(int orderId, int itemId, int productId, string productDescription, int qty, float unitPrice){
            this.orderId = orderId;
            this.itemId = itemId;
            this.productId = productId;
            this.productDescription = productDescription;
            this.orderQuantity = qty;
            this.unitPrice = unitPrice;
        }
        public int OrderId{
            get { return this.orderId; }
            set { this.orderId = value; }
        }
        public int ItemId{
            get { return this.itemId; }
            set { this.itemId = value; }
        }
        public int ProductId{
            get { return this.productId; }
            set { this.productId = value; }
        }
        public string ProductDescription{
            get { return this.productDescription; }
            set { this.productDescription = value; }
        }
        public int OrderQuantity{
            get { return this.orderQuantity; }
            set { this.orderQuantity = value; }
        }
        public float UnitPrice{
            get { return this.unitPrice; }
            set { this.unitPrice = value; }
        }
    }
}
using System;

namespace Doculabs.BusinessLogic
{
    [Serializable]
    public class Order
    {
        private int orderId;
        private int orderStatus;
        private DateTime orderDate;
        private float orderTotalAmount;
        private Customer customer;
        private LineItem[] lineItems;

        public Order()
        {
        }

        public Order(int orderId, int orderStatus, DateTime orderDate, float orderTotalAmount, Customer customer, LineItem[] lineItems)
        {
            this.orderId = orderId;
            this.orderStatus = orderStatus;
            this.orderDate = orderDate;
            this.orderTotalAmount = orderTotalAmount;
            this.customer = customer;
            this.lineItems = lineItems;
        }

        public int OrderId
        {
            get { return this.orderId; }
            set { this.orderId = value; }
        }

        public int OrderStatus
        {
            get { return this.orderStatus; }
            set { this.orderStatus = value; }
        }

        public DateTime OrderDate
        {
            get { return this.orderDate; }
            set { this.orderDate = value; }
        }

        public float OrderTotalAmount
        {
            get { return this.orderTotalAmount; }
            set { this.orderTotalAmount = value; }
        }

        public Customer Customer
        {
            get { return this.customer; }
            set { this.customer = value; }
        }

        public LineItem[] LineItems
        {
            get { return this.lineItems; }
            set { this.lineItems = value; }
        }
    }
}
Data Access Classes

Oracle Implementation

DatabaseOraDP.cs

```csharp
using System;
using System.Configuration;
using System.Data;
using System.Data.OracleClient;
using System.Collections;
namespace Doculabs.BusinessLogic{
    public abstract class Database {
        public static readonly string CONN_STRING = "Data Source=ORA03;user id=Doculabs;password=doculabs;Enlist=False;Min Pool Size=8; Max Pool Size=8";
        public static readonly string CONN_STRING2 = "Data Source=ORA03;user id=Doculabs;password=doculabs;Enlist=False;Min Pool Size=8; Max Pool Size=8";
        private static Hashtable parmCache = Hashtable.Synchronized(new Hashtable());
        public static int ExecuteNonQuery(string connString, CommandType cmdType, string cmdText, params OracleParameter[] cmdParms) {
            OracleCommand cmd = new OracleCommand();
            using (OracleConnection conn = new OracleConnection(connString)) {
                PrepareCommand(cmd, conn, null, cmdType, cmdText, cmdParms);
                int val = cmd.ExecuteNonQuery();
                cmd.Parameters.Clear();
                conn.Close();
                return val;
            }
        }
        public static int ExecuteNonQuery(OracleTransaction trans, CommandType cmdType, string cmdText, params OracleParameter[] cmdParms) {
            OracleCommand cmd = new OracleCommand();
            PrepareCommand(cmd, trans.Connection, trans, cmdType, cmdText, cmdParms);
            int val = cmd.ExecuteNonQuery();
            cmd.Parameters.Clear();
            cmd.Connection.Close();
            return val;
        }
        public static OracleDataReader ExecuteReader(string connString, CommandType cmdType, string cmdText, params OracleParameter[] cmdParms) {
            OracleCommand cmd = new OracleCommand();
            OracleConnection conn = new OracleConnection(connString);
            try {
                PrepareCommand(cmd, conn, null, cmdType, cmdText, cmdParms);
                OracleDataReader rdr = cmd.ExecuteReader(CommandBehavior.CloseConnection);
                return rdr;
            } catch (Exception e) {
                conn.Close();
                throw e;
            }
        }
        public static object ExecuteScalar(string connString, CommandType cmdType, string cmdText, params OracleParameter[] cmdParms) {
```
public static void CacheParameters(string cacheKey, params OracleParameter[] cmdParms) {
    parmCache[cacheKey] = cmdParms;
}

public static OracleParameter[] GetCachedParameters(string cacheKey) {
    OracleParameter[] cachedParms = (OracleParameter[])parmCache[cacheKey];
    if (cachedParms == null)
        return null;
    OracleParameter[] clonedParms = new OracleParameter[cachedParms.Length];
    for (int i = 0, j = cachedParms.Length; i < j; i++)
        clonedParms[i] = (OracleParameter)((ICloneable)cachedParms[i]).Clone();
    return clonedParms;
}

private static void PrepareCommand(OracleCommand cmd, OracleConnection conn, OracleTransaction trans, CommandType cmdType, string cmdText, OracleParameter[] cmdParms) {
    if (conn.State != ConnectionState.Open)
        conn.Open();
    cmd.Connection = conn;
    cmd.CommandText = cmdText;
    if (trans != null)
        cmd.Transaction = trans;
    cmd.CommandType = cmdType;
    if (cmdParms != null) {
        foreach (OracleParameter parm in cmdParms)
            cmd.Parameters.Add(parm);
    }
}

CustomerDAL.cs
using System;
using System.Collections;
using System.Data;
using System.Data.OracleClient;
namespace Doculabs.BusinessLogic {
    public class CustomerDAL {
        private static readonly string SQL_GET_SEQ = "SELECT Doculabs.CUSTOMER_SEQ.CURRVAL FROM DUAL";
        private static readonly string SQL_INSERT_CUSTOMER = "INSERT INTO Doculabs.Customers (CustomerID, ContactFirstName, ContactLastName, ContactPhone, LastActivityDate, BillingFirstName, BillingLastName, BillingAddress1, BillingAddress2, BillingCity, BillingState, BillingZip, BillingCreditCardNumber, BillingCreditCardExpire, ShippingFirstName, ShippingLastName, ShippingAddress1, ShippingAddress2, ShippingCity, ShippingState, ShippingZip) VALUES (Doculabs.CUSTOMER_SEQ.NEXTVAL, :ContactFirstName, :ContactLastName, :ContactPhone, :LastActivityDate, :BillingFirstName, :BillingLastName, :BillingAddress1, :BillingAddress2, :BillingCity, :BillingState, :BillingZip, :BillingCreditCardNumber, :BillingCreditCardExpire, :ShippingFirstName, :ShippingLastName, :ShippingAddress1, :ShippingAddress2, :ShippingCity, :ShippingState, :ShippingZip)";
    }
}
public static Customer GetCustomerNoDB(int customerId) {
    Address shipping = new Address("Shipping First Name", "Shipping Last Name", "Shipping Address1", "Shipping Address2", "Shipping City", "Shipping State", "Shipping Zip");
    DateTime lastActivity = DateTime.Now;
    return new Customer(customerId, "Contact First Name", "Contact Last Name", "Contact Phone", lastActivity, "Credit Card Number", "12/05", billing, shipping);
}

public static Customer GetCustomer(int customerId) {
    Customer customer = null;
    OracleParameter parm = new OracleParameter(":CUSTOMERID", OracleType.Int32);
    parm.Value = customerId;
    using (OracleDataReader rdr = Database.ExecuteReader(Database.CONN_STRING, CommandType.Text, SQL_SELECT_CUSTOMER, parm)) {
        int i = 1;
        if (rdr.Read()) {
            string contactFirstName = rdr.GetString(i++);
            string contactLastName = rdr.GetString(i++);
            string contactPhone = rdr.GetString(i++);
            DateTime lastActivityDate = rdr.GetDateTime(i++);
            string customerBillingFirstName = rdr.GetString(i++);
            string customerBillingLastName = rdr.GetString(i++);
            string customerBillingAddress1 = rdr.GetString(i++);
            string customerBillingAddress2 = rdr.GetString(i++);
            string customerBillingCity = rdr.GetString(i++);
            string customerBillingState = rdr.GetString(i++);
            string customerBillingZip = rdr.GetString(i++);
            string creditCardNumber = rdr.GetString(i++);
            string creditCardExpirationDate = rdr.GetString(i++);
            string customerShippingFirstName = rdr.GetString(i++);
            string customerShippingLastName = rdr.GetString(i++);
            string customerShippingAddress1 = rdr.GetString(i++);
            return new Customer(customerId, contactFirstName, contactLastName, contactPhone, lastActivityDate, customerBillingFirstName, customerBillingLastName, customerBillingAddress1, customerBillingAddress2, customerBillingCity, customerBillingState, customerBillingZip, creditCardNumber, creditCardExpirationDate, customerShippingFirstName, customerShippingLastName, customerShippingAddress1);
        }
    }
}

private static readonly string SQL_SELECT_CUSTOMER = 
    "SELECT CustomerID, ContactFirstName, ContactLastName, ContactPhone, LastActivityDate, BillingFirstName, BillingLastName, BillingAddress1, BillingAddress2, BillingCity, BillingState, BillingZip, BillingCreditCardNumber, BillingCreditCardExpire, ShippingFirstName, ShippingLastName, ShippingAddress1, ShippingAddress2, ShippingCity, ShippingState, ShippingZip FROM Doculabs.Customers WHERE CustomerID = :CUSTOMERID";
```csharp
    string customerShippingAddress2 = rdr.GetString(i++);
    string customerShippingCity = rdr.GetString(i++);
    string customerShippingState = rdr.GetString(i++);
    string customerShippingZip = rdr.GetString(i++);

    Address billing = new Address(customerBillingFirstName, customerBillingLastName, 
        customerBillingAddress1, customerBillingAddress2, 
        customerBillingCity, customerBillingState, customerBillingZip);

    Address shipping = new Address(customerShippingFirstName, customerShippingLastName, 
        customerShippingAddress1, customerShippingAddress2, 
        customerShippingCity, customerShippingState, customerShippingZip);

    customer = new Customer(customerId, 
        contactFirstName, contactLastName, contactPhone, 
        lastActivityDate, 
        creditCardNumber, creditCardExpirationDate, billing, shipping);

    return customer;
}
}

public static int InsertCustomer(Customer customer)
{
    int customerId = -1;

    Address billing = customer.BillingAddress;
    Address shipping = customer.ShippingAddress;

    OracleParameter[] customerParms = GetCustomerParameters(SQL_INSERT_CUSTOMER);

    customerParms[0].Value = customer.ContactFirstName;
    customerParms[1].Value = customer.ContactLastName;
    customerParms[2].Value = customer.ContactPhone;
    customerParms[3].Value = customer.LastActivityDate;
    customerParms[4].Value = billing.FirstName;
    customerParms[5].Value = billing.LastName;
    customerParms[6].Value = billing.Address1;
    customerParms[7].Value = billing.Address2;
    customerParms[8].Value = billing.City;
    customerParms[9].Value = billing.State;
    customerParms[10].Value = billing.Zip;
    customerParms[12].Value = customer.CreditCardExpirationDate;
    customerParms[13].Value = shipping.FirstName;
    customerParms[14].Value = shipping.LastName;
    customerParms[15].Value = shipping.Address1;
    customerParms[16].Value = shipping.Address2;
    customerParms[17].Value = shipping.City;
    customerParms[18].Value = shipping.State;
    customerParms[19].Value = shipping.Zip;

    Database.ExecuteNonQuery(Database.CONN_STRING, 
        CommandType.Text, SQL_INSERT_CUSTOMER, customerParms);

    customerId = Convert.ToInt32(Database.ExecuteScalar(Database.CONN_STRING, 
        CommandType.Text, SQL_GET_SEQ, null));

    return customerId;
}

private static OracleParameter[] GetCustomerParameters(string key)
{
    OracleParameter[] parms = Database.GetCachedParameters(key);

    if (parms == null)
    
```
```csharp
{ new OracleParameter("ContactFirstName", OracleType.VarChar, 50),
  new OracleParameter("ContactLastName", OracleType.VarChar, 50),
  new OracleType.VarChar, 50),
  new OracleParameter("ContactPhone",
  new OracleType.VarChar, 50),
  new OracleParameter("LastActivityDate", OracleType.DateTime),
  new OracleParameter("BillingFirstName", OracleType.VarChar, 50),
  new OracleParameter("BillingLastName", OracleType.VarChar, 50),
  new OracleParameter("BillingAddress1", OracleType.VarChar, 50),
  new OracleParameter("BillingAddress2", OracleType.VarChar, 50),
  new OracleParameter("BillingCity", OracleType.VarChar, 50),
  new OracleParameter("BillingState", OracleType.VarChar, 50),
  new OracleParameter("BillingZip", OracleType.VarChar, 50),
  new OracleParameter("BillingCreditCardNumber", OracleType.VarChar, 50),
  new OracleParameter("BillingCreditCardExpire", OracleType.VarChar, 50),
  new OracleParameter("ShippingFirstName", OracleType.VarChar, 50),
  new OracleParameter("ShippingLastName", OracleType.VarChar, 50),
  new OracleParameter("ShippingAddress1", OracleType.VarChar, 50),
  new OracleParameter("ShippingAddress2", OracleType.VarChar, 50),
  new OracleParameter("ShippingCity", OracleType.VarChar, 50),
  new OracleParameter("ShippingState", OracleType.VarChar, 50),
  new OracleParameter("ShippingZip", OracleType.VarChar, 50)};
DataBase.CacheParameters(key, parms);
return parms;
}

OrderDAL.cs
using System;
using System.Collections;
using System.Data;
using System.Data.OracleClient;
namespace Doculabs.BusinessLogic {
    public class OrderDAL {
        private static readonly string SQL_SELECT_ORDER = "SELECT ORD.CUSTOMERID, ORD.ORDERSTATUS, ORD.ORDERDATE, ORD.TOTALAMOUNT, CUSTCONTACTFIRSTNAME, CUSTCONTACTLASTNAME, CUSTCONTACTPHONE, CUSTLASTACTIVITYDATE, CUSTBILLINGCREDITCARDNUMBER, CUSTBILLINGCREDITCARDEXPRIE, CUSTBILLINGADDRESS1, CUSTBILLINGADDRESS2, CUSTBILLINGCITY, CUSTBILLINGSTATE, CUSTBILLINGZIP, CUSTSHIPPINGFIRSTNAME, CUSTSHIPPINGLASTNAME, CUSTSHIPPINGADDRESS1, CUSTSHIPPINGADDRESS2, CUSTSHIPPINGCITY, CUSTSHIPPINGSTATE, CUSTSHIPPINGZIP, LITM.QUANTITY, LITM.AMOUNT " + "FROM ORDERS ORD, CUSTOMERS CUST, LINEITEMS LITM, PRODUCTS PROD " + "WHERE ORD.CUSTOMERID = CUST.CUSTOMERID AND ORD.ORDERID = LITM.ORDERID AND LITM.PRODUCTID = PROD.PRODUCTID;"
    }
}
```
WHERE ORD.ORDERID = :orderId AND CUST.CUSTOMERID = ORD.CUSTOMERID AND PROD.PRODUCTID = LITM.PRODUCTID AND LITM.ORDERID = ORD.ORDERID *;

private static readonly string PARM_ORDERID = " :orderId ";

public static Order GetOrderDetails(int orderId)
{
    Order order = null;
    OracleParameter parm = new OracleParameter(PARM_ORDERID, OracleType.Int32);
    parm.Value = orderId;
    using (OracleDataReader rdr =
        Database.ExecuteReader(Database.CONN_STRING, CommandType.Text,
            SQL_SELECT_ORDER, parm))
    {
        if (rdr.Read())
        {
            int customerId = rdr.GetInt32(0);
            int orderStatus = rdr.GetInt32(1);
            DateTime orderDate = rdr.GetDateTime(2);
            float orderTotalAmount = rdr.GetFloat(3);
            string contactFirstName = rdr.GetString(4);
            string contactLastName = rdr.GetString(5);
            string contactPhone = rdr.GetString(6);
            DateTime lastActivityDate = rdr.GetDateTime(7);
            string customerBillingFirstName = rdr.GetString(8);
            string customerBillingLastName = rdr.GetString(9);
            string customerBillingAddress1 = rdr.GetString(10);
            string customerBillingAddress2 = rdr.GetString(11);
            string customerBillingCity = rdr.GetString(12);
            string customerBillingState = rdr.GetString(13);
            string customerBillingZip = rdr.GetString(14);
            string creditCardNumber = rdr.GetString(15);
            string creditCardExpirationDate = rdr.GetString(16);
            string customerShippingFirstName = rdr.GetString(17);
            string customerShippingLastName = rdr.GetString(18);
            string customerShippingAddress1 = rdr.GetString(19);
            string customerShippingAddress2 = rdr.GetString(20);
            string customerShippingCity = rdr.GetString(21);
            string customerShippingState = rdr.GetString(22);
            string customerShippingZip = rdr.GetString(23);

            Address billing = new Address(customerBillingFirstName, customerBillingLastName, customerBillingAddress1, customerBillingAddress2, customerBillingCity, customerBillingState, customerBillingZip);
            Address shipping = new Address(customerShippingFirstName, customerShippingLastName, customerShippingAddress1, customerShippingAddress2, customerShippingCity, customerShippingState, customerShippingZip);

            Customer customer = new Customer(customerId, contactFirstName, contactLastName, contactPhone, lastActivityDate, orderDate, orderStatus, orderTotalAmount, contactFirstName, contactLastName, contactPhone, billing, shipping, orderId);
            order.Customer = customer;
        }
        return order;
    }
}
creditCardNumber, creditCardExpirationDate, billing, shipping);

ArrayList items = new ArrayList();
do{
    int itemId = rdr.GetInt32(24);
    int productId=rdr.GetInt32(25);
    string itemDescription=rdr.GetString(26);
    rdr.GetInt32(27);
    rdr.GetFloat(28);
    LineItem item = new LineItem(orderId, itemId, productId, itemDescription, quantity, unitPrice)
    items.Add(item);
} while ( rdr.Read() );

order = new Order(orderId,
orderStatus, orderDate, orderTotalAmount, customer, (LineItem[])items.ToArray(typeof(LineItem))
);

return order;

}
```csharp
int val = cmd.ExecuteNonQuery();
cmd.Parameters.Clear();
return val;
}

public static SqlDataReader ExecuteReader(string connString, CommandType cmdType, string cmdText, params SqlParameter[] cmdParms)
{
    SqlCommand cmd = new SqlCommand();
    SqlConnection conn = new SqlConnection(connString);
    try
    {
        PrepareCommand(cmd, conn, null, cmdType,
                        cmdText, cmdParms);
        SqlDataReader rdr = cmd.ExecuteReader(CommandBehavior.CloseConnection);
        cmd.Parameters.Clear();
        return rdr;
    }
    catch
    {
        conn.Close();
        throw;
    }
}

public static object ExecuteScalar(string connString, CommandType cmdType, string cmdText, params SqlParameter[] cmdParms)
{
    SqlCommand cmd = new SqlCommand();
    using (SqlConnection conn = new SqlConnection(connString))
    {
        PrepareCommand(cmd, conn, null, cmdType,
                        cmdText, cmdParms);
        object val = cmd.ExecuteScalar();
        cmd.Parameters.Clear();
        return val;
    }
}

public static SqlParameter[] CacheParameters(string cacheKey, params SqlParameter[] cmdParms)
{
    parmCache[cacheKey] = cmdParms;
}

public static SqlParameter[] GetCachedParameters(string cacheKey)
{
    SqlParameter[] cachedParms = (SqlParameter[])parmCache[cacheKey];
    if (cachedParms == null)
        return null;
    SqlParameter[] clonedParms = new SqlParameter[cachedParms.Length];
    for (int i = 0, j = cachedParms.Length; i < j; i++)
        clonedParms[i] = (SqlParameter)((ICloneable)cachedParms[i]).Clone();
    return clonedParms;
}

private static void PrepareCommand(SqlCommand cmd, SqlConnection conn, SqlTransaction trans, CommandType cmdType, string cmdText, SqlParameter[] cmdParms)
{
    if (conn.State != ConnectionState.Open)
        conn.Open();
    cmd.Connection = conn;
    cmd.CommandText = cmdText;
    if (trans != null)
        cmd.Transaction = trans;
    if (!trans.IsOpen)
        throw new ArgumentException
```
CustomerDAL.cs

```csharp
using System;
using System.Collections;
using System.Data;
using System.Data.SqlClient;
namespace Doculabs.BusinessLogic
{
    public class CustomerDAL
    {
        private static readonly string SQL_INSERT_CUSTOMER =
            "INSERT INTO Customers (ContactFirstName, ContactLastName, ContactPhone,
            LastActivityDate, BillingFirstName, BillingLastName, BillingAddress1,
            BillingAddress2, BillingCity, BillingState, BillingZip,BillingCreditCardNumber, BillingCreditCardExpire,
            ShippingFirstName, ShippingLastName, ShippingAddress1, ShippingAddress2,
            ShippingCity, ShippingState, ShippingZip) VALUES (@ContactFirstName,
            @ContactLastName, @ContactPhone, @LastActivityDate, @BillingFirstName,
            @BillingLastName, @BillingAddress1, @BillingAddress2, @BillingCity,
            @BillingState, @BillingZip, @BillingCreditCardNumber, @BillingCreditCardExpire,
            @ShippingFirstName, @ShippingLastName, @ShippingAddress1, @ShippingAddress2,
            @ShippingCity, @ShippingState, @ShippingZip); Select @@IDENTITY;"
        private static readonly string SQL_SELECT_CUSTOMER =
            "SELECT CustomerID, ContactFirstName, ContactLastName, ContactPhone,
            LastActivityDate, BillingFirstName, BillingLastName, BillingAddress1,
            BillingAddress2, BillingCity, BillingState, BillingZip,BillingCreditCardNumber, BillingCreditCardExpire,
            ShippingFirstName, ShippingLastName, ShippingAddress1, ShippingAddress2,
            ShippingCity, ShippingState, ShippingZip FROM Customers WHERE CustomerID = @CUSTOMERID;"

        public static Customer GetCustomerNoDB(int customerId) {
            return new Customer(customerId, "Contact First Name", "Contact Last Name", "Contact Phone",
                new DateTime(), "Credit Card Number", "12/05",
                new Address("Billing First Name", "Billing Last Name", "Billing Address1", "Billing Address2",
                "Billing City", "Billing State", "Billing Zip"),
                new Address("Shipping First Name", "Shipping Last Name", "Shipping Address1", "Shipping Address2",
                "Shipping City", "Shipping State", "Shipping Zip"),
                DateTime.Now);
        }

        public static Customer GetCustomer(int customerId) {
            Customer customer = null;
            SqlParameter parm = new SqlParameter("@CUSTOMERID", SqlDbType.Int);
            parm.Value = customerId;

            using (SqlDataReader rdr = Database.ExecuteReader(Database.CONN_STRING, CommandType.Text, SQL_SELECT_CUSTOMER, parm))
            {
                int i = 1;
                while (rdr.Read())
                {
                    int customerID = rdr.GetInt32(i);
                    string contactFirstName = rdr.GetString(i + 1);
                    string contactLastName = rdr.GetString(i + 2);
                    string contactPhone = rdr.GetString(i + 3);
                    DateTime lastActivity = rdr.GetDateTime(i + 4);
                    string billingFirstName = rdr.GetString(i + 5);
                    string billingLastName = rdr.GetString(i + 6);
                    string billingAddress1 = rdr.GetString(i + 7);
                    string billingAddress2 = rdr.GetString(i + 8);
                    string billingCity = rdr.GetString(i + 9);
                    string billingState = rdr.GetString(i + 10);
                    string billingZip = rdr.GetString(i + 11);
                    string billingCreditCardNumber = rdr.GetString(i + 12);
                    string billingCreditCardExpire = rdr.GetString(i + 13);
                    string shippingFirstName = rdr.GetString(i + 14);
                    string shippingLastName = rdr.GetString(i + 15);
                    string shippingAddress1 = rdr.GetString(i + 16);
                    string shippingAddress2 = rdr.GetString(i + 17);
                    string shippingCity = rdr.GetString(i + 18);
                    string shippingState = rdr.GetString(i + 19);
                    string shippingZip = rdr.GetString(i + 20);

                    customer = new Customer(customerID, contactFirstName, contactLastName, contactPhone,
                        lastActivity, billingFirstName, billingLastName, billingAddress1, billingAddress2,
                        billingCity, billingState, billingZip, billingCreditCardNumber, billingCreditCardExpire,
                        shippingFirstName, shippingLastName, shippingAddress1, shippingAddress2,
                        shippingCity, shippingState, shippingZip);
                }
            }
            return customer;
        }
    }
}
```
if ( rdr.Read() ) {
    string contactFirstName = rdr.GetString(i++);
    string contactLastName = rdr.GetString(i++);
    string contactPhone = rdr.GetString(i++);
    DateTime lastActivityDate = rdr.GetDateTime(i++);
    string customerBillingFirstName = rdr.GetString(i++);
    string customerBillingLastName = rdr.GetString(i++);
    string customerBillingAddress1 = rdr.GetString(i++);
    string customerBillingAddress2 = rdr.GetString(i++);
    string customerBillingCity = rdr.GetString(i++);
    string customerBillingState = rdr.GetString(i++);
    string customerBillingZip = rdr.GetString(i++);
    string creditCardNumber = rdr.GetString(i++);
    string creditCardExpirationDate = rdr.GetString(i++);
    string customerShippingFirstName = rdr.GetString(i++);
    string customerShippingLastName = rdr.GetString(i++);
    string customerShippingAddress1 = rdr.GetString(i++);
    string customerShippingAddress2 = rdr.GetString(i++);
    string customerShippingCity = rdr.GetString(i++);
    string customerShippingState = rdr.GetString(i++);
    string customerShippingZip = rdr.GetString(i++);

    Address billing = new Address(customerBillingFirstName, customerBillingLastName, customerBillingAddress1, customerBillingAddress2, customerBillingCity, customerBillingState, customerBillingZip);
    Address shipping = new Address(customerShippingFirstName, customerShippingLastName, customerShippingAddress1, customerShippingAddress2, customerShippingCity, customerShippingState, customerShippingZip);

    customer = new Customer(customerId, contactFirstName, contactLastName, contactPhone, lastActivityDate, creditCardNumber, creditCardExpirationDate, billing, shipping);
}

return customer;

public static int InsertCustomer(Customer customer) {
    int customerId = -1;

    Address billing = customer.BillingAddress;
    Address shipping = customer.ShippingAddress;

    SqlParameter[] customerParms = GetCustomerParameters(SQL_INSERT_CUSTOMER);
    customerParms[0].Value = customer.ContactFirstName;
    customerParms[1].Value = customer.ContactLastName;
    customerParms[2].Value = customer.LastActivityDate;
    customerParms[3].Value = billing.FirstName;
    customerParms[4].Value = billing.LastName;
customerParms[6].Value = billing.Address1;
customerParms[7].Value = billing.Address2;
customerParms[8].Value = billing.City;
customerParms[9].Value = billing.State;
customerParms[10].Value = billing.Zip;
customerParms[12].Value =
customer.CreditCardExpirationDate;
customerParms[13].Value = shipping.FirstName;
customerParms[14].Value = shipping.LastName;
customerParms[15].Value = shipping.Address1;
customerParms[16].Value = shipping.Address2;
customerParms[17].Value = shipping.City;
customerParms[18].Value = shipping.State;
customerParms[19].Value = shipping.Zip;

customerId = Convert.ToInt32(Database.ExecuteScalar(Database.CONN_STRING,
CommandType.Text, SQL_INSERT_CUSTOMER, customerParms));

return customerId;

 private static SqlParameter[] GetCustomerParameters(string key)
{
 SqlParameter[] parms =
 Database.GetCachedParameters(key);

 if (parms == null)
 {
 parms = new SqlParameter[] {
 new SqlParameter("@ContactFirstName", SqlDbType.VarChar, 50),
 new SqlParameter("@ContactLastName", SqlDbType.VarChar, 50),
 new SqlParameter("@ContactPhone", SqlDbType.VarChar, 50),
 new SqlParameter("@LastActivityDate", SqlDbType.DateTime),
 new SqlParameter("@BillingFirstName", SqlDbType.VarChar, 50),
 new SqlParameter("@BillingLastName", SqlDbType.VarChar, 50),
 new SqlParameter("@BillingAddress1", SqlDbType.VarChar, 50),
 new SqlParameter("@BillingAddress2", SqlDbType.VarChar, 50),
 new SqlParameter("@BillingCity", SqlDbType.VarChar, 50),
 new SqlParameter("@BillingState", SqlDbType.VarChar, 50),
 new SqlParameter("@BillingZip", SqlDbType.VarChar, 50),
 new SqlParameter("@BillingCreditCardNumber", SqlDbType.VarChar, 50),
 new SqlParameter("@BillingCreditCardExpire", SqlDbType.VarChar, 50),
 new SqlParameter("@ShippingFirstName", SqlDbType.VarChar, 50),
 new SqlParameter("@ShippingLastName", SqlDbType.VarChar, 50),
 new SqlParameter("@ShippingAddress1", SqlDbType.VarChar, 50),
 new SqlParameter("@ShippingAddress2", SqlDbType.VarChar, 50),
 new SqlParameter("@ShippingCity", SqlDbType.VarChar, 50),
 new SqlParameter("@ShippingState", SqlDbType.VarChar, 50),
 new SqlParameter("@ShippingZip", SqlDbType.VarChar, 50)
 };

 Database.CacheParameters(key, parms);

 return parms;
}
OrderDAL.cs

```csharp
using System;
using System.Collections;
using System.Data;
using System.Data.SqlClient;
namespace Doculabs.BusinessLogic {
    public class OrderDAL {
        private static readonly string SQL_SELECT_ORDER = "SELECT
ORD.CUSTOMERID, ORD.ORDERSTATUS, ORD.ORDERDATE, ORD.TOTALAMOUNT,
CUST.CONTACTFIRSTNAME, CUST.CONTACTLASTNAME, " +
"CUST.CONTACTPHONE, CUST.LASTACTIVITYDATE,
CUST.BILLINGCREDCARDNUMBER, CUST.BILLINGCREDITCARDEXPIRE, " +
"CUST.BILLINGFIRSTNAME, CUST.BILLINGLASTNAME,
CUST.BILLINGADDRESS1, CUST.BILLINGADDRESS2, CUST.BILLINGCITY, " +
"CUST.BILLINGSTATE, CUST.BILLINGZIP,
CUST.SHIPPINGFIRSTNAME, CUST.SHIPPINGLASTNAME, CUST.SHIPPINGADDRESS1, " +
"CUST.SHIPPINGADDRESS2, CUST.SHIPPINGCITY,
CUST.SHIPPINGSTATE, CUST.SHIPPINGZIP, LITM.ITEMID, " +
"LITM.PRODUCTID, PROD.PRODUCTDESCRIPTION,
LITM.QUANTITY, LITM.AMOUNT " +
"FROM ORDERS ORD, CUSTOMERS CUST, LINEITEMS LITM,
PRODUCTS PROD " +
"WHERE ORD.ORDERID = @orderId AND
CUST.CUSTOMERID=ORD.CUSTOMERID AND PROD.PRODUCTID=LITM.PRODUCTID AND
LITM.ORDERID=ORD.ORDERID;"
        
        private static readonly string PARM_ORDERID = "@orderId";
        public static Order GetOrderDetails(int orderId)
        {
            Order order = null;
            SqlParameter parm = new SqlParameter(PARM_ORDERID,
SqlDbType.Int);
            parm.Value = orderId;
            using (SqlDataReader rdr =
            Database.ExecuteReader(Database.CONN_STRING, CommandType.Text,
            SQL_SELECT_ORDER, parm)){
                if (rdr.Read()) {
                    int customerId=rdr.GetInt32(0);
                    int orderStatus=rdr.GetInt32(1);
                    DateTime orderDate=rdr.GetDateTime(2);
                    float orderTotalAmount = (float)
                    rdr.GetDecimal(3);
                    string contactFirstName=rdr.GetString(4);
                    string contactLastName=rdr.GetString(5);
                    string contactPhone=rdr.GetString(6);
                    DateTime lastActivityDate=rdr.GetDateTime(7);
                    string customerBillingFirstName=rdr.GetString(8);
                    string customerBillingLastName=rdr.GetString(9);
                    string customerBillingAddress1=rdr.GetString(10);
                    string customerBillingAddress2=rdr.GetString(11);
                    string customerBillingCity=rdr.GetString(12);
                    string customerBillingState=rdr.GetString(13);
                    string customerBillingZip=rdr.GetString(14);
                    string creditCardNumber=rdr.GetString(15);
```
string creditCardExpirationDate = rdr.GetString(16);
string customerShippingFirstName = rdr.GetString(17);
string customerShippingLastName = rdr.GetString(18);
string customerShippingAddress1 = rdr.GetString(19);
string customerShippingAddress2 = rdr.GetString(20);
string customerShippingCity = rdr.GetString(21);
string customerShippingState = rdr.GetString(22);
string customerShippingZip = rdr.GetString(23);
    Address billing = new Address(customerBillingFirstName, customerBillingLastName, customerBillingAddress1, customerBillingAddress2, customerBillingState, customerBillingCity, customerBillingZip);
    Address shipping = new Address(customerShippingFirstName, customerShippingLastName, customerShippingAddress1, customerShippingAddress2, customerShippingState, customerShippingCity, customerShippingZip);
    Customer customer = new Customer(customerId, contactFirstName, contactLastName, contactPhone, lastActivityDate, creditCardNumber, creditCardExpirationDate, billing, shipping);
    ArrayList items = new ArrayList();
    do {
        int itemId = rdr.GetInt32(24);
        int productId = rdr.GetInt32(25);
        string itemDescription = rdr.GetString(26);
        rdr.GetInt32(27);
        rdr.GetDecimal(28);
        LineItem item = new LineItem(orderId, itemId, productId, itemDescription, quantity, unitPrice);
        items.Add(item);
    } while (rdr.Read());
    order = new Order(orderId, orderStatus, orderDate, orderTotalAmount, customer, (LineItem[])items.ToArray(typeof(LineItem)));
    return order;
Business Logic Classes

CustomerBL.cs

```csharp
namespace Doculabs.BusinessLogic
{
    public class CustomerBL
    {
        public Customer GetCustomer(int customerId)
        {
            return CustomerDAL.GetCustomer(customerId);
        }

        public int InsertCustomer(Customer customer)
        {
            return CustomerDAL.InsertCustomer(customer);
        }
    }
}
```

OrderBL.cs

```csharp
namespace Doculabs.BusinessLogic
{
    public class OrderBL
    {
        public Order GetOrderDetails(int orderId)
        {
            return OrderDAL.GetOrderDetails(orderId);
        }
    }
}
```
.NET Web Service Facade Tier

Web Service

wsFacade.asmx

```
<%@ WebService Language="c#" Codebehind="wsFacade.asmx.cs"
Class="WebService.wsFacade" Debug="false"%>
```

wsFacade.asmx.cs

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;
using Doculabs.BusinessLogic;

namespace WebService
{
    /// <summary>
    /// Summary description for wsFacade.
    /// </summary>
    {
        public wsFacade()
        {
            //CODEGEN: This call is required by the ASP.NET Web Services Designer
            InitializeComponent();
        }

        [WebMethod]
        public Order GetOrderDetails(int orderId){
            OrderBL order = new OrderBL();
            return order.GetOrderDetails(orderId);
        }

        [WebMethod]
        public Customer GetCustomer(int customerId){
            CustomerBL customerBL = new CustomerBL();
            return customerBL.GetCustomer(customerId);
        }

        [WebMethod]
        public int InsertCustomer(Customer customer){
            CustomerBL wsCustomer = new CustomerBL();
            return wsCustomer.InsertCustomer(customer);
        }
    }
}
```
/// Clean up any resources being used.
/// </summary>
protected override void Dispose(bool disposing) {
    if (disposing && components != null) {
        components.Dispose();
    }
    base.Dispose(disposing);
}

.NET Web Service Client

Visual Studio .NET was used to add a reference to the web service in the client application project. The framework utility wsdl.exe could also have been used directly from the command line to generate the web service proxy class. The generated proxy is then referenced directly from client aspx pages or from middle tier business logic files. For this test there was a client page for each method exposed as a web service, but for simplicity we have just listed the GetCustomer page.

Client_InsertCustomer.aspx

```csharp
<%@ Page language="c#" %>
<%@ Import Namespace="DoculabsDotNetProxy.DotNetWS" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<title>.NET Web Services Insert Customer Client</title>
</head>
<body>
<% string randomString = "RANDOM DATA " + (Request["randomNumber"] != null) ? Request["randomNumber"] : "1";
wsFacade ws = new wsFacade();
customer customer = new Customer();
customer.ContactFirstName = randomString;
customer.ContactLastName = randomString;
customer.ContactPhone = randomString;
customer.CreditCardNumber = randomString;
customer.CreditCardExpirationDate = randomString;
Address address = new Address();
address.FirstName = randomString;
address.LastName = randomString;
address.Address1 = randomString;
address.Address2 = randomString;
address.City = randomString;
address.State = randomString;
address.Zip = randomString;
customer.ShippingAddress = address;
customer.BillingAddress = address;
customer.LastActivityDate = DateTime.Now;
customer.CustomerId = 0;
int customerId = ws.InsertCustomer(customer);
```
<form id="Form1" method="post">
  <p>
    <table id="Table1" cellspacing="1" cellpadding="1" width="400" border="1">
      <tr bgcolor="yellow">
        <td><strong>Test</strong></td>
        <td><strong>Attribute</strong></td>
        <td><strong>Value</strong></td>
      </tr>
      <tr>
        <td>InsertCustomer</td>
        <td>Contact First Name</td>
        <td>\%customer\Contact First Name\%</td>
      </tr>
    </table>
  </p>
</form>
Appendix I: About Doculabs

Doculabs is an independent research and consulting firm that improves the way companies plan for, select, and optimize emerging technologies through project-based services. Its clients include both the companies that purchase emerging technologies and the leading vendors that supply those technologies.

Based in Chicago and founded in 1993, Doculabs provides consulting services that are grounded in research that combines hands-on evaluation of technology with real-time business knowledge gained from engagements with Fortune 1000 clients. Doculabs helps its clients deliver on their business strategies through solutions in areas such as enterprise content management, relationship management, and infrastructure.

Doculabs leverages its research to provide deliverables customized to vendors’ go-to-market initiatives. Specifically, it offers short-term consulting projects designed to:

- Improve time to market by accelerating strategy development and product lifecycle
- Reduce product development costs by leveraging critical market information
- Increase market penetration by enhancing a competitive value proposition

In addition to fixed-bid, short-term project engagements, Doculabs channels its analysis through speaking engagements, interviews, and authored articles to a number of leading trade organizations and publications, including AIIM, BAI, DIA, NACHA, InformationWeek, CIO, Insurance & Technology, Transform, Energy Business and Technology, The Wall Street Journal, and BusinessWeek, among many others.