WEB APPLICATION DEVELOPMENT

MODULE 1: INTRODUCTION

1.1 The WEB PERSPECTIVE

A little more than a decade ago at CERN (European Organization for Nuclear Research - the scientific research laboratory near Geneva, Switzerland), Tim Berners-Lee presented a proposal for an information management system that would enable the sharing of knowledge and resources over a computer network.

The system he proposed has propagated itself into what can truly be called a *World Wide Web*, as people all over the world use it for a wide variety of purposes:

- Educational institutions and research laboratories were among the very first users of the Web, employing it for sharing documents and other resources across the Internet.
- Individuals today use the Web (and the underlying Internet technologies that support it) as an instantaneous international postal service, as a worldwide community bulletin board for posting virtual photo albums, and as a venue for holding global yard sales.
- Businesses engage in *e-commerce*, offering individuals a medium for buying and selling goods and services over the net. They also communicate with other businesses through *B2B* (business-to-business) data exchanges, where companies can provide product catalogues, inventories, and sales records to other companies.

1.2 THE ORIGINS OF THE WEB

Tim Berners-Lee originally promoted theWorldWideWeb as a virtual library, a document control system for sharing information resources among researchers. Online documents could be accessed via a unique document address, a *Universal Resource Locator* (URL). These documents could be cross-referenced via *hypertext links*.

From the very beginnings of Internet technology, there has been a dream of using the Internet as a universal medium for exchanging information over computer networks. Many people shared this dream. Ted Nelson's *Xanadu* project aspired to make that dream a reality, but the goals were lofty and were never fully realized.

Internet file sharing services (such as *FTP* and *Gopher*) and message forum services (such as *Netnews*) provided increasingly powerful mechanisms for this sort of information exchange, and certainly brought us closer to fulfilling those goals.

However, it took Tim Berners-Lee to (in his own words) "marry together" the notion of hypertext with the power of the Internet, bringing those initial dreams to fruition in a way that the earliest developers of both hypertext and Internet technology might never have imagined. His vision was to connect literally *everything* together, in a uniform and universal way.

1.3 FROM WEB PAGES TO WEB SITES

The explosively exponential growth of the Web can at least partially be attributed to its grass roots proliferation as a tool for personal publishing. The fundamental technology behind the Web is relatively simple. A computer connected to the Internet, running a Web server, was all that was necessary to serve documents. Both CERN and the National Center for Supercomputer Applications (NCSA) at the University of Illinois had developed freely available Web server software. A small amount of HTML knowledge (and the proper computing resources) got you something that could be called a Web site.

When the Web was in its infancy, the only computers connected to the Internet and capable of running server software were run by academic institutions and well-connected technology companies. Smaller computers, in any case, were hardly in abundance back then. In those days, a 'personal' computer sitting on your desktop was still a rarity. If you wanted access to any sort of computing power, you used a terminal that let you 'log in' to a large server or mainframe over a direct connection or dialup phone line. Still, among those associated with such organizations, it quickly became a very simple process to create your own Web pages. Moreover, all that was needed was a simple text editor. The original HTML language was simple enough that, even without the more sophisticated tools we have at our disposal today, it was an easy task for someone to create a Web page.

There is a big difference between a Web page and a Web site. A Web site is more than just a group of Web pages that happen to be connected to each other through hypertext links. At the lowest level, there are content-related concerns. Maintaining thematic consistency of content is important in giving a site some degree of identity. There are also aesthetic concerns. In addition to having thematically-related content, a Web site should also have a common look and feel across all of its pages, so that site visitors know they are looking at a particular Web site. This means utilizing a common style across the site: page layout, graphic design, and typographical elements should reflect that style. There are also architectural concerns. As a site grows in size and becomes more complex, it becomes critically important to organize its content properly. This includes not just the layout of content on individual pages, but also the interconnections between the pages themselves. Some of the symptoms of bad site design include links targeting the wrong frame (for frame-based Web sites), and links that take visitors to a particular page at an appropriate time (e.g. at a point during the visit when it is impossible to deliver content to the visitors). If your site becomes so complex that visitors cannot navigate their way through it, even with the help of site maps and navigation bars, then it needs to be reorganized and restructured.

1.4 FROM WEB SITES TO WEB APPLICATIONS

Initially, what people shared over the Internet consisted mostly of static information found in files. They might edit these files and update their content, but there were few truly dynamic information services on the Internet. Granted, there were a few exceptions: search applications for finding files found on FTP archives and Gopher servers; and services that provided dynamic information directly, like the weather, or the availability of cans from a soda dispensing machine. (One of the first Web applications that Tim Berners-Lee demonstrated at CERN was a gateway for looking up numbers from a phone book database using a Web browser.)
However, for the most part the information resources shared on the Web were static documents. Dynamic information services—from search engines to CGI scripts to packages that connected the Web to relational databases—changed all that. With the advent of the *dynamic web*, the bar was raised even higher. No longer was it sufficient to say that you were designing a 'Web site' (as opposed to a motley collection of 'Web pages'). It became necessary to design a *Web application*.

1.4.1 Definition of a Web Application

What is a 'Web application?' By definition, it is something more than just a 'Web site.' It is a *client/server* application that uses a Web browser as its client program, and performs an interactive service by connecting with servers over the Internet (or Intranet).
A Web site simply delivers content from static files. A Web application presents dynamically tailored content based on request parameters, tracked user behaviours, and security considerations.

1.5 Principles of web application design

What do we mean when we discuss the *general principles* that need to be understood to properly design and develop Web applications?

We mean the core set of protocols and languages associated with Web applications. This includes, of course, *HTTP* (*HyperText Transfer Protocol*) and *HTML* (*HyperText Markup Language*), which are fundamental to the creation and transmission of Web pages. It also includes the older Internet protocols like *Telnet* and *FTP*, protocols used for message transfer like *SMTP* and *IMAP*, plus advanced protocols and languages like *XML*. Additionally, it includes knowledge of *databases* and *multimedia* presentation, since many sophisticated Web applications make use of these technologies extensively.

The ideal *Web application architect* must in some sense be a 'jack of all trades'. People who design Web applications must understand not only HTTP and HTML, but the other underlying Internet protocols as well. They must be familiar with JavaScript, XML, relational databases, graphic design and multimedia. They must be well versed in application server technology, and have a strong background in information architecture. If you find people with all these qualifications, please let us know—we would love to hire them! Rare is the person who can not only architect a Web site, but also design the graphics, create the database schema, produce the multimedia programs, and configure the e-commerce transactions.

MODULE 2: BEFORE THE WEB: TCP/IP

In this chapter, we examine the core Internet protocols that make up the *TCP/IP protocol suite*, which is the foundation for Web protocols, discussed in the next chapter. We begin with a brief historical overview of the forces that led to the creation of TCP/IP. We then go over the layers of the TCP/IP stack, and show where various protocols fit into it. Our description of the client-server paradigm used by TCP/IP applications is followed by a discussion of the various TCP/IP application services, including Telnet, electronic mail, message forums, live messaging, and file servers.

2.1 HISTORICAL PERSPECTIVE

The roots of Web technology can be found in the original Internet protocols (known collectively as TCP/IP), developed in the 1980s. These protocols were an outgrowth of work done for the United States Defence Department to design a network called the *ARPANET*.

The ARPANET was named for *ARPA*, the *Advanced Research Projects Agency* of the United States Department of Defence. It came into being as a result of efforts funded by the Department of Defence in the 1970s to develop an open, common, distributed, and decentralized computer networking architecture. There were a number of problems with existing network architectures that the Defence Department wanted to resolve. First and foremost was the centralized nature of existing networks. At that time, the typical network topology was *centralized*. A computer network had a single point of control directing communication between all the systems belonging to that network. From a military perspective, such a topology had a critical flaw: Destroy that central point of control, and all possibility of communication was lost.

Another issue was the *proprietary* nature of existing network architectures. Most were developed and controlled by private corporations, who had a vested interest both in pushing their own products and in keeping their technology to themselves.

Further, the proprietary nature of the technology limited the interoperability between different systems. It was important, even then, to ensure that the mechanisms for communicating across computer networks were not proprietary, or controlled in any way by private interests,

lest the entire network become dependent on the whims of a single corporation. Thus, the Defence Department funded an endeavour to design the protocols for the next generation of computer communications networking architectures.
Establishing a *decentralized, distributed network topology* was foremost among the design goals for the new networking architecture. Such a topology would allow communications to continue, for the most part undisrupted, even if anyone system was damaged or destroyed. In such a topology, the network 'intelligence' would not reside in a single point of control. Instead, it would be distributed among many systems throughout the network.
To facilitate this (and to accommodate other network reliability considerations), they employed a *packet-switching* technology, whereby a network 'message' could be split into packets, each of which might take a different route over the network, arrive in completely mixed-up order, and still be reassembled and understood by the intended recipient.
To promote *interoperability*, the protocols needed to be *open*: be readily available to anyone who wanted to connect their system to the network. An infrastructure was needed to design the set of agreed-upon protocols, and to formulate new protocols for new technologies that might be added to the network in the future.
An *Internet Working Group* (INWG) was formed to examine the issues associated with connecting heterogeneous networks together in an open, uniform manner. This group provided an open platform for proposing, debating, and approving protocols.
The Internet Working Group evolved over time into other bodies, like the IAB (Internet Activities Board, later renamed the Internet Architecture Board), the IANA (Internet Assigned Numbers Authority), and later, the IETF (Internet Engineering Task Force) and IESG (Internet Engineering Steering Group). These bodies defined the standards that 'govern' the Internet. They established the formal processes for proposing new protocols, discussing and debating the merits of these proposals, and ultimately approving them as accepted Internet standards.
Proposals for new protocols (or updated versions of existing protocols) are provided in the form of *Requests for Comments*, also known as RFCs. Once approved, the RFCs are treated as the standard documentation for the new or updated protocol.

2.2 TCP/IP

The original ARPANET was the first fruit borne of this endeavor. The protocols behind the ARPANET evolved over time into the *TCP/IP Protocol Suite*, a layered taxonomy of data communications protocols. The name TCP/IP refers to two of the most important protocols within the suite: TCP (*Transmission Control Protocol*) and IP (*Internet Protocol)*, but the suite is comprised of many other significant protocols and services.

2.2.1 Layers

The protocol layers associated with TCP/IP (above the 'layer' of physical interconnection) are:
1. Network Interface layer,
2. Internet layer,
3. Transport layer, and
4. Application layer.

Because this protocol taxonomy contains layers, implementations of these protocols are often known as a protocol stack.

1. The Network Interface layer is the layer responsible for the lowest level of data transmission within TCP/IP, facilitating communication with the underlying physical network.
2. The Internet layer provides the mechanisms for intersystem communications, controlling message routing, validity checking, and message header composition/decomposition.

The protocol known as IP (which stands for Internet Protocol) operates on this layer, as does ICMP (the Internet Control Message Protocol). ICMP handles the transmission of control and error messages between systems. Ping is an Internet service that operates through ICMP.

3. The Transport layer provides message transport services between applications running on remote systems. This is the layer in which TCP (the Transmission Control Protocol) operates. TCP provides reliable, connection-oriented message transport. Most of the well-known Internet services make use of TCP as their foundation. However, some services that do not require the reliability (and overhead) associated with TCP and therefore make use of UDP (which stands for User Datagram Protocol). For instance, streaming audio and video services would gladly sacrifice a few lost packets to get faster performance out of their data streams, so these services often operate over UDP, which trades reliability for performance.
4. The Application layer is the highest level within the TCP/IP protocol stack. It is within this layer that most of the services we associate with 'the Internet' operate.

2.3 OSI

On the other hand, during the period that TCP/IP was being developed, the International Standards Organization (ISO) was also working on a layered protocol scheme, called 'Open Systems Interconnection', or OSI. While the TCP/IP taxonomy consisted of five layers (if you included the lowest physical connectivity medium as a layer), OSI had seven layers: Physical, Data Link, Network, Transport, Session, Presentation, and Application.
There is some parallelism between the two models. TCP/IP's Network Interface layer is sometimes called the Data Link layer to mimic the OSI Reference Model, while the Internet layer corresponds to OSI's Network layer. Both models share the notion of a Transport layer, which serves roughly the same functions in each model. And the Application layer in TCP/IP combines the functions of the Session, Presentation, and Application layers of OSI. But OSI never caught on, and while some people waited patiently for its adoption and propagation, it was TCP/IP that became the ubiquitous foundation of the Internet as we know it today.

2.2.2 The client/server paradigm

TCP/IP applications tend to operate according to the client/server paradigm. This simply means that, in these applications, servers (also called services and daemons, depending on the language of the underlying operating system) execute by (1) waiting for requests from client programs to arrive, and then (2) processing those requests.
Client programs can be applications used by human beings, or they could be servers that need to make their own requests that can only be fulfilled by other servers. More often than not, the client and server the client and server run on separate machines, and communicate via a connection across a network.

Early implementations of client/server architectures did not make use of open protocols. What this meant was that client programs needed to be as 'heavy' as the server programs. A 'lightweight' client (also called a *thin client*) could only exist in a framework where common protocols and application controls were associated with the client machine's operating system. Without such a framework, many of the connectivity features had to be included directly into the client program, adding to its weight.
One advantage of using TCP/IP for client/server applications was that the protocol stack was installed on the client machine as part of the operating system, and the client program itself could be more of a thin client.

Web applications are a prime example of the employment of thin clients in applications. Rather than building a custom program to perform desired application tasks, web applications use the web browser, a program that is already installed on most users' systems. You cannot create a client much thinner than a program that users have already installed on their desktops!

Client/Server Communications

Requests transmitted between client and server programs take the form of command line interactions. The imposition of this constraint on Internet communication protocols means that even the most primitive command-line oriented interface can make use of TCP/IP services. More sophisticated GUI-based client programs often hide their command-line details from their users, employing point-and-click and drag-and-drop functionality to support underlying command-line directives.

After the server acknowledges the success of the connection, the client sends commands on a line-by-line basis. There are single-line and block commands. A block command begins with a line indicating the start of the command (e.g., a line containing only the word 'DATA') and terminates with a line indicating its end (e.g., a line containing only a period). The server then responds to each command, usually with a line containing a response code.

A stateful protocol allows a request to contain a sequence of commands. The server is required to maintain the "state" of the connection throughout the transmission of successive commands, until the connection is terminated. The sequence of transmitted and executed commands is often called a session. Most Internet services (including SMTP) are session-based, and make use of stateful protocols.

HTTP, however, is a stateless protocol. An HTTP request usually consists of a single block command and a single response. On the surface, there is no need to maintain state between transmitted commands. We will discuss the stateless nature of the HTTP protocol in a later chapter.

Example of How TCP/IP Clients and Servers Communicate with Each Other?

To talk to servers, TCP/IP client programs open a socket, which is simply a TCP connection between the client machine and the server machine. Servers listen for connection requests that come in through specific ports. A port is not an actual physical interface between the computer and the network, but simply a numeric reference within a request that indicates which server program is its intended recipient.

There are established conventions for matching port numbers with specific TCP/IP services. Servers listen for requests on well-known port numbers. For example, Telnet servers normally listen for connection requests on port 23, SMTP servers listen to port 25, and web servers listen to port 80.

2.4 TCP/IP APPLICATION SERVICES

In this section, we discuss some of the common TCP/IP application services, including Telnet, electronic mail, message forums, live messaging, and file servers.

2.4.1 Telnet

The Telnet protocol operates within the Application layer. It was developed to support Network Virtual Terminal functionality, which means the ability to 'log in' to a remote machine over the Internet. The latest specification for the Telnet protocol is defined in Internet RFC 854.

Remember that before the advent of personal computers, access to computing power was limited to those who could connect to a larger server or mainframe computer, either through a phone dialup line or through a direct local connection.
Whether you phoned in remotely or sat down at a terminal directly connected to the server, you used a command-line interface to log in. You connected to a single system and your interactions were limited to that system.
With the arrival of Internet services, you could use the Telnet protocol to log in remotely to other systems that were accessible over the Internet. As mentioned earlier, Telnet clients are configured by default to connect to port 23 on the server machine, but the target port number can be over-ridden in most client programs.
This means you can use a Telnet client program to connect and ‘talk’ to *any* TCP server by knowing its address and its port number.

2.4.2 Electronic mail

Electronic mail, or *e-mail*, was probably the first ‘killer app’ in what we now call cyberspace. Since the net had its roots in military interests, naturally the tone of electronic mail started out being formal, rigid, and business-like. But once the body of people using e-mail expanded, and once these people realized what it could be used for, things lightened up quite a bit.
Electronic *mailing lists* provided communities where people with like interests could exchange messages. These lists were closed systems, in the sense that only subscribers could post messages to the list, or view messages posted by other subscribers.
Obviously, lists grew, and list managers had to maintain them. Over time, automated mechanisms were developed to allow people to subscribe (and, just as importantly, to unsubscribe) without human intervention. These mailing lists evolved into *message forums*, where people could publicly post messages, on an *electronic bulletin board*, for everyone to read.
These services certainly existed before there was an Internet. Yet in those days, users read and sent their e-mail by *logging in* to a system directly (usually via telephone dialup or direct local connection) and running programs on that system (usually with a command-line interface) to access e-mail services. The methods for using these services varied greatly from system to system, and e-mail connectivity between disparate systems was hard to come by. With the advent of TCP/IP, the mechanisms for providing these services became more consistent, and e-mail became uniform and ubiquitous.
The transmission of electronic mail is performed through the SMTP protocol. The reading of electronic mail is usually performed through either POP or IMAP.

SMTP

SMTP stands for *Simple Mail Transfer Protocol*. As an application layer protocol, SMTP normally runs on top of TCP, though it can theoretically use any underlying transport protocol.
The application called ‘sendmail’ is an implementation of the SMTP protocol for UNIX systems.
The latest specification for the SMTP protocol is defined in *Internet RFC 821*, and the structure of SMTP messages is defined in
Internet RFC 822.
SMTP, like other TCP/IP services, runs as a *server, service*, or *daemon*. In a TCP/IP environment, SMTP servers usually run on port 25. They wait for requests to send electronic mail messages, which can come from local system users or from across the network. They are also responsible for evaluating the recipient addresses found in e-mail messages and determining whether they are valid, and/or whether their final destination is another recipient (e.g. a forwarding address, or the set of individual recipients subscribed to a mailing list).

If the message embedded in the request is intended for a user with an account on the local system, then the SMTP server will deliver the message to that user by appending it to their *mailbox*. Depending on the implementation, the mailbox can be anything from a simple text file to a complex database of e-mail messages. If the message is intended for a user on another system, then the server must figure out how to transmit the message to the appropriate system.
This may involve direct connection to the remote system, or it may involve connection to a *gateway* system. A gateway is responsible for passing the message on to other gateways and/or sending it directly to its ultimate destination. Before the advent of SMTP, the underlying mechanisms for sending mail varied from system to system. Once SMTP became ubiquitous as the mechanism for electronic mail transmission, these mechanisms became more uniform.
The applications responsible for transmitting e-mail messages, such as SMTP servers, are known as MTAs (Mail Transfer Agents). Likewise, the applications responsible for retrieving messages from a mailbox, including POP servers and IMAP servers, are known as MRAs (Mail Retrieval Agents).
E-mail client programs have generally been engineered to allow users to both read mail and send mail. Such programs are known as MUAs (Mail User Agents).
MUAs talk to MRAs to read mail, and to MTAs to send mail. In a typical e-mail client, this is the process by which a message is sent. Once the user has composed a message, the client program directs it to the SMTP server. First, it must *connect* to the server. It does this by opening a TCP socket to port 25 (the SMTP port) of the server. (This is true even if the server is running on the user’s machine.)

Originally, SMTP servers executed in a very open fashion: anyone knowing the address of an SMTP server could connect to it and send messages. In an effort to discourage spamming (the sending of indiscriminate mass e-mails in a semi-anonymous fashion), many SMTP server implementations allow the system administrator to configure the server so that it only accepts connections from a discrete set of systems, perhaps only those within their local domain.
When building web applications that include e-mail functionality (specifically the *sending* of e-mail), make sure your configuration includes the specification of a working SMTP server system, which will accept your requests to transmit messages.
To maximize application flexibility, the address of the SMTP server should be a parameter that can be modified at run-time by an application administrator.

Example 1: Command line interaction with an SMTP server

```
220 mail.hoboken.company.com ESMTP xxxx 3.21 #1 Fri, 23 Feb 2001
13:41:09 -0500
HELO ubizmo.com
250 mail.hoboken.company.com Hello neurozen.com [xxx.xxx.xxx.xxx]
MAIL FROM:<rrosen@neurozen.comt>
250 <rrosen@neurozen.com> is syntactically correct
RCPT TO:<shklar@cs.rutgers.edu>
250 <shklar@cs.rutgers.edu> is syntactically correct
RCPT TO:<rr-booknotes@neurozen.com>
250 <rr-booknotes@neurozen.com> is syntactically correct
DATA
354 Enter message, ending with "." on a line by itself
From: Rich Rosen <rrosen@neurozen.com>
To: shklar@cs.rutgers.edu
Cc: rr-booknotes@neurozen.com
```

```
Subject: Demonstrating SMTP
Leon,
Please ignore this note. I am demonstrating the art of connecting to
an SMTP server for the book. :-)
Rich
.
250 OK id=xxxxxxxx
QUIT
```

As shown in example 1, the client program identifies itself (and the system on which it is running) to the server via the ‘HELO’ command. The server decides (based on this identification information) whether to accept or reject the request. If the server accepts the request, it waits for the client to send further information.
One line at a time, the client transmits commands to the server, sending information about the originator of the message (using the ‘MAIL’ command) and each of the recipients (using a series of ‘RCPT’ commands). Once all this is done, the client tells the server it is about to send the actual data: the message itself. It does this by sending a command line consisting of only the word ‘DATA’. Every line that follows, until the server encounters a line containing only a period, is considered part of the message body. Once it has sent the body of the message, the client signals the server that it is done, and the server transmits the message to its destination (either directly or through gateways).
Having received confirmation that the server has transmitted the message, the client closes the socket connection using the ‘QUIT’ command.

MIME

Originally, e-mail systems transmitted messages in the form of standard ASCII text. If a user wanted to send a file in a non-text or ‘binary’ format (e.g. an image or sound file), it had to be encoded before it could be placed into the body of the message. The sender had to communicate the nature of the binary data directly to the receiver, e.g., ‘The block of encoded binary text below is a GIF image.’
Multimedia Internet Mail Extensions (MIME) provided uniform mechanisms for including encoded attachments within a multipart e-mail message. MIME supports the definition of boundaries separating the text portion of a message (the ‘body’) from its attachments, as well as the designation of attachment encoding methods, including ‘Base64’ and ‘quoted-printable’. MIME was originally defined in Internet RFC 1341, but the most recent specifications can be found in Internet RFCs 2045 through 2049.
It also supports the notion of content typing for attachments (and for the body of a message as well). MIME-types are standard naming conventions for defining what type of data is contained in an attachment. A MIME-type is constructed as a combination of a top-level data type and a subtype. There is a fixed set of top-level data types, including ‘text’, ‘image’, ‘audio’, ‘video’, and ‘application’. The subtypes describe the specific type of data, e.g. ‘text/html’, ‘text/plain’, ‘image/jpeg’, ‘audio/mp3’. The use of MIME content typing is discussed in greater detail in a later chapter.

POP

POP, the *Post Office Protocol*, gives users direct access to their e-mail messages stored on remote systems. *POP3* is the most recent version of the POP protocol. Most of the popular e-mail clients (including Eudora, Microsoft Outlook, and Netscape Messenger) use POP3 to access user e-mail. (Even proprietary systems like Lotus Notes offer administrators the option

to configure remote e-mail access through POP.) POP3 was first defined in *Internet RFC 1725*, but was revised in *Internet RFC 1939*.
Before the Internet, as mentioned in the previous section, people read and sent e-mail by logging in to a system and running command-line programs to access their mail. User messages were usually stored locally in a mailbox file on that system.
Even with the advent of Internet technology, many people continued to access email by Telnetting to the system containing their mailbox and running command-line programs (e.g. from a UNIX shell) to read and send mail. (Many people who prefer command-line programs still do!)
Let us look at the process by which POP clients communicate with POP servers to provide user access to e-mail. First, the POP client must connect to the POP server (which usually runs on port 110), so it can identify and authenticate the user to the server. This is usually done by sending the user 'id' and password one line at a time, using the 'USER' and 'PASS' commands. (Sophisticated POP servers may make use of the 'APOP' command, which allows the secure transmission of the user name and password as a single encrypted entity across the network.)
Once connected and authenticated, the POP protocol offers the client a variety of commands it can execute. Among them is the 'UIDL' command, which responds with an ordered list of message numbers, where each entry is followed by a unique message identifier. POP clients can use this list (and the unique identifiers it contains) to determine which messages in the list qualify as 'new' (i.e. not yet seen by the user through this particular client).
Having obtained this list, the client can execute the command to retrieve a message ('RETR *n*'). It can also execute commands to delete a message from the server ('DELE *n*'). It also has the option to execute commands to retrieve just the header of a message ('TOP *n* 0').
Message headers contain *metadata* about a message, such as the addresses of its originator and recipients, its subject, etc. Each message contains a message header block containing a series of lines, followed by a blank line indicating the end of the message header block.
The information that e-mail clients include in message lists (e.g. the 'From', 'To', and 'Subject' of each message) comes from the message headers. As e-mail technology advanced, headers began representing more sophisticated information, including MIME-related data (e.g. content types) and attachment encoding schemes.

Example 2: command-line interaction between a client and a POP server

```
From: Rich Rosen <rr-booknotes@neurozen.com>
To: Leon Shklar <shklar@cs.rutgers.edu>
Subject: Here is a message. . .
Date: Fri, 23 Feb 2001 12:58:21 -0500
Message-ID: <G987W90B.D43@neurozen.com>
```

The example above provides a simple command-line interaction between a client and a POP server. As mentioned previously, GUI-based clients often hide the mundane command line details from their users. The normal sequence of operation for most GUI-based POP clients today is as follows:
1. Get the user id and password (client may already have this information, or may need to prompt the user).
2. Connect the user and verify identity.
3. Obtain the UIDL list of messages.
4. Compare the identifiers in this list to a list that the client keeps locally, to determine which messages are 'new'.

5. Retrieve all the new messages and present them to the user in a selection list.
6. Delete the newly retrieved messages from the POP server (optional).

Example 3: command line interaction with a POP3 server

```
+OK mail Server POP3 v1.8.22 server ready
user shklar
+OK Name is a valid mailbox
pass xxxxxx
+OK Maildrop locked and ready
uidl
+OK unique-id listing follows
1 2412
2 2413
3 2414
4 2415
.
retr 1
+OK Message follows
From: Rich Rosen <waa-booknotes@neurozen.com>
To: Leon Shklar <shklar@cs.havers.edu>
Subject: Here is a message...
Date: Fri, 23 Feb 2001 12:58:21-0500
Message-ID: <G987W90B.D43@neurozen.com>
The medium is the message.
--Marshall McLuhan, while standing behind a placard
in a theater lobby in a Woody Allen movie.
```

Users must wait for *all* of the messages (include the large, possibly unwanted ones) to download before viewing any of the messages they *want* to read. It would be more efficient for the client to retrieve *only* the message headers and display the header information about each message in a message list. It could then allow users the option to selectively download desired messages for viewing, or to delete unwanted messages without downloading them. A web-based e-mail client could remove some of this inefficiency. (We discuss the construction of a web-based e-mail client in a later chapter.)

IMAP

Some of these inefficiencies can be alleviated by the *Internet Message Access Protocol (IMAP)*. IMAP was intended as a successor to the POP protocol, offering sophisticated services for managing messages in remote mailboxes. IMAP servers provide support for multiple remote *mailboxes* or *folders*, so users can move messages from an incoming folder (the ‘inbox’) into other folders kept on the server. In addition, they also provide support for saving sent messages in one of these remote folders, and for multiple simultaneous operations on mailboxes.

IMAP4, the most recent version of the IMAP protocol, was originally defined in *Internet RFC 1730*, but the most recent specification can be found in *Internet RFC 2060*.
The IMAP approach differs in many ways from the POP approach. In general, POP clients are supposed to download e-mail messages from the server and then delete them. (This is the default behavior for many POP clients.) In practice, many users elect to leave viewed

messages on the server, rather than deleting them after viewing. This is because many people who travel extensively want to check e-mail while on the road, but want to see *all* of their messages (even the ones they've seen) when they return to their 'home machine.'
While the POP approach 'tolerates' but does not encourage this sort of user behavior, the IMAP approach eagerly embraces it. IMAP was conceived with 'nomadic' users in mind: users who might check e-mail from literally anywhere, who want access to all of their saved *and* sent messages wherever they happen to be. IMAP not only allows the user to leave messages on the server, it provides mechanisms for storing messages in user-defined folders for easier accessibility and better organization.
Moreover, users can save sent messages in a designated remote folder on the IMAP server. While POP clients support saving of sent messages, they usually save those messages locally, on the client machine.
The typical IMAP e-mail client program works very similarly to typical POP e-mail clients. (In fact, many e-mail client programs allow the user to operate in either POP or IMAP mode.) However, the automatic downloading of the content (including attachments) of *all* new messages does not occur by default in IMAP clients. Instead, an IMAP client downloads only the header information associated with new messages, requesting the body of an individual message only when the user expresses an interest in seeing it.
Because the IMAP protocol offers many more options than the POP protocol, the possibilities for what can go on in a user session are much richer. After connection and authentication, users can look at new messages, recently seen messages, unanswered messages, flagged messages, and drafts of messages yet to be sent. They can view messages in their entirety or in part (e.g. header, body, attachment), delete or move messages to other folders, or respond to messages or forward them to others.
IMAP need not be used strictly for e-mail messages. As security features allow mailbox folders to be designated as 'read only', IMAP can be used for 'message board' functionality as well. However, such functionality is usually reserved for message forum services.

DIFFERENCES BEWTWEEN POP vs. IMAP

Although it is possible to write a POP client that operates this way, most do not. POP clients tend to operate in 'burst' mode, getting all the messages on the server in one 'shot.' While this may be in some respects inefficient, it is useful for those whose online access is not persistent. By getting all the messages in one burst, users can work 'offline' with the complete set of downloaded messages, connecting to the Internet again only when they want to send responses and check for new mail.
IMAP clients assume the existence of a persistent Internet connection, allowing discrete actions to be performed on individual messages, while maintaining a connection to the IMAP server. Thus, for applications where Internet connectivity may not be persistent (e.g. a handheld device where Internet connectivity is paid for by the minute), POP might be a better choice than IMAP.

2.4.3 Message forums

Message forums are online services that allow users to write messages to be posted on the equivalent of an electronic bulletin board, and to read similar messages that others have posted. These messages are usually organized into categories so that people can find the kinds of messages they are looking for.
For years, online message forums existed in various forms. Perhaps the earliest form was the *electronic mailing list*. As we mentioned earlier, mailing lists are closed systems: only subscribers can view or post messages. In some situations, a closed private community may

be exactly what the doctor ordered. Yet if the goal is open public participation, publicly accessible message forums are more appropriate.
Although message forums were originally localized, meaning that messages appeared only on the system where they were posted, the notion of distributed message forums took hold. Cooperative networks (e.g. FIDONET) allowed systems to share messages, by forwarding them to 'neighboring' systems in the network. This enabled users to see all the messages posted by anyone on any system in the network.
The Internet version of message forums is *Netnews*. Netnews organizes messages into *newsgroups*, which form a large hierarchy of topics and categories. Among the main divisions are *comp* (for computing related newsgroups), *sci* (for scientific newsgroups), *soc* (for socially oriented newsgroups), *talk* (for newsgroups devoted to talk), and *alt* (an unregulated hierarchy for 'alternative' newsgroups). The naming convention for newsgroups is reminiscent of domain names in reverse, e.g. *comp.infosystems.www*.

Today, Netnews is transmitted using an Internet protocol called *NNTP* (for *Network News Transfer Protocol*). NNTP clients allow users to read messages in newsgroups (and post their own messages as well) by connecting to NNTP servers.
These servers propagate the newsgroup messages throughout the world by regularly forwarding them to 'neighboring' servers. The NNTP specification is defined in *Internet RFC 977*.
Netnews functionality is directly incorporated into browsers like Netscape Communicator, which includes the functionality into its Messenger component, which is responsible for accessing electronic mail. It is also possible to create web applications that provide Netnews access through normal web browser interactions. One site, deja.com (now a part of Google), created an entire infrastructure for accessing current as well as archived newsgroup messages, including a powerful search engine for finding desired messages.

2.4.4 Live messaging

America Online's *Instant Messaging* service may be responsible for making the notion of *IM*-ing someone part of our collective vocabulary. But long before the existence of AOL, there was a *talk* protocol that enabled users who were logged in to network-connected UNIX systems to talk to each other.
A *talk* server would run on a UNIX machine, waiting for requests from other *talk* servers. (Since *talk* was a bi-directional service, servers had to run on the machines at both ends of a conversation.) A user would invoke the *talk* client program to communicate with a person on another machine somewhere else on the network, e.g. *elvis@graceland.org*. The *talk* client program would communicate with the local
talk server, which would ask the *talk* server on the remote machine whether the other person is on line. If so, and if that other person was accepting *talk* requests, the remote talk server would establish a connection, and the two people would use a screen mode interface to have an online conversation.
Today, the vast majority of Internet users eschew command-line interfaces, and the notion of being logged in to a particular system (aside from AOL, perhaps) is alien to most people. Thus, a protocol like *talk* would not work in its original form in today's diverse Internet world. Efforts to create an open, interoperable Instant Messaging protocol have been unsuccessful thus far. Proprietary 'instant messaging' systems (such as AOL's) exist, but they are exclusionary, and the intense competition and lack of cooperation between instant messaging providers further limits the degree of interoperability we can expect from them.

2.4.5 File servers

E-mail and live messaging services represent fleeting, transitory communications over the Internet. Once an instant message or e-mail message has been read, it is usually discarded. Even forum-based messages, even if they are archived, lack a certain degree of permanence, and for the most part those who post such messages tend not to treat them as anything more than passing transient dialogues (or, in some cases, monologues).

However, providing remote access to more persistent documents and files is a fundamental necessity to enable sharing of resources.

For years before the existence of the Internet, files were shared using *BB* 's (electronic *Bulletin Board Systems*). People would dial in to a BBS via a modem, and once connected, they would have access to directories of files to download (and sometimes to 'drop' directories into which their own files could be uploaded).

Various file transfer protocols were used to enable this functionality over telephone dialup lines (e.g. Kermit, Xmodem, Zmodem). To facilitate this functionality over the Internet, the *File Transfer Protocol* (*FTP*) was created.

FTP

An FTP server operates in a manner similar to an e-mail server. Commands exist to authenticate the connecting user, provide the user with information about available files, and allow the user to retrieve selected files. However, e-mail servers let you access only a preset collection of folders (like the inbox), solely for purposes of downloading message files. FTP servers also allow users to traverse to different directories within the server's local file system, and (if authorized) to upload files into those directories.

The FTP specification has gone through a number of iterations over the years, but the most recent version can be found in Internet RFC 959. It describes the process by which FTP servers make files available to FTP clients.

First, a user connects to an FTP server using an FTP client program. FTP interactions usually require *two* connections between the client and server. One, the *control connection*, passes commands and status responses between the client and the server. The other, the *data connection*, is the connection over which actual data transfers occur. User authentication occurs, of course, over the control connection.

Once connected and authenticated, the user sends commands to set transfer modes, change directories, list the contents of directories, and transfer files. Whether or not a user can enter specific directories, view directory contents, download files, and/or upload files depends on the security privileges associated with his/her user account on the server. (Note that the root directory of the FTP server need not be the same as the root directory of the server machine's local file system. System administrators can configure FTP servers so that only a discrete directory subtree is accessible through the FTP server.)

FTP servers can allow open access to files without requiring explicit user authentication, using a service called anonymous FTP. When an FTP server is configured to support anonymous FTP, a user ID called 'anonymous' is defined that will accept any password. 'Netiquette' (Internet etiquette) prescribes that users should provide their e-mail address as the password. The system administrator can further restrict
the file system subtree that is accessible to 'anonymous' users, usually providing read-only access (although it is possible to configure a 'drop' folder into which anonymous users can place files). Most of the FTP archives found on the Internet make use of anonymous FTP to provide open access to files.

Other file server protocols have come into being over the years, but none has achieved the popularity of FTP. With the advent of next generation distributed file sharing systems such as the one used by Napster, we can expect to see changes in the file server landscape over the next few years.

2.5 AND THEN CAME THE WEB. . .

While FTP provided interactive functionality for users seeking to transfer files across the Internet, it was not a very user-friendly service. FTP clients, especially the command-line variety, were tedious to use, and provided limited genuine interactivity.

Once you traversed to the directory you wanted and downloaded or uploaded your files, your 'user experience' was completed. Even GUI-based FTP clients did not appreciably enhance the interactivity of FTP. Other services sought to make the online experience more truly interactive. *Gopher* was a service developed at the University of Minnesota (hence the name—Minnesota is the 'gopher state') that served up *menus* to users. In Gopher, the items in menus were not necessarily actual file system directories, as they were in FTP.

They were logical lists of items grouped according to category, leading the user to other resources. These resources did not have to be on the same system as the Gopher menu. In fact, a Gopher menu could list local resources as well as resources on other systems, including other Gopher menus, FTP archives, and (finally) files.

Again, once you reached the level of a file, your traversal was complete. There was 'nowhere to go', except to retrace your steps back along the path you just took. Gopher only caught on as a mainstream Internet service in a limited capacity. Over time, for a variety of reasons, it faded into the woodwork, in part because a better and more flexible service came along right behind it. That system married the power of the Internet with the capabilities of *hypertext*, to offer a medium for real user interactivity.

Of course, as you have already figured out, that system is the one proposed by Tim Berners-Lee in the late 1980s and early 1990s, known as the World Wide Web.

MODULE 3: Birth of the World Wide Web: HTTP

The main subject of this chapter is the *HyperText Transfer Protocol* (HTTP). We begin with a short foray into the history of the World Wide Web, followed by a discussion of its core components, with the focus on HTTP. No matter how Web technology evolves in the future, it will always be important to understand the basic protocols that enable communication between Web programs. This understanding is critical because it provides important insights into the inner workings of the wide range of Web applications.

3.1 HISTORICAL PERSPECTIVE

For all practical purposes, it all started at CERN back in 1989. That is when Tim Berners-Lee wrote a proposal for a hypertext-based information management system, and distributed this proposal among the scientists at CERN. Although initially interest in the proposal was limited, it sparked the interest of someone else at CERN, Robert Cailliau, who helped Berners-Lee reformat and redistribute the proposal, referring to the system as a 'World Wide Web'.

By the end of 1990, Berners-Lee had implemented a server and a command-line browser using the initial version of the *HyperText Transfer Protocol* (HTTP) that he designed for this system. By the middle of 1991, this server and browser were made available throughout CERN. Soon thereafter, the software was made available for anonymous FTP download on the Internet. Interest in HTTP and the Web grew, and many people downloaded the software. A newsgroup, *comp.infosystems.www*, was created to support discussion of this new technology.

Just one year later, at the beginning of 1993, there were about 50 different sites running HTTP servers. This number grew to 200 by the autumn of that year. In addition, since the specification for the HTTP protocol was openly available, others were writing their own server and browser software, including GUI-based browsers that supported typographic controls and display of images.

3.2 BUILDING BLOCKS OF THE WEB

There were three basic components devised by Tim Berners-Lee comprising the essence of Web technology:
1. A markup language for formatting hypertext documents.
2. A uniform notation scheme for addressing accessible resources over the network.
3. A protocol for transporting messages over the network.

The markup language that allowed cross-referencing of documents via hyperlinks was the *HyperText Markup Language* (HTML).We shall discuss HTML in a later chapter.
The uniform notation scheme is called the *Uniform Resource Identifier* (URI).
For historic reasons, it is most often referred to as the *Uniform Resource Locator* (URL).
HTTP is a core foundation of the World Wide Web. It was designed for transporting specialized messages over the network. The simplicity of the protocol does not always apply to HTTP interactions, which are complicated in the context of sophisticated Web applications. This will become apparent when we discuss the complex interactions between HTML, XML, and web server technologies (e.g. servlets and Java Server Pages).

3.3 THE UNIFORM RESOURCE LOCATOR

Tim Berners-Lee knew that one piece of the Web puzzle would be a notation scheme for referencing accessible resources anywhere on the Internet. He devised this notational scheme so that it would be flexible, so that it would be extensible, and so that it would support other protocols besides HTTP. This notational scheme is known as the *URL* or *Uniform Resource Locator*.

Participants in the original *World Wide Web Consortium* (also known as the *W3C*) had reservations about Berners-Lee's nomenclature. There were concerns about his use of the word 'universal' (*URL* originally stood for '*Universal Resource Locator*'), and about the way a URL specified a resource's location (which could be subject to frequent change) rather than a fixed immutable name. The notion of a fixed name for a resource came to be known as the *URN* or *Uniform Resource Name*.
URNs would be a much nicer mechanism for addressing and accessing web resources than URLs. URLs utilize 'locator' information that embeds both a server address and a file location. URNs utilize a simpler human-readable name that does not change even when the resource is moved to another location. The problem is that URNs have failed to materialize as a globally supported web standard, so for all practical purposes we are still stuck with URLs.
As a matter of convenience, W3C introduced the notion of the *URI* (or *Uniform Resource Identifier*) which was defined as the union of URLs and URNs. URL is still the most commonly used term, though URI is what you should use if you want to be a stickler for formal correctness.

Generalized notation associated with URLs

The generalized notation associated with URLs is as follows:

```
scheme://host[:port#]/path/.../[;url-params][?query-string][#anchor]
```

Let us break a URL down into its component parts:

• *scheme*—this portion of the URL designates the underlying protocol to be used (e.g. 'http' or 'ftp'). This is the portion of the URL preceding the colon and two forward slashes.

• *host*—this is either the name of the IP address for the web server being accessed. This is usually the part of the URL immediately following the colon and two forward slashes.

• *port#*—this is an optional portion of the URL designating the port number that the target web server listens to. (The default port number for HTTP servers is 80, but some configurations are set up to use an alternate port number. When they do, that number must be specified in the URL.) The port number, if it appears, is found right after a colon that immediately follows the server name or address.

• *path*—logically speaking, this is the file system path from the 'root' directory of the server to the desired document. (In practice, web servers may make use of aliasing to point to documents, gateways, and services that are not explicitly accessible from the server's root directory.) The path immediately follows the server and port number portions of the URL, and by definition *includes* that first forward slash.

• *url-params*—this once rarely used portion of the URL includes optional 'URL parameters'. It is now used somewhat more frequently, for session identifiers in web servers supporting the Java Servlet API. If present, it follows a semi-colon immediately after the path information.

• *query-string*—this optional portion of the URL contains other dynamic parameters associated with the request. Usually, these parameters are produced as the result of user-entered variables in HTML forms. If present, the query string follows a question mark in the URL. Equal signs (=) separate the parameters from their values, and ampersands (&) mark the boundaries between parameter value pairs.

• *anchor*—this optional portion of the URL is a reference to a positional marker within the requested document, like a bookmark. If present, it follows a hash mark or pound sign ('#').

Example 1: The breakout of a sample URL into components is illustrated below:

```
http://www.mywebsite.com/sj/test;id=8079?name=sviergn&x=true#stuff
SCHEME = http
HOST = www.mywebsite.com
PATH = /sj/test
URL PARAMS = id=8079
QUERY STRING = name=sviergn&x=true
ANCHOR = stuff
```

Note: note that the URL notation we are describing here applies to most protocols (e.g. http, https, and ftp). However, some other protocols use their own notations (e.g. "mailto:richr@neurozen.com").

3.4 FUNDAMENTALS OF HTTP

HTTP is the foundation protocol of the World Wide Web. It is simple, which is both a limitation and a source of strength. Many people in the industry criticized HTTP for its lack of state support and limited functionality, but HTTP took the world by storm while more advanced and sophisticated protocols never realized their potential.
HTTP is an *application level* protocol in the TCP/IP protocol suite, using TCP as the underlying Transport Layer protocol for transmitting messages. The fundamental things worth knowing about the HTTP protocol and the structure of HTTP messages are:

1. The HTTP protocol uses the *request/response paradigm*, meaning that an HTTP client program sends an HTTP request message to an HTTP server, which returns an HTTP response message.
2. The structure of request and response messages is similar to that of e-mail messages; they consist of a group of lines containing *message headers*, followed by a blank line, followed by a *message body*.
3. HTTP is a *stateless* protocol, meaning that it has no explicit support for the notion of state. An HTTP transaction consists of a single request from a client to a server, followed by a single response from the server back to the client.

3.4.1 HTTP servers, browsers, and proxies

Web servers and browsers exchange information using HTTP, which is why Web servers are often called HTTP servers. Similarly, Web browsers are sometimes referred to as HTTP clients, but their functionality is not limited to HTTP support.
It was Tim Berners-Lee's intent that web browsers should enable access to a wide variety of content, not just content accessible via HTTP. Thus, even the earliest web browsers were designed to support other protocols including FTP and Gopher.
Today, web browsers support not only HTTP, FTP, and local file access, but e-mail and netnews as well.
HTTP proxies are programs that act as both servers and clients, making requests to web servers on behalf of other clients. Proxies enable HTTP transfers across firewalls. They also provide support for caching of HTTP messages and filtering of HTTP requests. They also fill a variety of other interesting roles in complex environments.
When we refer to HTTP clients, the statements we make are applicable to browsers, proxies, and other custom HTTP client programs.

3.4.2 Request/response paradigm

First and foremost, HTTP is based on the *request/response paradigm*: browsers (and possibly proxy servers as well) send messages to HTTP servers. These servers generate messages that are sent back to the browsers. The messages sent to HTTP servers are called *requests*, and the messages generated by the servers are called *responses*.
In practice, servers and browsers rarely communicate directly—there are one or more proxies in between. A *connection* is defined as a *virtual circuit* that is composed of HTTP agents, including the browser, the server, and intermediate proxies participating in the exchange.

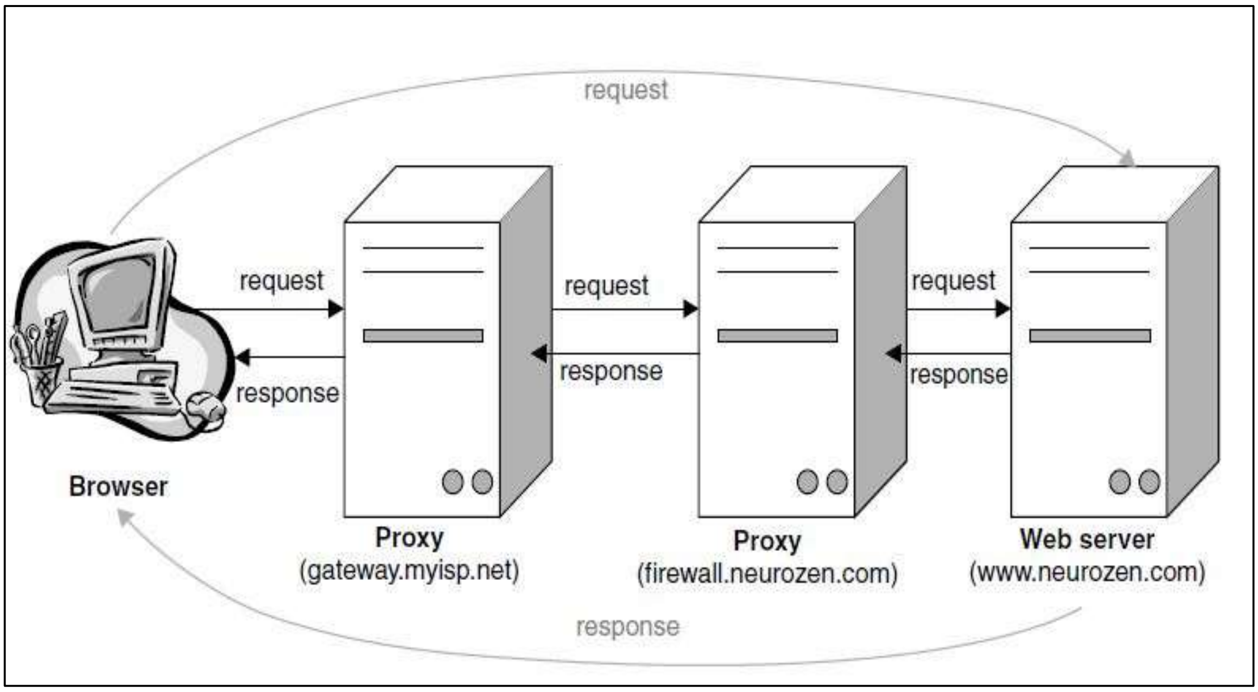


Figure 3.1: The request/response virtual circuit

3.4.3 Stateless protocol

As mentioned in the previous chapter, HTTP is a *stateless* protocol. When a protocol supports 'state', this means that it provides for the interaction between client and server to contain a sequence of commands. The server is required to maintain the 'state' of the connection throughout the transmission of successive commands, until the connection is terminated. The sequence of transmitted and executed commands is often called a *session*. Many Internet protocols, including FTP, SMTP and POP are 'stateful' protocols.

In contrast, HTTP is said to be 'stateless'. Defining HTTP as a stateless protocol made things simpler, but it also imposed limitations on the capabilities of Web applications. By definition, the lifetime of a connection was a single request/response exchange. This meant that there was no way to maintain persistent information about a 'session' of successive interactions between a client and server. This also meant that there was no way to 'batch' requests together—something that would be useful, for example, to ask a web server for an HTML page and all the images it references during the course of one connection.

The evolution of *cookies is usually used* as a mechanism for maintaining state in Web applications. Certain advanced strategies are used in HTTP/1.1 to support connections that outlive a single request/response exchange.
HTTP/1.1 assumes that the connection remains in place until it is broken, or until an HTTP client requests that it be broken. However, HTTP/1.1 is designed to support persistent connections for the sake of efficiency, but not to support state. We will come back to the technicalities of establishing and breaking HTTP connections when we discuss HTTP/1.1 in detail.

3.4.4 The structure of HTTP messages

HTTP messages (both requests and responses) have a structure similar to e-mail messages; they consist of a block of lines comprising the *message headers*, followed by a blank line, followed by a *message body*. The structure of HTTP messages, however, is more sophisticated than the structure of e-mail messages.

Let us start with a very simple example: loading a static web page residing on a web server. A user may manually type a URL into her browser, she may click on a hyperlink found within the page she is viewing with the browser, or she may select a bookmarked page to visit. In each of these cases, the desire to visit a particular URL is translated by the browser into an HTTP request. An HTTP request message has the following structure:

```
METHOD /path-to-resource HTTP/version-number
Header-Name-1: value
Header-Name-2: value
[ optional request body ]
```

Every request starts with the special *request line*, which contains a number of fields. The 'method' represents one of several supported *request methods*, chief among them 'GET' and 'POST'. The '/path-to-resource' represents the *path* portion of the requested URL. The 'version-number' specifies the version of HTTP used by the client.

After the first line we see a list of HTTP *headers*, followed by a blank line, often called a <CR><LF> (for '*carriage return and line feed* '). The carriage return means moving the cursor to the beginning of the line without advancing to the next line (the code is \r). The line feed means moving the cursor forward or to the next line without returning to the beginning of the line (the code is \n).
The blank line separates the request headers from the body of the request. The blank line is followed (optionally) by a *body*, which is in turn followed by another blank line indicating the end of the request message.

For our purposes, let http://www.mywebsite.com/sj/index.html be the requested URL. Here is a simplified version of the HTTP request message that would be transmitted to the web server at www.mywebsite.com:

```
GET /sj/index.html HTTP/1.1
Host: www.mywebsite.com
```

Note that the request message ends with a blank line. In the case of a GET request, there is no body, so the request simply ends with this blank line. Also, note the presence of a Host header. The server, upon receiving this request, attempts to generate a response message.

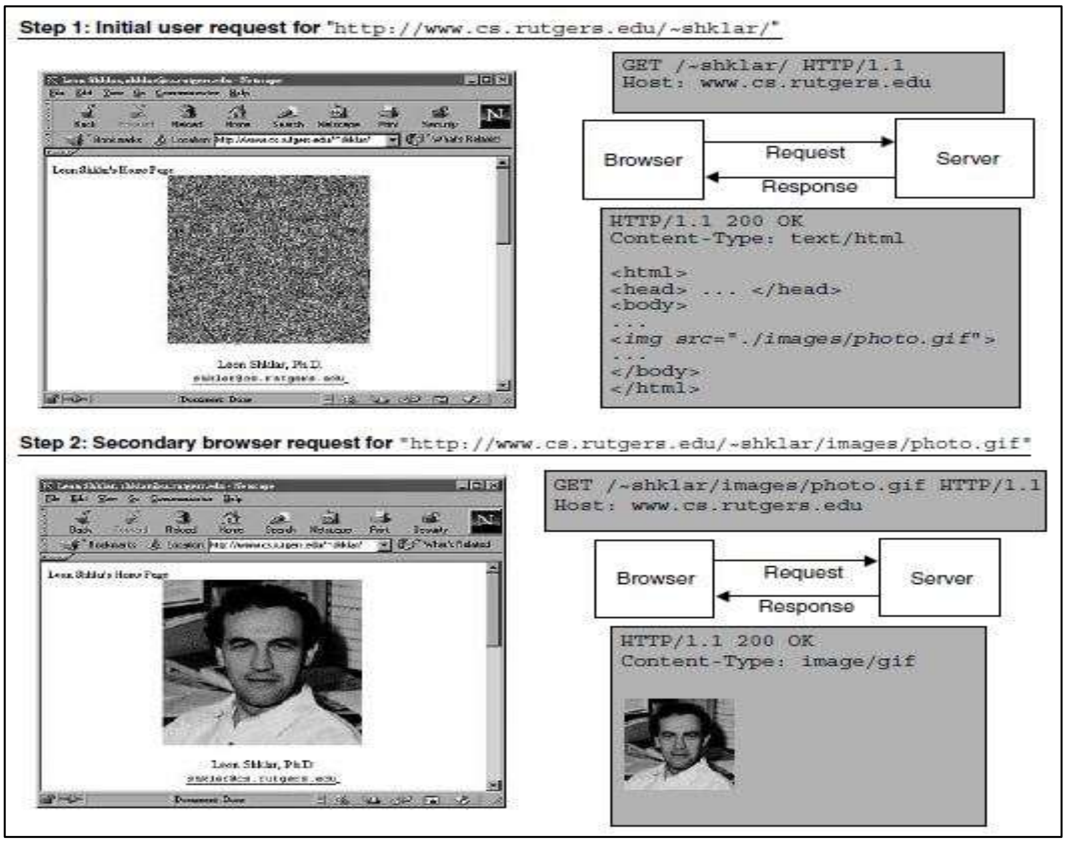
An HTTP response message has the following structure:

```
HTTP/version-number status-code message
Header-Name-1: value
Header-Name-2: value
[ response body ]
```

The first line of an HTTP response message is the *status line*. This line contains the version of HTTP being used, followed by a three-digit *status code*, and followed by a brief human-readable explanation of the status code. This is a simplified version of the HTTP response message that the server would send back to the browser, assuming that the requested file exists and is accessible to the requestor:

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 9934
...
<HTML>
<HEAD>
<TITLE>SJ_s Web Page</TITLE>
</HEAD>
<BODY BGCOLOR="#ffffff">
<H2 ALIGN="center">Welcome to Sviergn Jiernsen's Home Page</H2>
...
</H2>
</BODY>
</HTML>
```

Note that the response message begins with a *status line*, containing the name and version of the protocol in use, a numeric response status code, and a human-readable message. In this case, the request produced a successful response, thus we see a success code (200) and a success message (OK).
Note the presence of header lines within the response, followed by a blank line, followed by a block of text. (We shall see later how a browser figures out that this text is to be rendered as HTML.)



Step 1: Initial user request for "http://www.cs.rutgers.edu/~shklar/"

```
GET /~shklar/ HTTP/1.1
Host: www.cs.rutgers.edu
```

```
HTTP/1.1 200 OK
Content-Type: text/html

<html>
<head> ... </head>
<body>
...
<img src="./images/photo.gif">
...
</body>
</html>
```

Step 2: Secondary browser request for "http://www.cs.rutgers.edu/~shklar/images/photo.gif"

```
GET /~shklar/images/photo.gif HTTP/1.1
Host: www.cs.rutgers.edu
```

```
HTTP/1.1 200 OK
Content-Type: image/gif
```

Figure 3.2 Sequence of browser requests for loading a sample page

The process of transmitting requests and responses between servers and browsers is rarely this simplistic. Complex negotiations occur between browsers and servers to determine what information should be sent. For instance, HTML pages may contain references to other accessible resources, such as graphical images and Java applets. Clients that support the rendering of images and applets, which is most web browsers, must parse the retrieved HTML page to determine what additional resources are needed, and then send HTTP requests to retrieve those additional resources (Figure 3.2). Server-browser interactions can become much more complex for advanced applications.

Differences between E-Mail Messages vs. HTTP Messages

E-mail messages are intended to pass information directly between people. Thus, both the message headers and the body tend to be 'human-readable'. E-mail messages (at least originally) had message bodies that consisted simply of readable plain text, while their message headers included readable information like the sender address and the message subject.

Over time, e-mail message structure became more sophisticated, in part to provide support for MIME functionality. There were headers added to allow decompression, decoding, and reformatting of message content based on its MIME type. In addition, multi-part messages were supported, allowing messages to have multiple sections (often corresponding to a body and a set of attachments).

When HTTP servers and browsers communicate with each other, they perform sophisticated interactions, based on header and body content. Unlike e-mail messages, HTTP messages are not intended to be directly 'human-readable'.

Another fundamental difference is that HTTP request and response messages begin with special lines that do not follow the standard header format. For requests, this line is called the *request line*, and for responses, it is called the *status line*.

3.4.5 Request methods

There are varieties of *request methods* specified in the HTTP protocol. The most basic ones defined in HTTP/1.1 are GET, HEAD, and POST. In addition, there are the less commonly used PUT, DELETE, TRACE, OPTIONS and CONNECT.

Request methods impose constraints on message structure, and specifications that define how servers should process requests, including the *Common Gateway Interface* (CGI) and the *Java Servlet API*, include discussion of how different request methods should be treated.

GET

The simplest of the request methods is GET. When you enter a URL in your browser, or click on a hyperlink to visit another page, the browser uses the GET method when making the request to the web server.

GET requests date back to the very first versions of HTTP. A GET request does not have a body and, until the version 1.1, was not required to have headers.

(HTTP/1.1 requires that the Host header should be present in every request in order to support *virtual hosting*.)

In the previous section, we offered an example of a very simple GET request. In that example, we visited a URL, http://www.mywebsite.com/sj/index.html, using the GET method. Let's take a look at the request that gets submitted by an HTTP/1.1 browser when you fill out a simple HTML form to request a stock quote:

```
<HTML>
<HEAD><TITLE>Simple Form</TITLE></HEAD>
<BODY>
<H2>Simple Form</H2>
<FORM ACTION="http://finance.yahoo.com/q" METHOD="get">
Ticker: <INPUT SIZE="25" NAME="s">
<INPUT TYPE="submit" VALUE="Get Quote">
</FORM>
<BODY>
</HTML>
```

If we enter ‘YHOO’ in the form above, then the browser constructs a URL comprised of the ‘ACTION’ field from the form followed by a *query string* containing all of the form’s input parameters and the values provided for them. The boundary separating the URL from the query string is a question mark. Thus, the URL constructed by the browser is http://www.finance.yahoo.com/q?s=YHOO and the submitted request looks as follows:

```
GET /q?s=YHOO HTTP/1.1
Host: finance.yahoo.com
User-Agent: Mozilla/4.75 [en] (WinNT; U)
```

The response that comes back from the server looks something like this:

```
HTTP/1.0 200 OK
Date: Sat, 03 Feb 2001 22:48:35 GMT
Connection: close
Content-Type: text/html
Set-Cookie: B=9ql5kgct7p2m3&b=2;expires=Thu,15 Apr 2010 20:00:00 GMT;
path=/; domain=.yahoo.com
<HTML>
<HEAD><TITLE>Yahoo! Finance - YHOO</TITLE></HEAD>

<BODY>
...
</BODY>
</HTML>
```

POST

A fundamental difference between GET and POST requests is that POST requests have a *body*: content that follows the block of headers, with a blank line separating the headers from the body. Going back to the sample form in Section 3.2, let’s change the request method to POST and notice that the browser now puts form parameters into the body of the message, rather than appending parameters to the URL as part of a query string:

```
POST /q HTTP/1.1
Host: finance.yahoo.com
User-Agent: Mozilla/4.75 [en] (WinNT; U)
Content-Type: application/x-www-form-urlencoded
Content-Length: 6
s=YHOO
```

Note that the URL constructed by the browser does *not* contain the form parameters in the query string. Instead, these parameters are included after the headers as part of the message body:

```
HTTP/1.0 200 OK
Date: Sat, 03 Feb 2001 22:48:35 GMT
Connection: close
Content-Type: text/html
Set-Cookie: B=9ql5kgct7p2m3&b=2;expires=Thu,15 Apr 2010 20:00:00 GMT;
path=/; domain=.yahoo.com
<HTML>
<HEAD><TITLE>Yahoo! Finance - YHOO</TITLE></HEAD>
<BODY>
...
</BODY>
</HTML>
```

Note that the response that arrives from finance.yahoo.com happens to be exactly the same as in the previous example using the GET method, but only because designers of the server application decided to support both request methods in the same way.

HOW APPLICATIONS USE GET and POST METHODS

Many Web applications are intended to be 'sensitive' to the request method employed when accessing a URL. Some applications may accept one request method but not another. Others may perform different functions depending on which request method is used.
For example, some servlet designers write Java servlets that use the GET method to display an input form. The ACTION field of the form is the same servlet (using the same URL), but using the POST method. Thus, the application is constructed so that it knows to display a form when it receives a request using the GET method, and to process the form (and to present the results of processing the form) when it receives a request using the POST method.

HEAD

Requests that use the HEAD method operate similarly to requests that use the GET method, except that the server sends back only headers in the response. This means the *body* of the request is not transmitted, and only the response metadata found in the headers is available to the client. This response metadata, however, may be sufficient to enable the client to make decisions about further processing, and may possibly reduce the overhead associated with requests that return the actual content in the message body.

If we were to go back to the sample form and change the request method to HEAD, we would notice that the request does not change (except for replacing the word 'GET' with the word 'HEAD', of course), and the response contains the same headers as before but no body.

Historically, HEAD requests were often used to implement caching support. A browser can use a cached copy of a resource (rather than going back to the original source to re-request the content) if the cache entry was created after the date that the content was last modified. If the creation date for the cache entry is *earlier* than the content's last modification date, then a 'fresh' copy of the content should be retrieved from the source.

Suppose we want to look at a page that we visit regularly in our browser (e.g. Leon's home page). If we have visited this page recently, the browser will have a copy of the page stored in its cache. The browser can make a determination as to whether it needs to re-retrieve the page by first submitting a HEAD
request:

```
HEAD http://www.cs.rutgers.edu/~shklar/ HTTP/1.1
Host: www.cs.rutgers.edu
User-Agent: Mozilla/4.75 [en] (WinNT; U)
```

The response comes back with a set of headers, including content modification information:

```
HTTP/1.1 200 OK
Date: Mon, 05 Feb 2001 03:26:18 GMT
Server: Apache/1.2.5
Last-Modified: Mon, 05 Feb 2001 03:25:36 GMT
Content-Length: 2255
Content-Type: text/html
```

The browser (or some other HTTP client) can compare the content modification date with the creation date of the cache entry, and resubmit the same request with the GET method if the cache entry is obsolete. We save bandwidth when the content does not change by making a HEAD request. Since responses to HEAD requests do not include the content as part of the message body, the overhead is smaller than making an explicit GET request for the content.
Today, there are more efficient ways to support caching. The HEAD method is still very useful for implementing change tracking systems, for testing and debugging new applications, and for learning server capabilities.

3.4.6 Status codes

The first line of a response is the *status line*, consisting of the protocol and its version number, followed by a three-digit *status code* and a brief explanation of that status code. The status code tells an HTTP client (browser or proxy) either that the response was generated as expected, or that the client needs to perform a specific action (that may be further parameterized via information in the headers).
The explanation portion of the line is for human consumption; changing or omitting it will not cause a properly designed HTTP client to change its actions.
Status codes are grouped into categories. HTTP Version 1.1 defines five categories of response messages:

- ***1xx***—Status codes that start with '1' are classified as *informational*.
- ***2xx***—Status codes that start with '2' indicate *successful* responses.
- ***3xx***—Status codes that start with '3' are for purposes of *redirection*.
- ***4xx***—Status codes that start with '4' represent *client request errors*.
- ***5xx***—Status codes that start with '5' represent *server errors*.

1. Informational status codes (1xx)

These status codes serve solely informational purposes. They do not denote success or failure of a request, but rather impart information about how a request can be processed further. For example, a status code of "100" is used to tell the client that it may continue with a *partially submitted request*. Clients can specify a partially submitted request by including an 'Expect' header in the request message. A server can examine requests containing an 'Expect' header, determine whether or not it is capable of satisfying the request, and send an appropriate response. If the server *is* capable of satisfying the request, the response will contain a status code of ‘100’:

```
HTTP/1.1 100 Continue
...
```

If it cannot satisfy the request, it will send a response with a status code indicating a client request error, i.e. ‘417’:

```
HTTP/1.1 417 Expectation Failed
...
```

2. Successful response status codes (2xx)

The most common successful response status code is ‘200’, which indicates that the request was successfully completed and that the requested resource is being sent back to the client:

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 9934
...
<HTML>
<HEAD>
<TITLE>SJ’s Web Page</TITLE>
</HEAD>
<BODY BGCOLOR="#ffffff">
<H2 ALIGN="center">Welcome to Sviergn Jiernsen’s Home Page</H2>
...
</H2>
</BODY>
</HTML>
```

Another example is ‘201’, which indicates that the request was satisfied and that a new resource was created on the server.

3. Redirection status codes (3xx)

Status codes of the form ‘3xx’ indicate that additional actions are required to satisfy the original request. Normally this involves a *redirection*: the client is instructed to ‘redirect’ the request to another URL.
For example, ‘301’ and ‘302’ both instruct the client to look for the originally requested resource at the new location specified in the ‘Location’ header of the response. The difference between the two is that ‘301’ tells the client that the resource has ‘Moved Permanently’, and that it should always look for that resource at the new location. ‘302’ tells the client that the resource has ‘Moved Temporarily’, and to consider this relocation a one-time deal, just for purposes of this request. In either case, the client should, immediately upon receiving a 301 or 302 response, construct and transmit a new request ‘redirected’ at the new location.

Redirections happen all the time, often unbeknownst to the user. Browsers are designed to respond silently to redirection status codes, so that users never see redirection ‘happen’. A perfect example of such silent redirection occurs when a user enters a URL specifying a directory, but leaving off the terminating slash. To visit Leon's web site at Rutgers University, you could enter http://www.cs.rutgers.edu/~shklar in your browser. This would result in the following HTTP request:

```
GET /~shklar HTTP/1.1
Host: www.cs.rutgers.edu
```

But "~shklar" is actually a directory on the Rutgers web server, not a deliverable file. Web servers are designed to treat a URL ending in a slash as a request for a directory. Such requests may, depending on server configuration, return either a file with a default name (if present), e.g. index.html, or a listing of the directory's contents. In either case, the web server must first redirect the request, from http://www.cs.rutgers.edu/~shklar to http://www.cs.rutgers.edu/~shklar/, to properly present it:

```
HTTP/1.1 301 Moved Permanently
Location: http://www.cs.rutgers.edu/~shklar/
Content-Type: text/html
...
<html>
<head><title>301 Moved Permanently</title></head>
<body>
<h1>301 Moved Permanently</h1>
The document has moved
<a href="http://www.cs.rutgers.edu/~shklar/">here</a>.
</body>
</html>
```

Today's sophisticated browsers are designed to react to '301' by updating an internal relocation table, so that in the future they can substitute the new address prior to submitting the request, and thus avoid the relocation response. To support older browsers that do not support automatic relocation, web servers still include a message body that explicitly includes a link to the new location. This affords the user an opportunity to manually jump to the new location.

NOTE: Remember the Slash!
This example offers a valuable lesson: if you are trying to retrieve a directory listing (or the default page associated with a directory), don't forget the trailing '/'. When manually visiting a URL representing a directory in your browser, you may not even notice the redirection and extra connection resulting from omitting the trailing slash.
However, when your applications generate HTML pages containing links to directories, forgetting to add that trailing slash within these links will effectively double the number of requests sent to your server.

Client request error status codes (4xx)
Status codes that start with '4' indicate a problem with the client request (e.g. '400 Bad Request'), an authorization challenge (e.g. '401 Not Authorized'), or the server's inability to find the requested resource (e.g. '404 Not Found'). Although '400', '401', and '404' are the most common in this category, some less common status codes are quite interesting. We have already seen (in the section on 'Informational Status Codes') an example of the use of '417

Expectation Failed'. In another example, the client might use the 'If-Unmodified-Since' header to request a resource only if it has not changed since a specific date:

```
GET/~shklar/HTTP/1.1
Host: www.cs.rutgers.edu
If-Unmodified-Since: Fri, 11 Feb 2000 22:28:00 GMT
```

Since this resource did change, the server sends back the '412 Precondition Failed' response:

```
HTTP/1.1 412 Precondition Failed
Date: Sun, 11 Feb 2001 22:28:31 GMT
Server: Apache/1.2.5
```

5. Server error status codes (5xx)
Finally, status codes that start with '5' indicate a server problem that prevents it from satisfying an otherwise valid request (e.g. '500 Internal Server Error' or '501 Not Implemented').
Status codes represent a powerful means of controlling browser behavior. There are a large number of different status codes representing different response conditions, and they are well documented in Internet RFC 2616. Familiarity with status codes is obviously critical when implementing an HTTP server, but it is just as critical when building advanced Web applications.

3.5 BETTER INFORMATION THROUGH HEADERS

As we already know, HTTP headers are a form of message metadata. Enlightened use of headers makes it possible to construct sophisticated applications that establish and maintain sessions, set caching policies, control authentication, and implement business logic. The HTTP protocol specification makes a clear distinction between *general headers, request headers, response headers*, and *entity headers*.

1. *General headers* apply to both request and response messages, but do not describe the body of the message. Examples of general headers include

• `Date: Sun, 11 Feb 2001 22:28:31 GMT`
This header specifies the time and date that this message was created.
• `Connection: Close`
This header indicates whether or not the client or server that generated the message intends to keep the connection open.
• `Warning: Danger, Will Robinson!`
This header stores text for human consumption, something that would be useful when tracing a problem.

2. *Request headers* allow clients to pass additional information about themselves and about the request. For example:

• `User-Agent: Mozilla/4.75 [en] (WinNT; U)`
Identifies the software (e.g. a web browser) responsible for making the request
• `Host: www.neurozen.com`
This header was introduced to support virtual hosting, a feature that allows a web server to service more than one domain.

• `Referer: http://www.cs.rutgers.edu/~shklar/index.html`
This header provides the server with context information about the request. If the request came about because a user clicked on a link found on a web page, this header contains the URL of that referring page.
• `Authorization: Basic [encoded-credentials]`
This header is transmitted with requests for resources that are restricted only to authorized users. Browsers will include this header after being notified of an *authorization challenge* via a response with a '401' status code. They consequently prompt users for their credentials (i.e. *userid* and *password*). They will continue to supply those credentials via this header in all further requests during the current browser session that access resources within the same authorization realm. (See the description of the WWW-Authenticate header below, and the section on 'Authorization' that follows.)

3. *Response headers* help the server to pass additional information about the response that cannot be inferred from the status code alone. Here are some examples:

• `Location: http://www.mywebsite.com/relocatedPage.html`
This header specifies a URL towards which the client should redirect its original request. It always accompanies the '301' and '302' status codes that direct clients to try a new location.
• `WWW-Authenticate: Basic realm="KremlinFiles"`
This header accompanies the '401' status code that indicates an authorization challenge. The value in this header specifies the protected realm for which proper authorization credentials must be provided before the request can be processed.
In the case of web browsers, the combination of the '401' status code and the WWW-Authenticate header causes users to be prompted for ids and passwords.
• `Server: Apache/1.2.5`
This header is not tied to a particular status code. It is an optional header that identifies the server software.

4. *Entity headers* describe either message bodies or (in the case of request messages that have no body) target resources. Common entity headers include:

• `Content-Type: mime-type/mime-subtype`
This header specifies the MIME type of the message body's content.
• `Content-Length: xxx`
This optional header provides the length of the message body. Although it is optional, it is useful for clients such as web browsers that wish to impart information about the progress of a request. Where this header is omitted, the browser can only display the amount of data downloaded. But when the header is included, the browser can display the amount of data as a *percentage* of the total size of the message body.
• `Last-Modified: Sun, 11 Feb 2001 22:28:31 GMT`
This header provides the last modification date of the content that is transmitted in the body of the message. It is critical for the proper functioning of caching mechanisms.

3.5.1 Type support through content-type

So far, we were concentrating on message metadata, and for a good reason: understanding metadata is critical to the process of building applications. Still, somewhere along the line, there will be some content. After all, without content, Web applications would have nothing to present for end users to see and interact with.

When it comes to content you view on the Web, your browser might do one of several things. It might:
• render the content as an HTML page,
• launch a *helper application* capable of presenting non-HTML content,
• present such content inline (within the browser window) through a *plug-in*, or
• get confused into showing the content of an HTML file as plain text without attempting to render it.
So, browsers do *something* to determine the content type and to perform actions appropriate for that type. Therefore, HTTP borrows its content typing system from *Multipurpose Internet Mail Extensions* (*MIME*). MIME is the standard that was designed to help e-mail clients to display non-textual content.

As in MIME, the data type associated with the body of an HTTP message is defined via a two-layer ordered encoding model, using Content-Type and Content-Encoding headers. In other words, for the body to be interpreted according to the type specified in the Content-Type header, it has to first be decoded according to the encoding method specified in the Content-Encoding header.
In HTTP/1.1, defined content encoding methods for the Content-Encoding header are "gzip", "compress" and "deflate". The first two methods correspond to the formats produced by GNU zip and UNIX compress programs. The third method, "deflate", corresponds to the zlib format associated with the deflate compression mechanism documented in RFC 1950 and 1951. Note that "x-gzip" and "x-compress" are equivalent to "gzip" and "compress" and should be supported for backward compatibility.

Obviously, if web servers encode content using these encoding methods, web browsers (and other clients) must be able to perform the reverse operations on encoded message bodies prior to rendering or processing of the content. Browsers are intelligent enough to open a compressed document file (e.g. test.doc.gz) and automatically invoke Microsoft Word to let you view the original test.doc file.
It can do this if the web server includes the "Content-Encoding: gzip" header with the response. This header will cause a browser to *decode* the encoded content prior to presentation, revealing the test.doc document inside.
The Content-Type header is set to a media-type that is defined as a combination of a type, subtype and any number of optional attribute/value pairs:

```
media-type = type "/" subtype *( ";" parameter-string )
type = token
subtype = token
```

The most common example is "Content-Type: text/html" where the type is set to "text" and the subtype is set to "html". This obviously tells a browser to render the message body as an HTML page. Another example is:

```
Content-Type: text/plain; charset = ’us-ascii’
```

Here the subtype is "plain", plus there is a parameter string that is passed to whatever client program ends up processing the body whose content type is "text/plain".
The parameter may have some impact on how the client program processes the content. If the parameter is not known to the program, it is simply ignored. Some other examples of MIME

types are "text/xml" and "application/xml" for XML content, "application/pdf" for Adobe Portable Data Format, and "video/x-mpeg" for MPEG2 videos.
Since MIME was introduced to support multi-media transfers over e-mail, it is not surprising that it provides for the inclusion of multiple independent entities within a single message body. In e-mail messages, these *multipart messages* usually take the form of a textual message body plus attachments.
This multipart structure is very useful for HTTP transfers going in both directions (client-to-server and server-to-client). In the client-to-server direction, form data submitted via a browser can be accompanied by file content that is transmitted to the server. We will discuss multipart messages used for form submission when we talk about HTML in a later chapter.
In the server-to-client direction, a web server can implement primitive image animation by feeding browsers a multipart sequence of images. Netscape's web site used to include a demo of the primitive image animation technique that generated a stream of pictures of Mozilla (the Godzilla-like dragon that was the mascot of the original Netscape project):

```
GET /cgi-bin/doit.cgi HTTP/1.1
Host: cgi-bin.netscape.com
Date: Sun, 18 Feb 2001 06:22:19 GMT
```

The response is a "multipart/x-mixed-replace" message as indicated by the Content-Type header. This content type instructs the browser to render enclosed image bodies one at a time, but within the same screen real estate. The individual images are encoded and separated by the *boundary* string specified in the header:

```
HTTP/1.1 200 OK
Server: Netscape-Enterprise-3.6 SP1
Date: Date: Sun, 18 Feb 2001 06:22:31 GMT
Content-Type: multipart/x-mixed-replace; boundary=ThisRandomString
Connection: close
--ThisRandomString
Content-Type: image/gif
...
--ThisRandomString
Content-Type: image/gif
...
--ThisRandomString
Content-Type: image/gif
...
--ThisRandomString
```

Message typing is necessary to help both servers and browsers determine proper actions in processing requests and responses. Browsers use types and sub-types to either select a proper rendering module or to invoke a third-party tool (e.g. Microsoft Word). Multipart rendering modules control recursive invocation of proper rendering modules for the body parts.

In the example above, the browser's *page* rendering module for the multipart message of type 'multipart/x-mixed-replace' invokes the browser's *image* rendering module once per image while always passing it the same screen location. Server-side applications use type information to process requests. For, example, a server-side application responsible for receiving files from browsers and storing them locally needs type information, to separate file content from accompanying form data that defines file name and target location.

3.5.2 Caching control through Pragma and Cache-Control headers

Caching is a set of mechanisms allowing responses to HTTP requests to be held in some form of temporary storage medium, as a means of improving server performance.
Instead of satisfying future requests by going back to the original data source, the held copy of the data can be used. This eliminates the overhead of re-executing the original request and greatly improves server throughput.
There are three main types of caching that are employed in a Web application environment: *server-side* caching, *browser-side* caching, and *proxy-side* caching.

When is the use of a cached response appropriate? This is a decision usually made by the server, or by Web applications running on the server. Many requests arrive at a given URL, but the server may deliver different content for each request, as the underlying source of the content is constantly changing. If the server 'knows' that the content of a response is relatively static and is not likely to change, it can instruct browsers, proxies, and other clients to cache that particular response. If the content is so static that it is never expected to change, the server can tell its clients that the response can be cached for an arbitrarily long amount of time. If the content has a limited lifetime, the server can still make use of caching by telling its clients to cache the response but only for that limited period. Even if the content is constantly changing, the server can make the decision that its clients can 'tolerate' a cached response (containing somewhat out-of-date content) for a specified time period.
Web servers and server-side applications are in the best position to judge whether clients should be allowed to cache their responses. There are two mechanisms for establishing caching rules. The first is associated with an older version of the HTTP protocol, version 1.0. The second is associated with HTTP version 1.1. Because there are web servers and clients that still support only HTTP 1.0, any attempt to enable caching must support both mechanisms in what is hopefully a backward compatible fashion.

HTTP/1.1 provides its own mechanism for enforcing caching rules: the Cache-Control header. Valid settings include public, private, and no-cache. The public setting removes all restrictions and authorizes both shared and non-shared caching mechanisms to cache the response.

The private setting indicates that the response is directed at a single user and should not be stored in a shared cache. For instance, if two authorized users both make a secure request to a particular URL to obtain information about their private accounts, obviously it would be a problem if an intermediate proxy decided it could improve performance for the second user by sending her a cached copy of the first user's response.

The no-cache setting indicates that neither browsers nor proxies are allowed to cache the response. However, there are a number of options associated with this setting that make it somewhat more complicated than that. The header may also list the names of specific HTTP headers that are 'non-cached' (i.e. that must be re-acquired from the server that originated the cached response). If such headers *are* listed, then the response may be cached, *excluding* those listed headers.

HTTP/1.0 browsers and proxies are not guaranteed to obey instructions in the Cache-Control header that was first introduced in HTTP/1.1. For practical purposes, this means that this mechanism is only reliable in very controlled environments where you know for sure that all your clients are HTTP/1.1 compliant. In the real world, there are still many HTTP/1.0 browsers and proxies out there, so this is not practical.

A partial solution is to use the deprecated Pragma header that has only one defined setting: no-cache. When used with the Cache-Control header, it will prevent HTTP/1.0 browsers and proxies from caching the response. However, this alone may not have the desired effect on clients that are HTTP/1.1 compliant, since the Pragma header is deprecated and may not be properly supported in those clients. Thus, a more complete backwards-compatible solution would be to include both Pragma and Cache-Control headers, as in the following example:

```
HTTP/1.1 200 OK
Date: Mon, 05 Feb 2001 03:26:18 GMT
Server: Apache/1.2.5
Last-Modified: Mon, 05 Feb 2001 03:25:36 GMT
Cache-Control: private
Pragma: no-cache
Content-Length: 2255
Content-Type: text/html
<html>
. . .
</html>
```

This response is guaranteed to prevent HTTP/1.0 agents from caching the response and to prevent HTTP/1.1 agents from storing it in a shared cache. HTTP/1.1 agents may or may not ignore the Pragma: no-cache header, but we played it safe in this example to ensure that we do not implement a potentially more restrictive caching policy than intended.

3.5.3 Security through WWW-Authenticate and Authorization headers

HTTP provides built-in support for *basic authentication*, in which authorization credentials (userid and password) are transmitted via the Authorization header as a single encoded string. Since this string is simply encoded (not encrypted), this mechanism is only safe if performed over a secure connection.

Many Web applications implement their own authentication schemes that go above and beyond basic HTTP authentication. It is very easy to tell whether an application is using built-in HTTP authentication or its own authentication scheme. When a Web application is using built-in HTTP authentication, the browser brings up its own authentication dialog, prompting the user for authorization credentials, rather than prompting the user for this information within one of the browser's page rendering windows. Application-specific schemes *will* prompt users in the main browser window, as part of a rendered HTML page.

When built-in HTTP authentication is employed, browsers are responding to a pre-defined status code in server responses, namely the '401' status code indicating that the request is not authorized. Let's take a look at the server response that tells the browser to prompt for a password:

```
HTTP/1.1 401 Authenticate
Date: Mon, 05 Feb 2001 03:41:23 GMT
```

```
Server: Apache/1.2.5
```
WWW-Authenticate: Basic realm="Chapter3"

When a request for a restricted resource is sent to the server, the server sends back a response containing the ‘401’ status code. In response to this, the browser prompts the user for a userid and password associated with the realm specified in the WWW-Authenticate header (in this case, "Chapter3").

The realm name serves both as an aid in helping users to retrieve their names and passwords, and as a logical organizing principle for designating which resources require what types of authorization. Web server administrative software gives webmasters the ability to define realms, to decide which resources ‘belong’ to these realms, and to establish userids and passwords that allow only selected people to access resources in these realms.

In response to the browser prompt, the user specifies his name and password. Once a browser has collected this input from the user, it resubmits the original request with the additional Authorization header. The value of this header is a string containing the type of authentication (usually "Basic") and a Base64-encoded representation of the concatenation of the user name and password (separated by a colon):

```
GET /book/chapter3/index.html HTTP/1.1
Date: Mon, 05 Feb 2001 03:41:24 GMT
Host: www.neurozen.com
```
Authorization: Basic eNCoDEd-uSErId:pASswORd

The server, having received the request with the Authorization header, attempts to verify the authorization credentials. If the userid and password match the credentials defined within that realm, the server then serves the content. The browser associates these authorization credentials with the authorized URL, and uses them as the value of the Authorization header in future requests to *dependent URLs*. Since the browser does this automatically, users do not get prompted again until they happen to encounter a resource that belongs to a different security realm.

If the server fails to verify the userid and password sent by the browser, it either resends the security challenge using the status code 401, or refuses to serve the requested resource outright, sending a response with the status code of 403 Forbidden.
The latter happens when the server exceeds a defined limit of security challenges. This limit is normally configurable and is designed to prevent simple break-ins by trial-and-error.

We described the so-called *basic* authentication that is supported by both HTTP/1.0 and HTTP/1.1. It is a bit simplistic, but it does provide reasonable protection—as long as you are transmitting over a secure connection. Most commercial applications that deal with sensitive financial data use their own authentication mechanisms that are not a part of HTTP. Commonly, user names and passwords are transmitted in bodies of POST requests over secure connections. These bodies are interpreted by server applications that decide whether to send back content, repeat the password prompt or display an error message. These server applications don’t use the 401 status code that tells the browser to use its built-in authentication mechanism, though they may choose to make use of the 403 status code indicating that access to the requested resource is forbidden.

3.5.3.1 The Insecurity of the HTTP Authentication and How to solve it

Note that the user name and password are *encoded* but not *encrypted*. Encryption is a secure form of encoding, wherein the content can only be decoded if a unique key value is known. Simple encoding mechanisms, like the Base64 encoding used in basic HTTP authentication, can be decoded by anyone who knows the encoding scheme. Obviously this is very dangerous when encoded (not encrypted) information is transmitted over an insecure connection.

Secure connections (using extensions to the HTTP protocol like Secure HTTP) by definition *encrypt* all transmitted information, thus sensitive information (like passwords) is secure. It is hard to believe that there are still a large number of web sites—even ecommerce sites—that transmit passwords over open connections and establish secure connections only after the user has logged in!

As a user of the web, whenever you are prompted for your name and password, you should always check whether the connection is secure. With HTTP-based authentication, you should check whether the URL of the page you are attempting to access uses https (Secure HTTP) for its protocol. With proprietary authentication schemes, you should check the URL that is supposed to process your user name and password.

For example, with a forms-based login you should check the URL defined in the 'action' attribute. As a designer of applications for the Web, make sure that you incorporate these safeguards into your applications to ensure the security of users' sensitive information.

3.5.3.2 Dependent URLs

We say that one URL 'depends' on another URL if the portion of the second URL up to and including the last slash is a prefix of the first URL. For example, the URL http://www.cs.rutgers.edu/~shklar/classes/ depends on the URL http://www.cs.rutgers.edu/~shklar/. This means that, having submitted authorization credentials for http://www.cs.rutgers.edu/~shklar/, the browser would know to resubmit those same credentials within the Authorization header when requesting http://www.cs.rutgers.edu/~shklar/classes/.

3.5.4 Session support through Cookie and Set-Cookie headers

We've mentioned several times now that HTTP is a stateless protocol. So what do we do if we need to implement *stateful* applications?
To enable the maintenance of state between HTTP requests, it will suffice to provide some mechanism for the communicating parties to establish agreements for transferring state information in HTTP messages. HTTP/1.1 establishes these agreements through Set-Cookie and Cookie headers. Set-Cookie is a response header sent by the server to the browser, setting attributes that establish state within the browser.

Cookie is a request header transmitted by the browser in subsequent requests to the same (or related) server. It helps to associate requests with *sessions.* Server applications that want to provide 'hints' for processing future requests can do that by setting the Set-Cookie header:

```
Set-Cookie: <name>=<value>[; expires=<date>][; path=<path>]
[; domain=<domain name>][; secure]
```

Here, <name>=<value> is an *attribute/value pair* that is to be sent back by the browser in qualifying subsequent requests. The *path* and *domain* portions of this header delimit which requests qualify, by specifying the server domains and URL paths to which this cookie applies.

Browsers and other HTTP clients must maintain a ‘registry’ of cookies sent to them by servers. For cookies that are intended to last only for the duration of the current browser session, an in-memory table of cookies is sufficient. For cookies that are intended to last beyond the current session, persistent storage mechanisms for cookie information are required. Netscape Navigator keeps stored cookies in a cookies.txt file, while Internet Explorer maintains a folder where each file represents a particular stored cookie.

Domains may be set to suffixes of the originating server’s host name containing at least two periods (three for domains other than com, org, edu, gov, mil, and int). The value of the domain attribute must represent the same domain to which the server belongs. For example, an application running on cs.rutgers.edu can set the domain to .rutgers.edu, but not to .mit.edu. A domain value of .rutgers.edu means that this cookie applies to requests destined for hosts with names of the form *.rutgers.edu. The value for the path attribute defaults to the path of the URL of the request, but may be set to any path prefix beginning at ‘/’ which stands for the server root. For subsequent requests directed at URLs where the domain and path match, the browser must include a Cookie header with the appropriate attribute/value pair.

The *expires* portion of the header sets the cutoff date after which the browser will discard any attribute/value pairs set in this header. (If the cutoff date is not specified, this means that the cookie should last for the duration of the current browser session only.) Finally, the *secure* keyword tells the browser to pass this cookie only through secure connections.

In this example, a server application running on the cs.rutgers.edu server generates a Set-Cookie header of the following form:

```
HTTP/1.1 200 OK
Set-Cookie: name=Leon; path=/test/; domain=.rutgers.edu
```

The domain is set to .rutgers.edu and the path is set to /test/. This instructs the browser to include a Cookie header with the value Name=Leon every time thereafter that a request is made for a resource at a URL on any Rutgers server where the URL path starts with /test/. The absence of the expiration date means that this cookie will be maintained only for the duration of the current browser session.

Now let’s consider a more complicated example in which we rent a movie. We start with submitting a registration by visiting a URL that lets us sign in to a secure movie rental web site. Let’s assume we have been prompted for authorization credentials by the browser and have provided them so that the browser can construct the Authorization header:

```
GET /movies/register HTTP/1.1
Host: www.sample-movie-rental.com
Authorization:. . .
```

Once the server has recognized and authenticated the user, it sends back a response containing a Set-Cookie header containing a client identifier:

```
HTTP/1.1 200 OK
```
Set-Cookie: CLIENT=Rich; path=/movies
```
...
```

From this point on, every time the browser submits a request directed at "http://www.sample-movie.rental.com/movies/*", it will include a Cookie header containing the client identifier:

```
GET /movies/rent-recommended HTTP/1.1
Host: www.sample-movie-rental.com
```
Cookie: CLIENT=Rich

In this case, we are visiting a recommended movie page. The server response now contains a movie recommendation:

```
HTTP/1.1 200 OK
```
Set-Cookie: MOVIE=Matrix; path=/movies/
```
...
```

Now we request access to the movie. Note that, given the URL, we are sending back both the client identifier and the recommended movie identifier within the Cookie header.

```
GET /movies/access HTTP/1.1
Host: www.sample-movie-rental.com
```
Cookie: CLIENT=Rich; MOVIE=Matrix

We get back the acknowledgement containing access information to the recommended movie for future status checks:

```
HTTP/1.1 200 OK
```
Set-Cookie: CHANNEL=42; PASSWD=Matrix007; path=/movies/status/
```
...
```

Note that there are two new cookie values, ‘CHANNEL’ and ‘PASSWD’, but they are associated with URL path /movies/status/. Now, the browser will include movie access information with a status check request. Note that the Cookie header contains cookie values applicable to both the /movies/ path and the /movies/status/ path:

```
GET /movies/status/check HTTP/1.1
Host: www.sample-movie-rental.com
```
Cookie: CLIENT=Rich; MOVIE=Matrix; CHANNEL=42; PASSWD=Matrix007

Requests directed at URLs within the /movies/ path but not within the /movies/status/ path will not include attribute-value pairs associated with the /movies/status/ path:

```
GET /movies/access HTTP/1.1
Host: www.sample-movie-rental.com
```
Cookie: CLIENT=Rich; MOVIE=Matrix

3.6 EVOLUTION

HTTP has evolved a good deal since its inception in the early nineties, but the more it evolves, the more care is needed to support backward compatibility. Even though it has been a number of years since the introduction of HTTP/1.1, there are still many servers, browsers, and proxies in the real world that are HTTP/1.0 compliant but do not support HTTP/1.1. What's more, not all HTTP/1.1 programs revert to the HTTP/1.0 specification when they receive an HTTP/1.0 message.
In this section, we will discuss the reasoning behind some of the most important changes that occurred between the versions, and the compatibility issues that affected protocol designers' decisions, and the challenges facing Web application developers in dealing with these issues.

3.6.1 Virtual hosting

One of the challenges facing HTTP/1.1 designers was to provide support for *virtual hosting*, which is the ability to map multiple host names to a single IP address.
For example, a single server machine may host web sites associated with a number of different domains. There must be a way for the server to determine the host for which a request is intended. In addition, the introduction of proxies into the request stream creates additional problems in ensuring that a request reaches its intended host.

In HTTP/1.0, a request passing through a proxy has a slightly different format from the request ultimately received by the destination server. As we have seen, the request that reaches the host has the following form, including only the *path* portion of the URL in the initial request line:

```
GET /q?s=YHOO HTTP/1.0
```

Requests that must pass through proxies need to include some reference to the destination server, otherwise that information would be lost and the proxy would have no idea which server should receive the request. For this reason, the full URL of the request is included in the initial request line, as shown below:

```
GET http://finance.yahoo.com/q?s=YHOO HTTP/1.0
```

Proxies that connect to the destination servers are responsible for editing requests that pass through them, to remove server information from request lines.
With the advent of HTTP/1.1, there is support for virtual hosting. Thus, we now need to retain server information in all requests since servers need to know which of the virtual hosts associated with a given web server is responsible for processing the request. The obvious solution would have been to make HTTP/1.1 browsers and proxies to always include server information:

```
GET http://finance.yahoo.com/q?s=YHOO HTTP/1.1
```

This would have been fine except that there are still HTTP/1.0 proxies out there that are ready to cut server information from request URLs every time they see one.
Obviously, HTTP/1.0 proxies don't know anything about HTTP/1.1 and have no way of making a distinction between the two. Nonetheless, it is worth it to have this as a legal request format for both HTTP/1.1 servers and proxies.

For now, we need a redundant source of information that will not be affected by any actions of HTTP/1.0 proxies. This is the reason for the Host header, which must be included with every HTTP/1.1 request:

```
GET http://finance.yahoo.com/q?s=YHOO HTTP/1.1
Host: finance.yahoo.com
```

Whether this request passes through either an HTTP/1.0 proxy or an HTTP/1.1 proxy, information about the ultimate destination of the request is preserved. Obviously, the request with abbreviated URL format (path portion only) must be supported as well:

```
GET /q?s=YHOO HTTP/1.1
Host: finance.yahoo.com
```

3.6.2 Caching support

In an earlier section, we described the mechanisms through which servers provide information about caching policies for server responses to browsers, proxies, and other clients. If the supplied headers tell the client that caching is feasible for this particular response, the client must then make a decision as to whether or not it should indeed use a cached version of the response that it already has available, rather than going back to the source location to retrieve the data.

In HTTP/1.0, the most popular mechanism for supporting browser-side caching was the use of HEAD requests. A request employing the HEAD method would return exactly the same response as its GET counterpart, but without the body. In other words, only the headers would be present, providing the requestor with all of the response's metadata without the overhead of transmitting the entire content of the response. Thus, assuming you have a cached copy of a previously requested resource, it is a sensible approach to submit a HEAD request for that resource, check the date provided in the Last-Modified header, and only resubmit a GET request if the date is later than that of the saved cache entry. This improves server throughput by eliminating the need for unnecessary extra requests. The only time the actual data need actually be retrieved is when the cache entry is deemed to be out of date.

HTTP/1.1 uses a more streamlined approach to this problem using two new headers:
If-Modified-Since and If-Unmodified-Since. Going back to one of our earlier examples:

```
GET /~shklar/ HTTP/1.1
Host: www.cs.rutgers.edu
If-Modified-Since: Fri, 11 Feb 2001 22:28:00 GMT
```

Assuming there is a cache entry for this resource that expires at 22:28 on February 11, 2001, the browser can send a request for this resource with the If-Modified-Since header value set to that date and time. If the resource has *not* changed since that point in time, we get back the response with the 304 Not Modified status code and no body. Otherwise, we get back the body (which may itself be placed in the cache, replacing any existing cache entry for the same resource).

Alternatively, let's examine the following request:

```
GET /~shklar/ HTTP/1.1
Host: www.cs.rutgers.edu
If-Unmodified-Since: Fri, 11 Feb 2000 22:28:00 GMT
```

For this request, we either get back the unchanged resource or an empty response (no body) with the 412 Precondition Failed status code.
Both headers can be used in HTTP/1.1 requests to eliminate unnecessary data transmissions without the cost of extra requests.

3.6.3 Persistent connections

Since HTTP is by definition a *stateless* protocol, it was not designed to support persistent connections. A connection was supposed to last long enough for a browser to submit a request and receive a response. Extending the lifetime of a request beyond this was not supported.
Since the cost of connecting to a server across the network is considerable, there are a variety of mechanisms within many existing network protocols for reducing or eliminating that overhead by creating persistent connections. In HTTP, *cookies* provide a mechanism for persisting an application's *state* across connections, but it is frequently useful to allow connections themselves to persist for performance reasons.

For HTTP applications, developers came up with workarounds involving multipart MIME messages to get connections to persist across multiple independent bodies of content. (We saw an example of this when we discussed image animation using server push via multipart messages.)
However, late in the lifecycle of HTTP/1.0, makers of HTTP/1.0 servers and browsers introduced the proprietary Connection: Keep-Alive header, as part of a somewhat desperate effort to support persistent connections in a protocol that was not designed to do so.

Not surprisingly, it does not work that well. Considering all the intermediate proxies that might be involved in the transmission of a request, there are considerable difficulties in keeping connections persistent using this mechanism. Just one intermediate proxy that lacks support for the Keep-Alive extension is enough to cause the connection to be broken.

HTTP/1.1 connections are all persistent by default, except when explicitly requested by a participating program via the Connection: Close header. It is entirely legal for a server or a browser to be HTTP/1.1 compliant without supporting persistent connections as long as they include Connection: Close with every message. Theoretically, including the Connection: Keep-Alive header in HTTP/1.1 messages makes no sense, since the *absence* of Connection: Close already means that the connection needs to be persistent.

However, there is no way to ensure that all proxies are HTTP/1.1 compliant and know to maintain a persistent connection. In practice, including Connection: Keep-Alive does provide a partial solution: it will work for those HTTP/1.0 proxies that support it as a proprietary extension.

HTTP/1.1 support for persistent connections includes *pipelining* requests: browsers can queue request messages without waiting for responses. Servers are responsible for submitting responses to browser requests in the order of their arrival.

Browsers that support this functionality must maintain request queues, keep track of server responses, and resubmit requests that remain on queues if connections get dropped and re-established. We will discuss HTTP/1.1 support for persistent connections in further detail when we discuss server and browser architecture.

MODULE 4: WEB SERVERS

Web servers enable HTTP access to a 'Web site,' which is simply a collection of documents and other information organized into a tree structure, much like a computer's file system. In addition to providing access to static documents, modern Web servers implement a variety of protocols for passing requests to custom software applications that provide access to dynamic content. This chapter begins by describing the process of serving static documents, going on to explore the mechanisms used to serve dynamic data.

Dynamic content can come from a variety of sources. Search engines and databases can be queried to retrieve and present data that satisfies the selection criteria specified by a user. Measuring instruments can be probed to present their current readings (e.g. temperature, humidity). News feeds and wire services can provide access to up-to-the-minute headlines, stock quotes, and sports scores.

There are many methodologies for accessing dynamic data. The most prominent approach based on open standards is the *Common Gateway Interface* (CGI). While CGI is in widespread use throughout the Web, it has its limitations, which we discuss later in this chapter.

As a result, many alternatives to CGI have arisen. These include a number of proprietary template languages (some of which gained enough following to become *de facto* standards) such as *PHP, Cold Fusion*, Microsoft's *Active Server Pages* (ASP), and Sun's *Java Server Pages* (JSP), as well as Sun's *Java Servlet API*.

An ideal approach would allow the processes by which Web sites serve dynamic data to be established in a declarative fashion, so that those responsible for maintaining the site are not required to write custom code. This is an important thread in the evolution of Web servers, browsers and the HTTP protocol, but we have not yet reached this goal.

4.1 BASIC OPERATION

Web servers, browsers, and proxies communicate by exchanging HTTP messages. The server receives and interprets HTTP requests, locates and accesses requested resources, and generates responses, which it sends back to the originators of the requests. The process of interpreting incoming requests and generating outgoing responses is the main subject of this section.

In Figure 4.1, we can see how a Web server processes incoming requests, generates outgoing responses, and transmits those responses back to the appropriate requestors. The *Networking* module is responsible for both receiving requests and transmitting responses over the network. When it receives a request, it must first pass it to the *Address Resolution* module, which is responsible for analyzing and 'pre-processing' the request. This pre-processing includes:

1. *Virtual Hosting*: if this Web server is providing service for multiple domains, determine the domain for which this request is targeted, and use the detected domain to select configuration parameters.
2. *Address Mapping*: determine whether this is a request for static or dynamic content, based on the URL path and selected server configuration parameters, and resolve the address into an actual location within the server's file system.
3. *Authentication*: if the requested resource is protected, examine authorization credentials to see if the request is coming from an authorized user.

Once the pre-processing is complete, the request is passed to the *Request Processing* module, which invokes sub-modules to serve static or dynamic content as appropriate.

When the selected sub-module completes its processing, it passes results to the *Response Generation* module, which builds the response and directs it to the *Networking* module for transmission.

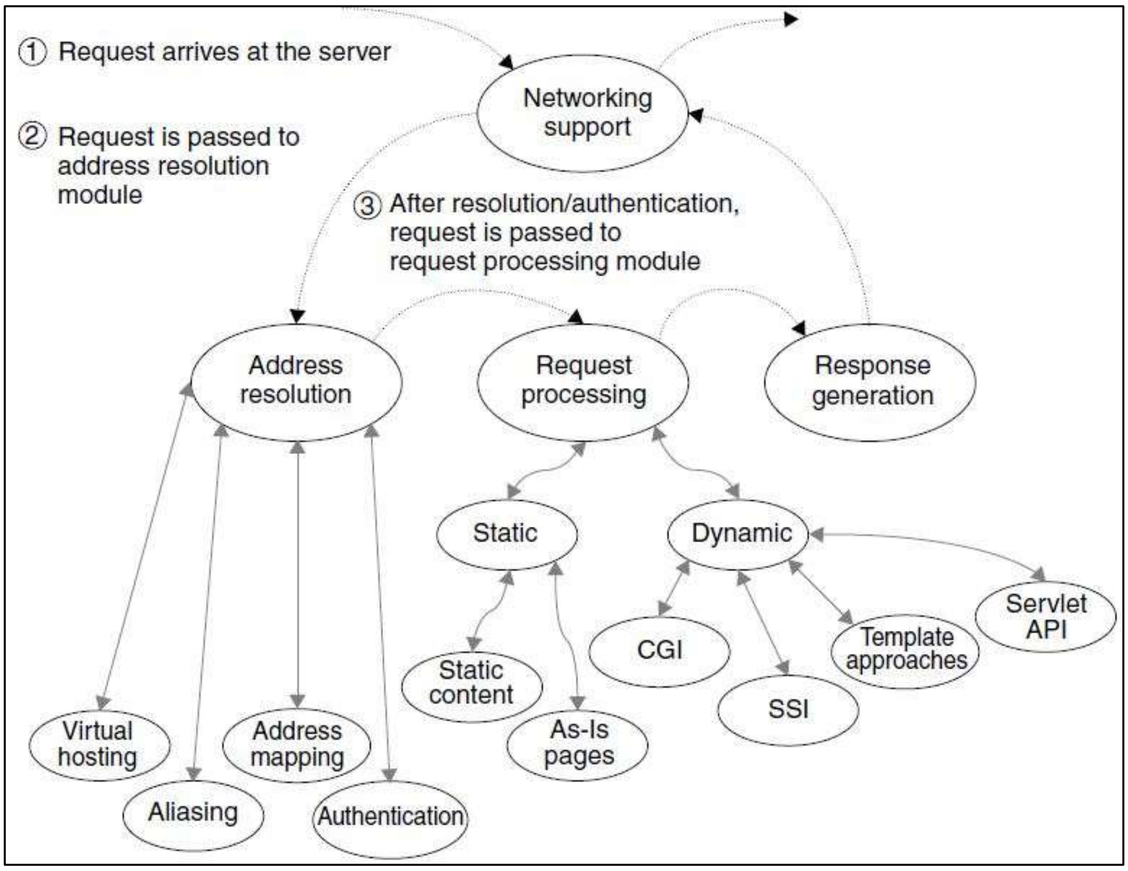


Figure 4.1: Server Operation

It is important to remember that, since the HTTP protocol is stateless, the only information available to the server about a request is that which is contained within that request. As we shall learn later in this chapter, state may be maintained in the form of session information by server-side applications and application environments (e.g. servlet runners), but the server itself does not maintain this information.

4.1.1 HTTP request processing

Let us take a step back and recollect what has to happen for an HTTP request to arrive at the server. For the purposes of this example, we shall examine a series of transactions in which an end-user is visiting her friend's personal web site found at `http://mysite.org/`.

The process begins when the end user tells the browser to access the page found at the URL `http://mysite.org/pages/simple-page.html`. When the browser successfully receives and renders the page (Example 1 and 2), the user sees that it has links to two other pages, which contain her friend's 'school links' (school.html) and 'home links' (home.html). Suppose now that the end user follows the link to her friend's 'school links' page.

Example 1: Browser request to load the *simple-page.html* page

```
GET http://mysite.org/pages/simple-page.html HTTP/1.1
Host: mysite.org
User-Agent: Mozilla/4.75 [en] (WinNT; U)
```

Example 2: Simple HTML page

```
<HTML>
<HEAD><TITLE>Simple Page</TITLE></HEAD>
<BODY>
<H2>My Links</H2>
<ul>
<li><a href ="school.html">My school links</a></li>
<li><a href ="home.html">My home links</a></li>
</ul>
</BODY>
</HTML>
```

Notice that the request in Example 4.2 does not contain the Connection: close header. This means that, if possible, the connection to the server should be left open so that it may be used to transmit further requests and receive responses. However, there is no guarantee that the connection will still be open at the time the user requests school.html. By that time, the server, a proxy, or even the browser itself might have broken it. Persistent connections are designed to improve performance, but should never be relied upon in application logic.
If the connection is still open, the browser uses it to submit the request for the school links page (Figure 4.4). Otherwise, the browser must first re-establish the connection. Depending on the browser configuration, it may either attempt to establish a direct connection to the server or connect via a proxy. Consequently, the server receives the request either directly from the browser or from the proxy.
For persistent connections, the server is responsible for maintaining queues of requests and responses. HTTP/1.1 specifies that within the context of a single continuously open connection, a series of requests may be transmitted. It also specifies that responses to these requests must be sent back in the order of request arrival (FIFO).
One common solution is for the server to maintain both the input and output queues of requests. When a request is submitted for processing, it is removed from the input queue and inserted on the output queue. Once the processing is complete, the request is marked for release, but it remains on the output queue while at least one of its predecessors is still there. When all of its predecessors have gone from the output queue it is released, and its associated response is sent back to the browser either directly or through a proxy.
Once the request is picked from the queue, the server resolves the request URL to the physical file location and checks whether the requested resource requires authentication (Example 4.1). If the authentication fails, the server aborts further processing and generates the response indicating an error condition (Section 3.4.3). If the authentication is not necessary or is successful, the server decides on the kind of processing required.

Example 4: Browser request to load the *school.html* page

```
GET http://mysite.org/pages/school.html HTTP/1.1
Host: mysite.org
User-Agent: Mozilla/4.75 [en] (WinNT; U)
```

4.1.2 Delivery of static content

Web servers present both *static content* and *dynamic content*. Static content falls into two categories:

1. *static content pages*: static files containing HTML pages, XML pages, plain text, images, etc., for which HTTP responses must be constructed (including headers); and

2. *as-is pages*: pages for which complete HTTP responses (including headers) already exist and can be presented 'as is'.

For dynamic content, the server must take an explicit programmatic action to generate a response, such as the execution of an application program, the inclusion of information from a secondary file, or the interpretation of a template. This mode of processing includes *Common Gateway Interface* (CGI) programs, *Server-Side Include* (SSI) pages, *Java Server Pages* (JSP), *Active Server Pages* (ASP), and Java Servlets, among others.

NOTE: We shall not attempt to describe all the details of these and other server mechanisms. Instead, we concentrate on describing the most common mechanisms and the underlying principles associated with them. Understanding the operating principles makes it easier to learn other similar mechanisms, and to develop new better mechanisms in the future.

Web servers use a combination of filename suffixes/extensions and URL prefixes to determine which processing mechanism should be used to generate a response.
By default, it is assumed that a URL should be processed as a request for a static content page. However, this is only one of a number of possibilities. A URL path beginning with /servlet/ might indicate that the target is a Java servlet. A URL path beginning with /cgi-bin/ might indicate that the target is a CGI script. A URL where the target filename ends in .cgi might indicate this as well. URLs where the target filename ends in .php or .cfm might indicate that a template processing mechanism (e.g. PHP or Cold Fusion) should be invoked. We shall discuss address resolution in more detail in the section describing server configuration.

Static content pages
For a *static content page*, the server maps the URL to a file location relative to the server document root. In the example presented earlier in the chapter, we visited a page on someone's personal web site, found at `http://mysite.org/pages/school.html`. The path portion of this URL, `/pages/school.html`, is mapped to an explicit filename within the server's local file system. If the Web server is configured so that the *document root* is /www/doc, then this URL will be mapped to a server file at `/www/doc/pages/school.html`.

Example 5: Sample response to the request in Example 4
```
HTTP/1.1 200 OK
Date: Tue, 29 May 2001 23:15:29 GMT
Last-Modified: Mon, 28 May 2001 15:11:01 GMT
Content-type: text/html
Content-length: 193
Server: Apache/1.2.5
<HTML>
<HEAD><TITLE>School Page</TITLE></HEAD>
<BODY>
<H2>My Links</H2>
<ul>
<li><a href ="classes.html">My classes</a></li>
<li><a href ="friends.html">My friends</a></li>
</ul>
</BODY>
</HTML>
```

For static pages, the server must retrieve the file, construct the response, and transmit it back to the browser. For persistent connections, the response first gets placed in the output queue before the transmission. Example 5 shows the response generated for the HTTP request in Example 4 As we discussed in the previous chapter, the first line of the response contains the status code that summarizes the result of the operation. The server controls browser rendering of the response through the Content-Type header that is set to a MIME type. Setting MIME types for static files is controlled through server configuration. In the simplest case, it is based on the mapping between file extensions and MIME types.

Even though desktop browsers have their own mappings between MIME types and file extensions, it is the server-side mapping that determines the Content-Type header of the response. It is this header, which determines how the browser should render the response content—*not* the filename suffix, or a browser heuristic based on content analysis.

With all this in mind, it is important that the server set the Content-Type header to the appropriate MIME type so that the browser can render the content properly. The server may also set the Content-Length header to instruct the browser about the length of the content. This header is optional, and may not be present for dynamic content, because it is difficult to determine the size of the response before its generation is complete. Still, if it *is* possible to predict the size of the response, it is a good idea to include the Content-Length header.

We have already discussed HTTP support for caching. Later we come back to this example to discuss the Last-Modified header and its use by the browser logic in forming requests and reusing locally cached content. Note that, even though Last-Modified is not a required header, the server is expected to make its best effort to determine the most recent modification date of requested content and use it to set the header.

As-is pages

Suppose now that for whatever reason you do not want server logic to be involved in forming response headers and the status code. Maybe you are testing your browser or proxy, or maybe you want a quick and dirty fix by setting a special Content-Type for some pages. Then again, maybe you want an easy and convenient way to regularly change redirection targets for certain pages. It turns out there is a way to address all these situations—use the so-called 'as-is' pages. The idea is that such pages contain complete responses and the server is supposed to send them back 'as-is' without adding status codes or headers. That means that you can put together a desired response, store it on the server in a file that would be recognized by the server as an 'as-is' file, and be guaranteed to receive your unchanged response when the file is requested.

Using the 'as-is' mechanism we can control server output by manually creating and modifying response messages. The word 'manually' is of course the key here—whenever you want to change the response, you have to have to go in and edit it. This is convenient for very simple scenarios but does not provide an opportunity to implement even very basic processing logic.

4.1.3 Delivery of dynamic content

The original mechanisms for serving up dynamic content are *CGI* (*Common Gateway Interface*) and *SSI* (*Server Side Includes*). Today's Web servers use more sophisticated and more efficient mechanisms for serving up dynamic content, but CGI and SSI date back to the very beginnings of the World Wide Web, and it behoves us to understand these mechanisms before delving into the workings of the newer approaches.

CGI **(*Common Gateway Interface*)**
CGI was the first consistent server-independent mechanism, dating back to the very early days of the World Wide Web. The original CGI specification can be found at http://hoohoo.ncsa.uiuc.edu/cgi/interface.html.
The CGI mechanism assumes that, when a request to execute a CGI script arrives at the server, a new ‘process’ will be ‘spawned’ to execute a particular application program, supplying that program with a specified set of parameters. (The terminology of ‘processes’ and ‘spawning’ is UNIX-specific, but the analogue of this functionality is available for non-UNIX operating systems).

The heart of the CGI specification is the designation of a fixed set of *environment variables* that all CGI applications know about and have access to. The server is supposed to use request information to populate variables in Table 4.1 (non-exhaustive list) from request information other than HTTP headers.

The server is responsible for always setting the SERVER SOFTWARE, SERVER NAME, and GATEWAY INTERFACE environment variables, independent of information contained in the request. Other pre-defined variable names include CONTENT TYPE and CONTENT LENGTH that are populated from Content-Type and Content-Length headers. Additionally, every HTTP header is mapped to an environment variable by converting all letters in the name of the header to upper case, replacing dash with the underscore, and pre-pending the HTTP prefix. For example, the value of the User-Agent header gets stored in the HTTP USER AGENT environment variable, and the value of the Content-Type header gets stored in both CONTENT TYPE and HTTP CONTENT TYPE environment variables.

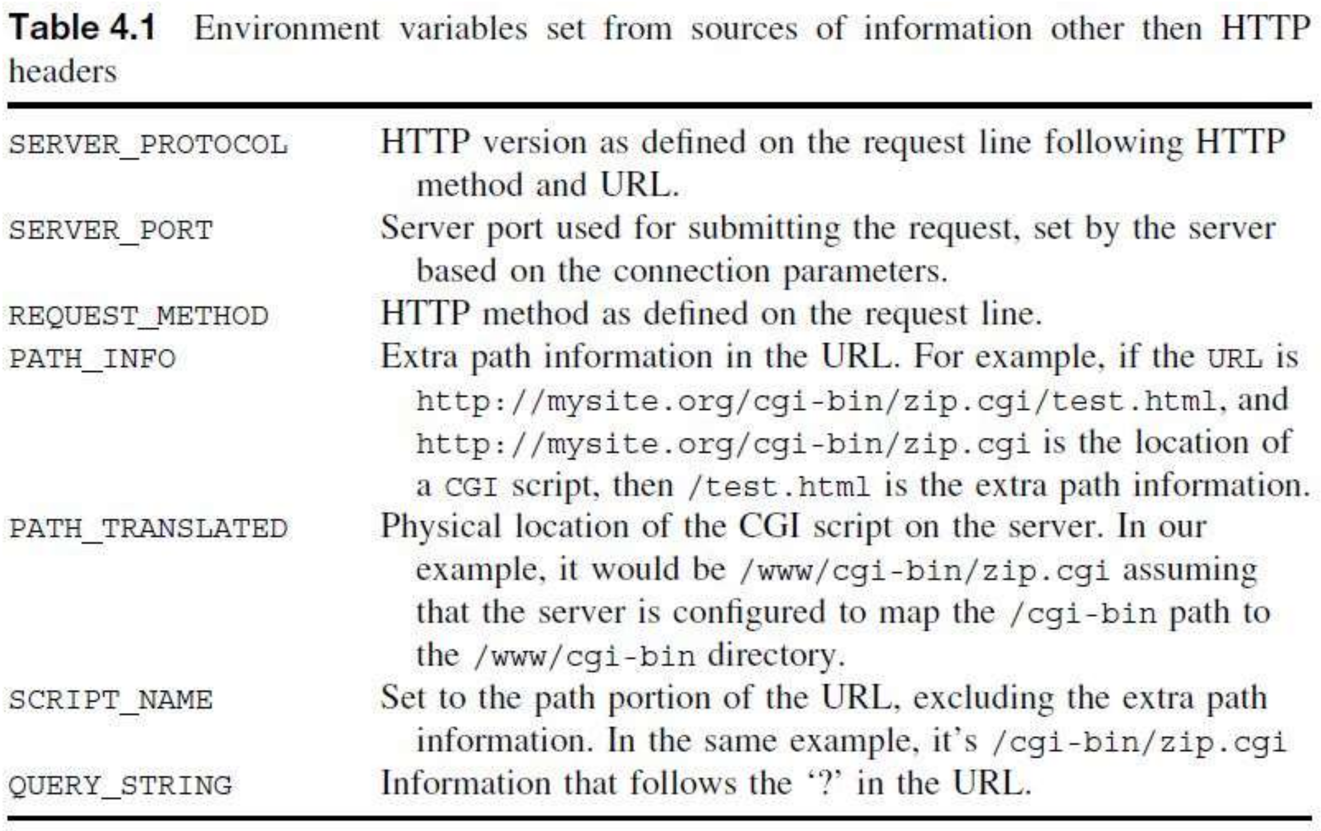


Table 4.1 Environment variables set from sources of information other then HTTP headers

`SERVER_PROTOCOL`	HTTP version as defined on the request line following HTTP method and URL.
`SERVER_PORT`	Server port used for submitting the request, set by the server based on the connection parameters.
`REQUEST_METHOD`	HTTP method as defined on the request line.
`PATH_INFO`	Extra path information in the URL. For example, if the `URL` is `http://mysite.org/cgi-bin/zip.cgi/test.html`, and `http://mysite.org/cgi-bin/zip.cgi` is the location of a `CGI` script, then `/test.html` is the extra path information.
`PATH_TRANSLATED`	Physical location of the CGI script on the server. In our example, it would be `/www/cgi-bin/zip.cgi` assuming that the server is configured to map the `/cgi-bin` path to the `/www/cgi-bin` directory.
`SCRIPT_NAME`	Set to the path portion of the URL, excluding the extra path information. In the same example, it’s `/cgi-bin/zip.cgi`
`QUERY_STRING`	Information that follows the ‘?’ in the URL.

It is important to remember that while names of HTTP headers do not depend upon the mechanism (CGI, servlets, etc.), names of the environment variables are specific to the CGI mechanism. Some early servlet runners expected names of the CGI environment variables to retrieve header values because their implementers were CGI programmers that did not make the distinction. These servlet runners performed internal name transformations according to the rules defined for the CGI mechanism (e.g. Content-Type to CONTENT TYPE), which was a wrong thing to do. These problems are long gone, and you would be hard pressed to find a servlet runner that does it now, but they illustrate the importance of understanding that names of CGI environment variables have no meaning outside of the CGI context.

The CGI mechanism was defined as a set of rules, so that programs that abide by these rules would run the same way on different types of HTTP servers and operating systems. That works as long as these servers support the CGI specification, which evolved to a suite of specifications for different operating systems. CGI was originally introduced for servers running on UNIX, where a CGI program always executes as a process with the body of the request available as standard input, and with HTTP headers, URL parameters, and the HTTP method available as environment variables. For Windows NT/2000 servers the CGI program runs as an *application process*. With these systems, there is no such thing as 'standard input' (as there is in a UNIX environment), so standard input is simulated using temporary files. Windows environment variables are similar to UNIX environment variables. Information passing details are different for other operating systems. For example, Macintosh computers pass system information through Apple Events. For simplicity, we use the CGI specification as it applies to UNIX servers for the rest of this section. You would do well to consult server documentation for information passing details for your Web server.

Since the CGI mechanism assumes spawning a new process per request and terminating this process when the request processing is complete, the lifetime of a CGI process is always a single request. This means that even processing that is common for all requests has to be repeated every time. CGI applications may sometimes make use of persistent storage that survives individual requests, but persistence is a non-standard additional service that is out of scope of what is guaranteed by HTTP servers.

Example 6 shows a simple HTML page containing a form that lets users specify their names and zip codes. The ACTION parameter of a FORM tag references a server application that can process form information. (It does not make sense to use forms if the ACTION references a static page!)
Example 7 shows the HTTP request that is submitted to the server when the user fills out the form and clicks on the 'submit' button. Notice that the entered information is *URL-encoded*: spaces are converted to plus signs, while other punctuation characters (e.g. equal signs and ampersands) are transformed to a percent sign ('%') followed by two-digit hexadecimal equivalents of replaced characters in the ASCII character set.
Also, notice that the Content-Type header is set to application/x-www-formurlencoded, telling the server and/or server applications to expect form data. Do not confuse the Content-Type of the response that caused the browser to render this page (in this case text/html) with the Content-Type associated with the request generated by the browser when it submits the form to the server. Form data submitted from an HTML page, WML page, or an applet would have the same Content-Type—application/x-www-form-urlencoded.

Example 6: Sample form for submitting user name and zip code

```
<HTML>
<HEAD><TITLE>Simple Form</TITLE></HEAD>
<BODY>
<H2>Simple Form</H2>
<FORM ACTION="http://mysite.org/cgi-bin/zip.cgi" METHOD="post">
Zip Code: <INPUT SIZE="5" NAME="zip">
Name: <INPUT SIZE="30" NAME="name">
<INPUT TYPE="submit" VALUE="set zip">
</FORM>
<BODY>
</HTML>
```

Example 7: HTTP request submitted by the browser for the zip code example in Figure 4.6

```
POST http://mysite.org/cgi-bin/zip.cgi HTTP/1.1
Host: mysite.org
User-Agent: Mozilla/4.75 [en] (WinNT; U)
Content-Length: 26
Content-Type: application/x-www-form-urlencoded
Remote-Address: 127.0.0.1
Remote-Host: demo-portable
zip=08540&name=Leon+Shklar
```

The server, having received the request, performs the following steps:

1. Determines that /cgi-bin/zip.cgi has to be treated as a CGI program. This decision may be based on either a configuration parameter declaring /cgi-bin to be a CGI directory, or the .cgi file extension that is mapped to the CGI processing module.
2. Translates /cgi-bin/zip.cgi to a server file system location based on the server configuration (e.g. /www/cgi-bin/zip.cgi).
3. Verifies that the computed file system location (/www/cgi-bin/) is legal for CGI executables.
4. Verifies that zip.cgi has *execute* permissions for the user id that is used to run the server (e.g. nobody). (This issue is relevant only on UNIX systems, where processes run under the auspices of a designated user id. This may not apply to non-UNIX systems).
5. Sets environment variables based on the request information.
6. Creates a child process responsible for executing the CGI program, passes it the body of the request in the standard input stream, and directs the standard output stream to the server module responsible for processing the response and sending it back to the browser.
7. On the termination of the CGI program, the response processor parses the response and, if missing, adds the default status code, default Content-Type, and headers that identify the server and server software.

To avoid errors in processing request parameters that may either be present in the query string of the request URL (for GET requests) or in the body of the request (for POST requests), CGI applications must decompose the ampersand-separated parameter string into URL-encoded name/value pairs prior to decoding them.

SSI mechanism (Server Side Includes)

SSI (Server Side Includes) are directives that are placed in HTML pages, and evaluated on the server while the pages are being served. They let you add dynamically generated content to an existing HTML page, without having to serve the entire page via a CGI program, or other dynamic technology.

For example, you might place a directive into an existing HTML page, such as:

```
<!--#echo var="DATE_LOCAL" -->
```

And, when the page is served, this fragment will be evaluated and replaced with its value:

```
Tuesday, 15-Jan-2013 19:28:54 EST
```

The decision of when to use SSI, and when to have your page entirely generated by some program, is usually a matter of how much of the page is static, and how much needs to be recalculated every time the page is served. SSI is a great way to add small pieces of information, such as the current time - shown above. But if a majority of your page is being generated at the time that it is served, you need to look for some other solution.

Configuring your server to permit SSI

To permit SSI on your server, you must have the following directive either in your `httpd.conf` file, or in a `.htaccess` file:

```
Options +Includes
```

This tells Server that you want to permit files to be parsed for SSI directives. Note that most configurations contain multiple `Options` directives that can override each other. You will probably need to apply the `Options` to the specific directory where you want SSI enabled in order to assure that it gets evaluated last.

Not just any file is parsed for SSI directives. You have to tell Apache which files should be parsed. There are two ways to do this. You can tell Apache to parse any file with a particular file extension, such as `.shtml`, with the following directives:

```
AddType text/html .shtml

AddOutputFilter INCLUDES .shtml
```

One disadvantage to this approach is that if you wanted to add SSI directives to an existing page, you would have to change the name of that page, and all links to that page, in order to give it a `.shtml` extension, so that those directives would be executed.

The other method is to use the `XBitHack` directive:

```
XBitHack on
```

XBitHack tells Apache to parse files for SSI directives if they have the execute bit set. So, to add SSI directives to an existing page, rather than having to change the file name, you would just need to make the file executable using `chmod`.

```
chmod +x pagename.html
```

A brief comment about what not to do. You'll occasionally see people recommending that you just tell Apache to parse all `.html` files for SSI, so that you don't have to mess with `.shtml` file names. These folks have perhaps not heard about XBitHack. The thing to keep in mind is that, by doing this, you're requiring that Apache read through every single file that it sends out to clients, even if they don't contain any SSI directives. This can slow things down quite a bit, and is not a good idea.

Of course, on Windows, there is no such thing as an execute bit to set, so that limits your options a little.

In its default configuration, Apache does not send the last modified date or content length HTTP headers on SSI pages, because these values are difficult to calculate for dynamic content. This can prevent your document from being cached, and result in slower perceived client performance. There are two ways to solve this:

1. Use the `XBitHack Full` configuration. This tells Apache to determine the last modified date by looking only at the date of the originally requested file, ignoring the modification date of any included files.
2. Use the directives provided by mod_expires to set an explicit expiration time on your files, thereby letting browsers and proxies know that it is acceptable to cache them.

Basic SSI directives

SSI directives have the following syntax:

```
<!--#function attribute=value attribute=value ... -->
```

It is formatted like an HTML comment, so if you don't have SSI correctly enabled, the browser will ignore it, but it will still be visible in the HTML source. If you have SSI correctly configured, the directive will be replaced with its results.

The function can be one of a number of things, and we'll talk some more about most of these in the next installment of this series. For now, here are some examples of what you can do with SSI

Today's date

```
<!--#echo var="DATE_LOCAL" -->
```

The `echo` function just spits out the value of a variable. There are a number of standard variables, which include the whole set of environment variables that are available to CGI programs. Also, you can define your own variables with the `set` function.

If you don't like the format in which the date gets printed, you can use the `config` function, with a `timefmt` attribute, to modify that formatting.

```
<!--#config timefmt="%A %B %d, %Y" -->
Today is <!--#echo var="DATE_LOCAL" -->
```

Modification date of the file

```
This document last modified <!--#flastmod file="index.html" -->
```

This function is also subject to `timefmt` format configurations.

Including the results of a CGI program

This is one of the more common uses of SSI - to output the results of a CGI program, such as everybody's favorite, a ``hit counter."

```
<!--#include virtual="/cgi-bin/counter.pl" -->
```

Additional examples

Following are some specific examples of things you can do in your HTML documents with SSI.

When was this document modified?

Earlier, we mentioned that you could use SSI to inform the user when the document was most recently modified. However, the actual method for doing that was left somewhat in question. The following code, placed in your HTML document, will put such a time stamp on your page. Of course, you will have to have SSI correctly enabled, as discussed above.

```
<!--#config timefmt="%A %B %d, %Y" -->
This file last modified <!--#flastmod file="ssi.shtml" -->
```

Of course, you will need to replace the `ssi.shtml` with the actual name of the file that you're referring to. This can be inconvenient if you're just looking for a generic piece of code that you can paste into any file, so you probably want to use the `LAST_MODIFIED` variable instead:

```
<!--#config timefmt="%D" -->
This file last modified <!--#echo var="LAST_MODIFIED" -->
```

For more details on the `timefmt` format, go to your favorite search site and look for `strftime`. The syntax is the same.

Including a standard footer

If you are managing any site that is more than a few pages, you may find that making changes to all those pages can be a real pain, particularly if you are trying to maintain some kind of standard look across all those pages.

Using an include file for a header and/or a footer can reduce the burden of these updates. You just have to make one footer file, and then include it into each page with the `include` SSI command. The `include` function can determine what file to include with either the `file` attribute, or the `virtual` attribute. The `file` attribute is a file path, *relative to the current directory*. That means that it cannot be an absolute file path (starting with /), nor can it contain ../ as part of that path. The `virtual` attribute is probably more useful, and should specify a URL relative to the document being served. It can start with a /, but must be on the same server as the file being served.

```
<!--#include virtual="/footer.html" -->
```

I'll frequently combine the last two things, putting a `LAST_MODIFIED` directive inside a footer file to be included. SSI directives can be contained in the included file, and includes can be nested - that is, the included file can include another file, and so on.

What else can I config?

In addition to being able to `config` the time format, you can also `config` two other things.

Usually, when something goes wrong with your SSI directive, you get the message

```
[an error occurred while processing this directive]
```

If you want to change that message to something else, you can do so with the `errmsg` attribute to the `config` function:

```
<!--#config errmsg="[It appears that you don't know how to use SSI]"
-->
```

Hopefully, end users will never see this message, because you will have resolved all the problems with your SSI directives before your site goes live. (Right?)

And you can `config` the format in which file sizes are returned with the `sizefmt` attribute. You can specify `bytes` for a full count in bytes, or `abbrev` for an abbreviated number in Kb or Mb, as appropriate.

Executing commands

Here's something else that you can do with the `exec` function. You can actually have SSI execute a command using the shell (`/bin/sh`, to be precise - or the DOS shell, if you're on Win32). The following, for example, will give you a directory listing.

```
<pre>
<!--#exec cmd="ls" -->
</pre>
```

or, on Windows

```
<pre>
<!--#exec cmd="dir" -->
</pre>
```

You might notice some strange formatting with this directive on Windows, because the output from `dir` contains the string ``<`dir`>" in it, which confuses browsers.

Note that this feature is exceedingly dangerous, as it will execute whatever code happens to be embedded in the `exec` tag. If you have any situation where users can edit content on your web pages, such as with a ``guestbook'', for example, make sure that you have this feature disabled. You can allow SSI, but not the `exec` feature, with the `IncludesNOEXEC` argument to the `Options` directive.

Advanced SSI techniques

In addition to spitting out content, Apache SSI gives you the option of setting variables, and using those variables in comparisons and conditionals.

Setting variables

Using the `set` directive, you can set variables for later use. We'll need this later in the discussion, so we'll talk about it here. The syntax of this is as follows:

```
<!--#set var="name" value="Rich" -->
```

In addition to merely setting values literally like that, you can use any other variable, including environment variables or the variables discussed above (like `LAST_MODIFIED`, for example) to give values to your variables. You will specify that something is a variable, rather than a literal string, by using the dollar sign ($) before the name of the variable.

```
<!--#set var="modified" value="$LAST_MODIFIED" -->
```

To put a literal dollar sign into the value of your variable, you need to escape the dollar sign with a backslash.

```
<!--#set var="cost" value="\$100" -->
```

Finally, if you want to put a variable in the midst of a longer string, and there's a chance that the name of the variable will run up against some other characters, and thus be confused with those characters, you can place the name of the variable in braces, to remove this confusion. (It's hard to come up with a really good example of this, but hopefully you'll get the point.)

```
<!--#set var="date" value="${DATE_LOCAL}_${DATE_GMT}" -->
```

Conditional expressions

Now that we have variables, and are able to set and compare their values, we can use them to express conditionals. This lets SSI be a tiny programming language of sorts. `mod_include` provides an `if`, `elif`, `else`, `endif` structure for building conditional statements. This allows you to effectively generate multiple logical pages out of one actual page.

The structure of this conditional construct is:

```
<!--#if expr="test_condition" -->
<!--#elif expr="test_condition" -->
<!--#else -->
<!--#endif -->
```

A *test_condition* can be any sort of logical comparison - either comparing values to one another, or testing the ``truth'' of a particular value. (A given string is true if it is nonempty.) For a full list of the comparison operators available to you, see the `mod_include` documentation.

For example, if you wish to customize the text on your web page based on the time of day, you could use the following recipe, placed in the HTML page:

```
Good <!--#if expr="%{TIME_HOUR} <12" -->
morning!
<!--#else -->
afternoon!
<!--#endif -->
```

Any other variable (either ones that you define, or normal environment variables) can be used in conditional statements. See Expressions in Apache HTTP Server for more information on the expression evaluation engine.

With Apache's ability to set environment variables with the `SetEnvIf` directives, and other related directives, this functionality can let you do a wide variety of dynamic content on the server side without resorting a full web application.

4.2 SERVER CONFIGURATION

Web server behavior is controlled by its configuration. While details of configuring a Web server differ greatly between different implementations, there are important common concepts that transcend server implementations. For example, any HTTP server has to be configured to map file extensions to MIME types, and any server has to be configured to resolve URLs to addresses in the local file system. For the purpose of this section, we make use of Apache configuration examples. Note that we refer to Apache configuration as a case study and have no intent of providing an Apache configuration manual, which is freely available from the Apache site anyway. Instead, we concentrate on the concepts and it remains your responsibility to map these concepts to configuring your servers.

4.2.1 Directory structure

An HTTP server installation directory is commonly referred to as the *server root*. Most often, other directories (document root, configuration directory, log directory, CGI and servlet root directories, etc.) are defined as subdirectories of the server root. There is normally the initial configuration file that gets loaded when the server comes up, which contains execution parameters, information about the location of other configuration files, and the location of most important directories. Configuration file formats vary for different servers—from traditional attribute-value pairs to XML formats.

There exist situations when it is desirable to depart from the convention that calls for the most important directories to be defined as subdirectories of the server root. For example, you may have reasons to run different servers interchangeably on the same machine, which is particularly common in a development environment.

In this case, you may want to use the same independently located document root for different servers. Similarly, you may need to be able to execute the same CGI scripts and servlets independent of which server is currently running. It is important to be particularly careful when sharing directories between different processes—it is enough for one of the processes to be insecure and the integrity of your directory structure is in jeopardy.

4.2.2 Execution

HTTP server is a set of processes or threads (for uniformity, we always refer to them as *threads*), some of which listen on designated ports while others are dedicated to processing incoming requests. Depending on the load, it may be reasonable to keep a number of threads running at all times so that they do not have to be started and initialized for every request.

Figure 4.15 contains a fragment of a sample configuration file for Apache installation on a Windows machine. The ‘standalone’ value of the server type indicates that the server process is always running and, as follows from the value of the ‘Port’ parameter, is listening on port 80. The server is configured to support persistent connections, and every connection is configured to support up to a hundred requests.

```
ServerName demo
ServerRoot "C:/Program Files/Apache Group/Apache"
ServerType standalone
Port 80
KeepAlive On
MaxKeepAliveRequests 100
KeepAliveTimeout 15
MaxRequestsPerChild 200
Timeout 300
```

Figure 4.15 Fragment of a sample configuration file

At the same time, the server is supposed to break the connection if 15 seconds go by without new requests. Many servers make it possible to impose a limit on a number of requests processed without restarting a child process. This limit was introduced for very pragmatic reasons—to avoid prolonged use that results in leaking memory or other resources.
The nature of the HTTP protocol, with its independent requests and responses makes it possible to avoid the problem simply by restarting the process. Finally, the timeout limits processing time for individual requests.

HTTP/1.1 is designed to support virtual hosting—the ability of a single server to accept requests targeted to different domains (this of course requires setting DNS aliases). It is quite useful when getting your favorite Internet Service Provider to host your site. As we have already discussed, this functionality is the reason for requiring the Host header in every request. Every virtual host may be configured separately. This does not apply to operational parameters that were discussed in this section. After all, different virtual hosts still share the same physical resources.

4.2.3 Address resolution

An HTTP request is an instruction to the server to perform specified actions. In fact, you may think of HTTP as a language, HTTP request as a program, and the server as an interpreter for the language. Requests are interpreted largely by specialized server modules and by server applications. For example, the servlet runner is responsible for interpreting session ids in Cookie headers and mapping them to server-side session information. Application logic is normally responsible for interpreting URL parameters, request bodies, and additional header information (e.g. Referer).
The core server logic is responsible for the initial processing and routing of requests. First and most important steps are to select the proper virtual host, resolve aliases, analyze the URL, and choose the proper processing module. In both sample URLs in Figure 4.16,

www.neurozen.com is a virtual host. The server has to locate configuration statements for this virtual host and use them to perform address translation.

```
1. http://www.neurozen.com/test?a=1&b2
2. http://www.neurozen.com/images/news.gif
<VirtualHost www.neurozen.com>
ServerAdmin webmaster@neurozen.com
Alias /test /servlet/test
Alias /images /static/images
DocumentRoot /www/docs/neurozen
ServerName www.neurozen.com
ErrorLog logs/neurozen-error-log
CustomLog logs/neurozen-access-log common
</VirtualHost>
```

Figure 4.16 Sample URLs and a configuration fragment

In the first URL, `/test` is defined to be an alias for `/servlet/test`. The server would first resolve aliases and only then use module mappings to pass the URL to the servlet runner that, in turn, invokes the *test* servlet. In the second URL, `/images` is defined to be an alias for `/static/images`, which is not explicitly mapped to a module and is assumed a static file. Consequently, the server translates `/static/images` to the path starting at the document root and looks up the image with the path `/www/docs/neurozen/static/news.gif`.
Syntax of configuration fragments in above examples is that of the Apache distribution. Do not get mislead by the presence of angle brackets—this syntax only resembles XML, perhaps it will evolve to proper XML in future versions.

Note that almost all configuration instructions may occur within the `VirtualHost` tags. The exception is configuration instructions that control execution parameters (Section 4.4.2). Instructions defined within the `VirtualHost` tag take precedence for respective host names over global instructions.

4.2.4 MIME support

Successful (200 OK) HTTP responses are supposed to contain the `Content-Type` header instructing browsers how to render enclosed bodies. For dynamic processing, responsibility for setting the `Content-Type` header to a proper MIME type is deferred to server applications that produce the response. For static processing, it remains the responsibility of the server. Servers set MIME types for static files based on file extensions. A server distribution normally contains a MIME configuration file that stores mappings between MIME types and file extensions.
In the example (Figure 4.17), `text/html` is mapped to two alternate file extensions (.`html` and .`htm`), `text/xml` is mapped to a single file extension (.`xml`), and `video/mpeg` is mapped to three alternate extensions (.`mpeg`, .`mpg`, and .`mpe`).

```
text/css css
text/html html htm
text/plain asc txt
text/xml xml
video/mpeg mpeg mpg mpe
```

Figure 4.17 Sample fragment of the Apache configuration file

There may be reasons for a particular installation to change or extend default mappings. Most servers provide for a way to do this without modifying the main MIME configuration file. For example, Apache supports special *add* and *update* directives that may be included with other global and virtual host-specific configuration instructions. The reason is to make it easy to replace default MIME mappings with newer versions without having to edit every new distribution. Such distributions are quite frequent, and are based on the work of standardization committees that are responsible for defining MIME types.

It is important to understand that MIME type mappings are not used exclusively for setting response headers. Another purpose is to aid the server in selecting processing modules. This is an alternative to path-based selections (Section 4.4.3). For example, a mapping may be defined to associate the .cgi extension with CGI scripts. Such a mapping means that the server would use the .cgi extension of the file name as defined in the URL to select CGI as the processing module. This does not change server behavior in performing path-based selections when MIME-based preferences do not apply. In the example, choosing CGI as the processing module does not have any effect on setting the Content-Type header, which remains the responsibility of the CGI script.

4.2.5 Server extensions

HTTP servers are packaged to support most common processing modules—As-Is, CGI, SSI, and servlet runners. Apache refers to these modules as *handlers*, and makes it possible not only to map built-in handlers to file extensions, but to define new handlers as well. In the example in Figure 4.18, the AddHandler directive is used to associate file instructions with handlers. The AddType directive is used to assign MIME types to the output of these handlers by associating both types and handlers with the same file extension. Further, the Action directive is designed to support introducing new handlers. In the example, the add-footer handler is defined as a Perl script that is supposed to be invoked for all .html files.

According to HTTP 1.0 and 1.1 specifications, the only required server methods are GET and HEAD. You would be hard pressed to find a widely used server that does not implement POST, but many of them do not implement other optional methods—PUT, DELETE, OPTIONS, TRACE, and CONNECT. The set of optional methods for a server may be extended but custom methods are bound to have proprietary semantics. The SCRIPT directive in the Apache example extends the server to support the PUT method by invoking the nph-put CGI program. As we discussed earlier (Section 4.1.3.1), the “nph-” prefix tells the server not to process the output of the CGI program.

```
AddHandler send-as-is .asis
AddType text /html .shtml
AddHandler server-parsed .shtml
Action add-footer /cgi-bin/footer.pl
AddHandler add-footer .html
Script PUT /cgi-bin/nph-put
```

Figure 4.18 Defining server extensions in Apache

4.3 SERVER SECURITY

Throughout the history of the human race, there has been a struggle between fear and greed. In the context of Internet programming, this tug of war takes the form of the struggle between server security and the amount of inconvenience to server administrators and application developers. Server security is about 80/20 compromises—attempts to achieve eighty percent

of desired security for your servers at the cost giving up twenty percent of convenience in building and maintaining applications.
Of course, there exist degenerate cases when no security is enough, but that is a separate discussion.
This section is not intended as a security manual, but rather as an overview of the most common security problems in setting up and configuring HTTP servers. We do not intend to provide all the answers, only to help you start looking for them. When it concerns security, being aware of a problem takes you more than half way to finding a solution.

4.3.1 Securing the installation

HTTP servers are designed to respond to external requests. Some of the requests may be malicious and jeopardize not only the integrity of the server, but of the entire network. Before we consider the steps necessary to minimize the effect of such malicious requests, we need to make sure that it is not possible to jeopardize the integrity of the server machine and corrupt the HTTP server installation.
The obvious precaution is to minimize remote login access to the server machine—up to disabling it completely (on UNIX that would mean disabling the *in.telnetd* and *in.logind* daemons). If this is too drastic a precaution for what you need, at least make sure that all attempts to access the system are monitored and logged, and all passwords are crack-resilient. Every additional process that is running on the same machine and serves outside requests adds to the risk—for example, *ftp* or *tftp*. In other words, it is better not to run any additional processes on the same machine. If you have to, at least make sure that they are secure. And do not neglect to check for obvious and trivial problems—like file permissions on configuration and password files. There are free and commercial packages that can aid you in auditing the file system and in checking for file corruption—clear indication of danger. After all, if the machine itself can be compromised, it does not matter how secure the HTTP server is that is running on that machine.

The HTTP server itself is definitely a source of danger. Back in the early days, when the URL string was limited to a hundred characters, everyone′s favorite way of getting through Web server defenses was to specify a long URL, and either achieves some sort of corruption, or force the server to execute instructions hidden in the trailing portions of these monster URLs. This is not likely to happen with newer HTTP servers but there are still gaping security holes that occasionally get exposed if server administrators are not careful.

As we already discussed in Section 4.1.3.2, SSI is fraught with dangers. Primarily, this is because it supports the execution of server-side programs. Subtler security holes may be exposed because of buggy parsing mechanisms that get confused when encountering illegal syntax—a variation on ancient monster URLs. In other words, you are really asking for trouble if your server is configured to support SSI pages in user directories. Similar precautions go for CGI scripts—enabling them in user directories is dangerous though not as much as SSI pages. At the risk of repeating ourselves—it is simple security oversights that cause most problems.

4.3.2 Dangerous practices

Speaking of the oversights, there are a few that seem obvious but get repeated over and over again. Quite a number of them have to do with the file system. We mentioned the file permissions, but another problem has to do with symbolic links between files and directories. Following a symbolic link may take the server outside the intended directory structure, often with unexpected results. Fortunately, HTTP servers make it possible to disable following links when processing HTTP requests.

Out of all different problems caused by lack of care in configuring the server, one that stands out has to do with sharing the same file system between different processes. How often do you see people providing both FTP and HTTP access to the same files? The problem is that you can spend as much effort as you want securing your HTTP server but it will not help if it is possible to establish an anonymous FTP connection to the host machine and post an executable in a CGI directory.
Now think of all different dangers of file and program corruption that may let outsiders execute their own programs on the server. It is bad enough that outside programs can be executed, but it is even worse if they can access critical system files. It stands to reason that an HTTP server should execute with permissions that don’t give it access to files outside of the server directory structure. This is why, if you look at server configuration files, you may notice that the user id defaults to ‘nobody’—the name traditionally reserved for user ids assigned to HTTP servers.

Unfortunately, not every operating system supports setting user ids when starting the server. Even less fortunately, system administrators, who log in with permissions that give them full access to system resources, are the ones to start the servers. As a result, the server process (and programs started through the server process) have full access to system resources. You know the consequences.

4.3.3 Secure HTTP

Let us assume for the time being that the server is safe. This is still not enough to guard sensitive applications (e.g. credit card purchases, etc.). Even if the server is safe, HTTP messages containing sensitive information are still vulnerable. The most obvious solution for guarding this information is, of course, encryption. HTTPS is the secure version of the HTTP protocol. All HTTPS messages are the same except that they are transmitted over a Secure Socket Layer (SSL) connection—messages are encrypted before the transmission and decrypted after being received by the server.

The SSL protocol supports the use of a variety of different cryptographic algorithms to authenticate the server and the browser to each other, transmit certificates, and establish session encryption keys. The SSL handshake protocol determines how the server and the browser negotiate what encryption algorithm to use. Normally, the server and the browser would select the strongest possible algorithm supported by both parties. Very secure servers may disable weaker algorithms (e.g. those based on 40-bit encryption). This can be a problem when you try to access your bank account and the server refuses the connection asking you to install a browser that supports 128-bit encryption.

As always, you may spend a lot of effort and get bitten by a simple oversight. Even now, after so many years of Internet commerce, you can find applications that all have the same problem—they secure the connection after authenticating a user but authentication is not performed over a secure connection, which exposes user names and passwords. Next time, before you fill out a form to login to a site that makes use of your sensitive information, check whether the action attribute on that form references an HTTPS URL. If it does not—you should run away and never come back.

4.3.4 Firewalls and proxies

Today, more than a third of all Internet sites are protected by firewalls. The idea is to isolate machines on a Local Area Network (LAN) and expose them to the outside world via a specialized gateway that screens network traffic. This gateway is what is customarily referred to as a *firewall*.

Firewall configurations
There exist many different firewall configurations that fall into two major categories: *dual-homed* gateways and *screened-host* gateways. Dual-homed firewall is a computer with two different interface cards, one of which is connected to the LAN and one to the outside world. With this architecture, there is no direct contact between the LAN and the world, so it is necessary to run a firewall proxy on the gateway machine and make this proxy responsible for filtering network packets and passing them between the interface cards. Passing every packet requires an explicit effort, and no information is passed if the firewall proxy is down. Such configuration is very restrictive and is used only in very secure installations.

Screened-host gateways are network routers that have the responsibility of filtering traffic between the LAN and the outside world. They may be configured to screen network packets based on source and destination addresses, ports, and other criteria. Normally, the router is configured to pass through only network packets that are bound for the firewall host and stops packets that are bound for other machines on the LAN. The firewall host is responsible for running a configurable filtering proxy that selectively passes through the network traffic. The screened-host configuration is very flexible—it is easy to open temporary paths to selected ports and hosts. This comes in handy when you need to show a demo running on an internal machine. HTTP proxies It is all well and good to create a firewall but what do you do if you need to make your HTTP server visible to the outside world? The seemingly easy answer—running it on the firewall machine—is not a good one. First, any serious load on the HTTP server that is running on the firewall machine may bring the LAN's connection to the outside world to its knees. After all, the HTTP server and network traffic filters would share the same resources. Secondly, any security breach that exposes the HTTP server could also expose the firewall machine and consequently the entire LAN. At the risk of repeating ourselves—it is a really bad idea to run the HTTP server on the firewall machine, and the reason why we keep repeating it over and over again is because people do it anyway.

An alternative is to take advantage of the flexibility of screened-host gateways and allow network traffic to an internal machine when directed to a certain port (e.g. 80). It is much less dangerous than running the server on the firewall machine but still fraught with problems since you are exposing an unprotected machine albeit in a very limited way. Additionally, this approach has functional limitations—how would you redirect the request to another server running on a different port or on a different machine?

It turns out there exists another solution. Let us go back and think about the reasons why it is not a good idea to run an HTTP server on the firewall machine. The first reason is processing load and the second reason is security. What if we limited the functionality of the HTTP server that is running on the firewall machine, to make it defer processing to machines inside the firewall? This would solve the problem with processing load. How about security? Well, if the HTTP server is not performing any processing on the firewall machine, and passes requests along to aninternal machine on the LAN, it is hard to break into this server. The simpler the functionality, the harder it is for malicious outsiders to break in.

This sounds good, but what we are doing is simply passing requests along to another machine that still has to process these requests. Can malicious outsiders break into *that* machine? Well, not so easily—even if they manage to wreak havoc on the HTTP server machine, they cannot access that machine directly and use it as a staging ground for further penetration.
To summarize, the solution is not to run a full-fledged HTTP server on the firewall machine but to replace it with an HTTP proxy that may be configured to screen HTTP requests and forward

them to proper internal hosts. Different proxy configurations may be selected depending on a wide range of circumstances but what is important is that no processing is performed on the firewall host and the internal machines are not exposed directly to the outside world.

MODULE 5: OPEN SYSTEM & INTEROPERABILITY

Open Systems Architecture—An Overview

A constantly changing technology landscape and expectations to adapt and innovate quickly make it challenging to acquire, integrate, and upgrade fielded systems, especially in the current economic environment. Highly integrated systems are often proprietary and vendor locked, expensive, and difficult to upgrade with emerging technology. A strategy for overcoming these challenges is to design highly interoperable systems. This means enabling systems or components to exchange services and information through seamless, end-to-end connectivity.

OSA is an integrated business and technical approach to acquire and assemble interoperable components using modular systems design. The business strategy is to drive down costs, enable systems to easily adapt to changing business needs, and increase the number of available vendors to create competition-driven product lines. The technical approach decomposes systems into components that interact through key interfaces according to formal specifications.

Interoperability

Interoperability is the ability of computer systems or software to exchange and make use of information. For example: "interoperability between devices made by different manufacturers”.
It is a characteristic of a product or system, whose interfaces are completely understood, to work with other products or systems, at present or in the future, in either implementation or access, without any restrictions.
Interoperability is used to describe the capability of different programs to exchange data via a common set of exchange formats, to read and write the same file formats, and to use the same protocols. (The ability to execute the same binary code on different processor platforms is 'not' contemplated by the definition of interoperability.) The lack of interoperability can be a consequence of a lack of attention to standardization during the design of a program. Indeed, interoperability is not taken for granted in the non-standards-based portion of the computing world.

OSA aims to enhance interoperability by realizing the following benefits:

Increased flexibility in vendor selection fostered by competitive marketplaces
- Interchangeable components to simplify maintenance, upgrades, and technology insertion
- Greater accessibility to innovative technology
- Shortened design times and streamlined development processes
- Improved information sharing and data quality
- Reduced total cost of ownership.

OSA applies to all types of systems. Although some of its most familiar uses are in computers, software, and electronics, this approach applies to other areas, such as communications, electricity production and use, and the design of weapons, vehicles, and artillery for armed forces. Computer networks are tightly integrated systems that employ standard hardware, such as cables, routers, and servers. This hardware uses standard protocol to enable devices and machines to communicate and exchange information. With constantly changing operational needs for new weapons and armor, the armed forces use a similar approach to enhance interoperability in military vehicles. By developing standard electronic platforms and mounting systems, military vehicles can quickly access electronic and information assets and introduce new weapon and sensor capabilities.

Components, Interfaces, and Standards
OSA abstracts systems into three key elements: components, interfaces, and standards. Figure 1 describes these elements and their characteristics. These are generic to any system and span several dimensions of interoperability, including technical, informational, and organizational. This involves the physical connections and communications between components, their information flows, and relationships between people and organizations.

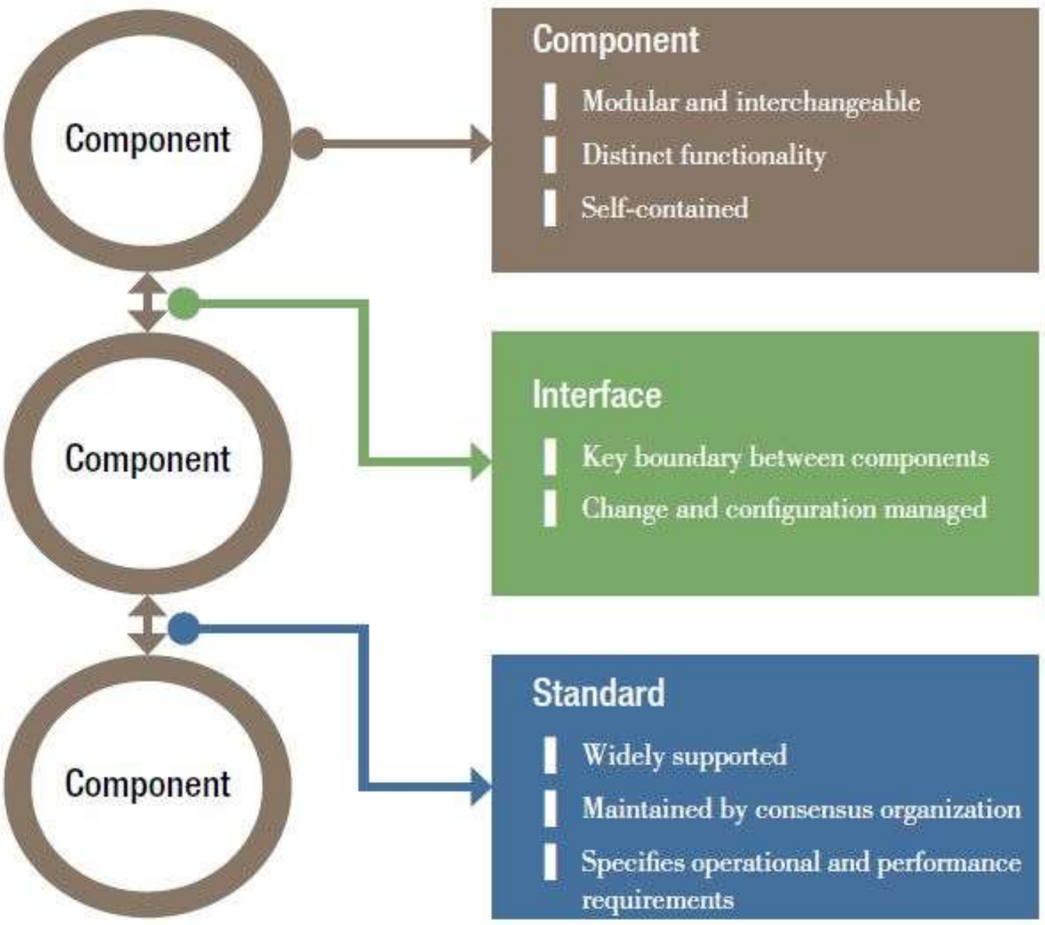

Component
Component
Component
Component
Modular and interchangeable
Distinct functionality
Self-contained
Interface
Key boundary between components
Change and configuration managed
Standard
Widely supported
Maintained by consensus organization
Specifies operational and performance requirements

Figure 5.1: Elements of Open Systems Architecture

Components are the physical modules of a system. Each component has distinct functionality and operates independently and with limited impact on the rest of the system. This decouples the components from each other and makes it possible to interchange units provided by alternate vendors. The desired functionality and inner workings of components may also vary across vendors.

Interfaces define the key boundaries between components and how they interact. The types of interfaces depend on the physical connections, information, or services the components need to exchange. Interfaces should be standardized, change and configuration managed, and publicly available.

Standards define the specifications for how components interact through defined interfaces. These include operational and performance requirements, such as security, reliability, and maintainability, that describe how an interface should perform. Standards should be managed by consensus groups and widely accepted to ensure they meet the requirements across all systems.

Design Principles to Ensure Interoperability

OSA considers interoperability through the entire life cycle of a system. A program must design its systems to be interoperable from the time it acquires and defines its components, interfaces, and standards through the time when those systems become operational and eventually are decommissioned. Several critical success factors contribute to successfully implementing OSA principles:

1. Firm commitments and well-defined governance
2. Available, reliable, and economical components
3. Controlled interfaces
4. Mature standards.

1. Firm Commitments and Well-Defined Governance

Interoperability requires cooperation. The programs and involved systems should be dedicated to an enterprise-wide strategy to implement and realize the benefits of OSA. This includes developing a strategic sourcing approach for acquiring system components, contributing to the ongoing development of open standards to meet business and system requirements, and providing guidance and oversight to align systems to OSA principles.

Political and financial support from program offices, project managers, and senior managers who understand the long-term benefits of OSA are crucial to its successful implementation. A program implementing an OSA system should establish enterprise governance through policy, guidance, and enterprise planning to develop and maintain its systems. Interdisciplinary practices, such as systems engineering and enterprise architecture, enable organizations to manage system complexity and align resources with an OSA strategy.

Organizations should establish governing bodies supported by communities of interest and working groups or committees to oversee design and implementation, champion enterprise-wide adoption, and assess benefits realization. Governing bodies should make funding and approval decisions for systems to proceed through key life-cycle milestones.

2. Available, Reliable, and Economical Components

OSA focuses on decomposing systems into modular components. In order for these components to be interoperable, easily upgraded, and maintained, there must be a broad range of components that meet the functional and performance requirements of a system. The specifications for components must be formal and publicly available to encourage broad commercial support. This will allow a number of vendors to produce the same or similar components with standardized functionality. This will also promote competition between vendors to produce usable, reliable, and economical components and to incentivize productivity and innovation.

3. Controlled Interfaces

Controlled and consistent interfaces enhance the interoperability of components. Interfaces should be controlled, monitored, and published to clearly and fully define all inputs and outputs of a component. Interfaces separate the functionality of each component and define the requirements that interface standards need to support. By monitoring the number of interfaces within a system, their rate of change, and their conformance with standards, a program will be able to assess the openness, interoperability, and affordability of a system over time.

4. Mature Standards

To mitigate the risks associated with enhancing interoperability, systems should use standards that are well-developed and stable and that have achieved widespread adoption by industry. This will ensure that interfaces meet current industry-wide operational and performance requirements, adapt to changes due to emerging technology or innovation, and are published. Programs should participate in standards development to ensure that their adopted standards continue to meet their business and technical requirements.

Standards organizations often manage the overall production and evolution of mature standards among a wide base of adopters. These organizations benefit from collaborative participation from industry, universities, and government to develop robust and comprehensive interface specifications. Well-known standards organizations such as the ISO, International Electro-technical Commission, and International Telecommunication Union have developed standards for all types of interfaces, including physical, data, network, and applications.

These standards support various OSA-based approaches, such as the Open Systems Interconnection (OSI) model and service-oriented architectures. Consistent with an OSA approach, OSI decomposes communications systems into functional layers where components within each layer interact through well-defined protocols. Similarly, service-oriented architectures separate software systems into loosely coupled pieces of software that communicate using standard web-based services and that can be published and discovered. In both cases, mature standards enable interoperable machine-to-machine interaction over a network.

Achieving Software interoperability

Software interoperability is achieved through five interrelated ways:

1. Product testing
   Products produced to a common standard, or to a sub-profile thereof, depend on the clarity of the standards, but there may be discrepancies in their implementations that system or unit testing may not uncover. This requires that systems formally be tested in a production scenario – as they will be finally implemented – to ensure they actually will intercommunicate as advertised, i.e. they are interoperable. Interoperable product testing is different from conformance-based product testing as conformance to a standard does not necessarily engender interoperability with another product which is also tested for conformance.
2. Product engineering
   Implements the common standard, or a sub-profile thereof, as defined by the industry/community partnerships with the specific intention of achieving interoperability with other software implementations also following the same standard or sub-profile thereof.
3. Industry/community partnership
   Industry-community partnerships, either domestic or international, sponsor standard workgroups with the purpose to define a common standard that may be used to allow software systems to intercommunicate for a defined purpose. At times, an industry/community will sub-profile an existing standard produced by another organization to reduce options and thus making interoperability more achievable for implementations.
4. Common technology and IP
   The use of a common technology or IP may speed up and reduce the complexity of interoperability by reducing variability between components from different sets of separately developed software products and thus allowing them to intercommunicate more readily. This technique has some of the same technical results as using a common vendor product to produce interoperability. The common technology can come through 3rd party libraries or open-source developments.
5. Standard implementation
   Software interoperability requires a common agreement that is normally arrived at via an industrial, national or international standard.

Each of these has an important role in reducing variability in intercommunication software and enhancing a common understanding of the end goal to be achieved.

Socket Programming

Socket programming is a way of connecting two nodes on a network to communicate with each other. One socket(node) listens on a particular port at an IP, while other socket reaches out to the other to form a connection. Server forms the listener socket while client reaches out to the server.

In other words, a *socket* is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent to.

Socket programs are used to communicate between various processes usually running on different systems. It is mostly used to create a client-server environment. It is an endpoint is a combination of an IP address and a port number. Every TCP connection can be uniquely identified by its two endpoints. That way you can have multiple connections between your host and the server.

Types of Sockets:

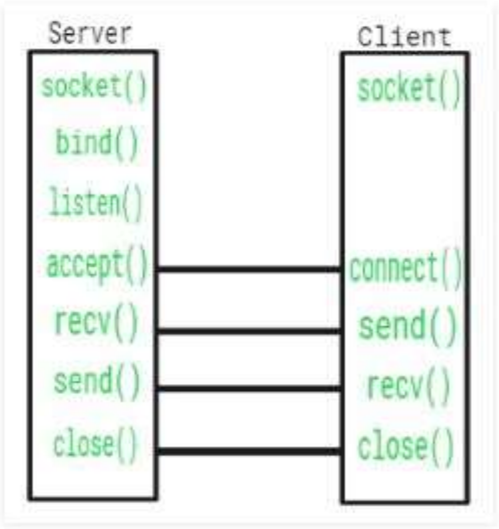
There are two types of Sockets: the **datagram** socket and the **stream** socket

1. **Datagram Socket**

   This is a type of network which has connection less point for sending and receiving packets. It is similar to mailbox. The letters (data) posted into the box are collected and delivered (transmitted) to a letterbox (receiving socket).

2. **Stream Socket**

   In Computer operating system, a stream socket is type of inter process communications socket or network socket which provides a connection-oriented, sequenced, and unique flow of data without record boundaries with well-defined mechanisms for creating and destroying connections and for detecting errors. It is similar to phone. A connection is established between the phones (two ends) and a conversation (transfer of data) takes place.



Function Call	Description
Create()	To create a socket
Bind()	It's a socket identification like a telephone number to contact
Listen()	Ready to receive a connection
Connect()	Ready to act as a sender

Function Call	Description
Accept()	Confirmation, it is like accepting to receive a call from a sender
Write()	To send data
Read()	To receive data
Close()	To close a connection

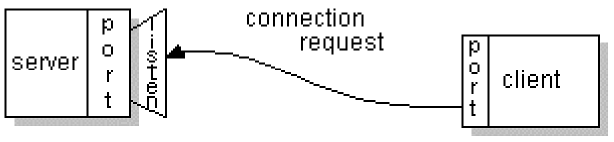
Implementing Sockets

For java, the java.net package in the Java platform provides a class, Socket, that implements one side of a two-way connection between your Java program and another program on the network. The Socket class sits on top of a platform-dependent implementation, hiding the details of any particular system from your Java program. By using the java.net.Socket class instead of relying on native code, your Java programs can communicate over the network in a platform-independent fashion. Additionally, java.net includes the ServerSocket class, which implements a socket that servers can use to listen for and accept connections to clients. This lesson shows you how to use the Socket and ServerSocket classes.

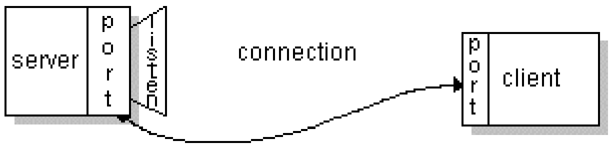
If you are trying to connect to the Web, the URL class and related classes (URLConnection, URLEncoder) are probably more appropriate than the socket classes. In fact, URLs are a relatively high-level connection to the Web and use sockets as part of the underlying implementation. See Working with URLs for information about connecting to the Web via URLs

Normally, a server runs on a specific computer and has a socket that is bound to a specific port number. The server just waits, listening to the socket for a client to make a connection request.

On the client-side: The client knows the hostname of the machine on which the server is running and the port number on which the server is listening. To make a connection request, the client tries to rendezvous with the server on the server's machine and port. The client also needs to identify itself to the server so it binds to a local port number that it will use during this connection. This is usually assigned by the system.



If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket bound to the same local port and also has its remote endpoint set to the address and port of the client. It needs a new socket so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client.



On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server. The client and server can now communicate by writing to or reading from their sockets.

Client Process

This is the process, which typically makes a request for information. After getting the response, this process may terminate or may do some other processing.

Example, Internet Browser works as a client application, which sends a request to the Web Server to get one HTML webpage.

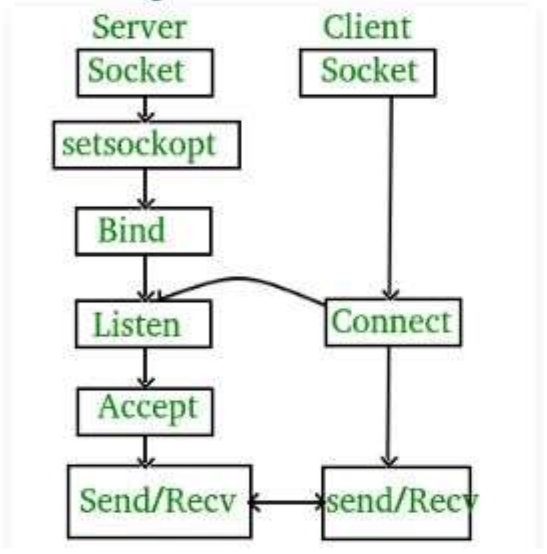
Server Process

This is the process which takes a request from the clients. After getting a request from the client, this process will perform the required processing, gather the requested information, and send it to the requestor client. Once done, it becomes ready to serve another client. Server processes are always alert and ready to serve incoming requests.

Example − Web Server keeps waiting for requests from Internet Browsers and as soon as it gets any request from a browser, it picks up a requested HTML page and sends it back to that Browser.

Note that the client needs to know the address of the server, but the server does not need to know the address or even the existence of the client prior to the connection being established. Once a connection is established, both sides can send and receive information.

State diagram for server and client model



2-tier and 3-tier architectures

There are two types of client-server architectures −

- **2-tier architecture** − In this architecture, the client directly interacts with the server. This type of architecture may have some security holes and performance problems. Internet Explorer and Web Server work on two-tier architecture. Here security problems are resolved using Secure Socket Layer (SSL).
- **3-tier architectures** − In this architecture, one more software sits in between the client and the server. This middle software is called 'middleware'. Middleware are used to perform all the security checks and load balancing in case of heavy load. A middleware takes all requests from the client and after performing the required authentication, it passes that request to the server. Then the server does the required processing and sends the response back to the middleware and finally the middleware passes this response back to the client. If you want to implement a 3-tier architecture, then you can keep any middleware like Web Logic or WebSphere software in between your Web Server and Web Browser.

Types of Server

There are two types of servers you can have −

- **Iterative Server** − This is the simplest form of server where a server process serves one client and after completing the first request, it takes request from another client. Meanwhile, another client keeps waiting.
- **Concurrent Servers** − This type of server runs multiple concurrent processes to serve many requests at a time because one process may take longer and another client cannot wait for so long. The simplest way to write a concurrent server under Unix is to *fork* a child process to handle each client separately.

How to Make Client

The system calls for establishing a connection are somewhat different for the client and the server, but both involve the basic construct of a socket. Both the processes establish their own sockets.

The steps involved in establishing a socket on the client side are as follows −

- Create a socket with the **socket()** system call.
- Connect the socket to the address of the server using the **connect()** system call.
- Send and receive data. There are a number of ways to do this, but the simplest way is to use the **read()** and **write()** system calls.

How to make a Server

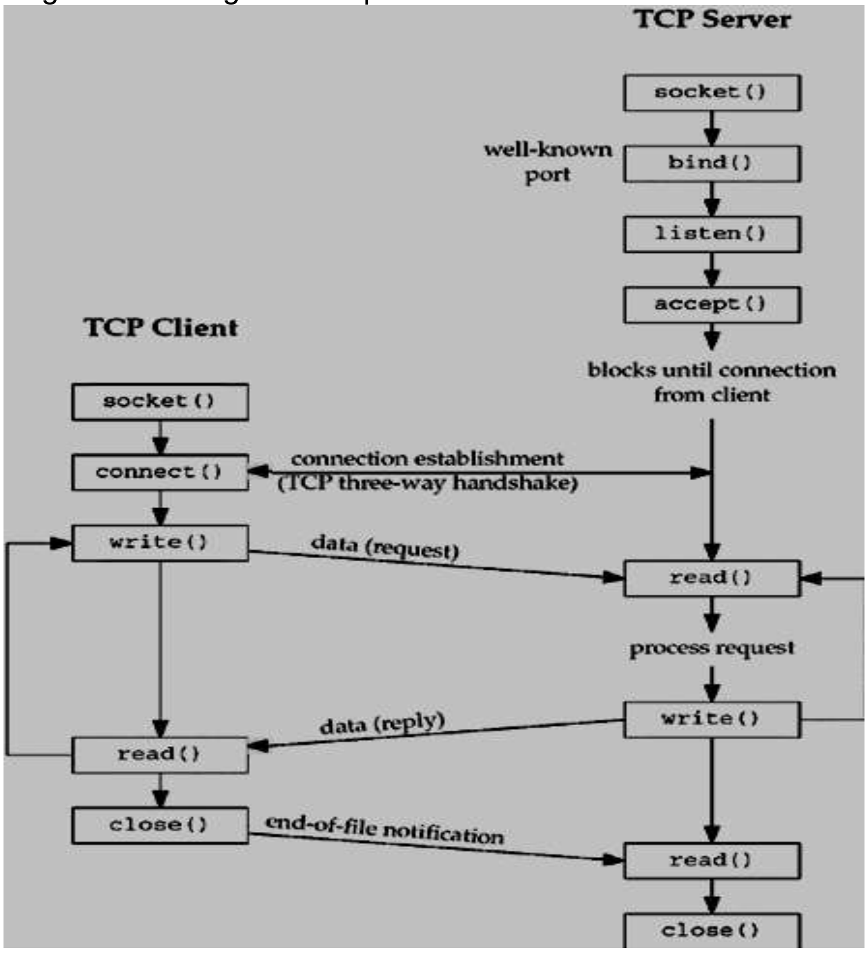
The steps involved in establishing a socket on the server side are as follows −

- Create a socket with the **socket()** system call.
- Bind the socket to an address using the **bind()** system call. For a server socket on the Internet, an address consists of a port number on the host machine.

- Listen for connections with the **listen()** system call.
- Accept a connection with the **accept()** system call. This call typically blocks the connection until a client connects with the server.
- Send and receive data using the **read()** and **write()** system calls.

Client and Server Interaction

Following is the diagram showing the complete Client and Server interaction −



Complete Client and Server interaction Diagram

Various structures are used in Unix Socket Programming to hold information about the address and port, and other information. Most socket functions require a pointer to a socket address structure as an argument. Structures defined in this chapter are related to Internet Protocol Family.

sockaddr

The first structure is *sockaddr* that holds the socket information −

```
struct sockaddr {
   unsigned short   sa_family;
   char             sa_data[14];
};
```

This is a generic socket address structure, which will be passed in most of the socket function calls. The following table provides a description of the member fields −

This is a generic socket address structure, which will be passed in most of the socket function calls. The following table provides a description of the member fields −

Attribute	Values	Description
sa_family	AF_INET AF_UNIX AF_NS AF_IMPLINK	It represents an address family. In most of the Internet-based applications, we use AF_INET.
sa_data	Protocol-specific Address	The content of the 14 bytes of protocol specific address are interpreted according to the type of address. For the Internet family, we will use port number IP address, which is represented by *sockaddr_in* structure defined below.

sockaddr in

The second structure that helps you to reference to the socket's elements is as follows −

```
struct sockaddr_in {
   short int            sin_family;
   unsigned short int   sin_port;
   struct in_addr       sin_addr;
   unsigned char        sin_zero[8];
};
```

Here is the description of the member fields −

Attribute	Values	Description
sa_family	AF_INET AF_UNIX AF_NS AF_IMPLINK	It represents an address family. In most of the Internet-based applications, we use AF_INET.
sin_port	Service Port	A 16-bit port number in Network Byte Order.
sin_addr	IP Address	A 32-bit IP address in Network Byte Order.

sin_zero	Not Used	You just set this value to NULL as this is not being used.

in addr

This structure is used only in the above structure as a structure field and holds 32 bit netid/hostid.

```
struct in_addr {
  unsigned long s_addr;
};
```

Here is the description of the member fields −

Attribute	Values	Description
s_addr	service port	A 32-bit IP address in Network Byte Order.

hostent

This structure is used to keep information related to host.

```
struct hostent {
  char *h_name;
  char **h_aliases;
  int h_addrtype;
  int h_length;
  char **h_addr_list

#define h_addr  h_addr_list[0]
};
```

Here is the description of the member fields −

Attribute	Values	Description
h_name	ti.com etc.	It is the official name of the host. For example, tutorialspoint.com, google.com, etc.
h_aliases	TI	It holds a list of host name aliases.

h_addrtype	AF_INET	It contains the address family and in case of Internet based application, it will always be AF_INET.
h_length	4	It holds the length of the IP address, which is 4 for Internet Address.
h_addr_list	in_addr	For Internet addresses, the array of pointers h_addr_list[0], h_addr_list[1], and so on, are points to structure in_addr.

NOTE − h_addr is defined as h_addr_list[0] to keep backward compatibility.

servent

This particular structure is used to keep information related to service and associated ports.

```
struct servent {
   char  *s_name;
   char  **s_aliases;
   int   s_port;
   char  *s_proto;
};
```

Here is the description of the member fields −

Attribute	**Values**	**Description**
s_name	http	This is the official name of the service. For example, SMTP, FTP POP3, etc.
s_aliases	ALIAS	It holds the list of service aliases. Most of the time this will be set to NULL.
s_port	80	It will have associated port number. For example, for HTTP, this will be 80.
s_proto	TCP UDP	It is set to the protocol used. Internet services are provided using either TCP or UDP.

Tips on Socket Structures

Socket address structures are an integral part of every network program. We allocate them, fill them in, and pass pointers to them to various socket functions. Sometimes we pass a pointer to one of these structures to a socket function and it fills in the contents.

We always pass these structures by reference (i.e., we pass a pointer to the structure, not the structure itself), and we always pass the size of the structure as another argument.

When a socket function fills in a structure, the length is also passed by reference, so that its value can be updated by the function. We call these value-result arguments.

Always, set the structure variables to NULL (i.e., '\0') by using memset() for bzero() functions, otherwise it may get unexpected junk values in your structure.

When a client process wants to a connect a server, the client must have a way of identifying the server that it wants to connect. If the client knows the 32-bit Internet address of the host on which the server resides, it can contact that host. But how does the client identify the particular server process running on that host?

To resolve the problem of identifying a particular server process running on a host, both TCP and UDP have defined a group of well-known ports.

For our purpose, a port will be defined as an integer number between 1024 and 65535. This is because all port numbers smaller than 1024 are considered *well-known* -- for example, telnet uses port 23, http uses 80, ftp uses 21, and so on.

The port assignments to network services can be found in the file /etc/services. If you are writing your own server then care must be taken to assign a port to your server. You should make sure that this port should not be assigned to any other server.

Normally it is a practice to assign any port number more than 5000. But there are many organizations who have written servers having port numbers more than 5000. For example, Yahoo Messenger runs on 5050, SIP Server runs on 5060, etc

Example of Ports and Services

Here is a small list of services and associated ports. You can find the most updated list of internet ports and associated service at IANA - TCP/IP Port Assignments.

Service	**Port Number**	**Service Description**
echo	7	UDP/TCP sends back what it receives.
discard	9	UDP/TCP throws away input.
daytime	13	UDP/TCP returns ASCII time.
chargen	19	UDP/TCP returns characters.
ftp	21	TCP file transfer.
telnet	23	TCP remote login.

smtp	25	TCP email.
daytime	37	UDP/TCP returns binary time.
tftp	69	UDP trivial file transfer.
finger	79	TCP info on users.
http	80	TCP World Wide Web.
login	513	TCP remote login.
who	513	UDP different info on users.
Xserver	6000	TCP X windows (N.B. >1023).

Port and Service Functions

Unix provides the following functions to fetch service name from the /etc/services file.

- **struct servent *getservbyname(char *name, char *proto)** − This call takes service name and protocol name, and returns the corresponding port number for that service.
- **struct servent *getservbyport(int port, char *proto)** − This call takes port number and protocol name, and returns the corresponding service name.

The return value for each function is a pointer to a structure with the following form −

```
struct servent {
   char  *s_name;
   char  **s_aliases;
   int   s_port;
   char  *s_proto;
};
```

Here is the description of the member fields −

Attribute	Values	Description
s_name	http	It is the official name of the service. For example, SMTP, FTP POP3,

		etc.
s_aliases	ALIAS	It holds the list of service aliases. Most of the time, it will be set to NULL.
s_port	80	It will have the associated port number. For example, for HTTP, it will be 80.
s_proto	TCP UDP	It is set to the protocol used. Internet services are provided using either TCP or UDP.

Application Programming Interface

An API (Application Programming Interface) is a set of functions that allows applications to access data and interact with external software components, operating systems, or microservices.

To simplify, an API delivers a user response to a system and sends the system's response back to a user. You click "add to cart;" an API tells the site you added a product to your cart; the website puts the product in your cart, and your cart is updated.

You may hear the term "microservices" come up in relation to API. These however, are not the same. Microservices are a style or architecture which divides functionality within a web application. While API is the framework which developers interact with a web application. Microservices can actually use API to communicate between each other.

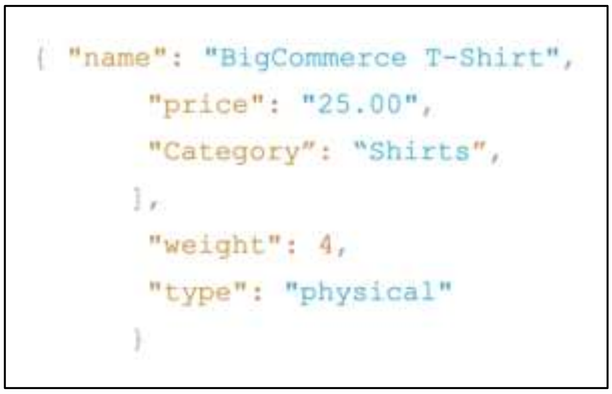
API lets a developer make a specific "call" or "request" in order to send or receive information. This communication is done using a programming language called "JSON." It can also be used to make a defined action such as updating or deleting data. There are four basic request methods that can be made with API:

1. GET – Gathers information (Pulling all Coupon Codes)
2. PUT – Updates pieces of data (Updating Product pricing)
3. POST – Creates (Creating a new Product Category)
4. DELETE – (Deleting a blog post)

You've probably heard the terms API, Public API, or Web API before. These are often used by software companies when speaking about an application, operating system or website. They are used everywhere in today's world and offer a tremendous benefit. But have you ever wondered what an API actually is, or how to use it?

So What is JSON and why is it used?

JSON (JavaScript Object Notation) is used to represent data on a server. It’s fairly easy to read by humans, and easy for machines/applications to understand. Let’s look at an example of JSON from a product on BigCommerce:



```
{ "name": "BigCommerce T-Shirt",
      "price": "25.00",
      "Category”: “Shirts",
     ],
      "weight": 4,
      "type": "physical"
     }
```

This is easy to understand as it’s outputted in key/value pairs, with the key on the left, and a value on the right. Keys are a fixed object defined by the application and will remain the same as with “category.” Whereas the values will be unique, such as “Shirts.”

What is an API Request?

There are several components of an API Request in order for it to function. Let’s go over these individually and how they can be used to build a request.

Endpoint
There are two key parts to an endpoint that are used when making an API request. One of which is the URL. BigCommerce uses https://api.bigcommerce.com/stores/ as the URL for all API Requests. This may look like a regular URL but if you plug this into a web browser, you will receive a 404 error message.

The second part is the path. The path will vary depending on what you are trying to accomplish. You can find a list of available paths for BigCommerce by visiting our developer documentation: https://developer.bigcommerce.com/api-reference. For this example we are going to use the product path which is /v3/catalog/products.

When we put these two parts together, we get a complete endpoint https://api.bigcommerce.com/stores/{store_hash}/v3/catalog/products. Now you may be saying to yourself “What is the {store_hash}?“ “Where did that come from?” This is what is known as a variable. Variables are unique components to an endpoint and will vary depending on your store’s information. You can spot a variable by the open and closed brackets “{ }”.

Header
Headers provide information to the client and server. Common examples of a header would be authentication credentials such as a “Auth Token” or “Client ID”. These credentials are provided to you automatically when you create an API Account. Another common header is referred to as the “Content Type,” which informs the server about what type of content will be

sent. For example, a commonly used content type is “application/json” which let’s the server know, we are sending JSON data across.

Method

Methods are the actions taken when sending a request. Think back to the beginning when we discussed GET, PUT, POST, and DELETE. These are all API Methods.

Data

The request data, also commonly referred to as the “body,” is information that will be either sent to or returned by a server. In the previous discussion of JSON, you can see an example of API data. The body of a request will sometimes require specific information before it can be delivered. An example of this is if you are editing a single product, the Product ID will be required before any change can be made.

What about REST & SOAP APIs?

While API follows a specific set of rules that determine how programs communicate with one another. REST & SOAP define how the API is presented. Each are similar in functionality but have several key differences and use cases.

REST stands for “Representational State Transfer” and is the set of rules that developers follow when creating an API. REST is read using JSON as we covered previously. One of these rules is that the API should be designed in a way that is easy to use and will make sense for developers. An example of not following this rule would be to have the product endpoint “prod_839” instead of just “products.” As this could cause the API to be fairly unpleasant to work with.

SOAP or Simple Object Access Protocol is another design modal for web services. Instead of the typical JSON that REST API uses. SOAP uses a language known as Extensible Markup Language (XML). XML is designed to be machine- and human-readable. SOAP follows a strong standard of rules, such as messaging structure and convention for providing request or responses.

Everyday Examples of APIs

API helps developers quickly deliver information to consumers and is used every day in today’s world. From shopping online, browsing a social media app, or playing a game on your smartphone. Every time you visit a page online, you’re interacting with API. Here are some real-world examples of how you interact with API and may not even realize.

Going to a bank

Think of yourself as a user and a bank teller as an API, while the bank is the system you interact with. When you want to take some money out of your account, you walk up to the teller(API) and say “I’d like $1,000 from this account”. The teller (API) then goes to the back, tells the bank manager (the system) “Mr/Ms.X would like $1,000”, the bank manager (the system) gives the teller (API) $1,000 who eventually gives it to you. As you can see the API, is a messenger between your needs and the system.

Searching for hotels

When you go onto a travel site, it may be linked to 10 other travel sites to find the best deal for you. When you input details like Atlanta, 2 nights, 1 Room, you send this request to those 10 different travel sites. The API takes your request for that specific location, date range, and room and pings the 10 sites, who send back the deals they have found. You look through the 10 deals and pick the best one. Again, the API is a messenger for your requests.

Finding a Facebook profile

Looking for someone on Facebook? Thanks to APIs, you can do it easily! If you type in "John Smith" on Facebook, the API informs Facebook's servers that you're looking for John Smith. Facebook then sends you a list of all the profiles that match that name (with factors like vicinity to you, or mutual friends). Now you can find John Smith!

Finding a new restaurant

Let's say you are traveling to a new city or state. You've just dropped everything off at the hotel and decide to grab some lunch. You grab your smart phone and look up restaurants nearby. Quickly you are shown dozens of local restaurants right outside your hotel. Thanks to Google Maps API; they are able to easily display business hours, reviews, phone numbers, and even times they are likely to be busy.

Why Modern Ecommerce Sites Use APIs

API's offer a wide range of benefits for Ecommerce Sites. They can help consumers easily find products, grow a company's brand, or even expand their earning potential by selling products on various marketplaces such as eBay, Amazon, and Facebook. Listed below are some benefits of why API is so important to ecommerce sites today.

Security

Security is enhanced when sites use APIs. Whenever you send a request, you aren't directly linked to a server. You send small amounts of information, the API delivers it, and the server sends it back. This minimizes the risk of a breach or someone accessing the backend of a server.

Speed

Without APIs, you would have to call a store and ask them to look at their inventory from all their suppliers, which they would eventually get back to you. This, instead of having an API where you could easily see what a product was, the price, or it's stock level.

Scalability

APIs allow scalability and flexibility when expanding your store's catalog, security, or data needs. Your store can grow at a faster rate when you don't have to factor in new code for every single product or user.

What are some types of API used?

There are 3 types of APIs used commonly today:

1. Open API
2. Partner API
3. Private API

Open APIs are publically available for anyone to use. For example, BigCommerce website uses roughly 25 different APIs, which is available for the public to use.

Partner APIs are designed by companies to offer API access to strategic business partners as an extra revenue channel for both parties. For example, Ticketmaster offers a Partner API to allow it's clients the ability to reserve, buy, and retrieve ticket/event information.

Private APIs are not designed for public use and are designed for internal use. Let's say you are traveling to a different city for a business meeting. You need to make a quick trip to the bank. You walk into "ABC Bank" and give the teller your account number. She quickly pulls up your account and you make a withdrawal. The teller was able to pull up your information by using ABC's internal system, which uses an API to pull your account information and to update your new account balance.

Common Ecommerce APIs

Ecommerce APIs are used in many ways. From displaying products on an online store to shipping them all over the world. APIs help owners manage their online business and connect with customers fast and reliably.

Product information APIs

Product information APIs are on every ecommerce site, grabbing the information about your products and serving it to customers.

Site search APIs

The ability to site search isn't automatic. Site searches need APIs to search through all your products containing a certain query and retrieve it for your user.

Payment APIs

If your online shop collects any form of electronic payments, you are using a payment API as the middleman between your shop and your processor.

Shipping APIs

Ever been to a site that asks you to put in your zip code to calculate shipping? That site is using an API with its shipping system or carrier to get you your best rate.

Currency conversion APIs

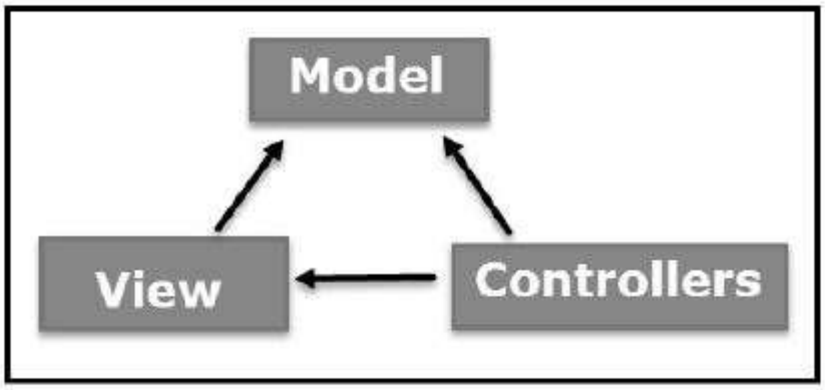
Buying shirts on a British site from a US IP used to be hard, now with currency conversion APIs, your favorite international stores can convert currency in an instant. This API opens hundreds of thousands of online shops to international customers.

Module 6: MODEL VIEW CONTROLLER

The **Model-View-Controller (MVC)** is an architectural pattern that separates an application into three main logical components: the **model**, the view, and the controller. Each of these components are built to handle specific development aspects of an application. MVC is one of the most frequently used industry-standard web development framework to create scalable and extensible projects.

MVC Components

Following are the components of MVC −



High-level architecture flow of MVC Framework

Model

The Model component corresponds to all the data-related logic that the user works with. This can represent either the data that is being transferred between the View and Controller components or any other business logic-related data. For example, a Customer object will retrieve the customer information from the database, manipulate it and update it data back to the database or use it to render data.

View

The View component is used for all the UI logic of the application. For example, the Customer view will include all the UI components such as text boxes, dropdowns, etc. that the final user interacts with.

Controller

Controllers act as an interface between Model and View components to process all the business logic and incoming requests, manipulate data using the Model component and interact with the Views to render the final output. For example, the Customer controller will handle all the interactions and inputs from the Customer View and update the database using the Customer Model. The same controller will be used to view the Customer data.

ASP.NET MVC Features

ASP.NET MVC provides the following features −

- Ideal for developing complex but lightweight applications.
- Provides an extensible and pluggable framework, which can be easily replaced and customized. For example, if you do not wish to use the in-built Razor or ASPX View

Engine, then you can use any other third-party view engines or even customize the existing ones.

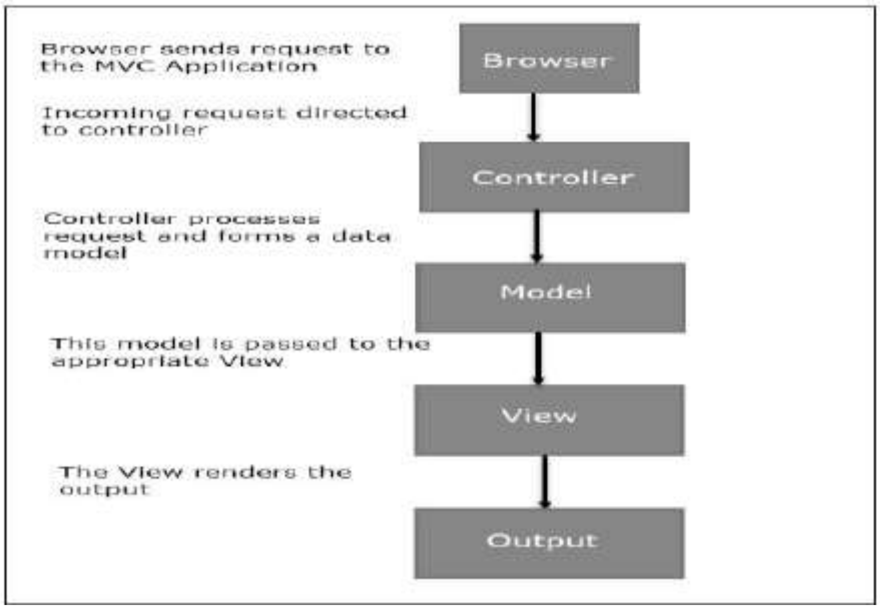
- Utilizes the component-based design of the application by logically dividing it into Model, View, and Controller components. This enables the developers to manage the complexity of large-scale projects and work on individual components.
- MVC structure enhances the test-driven development and testability of the application, since all the components can be designed interface-based and tested using mock objects. Hence, ASP.NET MVC Framework is ideal for projects with large team of web developers.
- Supports all the existing vast ASP.NET functionalities, such as Authorization and Authentication, Master Pages, Data Binding, User Controls, Memberships, ASP.NET Routing, etc.
- Does not use the concept of View State (which is present in ASP.NET). This helps in building applications, which are lightweight and gives full control to the developers.

Thus, you can consider MVC Framework as a major framework built on top of ASP.NET providing a large set of added functionality focusing on component-based development and testing.

ASP.NET MVC

ASP.NET supports three major development models: Web Pages, Web Forms and MVC (Model View Controller). ASP.NET MVC framework is a lightweight, highly testable presentation framework that is integrated with the existing ASP.NET features, such as master pages, authentication, etc. Within .NET, this framework is defined in the System.Web.Mvc assembly. The latest version of the MVC Framework is 5.0. We use Visual Studio to create ASP.NET MVC applications which can be added as a template in Visual Studio.

MVC Flow Diagram


Browser sends request to the MVC Application
Browser
Incoming request directed to controller
Controller
Controller processes request and forms a data model
Model
This model is passed to the appropriate View
View
The View renders the output
Output

Flow Steps

Step 1 − The client browser sends request to the MVC Application.

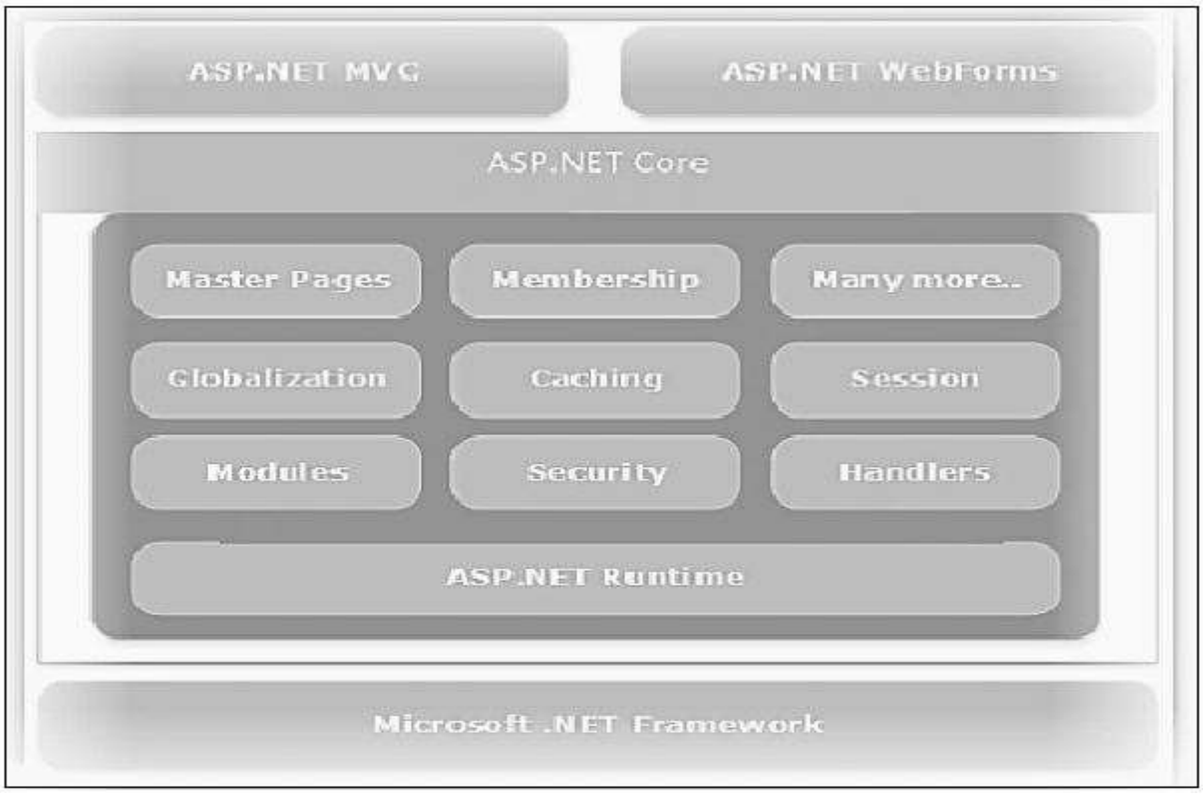
Step 2 − Global.ascx receives this request and performs routing based on the URL of the incoming request using the RouteTable, RouteData, UrlRoutingModule and MvcRouteHandler objects.

Step 3 − This routing operation calls the appropriate controller and executes it using the IControllerFactory object and MvcHandler object's Execute method.

Step 4 − The Controller processes the data using Model and invokes the appropriate method using ControllerActionInvoker object

Step 5 − The processed Model is then passed to the View, which in turn renders the final output.

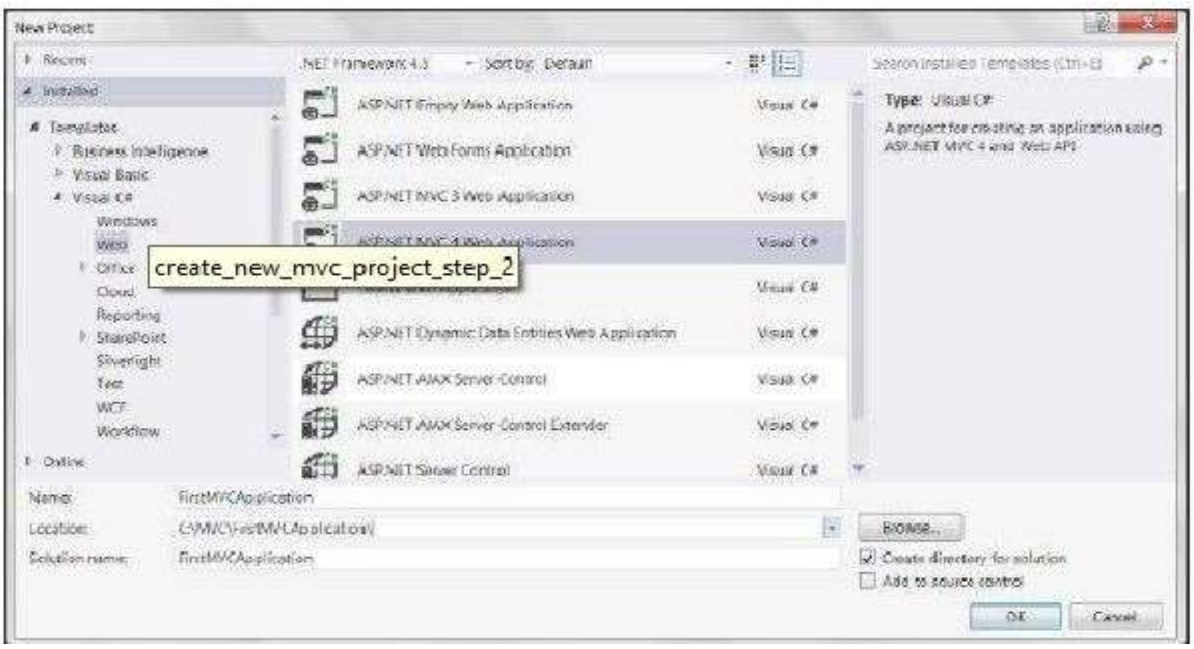
MVC and ASP.NET Web Forms are inter-related yet different models of development, depending on the requirement of the application and other factors. At a high level, you can consider that MVC is an advanced and sophisticated web application framework designed with separation of concerns and testability in mind. Both the frameworks have their advantages and disadvantages depending on specific requirements. This concept can be visualized using the following diagram −



MVC and ASP.NET Diagram

Creating MVC Application

Step 1 − Start your Visual Studio and select File → New → Project. Select Web → ASP.NET MVC Web Application and name this project as **FirstMVCApplicatio**. Select the Location as **C:\MVC**. Click OK.


New Project
Recent
Installed
Templates
Business Intelligence
Visual Basic
Visual C#
Windows
Web
Office
Cloud
Reporting
SharePoint
Silverlight
Test
WCF
Workflow
Online
.NET Framework 4.5
Sort by: Default
Search Installed Templates (Ctrl+E)
ASP.NET Empty Web Application
ASP.NET Web Forms Application
ASP.NET MVC 3 Web Application
ASP.NET MVC 4 Web Application
ASP.NET Dynamic Data Entities Web Application
ASP.NET AJAX Server Control
ASP.NET AJAX Server Control Extender
ASP.NET Server Control
Visual C#
create_new_mvc_project_step_2
Browse...
OK
Cancel

Type: Visual C#

A project for creating an application using ASP.NET MVC 4 and Web API

Name: FirstMVCApplication

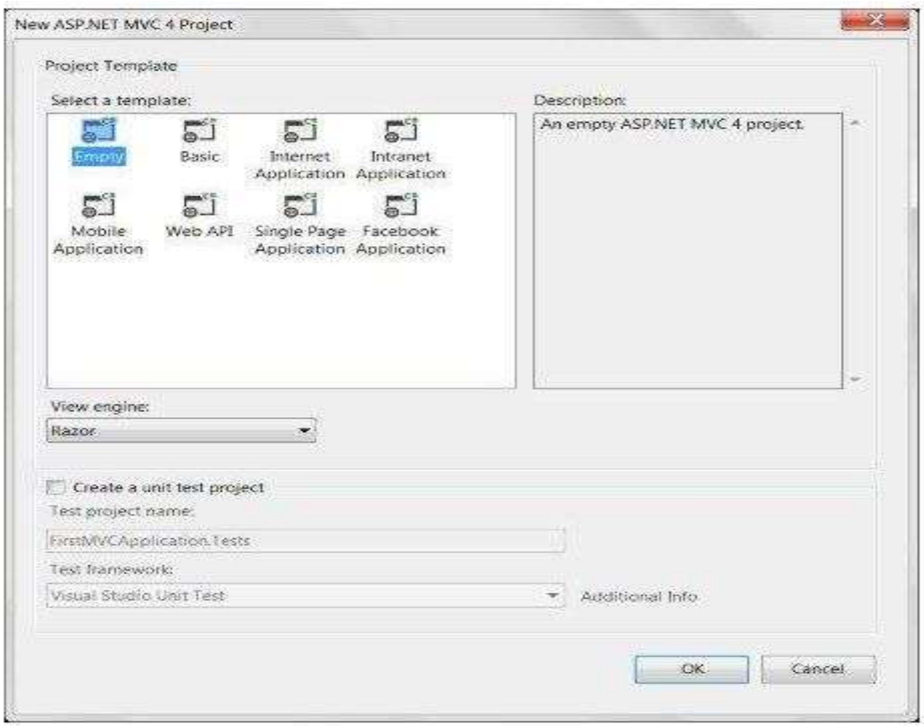
Location: C:\MVC\FirstMVCApplication\

Solution name: FirstMVCApplication

☑ Create directory for solution

☐ Add to source control

Step 2 − This will open the Project Template option. Select Empty template and View Engine as Razor. Click OK.



An empty ASP.NET MVC 4 project.

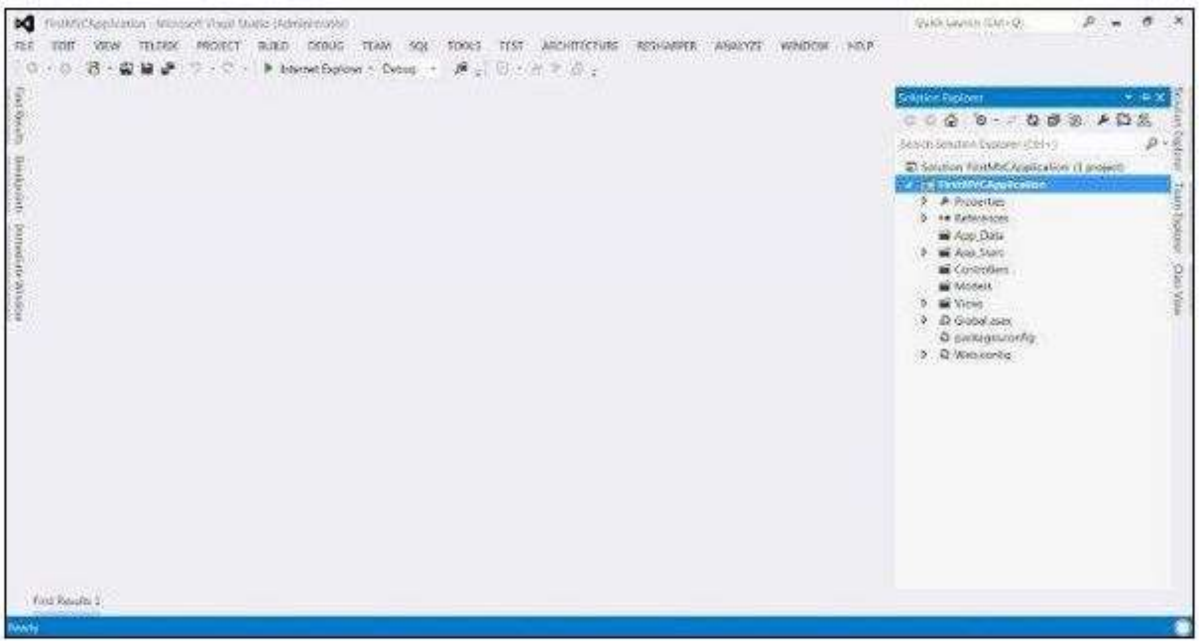
View engine: Razor

☐ Create a unit test project

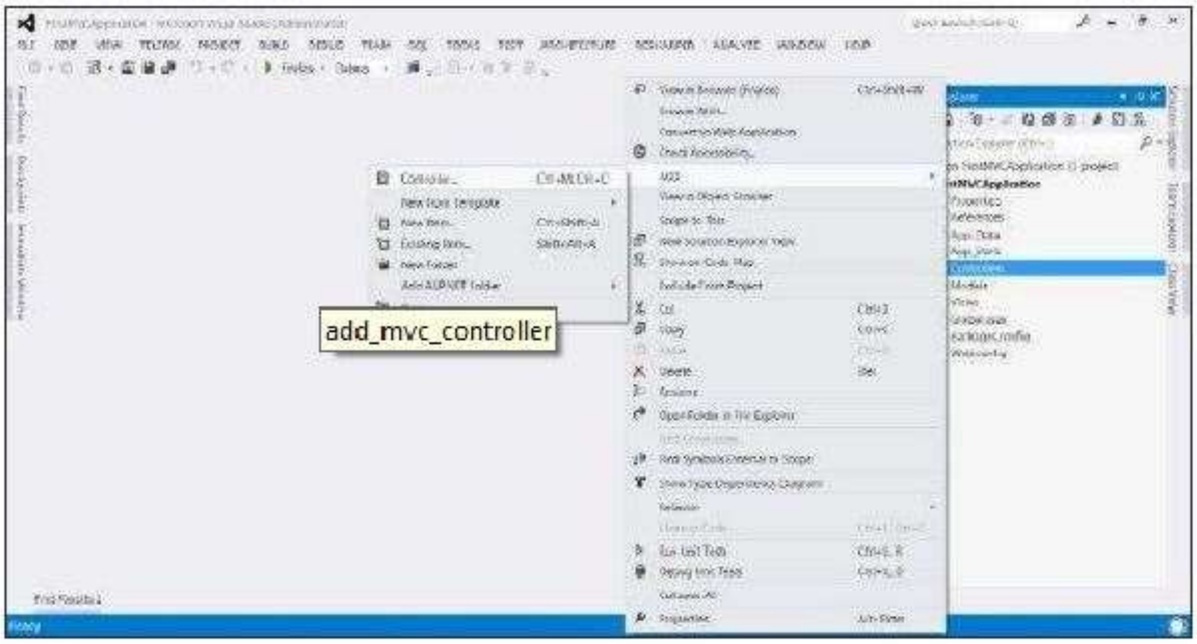
Test project name: FirstMVCApplication.Tests

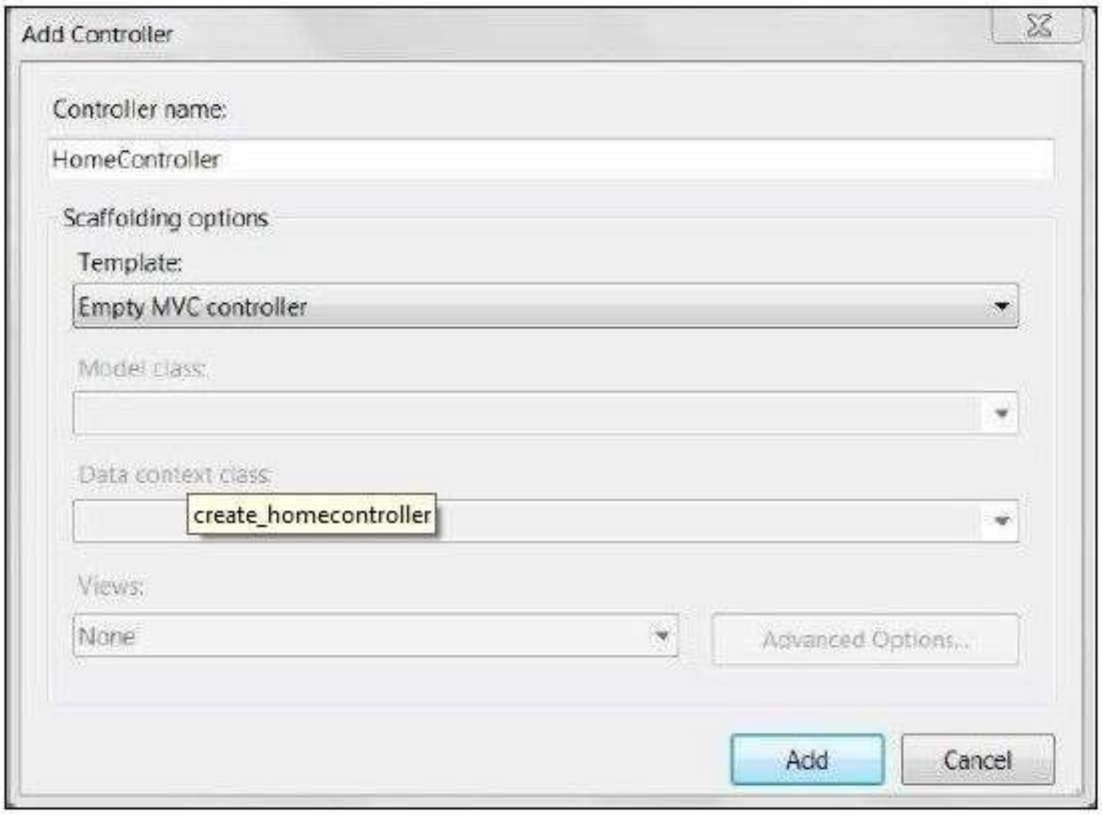
Test framework: Visual Studio Unit Test

Now, Visual Studio will create our first MVC project as shown in the following screenshot.



Step 3 − Now we will create the first Controller in our application. Controllers are just simple C# classes, which contains multiple public methods, known as action methods. To add a new Controller, right-click the Controllers folder in our project and select Add → Controller. Name the Controller as HomeController and click Add.





This will create a class file **HomeController.cs** under the Controllers folder with the following default code:

```
using System;
using System.Web.Mvc;

namespace FirstMVCApplication.Controllers {

   public class HomeController : Controller {

      public ViewResult Index() {
         return View();
      }
   }
}
```

The above code basically defines a public method Index inside our HomeController and returns a ViewResult object. In the next steps, we will learn how to return a View using the ViewResult object.

Step 4 − Now we will add a new View to our Home Controller. To add a new View, rightclick view folder and click Add → View

Adding MVC View

Step 5 − Name the new View as Index and View Engine as Razor (SCHTML). Click Add.

Create Index View

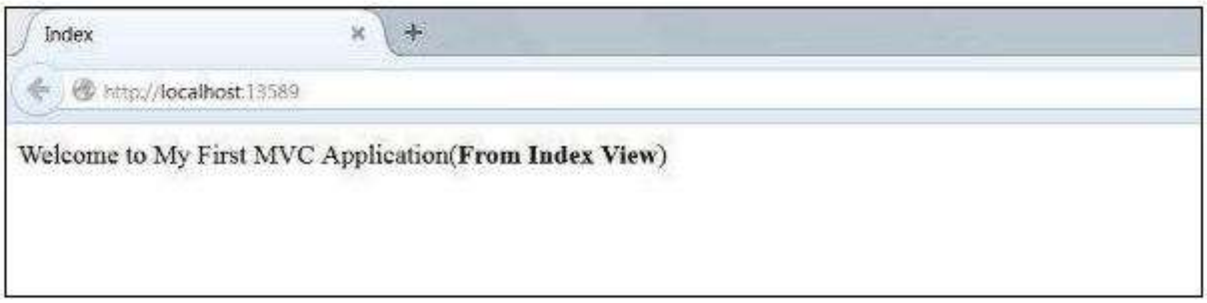
This will add a new **cshtml** file inside Views/Home folder with the following code −

```
@{
   Layout = null;
}

<html>
   <head>
      <meta name = "viewport" content = "width = device-width" />
      <title>Index</title>
   </head>

   <body>
      <div>

      </div>
   </body>
</html>
```

Step 6 − Modify the above View's body content with the following code –

```
<body>
   <div>
      Welcome to My First MVC Application (<b>From Index View</b>)
   </div>
</body>
```

Step 7 − Now run the application. This will give you the following output in the browser. This output is rendered based on the content in our View file. The application first calls the Controller which in turn calls this View and produces the output:



Welcome to My First MVC Application(**From Index View**)

In Step 7, the output we received was based on the content of our View file and had no interaction with the Controller. Moving a step forward, we will now create a small example to display a Welcome message with the current time using an interaction of View and Controller.

Step 8 − MVC uses the ViewBag object to pass data between Controller and View. Open the HomeController.cs and edit the Index function to the following code:

```
public ViewResult Index() {
   int hour = DateTime.Now.Hour;
```

```
  ViewBag.Greeting =
  hour < 12
  ? "Good Morning. Time is" +  DateTime.Now.ToShortTimeString()
  : "Good Afternoon. Time is " + DateTime.Now.ToShortTimeString();

  return View();
}
```

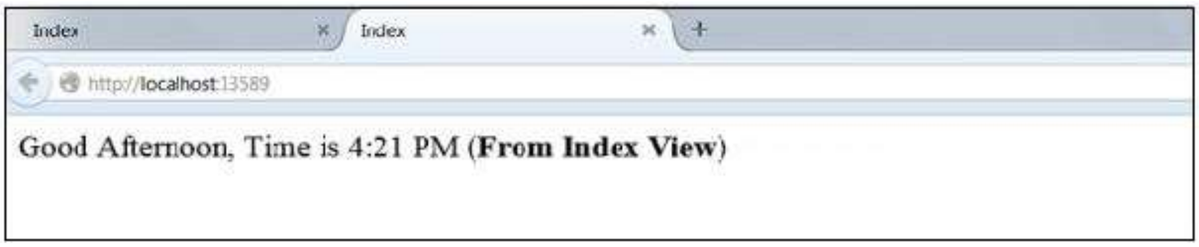
In the above code, we set the value of the Greeting attribute of the ViewBag object. The code checks the current hour and returns the Good Morning/Afternoon message accordingly using return View() statement. Note that here Greeting is just an example attribute that we have used with ViewBag object. You can use any other attribute name in place of Greeting.

Step 9 − Open the Index.cshtml and copy the following code in the body section.

```
<body>
  <div>
    @ViewBag.Greeting (<b>From Index View</b>)
  </div>
</body>
```

In the above code, we are accessing the value of Greeting attribute of the ViewBag object using @ (which would be set from the Controller).

Step 10 − Now run the application again. This time our code will run the Controller first, set the ViewBag and then render it using the View code. Following will be the output.

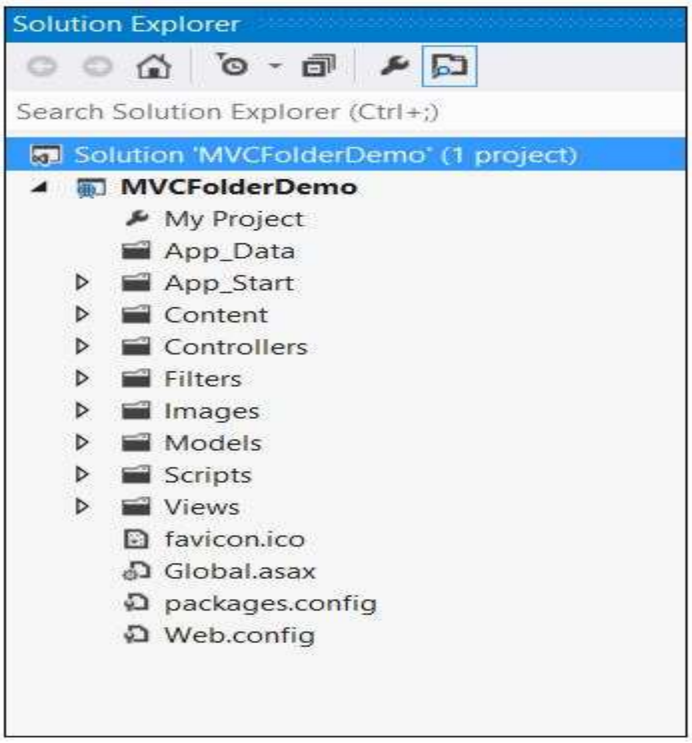


Good Afternoon, Time is 4:21 PM (**From Index View**)

MVC FOLDER

Now that we have already created a sample MVC application, let us understand the folder structure of an MVC project. We will create new a MVC project to learn this.

In your Visual Studio, open File → New → Project and select ASP.NET MVC Application. Name it as **MVCFolderDemo**.
Click OK. In the next window, select Internet Application as the Project Template and click OK.
This will create a sample MVC application as shown in the following screenshot.



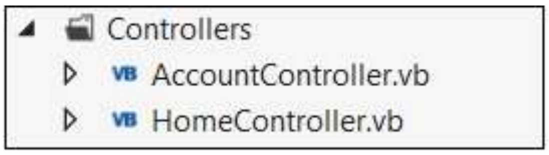
```
Solution 'MVCFolderDemo' (1 project)
◢ MVCFolderDemo
      My Project
      App_Data
   ▷  App_Start
   ▷  Content
   ▷  Controllers
   ▷  Filters
   ▷  Images
   ▷  Models
   ▷  Scripts
   ▷  Views
      favicon.ico
      Global.asax
      packages.config
      Web.config
```

Note – Files present in this project are coming out of the default template that we have selected. These may change slightly as per different versions.

Controllers Folder

This folder will contain all the Controller classes. MVC requires the name of all the controller files to end with Controller.

In our example, the Controllers folder contains two class files: AccountController and HomeController.

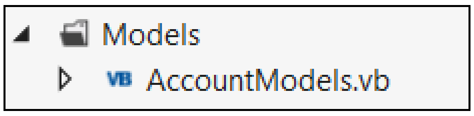


```
◢ Controllers
   ▷ VB AccountController.vb
   ▷ VB HomeController.vb
```

Models Folder

This folder will contain all the Model classes, which are used to work on application data.

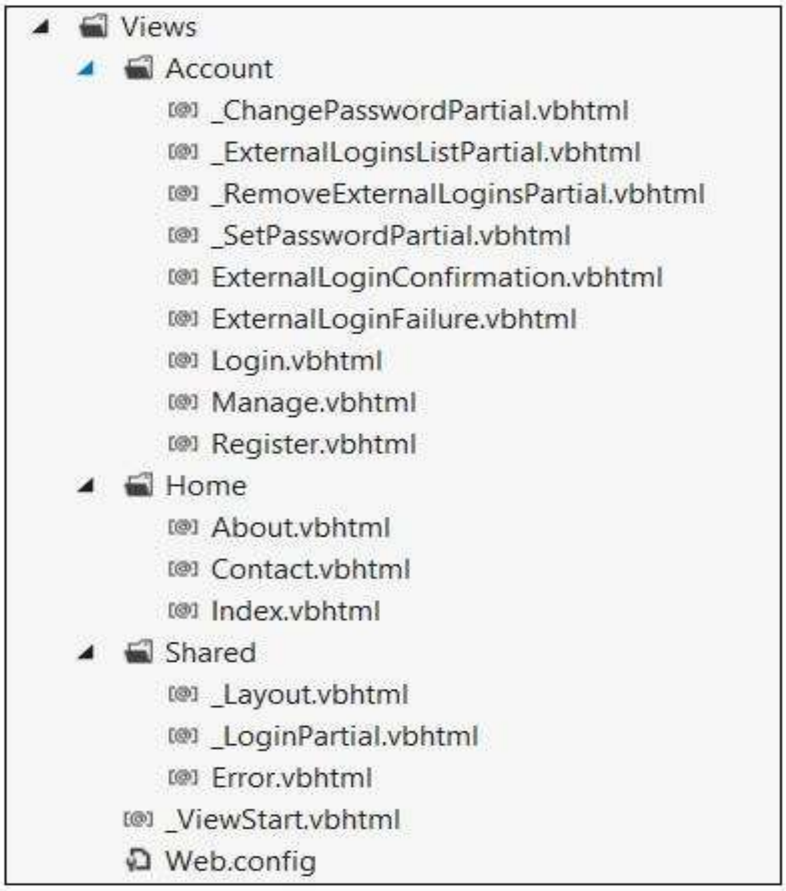
In our example, the Models folder contains AccountModels. You can open and look at the code in this file to see how the data model is created for managing accounts in our example.



```
◢ Models
   ▷ VB AccountModels.vb
```

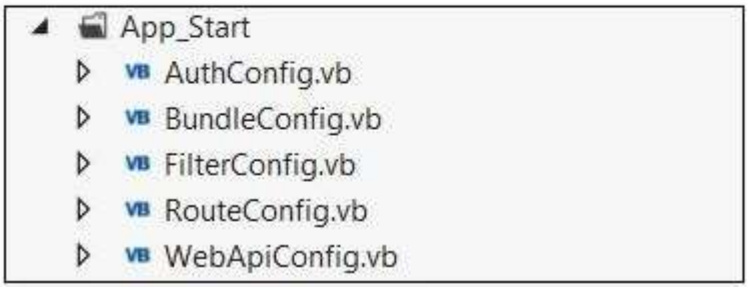
Views Folder

This folder stores the HTML files related to application display and user interface. It contains one folder for each controller.
In our example, you will see three sub-folders under Views, namely Account, Home and Shared which contains html files specific to that view area.


Views
Account
_ChangePasswordPartial.vbhtml
_ExternalLoginsListPartial.vbhtml
_RemoveExternalLoginsPartial.vbhtml
_SetPasswordPartial.vbhtml
ExternalLoginConfirmation.vbhtml
ExternalLoginFailure.vbhtml
Login.vbhtml
Manage.vbhtml
Register.vbhtml
Home
About.vbhtml
Contact.vbhtml
Index.vbhtml
Shared
_Layout.vbhtml
_LoginPartial.vbhtml
Error.vbhtml
_ViewStart.vbhtml
Web.config

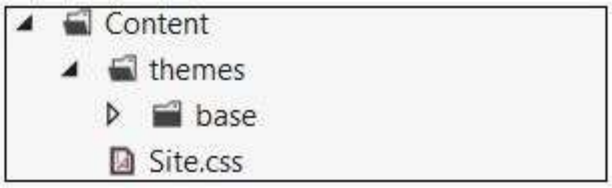
App_Start Folder

This folder contains all the files which are needed during the application load.
For e.g., the RouteConfig file is used to route the incoming URL to the correct Controller and Action.


App_Start
AuthConfig.vb
BundleConfig.vb
FilterConfig.vb
RouteConfig.vb
WebApiConfig.vb

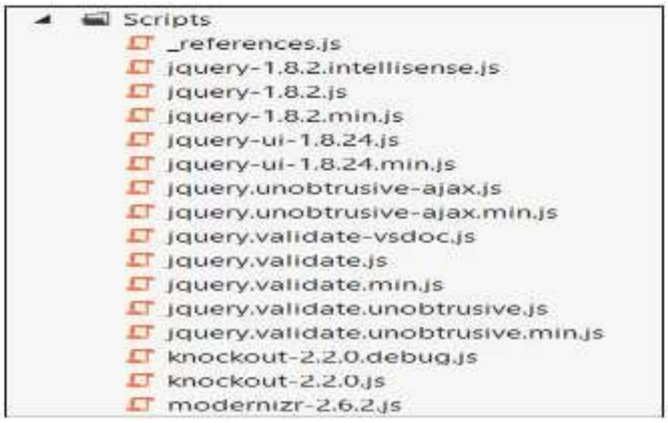
Content Folder

This folder contains all the static files, such as css, images, icons, etc.
The Site.css file inside this folder is the default styling that the application applies.


Content
themes
base
Site.css

Scripts Folder

This folder stores all the JS files in the project. By default, Visual Studio adds MVC, jQuery and other standard JS libraries.



```
Scripts
    _references.js
    jquery-1.8.2.intellisense.js
    jquery-1.8.2.js
    jquery-1.8.2.min.js
    jquery-ui-1.8.24.js
    jquery-ui-1.8.24.min.js
    jquery.unobtrusive-ajax.js
    jquery.unobtrusive-ajax.min.js
    jquery.validate-vsdoc.js
    jquery.validate.js
    jquery.validate.min.js
    jquery.validate.unobtrusive.js
    jquery.validate.unobtrusive.min.js
    knockout-2.2.0.debug.js
    knockout-2.2.0.js
    modernizr-2.6.2.js
```

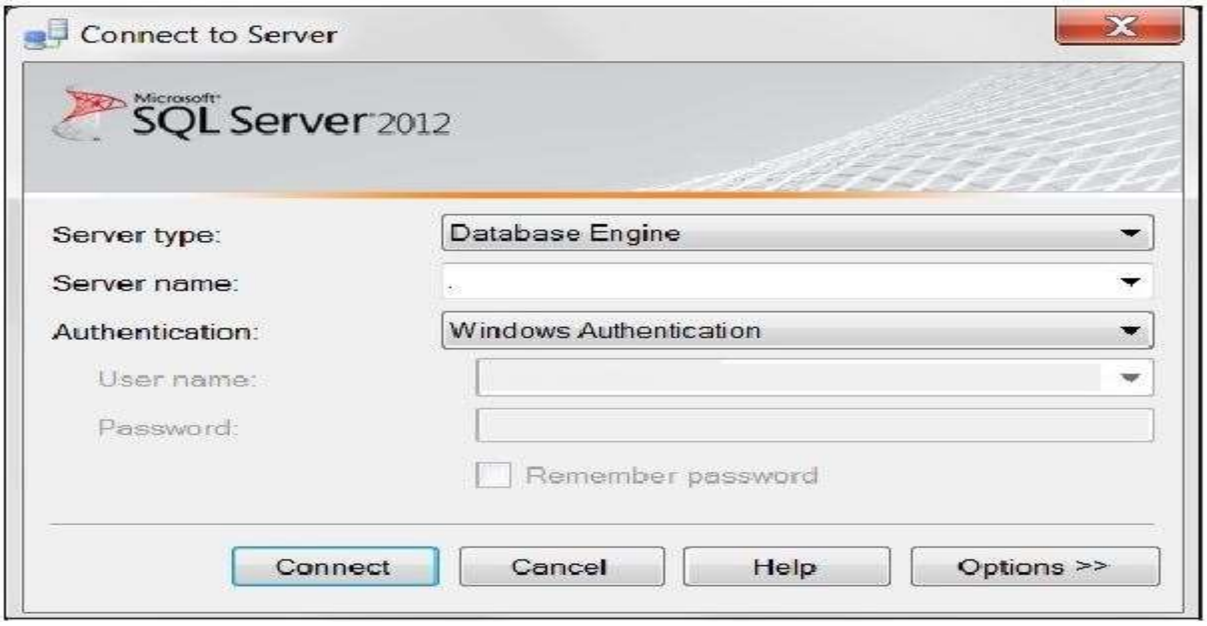
Component Model

The component 'Model' is responsible for managing the data of the application. It responds to the request from the view and it also responds to instructions from the controller to update itself.

Model classes can either be created manually or generated from database entities. We are going to see a lot of examples for manually creating Models in the coming chapters. Thus in this chapter, we will try the other option, i.e. generating from the database so that you have hands-on experience on both the methods.

Create Database Entities

Connect to SQL Server and create a new database



Connect to Server

Server type: Database Engine

Server name: .

Authentication: Windows Authentication

User name:

Password:

Remember password

Now run the following queries to create new tables.

```
CREATE TABLE [dbo].[Student](
  [StudentID]      INT          IDENTITY (1,1) NOT NULL,
  [LastName]       NVARCHAR (50) NULL,
  [FirstName]      NVARCHAR (50) NULL,
  [EnrollmentDate] DATETIME      NULL,
  PRIMARY KEY CLUSTERED ([StudentID] ASC)
)

CREATE TABLE [dbo].[Course](
  [CourseID] INT          IDENTITY (1,1) NOT NULL,
  [Title]    NVARCHAR (50) NULL,
  [Credits]  INT          NULL,
  PRIMARY KEY CLUSTERED ([CourseID] ASC)
)

CREATE TABLE [dbo].[Enrollment](
  [EnrollmentID] INT IDENTITY (1,1) NOT NULL,
  [Grade]        DECIMAL(3,2) NULL,
  [CourseID]     INT NOT NULL,
  [StudentID]    INT NOT NULL,
  PRIMARY KEY CLUSTERED ([EnrollmentID] ASC),
    CONSTRAINT [FK_dbo.Enrollment_dbo.Course_CourseID] FOREIGN KEY ([CourseID])
  REFERENCES [dbo].[Course]([CourseID]) ON DELETE CASCADE,
    CONSTRAINT [FK_dbo.Enrollment_dbo.Student_StudentID] FOREIGN KEY ([StudentID])
  REFERENCES [dbo].[Student]([StudentID]) ON DELETE CASCADE
)
```

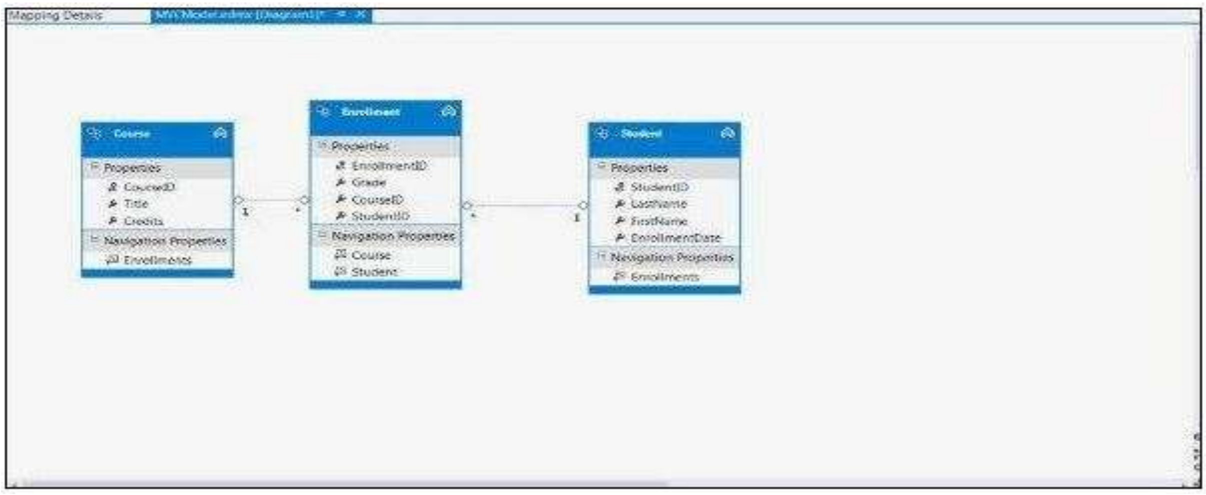
Generate Models Using Database Entities

After creating the database and setting up the tables, you can go ahead and create a new MVC Empty Application. Right-click on the Models folder in your project and select Add → New Item. Then, select ADO.NET Entity Data Model.

In the next wizard, choose Generate From Database and click Next. Set the Connection to your SQL database.

Select your database and click Test Connection. A screen similar to the following will follow. Click Next.

Select Tables, Views, and Stored Procedures and Functions. Click Finish. You will see the Model View created as shown in the following screenshot.



The above operations would automatically create a Model file for all the database entities. For example, the Student table that we created will result in a Model file Student.cs with the following code –

```
namespace MvcModelExample.Models {
  using System;
  using System.Collections.Generic;

  public partial class Student {

    public Student() {
      this.Enrollments = new HashSet();
    }

    public int StudentID { get; set; }
    public string LastName { get; set; }
    public string FirstName { get; set; }
    public Nullable EnrollmentDate { get; set; }
    public virtual ICollection Enrollments { get; set; }
  }
}
```

MVC Controllers

Asp.net MVC Controllers are responsible for controlling the flow of the application execution. When you make a request (means request a page) to MVC application, a controller is responsible for returning the response to that request. The controller can perform one or more actions. The controller action can return different types of action results to a particular request.

The Controller is responsible for controlling the application logic and acts as the coordinator between the View and the Model. The Controller receives an input from the users via the View, then processes the user's data with the help of Model and passes the results back to the View.

Create a Controller

To create a Controller −
Step 1 − Create an MVC Empty Application and then right-click on the Controller folder in your MVC application.
Step 2 − Select the menu option Add → Controller. After selection, the Add Controller dialog is displayed. Name the Controller as **DemoController**.
A Controller class file will be created.

Creating a Controller with IController

In the MVC Framework, controller classes must implement the IController interface from the System.Web.Mvc namespace.

```
public interface IController {
  void Execute(RequestContext requestContext);
}
```

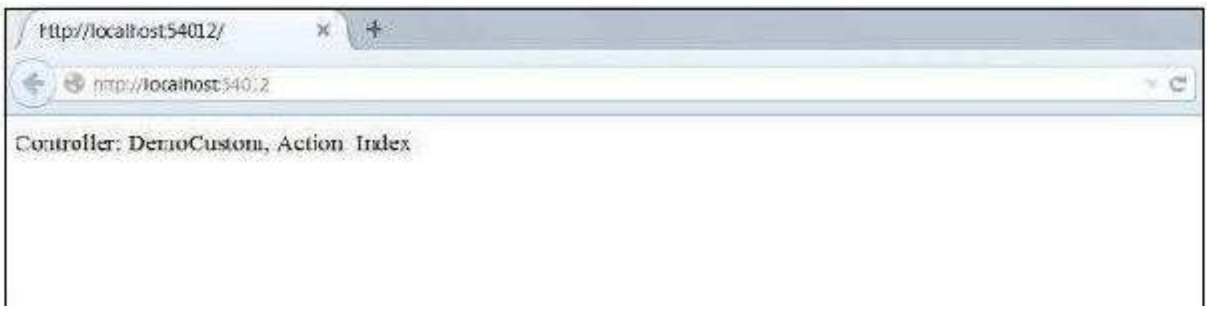
This is a very simple interface. The sole method, Execute, is invoked when a request is targeted at the controller class. The MVC Framework knows which controller class has been targeted in a request by reading the value of the controller property generated by the routing data.

Step 1 − Add a new class file and name it as DemoCustomController. Now modify this class to inherit IController interface.

Step 2 − Copy the following code inside this class.

```
public class DemoCustomController:IController {

  public void Execute(System.Web.Routing.RequestContext requestContext) {
    var controller = (string)requestContext.RouteData.Values["controller"];
    var action = (string)requestContext.RouteData.Values["action"];
    requestContext.HttpContext.Response.Write(
    string.Format("Controller: {0}, Action: {1}", controller, action));
  }
```

Step 3 − Run the application and you will receive the following output.



```
Controller: DemoCustom, Action: Index
```

As seen in the initial introductory chapters, View is the component involved with the application's User Interface. These Views are generally bind from the model data and have extensions such as html, aspx, cshtml, vbhtml, etc. In our First MVC Application, we had used Views with Controller to display data to the final user. For rendering these static and dynamic content to the browser, MVC Framework utilizes View Engines. View Engines are basically markup syntax implementation, which are responsible for rendering the final HTML to the browser.

MVC Framework comes with two built-in view engines −

Razor Engine − Razor is a markup syntax that enables the server side C# or VB code into web pages. This server side code can be used to create dynamic content when the web page is being loaded. Razor is an advanced engine as compared to ASPX engine and was launched in the later versions of MVC.

ASPX Engine − ASPX or the Web Forms engine is the default view engine that is included in the MVC Framework since the beginning. Writing a code with this engine is similar to writing a code in ASP.NET Web Forms.

Following are small code snippets comparing both Razor and ASPX engine.

Razor

```
@Html.ActionLink("Create New", "UserAdd")
```

ASPX

```
<% Html.ActionLink("SignUp", "SignUp") %>
```

Out of these two, Razor is an advanced View Engine as it comes with compact syntax, test driven development approaches, and better security features. We will use Razor engine in all our examples since it is the most dominantly used View engine.

These View Engines can be coded and implemented in following two types −

- Strongly typed
- Dynamic typed

These approaches are similar to early-binding and late-binding respectively in which the models will be bind to the View strongly or dynamically.

Strongly Typed Views

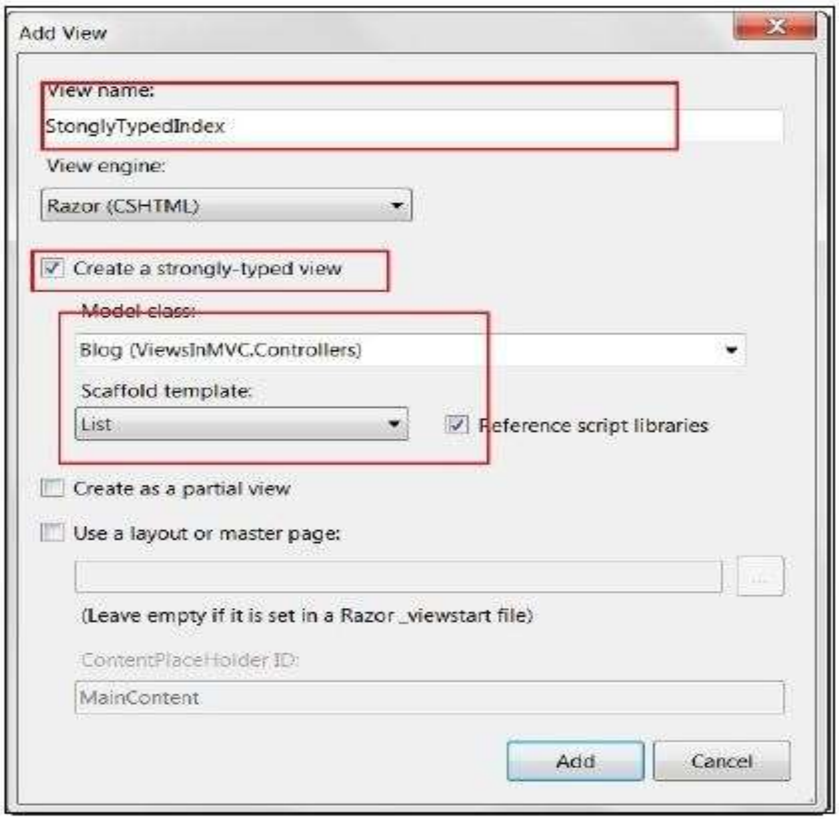
To understand this concept, let us create a sample MVC application (follow the steps in the previous chapters) and add a Controller class file named **ViewDemoController**.

Now, copy the following code in the controller file −

```
using System.Collections.Generic;
using System.Web.Mvc;

namespace ViewsInMVC.Controllers {

  public class ViewDemoController : Controller {

    public class Blog {
      public string Name;
      public string URL;
    }

    private readonly List topBlogs = new List {
      new Blog { Name = "Joe Delage", URL = "http://tutorialspoint/joe/"},
      new Blog {Name = "Mark Dsouza", URL = "http://tutorialspoint/mark"},
      new Blog {Name = "Michael Shawn", URL = "http://tutorialspoint/michael"}
    };

    public ActionResult StonglyTypedIndex() {
      return View(topBlogs);
    }

    public ActionResult IndexNotStonglyTyped() {
      return View(topBlogs);
    }
  }
}
```

In the above code, we have two action methods defined: **StronglyTypedIndex** and **IndexNotStonglyTyped**. We will now add Views for these action methods.

Right-click on StonglyTypedIndex action method and click Add View. In the next window, check the 'Create a strongly-typed view' checkbox. This will also enable the Model Class and Scaffold template options. Select List from Scaffold Template option. Click Add.


Add View

View name:

StonglyTypedIndex

View engine:

Razor (CSHTML)

☑ Create a strongly-typed view

Model class:

Blog (ViewsInMVC.Controllers)

Scaffold template:

List

☑ Reference script libraries

☐ Create as a partial view

☐ Use a layout or master page:

(Leave empty if it is set in a Razor _viewstart file)

ContentPlaceHolder ID:

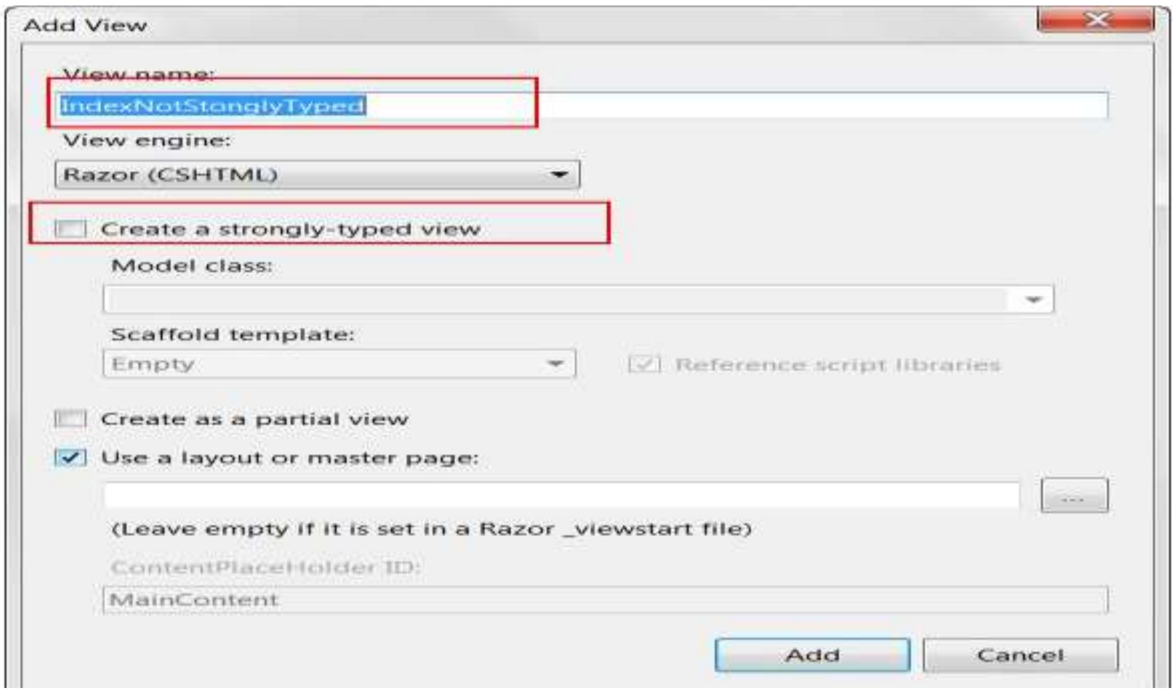
MainContent

A View file similar to the following screenshot will be created. As you can note, it has included the ViewDemoController's Blog model class at the top. You will also be able to use IntelliSense in your code with this approach.

Dynamic Typed Views

To create dynamic typed views, right-click the IndexNotStonglyTyped action and click Add View.

This time, do not select the 'Create a strongly-typed view' checkbox.


Add View

View name:

IndexNotStonglyTyped

View engine:

Razor (CSHTML)

☐ Create a strongly-typed view

Model class:

Scaffold template:

Empty

☑ Reference script libraries

☐ Create as a partial view

☑ Use a layout or master page:

(Leave empty if it is set in a Razor _viewstart file)

ContentPlaceHolder ID:

MainContent

The resulting view will have the following code −

```
@model dynamic

@{
  ViewBag.Title = "IndexNotStonglyTyped";
}

<h2>Index Not Stongly Typed</h2>
<p>
  <ul>

    @foreach (var blog in Model) {
      <li>
        <a href = "@blog.URL">@blog.Name</a>
      </li>
    }

  </ul>
</p>
```

As you can see in the above code, this time it did not add the Blog model to the View as in the previous case. Also, you would not be able to use IntelliSense this time because this time the binding will be done at run-time.

Strongly typed Views is considered as a better approach since we already know what data is being passed as the Model unlike dynamic typed Views in which the data gets bind at runtime and may lead to runtime errors, if something changes in the linked model.

Layouts are used in MVC to provide a consistent look and feel on all the pages of our application. It is the same as defining the Master Pages but MVC provides some more functionalities.

Creating MVC Layouts

Step 1 − Create a sample MVC application with Internet application as Template and create a Content folder in the root directory of the web application.

Step 2 − Create a Style Sheet file named MyStyleSheet.css under the CONTENT folder. This CSS file will contain all the CSS classes necessary for a consistent web application page design.

Step 3 − Create a Shared folder under the View folder.

Step 4 − Create a MasterLayout.cshtml file under the Shared folder. The file MasterLayout.cshtml represents the layout of each page in the application. Right-click on the Shared folder in the Solution Explorer, then go to Add item and click View. Copy the following layout code

Layout Code

```
<!DOCTYPE html>

<html lang = "en">
  <head>
    <meta charset = "utf-8" />
    <title>@ViewBag.Title - Tutorial Point</title>
    <link href = "~/favicon.ico" rel = "shortcut icon" type = "image/x-icon" />
    <link rel = "stylesheet" href = "@Url.Content("~/Content/MyStyleSheet.css")" />
  </head>

  <body>
    <header>

      <div class = "content-wrapper">
        <div class = "float-left">
          <p class = "site-title">
            @Html.ActionLink("Tutorial Point", "Index", "Home")
          </p>
        </div>

        <div class = "float-right">
          <nav>
            <ul id = "menu">
              <li>@Html.ActionLink("Home", "Index", "Home")</li>
               <li>@Html.ActionLink("About", "About", "Home")</li>
            </ul>
          </nav>
        </div>
      </div>

    </header>
    <div id = "body">
      @RenderSection("featured", required: false)
      <section class = "content-wrapper main-content clear-fix">
        @RenderBody()
      </section>
    </div>

    <footer>
      <div class = "content-wrapper">
        <div class = "float-left">
          <p>© @DateTime.Now.Year - Tutorial Point</p>
        </div>
      </div>
    </footer>

  </body>
</html>
```

In this layout, we are using an HTML helper method and some other system-defined methods, hence let's look at these methods one by one.

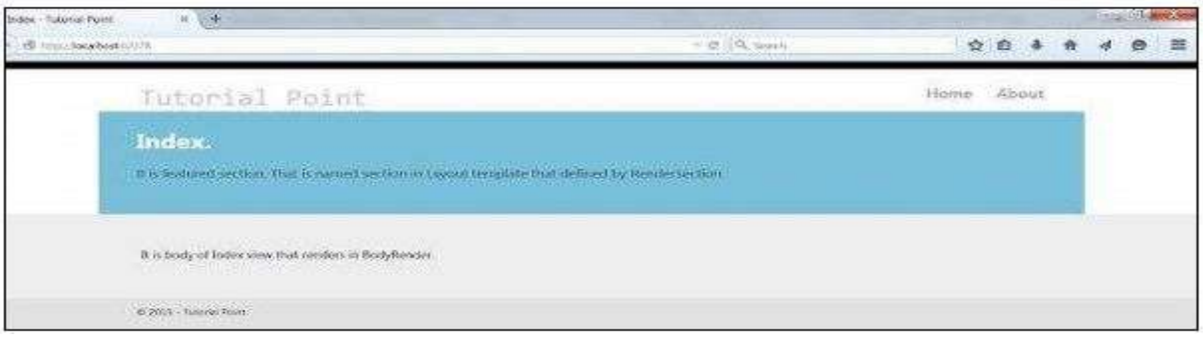
- **Url.Content()** − This method specifies the path of any file that we are using in our View code. It takes the virtual path as input and returns the absolute path.
- **Html.ActionLink()** − This method renders HTML links that links to action of some controller. The first parameter specifies the display name, the second parameter specifies the Action name, and the third parameter specifies the Controller name.

- **RenderSection()** − Specifies the name of the section that we want to display at that location in the template.
- **RenderBody()** − Renders the actual body of the associated View.

Step 5 − Finally, open the _ViewStart.cshtml file inside Views folder and add the following code −

```
@{
  Layout = "~/Views/Shared/_Layout.cshtml";
}
```

NOTE: If the file is not present, you can create the file with this name.
Step 6 − Run the application now to see the modified home page



Index.

It is featured section. That is named section in Layout template that defined by RenderSection

It is body of Index view that renders in BodyRender

© 2015 - Tutorial Point

Routing Engine

ASP.NET MVC Routing enables the use of URLs that are descriptive of the user actions and are more easily understood by the users. At the same time, Routing can be used to hide data which is not intended to be shown to the final user.

For example, in an application that does not use routing, the user would be shown the URL as http://myapplication/Users.aspx?id=1 which would correspond to the file Users.aspx inside myapplication path and sending ID as 1, Generally, we would not like to show such file names to our final user.

To handle MVC URLs, ASP.NET platform uses the routing system, which lets you create any pattern of URLs you desire, and express them in a clear and concise manner. Each route in MVC contains a specific URL pattern. This URL pattern is compared to the incoming request URL and if the URL matches this pattern, it is used by the routing engine to further process the request.

MVC Routing URL Format

To understand the MVC routing, consider the following URL −
http://servername/Products/Phones

In the above URL, Products is the first segment and Phone is the second segment which can be expressed in the following format −

{controller}/{action}

The MVC framework automatically considers the first segment as the Controller name and the second segment as one of the actions inside that Controller.

Note − If the name of your Controller is ProductsController, you would only mention Products in the routing URL. The MVC framework automatically understands the Controller suffix.

Create a Simple Route

Routes are defined in the RouteConfig.cs file which is present under the App_Start project folder.

You will see the following code inside this file −

```
public class RouteConfig {

   public static void RegisterRoutes(RouteCollection routes) {
      routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

      routes.MapRoute(
         name: "Default",
         url: "{controller}/{action}/{id}",
         defaults: new { controller = "Home", action = "Index",
            id = UrlParameter.Optional }
      );
   }
}
```

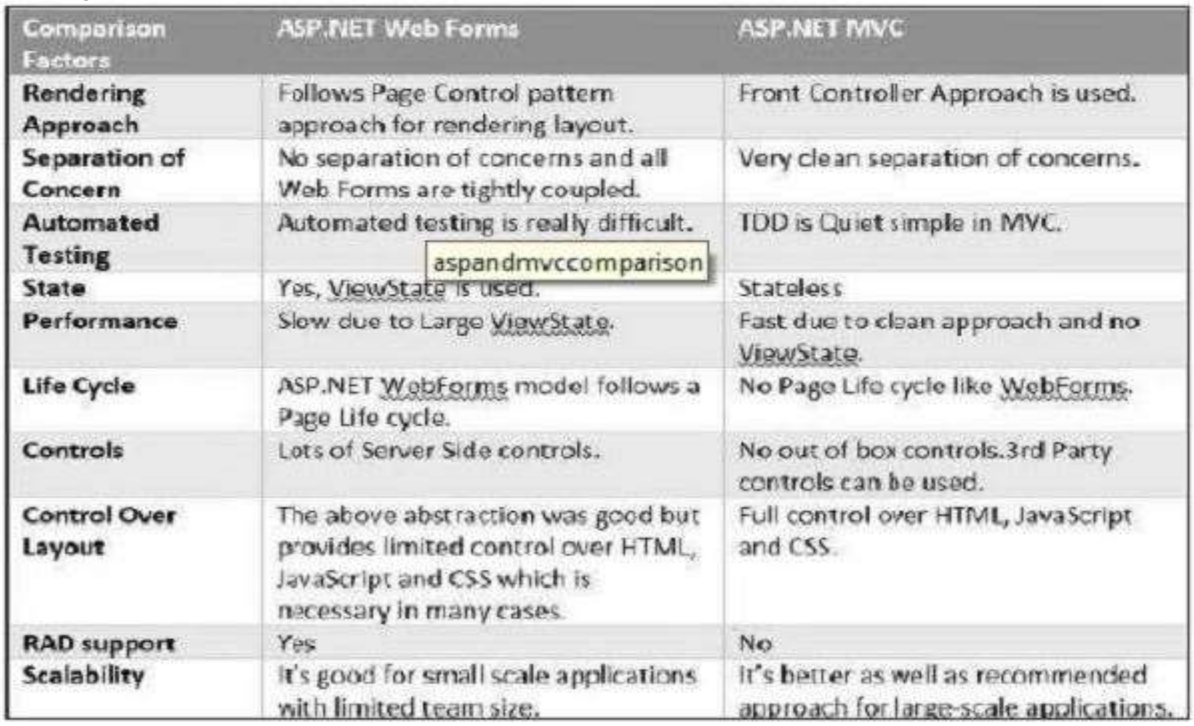
This RegisterRoutes method is called by the Global.ascx when the application is started. The Application_Start method under Global.ascx calls this MapRoute function which sets the default Controller and its action (method inside the Controller class).

To modify the above default mapping as per our example, change the following line of code −

```
defaults: new { controller = "Products", action = "Phones", id = UrlParameter.Optional }
```

This setting will pick the ProductsController and call the Phone method inside that. Similarly, if you have another method such as Electronics inside ProductsController, the URL for it would be − **http://servername/Products/Electronics**

Comparison Table of ASP.NET Web Forms and ASP.NET MVC


aspandmvccomparison

Comparison Factors	ASP.NET Web Forms	ASP.NET MVC
Rendering Approach	Follows Page Control pattern approach for rendering layout.	Front Controller Approach is used.
Separation of Concern	No separation of concerns and all Web Forms are tightly coupled.	Very clean separation of concerns.
Automated Testing	Automated testing is really difficult.	TDD is Quiet simple in MVC.
State	Yes, ViewState is used.	Stateless
Performance	Slow due to Large ViewState.	Fast due to clean approach and no ViewState.
Life Cycle	ASP.NET WebForms model follows a Page Life cycle.	No Page Life cycle like WebForms.
Controls	Lots of Server Side controls.	No out of box controls.3rd Party controls can be used.
Control Over Layout	The above abstraction was good but provides limited control over HTML, JavaScript and CSS which is necessary in many cases.	Full control over HTML, JavaScript and CSS.
RAD support	Yes	No
Scalability	It's good for small scale applications with limited team size.	It's better as well as recommended approach for large-scale applications.