CHAPTER 1

Introduction to Web Development

To understand web development, you have to understand the Web, and to understand the Web, you have to understand the Internet that the Web is built on. This chapter will give you a brief history of the connected world, discussing first the origins of the Internet, then the origins of the Web, and finally the technologies used by developers to build applications on the Web. It will hopefully be a fun and informative ride!

The Internet and the Birth of the Web

The Internet dates back to the early development of general communication networks. At its heart, this concept of a computer network is an infrastructure that enables computers and their users to communicate with each other. There have been many types of computer networks over time, but one has grown to near ubiquity: the Internet.

Its history dates back to the 1960s and the development of networks to support the Department of Defense as well as various academic institutions. Interoperability of these different networks was a problem. In 1973, Robert E. Kahn of United States Defense Advanced Research Projects Agency (DARPA and ARPANET) and Vinton Cerf of Stanford University worked out a common "internetwork protocol" that they called the TCP/IP Internet Protocol Suite. This was both a set of standards that defined how computers would communicate as well as conventions for how the computers should be named and addressed, and thus how traffic would be routed between them.

At its core, TCP/IP follows most of the OSI (Open Systems Interconnection) model, which defines a network as an entity of seven layers:

Application layer: Provides the user interface (UI) to the network as well as the application services required by this interface, such as file access. In terms of the Internet, these application services are those typically provided by the browser, giving access to the file system to save favorites, print, and more.

Presentation layer: Translates the data from the network into something that the user can understand and vice versa, translating user input into the language used by the network to communicate. For example, when you use a browser to access a page, you type the address of that page. This address gets translated for you into an `HTTP-GET` command by the browser.

Session layer: Used to establish communication between applications running on different nodes of the network. For example, your request from the browser sets up a session that is used to communicate with the server that serves up the page. The lower levels are used to discover that server and facilitate the flow, but at this level you have a communication session storing all the data needed to manage the flow of data to the presentation tier for representation on the client.

Transport layer: Handles the task of managing message delivery and flow between nodes on the network. Networks are fundamentally unreliable because packets of data can be lost or received out of sync. Thus, a network protocol has to ensure delivery in a timely manner or trigger an action upon nondelivery. It also works to ensure that the messages are routed, assembled, and delivered correctly. For example, when using an Internet browsing session, your message is broken down into small packets as part of the TCP/IP protocol. The TCP/IP stack manages sending these packets to the server and assembling them correctly once they reach it.

Network layer: Used to standardize the way that addressing is accomplished between different linked networks. For data to flow from A to B, the locations and paths between A and B need to be known. This address awareness and routing is achieved by the network layer. There is almost never a direct connection between a client and a server. Instead, the traffic has to be routed across a number of networks through connections that involve changes of addresses between them. For example, a server might have the Internet protocol (IP) address 192.168.0.1 on its internal network, but that internal network faces the world using a different IP. A client calling the server calls the external IP, and has traffic on the web port routed to 192.168.0.1. This is all handled by the network layer.

Data link layer: Defines how the physical layer is accessed—that is, what protocols are used to talk to it, how the data is broken up into packets and frames, and how the addressing is managed on the physical layer. In the example just described, TCP/IP is the protocol. This is used on the network layer and above to manage communication between the client and the server. At this layer, the frames and routing information for TCP/IP packets are defined as to how they will run on the line as electrical signals. Network bridges operate at this layer.

Physical layer: Defines the type of medium, the transmission method, and the rates available for the network. Some networks have broadband communication, measured in megabits per second, whereas others have narrower communication in the kilobit range or less. Because of the wide variance in bandwidth, different behavior can be expected, and applications that implement network protocols have to be aware of this. Thus, the physical layer has to be able to provide this data to the next layer up, and respond to command instructions from it.

You can see this seven-layer model, and how typical TCP/IP applications such as the web browser fit into it, in Figure 1-1.

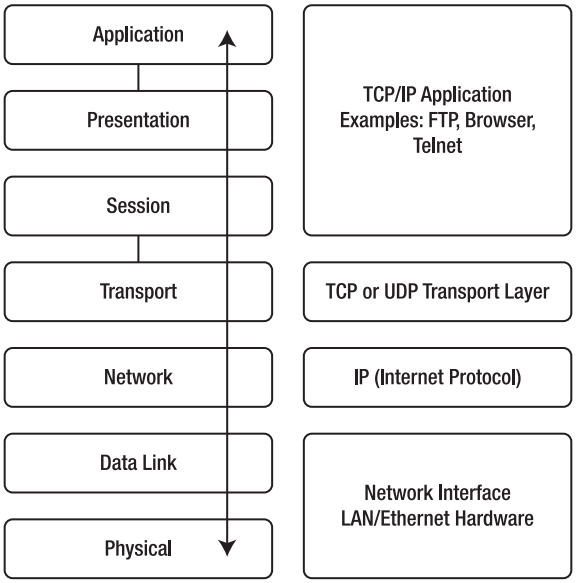

Figure 1-1. *OSI seven-layer model*

This model provided the foundation for what would become the Internet, and the Internet provided the foundation for what would become the World Wide Web.

By using TCP/IP, it became possible to build first a file transfer protocol (FTP) and ensuing application. From there, Tim Berners-Lee expanded the idea to having a "live" view of the file in an application called a "browser." Instead of just transferring a document from a distant machine to your own, the application would also render the file. To do this, the file would need to be marked up in a special way that could be understood by the browser. A natural progression to this is to allow documents to be linked to each other, so that when the user selects a link, they are taken to the document that the link refers to. This introduced the concept of *hypertext,* and thus HTML (Hypertext Markup Language) was invented.

An *HTML document* is a text document containing special markup that provides instructions to the browser for how to render the document. *Tags* such as `<H1>` and `<H2>` are used to provide styling information so that the document can be viewed as more than just text. Links to other documents are provided via another tag, `<a>` (for anchor), where a piece of text is defined as linking to another document, and the browser renders it differently (usually with a blue underline).

Once HTML was developed, the Web grew rapidly. Thanks to TCP/IP and the Internet, people could put documents on any server in the world and provide an address, called a *URL* (Universal Resource Locator), which could enable these documents to be found. These URLs could then be embedded as links in other documents and used to find those documents. Quickly, a huge network of interconnected, browsable documents was created: the World Wide Web.

Going Beyond the Static Web

This Web—a network of linked documents—was very useful, but in essence very static. Consider the scenario of a store wanting to provide links to potential customers of their current products. Their inventory changes rapidly, and static documents require people who understand the inventory and can constantly generate documents containing new details. Every time something is bought or sold by the store, these documents need to be updated. This, as you can imagine, is a time-consuming, difficult, and non-cost-effective task!

We needed some way to automatically generate documents instead of creating them manually. Also, these documents needed to be generated not in overnight batch runs, but rather upon request so that the information would always be up-to-date.

Thus, the "active" Web was born. New servers were written on the Common Gateway Interface (CGI) standard, which allowed developers to write code (usually in C) that executed in response to user requests. When a request came in for a document, this code could run, and in the case of our store scenario, that code could read a database or an inventory system for the current status and generate the results as an HTML document. This document would then be sent back to the browser. This system worked well, and was very powerful and widely used.

Maintenance of CGI applications became quite difficult, however, and CGI applications were also platform-specific, so if you had a cluster of servers, some of which were based on different technologies and/or versions of operating systems, you could end up with multiple versions of the same program to support! So, for example, if you wanted to run the same program on your cluster, but had different versions of an operating system, your code would have to be tailored for each machine.

But when there is a problem, there is also opportunity. And where there is opportunity, there is innovation. One of these opportunities was moving toward a *managed cross-platform code* approach in which a high-level language such as Java could be used

to build an application that generates dynamic pages. Because Java is a cross-platform language, the platform on which the code ran no longer mattered, and server-side Java, called *servlets*, became an effective replacement for CGI.

But the problem of generating HTML still existed. In these applications, string management, or `printf` statements, were used to write HTML, leading to ugly and onerous code. In another approach, HTML defined the output, and special extension tags instructed the server to do something when it reached those tags and fill in the placeholders. The code would look like the pseudocode here:

```
<h3>We have <% nQuantity %> widgets in stock. </h3>
```

The `<% %>` contains code that will be executed by the server. In this case, it is a value calculated on the fly as part of the session. When it's evaluated, the result is injected into the HTML and returned to the browser.

This is the underpinning of technologies such as classic ASP, which runs on Internet Information Server (IIS) and uses a Microsoft Visual Basic–like language between the tags. A similar architecture is used by Personal Hypertext Processor (PHP), which runs on its own interpreter that can be an IIS or other web server extension, and uses its own C++-like language. There are many other examples as well, such as Java Server Pages (JSP), which uses an approach where the HTML is not written out using code, but instead contains tags that are interpreted and replaced at runtime with calculated values; or Ruby, an object-oriented scripting language that works well for generating web content.

The opportunity was there for a best-of-both-worlds approach. And here is where ASP.NET arrived to fill the gap.

The Arrival of ASP.NET

ASP.NET was the result of developers and architects sitting back and thinking about the direction in which web development had taken to date. In some ways, developers had been painted into a corner by rapid innovation and were now in a nonoptimal development environment.

ASP.NET was designed to get around a number of issues regarding how to develop web applications at the time. At the same time, it began spurring innovation of new types of applications that previously might not have been possible.

First, it was designed to be a code-friendly environment using sophisticated object-oriented methodology that allowed for rapid code development and reuse, as opposed to the scripting-like environment used previously.

Second, it was designed to be a multiple-language single runtime, allowing developers from different backgrounds to use it with minimal retraining. For the Visual Basic folk, Visual Basic .NET was available, and for those used to more traditional object-oriented languages such as C++ or Java, a new language—C#—was introduced.

Third, the concept of *web services* was identified as being vital for the future of the Web, because they are a device-agnostic, technology-agnostic means for sharing data across the multi-platform Internet. ASP.NET was designed to make the complicated process of creating, exposing, and consuming web services as simple as possible.

Finally, performance of the Web depends not only on the speed of the network, but also on the speed of the application serving you. Absolute performance, defined as the overall speed of the application, is difficult enough, but application performance under different user loads implemented concurrently across multiple servers is more of a trick. ASP.NET was designed with optimizations for this in mind, including a compiled code model, where all the source code is turned into native machine language ahead of time, instead of an interpreted one, where all the source code is turned into native machine language step by step as it executes. It also includes a scalable data access mode, a way to keep state between client and server, data caching, and much more.

Summary

This chapter has given you a very brief background on what the Internet is, how the Web fits into the Internet, and how web application development has evolved to this point. It has also introduced you to the ASP.NET technology.

In this book, you'll look at ASP.NET in the .NET Framework and how it is used to build the web applications and services of today and tomorrow. In Part 1, you'll learn about the framework for building traditional web applications. Then, in Part 2, you'll move on to looking at how innovations for technologies such as Ajax and Windows Presentation Foundation (WPF) allow you to start improving the overall user experience. You'll also look at the development frameworks of tomorrow to learn how you can take the Web into the next phase of its evolution—that is, toward the next-generation Web, where the user experience is at the heart of everything you do.