The success of enterprise applications developed using the Microsoft® .NET platform hinges upon a well-designed data center and a meticulously planned deployment process. This article introduces techniques for helping to ensure successful roll-outs of applications.

Deciding on a reference architecture
Application developers usually receive an explicit set of functional requirements for the software. However, several implicit requirements should not be overlooked:

- The new application should be deployed in a secure and controlled manner.
- Web services and Web applications must be consistently available.
- Web services and Web applications must be scalable as business requirements increase.

By following a standard data center configuration, organizations can quickly and efficiently build Internet applications that meet long-term business goals.

A typical enterprise Web application based on .NET consists of three logical tiers:

- **Web application tier**: Provides HTML directly to users so they can interact with the application
- **Web services tier**: Provides business logic and functional services through .NET remoting or Web services; also known as the business logic tier
- **Data tier**: Provides data storage and retrieval services, typically over Remote Procedure Call (RPC)

When rolling out an application, administrators can decide to implement it with one to three, or more, physical tiers. The reference architecture scenario described in this article uses one physical tier for each logical tier, for a total of three physical tiers. The scenario assumes that usage requirements call for a farm of four Web servers on the Web application tier, a farm of three business logic servers on the Web services tier, and a database cluster on the data tier. In addition to the servers in the data center, the system includes appropriate switching and firewall hardware to accommodate multiple virtual LANs (VLANs) and system security. Figure 1 highlights the key servers and equipment that constitute a typical system architecture for .NET applications.

An appropriate data center design is only the first step in a successful deployment. Key aspects of a .NET enterprise application roll-out include careful attention to testing, staging, distribution, and upgrades.

Planning the deployment process
The system development life cycle (SDLC) includes common activities such as designing, coding, and testing,
as well as activities sometimes given less attention, such as staging, distribution, and in-place upgrades. A well-thought-out deployment plan is critical, yet many development teams treat it as an afterthought. The following section describes what can happen with inadequate planning.

**Lack of planning: Unexpected problems**

In a common scenario, a development team uses the Microsoft Visual Studio® .NET development platform and achieves successful operation of the application in a test environment. On the day that the system is supposed to go live, the team uses xcopy deployment to move the compiled components and Web pages to their respective servers. However, settings and dependencies that were taken for granted on the development machines, and some Component Object Model (COM) dependencies for legacy support, are suddenly missing, because they were installed and configured with the developer tools.

To solve the problem, the development team hurriedly tries several solutions. The team changes some settings on the production server (but does not record which ones), copies over additional files (again, without noting the details), and finally installs Visual Studio .NET on the production server to make the application work. However, the system now has many problems: developers have access rights to the production environment, creating an internal security and stability risk; development tools and source code reside on the production servers, opening security holes; and a non-repeatable installation procedure now exists, preventing reliable upgrades.

The team in this scenario learns from its mistakes, implements a staging environment, and rolls out the application to the staging environment where it passes all tests. When moved to production, however, the application fails again because the staging servers were not set up to mirror the production environment’s load balancing and authentication techniques.

**Appropriate planning: A controlled environment**

Complex applications such as .NET-compatible enterprise solutions must be tested in a controlled environment separate from the development environment. Deployments should be staged by a roll-out manager, typically on a platform that is identical to, but separate from, the production environment. At minimum, four environments should be used (see Figure 2).

- **Development environment**
  - What: Coding and unit testing sandbox
  - Owner: Development team
  - Deploy Visual Studio .NET? Yes

- **Test/QA environment**
  - What: Integration testing environment
  - Owner: QA lead
  - Deploy Visual Studio .NET? No

- **Staging environment**
  - What: Final testing ground—mirrors production
  - Owner: Roll-out manager
  - Deploy Visual Studio .NET? No

- **Production environment**
  - What: Live system
  - Owner: Production administrator
  - Deploy Visual Studio .NET? No

Figure 2. Key environments in a .NET application deployment process
the integrated application. Roll-out to the test environment should include obtaining the latest source code from the source control server (typically Microsoft Visual SourceSafe®), packaging the installation, and running the installation program. The team should avoid installing development tools on the test servers, because these tools may introduce dependencies that could mask problems that may surface later.

**Staging environment.** Often the most neglected of the environments, the staging servers are the most important to a smooth roll-out. The staging environment provides a place to house the application after it has been fully tested in the test environment, but before rolling it out to users. The staging environment provides the last opportunity to find problems before end users see the application. For this reason, the staging environment must be virtually identical to the production environment and use the same operating system, applications, and drivers. The staging environment also should mirror the production network topology and use its lockdown procedures and policies. If the production environment uses clustering or load balancing, these techniques should be used in the staging environment first.

**Production environment.** The production environment is the live site, the actual network to which end users connect, so downtime must be minimized and changes to applications must proceed smoothly. Even minor changes can cause site failure.

**Creating the ideal staging environment**

The staging environment should resemble the production environment as closely as possible. Creating the ideal staging environment requires starting with a clean set of servers, the same type already used in production. The staging network should include the same type and number of switches, routers, firewalls, and storage hardware as well. Time and money saved by preventing deployment problems should eventually compensate for the cost of extra equipment.

Although every aspect of production should be mirrored, deployment teams can scale down the staging environment. If four Web servers are in production, two Web servers might be sufficient for the staging environment, enabling proper testing of load balancing and failover scenarios.

The staging environment should include redundant machines; otherwise, certain situations may not be replicated. For example, a production environment could include a stateless Web service that updates a multi-master, synchronized data store, such as the Microsoft Active Directory® directory service. A subsequent read by the Web service could have a problem, because another Active Directory server—one that has not yet received the data sync—may respond to the request, causing a read failure. Unless the deployment team mimics the production environment with redundant hardware, the team is unlikely to discover this problem. Figure 3 shows a staging environment that suitably mimics the production architecture in Figure 1.

Setup of the staging servers begins with a clean installation of the operating system. Then, during the installation procedure of the application, the deployment team should clearly document every step and action, no matter how small. The team should note the location of application files and all configuration settings, as well as all temporary files, registry settings, and user accounts that need to be created. The completed document should mimic every aspect of the installation process in painstaking detail, with a final section describing steps to uninstall the application.

After completing the installation process, the team should test to ensure that the application is working properly and that the installation document is complete. Then the team should uninstall all software in the staging environment, leaving a bare machine. Next, the team should install the application again, precisely following the installation document. Any action that is not reflected...
in the document, no matter how trivial, must be added to it. The team should repeat the entire process until it is certain that the production rollout will be a success.

**Deploying the .NET Framework**

When rolling out a .NET application, the deployment team may encounter missing dependencies, such as the .NET Framework, code assemblies, COM objects, and third-party components. In addition, incorrect or inconsistent application configuration or security settings might affect deployment. Of these, the most significant and most common dependencies are the common language runtime (CLR) and the .NET Framework class library.

The CLR is the foundation of the .NET Framework. The CLR manages the code and objects at run time, providing core services such as memory management, thread management, and remoting services. Code that uses these core services through .NET is known as managed code. Components that do not target the CLR, but instead call directly to the operating system application programming interfaces (APIs), are known as unmanaged code. Managed code requires the .NET Framework on the physical machine on which it runs.

The .NET Framework class library is a collection of reusable classes provided as the basis for applications. These classes provide capabilities such as graphical user interface (GUI) elements, file I/O, data access, load balancing, message queuing, and transactions. Applications that make use of the .NET Framework classes require the .NET Framework on the physical machine.

The following is a list of machines in a typical development scenario and guidelines regarding whether they require the .NET Framework:

- **End-user machine:** End users generally access a Web application through a browser. Because the server machine runs the .NET-based code and passes down plain HTML to the browser, the .NET Framework typically is not needed on the client machine. An important exception is browser-based managed controls, which are assemblies referenced by Web pages that are downloaded to the user’s computer and run locally. Browser-based managed controls are similar in concept to ActiveX® controls, except that they require the .NET Framework. When an application uses managed code, the .NET Framework must be installed on client computers.

- **Web servers:** Any .NET-based application that uses managed code must have the .NET Framework installed. Web server applications are no exception, and thus physical Web servers require the .NET Framework. When upgrading production servers, the staging servers also should be upgraded so that the next deployment will run smoothly. Administrators who follow these techniques are more likely to achieve successful application deployments.

- **Business logic servers:** Many enterprise .NET applications implement a physically separate tier to provide business logic. If these applications use the .NET Framework to provide or consume Web services, to access remote data stores, or to perform any other activity, these servers will need the .NET Framework installed.

- **Database servers:** Because database products, such as Microsoft SQL Server 2000, are native applications, they do not require the .NET Framework. This can be confusing, especially since they are often described as .NET Enterprise Servers.

The redistributable version of .NET (dotnetfx.exe) is required to deploy the .NET Framework to a server. This package is available from Microsoft and freely distributable to those with a licensed copy of Visual Studio .NET or the .NET Framework Software Development Kit (SDK). The SDK contains both the CLR and the .NET Framework classes.

Deploying the .NET Framework to older machines is not recommended. Improved performance and lower maintenance costs will likely offset the purchase price of buying new machines. Minimum requirements for existing hardware include an Intel® Pentium® processor at 133 MHz and 256 MB of RAM to run the .NET Framework on a server. For a client machine, minimum requirements are an Intel Pentium processor at 90 MHz and 96 MB of RAM. However, it is highly recommended that servers have at least an Intel Pentium 4 processor and 1 GB of RAM, and that client machines have at least 256 MB of RAM. Machines running Microsoft Windows NT® Server 4.0 must be upgraded to Microsoft Windows® 2000 Server or Windows Server 2003 to run ASP.NET.

**Achieving successful deployments**

Having an appropriate set of staging and production environments, in addition to a well-planned deployment process, is vital to a successful roll-out of .NET-based applications. Development and deployment teams should account for dependencies, configure staging servers properly, and document the deployment process meticulously. When upgrading production servers, the staging servers also should be upgraded so that the next deployment will run smoothly. Administrators who follow these techniques are more likely to achieve successful application deployments.

Scott Jamison (scott_jamison@dell.com) is a technical strategist at Dell Professional Services. He is a Microsoft Certified Solution Developer for .NET and has been building solutions using Microsoft technology for more than 10 years. Scott has a master’s degree in Computer Science from Boston University.

---

http://www.microsoft.com/net