Servlet and JSP technology has become the technology of choice for developing online stores, interactive Web applications, and other dynamic Web sites. Servlets are Java programs that run on Web or application servers, acting as a middle layer between requests coming from Web browsers or other HTTP clients and databases or applications on the HTTP server. Their job is to perform the following tasks.

1. Read the explicit data sent by the client.
   The end user normally enters this data in an HTML form on a Web page. However, the data could also come from an applet or a custom HTTP client program.

2. Read the implicit HTTP request data sent by the browser.
   Above Figure shows a single arrow going from the client to the Web server (the layer where servlets and JSP execute), but there are really two varieties of data: the explicit data that the end user enters in a form and the behind-the-scenes HTTP information. Both varieties are critical. The HTTP information includes cookies, information about media types and compression schemes the browser understands, and so forth.

3. Generate the results.
   This process may require talking to a database, executing an RMI or EJB call, invoking a Web service, or computing the response directly. Your real data may be in a relational database. Fine. But your database probably doesn’t speak HTTP or return results in HTML, so the Web browser can’t talk directly to the database. Even if it could, for security reasons, you probably wouldn’t want it to. The same argument applies to most other applications. You need the Web middle layer to extract the incoming data from the HTTP stream, talk to the application, and embed the results inside a document.

4. Send the explicit data (i.e., the document) to the client.
   This document can be sent in a variety of formats, including text (HTML or XML), binary (GIF images), or even a compressed format like gzip that is layered on top of some other underlying format. But, HTML is by far the most common format, so an important servlet/JSP task is to wrap the results inside of HTML.

5. Send the implicit HTTP response data.
   Sending HTTP response data involves telling the browser or other client what type of document is being returned (e.g., HTML), setting cookies and caching parameters, and other such tasks.

Why wait until the client requests the page and then have a program build the result? Why not just build the Web page ahead of time?

Yes, many client requests can be satisfied by prebuilt documents, and the server would handle these requests without invoking servlets. In many cases a static result is not sufficient, and a page needs to be generated for each request. There are a number of reasons why Web pages need to be built on-the-fly:

• The Web page is based on data sent by the client.
   For instance, the results page from search engines and order confirmation pages at online stores are specific to particular user requests. You don’t know what to display until you read the data that the user submits.

• The Web page is derived from data that changes frequently.
   If the page changes for every request, then you certainly need to build the response at request time. If it changes only periodically, however, you could do it two ways: you could periodically build a new Web page on the server (independently of client requests), or you could wait and only build the page when the user requests it. The right approach depends on the situation, but sometimes it is more convenient to do the latter: wait for the user request. For example, a weather report or news headlines site might build the pages dynamically, perhaps returning a previously built page if that page is still up to date.

• The Web page uses information from corporate databases or other server-side sources.
   If the information is in a database, you need server-side processing even if the client is using dynamic Web content such as an applet. Imagine using an applet by itself for a search engine site:
   “Downloading 50 terabyte applet, please wait!” Obviously, that is silly; you need to talk to the database. Going from the client to the Web tier to the database (a three-tier approach) instead of from an applet directly to a database (a two-tier approach) provides increased flexibility and security with little or no performance penalty. After all, the database call is usually the rate-limiting step, so going through the Web server does not slow things down. In fact, a three-tier approach is often faster because the middle tier can perform caching and connection pooling.

A Quick Peek at Servlet Code

```java
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.PrintWriter;
public class FirstServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response) throws java.io.IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head><title>My First Servlet</title></head>");
        out.println("<body bgcolor='#CCFFCC'>");
        out.println("My First Servlet");
        out.println("</body>");
        out.println("</html>");
    }
    public void doPost(HttpServletRequest request, HttpServletResponse response) {
    }
}
```
How to Deploy a Web Application?

1. Create a directory under C:\yourTomcatInstallation\webapps
2. Create the WEB-INF directory
3. Create deployment descriptor web.xml
4. Place the web.xml under WEB-INF folder
5. Create classes directory under WEB-INF
6. Compile your servlet and place it under classes
The Advantages of Servlets Over "Traditional" CGI

Java servlets are more efficient, easier to use, more powerful, more portable, safer, and cheaper than traditional CGI and many alternative CGI-like technologies.

**Efficient**

With traditional CGI, a new process is started for each HTTP request. If the CGI program itself is relatively short, the overhead of starting the process can dominate the execution time. With servlets, the Java virtual machine stays running and handles each request with a lightweight Java thread, not a heavyweight operating system process. Similarly, in traditional CGI, if there are \( N \) requests to the same CGI program, the code for the CGI program is loaded into memory \( N \) times. With servlets, however, there would be \( N \) threads, but only a single copy of the servlet class would be loaded. This approach reduces server memory requirements and saves time by instantiating fewer objects. Finally, when a CGI program finishes handling a request, the program terminates. This approach makes it difficult to cache computations, keep database connections open, and perform other optimizations that rely on persistent data. Servlets, however, remain in memory even after they complete a response, so it is straightforward to store arbitrarily complex data between client requests.

**Convenient**

Servlets have an extensive infrastructure for automatically parsing and decoding HTML form data, reading and setting HTTP headers, handling cookies, tracking sessions, and many other such high-level utilities. In CGI, you have to do much of this yourself. Besides, if you already know the Java programming language, why learn Perl too? You're already convinced that Java technology makes for more reliable and reusable code than does Visual Basic, VBScript, or C++. Why go back to those languages for server-side programming?

**Powerful**

Servlets support several capabilities that are difficult or impossible to accomplish with regular CGI. Servlets can talk directly to the Web server, whereas regular CGI programs cannot, at least not without using a server-specific API. Communicating with the Web server makes it easier to translate relative URLs into concrete path names, for instance. Multiple servlets can also share data, making it easy to implement database connection pooling and similar resource-sharing optimizations. Servlets can also maintain information from request to request, simplifying techniques like session tracking and caching of previous computations.

**Portable**

Servlets are written in the Java programming language and follow a standard API. Servlets are supported directly or by a plugin on virtually every major Web server. Consequently, servlets written for, say, Macromedia JRun can run virtually unchanged on Apache Tomcat, Microsoft Internet Information Server (with a separate plugin), IBM WebSphere, iPlanet Enterprise Server, Oracle9i AS, or StarNine WebStar. They are part of J2EE so industry support for servlets is becoming even more pervasive.

**Inexpensive**

A number of free or very inexpensive Web servers are good for development use or deployment of low- or medium-volume Web sites. Thus, with servlets and JSP you can start with a free or inexpensive server and migrate to more expensive servers with high-performance capabilities or advanced administration utilities only after your project meets initial success. This is in contrast to many of the other CGI alternatives, which require a significant initial investment for the purchase of a proprietary package. Price and portability are somewhat connected.

**Secure**

One of the main sources of vulnerabilities in traditional CGI stems from the fact that the programs are often executed by general-purpose operating system shells. So, the CGI programmer must be careful to filter out characters such as backquotes and semicolons that are treated specially by the shell. Implementing this precaution is harder than one might think, and weaknesses stemming from this problem are constantly being uncovered in widely used CGI libraries. A second source of problems is the fact that some CGI programs are processed by languages that do not automatically check array or string bounds. For example, in C and C++ it is perfectly legal to allocate a 100-element array and then write into the 999th "element," which is really some random part of program memory. So, programmers who forget to perform this check open up their system to deliberate or accidental buffer overflow attacks. Servlets suffer from neither of these problems. Even if a servlet executes a system call (e.g., with Runtime.exec or JNI) to invoke a program on the local operating system, it does not use a shell to do so. And, of course, array bounds checking and other memory protection features are a central part of the Java programming language.

**Mainstream**

There are a lot of good technologies out there. But if vendors don't support them and developers don't know how to use them, what good are they? Servlet and JSP technology is supported by servers from Apache, Oracle, IBM, Sybase, BEA, Macromedia, Caucho, Sun/iPlanet, New Atlanta, ATG, Fujitsu, Lutris, Silverstream, the World Wide Web Consortium (W3C), and many others. Several low-cost plugins add support to Microsoft IIS and Zeus as well. They run on Windows, Unix/Linux, MacOS, VMS, and IBM mainframe operating systems. They are the single most popular application of the Java programming language. They are arguably the most popular choice for developing medium to large Web applications. They are used by the airline industry (most United Airlines and Delta Airlines Web sites), e-commerce (ofoto.com), online banking (First USA Bank, Banco Popular de Puerto Rico), Web search engines/portals (excite.com), large financial sites (American Century Investments), and hundreds of other sites that you visit every day. Of course, popularity alone is no proof of good technology. Numerous counter-examples abound. But our point is that you are not experimenting with a new and unproven technology when you work with server-side Java.
The init Method

Most of the time, your servlets deal only with per-request data, and doGet or doPost are the only life-cycle methods you need. Occasionally, however, you want to perform complex setup tasks when the servlet is first loaded, but not repeat those tasks for each request. The init method is designed for this case; it is called when the servlet is first created, and not called again for each user request. So, it is used for one-time initializations, just as with the init method of applets. The servlet is normally created when a user first invokes a URL corresponding to the servlet, but you can also specify that the servlet be loaded when the server is first started. The init method definition looks like this:

```java
public void init() throws ServletException
{
    // Initialization code...
}
```

The init method performs two varieties of initializations: general initializations and initializations controlled by initialization parameters.

I. General Initializations

With the first type of initialization, init simply creates or loads some data that will be used throughout the life of the servlet, or it performs some one-time computation. If you are familiar with applets, this task is analogous to an applet calling getInitImage to load image files over the network: the operation only needs to be performed once, so it is triggered by init. Servlet examples include setting up a database connection pool for requests that the servlet will handle or loading a data file into a HashMap.

LotteryNumbers.java shows a servlet that uses init to do two things.

First, it builds an array of 10 integers. Since these numbers are based upon complex calculations, we don't want to repeat the computation for each request.

```java
// Initialization code...
{...
```

So, doGet looks up the values that init computed, instead of generating them each time.

Second, the output of the servlet does not change except when the server is rebooted, init also stores a page modification date that is used by the getLastModified method. This method should return a modification time expressed in milliseconds since 1970, as is standard with Java dates. The time is automatically converted to a date in GMT appropriate for the Last-Modified header. More importantly, if the server receives a conditional GET request (one specifying that the client only wants pages marked If-Modified-Since a particular date), the system compares the specified date to that returned by getLastModified, returning the page only if it has been changed after the specified date. Browsers frequently make these conditional requests for pages stored in their caches, so supporting conditional requests helps your users (they get faster results) and reduces server load (you send fewer complete documents). Since the Last-Modified and If-Modified-Since headers use only whole seconds, the getLastModified method should round times down to the nearest second.

II. Initializations Controlled by Initialization Parameters

In the previous example, the init method computed some data that was used by the doGet and getLastModified methods. Although this type of general initialization is quite common, it is also common to control the initialization by the use of initialization parameters. To understand the motivation for init parameters, you need to understand the categories of people who might want to customize the way a servlet or JSP page behaves.

1. Developers.
2. End users.
3. Deployers.

Developers change the behavior of a servlet by changing the code. End users change the behavior of a servlet by providing data to an HTML form (assuming that the developer has written the servlet to look for this data). But what about developers? There needs to be a way to let administrators move servlets from machine to machine and change certain parameters (e.g., the address of a database, the size of a connection pool, or the location of a data file) without modifying the servlet source code. Providing this capability is the purpose of init parameters.

Because the use of servlet initialization parameters relies heavily on the deployment descriptor (web.xml), we postpone details and examples on init parameters until the deployment descriptor chapter in Volume 2 of this book. But, here is a brief preview:

1. Use the web.xml servlet element to give a name to your servlet.
2. Use the web.xml servlet-mapping element to assign a custom URL to your servlet.
3. Add init-param subelements to the web.xml servlet element to assign names and values of initialization parameters.
4. From within your servlet's init method, call ServletConfig to obtain a reference to the ServletConfig object.
5. Call the getInitParameter method of ServletConfig with the name of the init parameter. The return value is the value of the init parameter or null if no such init parameter is found in the web.xml file.

The destroy Method

The server may decide to remove a previously loaded servlet instance, perhaps because it is explicitly asked to do so by the server administrator or perhaps because the servlet is idle for a long time. Before it does, however, it calls the servlet's destroy method. This method gives your servlet a chance to close database connections, halt background threads, write cookie lists or hit counts to disk, and perform other such cleanup activities. Be aware, however, that it is possible for the Web server to crash (remember those California power outages?). So, don't count on destroy as the only mechanism for saving state to disk. If your servlet performs activities like counting hits or accumulating lists of cookie values that indicate special access, you should also proactively write the data to disk periodically.
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.PrintWriter;
import java.io.IOException

public class LotteryNumbers extends HttpServlet
{
    private long modTime;
    private int[] numbers = new int[10];
    //The init method is called only when the servlet is first
    //loaded, before the first request is processed.
    public void init()
    {
        for(int i = 0; i < numbers.length; i++)
            numbers[i] = (int) (Math.random() * 100);
    }

    public void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>\n        <head>\n        <title>Your Lottery Numbers</title>
        <</head>\n        <body bgcolor='CCFFCC'>\n        <H2 ALIGN=CENTER>Your Lottery Numbers</H2>
        <B>Based upon extensive research of astro-illogical trends, psychic farces, and detailed statistical claptrap, we have chosen the best lottery numbers for you.</B>
        \n        <OL>
        for(int i=0; i<numbers.length; i++)
            out.println(" <LI>" + numbers[i]);
        \n        </OL>
        \n        </body>
        </html>\n    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
    {
    }
}
Reading Form Data from Servlets

One of the nice features of servlets is that all of this form parsing is handled automatically. You call request.getParameter to get the value of a form parameter. You can also call request.getParameterValues if the parameter appears more than once, or you can call request.getParameterNames if you want a complete list of all parameters in the current request.

```xml
<web-app>
  <servlet>
    <servlet-name>readParams</servlet-name>
    <servlet-class>ReadFormElements</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>readParams</servlet-name>
    <url-pattern>/params</url-pattern>
  </servlet-mapping>
</web-app>

<HTML>
<HEAD>
<TITLE>Collecting Three Parameters</TITLE>
</HEAD>
<BODY BGCOLOR="#FDF5E6">
<H2>Collecting Three Parameters</H2>
<form action="/forms/params">
  First Parameter: <input type="text" name="param1">
  Second Parameter: <input type="text" name="param2">
  Third Parameter: <input type="text" name="param3">
  <input type="submit">
</form>
</BODY>
</HTML>

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.PrintWriter;
import java.io.IOException
public class ReadFormElements extends HttpServlet {
  private String param1;
  private String param2;
  private String param3;
  public void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException {
    param1 = request.getParameter("param1");
    param2 = request.getParameter("param2");
    param3 = request.getParameter("param3");
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<html>");
    out.println("<head>");
    out.println("<title>Read Form Data</title>");
    out.println("</head>");
    out.println("<body bgcolor="#CCFFCC">");
    out.println("<p>Param 1 = " + param1 + "<br>");
    out.println("Param 2 = " + param2 + "<br>");
    out.println("Param 3 = " + param3 + "<br>");
    out.println("</body>");
    out.println("</html>");
  }
  public void doPost(HttpServletRequest request, HttpServletResponse response) {
  }
}
```