

Chapter 13

Web Applications and the Internet



Now that we have covered the basics of programming with PHP and MySQL, it is time to turn our attention to the task at hand—writing web applications. Before we begin to show you some of the techniques and tools for doing this, we are going to spend time discussing many of the key concepts, considerations, and issues to keep in mind when writing them.

Over the course of this chapter, we will

- Discuss the technologies and protocols that make up the World Wide Web and learn how they work
- Define *web applications* and discuss how we will structure them
- Learn about 3-tier and *n*-tier architectures, including what might be placed in each tier
- Spend some time covering performance and scalability of our web applications

A Closer Look at the World Wide Web

We will begin this chapter by taking a closer look at the technologies that make the Internet—specifically the World Wide Web (WWW)—work. This discussion will be a useful refresher for many, and will provide a solid foundation for showing how our web applications work. It will also frame our later discussions of performance, scalability, and security.

The Internet: It's Less Complicated Than You Think

Many beginning programmers will confess that the workings of the World Wide Web are something only slightly less fantastical than black magic. However, this could not be further from the truth. One of the most powerful features of the Internet, and surely one of the greatest factors leading to its rapid acceptance and widespread adoption, is that it is based almost entirely on simple, open specifications and technologies that are freely available for all.

Most of the work on the Internet is done by a few key protocols or mechanisms by which computers talk to each other. For example, TCP/IP is the base protocol by which computers communicate. Other protocols, which provide progressively more and more functionality, are built on top of this. As we shall see, the *Hypertext Transfer Protocol* (HTTP) is simply a series of text messages (albeit with some well-defined structure to them) sent from one computer to another via TCP/IP.

While this flexibility and openness facilitates adoption and allows for easy customization and extensions, it does have its drawbacks, most notably in the realm of security. Because of its well-known structure and the availability of free implementations, there are many opportunities for people with less than noble intentions to exploit this openness. We will cover this in greater detail in Chapter 16, "Securing Your Web Applications: Planning and Code Security."

One important factor that should be considered when writing a web application is the speed at which users can connect to the Internet. Although there has been a surge in availability of high-bandwidth connections over mediums such as DSL, television cable, and satellite, a large portion of users are still using standard modems no faster than 56kbps.

If you are designing an application that you know will only be used by corporate customers with high-speed connections, it might not be a problem to include large, high-resolution images or video in your site. However, home users might be quickly turned off, opting to go somewhere less painfully slow.

Computers Talking to Computers

The key technology that makes the Internet work as we know it today is the TCP/IP protocol, which is actually a pair of protocols. The *Internet Protocol* (IP) is a mechanism by which computers identify and talk to each other. Each computer has what is called an *IP address*, or a set of numbers (not entirely unlike a telephone number) that identify the computer on the Internet. The Internet Protocol allows two computers with IP addresses to send each other messages.

The format of these IP addresses depends on the version of IP in use, but the one most commonly used today is *IPv4*, where IP addresses consist of four one-byte digits ranging from 0–254 (255 is reserved for broadcasting to large numbers of computers), and is typically written as xxx.yyy.zzz.www (such as 192.168.100.1). There are various ways of grouping the possible IPv4 addresses so that when a particular machine wants to send a message to another, it does not need to know the destination's exact location, but instead can send the message to intermediaries that know how to ensure the message ends up at the correct computer (see Figure 13-1).

However, one problem with IPv4 is that the number of unallocated IP addresses is running low, and there is often an uneven distribution of addresses. (For example, there are a few universities in the USA with more IP addresses than all of China!) One way of conserving IP addresses is for organizations to use a couple of reserved address ranges for

internal use and only have a few computers directly exposed to the Internet. These reserved ranges (192.168.x.y and 10.x.y.z) can be used by anybody and are designed to be used for internal networks. They usually have their traffic routed to the Internet by computers running *network address translators* (NAT), which allow these “nonpublic” addresses to take full advantage of the features available.

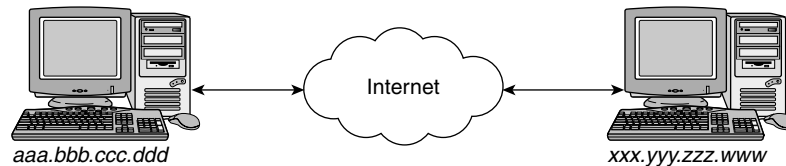


Figure 13-1: Two computers talking over the Internet via TCP/IP.

A new version of IP, IPv6, has been developed and is seeing increasing adoption. While it would not hurt to learn about this new version and its addressing scheme (which has a significantly larger address space than IPv4), we mostly use IPv4 in our examples (although we do not do anything to preclude the use of IPv6). Many key pieces of software, including Apache HTTP Server and PHP, are including IPv6 support in newer releases for early adopters.

The Internet Protocol does little else than allow computers to send messages to each other. It does nothing to verify that messages arrive in the order they were sent without corruption. (Only the key header data is verified.)

To provide this functionality, the *Transmission Control Protocol* (TCP) was designed to sit directly on top of IP. TCP makes sure that packets actually arrive in the correct order and makes an attempt to verify that the contents of the packet are unchanged. This implies some extra overhead and less efficiency than IP, but the only other alternative would be for every single program to do this work itself—a truly unpleasant prospect.

TCP introduces a key concept on top of the IP address that permits computers to expose a variety of services or offerings over the network called a *port*. Various port numbers are reserved for different services, and these numbers are both published and well known. On the machine exposing the services, there are programs that listen for traffic on a particular port—*services*, or *daemons*. For example, most e-mail occurs over port 25, while HTTP traffic for the WWW (with which we are dealing extensively in this book) occurs on port 80. You will occasionally see a reference to a web site (URL) written as `http://www.mywebsitehooray.com:8080`, where the `:8080` tells your web browser what port number to use (in this case, port 8080).

The one other key piece of the Internet puzzle is the means by which names, such as `www.warmhappybunnies.com`, are mapped to an IP address. This is done by a system called the *Domain Name System*, or DNS. This is a hierarchical system of naming that maps names onto IP addresses and provides a more easily read and memorized way of remembering a server.

The system works by having a number of “top level” domains (com, org, net, edu, ca, de, cn, jp, biz, info, and so on) that then have their own domains within them. When you enter a name, such as `www.warmhappybunnies.com`, the software on your computer knows to connect to a DNS server (also known as a “name server”) which, in turn, knows to go to a “root name server” for the com domain and get more information for the

warmhappybunnies domain. Eventually, your computer gets an IP address back, which the TCP/IP software in the operating system knows how to use to talk to the desired server.

The Hypertext Transfer Protocol

The web servers that make the WWW work typically “listen” on port 80, the port reserved for HTTP traffic. These servers operate in a very simple manner. Somebody, typically called the “client,” connects to the port and makes a request for information from the “server.” The request is analyzed and processed. Then a response is sent, with content or with an error message if the request was invalid or unable to be processed. After all of this, the connection is closed and the server goes back to listening for somebody else to connect. The server does not care who is connecting and asking for data and—apart from some simple logging—does not remember anything about the connection. This is why HTTP is sometimes called a *stateless protocol*—no information is shared between connections to the server.

The format of both the HTTP request and response is very simple. Both share the following plain text format:

```
Initial request/response line  
Optional Header: value  
[Other optional headers and values]  
[blank line, consisting of CR/LF]  
Optional Body that comes after the single blank line.
```

An HTTP request might look something like the following:

```
GET /index.php HTTP/1.1  
Host: www.myhostnamehooray.com  
User-Agent: WoobaBrowser/3.4 (Windows)  
[this is a blank line]
```

The response to the previous request might be something as follows:

```
HTTP/1.1 200 OK  
Date: Wed, 8 Aug 2001 18:08:08 GMT  
Content-Type: text  
Content-Length: 1234  
  
<html>  
<head>  
  <title>Welcome to my happy web site!</title>  
</head>  
<body>  
  <p>Welcome to our web site !!! </p>
```

```

...
..
.
</body>
</html>

```

There are a couple of other HTTP methods similar to the GET method, most notably POST and HEAD. The HEAD method is similar to the GET method with the exception that the server only sends the headers rather than the actual content.

As we saw in Chapter 7, “Interacting with the Server: Forms,” The POST method is similar to the GET method but differs in that it is typically used for sending data for processing to the server. Thus, it also contains additional headers with information about the format in which the data is presented, the size of this data, and a message body containing it. It is typically used to send the results from forms to the web server.

```

POST /createaccount.php HTTP/1.1
Host: www.myhostnamehooray.com
User-Agent: WoobaBrowser/3.4 (Windows)
Content-Type: application/x-www-form-urlencoded
Content-Length: 64

```

```
username=Marc&address=123+Somewhere+Lane&city=Somewhere&state=WA
```

The *Content-Type* of this request tells us how the parameters are put together in the message body. The *application/x-www-form-urlencoded* type means that the parameters have the same format for adding parameters to a GET request as we saw in Chapter 7 with the format

```
http://hostname/filename?param1=value1[&param2=value2 etc...]
```

Thus, the POST request can be sent as a GET :

```

GET /createaccount.php?username=Marc&address=123
+Some+Lane&city=Somewhere&state=WA HTTP/1.1
Host: www.myhostnamehooray.com
User-Agent: WoobaBrowser/3.4 (Windows)
[this is a blank line]

```

Many people assume that since they cannot easily see the values of the form data in POST requests that are sent to the server, they must be safe from prying eyes. However, as we have just seen, the only difference is where they are placed in the plain-text HTTP request. Anybody with a packet-sniffer who is looking at the traffic between the client and the server is just as able to see the POST data as a GET URI. Thus, it would certainly be risky to send passwords, credit card numbers, and other information unchanged in a regular HTTP request or response.

TELNET: A HELPFUL DEBUGGING TOOL

It can be difficult to understand the simple text nature of Internet protocols such as HTTP, but fortunately there is a tool that ships with most modern operating systems that lets us see this more clearly and even view the responses from the server—*telnet*. It is quite easy to use. On the command line, you specify the host to which you would like to connect and the port number on that host to use. You can specify the numeric value for the port (80) or the name commonly assigned to the port (*http*). In the absence of a port number, a port number for an interactive Telnet login session is assumed. (This is not supported by all hosts.)

On Windows, you can type from the command line or Start/Run dialog box:

```
telnet.exe webserverhostname http
```

On Unix and Mac OS X systems, you can enter

```
# telnet webserverhostname http
```

Most systems wait after showing something similar to the following:

```
# telnet phpwebserver http
Trying 192.168.1.95...
Connected to phpwebserver.intranet.org.
Escape character is '^]'
```

You can now type a simple HTTP request to see what happens. For example, to see the headers that would be returned for asking to see the home page for a server, you can enter the following (changing Host: to the correct host name for your server):

```
HEAD / HTTP/1.1
Host: phpwebserver
User-Agent: EnteredByHand (FreeBSD)
```

After entering a blank line to indicate that you are done with the headers, you will see something similar to the following:

```
HTTP/1.1 200 OK
Date: Sun, 26 Dec 2004 15:35:31 GMT
Server: Apache/1.3.33 (Unix) PHP/5.0.3 mod_ssl/2.8.16
OpenSSL/0.9.7d
X-Powered-By: PHP/5.0.3
Content-Type: text/html
```

To fetch a page, you can change the request to

```
GET / HTTP/1.1
Host: phpwebserver
User-Agent: EnteredByHand (FreeBSD)
```

After entering the blank line, you will receive all of the HTML that the page would return (or an error if there is no page).

Telnet is an extremely simple yet handy utility we can use at various points to help us debug our web applications.



MIME Types

The application/x-form-urlencoded Content-Type shown in the previous section is an example of what are called *Multipurpose Internet Mail Extensions* (MIME). This is a specification that came about from the need to have Internet mail protocols support more than plain ASCII (US English) text. As it was recognized that these types would be useful beyond simple email, the number of types has grown, as has the number of places in which each is used—including our HTTP headers.

MIME types are divided into two parts—the media type and its subtype—and are separated by a forward slash character:

```
image/jpeg
```

Common media types you will see are

- text
- image
- audio
- video
- application

Subtypes vary greatly for various media types, and you will frequently see some of the following combinations:

- text/plain
- text/html
- image/jpeg
- image/gif
- application/x-form-urlencoded
- audio/mp3

Some MIME types include additional attributes after the type to specify things, such as the character set they are using or the method via which they were encoded:

```
text/html; charset="utf-8"
```

We will not often need MIME types throughout this book, but when we do, the values will mostly be the preceding application/x-form-urlencoded, text/html, and image/jpeg types.

Secure Sockets Layer (SSL)

As people came to understand the risks associated with transmitting plain text over the Internet, they started to look at ways to encrypt this data. The solution most widely used today across the WWW is *Secure Sockets Layer* (SSL) encryption. SSL is largely a transport level protocol. It is strictly a way to encode the TCP/IP traffic between two computers and does not affect the plain text HTTP traffic that is sent across the secure transaction.

What SSL Is

SSL is a protocol for secure network communications between computers on the Internet. It provides a way to encrypt the TCP/IP traffic on a particular port between two computers. This makes the traffic more difficult to view by people watching the network traffic between the two machines.

SSL is based on *public key cryptography*, a mechanism by which information is encrypted and decrypted using pairs of keys (one of which is a *public key*) instead of single keys. It is

because of the existence of these public keys that we can use SSL and public key cryptography in securing traffic to and from web servers over HTTP, the variant of which is often called HTTPS.

What SSL Is Not

Before we get to the details of how SSL works, it is important to realize that SSL is not a panacea for security problems. While it can be used to make your web applications more secure, designers and developers alike should avoid falling into the trap of thinking that using SSL solves all of our security concerns.

It neither relieves us of the need to filter all of our input from external sources, nor prevents us from having to pay attention to our servers, software bugs, or other means through which people might attack our applications. Malicious users can still connect to our web application and enter mischievous data to try to compromise our security. SSL merely provides a way to prevent sensitive information from being transmitted over the Internet in an easily readable format.

Encryption Basics

A reasonably straightforward form of encryption that many users will be familiar with is *symmetric encryption*. In this, two sources share the same private key (like a password). One source encrypts a piece of information with this private key and sends the data to the other source, which in turn decrypts it with the same private key.

This encryption tends to be fast, secure, and reliable. Algorithms for symmetric encryption of chunks of data include DES, 3DES, RC5, and AES. While some of these algorithms have proved increasingly weak as the computing ability of modern computers has helped people crack passwords, others have continued to hold up well.

The big problem with symmetric encryption is the shared private key. Unless your computer knows the private key that the other computer is using, you cannot encrypt and decrypt traffic. The other computer cannot publish what the key is either, because anybody could use it to view the traffic between the two computers.

A solution to this problem exists in the form of an encryption innovation known as *public key cryptography* or *asymmetric encryption*. The algorithms for this type of encryption use two keys—the public key and the private key. The private key is kept secret on one source (typically a server of some sort). The public key, on the other hand, is shared with anyone who wants it.

The magic of the algorithm is that once you encrypt data with one of the keys, only the holder of the other key can in turn decrypt that data. Therefore, there is little security risk in people knowing a server's public key—they can use it only to encrypt data that the server can decrypt with its private key. People viewing the traffic between two servers cannot analyze it without that private key.

One problem with public key cryptography is that it tends to be slower than symmetric encryption. Thus, most protocols that use it (such as SSL) make the connection initially with public key cryptography and then exchange a symmetric private key between the two computers so that subsequent communications can be done via the symmetric encryption methods. To prevent tampering with the encrypted data, an algorithm is run on the data before it is encrypted to generate a *message address code* (MAC), which is then sent along with the encrypted data. Upon decryption at the other end, the same algorithm is run on the unpacked data and the results are compared to make sure nobody has tampered with or corrupted the data (see Figure 13-2).

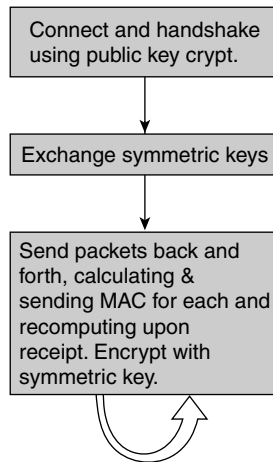


Figure 13-2: SSL in a nutshell.

How Web Servers Use SSL

As we can see, public key cryptography and SSL are perfect for our web server environment where we would like random computers to connect to our server but still communicate with us over an encrypted connection to protect the data being sent between client and application. However, we still have one problem to solve—how are we sure that we are connecting to the real server and not somebody who has made his computer appear like the server? Given that there are ways to do this, the fake server could give us its own public key that would let it decrypt and view all of the data being sent. In effect, we need to authenticate servers by creating a trusted link between a server and its public key.

This is done via the mechanism of *digital certificates*. These are files with information about the server, including the domain name, the company that requested the certificate, where it conducts business, and its public key. This digital certificate is in turn encrypted by a signing authority using its own private key.

Your client web browser stores the public keys for a number of popular signing authorities and implicitly trusts these. When it receives a certificate from a web site, it uses the public key from the authority that generated the certificate to decrypt the encoded signature that the signing authority added to the end with its private key. It also verifies that the domain name encoded in the certificate matches the name of the server to which you have just connected. This lets it verify that the certificate and server are valid.

Thus, we can be sure that the server is the one from it was supposed to come from. If somebody merely copied the certificate to a false server, he would never be able to decrypt traffic from us because he would not have the correct private key. He could not create a “fake” certificate because it would not be signed by a signing authority, and our computer would complain and encourage us not to trust it.

The sequence of events for a client connecting to a web server via SSL is as follows:

1. The client connects to the server, usually on port 443 (https, instead of port 80 for http), and as part of the normal connection negotiations (*handshaking*) asks for the certificate.
2. The server returns the certificate to the client, who determines which public signing authority authorized and signed the certificate and verifies that the signature is valid

- by using the public key for that authority. The client also makes sure that the domain listed in the certificate is the domain to which it is connecting.
3. If the certificate is found to be valid, symmetric keys are generated and encrypted using the public key. The server then decrypts these with its private key, and the two computers begin communication via symmetric encryption.
 4. If the certificate, domain name, or keys do not match up correctly, the client or server machines abort the connection as unsafe.

While this is a simplification of the process (these protocols are remarkably robust and have many additional details to prevent other attacks and problems), it is entirely adequate for our needs in this book.

We will see more about how we use SSL in PHP and our web servers in Chapter 17, “Securing Your Web Applications: Software and Hardware Security.”

Other Important Protocols

There are a few other protocols that are widely used today which you might encounter while writing various web applications.

Simple Mail Transfer Protocol (SMTP)

The protocol by which a vast majority of e-mail (or spam) is sent over the Internet today is a protocol known as the *Simple Mail Transfer Protocol* (SMTP). It allows computers to send each other e-mail messages over TCP/IP and is sufficiently flexible to allow all sorts of rich message types and languages to pass over the Internet. If you plan to have your PHP web applications send e-mail, this is the protocol they will use.

Simple Object Access Protocol (SOAP)

A reasonably new protocol, the *Simple Object Access Protocol* (SOAP) is an XML-based protocol that allows applications to exchange information over the Internet. It is commonly used to allow applications to access XML Web Services over HTTP (or HTTP and SSL). This mechanism is based on XML, a simple and powerful markup language (see Chapter 23, “XML and XHTML”) that is platform independent, meaning that anybody can write a client for a known service.

We will learn more about SOAP in Chapter 27, “XML Web Services and SOAP.”

Designing Web Applications

The main theme of this book is writing web applications using PHP and MySQL, but we have not yet defined what exactly we mean by this. It is important to understand that when we say web application, we are talking about something very different from a simple web site that serves up files such as HTML, XML, and media.

Terminology

We will define a *web application* as a dynamic program that uses a web-based interface and a client-server architecture. This is not to say that web applications have to be complicated and difficult to implement—we will demonstrate some extremely simple ones in later chapters—but they definitely have a more dynamic and code-oriented nature than simple sites.

When we talk about the server, we are primarily referring to the machine or group of machines that acts as the web server and executes PHP code. It is important to note that “the server” does not have to be one machine. Any number of components that the server

uses to execute and serve the application might reside on different machines, such as the application databases, web services used for credit card processing, and so on. The web servers might reside on multiple machines to help handle large demand.

As for the client, we are referring to a computer that accesses the web application via HTTP by using a web browser. As we write user interfaces for our web applications, we will resist the urge to use features specific to individual browsers, ensuring the largest possible audience for our products. As we mentioned before, some of these clients will have high-speed connections to the Internet and will be able to transfer large amounts of data, while others will be restricted to modem-based connections with a maximum bandwidth of 56K.

Basic Layout

Every web application that you write will have a different layout, flow of execution, and way of behaving. It can prove helpful to follow some general strategies for organizing code and functionality to help with concerns such as maintainability, performance, scalability, and security.

As far as the average end user is concerned, web applications and web sites have a very simple architecture, as you can see in Figure 13-3.

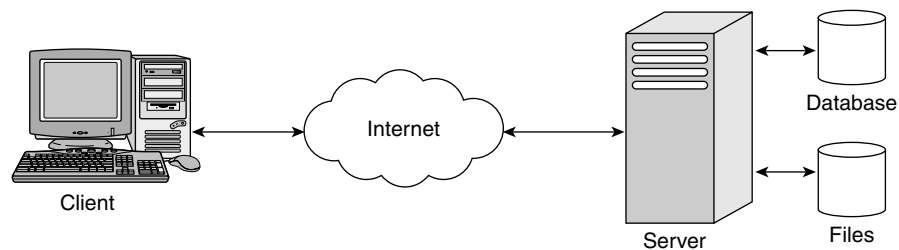


Figure 13-3: How end users and clients view interactions with web sites.

The user strictly sees a program on his computer talking to another computer, which is doing all sorts of things, such as consulting databases, services, and so on. As we will mention again in Chapter 14, “Implementing a User Interface,” users largely do not think of browsers and the content they serve up as different things—it is all part of the Internet to them.

As authors of web application software, the initial temptation for us might be to put all of the code for the web application in one place—writing scripts that queried the database for information, printed the results for the user, and then did some credit card processing or other things.

While this does have the advantage of being reasonably quick to code, the drawbacks become quickly apparent:

- The code becomes difficult to maintain when we try to change and upgrade the functionality. Instead of finding clear and well-known places for particular pieces of functionality, we have to go through various files to find what needs to be changed.
- The possibilities for code reuse are reduced. For example, there is no clear place where user management takes place; if we wanted to add a new page to do user account maintenance, we would end up duplicating a lot of existing code.

- If we wanted to completely change the way our databases were laid out and move to a new model or schema, we would have to touch all of our files and make large numbers of potentially destabilizing changes.
- Our monolithic program becomes very difficult to analyze in terms of performance and scalability. If one operation in the application is taking an unusually long time, it is challenging (if not impossible) to pinpoint what part of our system is causing this problem.
- With all of our functionality in one place, we have limited options for scaling the web application to multiple systems and splitting the various pieces of functionality into different modules.
- With one big pile of code, it is also difficult to analyze its behavior with regards to security. Analyzing the code and identifying potential weak points or tracking down known security problems ends up becoming harder than it needs to be.

Thus, we will choose to use a multi-tiered approach for the server portion of our web application, where we split key pieces of functionality into isolatable units. We will use a common “3-tiered” approach, which you can see in Figure 13-4.

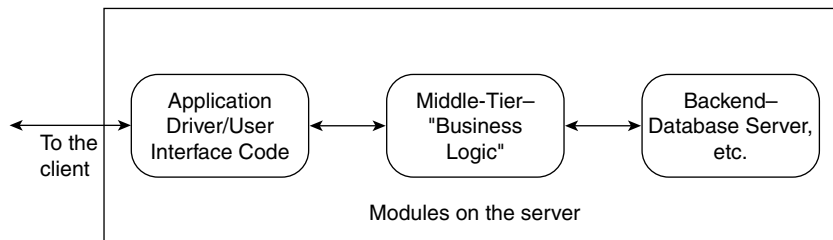


Figure 13-4: Dividing our applications logically into multiple tiers.

This architecture provides us with a good balance between modularization of our code for all of the reasons listed previously but does not prove to be so overly modularized that it becomes a problem. (See the later section titled “n-Tier Architectures.”)

Please note that even though these different modules or tiers are logically separate, they do not have to reside on different computers or different processes within a given computer. In fact, for a vast majority of the samples we provide, these divisions are more logical than anything else. In particular, the first two tiers reside in the same instance of the web server and PHP language engine. The power of web applications lies in the fact that they can be split apart and moved to new machines as needs change, which lets us scale our systems as necessary.

We will now discuss the individual pieces of our chosen approach.

User Interface

The layer with which the end user interacts most directly is the user-interface portion of our application, or the *front end*. This module acts as the main driving force behind our web application and can be implemented any way we want. We could write a client for Windows or the Apple Macintosh or come up with a number of ways to interact with the application.



However, since the purpose of this book is to demonstrate web applications, we will focus our efforts on HTML—specifically XHTML, an updated version of HTML that is fully compliant with XML and is generally cleaner and easier to parse than HTML. XML is a highly organized markup language in which tags must be followed by closing tags, and the rules for how the tags are placed are more clearly specified—in particular, no overlapping is allowed.

Thus, the following HTML code

```
<br>
<br>
<B>This is an <em>example of </B>overlapping tags</em>
```

is not valid in XHTML, since it is not valid XML. (See Chapter 23 for more detail.) The `
` tags need to be closed either by an accompanying `</br>` tag or by replacing them with the empty tag, `
`. Similarly, the `` and `` tags are not allowed to overlap. To write this code in XHTML, we simply need to change it to

```
<br/>
<br/>
<b>This is an <em>example of</em></b><em>overlapping tags</em>
```

For those who are unfamiliar with XHTML, we will provide more details on it in Chapter 23. For now, it is worth noting that it is not very different from regular HTML, barring the exceptions we mentioned previously. Throughout this book, we will use HTML and XHTML interchangeably—when we mention the former, chances are we are writing about the latter.

When designing the user interface for our application, it is important to think of how we want the users to interact with it. If we have a highly functional web application with all the features we could possibly want, but it is completely counterintuitive and indecipherable to the end user, we have failed in our goal of providing a useful web application.

As we will see in Chapter 14, it is very important to plan the interface to our application in advance, have a few people review it, and even prototype it in simple HTML (without any of the logic behind it hooked up) to see how people react to it. More time spent planning at the beginning of the project translates into less time spent on painful rewrites later on.

Business Logic

As we mentioned in the “Basic Layout” section, if our user interface code were to talk to all of the backend components in our system, such as databases and web services, we would quickly end up with a mess of “spaghetti code.” We would find ourselves in serious trouble if we wanted to remove one of those components and replace it with something completely different.

Abstracting Functionality

To avoid this problem, we are going to create a middle tier in our application, often referred to as the “business logic” or “biz-logic” part of the program. In this, we can create and implement an abstraction of the critical elements in our system. Any complicated logic for rules, requirements, or relationships is managed here, and our user interface code does not have to worry about it.

The middle tier is more of a logical abstraction than a separate system in our program. Given that our options for abstracting functionality into different processes or services are limited in PHP, we will implement our business logic by putting it into separate classes, separate directories, and otherwise keep the code separate but still operating from the same PHP scripts. However, we will be sure that the implementation maintains these abstractions—the user interface code will only talk to the business logic, and the business logic will be the code that manages the databases and auxiliary files necessary for implementation.

For example, our business logic might want to have a “user” object. As we design the application, we might come up with a `User` and a `UserManager` object. For both objects, we would define a set of operations and properties for them that our user interface code might need.

```
User:
{
    Properties:
    - user id
    - name
    - address, city, province/state, zip/postal code, country
    - phone number(s)
    - age
    - gender
    - account number

    Operations:
    - Verify Account is valid
    - Verify Old Enough to create Account
    - Verify Account has enough funds for Requested Transactions
    - Get Purchase History for User
    - Purchase Goods
}

userManager:
{
    Properties:
    - number of users
```

```
Operations:  
- Add new user  
- Delete User  
- Update User Information  
- Find User  
- List All Users  
}
```

Given these crude specifications for our objects, we could then implement them and create a *bizlogic* directory and a number of *.inc* files that implement the various classes needed for this functionality. If our *UserManager* and *User* objects were sufficiently complex, we could create a separate *userman* directory for them and put other middle-tier functionality, such as payment systems, order tracking systems, or product catalogues into their own directories. Referring back to Chapter 3, “Code Organization and Reuse,” we might choose a layout similar to the following:

Web Site Directory Layout:

```
www/  
  generatepage.php  
  uigeneration.inc  
images/  
  homepage.png  
bizlogic/  
  userman/  
    user.inc  
    usermanager.inc  
  payments/  
    payment.inc  
    paymentmanager.inc  
  catalogues/  
    item.inc  
    catalogue.inc  
    cataloguemanager.inc  
  orders/  
    order.inc  
    ordermanager.inc
```

If we ever completely changed how we wanted to store the information in the database or how we implemented our various routines (perhaps our “Find User” method had poor performance), our user interface code would not need to change. However, we should try not to let the database implementation and middle tier diverge too much. If our middle tier is spending large amounts of time creating its object model on top of a database that is now completely different, we are unlikely to have an efficient application. In this case, we

might need to think about whether we want to change the object model exposed by the business logic (and therefore also modify the front end) to match this, or think more about making the back end match its usage better.

What Goes into the Business Logic

Finally, in the middle tier of our web applications, we should change how we consider certain pieces of functionality. For example, if we had a certain set of rules to which something must comply, there would be an instinctive temptation to add PHP code to verify and conform to those rules in the implementation of the middle tier. However, rules change—often frequently. By putting these rules in our scripts, we have made it harder for ourselves to find and change them and increased the likelihood of introducing bugs and problems into our web application.

We would be better served by storing as many of these rules in our database as possible and implementing a more generic system that knows how to process these rules from tables. This helps reduce the risk and cost (testing, development, and deployment) whenever any of the rules change. For example, if we were implementing an airline frequent flyer miles management system, we might initially write code to see how miles can be used:

```
<?php
  if ($user_miles < 50000)
  {
    if ($desired_ticket_cat == "Domestic")
    {
      if ($destination == "NYC" or $destination == "LAX")
      {
        $too_far = TRUE;
      }
      else if (in_range($desired_date, "Dec-15", "Dec-31"))
      {
        $too_far = TRUE;
      }
      else ($full_moon == TRUE and is_equinox($desired_date))
      {
        $too_far = FALSE; // it's ok to buy this ticket
      }
    }
    else
    {
      $too_far = TRUE;
    }
  }
  else
  {
    // etc.
  }
?>
```

Any changes to our rules for using frequent flyer miles would require changing this code, with a very high likelihood of introducing bugs. However, if we were to codify the rules into the database, we could implement a system for processing these rules.

MILES_REQUIRED	DESTINATION	VALID_START_DATE	VALID_END_DATE
55000	LAX	Jan-1	Dec-14
65000	LAX	Dec-15	Dec-31
55000	NYC	Jan-1	Dec-14
65000	NYC	Dec-15	Dec-31
45000	DOMESTIC	Jan-1	Dec-14
55000	DOMESTIC	Dec-15	Dec-31
etc...			

We will see more about how we organize our middle tier in later chapters, when we introduce new functionality that our business logic might want to handle for us.

Back End/Server

Our final tier is the place without which none of our other tiers would have anything to do. It is where we store the data for the system, validate the information we are sending and receiving, and manage the existing information. For most of the web applications we will be writing, it is our database engine and any additional information we store on the file system (most notably *.xml* files, which we will see in Chapter 23).

As we discussed in Chapter 8, “Introduction to Databases,” and Chapter 9, “Designing and Creating Your Database,” a fair amount of thought should go into the exact layout of your data. If improperly designed, it can end up being a bottleneck in your application that slows things down considerably. Many people fall into the trap of assuming that once the data is put in a database, accessing it is always going to be fast and easy.

As we will show you in the examples throughout this book, while there are no hard and fast rules for organizing your data and other back end information, there are some general principles we will try to follow (performance, maintainability, and scalability).

While we have chosen to use MySQL as the database for the back-end tier of our application, this is not a hard requirement of all web applications. We also mentioned in Chapter 8 that there are a number of database products available that are excellent in their own regard and suited to certain scenarios. We have chosen to use MySQL due to its popularity, familiarity to programmers, and ease of setup.

However, as we show in Appendix B, “Database Function Equivalents,” there is nothing preventing us from trying others. And, having chosen a 3-tier layout for our web applications, the cost of switching from one database to another is greatly reduced (though still not something to be taken lightly).

n-Tier Architectures

For complicated web applications that require many different pieces of functionality and many different technologies to implement, we can take the abstraction of *tiers* a bit further and abstract other major blocks of functionality into different modules.

If we were designing a major E-Commerce application that had product catalogs, customer databases, payment processing systems, order databases, and inventory systems,

we might want to abstract many of these components so that changes in any of them would have minimal impact on the rest of the system (see Figure 13-5).

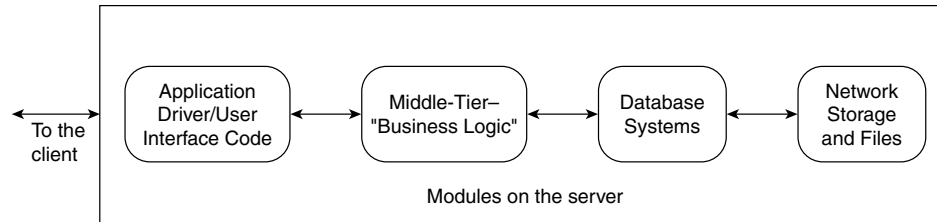


Figure 13-5: An n-tiered web application setup example.

We should be constantly thinking about maintaining a balance between abstraction and its cost. The more we break areas of functionality into their own modules, the more time we spend implementing and executing code to communicate between the different modules. If we see something that is likely to be reused, has a particular implementation that lends itself to an isolated implementation, or is very likely to change, then we would probably consider putting it in its own module. Otherwise, we should be asking ourselves about the benefits of doing this.

Any way that we decide to implement our web application, we will stress the importance of planning in advance. Before you start writing code, you should have an idea of how the various layers of the application will be implemented, where they will be implemented, and what the interface will look like. Not having this in mind before you begin is a sure way to guarantee wasted time as problems crop up. (This is not to say that a well-planned system will not encounter problems as it is being implemented, but the goal is to make them fewer and less catastrophic.)

Performance and Stability

We have frequently mentioned that the performance and scalability of our web applications are of high concern to us, but we must first define these. Without this, it is difficult to state what our goals are or to decide how to measure our success against them.

Performance

Performance is very easy to define for our web applications. It is simply a measure of how much time elapses between when the user asks for something to happen and his receiving confirmation of its happening, usually in the form of a page. For example, an E-Commerce shoe store that takes 20 seconds to show you any pair of shoes you ask to see is not going to be perceived as having favorable performance. A site where that same operation takes 2 seconds will be better received by the user. The potential locations for performance problems are many. If our web application takes forever to compute things before sending a response to the user, our database server is particularly slow, or our computer hardware is out of date, performance may suffer.

However, beyond things over which we have complete control, there are other problems that can adversely affect our performance. If our ISP (Internet Service Provider) or the entire Internet slows down significantly (as it has a tendency to do during large news events or huge virus/worm outbreaks), our web site's performance could suffer. If we find ourselves under a *denial-of-service* (DoS) attack, our web site could appear unusually slow and take many seconds to respond to seemingly simple requests.

The key item to remember is that the user does not know—and probably does not care—what the source of the problem is. To his eyes, our web application is slow, and after a certain point, it irritates or discourages him from using our application. As web application authors, we need to constantly be thinking about the potential problems we might encounter and ways to deal with them as best we can.

Scalability

While there is a tendency to group them together, scalability and performance are very different beasts. We have seen that performance is a measure of how quickly a web site or web application responds to a particular request. Scalability, on the other hand, measures the degree to which the performance of our web site degrades under an increasing load.

A web application that serves up a product catalog page in 0.5 seconds when 10 users are accessing the site but 5 seconds when 1,000 users are using the site sees 1,000 percent degradation in performance from 10 to 1,000 users. An application that serves up a product catalogue in 10 seconds for 10 users and 11 seconds for 1,000 users only sees 10 percent degradation in performance and has better scalability (but obviously worse performance)!

While we will not argue that the latter server is the better setup, it serves to demonstrate that observing great performance on a simple test system is not sufficient—we need to think about how that same application will respond to many users accessing it at the same time. As we will discuss in Chapter 29, “Development and Deployment,” you will want to consider using testing tools that simulate high loads on your machines to see how the application responds.

Improving Performance and Scalability

Many people, when faced with the possibility of suboptimal performance or scalability, suggest that you “throw more hardware” at the problem—buy a few more servers and spread the load across more machines. While this is a possible approach to improving performance, it should never be the first one taken or considered.

If our web application was so poorly designed that adding a \$20,000 server only gave us the ability to handle 10 more users at a time or improved the speed of our product catalog listings by 5 percent, we would not be spending our money wisely.

Instead, we should focus on designing our applications for performance, thinking about potential bottlenecks, and taking the time to test and analyze our programs as we are writing them. Unfortunately, there are no hard and fast rules or checklists we can consult to find the magic answers to our performance problems. Every application has different requirements and performance goals, and each application is implemented in a very different way.

However, we will endeavor to show you how we think about performance in the design of any samples or web applications we develop.

Summary

Writing web applications without a fundamental understanding of the underlying technologies is an unwise task. By having a solid foundation for the workings of the Internet and the various technologies we use in our programs, we can help design them for optimal functioning, security, and performance. While it is not necessary to become an expert in these areas, we should at least have a passing familiarity with how they work.

In this chapter, we discussed the various Internet technologies, such as the TCP/IP protocol, HTTP, and Secure Sockets Layer (SSL), and we have defined web applications.

We then looked at how we might go about implementing web applications, specifically looking at 3-tiered and n-tiered application architectures. By dividing our applications into presentation, business logic, and database layers, we can organize our code and reduce the chance of bugs when we make modifications later.

With this basic understanding of technologies and our web applications, the next chapter will discuss how to design and implement the user interface portions of our web application. We will see that this is more than simply writing HTML pages and adding flashy graphics.