NEAR EAST UNIVERSITY

Faculty of Engineering

Department of Computer Engineering

E-COMMERCE USING JAVA SERVLET PROGRAMMING

Graduation Project
COM-400

Student: Altyep Saada

Supervisor: Assoc.Prof.Dr Adnan Khashman

Nicosia – 2002

ACKNOWLEDGMENTS

"First, I would like to thank my supervisor Assoc.Prof.Dr.Adnan Khashman for his invaluable advice and belief in my work and my self over the course of this Graduation Project.

Second, I would like to say big thank you java servlet programming, for the knowledge that give it to me, and the experience and the answers of most questions of the network and the internet that I am always ask my self about it.

Third, I would like to thank my friends for their advice and support.

Finally, but no means least, thanks Mom and Dad for their years of support, and to my brother engineer wasem for help me to learn java programming language, and for advice me during writing my java servlet code. "

ABSTRACT

There is much excitement over the Internet and the World Wide Web. The Internet ties the "information world" together. The World Wide Web makes the Internet easy to use and gives it the flair and sizzle of multimedia. Java provides a number of built- in networking capabilities that make it easy to develop Internet-based and Web-based applications. Java's network capabilities are grouped into several packages. The fundamental networking capabilities are defined by classes and interfaces of package java.net, through which java offers socket-based communications that enable applications to a socket as simply as reading from a file or writing to a file. The classes and interfaces of package java.net also offer packet-based communications that enable individual packets of information to be transmitted-commonly used to transmit audio and video over the Internet.

The project discuss of networking over the next chapters focuses on both sides of a client-server relationship. The client requests that some action be performed and server performs the action and responds to the client. This request-response model of communication is the foundation for the highest-level view of networking in java – servlets. A servlet extends the functionality of a server. The javax.servlet package and the javax.servlet.http package provide the classes and interfaces to define servlets. The servlets enhance the functionality of World Wide Web servers- the most common form of servlet today. Servlet technology today is primarily designed for use with the HTTP protocol of the World Wide Web, but servlets are being developed for other technologies. Servlets are effective for developing Web-based solutions that help provide secure access to a Web site, that interact with databases on behalf of a client, that dynamically generate custom HTML documents to be displayed by browsers and that maintain unique session information for each client.

TABLE OF CONTENTS

INTRODUCTION

The rise of server-side java applications is one of the latest and most exciting trends in java programming. The java language was originally intended for use in small, embedded devices. It was first hyped as a language for developing elaborate client-side web content in the form of applets. Until recently, java's potential as a server-side development platform had been sadly overlooked. Now, java is coming into its own as a language ideally suited for server-side development.

Business in particular has been quick to recognize java's potential on the server-java is inherently suited for large client/server applications. The cross platform nature of java is extremely useful for organizations that have a heterogeneous collection of servers running various flavors of the UNIX and windows operating systems. Java's modern, object-oriented, memory-protected design allows developers to cut development cycles and increase reliability. In addition, java's built-in support for networking and enterprise APIs provides access to legacy data, easing the transition from older client/server systems.

Java servlets are key component of server-side java development. A servlet is a small, pluggable extension to a server that enhances the server's functionality. Servlets allow developers to extend and customize any java-enabled server-a web server, a mail server, an application server, or any custom server-with a hitherto unknown degree of portability, flexibility, and ease.

In this project the portability, flexibility, are considered. The project consists of seven chapters and conclusion.

Chapter one describe the history of web applications and the support of servlets where the user can found the servlet and why servlet programming.

Chapter two describe some things of HTTP servlets that can be doing, such as HTTP basics, requests, responses, headers. The two most important methods GET and POST, class HTTPServlet, and all the methods of this class. Two interfaces classes which is HttpServletRequest and HttpServletResponse and all the methods of these interfaces.

simple Hello World class how the user can run this code and what the output look like to the user, and final topic which is the servlet chaining filters.

Chapter three describe the servlet life cycle. Init and destroy methods. Status code, HTTP headers, and exceptions in the servlet when an unexpected error occurs how the servlet by using some ways of the exception and logging can be caught these errors.

Chapter four which is the main chapter, it describes all the methods can be used to write the servlets. Web pages session and cookies and all the methods of each one.

Chapter five describes the security of the servlet, HTTP authentication, Digital Certificates, Secure Sockets Layer (SSL), and finally Running Servlets Security.

Chapter Six describe database connectivity, how the servlet can hold database by using the servlet and java database connectivity (JDBC) by using some method and classes.

Chapter Seven describe the hall program and the result of the program, how the output looks like.

Finally, the conclusion section presents the important results obtained within the project.

Chapter One
Background of Servlets

1.1 Overview

The java server API is an asset of classes, a framework, for the development of IP servers, especially http servers. The server API is a standard *extension* to Java. Which means that it is not included in the base Java release? If has being used in a program, first thing need to download and install the necessary classes from sun. The servlet API it does not use in an applet. There are many interesting aspects of the server API; the most interesting being a concept called servlets; these will be presented this chapter.

1.2 History of web applications

While servlets can be used to extend the functionality of any java –enabled server, today they are most often used to extend web servers, providing a powerful, efficient replacement for CGI scripts. When a servlet is used to create dynamic content for a web page or otherwise extend the functionality of web server, in effect a *web application* is being created. While a web page merely displays static content and lets the user navigate through that content, a web application provides a more interactive experience. A web application keyword search on a document archive or as complex as an electronic storefront. Web applications are being deployed on the internet and on corporate intranets and extranets, where they have the potential to increase productivity and change the way that companies, large and small, do business.

There are some approaches that can be used to create web applications:

- **Common Gateway Interface**

The common gateway interface referred to as CGI, was one of the first practical techniques for creating dynamic content. With CGI, a web server passes certain requests to an external program. The output of this program is then sent to the client in place of a static file. The advent of CGI made it possible to implement all sorts of new functionality in web pages, and CGI quickly became a defacto standard, implemented on dozens of a web servers.

It's interesting to note that the ability of CGI programs to create dynamic web pages is a side effect of its intended purpose: to define a standard method for an information server to talk with external applications. The origin explains why CGI has perhaps the worst life cycle imaginable. When a server receives a request that accesses a CGI program, it must create a new process to run the CGI program and then pass to it, via environment variables and standard input, every bit of information that might be necessary to generate a response. Creating a process for every such request requires time and significant server resources, which limits the number of request a server can handle concurrently.

Figure 1.1 shows the CGI life cycle.

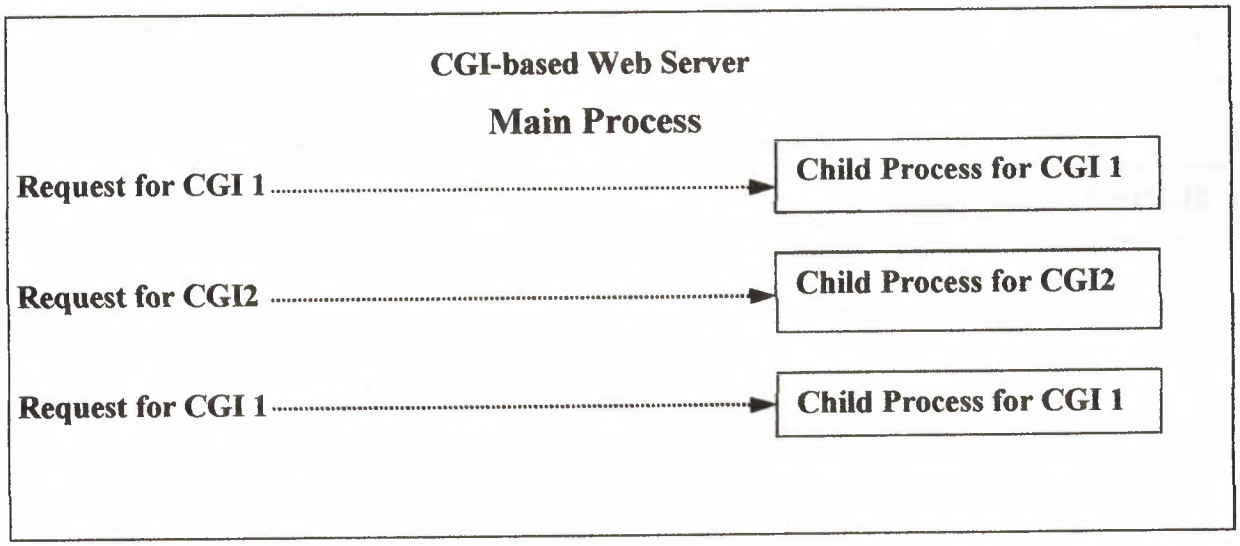

Figure 1.1. The CGI life cycle

Even though a CGI program can be written in almost any language, the Perl programming language has became the predominate choice. Its advanced text-processing capabilities are a big help in managing the details of the CGI interface. Writing a CGI script in perl gives it a semblance of platform independence, but it also requires that each request start a separate Perl interpreter, which takes even more time and requires extra resources.

Another often –overlooked problem with CGI is that a CGI program cannot interact with the web server or take a advantage of the server's abilities once it begins execution

because it is running in a separate process. For example, a CGI script cannot write to the server's log file.

- **FastCGI**

A company named Open Market developed an alternative to standard CGI named FastCGI In many ways, FastCGI works just like CGI –the important difference is that FastCGI creates a single persistent process for each FastCGI program. As shown in Figure 1.2. This eliminates the need to create a new process for each request.

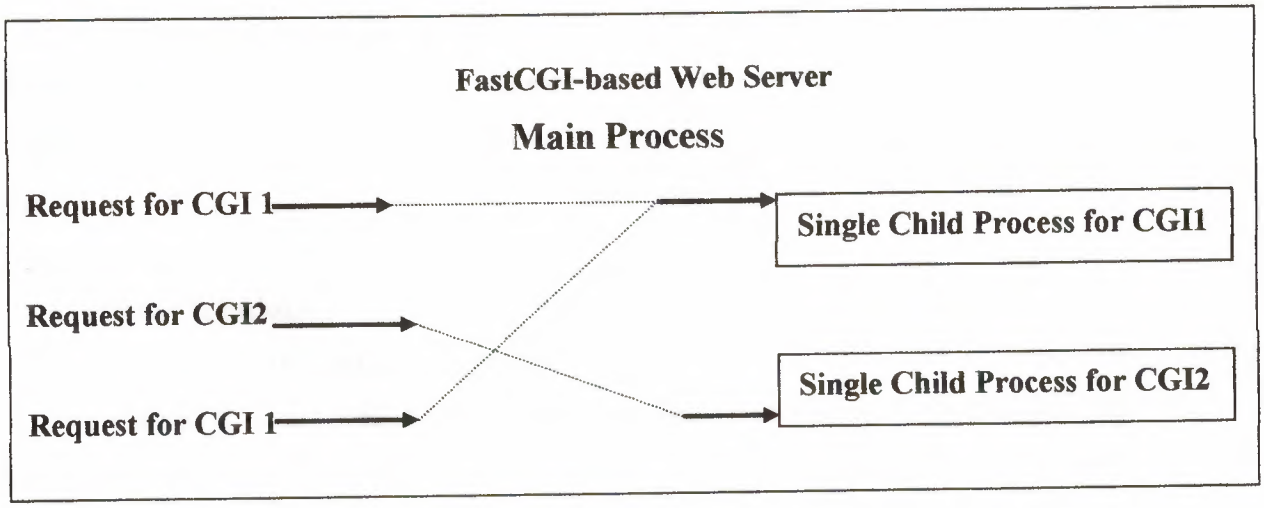

Figure 1.2. The FastCGI life cycle

Although FastCGI is a step in the right direction, it still has a problem with process proliferation: there is at least one process for each FastCGI program, if a FastCGI program is to handle concurrent requests, it needs a pool of processes, one per request. Consider that each process may be executing a Perl interpreter, this approach does not scale as well as one might hope. (Although, to its credit, FastCGI can distribute its processes across multiple servers.) Another problem with FastCGI is that it does nothing to help the FastCGI program more closely interact with the server. As of this writing, the FastCGI approach has not been implemented by some of the more popular servers, including Microsoft's Internet Information server.

Finally, FastCGI programs are only as portable as the language in which they're written.

- **mod_perl**

If the apache web server is used, another option for providing CGI performance is using mod_perl is a module for the Apache server that embeds a copy of the Perl interpreter into the Apache httpd executable. Providing complete access to Perl functionality within Apache. The effect is that your CGI scripts are precompiled by the server and executed without forking, thus running much more quickly and efficient.

- **PerlEx**

PerlEx. Developed by ActiveState, improves the performance of CGI scripts written in Perl that run on Windows NT web servers (Microsoft's FastTrack Server and Enterprise Server). PerlEx uses the web Server's native API to achieve its performance gains.

- **Other Solutions**

CGI/Perl has the advantage of being a more –or –less platform –independent way to produce dynamic web content. Other well-known technologies for creating web applications, such as ASP and server-side JavaScript, are proprietary solutions that work only with certain web servers.

- **Server Extension APIs**

Several companies have created proprieties server extension APIs for their web servers. For example, Netscape provides an internal API called NSAPI (now becoming WAI) and Microsoft provides ISAPI. Using one of these APIs, you can write server extensions that enhance or change the base functionality of the server, allowing the server to handle tasks that were once relegated to external CGI programs. As can be seen in Figure 1.3, server extensions exist within the main process of a web server.

Because server-specific APIs use linked C or C++ code, server extensions can run extremely fast and make full use of the server's resources. Server extensions, however, are not perfect solutions by any means. Besides being difficult to develop and maintain, they pose significant security and reliability hazards: a crashed server extension can bring down the entire server, and, of course, properties server extension are inextricably

tied to the server API for which they were written –and often tied to a particular operating system as well

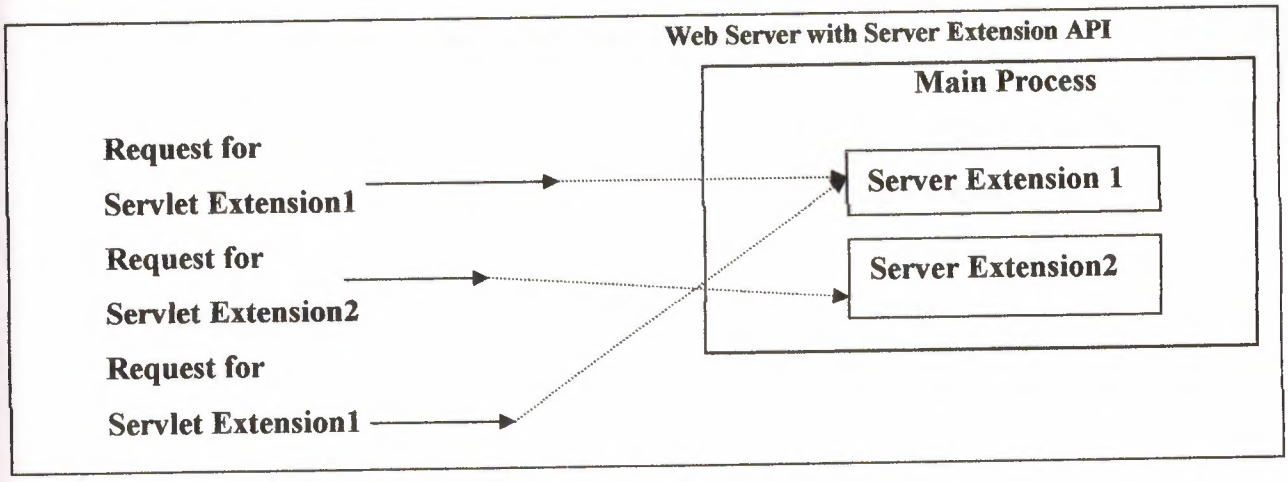

Figure 1.3. The server extension life cycle

- **Active Server Pages**

Microsoft has developed a technique for generating dynamic web content called Active Server Pages, or sometimes just ASP. With ASP, an HTML page on the web server can contain snippets of embedded code (usually VBScript or just Jscript although it's possible to use nearly any language). This code is read and executed by the web server before it sends the page to the client. ASP is optimized for generating small portions of dynamic content.

- **Server –side JavaScript**

Netscape too has a technique for server –side scripting, which it calls server side JavaScript, or SSJS for short. Like ASP, SSJS allows snippets of code to be embedded in HTML pages to generate dynamic web content. The difference is that SSJS uses JavaScript as the scripting language. With SSJS. Web pages are precompiled to improve performance.

- **Java Servlets**

Servlets are Java programs that make servers extensible. Much as you can load applets into a web browser, you can load servlets into a running server to extend its capabilities, servlets are commonly used with web servers, where they can take the

place of CGI scripts. A servlet is similar to a proprietary server extension, except that it runs inside a Java Virtual Machine (JVM) on the server (see the figure 1.4). Like applets, the byte codes for servlets can be read from the local file system or from the network. In the future, it will even be possible for clients to upload servlets to server to be run there, just as today clients download and run applets from a server.

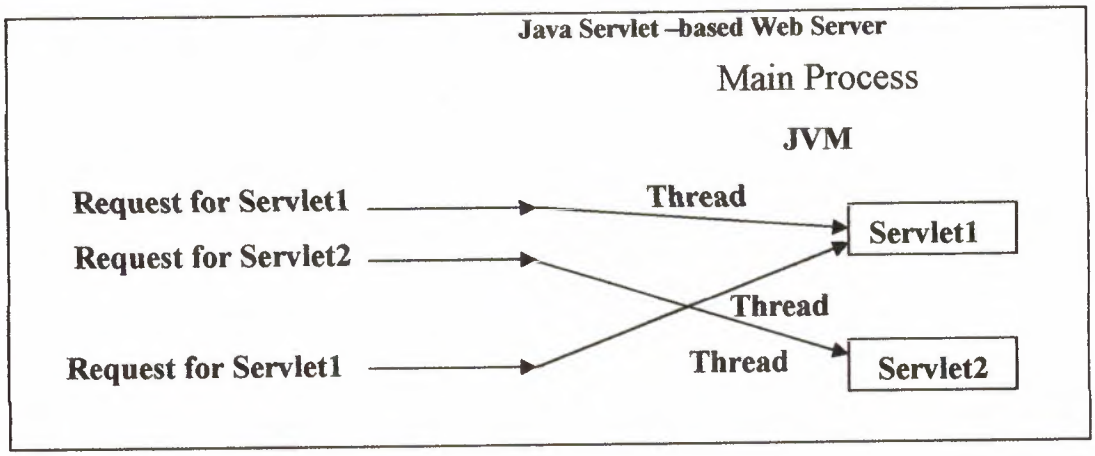

Figure 1.4. The servlet life cycle

A servlet can return data to the client as a new web page, as servlets themselves are faceless; they do not have a graphical interface or output stream to which they can send output. They rely on the web client to provide these services.

Servlets provide yet another way for a web to generate dynamic data. Normally, web server receives a URL requesting a certain page. The server maps this URL to a file, reads the file from the local disk, and sends the data to the client. In this situation, the data is static; it doesn't change until the server administrator changes the file. However a servlet can generate a new data for each request because it is a program. Thus it can deliver changing information such as a time of day or a stock price.

Of course, that's nothing really new; CGI programs can also dynamically generate data to send to a client. How does a servlet differ from a CGI program? Why should consider writing servlets, rather than sticking with CGI? Those are good questions. CGI has been around for a while, and it's well understood. It's easy to write a web page that invokes a CGI program, and easy to write a program that sends data back to the browser. Furthermore, virtually all web servers understand CGI; currently, Jeeves (a

demonstration server that sun supplies with the server distribution) is the only server that can handle servlets through there are more to come. Servlets are interesting precisely because they are more than just CGI programs written in Java; servlets can do many things that CGI programs cannot:

1. A servlet can contain continue to run in the background after it has finished processing a request so it's ready to process the next request without incurring additional startup costs. On an active server, the overhead of staring CGI programs is significant. Furthermore, a servlet can use threads to process simultaneous requests efficiently.aservlet can even pass data between multiple connections. For example, a servlet can act as a multiplayer game server that listens for input from multiple clients. And then broadcasts that data to all connected clients.

2. A servlet can communicate interactivity with an applet on the client. A CGI program receives a request from a client, and then sends a response; at that point the connection is closed. The client cannot send another request to the CGI program in response to the data it received. In constrast, a servlet –applet pair can carry on a conversion. Making many data transfers between the client and the server .they can even implement a new protocol if necessary. This is much easier to code than a CGI program that stores form of state on the server or in the URL.

3. Servlets can originate on the client. Like applets, servlets run in secure environment. So the server does not need to worry about hostile servlets. For example, a client can upload accustom that searches a web site for information. With local access to the files, the search can take place much more quickly than it would if the client had to download each file on the web site. Web spiders and indexers like Lycos can become far more efficient, and use far less internet bandwidth, since only the results of a site index need to be transmitted, not every page on the web site.

4. Servlets are a step towards agents. A servlet can be uploading to different servers technologies, performing the same action on each server in run. Until now, agent technologies shared several server limitations. First, hosts had to true the agents. Using java's security features, a web server can execute a servlet without worrying

that it may crash the system or open a security hole. Second agents could only run on certain platforms; java's portability means that servlets don't care what kind of system they run on. The possibilities for intelligent agents are almost dimities, including shopping agents that search for the best prices, clipping service agents that continuously comb the net for information, system management servlets that upload mirror sites with copies or changes files, servlets that back up data to a central host. And more. However servlets are not yet true agents because they must be explicitly invoked and moved by something other that themselves; that may change the future.

1.3 Supports for Servlets

Like java itself, servlets were design for portability. Servlets are support on all platforms that support java, and servlets work with all the major web servers. Java servlets, as defined by the java software division of sun Microsystems (formerly known as Java Soft), are the first standard extension to java. This means that servlets are efficiently blessed by sun and are part of the java language, but they are not part of the core java API.therefore, althougth they may work with any java Virtual Machine (JVM), servlet classes need not be bundled with all JVMs.

To make it easy for us to develop servlets. Sun has made publicly available a set of classes that provide basic servlet support. The **javax.servlet** and **javax.servlet.http** packages constitute this servlet API.Version 2.0 of these classes comes bundled with the java servlet development kit (**JSDK**) for use with the java Development kit 1.1 and above: the **JSDK** available for download from **http://java.sun.com/products/servlet/.** Many web server vendors have incorporated these classes into their servers to provide servlet support, and several have also provided additional functionality. Sun's Java Web Server, for instance, includes a proprietary interface to the server's security features.

It doesn't much matter where to get the servlet classes, as long as to have them on systems, since it is necessary to compile the servlets. In additional to the servlet classes, there is a servlet engine, so that, it test and deploy the servlets. The choice of servlet engine depends in part on the web server(s) are running.
There are three flavors of servlet engines: **standalone**, **add-on**, and **embeddable**.

1.4 Why Servlet Programming

So far, there is portrayed servlets as an alternative to other dynamic web content technologies, but really do not explained why servlet. What makes servlets a viable choice for web development? Servlets offer a number of advantages over approaches, including: **portability, power, efficiency, endurance, safety, elegance, integration, extensibility, and flexibility.** Let's examine each in turn.

Portability

Because servlets are written in Java and conform to a well-define and widely accepted API, they are highly portable across operating systems and across server implementation. The servlet has the ability to develop on a Windows NT machine running the Java Web Server and later deploy it effortlessly on a high-end UNIX server running Apache. With servlets can be truly "Write Once, server everywhere."

Power

Servlet can harness the full power of the core Java APIs: networking and URL access, multithreading, image manipulation, data compression, database connectivity, internationalization, remote method invocation (RMI). CORBA connectivity, and object serialization, among others. If has been written a web application that allows employees to query a corporate legacy database, it should be consider the advantage of all of the Java Enterprise APIs in doing so. Or, if need to be created a web –based directly lookup application. Make used of the JNDI API.

Efficiency and Endurance

Servlets invocation is highly efficient. Once a servlet is loaded, it generally remains in the server's memory as a single object instance.thereafter; the server invokes the servlet to handle a request using a simple, lightweight method invocation. Unlike with CGI, there's no process to spawn or immediately. Multiple, concurrent requests are handled by separate threads, so servlets are highly scalable.

Servlets, in general, are naturally enduring objects. Because a servlet stays in the server's memory as a single instance, it automatically maintains its state and can hold on to external resources, such as database connections, that may otherwise take a several seconds to establish.

Safety

Servlets support safe programming practices on a number of levels. Because they are written in Java.servlets inherit the strong type safety or the Java language. In addition, the servlets API is implemented to type-safe. While most values in a CGI program, including a numeric item like a server port number, are treated as strings, values are manipulated by the servlets API using their native types, so a server port number is represented as an integer. Java's automatic garbage collection and lack of pointers mean that servlets are generally safe from memory management problems like dangling pointers, invalid pointer refernces, and memory leaks.

Servlets can handle errors safety, due to Java's exception-handling mechanism. If a servlet divides by zero or performs some other illegal operation, it throws an exception that can be safety caught and handled by the server, which can politely log the error and apologize to the user. If C++ based server extension were to make the same mistake, it could potentially crash server.

A server can further protect itself from servlets thought the use of Java security manager. A server can execute its servlets under the watch of a strict security manager that, for example, enforces as security policy designed to prevent a malicious or poorly written servlet from damagining the servers file system.

Elegance

The elegance of servlet code is striking. Servlet code is clean, object oriented, modular, and amazing simple. One reason for this simplicity is the servlet API itself, which includes methods and classes to handle many of the routine chores of the servlet development. Even advanced operations, like cookie handling and session tracking, are abstract into convenient classes. A few more advanced but still common tasks were left out of the API, and, in those places, we have tried to step in and provide a set of helpful classes in the com.oreilly. Servlet package.

Integration

Servlets are tightly integrated with the server. This integration allows a servlet to cooperate with the server in ways that CGI program cannot. For example, a servlet can

use the server to translates file paths, performe logging, check authorization, perform MIME type mapping, and, in some cases, even add users to the server's user database. Server-specific extensions can do much of this, but the process is usually much more complex and error-prone.

Extensibility and Flexibility

The servlet API is designed to be easily extensibility. As it stands today, the API includes classes that are optimized for HTTP servlets. But at a later date, it could be extended and optimized for another type of servles, either by sun or by a third party. It is also possible that its support for HTTP servlets could be further enhanced.

Servlets are also quite flexible. An HTTP servlet can be used to generate a complete web page: it can be added to static page using a <Servlet> tag in what's known as a server-side include: and it can be used in cooperation with any number of other servlets to filter content in something called a servlet chain. In addition, sun introduced Java Server Pages, which offer a way to write snippets of servlet code directly within static HTML page, using syntax that is curiously similar to Microsoft's Active Server Pages (ASP).

1.5 Summery

This chapter has being a short introduction in history of web application, support of servlets, and finally why the servlet, what the power of servlets.

Chapter Two
HTTP Servlet Basics

2.1 Overview

This chapter provides a quick introduction to some of the things an HTTP servlet can do. For example, an HTTP servlet can generate an HTML page, either when the servlet is accessed explicitly by name, by following a hypertext link. Or as the result of a form submission. An HTTP servlet can also be embedded inside an HTML page; where it functions as a server –side include.servlets can be chained together to produce complex effects-one common use of this technique is for filtering content.

2.2 HTTP Basics

Before has be shown a simple HTTP servlet, it should be understanding how the protocol behind the web, HTTP, works.

2.2.1 Requets, Response, and Headers

HTTP is a simple, stateless protocol. A client, such as a web browser, makes a request, the web server responds, and the transaction is done. When the client sends a request, the first thing it specifies is an HTTP command, called a *method* that tells the server the type of action it wants performed. This first line of the request also specifies the address of a document (a URL) and the version of the HTTP protocol it is using. For example:

GET / intro.html HTTP/1.0

This request uses the GET method to ask for the document named *intro.html,* using HTTP Version 1.0. After sending the request, the client can send optional header information to tell the server extra information about the request, such as what software the client is running and what content types it understands. This information doesn't directly pertain to what was requested, but it could be used by the server in generating its response. Here are some sample request headers:

User –agent: Mozilla/4.0 (compatible; MSIE 4.0; Windows 95)
Accept: image/gif, image/jpeg, text/ *, */*

The User-agent header provides information about the client software, while the Accept header specifies the media (MIME) types that the client prefers to accept. After the headers, the client sends a blank line, to indicate the end of the header section. The client can also send additional data, if appropriate for the method being used, as it is with the POST method that will be discussed shortly. If the request doesn't send any data, it ends with an empty line.

After the client sends the request, the server processes it and sends back a response. The first line of the response is status line that specifies the version of the HTTP protocol the Server is using, a status code, and a description of the status code. For example:

HTTP/1.0 200 OK

This status line includes a status code of 200, which indicates that the request was successful, hence the description "OK". Another common status code is 404, with the description "Not Found"-as can be guessed; it means that the requested document was not found.

After the status line, the server sends response headers that tell the client things like what software the server is running and the content type of the server's response. for example:

Date: Saturday, 23-May-98 03:25:15 GMT
Server: JavaWebServer/1.1.1
MIME-Version: 1.0
Content-type: text/html
Content-length: 1029
Last-modified: Thursday, 7-May-98 12:15:35 GMT

The server header provides information about the server software, while the Content-type header specifies the MIME type of data included with the response. The server sends a blank line after the headers, to conclude the header section. If the request was successful, the requested data is then sending as part of the response. Otherwise, the response may contain human-readable data that explains why the server couldn't fulfill the request.

2.2.2 Get and Post

When a client connects to a server and an HTTP request can be of several different types, called methods. The most frequently used methods are GET and POST. PUT simply, the GET method is designed for getting information (a document, a chart, or the results from a database query), while the POST method is designed for posting information (a credit card number, some new chart data, or information that is to be stored in a database). To use a bulletin board analog, GET is for reading and POST is for tacking up new material.

The GET method, although it's designed for reading information, can include as part of the request some of its own information that better describes what to get such as an x, y scale for a dynamically created chart. This information is passed as a sequence of characters appended to the request. URL in what's called *a query string*. Placing the extra information in the URL in this way allows the page to be bookmarked or emailed like any other. Because GET requests theoretically shouldn't need to send large amounts of information, some servers limit the length of URLs and query strings to about 240 characters.

The POST method uses a different technique to the server because in some cases it may need to send megabytes of information. A POST request passes all its data, of unlimited length, directly over the socket connection as part of its HTTP request body. The exchange is invisible to the client. The URL doesn't change at all. Consequently, POST requests cannot be bookmarked or emailed, or in some cases, even reloaded. That's by the design- information send to the server, such as your credit card number, should be send sent only once.

In practice, the use of GET and POST has strayed from the original intent, it's common for long parameterized requests for information to use POST instead of GET to work around problems with overly-long URLs. It's also common for simple forms that upload information to use GET because, well-why not, it works!
Generally, this isn't much of a problem. Because they can be bookmarked so easily, should not be allowed to cause damage for which the client could be held responsible.

In other words, GET request should not be used to place an order, upload a database, or take an explicit client action in any way.

2.2.3 Other Methods

In addition to GET and POST, there are several other lesser-used HTTP methods. There's the HEAD method, which is sent by a client when it wants to see only the headers of the response, to determine the document's size, modification time, or general availability. There's also PUT, to place documents directly on the server, and DELETE, to do just the opposite. These last two aren't widely support due to complicated policy issues. The TRACE method is used as a debugging aid-it returns to the client the exact contents of its request. Finally, the OPTIONS method can be used to ask the server which means it supports or what options are available for a particular resource on the server.

2.3 The Servlet API

Architecturally, all servlets must implement the **Servlet** interface. As with many key applet methods, the methods of interface **Servlet** are invoked automatically (by the server on which the servlet installed). This interface defines five methods describes in figure 2.1. The servlet packages define two abstract classes that implement the interface Servlet-class **GenericServlet** (from the package **javax.servlet**) and class **HttpServlet** (from the package **javax.servlet.http**). These classes provide default implementation of all the Servlet methods. Most servlets extends either **GenericServlet** or **Httpservlet** and override some or all of their methods with appropriate customized behaviors.

Method	**Description**
Void service (ServletRequest request, ServletResponse response)	This is the first method called on every servlet to response to a client request.
String getServletInfo ()	This method is defined by a servlet programmer to return a String containing servlet information such as the servlet's author and version.

ServletConfig getServletConfig ()

This method returns a reference to an object That implements interface Servletconfig. This object provides access to the servlet's configuration information such as initialization parameters and the servlet's ServletContext, which provides the servlet with access to its environment.

Figure 2.1. Methods of interface **Servlet.**

2.4 HttpServlet class

Web-based servlets typically extends class **HttpServlet**. Class **HttpServlet** overrides method service to distinguish between the typical request received from a client web browser. The two most common HTTP request types (also known as request methods) are **GET** and **POST**. A **GET** request gets (or retrieves) information from the server. Common uses of **GET** requests are to retrieve an HTML document or an image. A POST request posts (or sends) data to the server. Common uses of **POST** requests are to send the server information from HTML form in which the client enters data, to send the server information so it can search the Internet or query a database for the client, to send authentication information to the server, etc.

Class **HttpServlet** defines methods **doGet** and **doPut** to response to **GET** and **POST** requests from a client, respectively. These methods are called by the **HttpServlet** class's service method, which is called when a request arrives at the server. Method service first determines the request type, and then calls the appropriate method.

Methods of class **HttpServlet** that response to other request types are shown in figure 2.3 (all receives parameters of type **HttpServletRequest** and **HttpServletResponse** and return void). Methods **doGet** and **doPost** receives as argument an **HttpServletRequest** object and an **HttpServletResponse** object that enable interaction between the client and the server.

Method	Description
doDelete	Called in response to an HTTP **DELETE** request. Such a request is normally used to delete a file from the server. This may not be available on some servers because of its inherent security risks.
doOptions	Called in response to an HTTP **OPTIONS** request. This returns information to the client indicating the HTTP options supported by the server.
doPut	Called in response to an HTTP PUT request. Such a request is normally used to store a file on the server. This may not be available on some servers because of its inherent security risks.
doTrace	Called in response to an HTTP TRACE request. Such a request is normally used for debugging. The implementation of this method automatically returns an HTML document to the client containing the request header information (data sent by the browser as part of the request).

Figure 2.2. Important methods of class **HttpServlet.**

2.4.1 HttpServletRequest Interface

Every call to **doGet** or **doPost** for an **HttpServlet** receives an object that implements interface **HttpServletRequest.** The web server that executes the servlet creates an HttpServletRequest object and passes this to the servlet's service method (which, in turn, passes it to **doGet** or **doPost**). This object contains the request from the client. A variety of methods are provided to enable the servlet to process the client's request. Some of these methods are form interface **ServletRequest**-the interface that **HttpServletRequest** extends. A few key methods used are presented in figure 2.4.

Method	Description
String getParameter (String name)	Return the value associated with a parameter to the servlet as part of a GET or POST request. The name argument represents the parameter name.
Enumeration getParameterNames()	

Return the names of all the parameters sent to the servlet as part of POST request.

String [] getParameterValues(String name)

Return an array of Strings containing the value for a specified servlet parameter.

Cookie [] getCookies ()

Returns an array of Cookie objects stored on the client by the server. Cookies can be used to uniquely identify clients to the servlet.

HttpSession getSession (Boolean create)

Return an HttpSession object associated with the client's current browsing session.

Figure 2.3. Important methods of interface HttpServletRequest.

2.4.2 HttpServletResponse Interface

Every call to **doGet** or **doPost** for an **HttpServlet** receives an object that implements interface **HttpServletResponse.** The web server that executes the servlet creates an HttpServletResponse object and passes this to the servlet's service method (which, in turn, passes it to **doGet** or **doPost**). This object contains the response from the client. A variety of methods are provided to enable the servlet to formulate the response to the client. Some of these methods are form interface **ServletResponse**-the interface that **HttpServletResponse** extends. A few key methods used are presented in figure 2.5.

Method	Description
void addCookie (Cookie cookie)	Used to add a cookie to the header of the response to the client. The Cookie's maximum age and wether the client allows Cookie to be saved determine whether or not Cookie will be stored on the client.

ServletOutputStream getOutputStream ()

Obtains a byte-based output stream that enables binary data to be sent to the client.

PrintWriter getWriter ()

Obtains a character-based output stream that enables text data to be sent to the client.

Void setContentType (string type)

Specifies the MIME type of the response to the browser. The mime type helps the browser determine how to display the data. For example, MIME type "text/html" indicates that the response is an HTML document, so the browser displays the HTML page.

Figure 2.4. Important methods of interface HttpServletResponse.

2.5 Page Generation

The most basic type of HTTP servlet generates a full HTML page. Such a servlet has access to the same information usually sent to a CGI script, plus a bit more. A servlet that generates an HTML page can be used for all the tasks where CGI is used currently, such as for processing HTML forms, producing reports from a database, taking orders, checking identifiers, and so forth.

2.5.1 Writing Hello World

Example below shows an HTTP servlet that generates a complete HTML page. This servlet just says "Hello World" every time it is accessed via a web browser.

Example 2-1. A servlet that prints "Hello World"

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class HelloWorld  extends HttpServlet{
```

```
public void doGet(HttpServletRequest req,HttpServletResponse res) throws
ServletException,IOException{

PrintWriter  out=res.getWriter();
res.setContentType("text/html");

out.println("<html><head><title>");
out.println("HelloWorld</title></head><body>");
out.println("<center><br><br><br><h1>Hello,World!</h1></center>");
out.println("</body></html>");
}}
```

This servlet extends the HttpServlet class and overloads the doGet () method inherited from it. Each time the web server receives a GET request for this servlet, the server invokes this doGet () method, passing it an HttpServletRequest object and an HttpServlet Response object.

The HttpServletRequest represents the client's request. This object gives a servlet access to information about the client, the parameters for this request, the HTTP headers passed along with the request, and so forth. After all, this example is going to say "Hello World" no matter what the request.

The HttpServletResponse represents the servlet's response. A servlet can use this object to return data to the client. This data can be of any content type, thought the type should be specified as part of the response. A servlet can also use this object to set HTTP response headers.

Our servlet uses the setContentType () method of the response object to set the content type of its response to "text/html", the standard MIME content type for HTML pages. Then, it uses the getWriter () method to retrieve a PrintWriter, the international-friendly counterpart to a PrintStream PrintWriter converts Java's Unicode characters to a locale-specific encoding. For an English locale, it behaves same as a PrintStream. Finally, the servlet uses this PrintWriter to send its "Hello World" HTML to the client.

2.5.2 Running Hello World

When developing servlets two things needed: the Servlet API class, which are used for compiling, and a servlet engine such as a web server, which is used for deployment. To obtain the servlet API class files, several options have needed:

- Install the Java Servlet Development Kit (JSDK), available for free at http://java.sun.com/proiducts/servlets/ JSDK2.0 Version 2.0 contains the class files for the servletAPI2.0, along with their source code and simple web server that acts as a servlet engine for HTTP servlets. It works with JDK1.1 and later.
- Install one of the many full – featured servlet engines, each of which typically bundles the Servlet API class files.

There are dozens of servlet engines available for servlet deployment, why not use the servlet engine included in jsdk2.0? Because the servlet engine is bare-bones simple. It implements the Servlet API 2.0 and nothing more. Features like robust session tracking, server-side includes, servlet chaining, and Java Server pages have been left out because they are technically not part of the Servlet API. For these features, has to be used a full-fledged servlet engine like the Java Web Server or one its competitors.

So, what has be doing to the code to make it run in a web server? Well, the server included with the JSDK2.1 is a small, multithreaded process that can run one or more servlets. Unlike some web servers, the JSDK server does not automatically reload updated servlets. However stopped and restarted the server with very little overhead to run a new version of a servlet.

2.5.3 Setting up the JSDK Server

It is possible to configure various properties of the JSDK server before starting it. These properties include the server's port, which defaults to `8080`, the hostname of the server, which defaults to local host, and the document root, which defaults to the webpages subdirectory of the JSDK installation. To look at or update these configuration values, edit the default.cfg file in the JSDK installation directory. Now after compile the servlet code by using javac compiler, take the (.class) file and put it in

Webpages/WEB-INF/servlets

2.5.4 Starting the JSDK Server

To start the server, use the Unix-based Korn-shell script or the Windows-based batch file that the JSDK provides in the installation directory of the JSDK.

The following command starts the server on UNIX:

% startserver

And the following command starts the server on Windows:

C:\jsdk\> startserver

2.5.5 Stopping the JSDK Server

To stop the server, the JSDK provides shut-down commands in the same directory as the start-up commands (the installation directory of the JSDK).

The following command stops the server on UNIX:

% stopserver

And the following command stops the server on Windows:

C:\jsdk\> stopserver

The output of HelloWorld similar to figure 2.3.

Figure 2.5. The Hello World servlet.

2.5.6 Configuring JSDK Servlets

Configure servlet applications that run on the JSDK 2.1 server by specifying properties. Properties are key-value pairs used for the configuration, creation, and initialization of a servlet. For example, test.code=Hello World is a property whose key is test.code and whose value is Hello World. Properties are stored in a text file with a

default name of servlets.properties. The file holds the properties for all the servlets that the servlet-running utility will run.

Using properties requires name of servlet. (The string catalog in the property names above is the catalog servlet's name.) The servlet name enables the servlet-running utilities to associate properties with the correct servlets. It is also the name that clients will use when they access the servlet.

2.5.7 Calling Servlets from a Browser

The URL for a servlet has the following general forms depending on which server has be using. For JSDK2.1, the URL is:

http://machine-name:port/servlet/Servlet-name

For example, to see the output of the previous example, type the following URL into the browser:

http://localhost:8080/servlet/test

The output like this:

Figure 2.6. The Hello World Servlet.

2.5.8 Handling form Data

The "Hello World" servlet is not very exciting, so let's try something slightly more ambitious. This time the user creates servlet by name. First, HTML form that ask the user for his or her name. The following page should suffice:

```
<html>
<head>
```

```
<title>Introduction</title>
</head>
<body>
<form action="/servlet/Hello"  method="get">
<p><em><b> if you donot mind me asking, what is your name?
<input  type="text" name="name">
</b></em></p>
<input  type="submit">
</form>
</body>
</html>
```

Figure 2.5 shows how this page appears to the user.

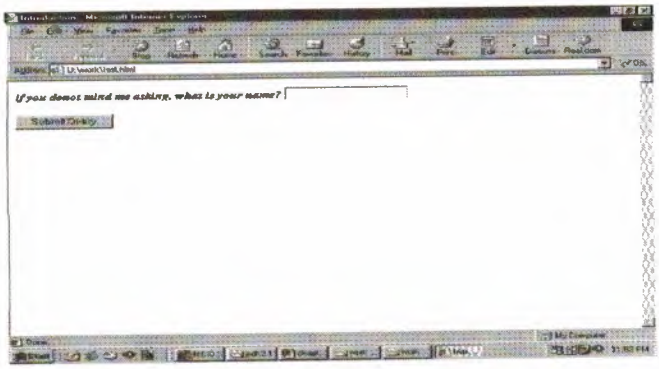

Figure 2.6. An HTML form

When the user submits this form, his name is sending to test servlet program because the action attribute has been set to point of the servlet. The form is using the GET method, so any data is appended to the request URL as query string. For example, if the user enters the name "Inigo Montoya", the request URL is "http://server:8080/servlet/Hello?name=Inigo+Montoya". The space in the name is specially encoded as plus sign by the browser because URLs cannot contain spaces.

A servlet HttpServletRequest object gives it access to the form data in its query string.

Example below shows a modified version of Hello servlet that uses its request object to

read the "name" parameter.

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
```

```
public class Hello extends HttpServlet{

 public void doGet(HttpServletRequest req,HttpServletResponse res) throws
ServletException,IOException
     {
     PrintWriter out=res.getWriter();
     res.setContentType("text/html");

    String  name=req.getParameter("name");
    out.println("<html>");
    out.println("<head><title> Hello,"+name+"</title></head>");
    out.println("<body>");
    out.println("Hello,"+name);
    out.println("</body></html>");
    }

   public  String getServletInfo(){
   return "A servlet that knows the name of the person to whom it's"+
   "saying hello";
    }
}
```

This servlet nearly identical to the HelloWorld servlet. The most important change is that it now calls req.getParameter ("name") to find out the name of the user and then prints this name. The getParameter () method gives a servlet access to the parameters in its query string. It returns the parameter's decoded value or null if the parameter was not specified. If the parameter was send but without a value, as in the case of an empty form field, getParameter () returns the empty string.

This servlet also adds a getServletInfo() method. A servlet can override this method to return descriptive information about itself, such as its purpose, author, version, and/or copyright. Its skin to an applet's getAppletInfo (). The method is used primarily for putting explanatory into a web server administration tool.

Figure 2.6. Shows how this page appears to the user.

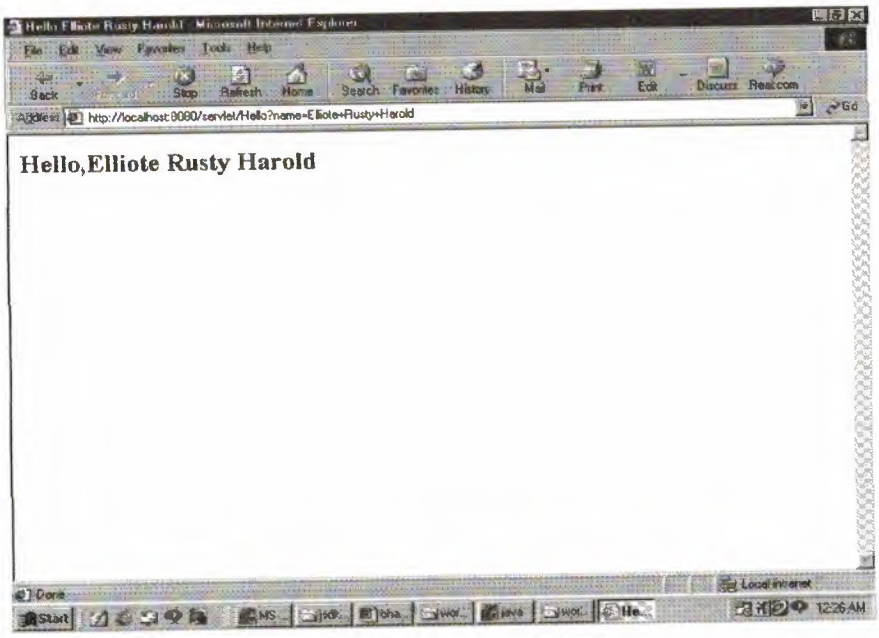

Figure 2.7. The Hello servlet using form data.

2.5.9 Handling Post Requests

Until now the servlets has be using the doGet () method. Now let's change Hello servlet to handle POST request. Because the behavior of with POST is the same for GET, simply dispatch all POST requests to the doGet () method with the following code:

```
Public void doPost (HttpServletRequest req, HttpservletRespone res) throws ServletException, IOException {
doGet ();
}
```

Now the Hello servlet can handle form submissions that use the POST method:

```
<form action ="/servlet/Hello" method="post">
```

in general, it is best if a servlet implements either doGet() or doPost(). Deciding which to implement depends on what sort of request the servlet needs to be able to handle. The code has be written to implement the methods is almost identical. The major difference is that the doPost () has the added ability to accept large amounts of inputs.

May be wondering what would have happened had the Hello servlet been accessed with POST request before implementing doPost (). The default behavior inherited from HttpServlet for both doGet () and doPost () is to return an error to the client saying that the requested URL does not support that method. As seeing from the figure 2.6.

2.5.10 Handling HEAD Requests

A bit of under-the- covers magic makes it trivial to handle HEAD requests (send by a client when it wants to see only the headers of the response). There is no doHead() method to write. Any servlet that subclasses HttpServlet and implements the doGet () method automatically supports HEAD requests.

Here's how it works. The service () method of the HttpServlet identifies HEAD requests and treats them specially. It constructs a modified HttpServletResponse object and passes it, along with an unchanged request, to the doGet() method. The doGet () method proceeds as normal, but only the headers it sets are returned to the client. The special response object effectively suppresses all body output. Figure 2-7 shows an HTTP servlet handle HEAD requests.

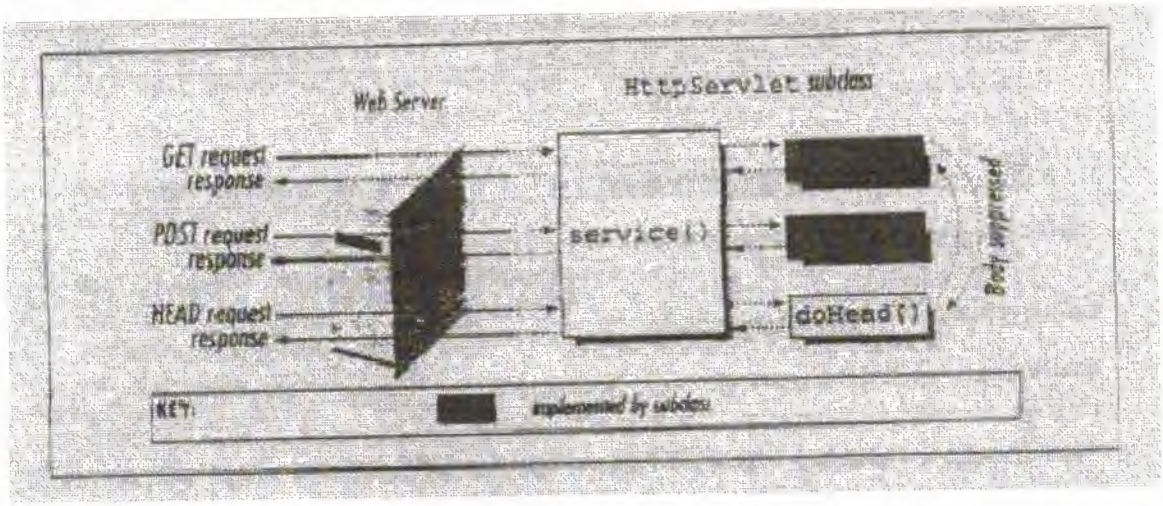

Figure 2.8. An Http servlet handling a HEAD request

2.6 Server-side includes

Servlets can also be embedded in an HTML page using the *server-side include (SSI).*in this case, the servlet only produces a part of the HTML that is sent to the client. For example, a servlet could return live data, like the current time, the number of hits to

a page, or the stock price of the company serving the page at a given moment. The rest of the page could display logos, headers, footers, setup backgrounds, and provide other information that doesn't change. To embed a servlet in an HTML file as a server-side include, use the <servlet> tag:

```
<servlet code=servletName codebase=http://server: port/dir
        initParam1=initValue1   initParam2=initValue2>
<param name=param1 value=value1>
<param name=param2 value=value2>
    If has being see this text, that means the web server
    Providing this page does not support the servlet tag.
</servlet>
```

The code attribute specifies the class name or registered name of the servlet to invoke. The codebase attribute is optional. It can refer to a remote location from which the servlet should be loaded. Without a codebase attribute, the servlet is assumed to be local.

Any number of parameters may be passed to the servlet using the<param> tag. The servlet can retrieve the parameter values using the getParameter () method of servletRequest. Any number of initialization (init) parameters may also be passed to the servlet appended to the end of the <servlet> tag.

A server that supports SSI detects the <servlet> tag in the process of returning the page and substitutes in its place the output from the servlet (as shown in figure 2.8). The server does not parse every page it returns, just those that are specially tagged. The java web server, by default. Parses only pages with a .shtml extension. Note that with the<servlet>, unlike the <applet> tag, the client browser never sees anything between <servlet> and

</servlet> unless the server does not support SSI, in which case the client receives the content, ignores the unrecognized tags, and displays the descriptive text.

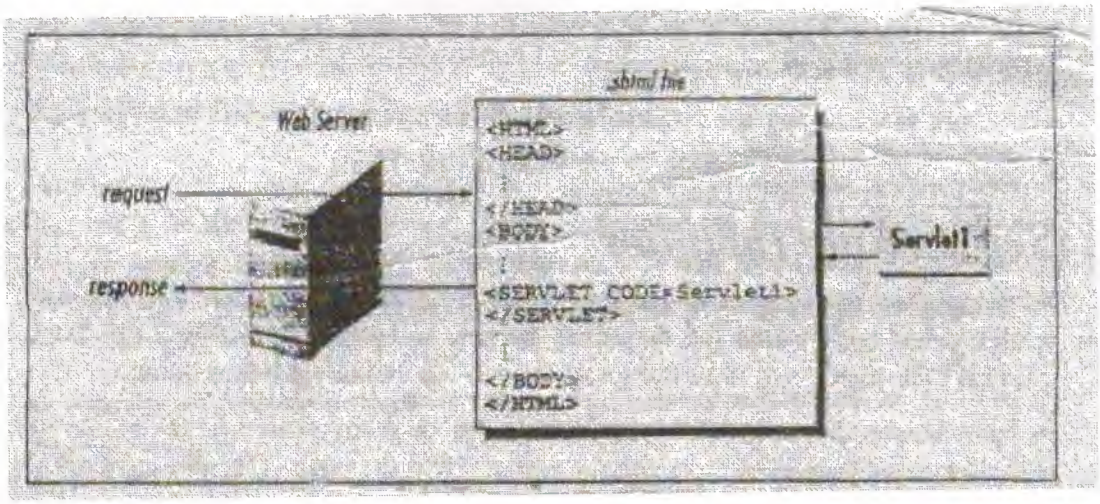

Figure 2.9. Server-side include expansion

2.7 Servlet Chaining and Filters

Now have be seen how an individual servlet can create content by generating a full page or by used in a server-side include (SSI). Servlets can also cooperate to create content in a process called *servlet chaining.*

In many servers that support servlets, a request can be handled by a sequence of servlets.

The request from the client browser is sent to the first servlet in the chain. The response from the last servlet in the chain is returned to the browser. In between, the output from each servlet is passed (piped) as input to the next servlet, so each servlet in the chain has option to change or extend the content. As shown in figure 2.10. There are two common ways to trigger a chain of servlets for an incoming request. First, tell the server that certain URLs should be handled by an explicitly specified chain. Or, tell the server to send all output of a particular content type through a specified servlet before it is returned to the client, effectively creating a chain on the fly.

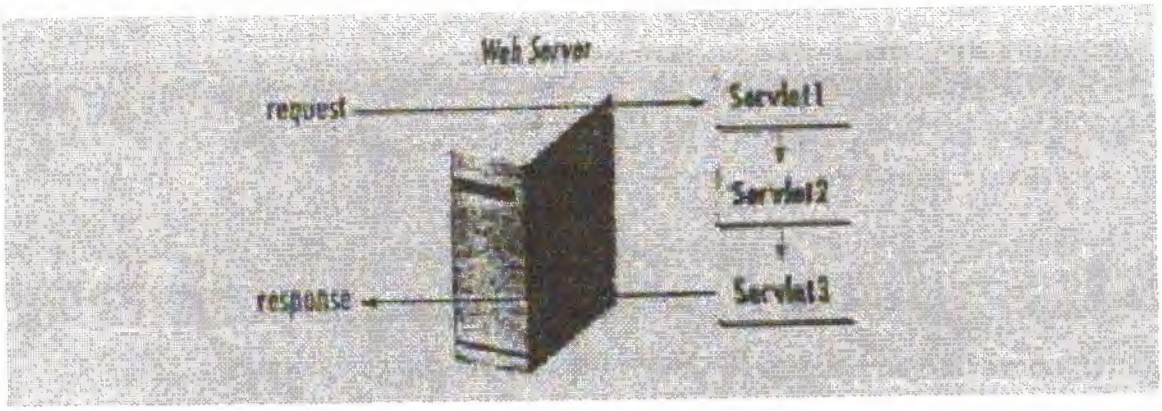

Figure 2.10. Servlet chaining

When a servlet converts one type of content into another, the technique is called *filtering.*

Servlet chaining can change the way of thinking about the web content creation. Here are some of the things have to do:

Quickly change the appearance of a page, a group of pages, or a type of content.

For example, improve the site by suppressing all <BLINK> tags from the pages of the server; speak to who don't understand English by dynamically translating the text from the pages to the language read by the client. Also suppress certain words that don't want everyone to read, be they the seven dirty words or words not everyone knows already, like the unreleased name of the secrete project. Also enhance certain words on the site, so an online new magazine could have a servlet detect the name of any fortune 1000 companies and automatically make each company name a link to its home page.

Take a kernel of content and display it in special formats.

For example, embed custom tags in the page and have a servlet replace them with HTML content. Imagine an <SQL> tag whose query contents are executed against a database and whose result are placed in an HTML table. This is, in fact similar to the java web server supports the <servlet> tag.

Support esoteric data types.

For example, serve unsupported image types with a filter that converts nonstandard image types to GIF or JPEG.

why would be used a servlet chain when instead could be written a script that edits the files in place- especially when there is an additional amount of overhead for each servlet invoked in handling a request?

The answer is that servlet chains have a threefold advantage:

- They can easily be undone, so when users riot against the tyranny of removing their <BLINK> freedom, quickly reverse the change and appease the masses.
- They handle dynamically created content, so that restrictions are maintained, the special tags are replaced, and dynamically converted Postscript images are properly displayed, even in the output of a servlet (or a CGI script).
- They handle the content of the future, so that don't have to run the script every time new content added.

2.8 Summary

This chapter presented Http Servlet Basics. And first hello word servlet program, and how we can compile the source code and running the servlet, and the different ways can be servlets used to handle a variety of web development tasks.

Chapter Three
The Servlet Life Cycle

3.1 Overview

The servlet life cycle is one of the most exciting features of servlets. This life cycle is a powerful hybrid of the life cycle used in CGI programming and lower-level NSAPI and ISAPI programming [1].

3.2 The Servlet Alternative

The servlet life cycle allows servlet engines to address both the performance and resources problems of CGI and the security concerns of low-level server API programming. A servlet engine may execute all its servlets in a single java virtual machine (JVM). Because they are in the same JVM, servlets can efficiency share data with each other, yet they are prevented by the java language from accessing one another's private data. Servlets may also be allowed to persist between requests as object instances, taking up far less memory than full-fledged processes.

Servers have significant leeway in how they chooce to support servlets. The only hard and fast rule is that a servlet engine must conform to the following life cycle contract:

1. Create and initialize the servlet.
2. Handle zero or more service calls from clients.
3. Server removes the servlets
   (Some servers do this step only when they shut down)

Figure 3.1 demonstrate these points.

It's perfectly legal for a servlet to be loaded, create and instantiated in its own JVM, only to be destroyed and garbage collected without handling any client requests or after handling just one request.

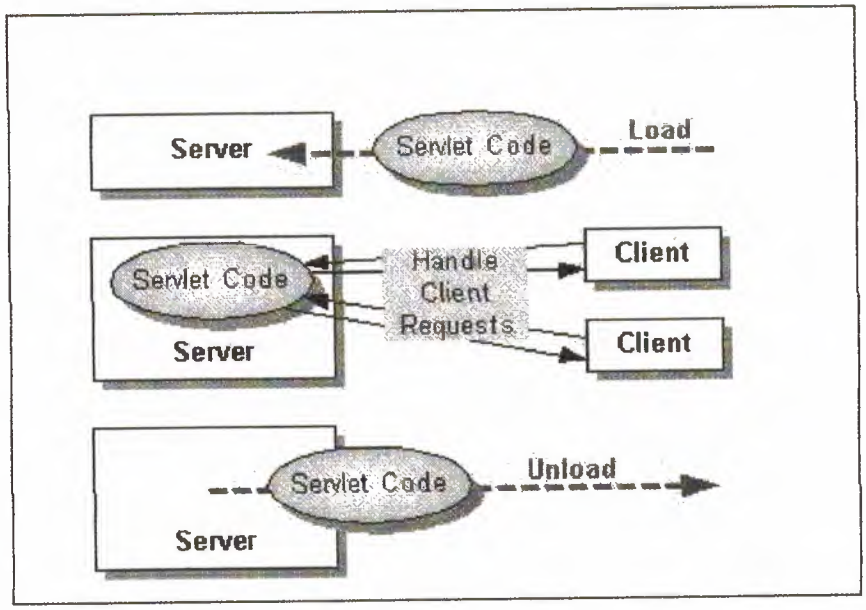

Figure 3.1. The life cycle of servlet.

3.2.1 A Single Java Virtual Machine

Most servlets engines want to execute all servlets in a single JVM. Where that JVM itself executes can differ depending on the server, through. With a server written in java, such as java web server, the server itself can execute inside a JVM right alongside its servlets.

With a single- process, multithreaded web server written in another language, the JVM can often be embedded inside the server process. Having the JVM be part of the server process maximizes performance because a servlet becomes, in a sense, just another low-level server API extension. Such a server can invoke a servlet with a lightweight context switch and can provide information about requests through direct method invocation.

A multiprocess web server (which runs several processes to handle requests) doesn't really have the choice to embed a JVM directly in its process because there is no one process. This kind of server usually runs an external JVM that its process can share. With this approach, each servlet access invokes a heavyweight context switch reminiscent of FastCGI [1]. All the servlets, however, still share the same external process.

Fortunately, from the perspective of the servlet, the server's implementation doesn't really matter because the server always behaves the same way.

3.3 Servlet Reloading

Most servers automatically reload a servlet after its class file (under the default servlet directly, such as **server_root/servlets**) changes. It's an on-the-fly upgrade procedure that greatly speeds up the development-test cycle and allows for long server uptimes.

Servlet reloading may appear to be a simple feature, but it's actually quite a trick and requires quite a hack. **ClassLoader** objects are designed to load a class just once. To get around this limitation and load servlets again and again, servers use custom class loaders that load servlets from the default servlets directory. This explains why the servlet classes are found in **server_root/servlets** even thought that directory doesn't appear in the server's classpath.

When a server dispatches a request to a servlet, it first checks if the servlet's class file has changed on disk. If it has changed, the server abandons the class loader used to load the old version and creates a new instance of the custom class loader to load the new version. Old servlet versions can stay in memory indefinitely (and, in fact, other classes can still hold references to the old servlet instances, causing old side effects), but the old versions are not used to handle any more requests.

Servlet reloading is not performed for classes found in the server's classpath (such as **server_root/classes**) because those classes are loaded by the core, primordial class loader. These classes are loaded once and retained in memory even when their class files change.

In general best to put servlet support classes somewhere in the server's classpath (such as server_root/classes) where they don't get reloaded. The reason is that support classes are not nicely reloaded like servlets. A support class, placed in the default servlets directly and accessed by a servlet, is loaded by the same class loader instance that loaded the servlet. It doesn't get its own class loader instance. Consequently, if the support class is recompiled but the servlet referring to it isn't, nothing happens. The server checks only the timestamp on servlet class files.

A frequently used trick to improve performance is to place servlet in the default servlet directly during development and move them to the server's classpath for deployment.

Having them out of the default directly eliminates the needless timestamp comparison for each request.

3.4 Init and Destroy

Just like applets, servlets can define init () and destroy () methods. A servlet's init (ServletConfig) method is called by the server immediately after the server constructor the server's instance. Depending on the server and its configuration, this can be at any of these times:

- When the server starts.
- When the servlet first requested, just before the service () method is invoked.
- At any request of server administrator.

In any case, init () is guaranteed to be called before the servlet handles its first request.

The init () method is typically used to perform servlet initialization- creating or loading objects that are used by the servlet in the handling of its requests. Why not use a constructor instead? Well, in JSDK1.0 (for which servlets were originally written [1]), constructors for dynamically loaded java classes (such as servlets) couldn't accept arguments. So, in order to provide a new servlet any information about itself and its environment, a server had to calla servlet's init () method and pass along an object that implements the ServletConfig interface. Also, java doesn't allow interfaces to declare constructors. This means that the **javax.servlet.Servlet** interface cannot declare a constructor that accepts a ServletConfig parameter. It has to declare another method, like init (). It's still possible, of course, to define constructors in servlets, but in the constructor has not been to access the ServletConfig object or the ability to throw a ServletException.

The ServletConfig object supplies a servlet with information about its initialization (init) parameters. These parameters are given to the servlet itself and are not associated with any single request. They can specify initial values, such as where counters begin to counting, or default values, perhaps a template to use when not specified by the request. In the java web server, init parameters for a servlet are usually set during the registration process.

Other servers set init parameters in different ways. Sometimes it involves editing a configuration file. One creative technique has been used with java web server, but currently by no other servers, is to treat servlets as JavaBeans [1]. Such servlets can be

loaded from serialized files or have their init properties set automatically by the server at load time using introspection.

The ServletConfig object also holds a reference to a ServletContext object that a servlet may use to investigate its environment.

The server calls a servlet's destroy () method when the servlet is about to be unloaded. In the destroy () method, a servlet should free any resources it has acquired that will not be garbage collected. The destroy () method also gives a servlet a chance to write out its unsaved cached information or any persistent information that should be read during the next call to init ().

3.5 Single -Thread Model

Although it is standard to have one servlet instance per registered servlet name, it is possible for a servlet to elect instead to have a pool of instances created for each of its names, all sharing the duty of handling requests. Such servlets indicate this desire by implementing the **javax.servlet.SingleThreadModel** interface. This is an empty, tag interface that defines no methods or variables and servers only to flag the servlet as waiting the alternate life cycle.

A server that loads a SingleThreadModel servlet must guarantee, according to the Servlet API documentation, "That no two threads will execute concurrently the service method of the servlet." To accomplish this, each thread uses a free servlet instance from the pool, as shown in figure 3.3. Thus any servlet implementing SingleThreadModel can be considered thread safe and isn't required to synchronize access to its instance variables.

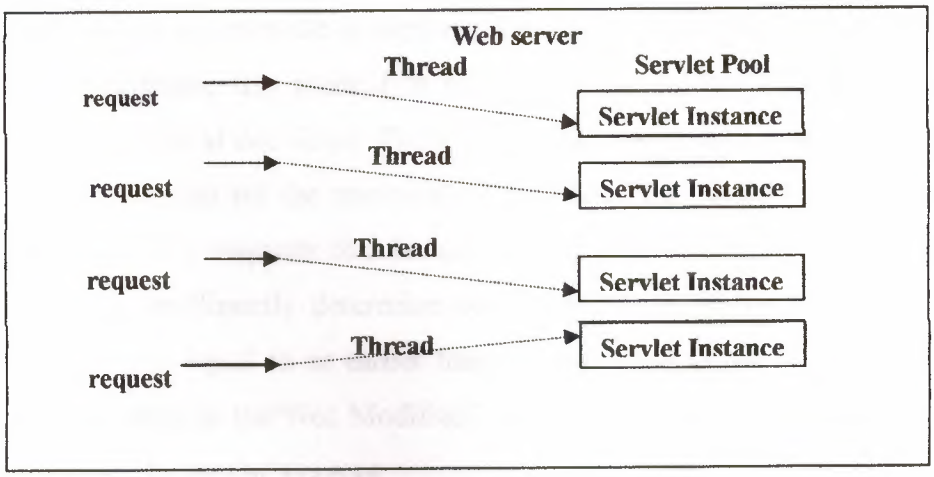

Figure 3.2. The single Thread Model

3.6 Last Modified Times

Most web servers, when they return a document, include as part of their response a last-modified header. An example last-modified header value might be:

Tue, 06-may-98 15:41:02 GMT

This header tells the client the time that page was last changed. That information alone is only marginally interesting, but it proves useful when a browser reloads a page.

Most web browsers, when they reload a page, include in their request an **IF-Modified-Since** header. It structure is identical to the **Last-modified** header:

Tue, 06-may-98 15:41:02 GMT

This header tells the server the last-modified time of the page when it was last downloaded by the browser. The server can read this header and determine if the file has changed since the given time. If the file has changed, the server must send the newer content. If the file hasn't changed, the server can reply with a simple, short response that tells the browser the page has not changed and it is sufficient to redisplay the cached version of the document. For those familiar with the details of HTTP, this response is the 304 "Not Modified" status code.

This technique works great for static pages: the server can use the file system to find out when any file was last modified. For dynamically generated content, through, such as that returned by servlets, the server needs some extra help. By itself, the best the server can do is safe and assume the content changes with every access, effectively eliminating the usefulness of the Last-Modified and If-modified-Since headers.

The extra help servlet can provide is implementing the **getLastModified ()** method. A servlet should implement this method to return the time is last changed its output. Servers call this method at two times. The first time the server calls it is when it returns a response, so that it can set the response's **Last-Modified** header. The second time occurs in handling GET requests that include the **IF-Modified-Since** header (usually reloads), so it can intelligently determine how to response. If the time returned by **getLastModified ()** is equal to or earlier than the time sent in the **If-Modified-Since** header, the server returns the"Not Modified" status code. Otherwise, the server calls doGet () and return the servlet's output.

Some servlets may find it difficult to determine their last modified time. For these situations, it's often best to use the "play it safe" default behavior. Many servlets,

however, should have little or no problem. Consider a "bulletin board" servlet where people post carpool openings or the need for recquetable partners. It can record and return when the bulletin board's contents were last changed. Even if the same servlet manages several bulletin boards, it can return a different modified time depending on the board given in the parameters of request. Here's a **getLastModified()** method.

```
Public long getLastModified(HttpServletRequest  req){
Return LastprimeModified.getTime()/1000*1000; }
```

Notice that this method returns a long value that represent the time as a number of milliseconds since midnight, January 1, 1970, GMT. This is the same representation used internally by java to store time values. Thus, the servlet uses the **getTime ()** method to retrieve LastprimeModified as long.

Before returning this time value, the servlet rounds it down to the nearest second by dividing by 1000 and then multiplying by 1000. All times returned by **getLastModified ()** should be rounded down like this. The reason is that the Last-Modified and If-Modified-Since headers are given to the nearest second. If **getLastMopdified ()** returns the same time but with higher resolution, it may erroneously appear to be a few milliseconds later than the time given by If –modified-Since. For example, let's assume LastprimeModified.getTime ()/1000*1000 is exactly 869127442359 milliseconds, this told the browser, but the only to the nearest second:

Thu, 17-Jul-97 09:17:22 GMT

Now let's assume that the user reloads the page and the browser tells the server, via the If-Modified-Since header, the time it believes its cashed page was last modified:

Thu, 17-Jul-97 09:17:22 GMT

Some servers have been known to receive this time, convert it to exactly 869127442000 milliseconds, find that this time is 359 milliseconds earlier than the time returned by **getLastModified ()**, and falsely assume that the servlet's content has changed. This is why, to play it safe, **getLastModified ()** should always round down to the nearest thousands milliseconds.

The **HttpServletRequest** object is passed to **getLastModified ()** in case the servlet needs to base its results on information specific to the particular request.

The generic bulletin board servlet can make use of this to determine which board was being requested.

3.7 Status Codes

The most common status code numbers are defined as mnemonic constants (public final static in fields) in **HttpServletResponse** class. A few of these are listed in Table 3.1.however, by using status codes, a servlet can do more with its response. For example, it can redirect a request or report a problem.

3.7.1 Setting a Status Code

A servlet can use setStatus () to set a response status code:

public void HttpServletResponse.setStatus(int sc)

public void HttpServletResponse.setStatus(int sc, String sm)

Both of these methods set HTTP status code to the given value. The code can be specified as a number or with one of the SC_XXX codes defined within HttpServletResponse. With the single-argument version of the method, the reason phrase is set to the default message for the given status code. The two-argument version allows to specify an alternate message.

Mnemonic Constant	code	Default Message	Meaning
SC_OK	200	OK	The client's request was successful, and the server's response contains the requested data. This is the default status code.
SC_NO_CONTENT	204	No Content	The request succeeded, but there was no new response body to return. Browsers receiving this code should retain their current document view. This is a useful code for a servlet to use when it accepts data from a form but wants the browser view to stay at the form, as it avoids the "Document contains no data" error message.
SC_MOVED_ PERMANENTLY	30 1	Moved Permanently	The requested resource has permanently moved to anew location. Future references should use the new URL in requests. The new location is given by the Location header. Most browser automatically access the new location
SC_MOVED TEMPORARILY	302	Moved Temporarily	The requested resource has temporarily moved to another location. But future references should still use the original URL to access the resource. The new location is given by the Location

			header. Most browsers automatically access the new location.
SC_UNAUTHORIZED	401	Unauthorized	The request lacked proper authorization. Used in conjunction with the **WWW**-authenticate and Authorization headers.
SC_NOT_FOUND	404	Not Found	The requested resource was not found or is not available.
SC_INTERNAL_ Server_ERROR	500	Internal Serve Error	an unexpected error occurred inside the server that prevented it from fulfilling the request.
SC_NOT_ IMPLEMENTED	501	Not Implemented	The server does not support the functionality needed to fulfill the request.
SC_SEVICE_ UNAVAILABLE	503	Service Unavailable	The service (server) is temporarily unavailable but should be restored in the future. If the server knows when it will be available again, a **Retry-After** header may also be supplied.

Table 3.3. HTTP Status Codes

The setStatus () method should be called before the servlets return any of its response body.

If a servlet sets a status code that indicates an error during the handling of the request, it can call sendError() instead of setStatus ():

public void HttpServletResponse.setError(int sc)
public void HttpServletResponse.setError(int sc, String sm)
a server may give the sendError () method different treatnment than setStatus (). When the two-argument version of the method is used, the status message parameter may be used to set an alternate reason phrase or it may be used directly in the body of the response, depending on the server's implementation.

3.8 HTTP Headers

A servlet can set HTTP headers to provide extra information about its response. Table 3.2 lists the HTTP headers that are most often set by the servlets as a part of a response.

3.8.1 Setting an HTTP Header

The HttpServletResponse class provides a number of methods to assist servlets in setting HTTP response headers. Use setHeader () to set the value of header:

Public void HttpServletResponse.setHeader (String name, String value)

This method sets the value of the named header as a String. The name is case insensitive, as it is for all these methods. If the header had already been set, the new value overwrites the previous one. Headers of all types can be set with this method.

If has be needed to specify a time stamp for a header, uses **setDateHeader() :**

Public void HttpServletResponse.setDateHeader(String name, long date)

This method sets the value of the named header to particular date and time. The method accepts the date value as long that represents the number of milliseconds since the epoch (midnight, January 1, 1970 GMT). If the header has already been set, the new value overwrites the pervious one.

Finally, to specify an integer value for a header has been used **setIntHeader ():**

Public void HttpServletResponse.setInitHeader (String name, int value)

This methods sets the value of the named header as int. if the header had already been set, the new value overwrites the previous one.

The containsHeader () method provides a way to check if a header already exists:

Public void HttpServletResponse.containsHeader(String name)

This method return true if the named header has already been set, false if not.

In addition, the HTML 3.2 specification defines an alternate way to set header values using the <META HTTP-EQUIV> tag inside the HTML page itself:

<**META HTTP-EQUIV**="name" **CONTENT**="value">

This tag must be sent as part of the <**HEAD**> section of the HTML page. This technique does not provide any special benefit to servlets; it was developed for use with static documents, which do not have access to their headers.

Header	Usage
Cache-Control	Specifies any special treatment a caching system should give to This document. The most common values are no-cashe (to indicate this document should not be cached), no-store (to indicate this document should not be cached or even stored by a proxy server, uaually due to its sensitive contents), and max-age=seconds (to indicate how long before the document should be considered stale). This header was introduced in HTTP 1.1.
Pragma	The HTTP 1.0 equivalent of cache-control, with no-caches as its only possible value.
connection	Used to indicate whether the server is willing to maintain an open persistent) connection to the client. If so, its value is set to keep-live. If not, its value is set to close. Most web servers handle this header on behalf of this servlets automatically setting its value to keep-alive when a servlet sets its Content-Length header.
Retry-After	Specifies a time when the server can again handle requests, used with the SC_SERVICE_UNAVAILABLE status code. Its value is either an int that represents the number of seconds or a date string that represents an actual time.
Expires	Specifies a time when the document may change or when its information will become invalid. It implies that it is unlikely the document will change before that time.
Location	Specifies a new location of a document, usually used with the Status codes SC_CREATED, SC_MOVED_PERMANENTLY, and SC_MOVED_TEMPORARITY. Its value must be a fully qualified URL (including "http://").
WWW-Authenticate	Specifies the authorization scheme and the realm of authorization required by the client to access the requested URL. Used with the status code SC_UNAUTHORIZED.
Content-Encoding	Specifies the scheme used to encode the response body. Possible values are gzip (or x-gzip) and compress (or x-compress). Multiple encodings should be represented as a comma-separated List in the order in which the encoding were applied to the data.

Table 3.4. Http Response Headers

3.9 Exceptions

Any exceptions is thrown but not caught by a servlet is caught by its server. How the server handles the exception is server-dependent: it may pass the client the message and the stack trace, or it may not. It may automatically log the exception, or it may even call destroy () on the servlet and reload it, or it may not.

Servlets designed and developed to run with a particular server can optimize for that server's behavior. A servlet designed to interoperate across several servers cannot expect any particular exception handling on the part of the server. If such a servlet requires special exception handling, it may catch its own exceptions and handle them accordingly.

There are some types of exceptions a servlet has no choice but to catch itself. A servlet propagate to its server only those exceptions that subclass **IOException, ServletException,** or **RunTimeException.** The reason has to do with method signatures. The **service ()** method of servlet declares in its throws clause that it throws **IOException** and **ServletException** exceptions. For it (or the **doGet ()** and **doPost()** methods it calls) to throw and not catch anything else causes a compile time error. The **RuntimeException** is special case exception that never needs to be declared in a throws clause. A common example is a **NullPointerException.**

The **init ()** and **destroy ()** methods have their own signatures as well. **The init ()** method declares that it throws only **ServletException** exceptions, and **destroy ()** declares that it throws no exceptions.

ServletException is a subclass of **java.lang.Exception** that is specific to servlets-the class is defined in the **javax.servlet** package. This exception is thrown to indicate a general servlet problem. It has the same constructors as **java.lang.Exception** on that takes no argument and one that takes a single message string. Servers catching this exception may handle it any way they see fit.

The **javax.servlet** package defines one subclass of **ServletException. UnavailableException,** this exception indicates a servlet is unavailable, either temporarily or permanently.

UnavailableException has two constructors:

javax.servlet.UnavailableException(Servlet servlet, String msg)

javax.servlet.UnavailableException(int seconds, Servlet servlet, String msg)

the two- argument constructor creates a new exception that indicates the given servlet is permanently unavailable, with an explanation given by msg. the three-argument version

creates a new exception that indicates the given servlet is temporarily unavailable is given by seconds. This time is only an estimate. If no estimate can be made, a nonpositive value may be used. Notice the nonstandard placement of the optional second's parameter as the first parameter instead of the last. This may be changed in an upcoming release. UnavailableException provides the **isPermanent (), getServlet (),** and **getUnavailableSeconds ()** methods to retrieve information about an exception.

3.9.1 Logging

Servlets have the ability to write their errors to a log file using the log () method:

Public void ServletContext.log(String msg)

Public void Servlet|Context.log(Exception e,String msg)

The single-argument method writes the given message to a servlet log, which is usually an event log file. The two-argument version writes the given message and exception's stack trace to a servlet log. Notice the nonstandard placement of the optional Exception parameter as the first parameter instead of the last for this method. For both methods, the output format and location of the log are server specific

The **GenericServlet.log**(String msg)

This is another version of the **ServletContext** method to **GenericServlet** for convenient. This method allows a servlet to call simply:

Log (msg)

To write to the servlet log. Note, however, the **GenericServlet** does not provide the two-argument version of log (). The absent of this method probably an oversight, to be rectified in a future release. For now, a servlet can perform the equivalent by calling:

getServletContext ().log (e,msg)

The log () method aids debugging by providing a way to track a servlet's actions. It also offers a way to save a complete description of any errors encountered by the servlet. The description can be the same as the one given to the client, or it can be more exhaustive and detailed.

3.10 Summary

This chapter presented the life cycle of servlet, and the init and destroy methods, and Status code, HTTP Headers and finally the exceptions in the servlet when bug happened how the servlet can be caught by using some ways of exceptions and logging.

Chapter Four
Retrieving Information

4.1 Overview

To build a successful web application, need to know a lot about the environment in which it is running, and also need to find out about the server that is executing the servlets or specific of the client that is sending requests. All these things will be presented in this chapter.

4.2 Initialization Parameters

Each registered servlet name can have specific initialization (init) parameters associated with it. Init parameters are available to the servlet at any time; they are often used in init () to set initial or default values for a servlet or to customize the servlet's behavior in some way.

4.2.1 Getting an Init Parameter

A servlet uses the **getInitParameter ()** method to get access to its init parameters:

Public String ServletConfig.getInitParameter(String name)

This method returns the value of the named init parameter or null if it does not exit. The return value is always a single String. It is up to the servlet to interpret the value.
The GenericServlet class implements the ServletConfig interface and thus provides direct access to the **getInitParameter ()** method. The method usually called like this:

```
Public void init (ServletConfig config) throws ServletException {
Supert.init(config);
String greeting=getInitParameter ("greeting");
}
```

A servlet that needs to establish a connection to a database can use its init parameter to define the details of the connection.

4.3 Getting Init Parameter Names

A servlet can examine all its init parameters using getInitParameterNames ():

Public Enumeration ServletConfig.getInitParameterNames ()

This method returns the names of all the servlet's init parameter as an Enumeration of String objects or an empty Enumeration if no parameters exit. It's most often used for debugging.

The GenericServlet class also makes this directly available to servlets.

4.4 The Server

A servlet can find out much about the server in which it is executing. It can learn the hostname, listening port, and server software, among other things. A servlet can display this information to a client; use it to customize its behavior based on a particular server package.

4.4.1 Getting Information About the Server

There are four methods that can use ti learn about its server: two that are called using the ServletRequest object passed to the servlet and two that are called from the ServletContext object in which the servlet is executing. A servlet can get the name of the server and port number for a particular request with **getServerName ()** and **getSerberPort ()**, respectively:

Public String ServletRequest.getServerName()

Public int ServletRequest.getServerPort()

These methods are attributing of ServletRequest because the values can change for different requests if the server has more than one name (a technique called virtual hosting). The return name might be something like"www.servlets.com" while the returned port might be something like "8080".

The **getServerInfo ()** and **getAttribute ()** methods of ServletContext provide information about the server software and its attribute:

Public String ServletContext.getServerInfo ()

Public Object ServletContext.getAttribute ()

getServerInfo () returns the name and version of the server software, separated by a slash. The string returned might be something like "JavaWebServer/1.1.1".

getAttribute () returns the value of the named server attribute as Object or null if the attribute does not exit. The attributes are server-dependent.

4.5 The Client

For each request, a servlet has the ability to find out about the client machine and, for pages requiring authentication, about the actual user. This information can be used for logging access data, associating information with individual users, or restricting access to certain clients.

4.5.1 Getting Information About the client Machine

A servlet can use **getRemoteAddr()** and **getRemoteHost ()** to retrieve the IP address and hostname of the client machine, respectively:

Public String ServletRequest.getRemoteaddr()

Public String ServletRequest.getRemoteHost()

Both values are returned as String objects, the information comes from the socket that connects the server to the client, so the remote address and hostname may be that of a proxy server. An example of remote address might be "192.26.80.118" while an example of remote host might be "dist.engr.sgi.com".

The IP address or remote hostname can be converted to **java.net.InetAddress** object using **InetAddress.getByName ():**

InetAddress remoteInetAddress =InetAddress.getByName (request.getRemoteAddr)

4.5.2 Getting Information About the User

What has to do when needing to restrict the access to someone of the web pages but wanting to have a bit more control over the restriction than this" continent by continent" approach? For example, an online magazine and need only paid subscribes to read the articles. Well, it does not need to servlets to do this.

Nearly every HTTP server has a built-in capability to restrict access to some or all of its pages to a given set of registered users. How the restricted access depends on the server, but here's how it works mechanically. The first time a browser attempts to access one of these pages, the HTTP server replies that it needs special user authentication. When the browser receives this response, it usually pops open a window asking the user for a name and password appropriate for the page, as shown in figure 4.1.

Figure 4.1. Please log in.

Once the user enters his information, the browser again attempts to access the page, this time attaching the user's name and password along with the request. If the server accepts the name/password pair, it happily handles the request. If, on the other hand, the server doesn't accept the name/password pair, the browser is again denied and the user swears under his breath about forgetting yet another password.

How this does involves servlets? When access to a servlet has been restricted by the server, the servlet can get the name of the user that was accepted by the server, using the **getRemoteUser ()** method:

Public String HttpServletRequest.getRemoteUser ()

This method returns the name of the user making the request as String or null if access to the servlet was not restricted.

A servlet can also use the **getAuthType ()** method to find out what type of authorization was used:

Public String HttpServletRequest.getAuthType ()

This method returns the type of authentication used or null if access to the servlet was not restricted. The most common authorization types are "BASIC" and "DIGEST".
With the remote user's name, a servlet can save information about each client. Over the long term, it can remember each individual's preference. It can remember the series of pages viewed by the client and use them to the add a sense of state to a stateless HTTP protocol.

4.6 The Request

In the previous sections have been shown how the servlet finds out about the server and about the client. Now, how the servlet can be finding what the client wants?

4.6.1 Request Parameters

Each access to a servlet can have any number of request parameters associated with it. These parameters are typically name/value pairs that tell the servlet any extra information it needs to handle the request.
An HTTP servlet gets its request parameters as part of its query string (for GET requests) or as encoded post data (for POST requests). Fortunately, even thought a servlet can receive parameters in a number of different ways, every servlet retrieves its parameters the same way, using **getParameter ()** and **getParametersValues ():**

Public String ServletRequest.getParameter(String name)

Public String[] ServletRequest.getParameterValues(String name)

getParameter () returns the value of the named parameters as String or null if the parameter was not specified. **getParameterValues ()** method returns all the values of the named parameter as an array of String objects or null if the parameters was not specified. A single value is returned in an array of length 1.
In addition to getting parameter values, a servlet can access parameter names using getParameterNames ():

Public Enumeration ServletRequest.getParameterNames ()

This method returns all parameter names as an Enumeration of String object or empty Enumeration if the servlet has no parameters. The method is most often used for debugging.

Finally, a servlet can retrieve the raw query of the request with **getQueryString ()**:

Public String ServletRequest.getQueryString ()

This method returns the raw query string (encoded GET parameter information) of the request or null if there was no query string. For example, "servlet/Sqrt? 576" the query string is 576.

4.6.2 Path Information

HTTP request can include something called "extra path information" or virtual path". In general, this extra path information is used to indicate a file on the server that the servlet should use for something. This path information is encoded in URL of an HTTP request. An example URL looks like this:

http://server:port/servlet/ViewFile/index.html

This invokes the ViewFile servlet, passing"index.html" as extra path information. A servlet can access this path information, and it can also translate the "/index.html" string into the real path of the index.html file. What is the real path of "/index.html"? It's the full file system path to the file- what the server would return if the client asked for "/index.html" directly. This probably turns out to be document_root/index.html, but, of course, the server could have special aliasing that changes this.

Besides being specified explicitly in a URL, this extra path information can also be encoded in ACTION parameter of HTML form:

<form method=GET ACTION="/servlet/Dictionary/dict/definitions.txt">

</form>

This form invokes the Dictionary servlet to handle its submissions and passes the Dictionary the extra path information "/dict/definition.txt". The same file client would see if it requested "/dict/definition.txt", probably server_root/public_html/dict/definition.txt.

Why extra path information? Why does HTTP have special support for extra path information? Isn't it enough to pass the servlet a path parameter? The answer is yes. Servlet don't need the special support. But CGI programs do.

A CGI program cannot interact with its server during execution, so it has no way to receive a path parameter. The server has no somehow translated the path before invoking the CGI program. This is why there needs to support for special "extra path information". Servers know to translate this extra path and send the translation to CGI program as an environment variable.

Of course, just because servlets don't need the special handling of "extra path information," it does not mean they should not use it. It provides a simple, convenient way to attach a path along with a request.

4.6.3 Getting Path information

A servlet can use **getPathInfo ()** method to get extra path information:

Public String HttpServletRequest.getPathInfo ()

This method returns the extra path information associated with the request or null if none was given. An example path is "/dict/definitions.txt ". The path information by itself, however, is only marginally useful. A servlet usually needs to know the actual file system location of the file given in the path info, which is where, **getPathTranslated ()** comes in:

Public String HttpServletRequest.getPathTranslated ()

This method returns the extra path information translated to real file system path or null if there is no extra path information. The returned path does not necessarily point to an existing file or directory. An example translated path is "C:\JavaWebServer1.1.1\public_html\dict\definition.txt".

4.6.4 Getting Mime Types

Once a servlet has the path to a file, it often needs to discover the type of the file. Use **getMimeType ()** to do this:

Public String ServletContext.getMimeType (String file)

This method returns the MIME type of the given file or null if it isn't known. Some of MIME returns are "text/plain", "text/html","text/gif ","text. jpeg".

4.6.5 How It Was Requested

A servlet has several ways of finding out details about how it was requested. The **getScheme ()** method returns the scheme used to make this request:

Public String ServletRequest.getScheme ()

as example of the scheme returns are "http","https","ftp",as well as the newer java-specific "jdbc","rmi". There is no direct CGI counterpart (thought CGI implementation have a SERVER_URL variable that includes the scheme).

The **getProtocol ()** method returns the protocol a version number used to make the request:

Public String ServletRequest.getProtocol ()

The protocol and version number are separated by slash. The method returns null if no protocol; could be determined. For HTTP servlets, the protocol is usually "vHTTP1.0" or "vHTTP1.1"

To find out what method was used for a request, a servlet uses **getMethod ()**:

Public String HttpServletRequest.getMethod ()

This method returns the HTTP method used to make the request. for example, "GET","POST","HEAD".

4.7 Session Tracking

May web sites today provide custom web pages and/or functionality on a client-by-client basic? For example, some web sites allow the user to customize their home page to suit what has been needed. An excellent example of this is the Yahoo! Web site. If the user has been visited the site

http://my.yahoo.com/

The user can be customized how the Yahoo! appear in the future when the user revisit the site. The HTTP protocol does not support port persistent that could help a web

server determine that a request is from a particular client. As far as a web server is concerned, every request could be from the same client or every request could be from a different client. Therefore the Session Tracking is:

A mechanism that servlets use to maintain state about a series of requests from the same user (that is, requests originating from the same browser) across some period of time.

4.7.1 Session -Tracking Basics

Every user of a site is associated with a **javax.servlet.http.HttpSession** object that servlets can use to store or retrieve information about that user. A servlet uses its request object's **getSession ()** method to retrieve the current **HttpSession** object:

Public HttpSession HttpServletRequest.getSession(Boolean crearte)

This method returns the current session associated with the user making the request. If the user has no current valid session, this method creates one if create is true or return null if create is false. This method must be called once before any output is written to the response.

To put data to an **HttpSession** object with the **putValue ()** method:

Public void HttpSession.putValue(string name, Object value)

This method binds the specific object value under the specific name. any existing binding with the same name is replaced. To retrieve an object from a session, use **getValue ():**

Public Object HttpSession.getValue(String name)

This method returns the object bound under the specific name or null if there is no binding. To get also the names of all the objects bound to a session with **getValueNames():**

Public String [] HttpSession.getValueNames ()

This method returns an array that contain the names of all objects bound to this session or an empty (zero length) array if there are no bindings. To remove an object from a session with **removeValue ():**

Public void HttpSession.removeValue (String name)

This method removes the object bound to the specific name or does nothing if there is no binding. Each of these methods can throw a **java.lang.IllegalStateException** if the session being accessed is invalid.

4.7.2 The Session Life Cycle

A session expires automatically, after a set time of interactivity (for the java web server the default is 30 minutes), when it is explicitly invalidated by a servlet. When a session expires (or invalidated), the **HttpSession** object and the data values it contains are removed from the system.

There are several methods involved in managing the session life cycle:

Public Boolean HttpSession.isNew ()

This method returns whether the session is new.

Public void HttpSession.invalidate ()

This method causes the session to be immediately invalidated. All objects stored in the session are unbound.

Public long HttpSession.getCreationTime ()

This method returns the time at which the session was created, as long value that represents the number of milliseconds.

Public long HttpSesion.getLastAccessTime ()

This method returns the time at which the client last sent a request associated with this session, as long value that represents the number of milliseconds.

Each of these methods can throw a **java.lang.IllegalStateException** if the session being accessed is invalid.

4.7.3 Putting Session in Context

How does a web server implement session tracking? When a user first access the site, that user is assigned a new HttpSession object and unique session ID. The session ID identifies the user and is used to match the user with the HttpSession object in subsequent requests. A servlet can discover a session's ID with the **getId()** method:

Public String HttpSession.getId()

This method returns the unique String identifier assigned to this session. The ID might be something like: HT04D1QAAAAABQDGPM5QAAA. The method throws an **IllegalStateException** if the session is invalid.

All valid sessions are grouped together in a HttpSessionContext object. Theoretically, a server may have multiple session contexts, although in practice most have just one. A reference to the server's HttpSessionContext is available via any session object's getSessionContext () method:

Public HttpSessionContext.HttpSession.getSessionContext ()

This method returns the context in which the session is bound; it throws an **IllegalStateException** if the session is invalid.

It is possible now to use HttpSessionContext to examine all the currently valid session with the following two methods:

Public Enumeration HttpSessionContext.getIds ()

Public HttpSession HttpSessionContext.getSession (String sessionId)

The **getIds ()** method returns an Enumeration that contains the session IDs for all currently valid sessions in this context or an empty Enumeration if there are no valid sessions. **getSession ()** returns the session associated with the given session ID. The session IDs returned by **getIds ()** should be held as a server secret, because any client with knowledge of another client's session ID can, with a forged URL join the second client's session.

4.8 Cookies

Cookies are a way for a server (or a servlet, as part of a server) to send some information to a client to store, and for the server to later retrieve its data from that client. Servlets send cookies to clients by adding fields to HTTP response headers. Clients automatically return cookies by adding fields to HTTP request headers.

4.8.1 Working with Cookies

Version 2.0 of the servlet API provides the **javax.servlet.http.Cookie** class for working with cookies. The HTTP headers details for the cookies are handled by the servlet API. To create a cookie with Cookie () constructor:

public Cookie (String name, string value)

This creates a new cookie with initial name and value. A servlet can send a ccokie to the client by passing a cookie object to the addCookie () method of HttpServletResponse :

Public void HttpServletResponse.addCookie(Cookie cookie)

This method adds the specified cookie to the response. Additional cookies can be added with subsequent calls to addCookie (). Because cookies are sent using HTTP headers, they should be added to the response before the user send any content. Browsers are only required to accept 20 cookies per site, 300 total per user, and they can limit each cookie's size to 4096 bytes.

The code to set a cookie looks like this:

```
Cookie cookie=new  Cookie ("ID","123");
Response.addCookie(cookie);
```

A servlet retrieves cookies by calling the getCookies() method of HttpServletRequest:

Public Cookie[] HttpServletRequest.getCookie()

This method returns an array of Cookie objects that contains all cookies sent by the browser as part of the request or null if no cookies were sent. The code to fetch cookies looks like thisL:

```
Cookie [] cookies=request.getCookies ();
If (cookies! =null) {
For (int i=0; i<cookies.lenght; i++) {
String name=cookies[i].getName ();
String value=cookies[i].getValue ();
}}
```

The following methods are used to set the attribute of Cookie:

Public void Cookie.setVersion (int v)

Sets the version of Cookie.

Public void Cookie.setDomain (String pattern)

Specifies a domain restriction patteren. A domain pattern speciefies the servers that sould see a cookie. By default, cookies are returned only to the host that saves them.

Public void Cookie.setMaxAge(int expiry)

Specifies the maximum age of the cookie in seconds before it expires. A negative value indicates the default, that the cookie should expire when the browser exits. A zero value tells the browser to delete the cookie immediately.

Public void Cookie.setPath (String uri)

Specifies a path for a cookie, which is the subset of URIs to which a cookie should be sent. By default, cookies are sent to the page that set the cookie and to all the pages in that directly or under that directly. For example, if "/servlet/CookieMonster" sets a cookie, the default path is "/servlet".

Public void Cookie.setSecure (Boolean flag)

Indicates whether the cookie should be sent only over a secure channel, such as SSI. By default, its value is false.

Public void Cookie.setComment (String comment)

Sets the comment field of the cookie. A comment describes the intended purpose of a cookie. Web browsers may choose to display this text to the user. Comments are not supported by version cookies.

Public void Cookie.setValue (String newValue)

Assigns a new value to a cookie. With version cookies, values should not contain the following: white space, breaks and parentheses, equals signs, commas, double quotes, and slashes, question marks at signs, colons, and semicolons. Empty values may not behave the same way on all browsers.

4.9 Summary

This chapter presented the most common methods that can be used to write the servlets, and the most important topic in the web pages session tracking and cookies, and the methods of each one.

Chapter Five
Security

5.1 Overview

This chapter introduces the basics of web security and digital certificate technology in the context of using servlets. It also discusses how to maintain the security of your web server when running servlets from untrusted third-parties.

5.2 What is the Security?

Security is the science of keeping sensitive information in the hands of authorized users. On the web, this boils down to three important issues:

Authentication

Being able to verify the identities of the parties involved

Confidentiality

Ensuring that only the parties involved can understand the communication

Integrity

Being able to verify that the content of the communication is not changed during transmission

A client wants to be sure that it is talking to a legitimate server (authentication), and it also want to be sure that any information it transmits, such as credit card numbers, is not subject to eavesdropping (confidentiality). The server is also concerned with authentication and confidentiality. If a company is selling a service or providing sensitive information to its own employees, it has a vested interest in making sure that nobody but an authorized user can access it. And both sides need integrity to make sure that whatever information they send gets to the other party unaltered.

Authentication, confidentiality, and integrity are all linked by digital certificate technology. Digital certificates allow web servers and clients to use advanced cryptographic techniques to handle identification and encryption in a secure manner. Thanks to Java's built-in support for digital certificates, servlets are an excellent platform for deploying secure web applications that use digital certificate technology. We'll be taking a closer look at them later.

Security is also about making sure that crackers can't gain access to the sensitive data on your web server. Because Java was designed from the ground up as a secure, network-oriented language, it is possible to leverage the built-in security features and make sure that server add-ons from third parties are almost as safe as the ones you write yourself.

5.2.1 HTTP Authentication

As discussed in Chapter 4, Retrieving information, the HTTP protocol provides built-in authentication support—called basic authentication—based on a simple challenge/response, username/password model. With this technique, the web server maintains a database of user-names and passwords and identifies certain resources (files, directories, servlets, etc.) As protected. When a user requests access to a protected resource, the server responds with a request for the client's username and password. At this point, the browser usually pops up a dialog box where the user enters the information, and that input is sent back to the server as part of a second authorized request if the submitted username and password match the information in the server's database, access is granted. The whole authentication process is handled by the server itself.

Basic authentication is very weak. It provides no confidentiality, no integrity, and only the most basic authentication. The problem is that passwords are transmitted over the network, thinly disguised by a well-known and easily reversed Base64 encoding. Anyone monitoring the TCP/IP data stream has full and immediate access to all the information being exchanged, including the username and password. Plus, password is often stored on server in clear text, making them vulnerable to anyone cracking into the server's file system. While it's certainly better than nothing, sites that rely exclusively on basic authentication cannot be considered really secure.

Digest authentication is a variation on the basic authentication scheme. Instead of transmitting a password over the network directly, a digest of the password is used instead. The digest is produced by taking a hash (using the very secure MD5 encryption algorithm) of the username, password, URI, HTTP request method, and a randomly generated "nonce" value provided by the server. Both

sides of the transaction know the password and use it to compute digests. If the digests match, access is granted. Transactions are thus somewhat more secure than they would be otherwise because digests are valid for only single URI request and nonce value. The server, however, must still maintain a database of the original password. And, as of this writing, digest authentication is not supports by very many browsers.

The moral of the story is that HTTP authentication can be useful in low-security environments. For example, a site that charges for access to contents—say, an online newspaper—is more concerned with ease of use and administration than lock-tight security, so HTTP authentication is often sufficient.

5.2.2 Retrieving Authentication Information

A server can retrieve information about the server's authentication using two method introduced in chapter 4: getRemoteUser ( ) and getAuthType ( ). Example 5.1 shows a simple servlet that tells the client its name and what kind of authentication has been performed (basic digest, or some alternative). To see this servlet in action, you should install it in your server and protect it with a basic or digest security scheme. Because web server implementation varies, you will need to check your documentation for the specifics on how to set this up.

Example 5.1. Snooping the authorization information

```
import  java.io .*;
import  javax.sevlet.*;
import  javax.sevlet.http.*;
Public class AuthorizationSnoop  extends HttpServlet {
Public void doGet (HttpSevletRequest  req, HttpSevletResponse  res)
                        Throws ServletException, IOException{
res.setContenType ("text/html");
PrintWriter out =res.getWriter ( );
out.println("<HTML><HEAD><TITEL>Authorization Snoop
</TITEL></HEAD><BODY>");
out.println ("<H1> this is a password protected resource</H>");
out.println ("<PRE>");
```

```
out.println ("User Name: "+ req.getRemoteUser ( ));
out.println ("Authorization Type: "+ req.getAuthType ( ));
out.println ("</PRE>");
out.println ("</BODY></HTML>");
}
```

5.2.3 Custom Authorization

Normally, client authentication is handled by the web server. The server administrator tells the server which resources are to be restricted to which users, and information about those users (such as their passwords) is somehow made available to the server.

This is often good enough, but sometimes the desired security policy cannot be implemented by the server. Maybe the user list needs to be stored in a format that is not readable by the server. Or maybe you want any username to be allowed, as image as it is given with the appropriate "skeleton key" password. To handle these situations, we can use servlets. A servlet can be implemented so that it learns about user from a specially formatted file or a relational database; it can also be written enforce any security policy you like. Such a servlet can even add, remove, or manipulate user entries—something that isn't supported directly in the Servlet API. Except through proprietary server extensions.

Servlet uses status codes and HTTP headers to manage its own security policy the servlet receives encoded authorization credentials in the Authorization reader. If it chooses to deny those credentials, it does so by sending the SC_UNAUORIZED status code and a WWW- Authenticate header that describes the desire credentials. A web server normally handles these details without involving its server. But for a servlet to do its own authorization, it must handle these details itself while the server is told *not* to restrict access to the servlet.

The Authorization header, if sent by the client, contains the client's username and password. With the basic authorization scheme, the authorization header contains the string of "username:password" encoded in Base 64.

For example the username of "webmaster" with the password "try2gueSS" is sent in an Authorization header with the value:

BASIC d2VibWFzdGVyOnaRyuIUzvUw

If a servlet needs to, it can send a WWW-Authenticate header to tell the client the authorization scheme and the realm against which users will be verified. A realm is simply a collection of user accounts and protected resources. For example, to tell the client to use basic authorization for the realm"admin", the WWW-Authenticate header is:

BASIC realm="Admin"

Example 5.2 shows a servlet that performs custom authorization, receiving an Authorization header and sending the SC_UNAUTHORIZED status code and WWW-Authenticate header when necessary. The servlet restricts access to its atop-secret stuff' to those users (and passwords) it recognizes in its user list. For this example, the list is kept in a simple Hashtable and its contents are hard-coded: this would, of course, be replaced with some other mechanism, such as an external relational database, for a production servlet.

To retrieve the Base64—encoded username and password, the servlet needs to use a Base64 decoder. Fortunately, there are several freely available decoders. For this servlet, we have chosen to use the sun.misc.BASEG4Decoder class that accompanies the JDK. Being in the sun. Hierarchy means it's unsupported and subject to change, but it also means it's probably already on your system. You can find the details of Base64 encoding in RFC 1521 at

http://ds.internic.net/,rfc/rftc152l.txt.

Example 5.*2.* Security in a servlet

```
import java.io.*;
import java.util.*;
import javax.sevlet.*;
import javax.sevlet.http.*;
Public class CustomAuth extends HttpServlet {
Hashtable users = new Hashtable ( );
Public void init (ServletConfig  config ) throws SevletException {
super.init (config);
users.put ("Wallace :cheese",          "allowed");
```

```
users.put ("Gromit: sheepnapper",     "allowed");
users.put ("Penguin:evil",            "allowed");
}
public void doGet (HttpSevletRequest  req,  HttpSevletResponse  res)
throws ServletException, IOException{
res.setContentType("text/plain");
PrintWriter out =res.getWriter( );
// Get Authorization header
String auth = res.getHeader ("Authorization");
//Do we allow that user?
if (!allowUser (auth))  {
//Not allowed, so rerport he's unauthorized
res.sendError (res.SC_UNAUTHORIZED );
res.setHeader ("WWW-Authenticate", "BASIC  realm=\"users\"");
//could offer to add him to the allowed user list
}
else {
//Allowed, so show him the sercret stuff
out.println ("Top-secret stuff");
}}
//This method checks the user information sent in the Authorization
//header against the database of users maintained in the users Hashtable.
Protected boolean allow.User (String auth) throws IOException {
if (auth == null) return false; // no auth
if (!auth.toUpperCase ( ) .startsWith ("BASIC"))
return false; // we only do BASIC
// Get encoded user and password, canes after "BASIC"
String userpassEncoded = auth. substring (6);
// Decode it. using any base 64 decoder
sun .misc.BASE64Decoder dec = new sun.misc.BASE64Decoder ( );
String userpassDecoded = new String (dec. decodeBuffer (userpassEncoded));
// Check our user list to see if that user and password are "allowed"
if ("allowed" .equals (users .get (userpassDecoded)))
return true;
```

```
else
return false;
}}
```

Although the web server is told to grant any client access to this servlet the servlet sends top-secret output only to those users it recognizes. With a few modifications, it could allow any user with a trusted skeleton password. Or, like anonymous FTP, it could allow the "anonymous" username with any email address given a password.

Custom authorization can be used for more than restricting access to a single servlet. Were we to add this logic to our ViewFile servlet, we could implement a custom access policy for an entire sect of files. Were we create a special subclass of HttpServlet and add this logic to that, we could easily restrict access to every servlet derived from that subclass. Our point is this: with custom authorization, the security policy limitations of the server do not limit the possible security policy implementations of its servlets.

5.2.4 Form-based Custom Authorization

Servlets can also perform custom authorization without relying on HTTP authorization, by using HTML forms and session tracking instead. It's a bit more effort to give users a well-designed. Descriptive and friendly login page. For example, imagine you're developing an online banking site. Would you rather let the browser present a generic prompt for username and password or provide your customers with a custom login form that politely asks for specific banking credentials, as shown in Figure 5.1

Many banks and other online services have chosen to use form-based custom authorization. Implementing such a system is relatively straightforward with servlets. First, we need the login page. It can be written like any other HTML form. Example 5.3 shows a sample login.html file that generates the form shown in the figure below.

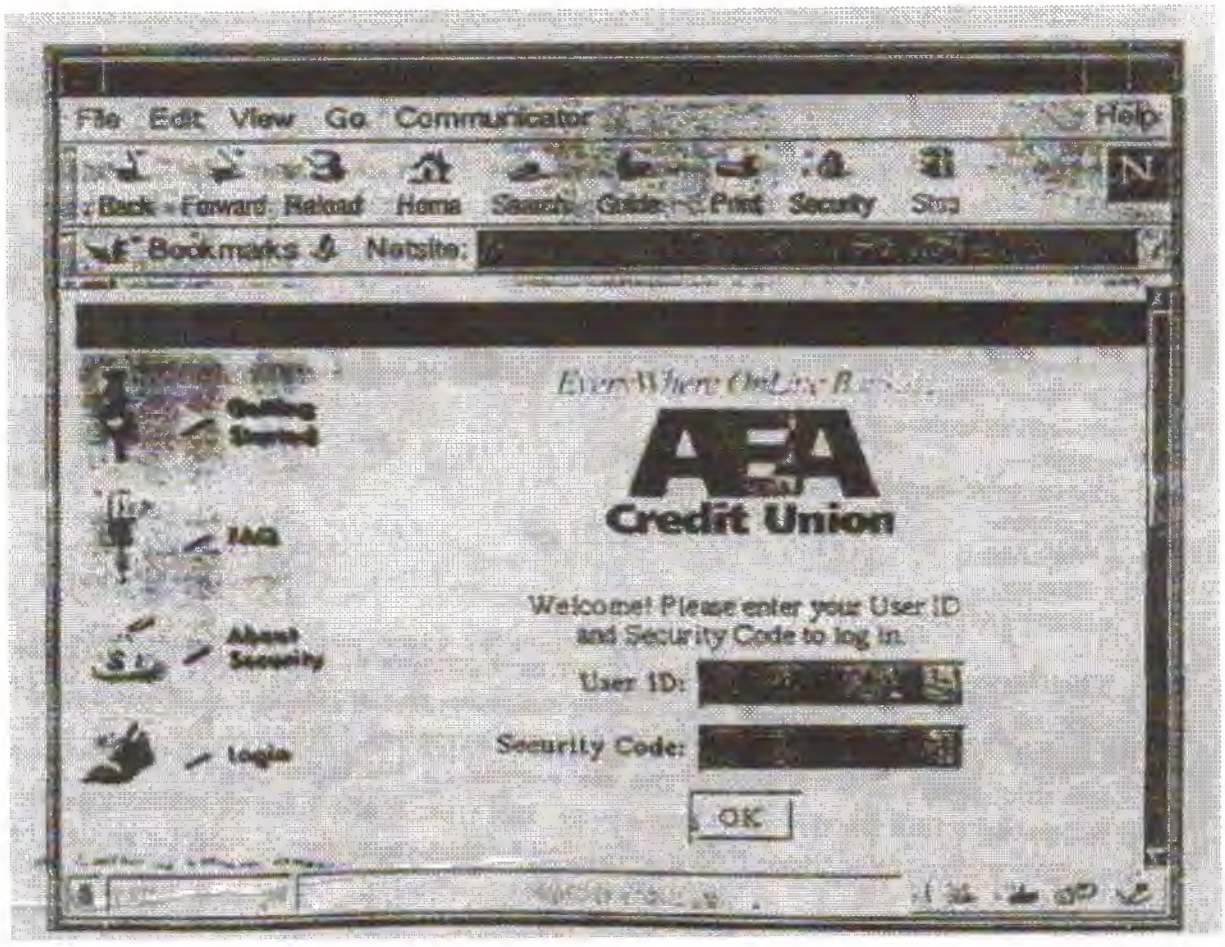

Figure 5.2. An online banking login screen

Example 5.3. The login.html file

```
<html>
<title>Login< /title>
<body>
<form  action= /servlet /loginHandler method=post>
<center>
<table border=0 >
<tr><td colspan=2 >
<p align=center>
Welcome! Please enter your Name<br>
And password to log  in.
</td>< /tr>
<tr><td>
<p align=right><b>Name:</b>
```

```
</td>
<p align=right ><b>Name:</b>
</td>
<td>
<p><input type= text  name="name"  value=" "  size=15>
</td></tr>
<tr><td>
<p align= right> Password: </b>
</td>
<p><input  type=password  name="password"  value=" "  size=15_>
</td</tr>
<tr><td colspan=2>
<center>
<input  type=submit  value=" OK  ">
<center>
</td></tr>
</table>
</body>
</html>
```

This form asks the client for her name and password, and then submits the information to the LoginHandler servlet that validates the login. We'll see the code for LoginHandler soon, but first we should. Ask ourselves, "When is the client going to see this login page?" It's clear she can browse to this login page directly, perhaps following a link on the site's front page. But what if she tries to access a protected resource directly without first logging? In that case, she should be redirected to

This login page and, after a successful login, be redirected back to the original target. The process should work as seamlessly as having the browser pop open a window—except in this case the site pops open an intermediary page.

Example 5.4 shows a servlet that implements this redirection behavior. It outputs its secret data only if the client's session object indicates she has already logged in. If she hasn't logged in, the servlet saves the request URL in her session for later use, and then redirects her to the login page for validation.

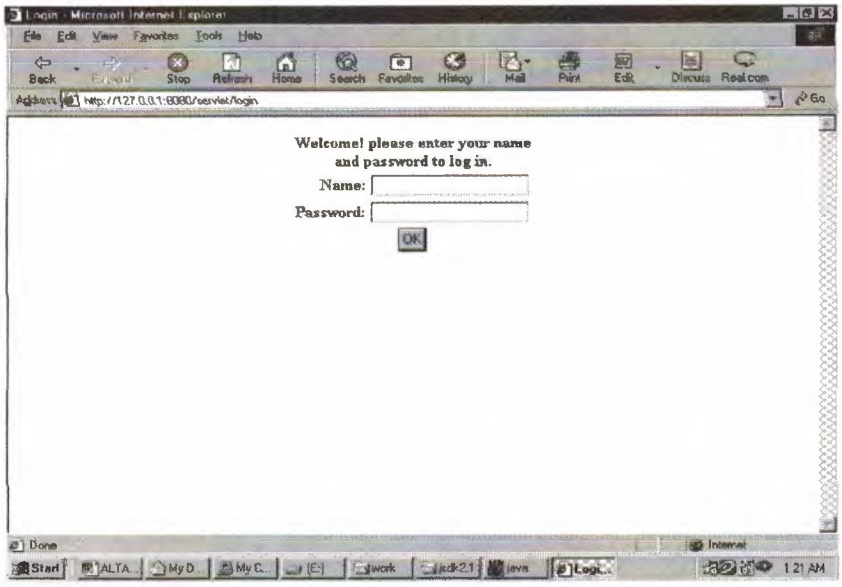

Figure 5.2. A friendly login form

Example 5.4. A protected resource

```
import java.io.*;
import java.util.*;
import java.servlet.* ;
import java.servlet.http.* ;
Public class protectedResource extends  HttpServlet  {
Public void doGet (HttpServletRequest  req, HttpServletResponse res)
                    Throws ServletException , IOException {
res.setContentType ("text/plain");
PrintWriter out = res.getWriter();
//Get the session
HttpSession = req.getSession (true);
//Does the session indicate this user already logged in?
Object done = session.getValue("logon.isDone");   // maker object
if  (done == null) {
//No logon.isDone means he hasn't logged in.
//save the request URL as the true target and redirect to the login page.
Session.putValue("login.target",
HttpUtils.getRequestURL (req).toString ( ));
Res.sendRedirect (req.getScheme ( )+ "://"+
                    req.getServerName()+":"+req.getServerPort( ) +
```

"/login.html");

return;

}

// if we get here, the user has logged in and can see the goods

out.println ("Unpublished O'Reilly book manuscripts await you !");

}}

This servlet sees if the client has already logged in by checking her session for an object with the name "logon.isDone". If such an object exists, the servlet knows that the client has already logged in and therefore allows her to see the secret goods. If it doesn't exist, the client must not have logged in, so the servlet saves the request URL under the name "login.target", and then redirects the client to the login page. Under form-based custom authorization, all protected resources (or the servlets that serve them) have to implement this behavior. Sub classing, or the use of a utility class, can simplify this task. Now for the login handler after the client enters her information on the login form, the data is posted to the LoginHandler servlet shown in Example 5.5. This servlet checks the username and password for validity, if the client fails the check; she is told that access is denied. If the client passes that fact is recorded in her session object and she is immediately redirected to the original target.

Example 5.5. Handling a login

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.* ;

public class LoginHandler extends HttpServlet    {
public void doPost (HttpServletRequest  req, HttpServletResponse  res)
                        throws ServletException,IOException  {
  res.setContetType("text/html");
  PrintWriter  out=res.getWriter ( );
// Get the user's name and password
String name = req.getParameter("name");
String passwd = raq.getParameter("pasawd");
//check the name and password for validity
```

```
if (!allowUser(name, passwd)) {
out.println("<HTML><HEAD><TITLE>Access Denied</TITLE></HEAD>");
out.println ("<BODY>Your login and password are invalid.<BR>");
out.println("You may want to <A HREF=\"/loginl.html\">try again</A>");
out.println( "</BODY></HTML>");
}
else {
//Valid login. Make a note in the session object.
HttpSession session = req.getSession(true);
session.putValue ("login.isDone",name); // just a marker object
/Try redirecting the client to the page he first tried to access
try {
String target = (String) session.getValue("login.target");
if (target !=null)
res . sendRedirect (target);
return;
}catch (Exception. ignored) { }
/Couldn't. redirect to the target. Redirect to the site's home page.
res.sendRedirect(req.getScheme( ) + "://" +
req.getServerName ( )  +":" + req.getServerPort ( ));
}}
protected boolean allowUser(String user, String passwd)  {
return true ;          //trust every one
 }}
```

The actual validity check in this servlet is quite simple: it assumes any username and password are valid. That keeps things simple, so we can concentrate on how the servlet behaves when the login is successful. The servlet saves the user's name (any old object will do) in the client's Session under the name "logon..isDone", as a marker that tells all protected resources this client is okay. It then redirects the client to the original target saved as 'login .target'. Seamlessly sending her where she wanted to go in the first place. If that fails for some reason, the servlet redirects the user to the site s home page.

5.3 Digital Certificates

Real applications require a higher level of security' than basic and digest authentication provide. They' also need guaranteed confidentiality and integrity, as well as more reliable authentication. Digital certificate technology' provides this.

The key concept is public key cryptography. In a public key cryptographic system, each participant has two keys that are used to encrypt or decrypt information. One is the public key, which is distributed freely. The other is a private key, which is kept secret. The keys are related, but one can not be derived from the other. To demonstrate, assume Jason wants to send a secret message to Will. He finds Wills public key and uses it to encrypt the message. When will get the message. He uses his private key to decrypt it. Anyone intercepting the message in transit is confronted with indecipherable gibberish.

Public key encryption schemes have been around for several years and are quite well developed. Most are based on the patented RSA algorithm developed by Ron Rivest, Adi Shamir, and Leonard Adelman. Individual keys come in varying lengths. Usually expressed in terms of the number of bits that make up the key. U.S. government export restrictions currently' limit key size to 40 bits (about a trillion possible keys). Within the United States, however, many systems use 128-bit keys (about 3.40282 x $_{10}$38 possible keys). Because there is no known way to decode an RSA *encrypted message short of brute-force* trial and *error, messages sent using* large keys are very, very secure.

Because keys are so large, it is not practical for a user to type one into her web browser for each request. Instead, keys are stored on disk in the form of digital certificates. Digital certificates can be generated by' software like Phil Zimmerman's *PGP* package, or they can be issued by a third party. The certificate files themselves can be loaded by most security-aware applications, such as servers, browsers, and email software. Public key cryptography solves the confidentiality problem because the communication is encrypted. It also solves the integrity' problem: Will know that the message he received was riot tampered with since it decodes properly'. So far, though, it does not provide any' authentication. Will has no idea whether Jason actually sent the message.

This is where digital signatures come into play. It happens that public and private keys are symmetrical - either key can be used to encode a message, and the alternate key then decodes what the first one encoded.
This means Jason can first use his private key to encode a message and then use Will's public key to encode it again. When Will gets the message, he decodes it first with his private key, and then with Jason's public key. Because only Jason can encode messages with his private key-messages that can be decoded only with his public key-Will knows that the message was truly sent by Jason.

This is different from simpler asymmetric key systems, where one key is only for encoding and another only for decoding. Using symmetric keys for authentication has the significant advantage that it allows secure communication without ever requiring a secure channel-eavesdropping on the exchange of public keys accomplishes nothing. Using symmetric keys, however, has the disadvantage of requiring much more computational muscle. As a compromise. Many encryption systems use symmetric public and private keys to identify each other and then confidentially exchange a separate asymmetric key for encrypting the actual exchange. These asymmetric keys are usually based on DES (Data Encryption Standard).

This leaves one final problem—how does one user know that another user is who she says she is? Jason and Will know each other, so Will trusts that the public. Key Jason gave him in person is the real one. On the other hand, if Lisa wants to give Jason her public key, but Jason and Lisa have never met, there is no reason for Jason to believe that Lisa is not actually Mark. But, if we assume that Will knows Lisa, we can have Will use his private key to sign Lisa's public key. Then, when Jason gets the key, he can detect that Will, whom he trusts, is willing to vouch for Lisa's identity. These introductions are sometimes called a "web of trust."

In the real world, this third-party vouching is usually handled by a specially established certificate authority, such as VeriSign Corporation. Because VeriSign is a well-known organization with a well-known public key, keys verified and signed by VeriSign can be assumed to be trusted, at least to the

extent that VeriSign received proper proof of the receiver's identity. VeriSign offers a number of classes of digital IDs, each with an increasing level of trust. You can get a Class 1 ID by simply filling out a form on the *VeriSign* web site and receiving an email. Higher classes are individually verified by VeriSign employees, using background checks and investigative services to verify identities.

When selecting a certificate authority, it is important to choose a firm with strong market presence. VeriSign certificates, for instance. Are included in Netscape Navigator and Microsoft Internet Explorer, so virtually every user on the Internet will trust and accept them. The following firms provide certificate authority services:

- VeriSigri *(http://www. verisign. Com/)*
- Thawte Consulting *(htcp://www. thawte. Com/)*
- Entrust Technologies *(http://www.entrust.com/)*
- Keywitness *(http://www.keywitness.ca/)*

5.4 Secure Sockets Layer (SSL)

The Secure Sockets Layer protocol, or SSL, sits between the application-level

protocol (in this case HTTP) and the low-level transport protocol (for the internet, almost exclusively TCP /IP). It handles the details of security in management using public key cryptography to encrypt all client/server communication. SSL was introduced by Netscape with Netscape Navigator 1. It has since become the de facto standard for secure online communications and forms the basis of the Transport Layer Security (TLS) protocol currently under development by the internet Engineering Task Force. SSL Version 2.0, the version first to gain widespread acceptance. includes support for server certificates only. It provides authentication of the server, confidentiality, and integrity. Here's how it works:

1. A user connects to a secure site using the HTTPS (HTTP plus SSL) protocol. (You can detect sites using the HTTPS protocol because their

URLs begin with https: instead of http).

2. The server signs its public key with its private key' and sends it back to the browser.

3. The browser uses the server's public key to verify that the same person who signed the key actually owns it.

4. The browser checks to see whether a trusted certificate authority signed the *key. If* one didn't *the browser asks* the user if the key can be trusted and *proceeds* as *directed.*

5. The client generates an asymmetric (DES) key for the session, which *is* encrypted with the server's public key and sent back to the server. This new key is used to encrypt all subsequent transactions. The asymmetric key is used because of the high computational cost of public key cryptosystems.

All this is completely transparent to servlets and servlet developers. You just need to obtain an appropriate server certificate, install it, and configure your server appropriately. Information transferred between servlets and clients is now encrypted. Voila, security!

5.5 SSL Client Authentication

Our security toolbox now includes strong encryption and strong server authentication, but only weak client authentication. Of course~ using SSL 2.0 puts us in better shape because SSL-equipped servers can use the basic authentication methods discussed at the beginning of this chapter without concern for eavesdropping. We still don't have proof of client identity, however—after all, anybody could have guessed or gotten a hold of a client username and password.

SSL 3.0 fixes this problem by providing support for client certificates. These are the same type of certificates that servers use, but they are registered to clients instead. As of this writing, VeriSign claims to have distributed more than

750,000 client certificates. SSL 3.0 with client authentication works the same way as SSL 2.0, except that after the client has authenticated the server, the server requests the client's certificate. The client then sends its signed certificate, and the server performs the same authentication process as the client did, comparing the client certificate to a library of existing certificates (or simply storing the certificate to identify the user on a return visit). As a security precaution, many browsers require the client user to enter a password before they will send the certificate.

Once a client has been authenticated, the server can allow access to protected resources such as servlets or files just as with HTTP authentication. The whole process occurs transparently, without inconveniencing the user. It also provides an extra level of authentication because the server knows the client with a John Smith certificate really is John Smith (and it can know which John Smith it is by reading his unique certificate). The disadvantages of client certificates are that users must obtain and install signed certificates, servers must maintain a database of all accepted public keys, and servers must support SSL 3.0 in the first place. As of this writing, most do, including the Java Web Server.

5.6 Retrieving SSL Authentication Information

As with basic and digest authentication, all of this communication is transparent to servlets. It is sometimes possible, though, for a servlet to retrieve the relevant SSL authentication information. The java. security package has some basic support for manipulating digital certificates and signatures. To retrieve a client's digital information, however, a servlet has to rely on a server-specific implementation of the request's
getAttribute() method.

The first certificate is the user's public key. The second is VeriSign's signature that vouches for the authenticity of the first signature. Of course, the information from these certificate chains isn't particularly useful to the application programmer. In some applications, it is safe to simply assume that a user is authorized if she got past the SSL authentication phase. For others, the

certificates can be picked apart using the javax.security.cert.X5O9certificate class. More commonly, a web server allows you to assign a username to each certificate you tell it to accept Servlets can then call getRemoteUser () to get a unique username. The latter solution works with almost all web servers.

5.7 Running Servlets Securely

CGI programs and C++-based plug-ins operate with relatively unfettered access to the server machine on which they execute (limited on UNIX machines by the user account permissions of the web server process). This isn't so bad for an isolated programmer developing for a single web server, but it's a security nightmare for Internet service providers (ISPs), corporations, schools, and everyone else running shared web servers.

For these sites, the problem isn't just protecting the server from malicious CCI programmers. The more troublesome problem is protecting from *careless* CGI programmers. There are dozens of well-known CCI programming mistakes that could let a malicious client gain unauthorized access to the server machine. One innocuous-looking but poorly written Perl eval function is all it takes.
To better understand the situation, imagine you're an ISP and want to give your customers the ability to generate dynamic content using CCI programs. What can you do to protect yourself? Historically, ISPs have chosen one of three options:

Have blind faith in the customer.

He's a good guy and a smart programmer, and besides, we have his credit card number.

Educate the customer.

If he reads the WWW Security FAQ and passes a written test, we'll let him write CGI programs for our server.

Review all code.

Before we install am CGI program on the server, we'll have our expert review it and scan for security problems.

None of these approaches works very well. Having blind faith is just asking for trouble. Programmer education helps, but programmers are human and bound to

make mistakes. As for code review, there's still no guarantee, plus it takes time and costs money to do the extra work.

Fortunately, with servlets there's another, better solution. Because servlets are written in Java they can be forced to follow the rules of a security manager (or access controller with JDK 1.2) to greatly limit the servers' exposure to risk, all with a minimal amount of human effort.

5.8 The Servlet Sandbox

Servlets built using JDK 1.1 generally operate with a security model called the "servlet sandbox." Under this model, servlets are either trusted or given open access to the server machine, or they're untrusted and have their access limited by a restrictive security manager. The model is very similar to the "applet sandbox," where untrusted applet code has limited access to the client machine.

What's a security manager? It's a class subclasses from java.lang.SecurityManager that is loaded by the Java environment to monitor all security-related operations: opening network connections, reading and 'writing files, exiting the program, and so on. Whenever an application, applet, or servlet performs an action that could cause a potential security' breach, the environment queries the security manager to check its permissions. For a normal Java application, there is no security manager. When a web browser loads an un trusted applet over the network, however, it loads a very restrictive security manager before allowing the applet to execute.

Servlets can use the same technology, if the web server implements it. Local servlets lets can be trusted to run without a security manager, or with a fairly lenient one. For the Java Web Server 1.1, this is what happens when servlets are placed in the default servlet directory or another local source. Servlets loaded from a remote source, on the other hand, are by nature suspect and untrusted, so the Java Web

Server forces them to run in a very restrictive environment where they can't access the local file system, establish network connections, and so on. All this

logic is contained within the server and is invisible to the servlet, except that the servlet may see a SecurityExcepton thrown when it tries to access a restricted resource. The servlet sandbox is a simple model, but it is already more potent than any other server extension technology to date.

Using digital signatures, it is possible for remotely loaded servlets to be trusted just like local servlets. Third-party servlets are often packaged using the Java Archive (JAR) file format. AJAR file collects a group of class files and other resources into a single archive for easy maintenance and fast download. Another nice feature of JAR files that is useful to servlets is that they can be signed using digital certificates. This means that anyone with the public key for "Crazy AL's Servlet Shack" can verify that her copy of Al's Guestbook Servlet actually came from Al. On some servers, including the Java Web Server, these authenticated servlets can then be trusted and given extended access to the system.

5.9 Fine-grained Control

This all-or-nothing approach to servlet permissions is useful, but it can be overly limiting. Consequently, some servlet engines have begun to explore a more fine grained protection of server resources—for example, allowing a specific servlet to establish a network connection but not write to the server's file system. This fine grained control is fairly awkward using the JDK 1.1 notion of a SecurityManager class and, therefore, isn't widely implemented, although it can be done, as the Java Web Server 1.1 proves.

The Java Web Server 1.1 includes eight permissions that can be granted to servlets:

Load servlet

Let the servlet load a named servlet.

Write files

Let the servlet write any file on the local file system.

Listen to socket

Allow the servlet to accept incoming socket (network) connections.

Link libraries

Allow the loading of native libraries, such as the JDBC-ODBC bridge.

Read files

Let the servlet read any file on the local file system.

Open remote socket

Allow the servlet to connect to an external host.

Execute Programs

Permit the servlet to execute external programs on the server. This is useful for servlets that absolutely require access to some system utilities, but it is very dangerous: rm and del qualify as an external program!

Access system properties

Grant access to java.lang.System properties.

A screen shot of the Administration Tool configuration page that assigns these permissions is shown in Figure 5.3.

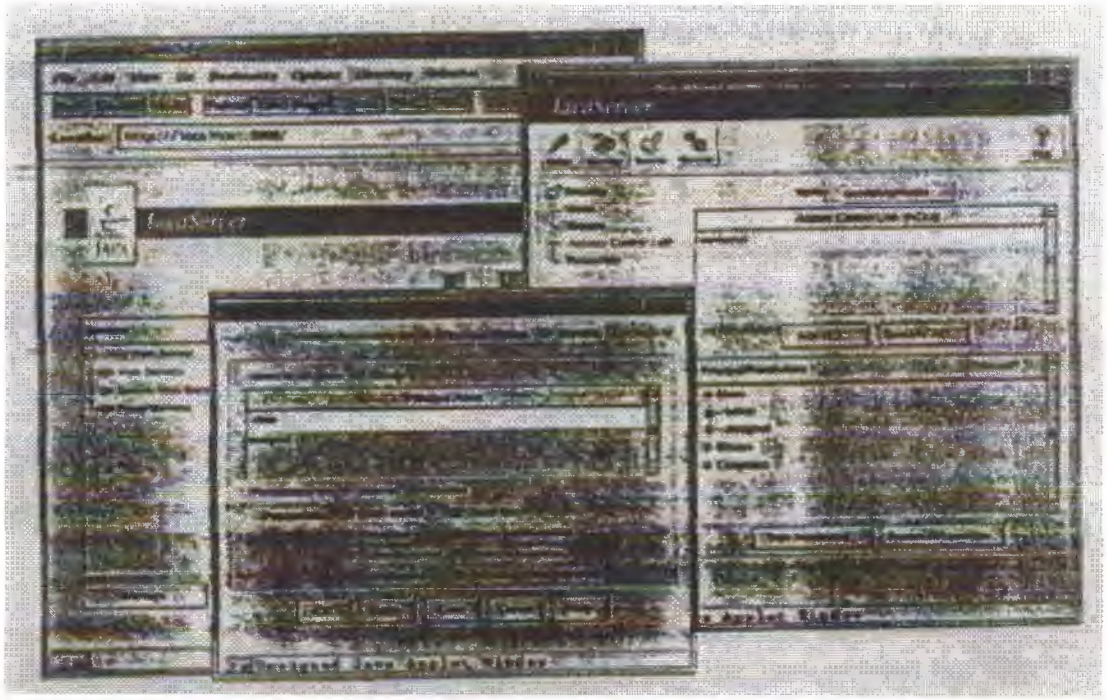

Figure 5.3. Eight Permissions

Theoretically, any criterion can be used to determine what a servlet can or cannot do. It's possible for the security manager to base its permission-granting decision on any factor, including these:

The servlet itself

For example, this servlet can read files and load native libraries but cannot write files.

The client user

For instance, any servlet responding to a request from this client user can

write files.

The client host

For example, any servlet responding to a request from this machine can establish network connections.

Digital signatures

For instance, any servlet in a JAR file signed by this entity has full reign on The server system.

Access Controllers

JSDK1.2 introduces a new extension to the security manager system: the access controller. The new architecture is quite similar to the live particular servlets particular privileges_ approach implemented by the Java Web Server 1.1, except that it applies to all JDK 1.2 programs and therefore makes fine-grained permission implementations much easier.

An access controller allows what might be called super-fine-grained permission control. Instead of granting a servlet the general ability to write files, with an access controller a servlet can be given the right to write to a single file—perfect for a counter servlet, for example. Or it can be given the right to read and write files only in the client user's home directory on the server—appropriate for a client/server application. With access controllers, servlets can be given the rights to do exactly what they need to do and nothing more.

Access controllers work by placing individual pieces of code, often identified by digital signatures, into particular virtual domains. Classes in these domains can be granted fine-grained permissions, such as the ability to read from the server's document root, write to a temporary directory, and accept socket connections. All permission policy decisions are managed by a single instance of the java.securtiy. AccessController class. This class bases its policy decisions on a simple configuration file, easily managed using a graphical user interface.

Now, instead of relying on complicated custom security managers as the Java Web Server team had to do, a servlet engine need only add a few lines of code to use an access controller. So, while the Java Web Server is the only servlet

implementation supporting fine-grained security as of early 1998, once JSDK 1.2 becomes popular, it should be easy for other servlet engine implementers to add the same level of fine-grained access control. These implementations may already be available *by* the time you read this.

5.10 Summary

This chapter presented more important topic especially on the Internet, which is Security, HTTP Authentication, Digital Certificates, Secure Sockets Layer (SSL), and finally Running Servlets Security.

Chapter Six
Database Connectivity

6.1 Overview

This chapter introduces relational databases, the Structures Query Language (SQL) used to manipulate those databases, and the java database connectivity (JDBC) API itself. Servlets, with their enduring life cycle, and JDBC, as well-defined database-independence database connectivity API, are elegant and efficient solutions for webmasters who need to hook their web sites to back-end databases.

6.2 Advantages for Using Servlet Database

The biggest advantage for servlets with regard to database connectivity is that the servlet life cycle allows servlets to maintain open database connections. An existing connection can trim several seconds from a response time, compared to a CGI script that has to reestablish its connection for every invocation. Exactly how to maintain the database connection depends on the task at hand.
Another advantages of servlets over CGI and many other technologies are that JDBC is database-independence. A servlet written to access a Sybase database can, with a two-line modification or a change in a properties file, begin accessing an Oracle database (assuming none of the database calls it makes are vendor-specific).

6.3 Relational Databases

Relational database management system (RDBMS), organizes data into tables. These tables are organized into rows and columns, much like a spreadsheet. Particular rows and columns in table can be related (hence the term "relational") to one or more rows and columns in another table.

One table in a relational database might contain information about customers, another might contain orders, and a third might contain information about individual items within an order. By including unique identifier (say, customer numbers and order

numbers), orders from the orders table can be linked to customer records and individual order components. Figure 6.1 shows how this might look if we drew it out on paper.

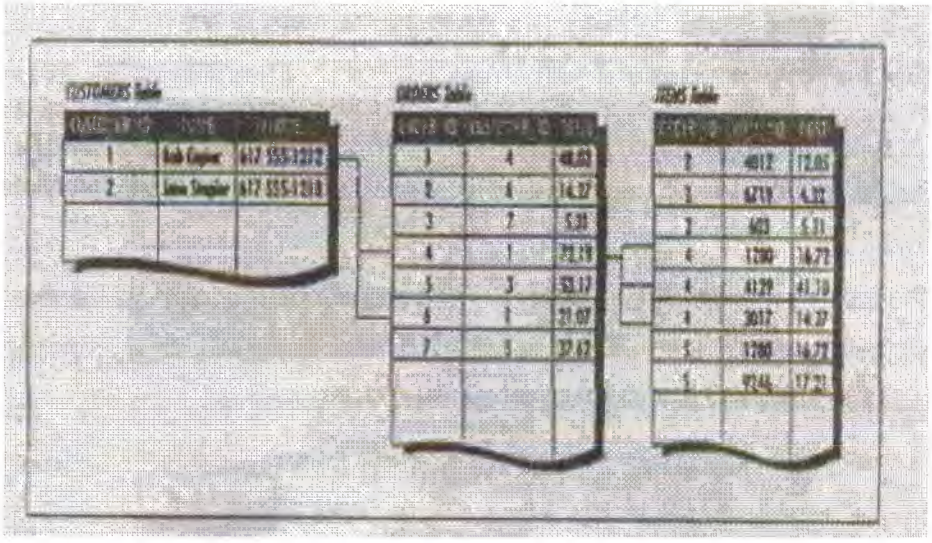

Figure 6.1. Related tables

Data in table can be read, update, appended, and deleted using the structured Query Language, or SQL, sometimes also referred to as the Standard Query Language. Java's JDBC API introduced in JDK1.1 uses a specific subset of SQL known as ANSI SQL-2 Entry Level. Unlike most programming languages, SQL are declarative: you say what you want, and the SQL interpreter gives it to you. Other languages, like C, C++, and java, by contrast, are essentially procedural, in that you specify the steps required to perform a certain task. SQL, while not prohibitively complex, is also rather too broad a subject to cover in great (or, indeed, merely adequate) detail here.

The simplest and most common SQL expression is the SELECT statement, which queries the database and returns a set of rows that matches a set of search criteria.
For example, the following SELECT statement selects everything from the CUSTOMERS table:

```
SELECT * FROM CUSTOMERS
```

SQL keywords like select and from and objects like CUSTOMERS are case insensitive but frequently written in uppercase. When run in Oracle's SQL*PLUS SQL interpreter, this query would produce something like the following output:

CUSTOMER_ID	NAME	PHONE
1	bob copier	617 555-1212
2	Janet stapler	617 555-1213
3	Joel laptop	508 555-7171
4	Larry Coffee	212 555-6225

More advance statements might restrict the query to particular columns or include some specific limiting criteria:

SELECT ORDER_ID, CUSTOMER_ID, TOTAL FROM ORDERS WHERE ORDER_ID=4

The statement selects the ORDER_ID, CUSTOMER_ID, and TOTAL columns from all records where the ORDER_ID field is equal to 4. Here's a possible result:

ORDER_ID	CUSTOMER_ID	TOTAL
4	1	72.19

A SELECT statement can also link two or more tables based on the values of particular fields. This can be either a one-to-one relationship or, more typically, a one-to-many relation, such as one customer to several orders:

SELECT CUSTOMERS. NAME, ORDERS.TOTAL FROM CUSTOMERS, ORDERS WHERE ORDERS.CUSTOMER_ID=CUSTOMERS.CUSTOMER_ID AND ORDERS.ORDER_ID=4

This statement connects (or, in database parlance, joins) the CUSTOMERS table with the ORDERS table via the CUSTOMER_ID field. Note that both tables have this field. The query returns information from both tables: the name of the customer who made order 4 and the total cost of that order. Here's some possible output:

NAME	TOTAL
Bob Copier	72.19

SQL is also used to update the database. For example:

INSERT INTO CUSTOMERS (CUSTOMER_ID, NAME, PHONE) VALUES (5,"Bob Smith","555 123-3456")

UPDATE CUSTOMERS SET NAME="Robert Copier" where CUSTOMER_ID=1

DELETE FROM CUSTOMERS WHERE CUSTOMER_ID =2

The first statement creates a new record in the CUSTOMERS table, filling in the CUSTOMER_ID, NAME AND PHONE

Fields with certain values. The second updates an existing record, changing the value of the NAME field for a specific customer. The last deletes any records with a CUSTOMER_ID of 2. Be very careful with all of these statements, especially DELETE. A DELETE statement without a WHERE clause will remove all the records in the table.

6.3.1 The JDBC API

JDBC is an SQL-level API-one that allows the user to execute SQL statements and retrieve the results, if any. The API itself is a set of interfaces and classes designed to perform actions against any database. Figure 6.2 shows how JDBC programs interactive with database.

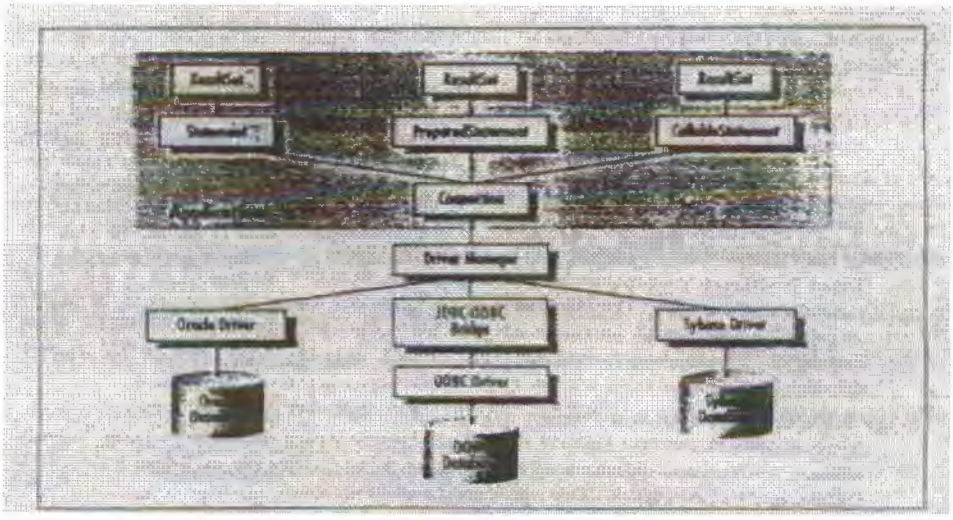

Figure 6.2. Java and the database

6.3.2 JDBC Drivers

The JDBC API, found in the java.sql package, contains only a few concrete classes. Much of the API is distributed as database-neural interfaces classes that specify behavior without providing any implementation. Third-party vendors provide the actual implementations.

An individual database system is accessed via a specific JDBC driver that implements the java.sql.Driver interface. A driver exits for nearly popular RDBMS systems, though few are available for tree. Sun bundles a free JDBC-ODBC bridge driver with the JDK

to allow access to standard ODBC data sources, such as a Microsoft Access database. However, sun advises against using the bridge driver for anything other than development and very limited deployment. Servlets developers in particular should heed this warning because any problem in the JDBC-ODBC bridge driver's native code section can crash the entire server, not just your servlets.

JDBC drivers are available for most database platforms, from a number of vendors and in a number of different flavors. There are four driver categories:

Type 1-jdbc-odbc Bridge Drivers

Type 1 drivers use a bridge technology to connect a java client to an ODBC database service. Sun's JDBC-ODBC bridge is the most common type 1 driver. These drivers are implemented using native code.

Type 2-native –API partly-java Driver

Type 2 drivers wrap a thin layer of java around database-specific native code libraries. For Oracle database, the native code libraries might be based on the OCI (Oracle Call Interface) libraries, which were originally designed for C/C++ programmers. Because type 2 drivers are implemented using native code, in some case they better performance than heir all-java counter-parts. They add an element of risk; however, because a defect in a driver's native code section can crash the entire server.

Type 3-nel –protocol All –Java Driver

Type 3 drivers communicate via a generic network protocol to a piece of custom middleware. The middleware component might use any type of driver to provide the actual database access. Web Logic's Tengah product line is an example. These drivers are all java, which means makes them useful for applet deployment and safe for servlet deployment.

Type 4 –Native-protocol All-java Driver

Type 4 drivers are the most direct of the lot. Written entirely in java, type 4 drivers understand database-specific networking protocols and can access the database directly without any additional software.

6.3.3 Getting a Connection

The first step in using a JDBC driver to get a database connection involves loading the specific driver class into the application's JVM. This makes the driver available later, when we need it for opening the connection. An easy way to load
The driver class is to use the Class.forName() method:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

When the driver is loaded into memory, it registers itself with the java.sql.DriverManager class as an available database driver.
The next step is to ask the DriverManager class to open a connection to a given database, where a specially formatted URL specifies the database. The method used to open the connection is DriverManager.getConnection (). It returns a class that implements the java.sql.Connection interface:

```
Connection con=
        DriverManager.getConnection
("jdbc:odbc:somedb","user","password");
```

A JDBC URL identifier an individual database in a driver-specific manner. Different drivers may need different information in the URL to specify the host database. JDBC URLs usually begin with jdbc:subprotocol:subname. For example, the Oracle JDBC-Thin driver uses a URL of the form of jdbc:oracle:thin:@dbhost:port:sid; the JDBC-ODBC bridge uses jdbc:odbc:datasourcename;odbcoptions.

During the call to getConnection(), the DriverManager object asks each registered driver if it recognizers the URL. If a driver says yes, the driver manager uses that driver to create the Connection object. Here is a snippet of code a servlet might use to load its database driver with the JDBC-ODBC Bridge and create an initial connection:

```
Connection con=null;
Try {
// load (and therefore regsiter) the JDBC-ODBC Bridge
// might throw a ClassNotFoundException

Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
// Get a connection to the database
// might throw an SQLException
con=DriverManager.getConnection("jdbc:odbc:somedb", "user","passwd");
// The rest of the code goes here
}catch(ClassNotFoundException e){
// Handle an error loading the driver
```

```
}catch(SQLException e){
// Handle an error getting the connection
}finally {
//close the connection to release the database resources immediately
try{
 if (con!=null)  con.close();
}catch(SQLException e){}
}
```

6.3.4 Executing SQL Queries

To really use a database, we need to have some way to execute queries. The simplest way to execute a query is to use the java.sql.Statement class. Statement objects are never instantiated directly; instead, a program calls the createStatement() method of Connection to obtain a new Statement object:

```
Statement  stmt=con.createStatement();
```

A query that returns data can be executed using the executeQuery () method of Statement. This method executes the statement and returns a java.sql.ResultSet that encapsulates the retrieved data:

```
ResultSet  rs=stm.executeQuery("SELECT * FROM CUSTOMERS");
```

You can thing of ResultSet object as a representation of the query result returned one row at a time. You use the next () method of ResultSet to move from row to row. The ResultSet interface also boasts a multiple of methods designed for retrieving data from the current row. The getString () and getObject() methods are among the most frequently used for retrieving column values:

```
While (rs.next()){
 String event=rs.getString();
Object count=(Integer)rs.getstring("count");
}
```

The ResultSet is linked to its parent Statement. Therefore, if a Statement is closed or used to execute another query, any related Resultset objects are closed automatically. Table 1.3 shows the java methods that be used to retrieve some common SQL data types from database. No matter what the type, the user can always use the getObject () method of the ResultSet, in which case the type of object returned is shown in the second column. You can also use a specific getXXX () method. These methods are shown in third column, along with the java data types they return.

SQL Data Type Alternative	Java Type Returned by getObject()	Recommended to getObject()
CHAR	String	String getString ()
VARCHAR	String	String getString ()
LONGVARCHAR	String	InputStream getAdciiStream() getUicodeStream()
NUMERIC	java.math.BigDecimal	java.math.BigDecimal getBigDecimal()
DECIMAL	java.math.BigDecimal	java.math.BigDecimal getBigDecimal()
BIT	Boolean	boolean getBoolean()
TINYINT	Integer	byte getByte()
SMALLINT	Integer	short getShort()
INTEGER	Integer	int getInt()
BIGINT	Long	long getLong()
REAL	Float	float getFloat()
FLOAT	Double	double getDouble()
BINARY	byte[]	byte[] getBytes()
VARBINARY	byte[]	byte[] getBytes()
LONGVARBINARY	byte[]	InputStream getBinaryStream ()
DATE	java.sql.Date	java.sql.Date getDate ()
TIME	Java.sql.Time	java.sql.Time getTime()
TimeStamp	Java.sql.Timestamp	java.sql.Timestamp getTimestamp ()

Table 1.3. Methods to retrieve Data from a ResultSet

6.3.5 Handling SQL Exceptions

Any exception statement must be inside try/catch block. This block catches two exceptions:ClassNotFoundException and SQLException. The former is thrown by the Class.forName() method when the JDBC driver class can not be loaded. The latter is thrown by any other exception type, with the additional feature that they can chain. The SQLException class defines an extra method getNextException (), that allows the exception to encapsulate additional Exception objects. Here is how to use it:

```
catch (SQLException e){
out.println(e.getMessage());
while ((e=e.getNextMessage())!=null){
out.println(e.getMessage());
}}
```

This code displays the message from the first exception and then loops though all the remaining exceptions, outputting the error message associated with each one. In practice, the first exception will generally include the most relevant information.

6.3.6 Handling Null Fields

Handling null database values with JDBC can be a little tricky. (A database field can be set to null to include that no value is present, in much the same way that a java object can be set to null.) a method that doesn't return an object, like getInt(), has no way of indicating wether a column is null or whether it contains actual information. (Some drivers returns a string that contains the text "null" when get String() is called on a null column!) any special value like -1, might be a legitimate value. Therefore, JDBC includes the wasNull () method in read was a true database null. This means that that you must read data from the ResultSet into a variable, call wasNull (), and proceed according. It's not pretty, but it works. Here's an example :

```
int age =rs.getInt("age");
if (!wasNull())
out.println("Age:"+age);
```

Another way to check for null values is to use the getObject () method. If a column is null, getObject() always returns null. Compare this to the getString () method that has been shown, in some implementations, to return the empty string if a column is null. Using getObject () eliminates the need to call wasNull () and leads to simpler code.

6.3.7 Using Prepared Statement

A prepared statement object is like a regular Statement object, in that it can be use to execute SQL statements. The important difference is that the SQL in a PreparedStatement is perecompiled by the database for faster execution. Once a PreparedStatemenst has been compiled, it can still be customized by adjusting prepared parameters. Prepared statements are usefull in application that have to run general SQL command over and over.

Use PreparedStatement(String) merthod of Connection to create Preparedstatement objects. Use the? Character as placeholder for values to be submitted later. For example:

```
PreparedStatement pstm=con.preparedStatement("INSERT INTO

(ORDER_ID,CUSTOMER_ID,TOTAL) VALUES(?,?,?)" );

pstm.clearParameters();          // clear any previus parameter values
pstm.setInt(1,2);                // set ORDER_ID
pstm.setInt(2,4);                // set CUSTOMER_ID
pstm.setDouble(3,53.43);         // set TOTAL
pstm.executeUpsdate();           // execute the stored SQL
```

The clearParameters() method removes any previously defined parameter value while the setXXX () methods are used to assign actual values to each of the placeholder question marks. Once you have assigned values for all the parameters. pstm.executeUpdate () to execute the PreparedStatement.

The preparedStatement class has an important application in conjunction with servlet. When loading user-submitted text into the database using Statement object and dynamic SQL, you must be careful not to accidentally introduce any SQL control characters (such as " or ') without escaping them in the manner required by your database. With a database like Oracle that surrounds strings with single quotes, an attempt to insert "John d'Artagan" into the database results in this corrupted SQL:

INSERT INTO MUSKETEERS (NAME) VALUES ('John d'Artagan")

As you can see, the string terminates twice. One solution is to manually replace the single quote ' with two single quotes ' ', the Oracle escape sequence for one single quote. This solution, requires you to escape every character that your database treats as special- not an easy task and not consistent with writing platform-independence code. A far better solution is to use a PreparedStatement and pass the string using its setString ()

method, as shown below. The PreparedStatement automatically escapes the string as necessary for your database:

```
PreparedStatement pstmt=con.preparedStatemnt(
        "INSERT INTO MUSKETEERS (NAME) VALUES(?)");
        pstmt.setString(1,"John d'Artagan");
        pstmt.executeUpdate();
```

6.4 Transactions

Most service –oriented web sites need to do more than run SELECT statements and insert single pieces of data. Let's look at an online banking application. To perform a transfer of $50.000 between accounts, your program needs to perform an operation that consists of two separate but related actions: credit one account and debit another. Now, imagine that for some reason or another, the SQL statement for the credit succeeds but the one for the debit fails. One account holder is $ 50.000 richer, but the other account has not been debited to much.

SQL failure is not only potential problem. If another user checks the account balance in between the credit and the debit, he will see the original balance. The database is shown in an invalid state. Granted, this of thing is unlikely to occur often, but in as a universe of infinite possibilities, it will almost certainly happen sometime. This kind of problem is similar to the synchronization issues. This time, instead of concerning ourselves with the validity of data stored in a servlet, we are concerned with the validity of an underlying database. Simple synchronization is not enough to solve this problem: multiple servlets may be accessing the same database. For system like banking software, chances are good that the database is being used by a number of entirely non-java applications as well.

Sounds like a fairly tricky problem, right? Fortunately, it was a problem long before java came along, so it has already been solved. Most major RDBMS systems support the concept of transactions. A transaction allows you to group multiple SQL statements together. Using a transaction-aware RDBMS, you can begin a transaction or roll back all your SQL statements. If we build our online banking application with a transaction-based system. The credit will automatically be canceled if the debit fails.

A transaction is isolated from the rest of the database until finished. As far as the rest of the database is concerned, everything takes place at once (in other words, transactions are atomic). This means that other users accessing the database will always see a valid

view of the data, although not necessarily an up-to-data view. If a user requests a report on widgets sold before your widget sales transaction is completed, the report will not include the most recent sale.

6.4.1 Optimized Transaction Processing

How do we use transactions without having to connect to the database every time a page is requested? There are several possibilities:

- Synchronization the doPost () method. This means each instance of the servlet deals with only one request at a time. This works well for very low traffic sites, but it does slow things down for your users because every transaction has to finish before the next can start. If the user needs to perform database-intensive updates and inserts, the delay will probably be unacceptable.
- Leave things as they are, but create a new Connection object for each transaction. If the user needs to update data only once in every few thousand page requests, this might be the simplest route.
- Create a pool of Connection objects in the init () method and hand them out as needed, as shown in Figure 6.3. This is probably the most efficient way to handle the problem, if done right. It can, however, become very complicated very quickly without third-party support classes.
- Create a single Connection object in the init () method and have the servlet implement SingleThreadModel, so the web server creates a pool of servlet instances with a Connection for each, as shown in Figure 6.4. This has the same effect as synchronizing doPost (), but because the web server has a number of servlet instances to choose from, the performance hit for the user is not as great. This approach is easy to implement, but it less robust than using a separate connection pool because the servlet has no control over how many servlet instances are created and how many connections are used. When creating single-threaded database servlets, be especially sure to have the destroy () method close any open database connections.
- Implement session tracking in the servlet and use the HttpSession object to hold onto a Connection for each user. This allows the user to go one step

beyond the other solutions and extend transaction across multiple page requests or even multiple servlets.

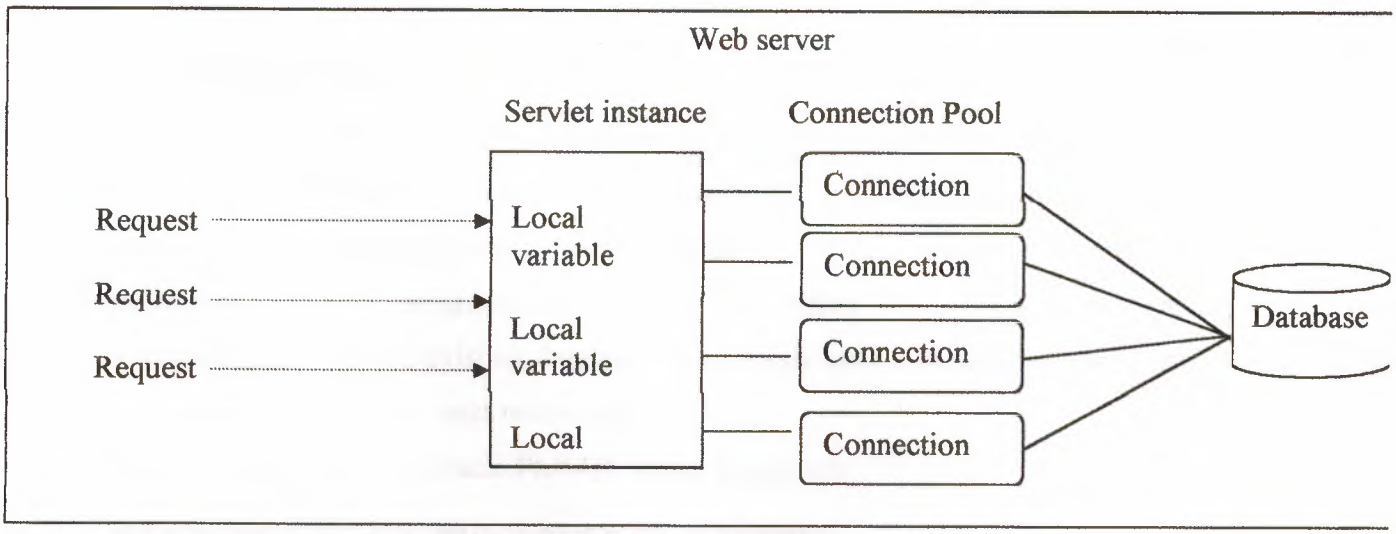

Figure 6.3. Servlets using a database connection pool

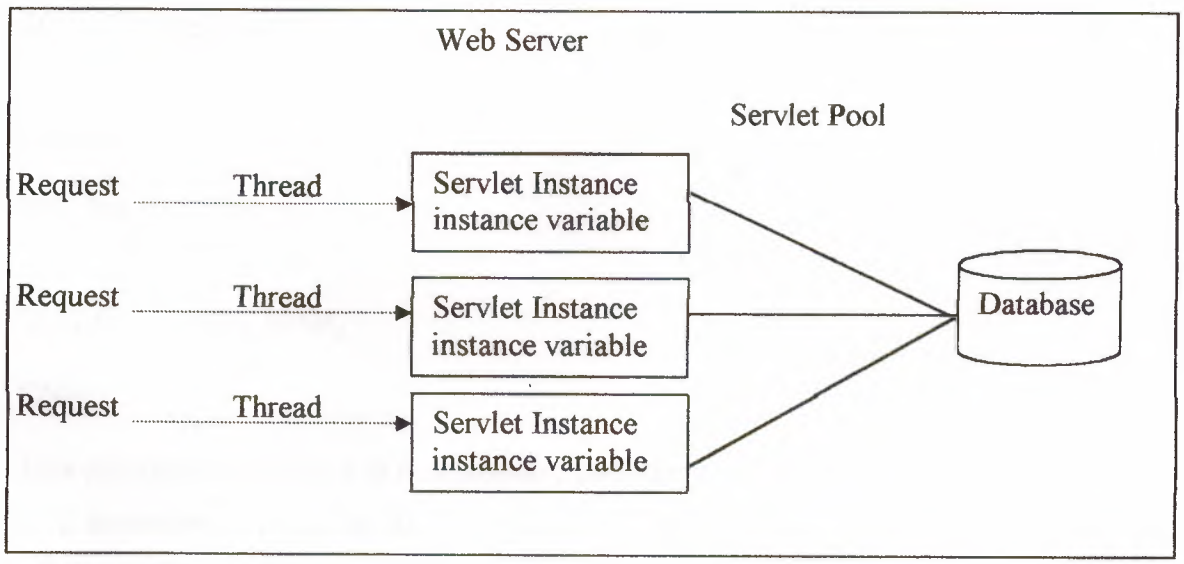

Figure 6.4. Servlets using SingleThreadModel for a server-managed connection pool

6.5 Advanced JDBC Techniques

Now that we've covered the basics, let's talk about a few advanced techniques that use servlets and JDBC. First, we'll examine how servlets can access stored database procedures. Then we'll look at how servlets can fetch complicated data types, such as binary data (images, application, etc.), large quantities of text, or even executable database-manipulation code, from a database.

6.5.1 Stored Procedures

Most RDBMS system includes some sort of internal programming language. One example is Oracle's PL/SQL. These languages allow database developers to embed procedural application code binary within a database and then call that code from other application. RDBMS programming languages are often well suited to performing certain database actions; existing database installations have a number of useful stored procedures already written and ready to go.

The following code is an Oracle PL/SQL stored procedure.

```
CREATE OR REPLACE PROCEDURE SP_INTEREST
(id  IN INTEGER
 bal IN OUT FLOAT ) IS
BEGIN
SELECT BALANCE
INTO bal
FROM accounts
WHERE account_id=id;

Bal:=bal+bal*0.03;
UPDATE accounts
SET balance =bal
WHERE  account_id=id;

END;
```

This procedure executes a SQL statement, performs a calculation, and executes another SQL statement. It would be fairly simple to write the SQL to handle this, so why bother with at all? There are several reasons:

- Stored procedures are precompiled in the RDBMS, so they run faster than dynamic SQL.
- A stored procedure executes entirely within the RDBMS, so they can perform multiple queries and updates without network traffic.

- Stored procedures allow the user to write database manipulation code once and use it across multiple applications in multiple languages.
- Changes in the underlying table structures require changes only in the stored procedures that access them; applications using the database are unaffected.
- Many older databases already have a lot of code written as stored procedures, and it would be nice to be able to leverage the effort.

The Oracle PL/SQL procedure in the previous example takes an input value, in this case an account ID, and returns an update balance. While each database has its own syntax for accessing stored procedures, JDBC creates a standardized eascape sequence for accessing stored procedures using the java.sql.CallableStatement class. The syntax for a procedure that doesn't return a result is "{Call procedure_name(?,?)}". The syntax for stored procedure that returns a result value is "{?=call procedure_name(?,?)}". The parameters inside the parentheses are optional.

Using the CallableStatement class is similar to using the PreparedStatement class:

```
CallableStatement cstm=con.prepareCall("{call sp_interest(?,?)}");
Cstm.registerOutParameter(2,java.sql.Types.FLOAT);
Cstm.setInt(1,accountID);
Cstm.execute();
Out.println("New Balance: "+cstm.getFloat(2));
```

This code first creates a Callstatement using the prepareCall() method of Connection. Because this stored procedure has an output parameter, it uses the registerOutParameter() merthod of CallableStatement to identify that parameter as output parameter of type FLOAT. Finally, the code executes the stored procedure and uses the getFloat() method of CallableStatement to display the new balance. The getXXX () methods in CallableStatement interface are similar to those in the ResultSet interface.

6.5.2 Binaries and Books

Most databases support data types to handle text strings up to several gigabytes in size, as well as binary information like multimedia files. Different databases handle this kind of data in different ways, but the JDBC methods for retrieving it are standard. The getAsciistream () method of ResultSet handles large text strings; getBinaryStream () works for large binary objects. Each of these methods returns an InputStream.

Support for large data types is one of the most common sources of JDBC problems. Make sure to test your drivers thoroughly, using the largest pieces of data your application will encounter. Oracle's JDBC driver is particularly prone to errors in this area.

Here's some code from a message board servlet that demonstrate reading a long ASCII string. Assume that connections, statements, and so on have already been created.

```
Try{
ResultSet  rs=stmt.executeQuery(
"SELECT TITLE,SENDER,,MESSAGE FROM MESSAGES WHERE
MESSAGE_ID=9");
if (rs.next()){
out.println("<h1>"+rs.getString("title")+"</h1>");
out.println("<b>From:</b>"+rs.getString("sender")+"<br>");
BufferReader  msgText=new BufferReader(
new InputStreamReader(rs.getAsciiStream("message")));

while (msgText.ready()){
out.println(msgText.readLine());
}}}
catch(SQLException e){
//Report it
}
```

While it is reading from the Inputstream, this servlet doesn't get the value of any other columns in the result set. This is important because calling any other getXXX () method of ReultSet closes the InputStream.

Binary data can be retrieved in the same manner using the ResultSet.getBinaryStream (). In this case, we need to set the content type as appropriate and write the output as bytes. As shown below servlet code to return a GIF file loaded from a database.

```
Import   java.io.*;
Import   java.sql.*;
Import   javax.servlet.*;
Import   javax.servlet.http.*;

Public class DBGifReader  extends  HttpServlet{
Connection con;
Public void init(ServletConfig config) throws SeervletException{
Super.init(config);
Try{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Con=DriverManager.getConnection("jdbc:odbc:imagedb","user","passw");
}
Catch(ClassNotFoundException e){
Throw new UnavailableException (this,"Couldn't load JdbcOdbcDriver");
```

```
}
Catch(SQLException e){
Throw new UnavailableException (this,"Couldn't get connection");
}}
public void doGet(HttpServletRequest req,HttpServletResponse res) throws
ServletException,IOException{
try{
res.setContentType("image/gif");
ServletOutputStream out=res.getOutputStream();
Statement stmt=con.createStatement();
ResultSet rs=stmt.executeQuery(
"SELECT IMAGE FROM PICTURES WHERE PID="+req.getParameter("PID"));

if (rs.next()){
BufferInputStream gifData=new BufferInputStream(rs.getBinaryStream("image"));
Byte[] buf=new byte[4*1024];
Int len;
While ((len=gifData.read(buf,0,buf.length))!=-1){
Out.write(buf,0,len);
}}
else{
res.sendError(res.SC_NOT_FOUND);
}}
catch(SQLException e){
// Report it
}}}
```

6.6 Summary

This chapter presented database connectivity, relational databases, the JDBC API, reusing Database Objects, Transactions, and finally advances of JDBC techniques.

Chapter 7
Result of the Work

7.1 Overview

This chapter is showing and giving general information about the outputs; so that it is very easy for the customer to enter and compile their needs over the Internet, and also will show all the available books, prices and their general information in our site.

7.2 The Main Page

First, any customers it will enter to the following address:

http://localhost:8080/servlet/welcome.

When the customers write this address he will be able to see the main page as shown in Figure 7.1.

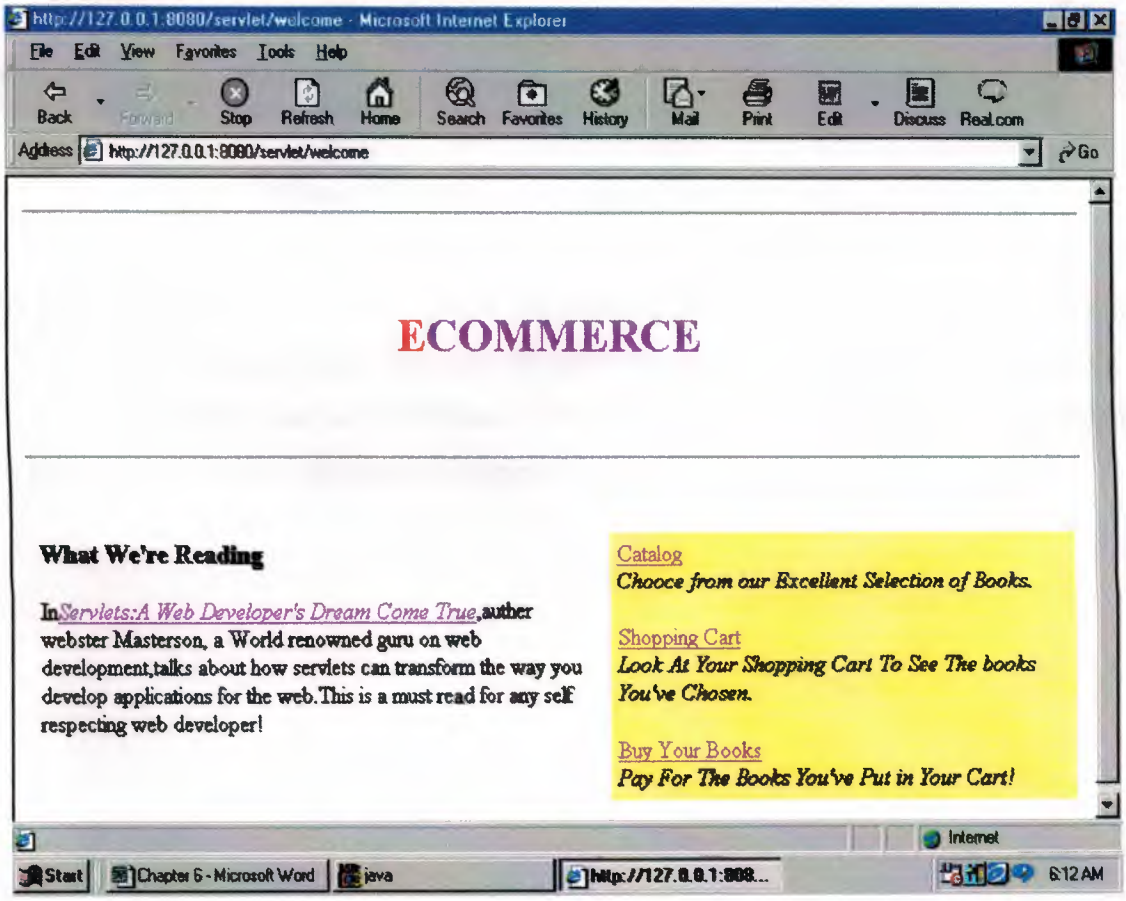

Figure 7.1. The Main Page

In the main page, the customers can be shown, simple paragraph in the left side to encourage his/her to purchase this item. By click on the "servlet: A Web developer's Dream Come True", the customers can see more information about this item as can be shown in Figure 7.2.

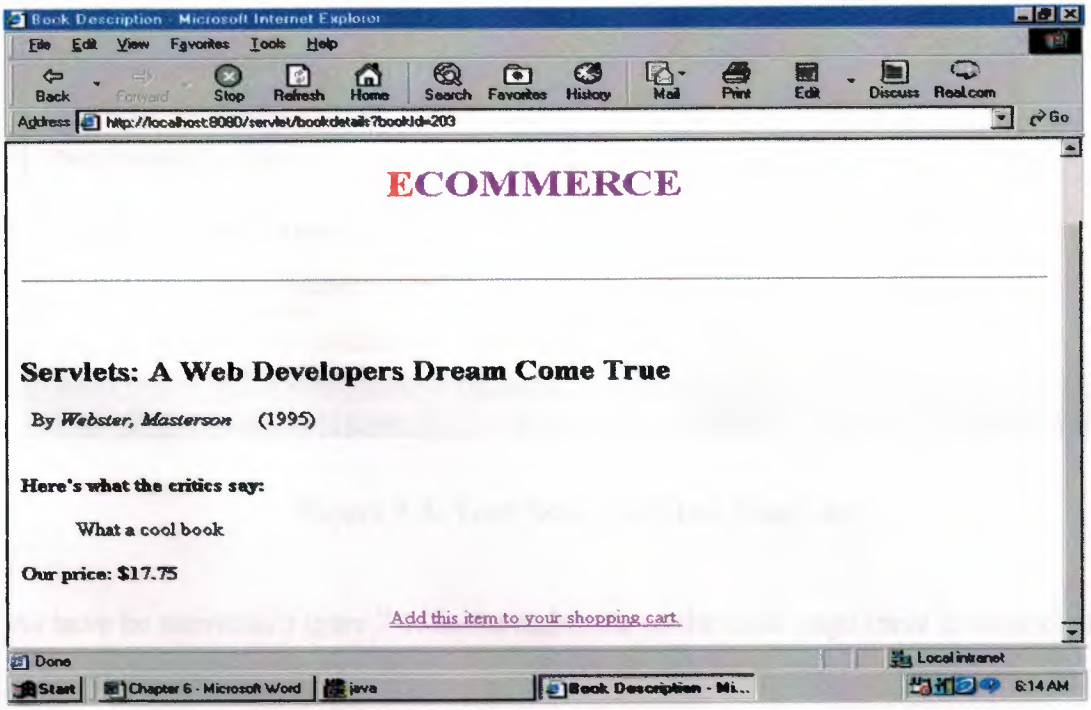

Figure 7.2. Information about Specific Book

As have be shown in Figure 7.2. More information about this item. And also shown in the button of this page"add this item to your shopping cart", this to encourage the customer to add this item to his/her cart. If the customer clicks it, the item will be added to his/her shopping cart as shown in Figure 7.3.

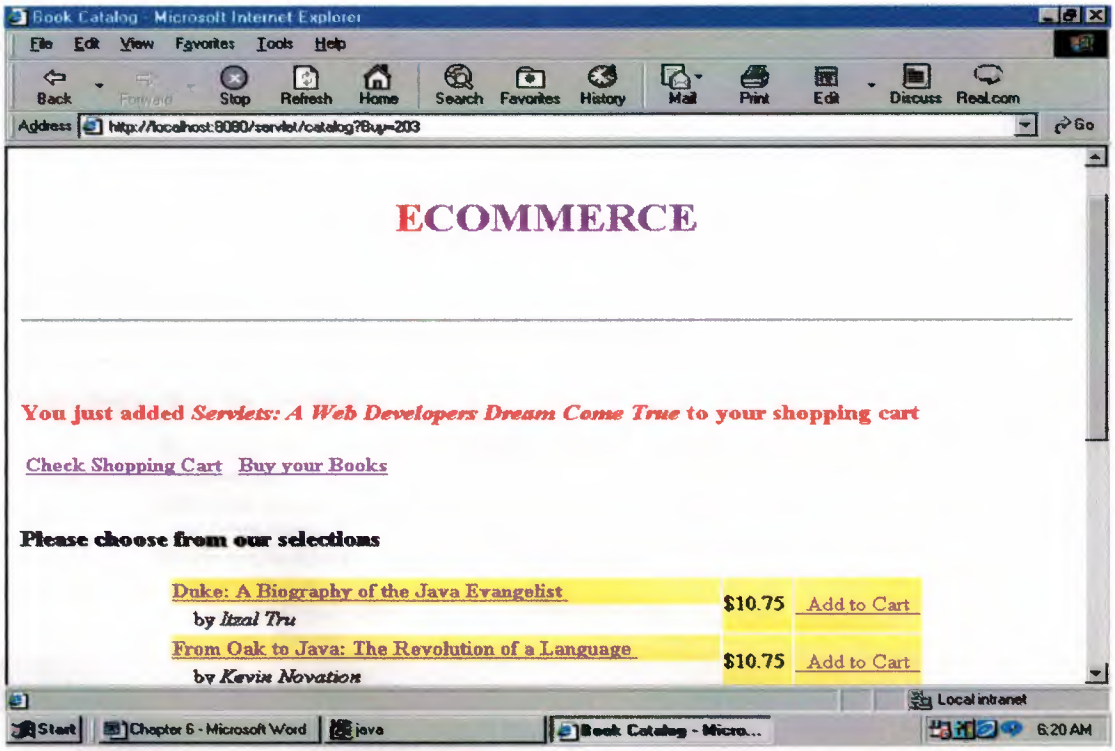

Figure 7.3. Your Book Added to Your Cart

As have be shown in Figure 7.1. In the right side of the main page there is three chooses to the customers:

7.2.1 Catalog

Which is simple HTTP servlet, this class listing to the user all the books available in the database. And encourage the customers to choose and add any items from the excellent selection to his/her shopping cart. As shown in figure 7.4.

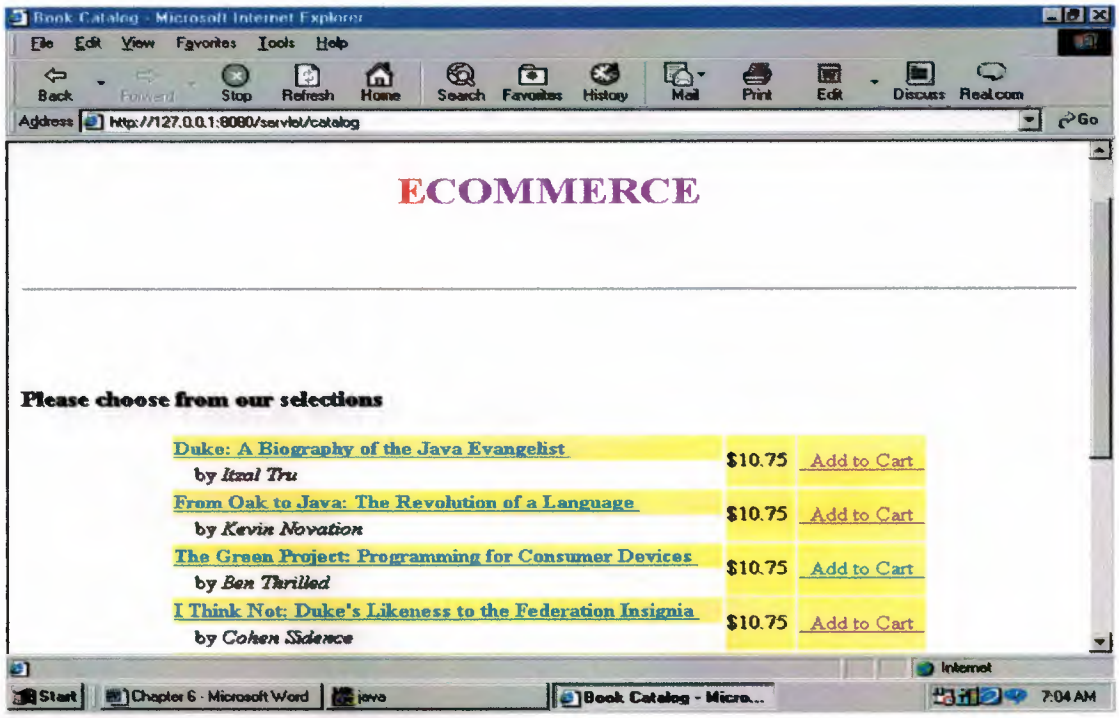

Figure 7.4. List all the Books

7.2.2 Shopping Cart

Which is a simple HTTP servlet, that displays the contents of a customer's shopping cart. It responds to the GET and HEAD methods of the HTTP protocol. for example, If the customers chosen an item from the selection of books lets say he/she chosen "Moving from C++ to the Java (tm) Programming Language" by added this item to to his/her Shopping Cart, if the customer's click on his/her Shopping Cart it will give it to his/her simple message to display to the customer's how many items in his/her Shopping Cart and the Grand Total of this item as shown in figure 7.5.

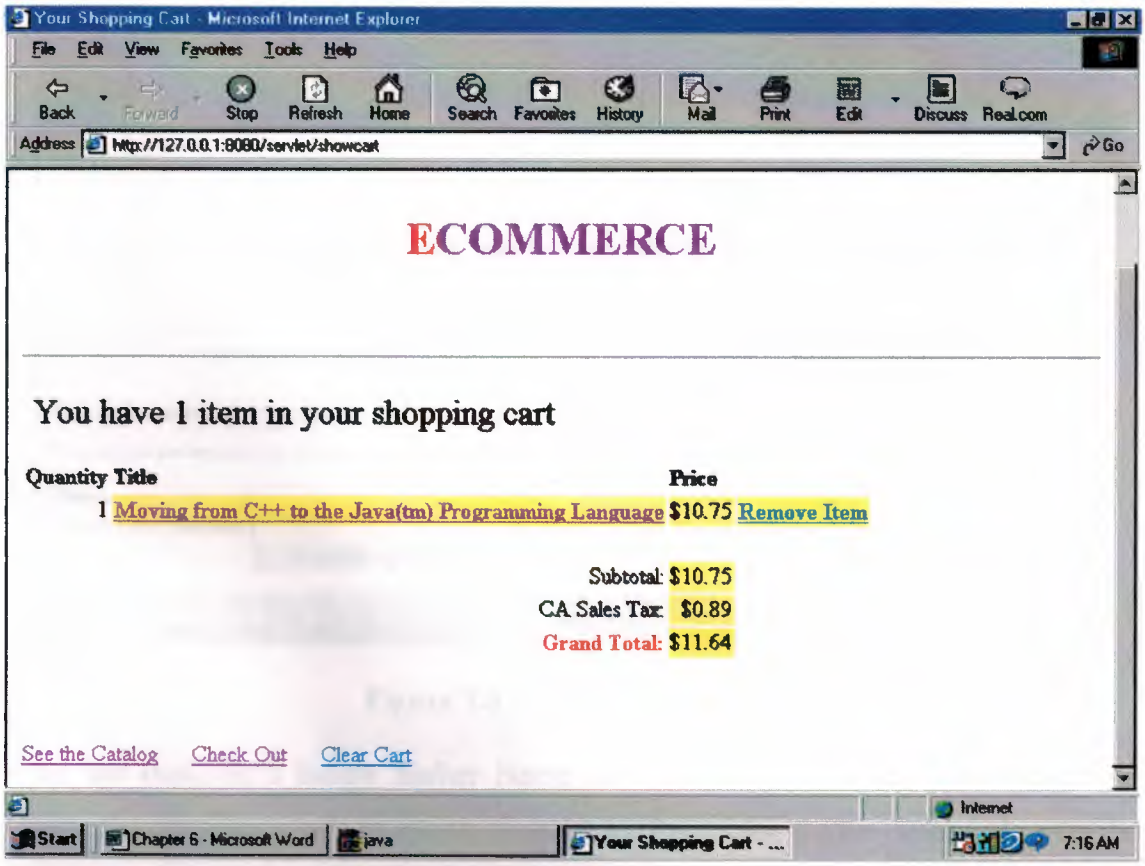

Figure 7.5. How Many Items in your Shopping Cart

and also give it to the customer's to Remove Item if the customer's decide to delete this item, and also give it to the customers three options shown in the bottom of the page which is "See the Catalog" to back to the "CatalogServlet" as shown in the figure 7.4. and "Check Out" to give to the customer the amount of item purchase that has been chosen , and to let the customer's to write his/her name and Credit Card Number shown in figure 7.6.

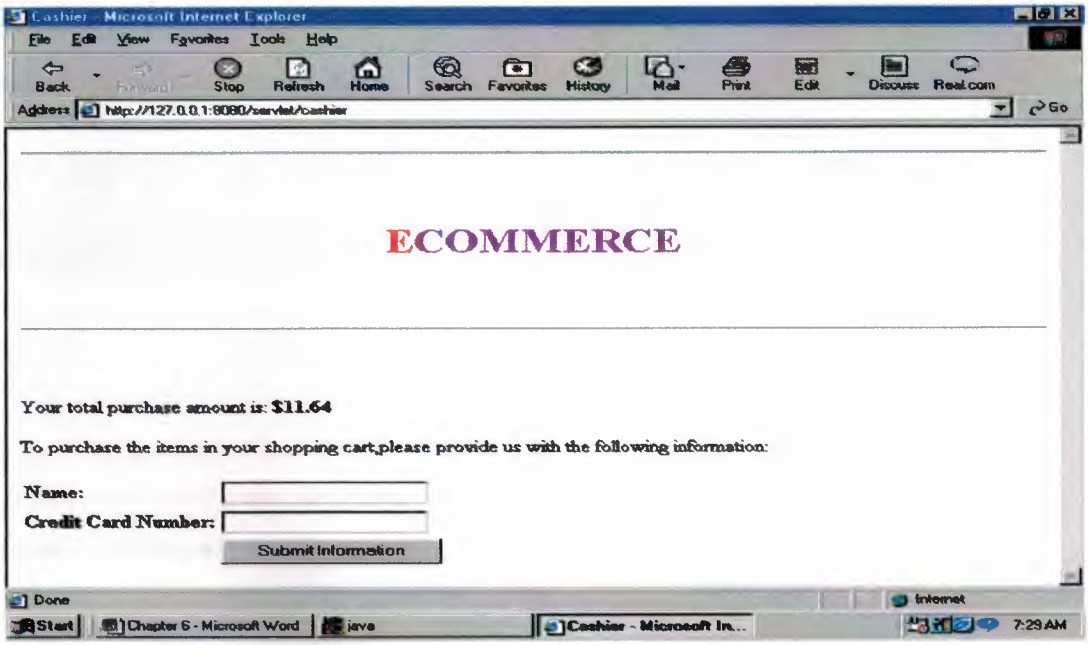

Figure 7.6. Your Total Purchase

After the customer's follow his/her Name and Credit Card Number and submit the Query, the customer's will be shown message to thank him for the order and resets the page to the main page as shown in figure 7.7.

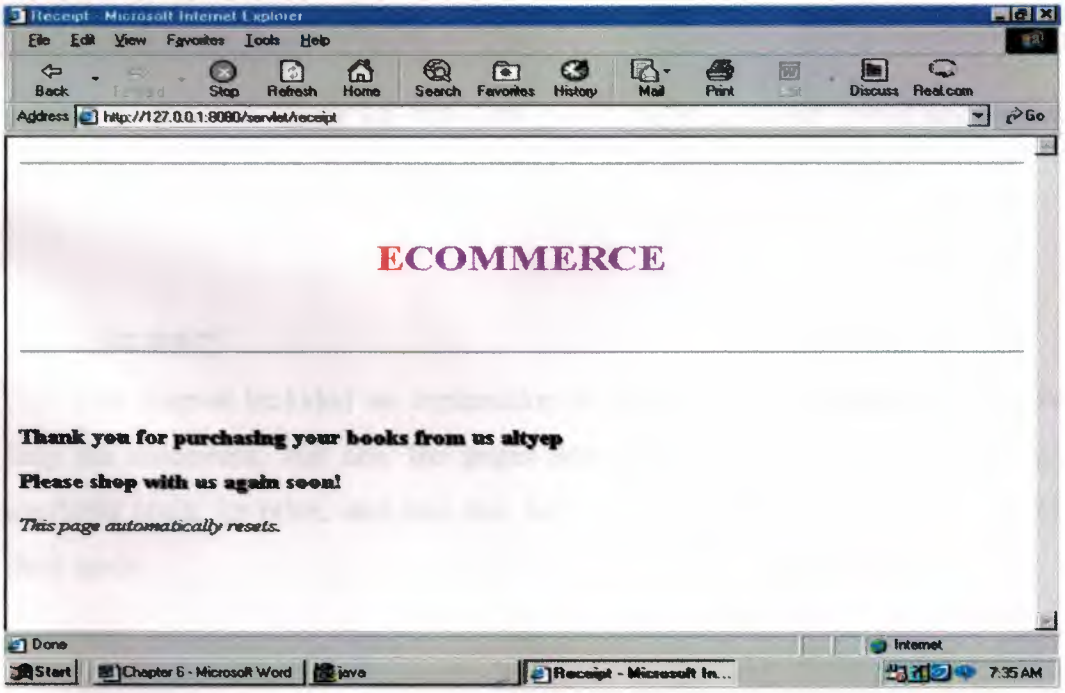

Figure 7.7. Thank the User

and last option which is "Clear Cart", which is give it to the customer's to clear his/her Cart. As shown in figure 7.8.

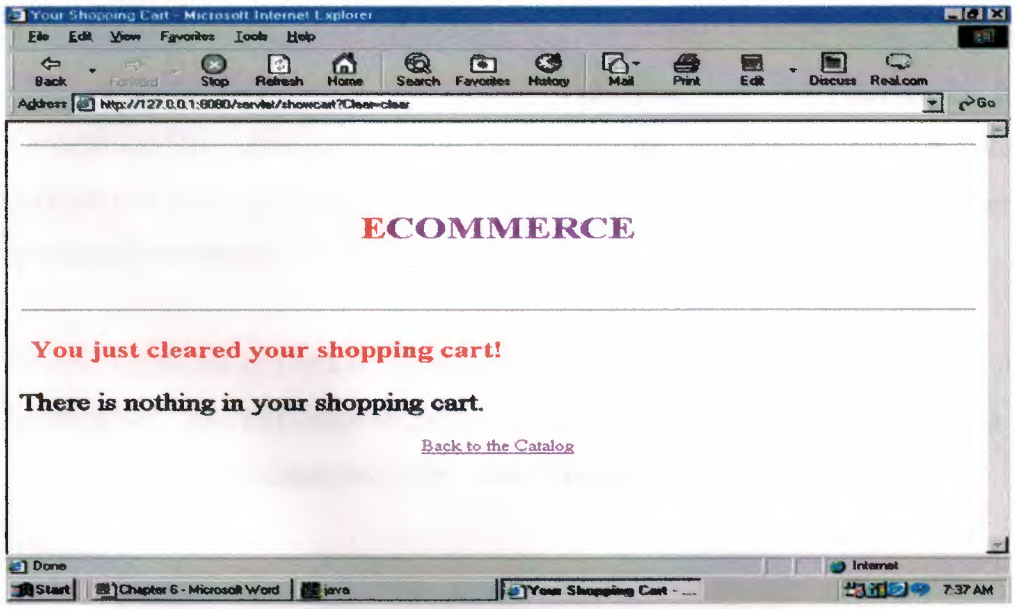

Figure 7.8. Clear Your Shopping Cart

7.2.3 Buy Your Book

As shown in figure 7.5. this class give it to the customer's total amount purchase and let the customer's to enter his/her Name and Credit Card Number.

7.3 Summary

This chapter included an explanation of the main page of the company that will help the customers, and also the pages that giving general information about each available book, its price, and also that how the customers can buy their books using their cards.

CONCLUSION

The project was about Electronic Commerce (Ecommerce), there is a revolution going on now in electronic commerce. As this project entered production already one-third of stock transactions were being transacted over the Internet by individuals. Companies like Amazon.com, barnesadnoble.com, and borders.com are handling huge volumes of electronic sales. Servlet technology will help more organizations get into electronic commerce.

There is also an explosion in so-called business-to-business transactions. As Internet transmission becomes more secure, more and more organizations will entrust increasing portions of their business activities to the Internet.

Companies are finding it possible to offer clients with Internet access all kinds of value-added services. Today, people often contact companies over the Internet and enter their own information rather than phoning it in verbally.

Two popular applications that many companies have off-loaded to the Web are shipment tracking and invoice tracking. People want to know where the products they have ordered are. The customers want to know when they will be paid.

Small business is particularly excited about electronic commerce possibilities. By putting up a respectable Web site, they present themselves to the world and can considerably leverage their business activities.

For client-server systems, many developers prefer a full java solution with applets on the client and servlets on the server.

REFERENCES

[1] Jason Hunter with William Crawford. Java servlet programming, O'reilly & associate. Inc., 101 Morris Street.1998.

[2] Elliote Rusty Harold, java network programming, O'reilly & associates, inc., 101 Morris Street 1997.

[3] Web site Address "http://java.sun.com/docs/tutorial/servlets/index.html".