Short description and examples

Servlet

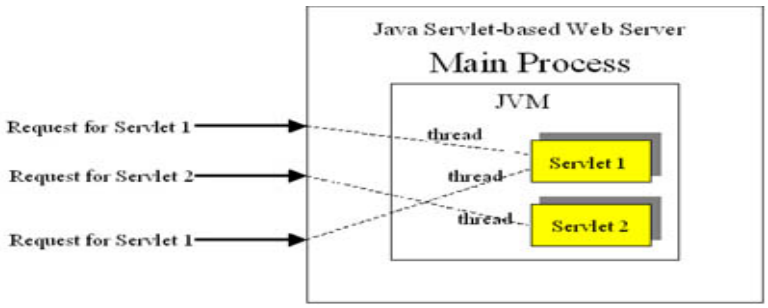
in Apache Jakarta-Tomcat

By Hamid Mosavi-Porasl

1. Few words about Servlet 3
1.1 The Servlet API 3
1.1.1 javax.servlet package 3
1.1 Writing Hello World 4
1.2 Running Hello World 4
2 Web Applications 4
2.1 Instance Persistence 5
2.2 A simple counter 5
2.3 Simple Synchronized Counter 6
2.4 A holistic counter 6
2.5 Init and Destroy 7
2.6 A Counter with Init 8
2.7 A counter with Init and Destroy 9
2.8 Single-Thread Model 11
2.9 Background Processing 13
2.10 Load on Startup 15
2.11 Client-Side Caching 16
2.12 Server-Side Caching 17
2.13 Getting a servlet init parameter 17
2.14 Getting a Servlet's Name 18
2.15 Getting Information about the server 18
2.16 Retrieving information about the Client 20
2.16.1 CGI environment variables and corresponding Servlet methods 20
2.16.2 Printing information about HTTP request header 20
2.17 Sending Multimedia Content 21
2.17.1 WAP and HTML 21
2.17.2 WML 22
2.17.3 WAP Device Simulators 24
2.18 Database Connectivity 24
2.18.1 The JDBC API 24
2.18.2 Getting Connection from a Servlet 24
2.18.3 A JDBC-Enabled Servlet 25
2.18.4 Result set in detail 26
2.18.5 Reusing Database Connections 27

1. Few words about Servlet

A Servlet is a generic server extension – a Java class that can be loaded dynamically to expand the functionality of a server. Servlets are commonly used with servers, where they can take the place of CGI scripts. Servlets operate solely within the domain of the server: unlike applets, they do not require support for Java in web browser.



This is easy to develop Servlets, Sun and Apache have made available the API classes separatly from any web engine. The **javax.servlet** and **javax.servlet.http** packages constitute this servlet API. The latest classes is available for download from http://java.sun.com/products/servlet/download.html.

It doesn't much matter where you get the servlet classes, as long as you have them on your system, since you need them to compile your servlets. In addition to the servlet classes, you need a servlet runner (technically called a ***servlet container***, sometimes called a ***servlet engine***)

1.1 The Servlet API

The servlets use classes and interfaces from two packages:

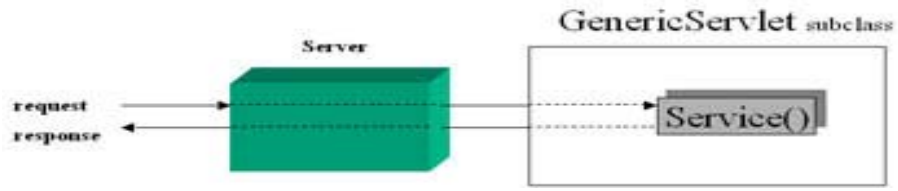
1. javax.servlet
2. javax.servlet.http

1.1.1 javax.servlet package

contains classes and interfaces to support generic, protocol-independent servlets. These classes are extended by the classes in the javax.servlet.http package to add HTTP-specific functionality.

Every servlet must implement the javax.servlet.servlet interfce. Most servlets implement this interface by extending one of two special classes: Javax.GenericServlet or javax.servlet.Http.HttpServlet. A protocol-independent servlet should subclass GenericServlet, while an HTTP servlet should subclass HttpServlet, which is itself a subclass of GenericServlet with added HTTP-specific functionality. Unlike a regular Java program, and just like an applet, a servlet does not have a main() method. Instead, certain methods of a servlet are invoked by the server in the process of handling requests. Each time the server dispatches a request to a servlet, it invokes the servlet's service() method.

A generic servlet should override its service() method to handle requests as appropriate for the servlet. The service() method accepts two parameters: a request object and a response object. The request object tells the servlet about the request, while response object is used to return a response.



1.1 Writing Hello World

```
import  java.io.*;
import javax.servlet.*;
import  javax.servlet.http.*;
public class HelloWorld extends HttpServlet{
        public void doGet(HttpServletRequest req, HttpServletResponse res)
                throws ServletException, IOException{
res.setContentType("text/html");
PrintWriter out =res.getWriter();
Out.println("<HTML>");
Out.println("<head><Title>Hello world</title></head>");
Out.println("<body>");
Out.println("<Big>Hello world</BIG>");
Out.println("</body></HTML>");
}
}
```

1.2 Running Hello World

When developing servlets we need two things: the servlet API class files, which are used for compiling, and a servlet container such as a web server, which is used for running the servlets.

A current list of servlet containers and what API levelthey support is available at http://www.servlets.com. If you are using the Apache Tomcat server , you should put the source code for the servlet in the server_root/webapps/ROOT/WEB_INF/classes directory. This is standard location for servlet classes. But other locations can also be used, i.e. server_root\webapps\servlets-examples\WEB-INF\classes. Add servlet information to server_root\webapps\servlets-examples\WEB-INF\web.xml, e.g.

```
<servlet>
      <servlet-name>Hello</servlet-name>
      <servlet-class>Hello</servlet-class>
  </servlet>
```

Due a bug in Tomcat maybe your changes in the servlet code can't be visible. In this case restart the server. Same bug does exist even for additional servlet classes added while server is up and running.

2 Web Applications

A web application (sometimes shorted to web apps) is a collection of servelts. Java Server Pages (JSP), HTML documents, images, templates and other web resources that are set up in such a way as to be portably deployed across any servlet-enabled web server.

All the files under *servlet_root/webapps/ROOT* belong to single web application. To simplify deployment , these files can be bounded into a single archive file and deployed to another server merely by placing the archive file into a specific directory. These archive file have extension "*.war*", which stands for *web application archive.* WAR files are actually JAR files (creating using the jar utility) saved with an alternate extension.

2.1 Instance Persistence

Servlets persists between requests as object instances. At the time the code for a servlet is loaded, the server createsa single instance. That single instance handle everz request made of the servlet. This improves performance in three ways:

- It keeps the memory footprint small
- It eliminates the object creation overhead that would otherwise be necessary to create a new servlet object. A servlet can already be loaded in a virtual machine when a request comes in, letting it begin executing right away.
- It enables persistence. A servlet can have already loaded anything it's likely to need during the handling of a request. For example, a database connection can be opened once and used repeatedly thereafter. The connection can even be used by a group of servlets. For example one background thread can perform some calculation while other threads display the latest results.

2.2 A simple counter

To demonstrate the servlet lifecycle check out following example:

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

public class SimpleCounter extends HttpServlet{
        int count=0;
        public void doGet(HttpServletRequest req,HttpServletResponse res)
                        throws ServletException,IOException{
                res.setContentType("text/plain");
                PrintWriter out=res.getWriter();
                count++;
                out.println("Since loading, this servlet has been accessed "+ count+" times");
        }
}
```

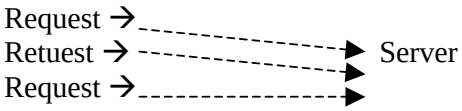
The code is simple - it just prints and increments the instance variable named count- but it shows the power of persistence.

When server loads this servlet, the servlet creates a single instance to handle every request made of the servlet. That's why this code can be so simple. The same instance variables exist between invocations and for all invocations.

I placed my SimpleCounter class file at:
C:\tomcat\webapps\servlets-examples\WEB-INF\classes, which is accessible via

http://localhost:8080/servlets-examples/servlet/SimpleCounter. I open several different http connections to same servlet. Then I can see that value of instance variable count increasing by one, independent from client instance. For example if client 1 tries to access the servlet 5 times, it can read value 1 to 5. Client 2, which starts after client 1 will see value 6 in its first try to access the servlet. This happens since variable count is declared as global in the class.

2.3 Simple Synchronized Counter



It is possible that if two requests are made to SimpleCounter around the same time, each will print the same value for count. The order of execution goes something like this:

```
Count ++        //Thread 1
Count++         //Thread 2
Out.println     // Thread 1
Out.println     // Thread 2
```

To prevent these types of problems and the inconsistencies that come with them, we can add one or more synchronized blocks to the code as below:

```
Public synchronized void doGet (HttpServletRequest req,
                                HttpServletResponse res)
```

Another option is to synchronize just tw0 lines we want to execute atomically:

```
PrintWriter out = res.getWriter();
Synchronized (this){
        Count++;
        Out.println("Since loading, this servlet has been accessed"+ count + " times.");
```

Third option would be to save the incremented value in a local variable in a synchronized statement and print out the local variable:

```
PrintWriter out = res.getWriter();
int local_count;
synchronized(this){
        local_count=++count;
        }
out.println("Since loading, this servlet has been accessed"+ local_count + " times.");
```

2.4 A holistic counter

The truth is that each registered name (but not each URL pattern match) for a servlet is associated with one instance of the servlet. This makes sense because the impression to the client should be that differently named servlets operate independently. The separate servlets are also a requirement for servlets that accept initialization parameters. Following example demonstrates with a servlet that counts three things. The time it has been accessed, the number of instances created by the server (one per name), and the total times all of them have been accessed.

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

public class HolisticCounter extends HttpServlet{
   static int classCount=0;   // shared by all insatnces
   int count=0;               // separated for each servlet
   static Hashtable instances= new Hashtable();   //also shared


  public void doGet(HttpServletRequest req, HttpServletResponse res)
                  throws ServletException, IOException{
    res.setContentType("text/plain");
    PrintWriter out =res.getWriter();
    count++;
    out.println("Since loading, this servlet instance has been accessed "+
                  count +" times");

    // Keep track of the instance count by putting a reference to this
    // instance in a hashtable. Duplicate entries are ignored.
    // The size() method returns the number of unique instances stored.
    instances.put(this,this);
    out.println("There are currently "+ instances.size() + "instances. ");
    classCount++;

    out.println("Access all instances, this servlet class has been "+" accessed "+classCount +"
times.");

   }
}
```

I invoke this servlet as follow:
http://localhost:8080/servlets-examples/servlet/HolisticCounter

For example out put is like follow:

```
Since loading, this servlet instance has been accessed 8 times
There are currently 1instances.
Access all instances, this servlet class has been accessed 8 times.
```

2.5 Init and Destroy

Just like applets, servlets can define init() and destroy() methods(). The server calls a servlet's init() method after the server constructs the servlet instance and before the servlet handles any requests. The server calls the destroy() method after the servlet has been taken out of service and all pending requests to the servlet have completed or time out. Depending on the server and the web application configuration, the init() method may be called at any of these times:

- When the server starts
- When the servlet is first requested, just before the service() method is invoked.

- At the request of the server administrator

In any case, init() is guaranteed to be called and completed before the servlet handles its first request. During the init() method a servlet may want to read its initialization (init) parameters. Init parameters for a servlet are set in web.xml deployment descriptor.

Example: Setting init parameters in the Deployment Descriptor

```
<?xml version=”1.0” encoding=”ISO-8859-1”?>
<!DOCTYPE web-app
 PUBLIC “-//Sun Microsystems, Inc.//DTD Web Application 2.2 //EN”
http://java.sun.com/j2ee/dtds/web-apps_2_2.dtd>

<web-app>
  <servlet>
    <servlet-name>
     counter
  </servlet-name>
 <servlet-class>
  InitCounter
</servlet-class>
<init-param>
  <param-name>
     initial
  </param-name>
<param-value>
1000
</param-value>
<description>
The initial value for the counter <!—optional-- >
</description>
</init-param>
</servlet>
</web-app>
```

Multiple <init-param> entries can be placed within the <servlet> tag. In a destroy() method, a servlet should free any resources it has acquired that will not be garbage collected.

2.6 A Counter with Init

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

public class InitCounter extends HttpServlet{
        int count;
        public void init() throws ServletException{
                String initial=getInitParameter(“initial”);
                Try{
                 Count=Integer.parseInt(initial);
                }
```

```
                catch(NumerFormatException e){
            count=0;
        }
}

public void doGet(HttpServletRequest req, HttpServletResponse res)
                throws ServletException, IOException{
        res.setContentType(“text/plain);
        PrintWriter out=res.getWriter();
        count++;
        out.println(“Since loading (and with a possible initialisation”);
        out.println(“Parameter figured in), this servlet has been accessed”);
        out.println(count+” times.”);
  }
}
```

2.7 A counter with Init and Destroy

Counter examples have demonstrated how servlet state persists between accesses. Every time the server is shut down or the servlet is reloaded, the count begins again. What we need is persistence across loads - a counter that doesn’t have to start over. Giving the servlet the ability to save its state in destroy() and load the state again in init().

Example below shows a fully persistent counter

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

public class InitDestroyCounter extends HttpServlet{
        int count;
        public void init()throws ServletException{
        // Try to load the initial count from our saved persistent state
        FileReader fileReader = null;
        BufferedReader bufferedReader =null;
        try{
        fileReader = new FileReader("InitDestroyCounter.initial");
        bufferedReader = new BufferedReader(fileReader);
        String initial = bufferedReader.readLine();
        count=Integer.parseInt(initial);
        return;
        }
        catch(FileNotFoundException ignored){}//no saved state
        catch(IOException ignored){}//Problem during read
        catch(NumberFormatException ignored){}//corrupt saved state
        finally{
        // Make sure to close the file
                try{
                if(bufferedReader != null){
                bufferedReader.close();
                }
```

```
        }
        catch (IOException ignored){}
        }

        // No luck with the saved state, check for an init parameter
        String initial = getInitParameter("initial");
        try{
        count =Integer.parseInt(initial);
        return;
        }
        catch(NumberFormatException ignored){ }//null or non-integer value
        //Default to an initial count of "0"
        count=0;
        }

        public void doGet(HttpServletRequest req,HttpServletResponse res)
                        throws ServletException, IOException{
                res.setContentType("text/plain");
                PrintWriter out = res.getWriter();
                count++;
            out.println("Since the begining, this servlet has been accessed"+
                count +" times.");
        }

        public void destroy(){
                super.destroy();//entirely optional
                saveState();
        }

        public void saveState(){
                //Try to save the accumulated count
                FileWriter fileWriter = null;
                PrintWriter printWriter= null;
                try{
                fileWriter = new FileWriter("InitDestroyCounter.initial");
                printWriter = new PrintWriter(fileWriter);
                printWriter.println(count);
                return;
                }
                catch(IOException e){// problem during write
                        //log the exception.
                }
                finally{// Make sure to close the file
                        if(printWriter != null){
                        printWriter.close();
                        }
                 }
        }
}
```

Don’t forget to add this class to web.xml file. Each time this servlet is unloaded, it saves its state in a file named InitDestroyCounter.initial. In the absence of a supplied path, the file is saved in the server process’s current directory, usually the startup directory ($TOMCAT_HOME). This file contains a single integer, saved as a string, that represents the latest count.

Each time this servlet is loaded, it tries to read the saved count from the file. If, for some reason, the read fails (as it does the first time the servlet runs because the file doesn’t yet exist), the servlet checks if an init parameter specifies the starting count. If that fails too, it starts fresh with zero. Servers can save their state in many different ways. Some may use a custom file format, as was done here. Others may save their state as serialized Java objects or put it into a database. Some may even perform *journaling*, a technique common to databases and tape backups, where the servlet’s full state is saved infrequently, while a journal file stores incremental updates as things change. It should be satisfactory to save state every 10 accesses. To implement this, we can add the following line at the end of doGet():

If (count % 10==0) saveState();

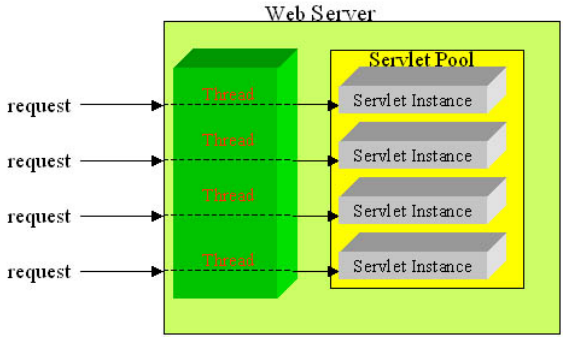
This possibility did not exist when saveState() was called only from the destroy() method: The destroy() method is called just once per servlet instance. Now that saveState() is called in doGet() method. It’s likely that two servlets (10 requests apart) will be in saveState() at the same time. This may result in a corrupted datafile. It’s also possible that two threads will increment count before either thread notices it was time to call saveState(). The fix is easy: move the count check into the synchronized block were count is incremented:

```
int local_count;
synchronized(this){
        local_count=++count;
        if(count %10==0) saveState();
}
out.println(“Since loading, this servlet has been accessed “+ local_count+” times.”);
```

2.8 Single-Thread Model

Although the normal situation is 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.

A servlet that loads a SingleThreadModel servlet must guarantee, according to the servlet API documentation , “that no two threads will execute concurrently in the servlet’s service method. To accomplish this, each thread uses a free servlet instance from the pool as shown in figure below:



Any servlet implementing SingleThreadModel can be considered thread safe and isn't required to synchronize access to its instance variables. Some servers use pools with just one instance, causing behavior identical to a synchronized service() method.

A servlet that connects to a database sometimes needs to perform several database commands atomically as part of a single transaction. Each database transaction requires a dedicated database connection object, so the servlet sometime. This could be done using synchronization, letting the servlet manage just one request at a time. By instead implementing SingleThreadModel and having one "connection" instance variable per servlet, a servlet can easily handle concurrent requests because each instance has its own connection.

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

public class SingleThreadConnection extends HttpServlet
   implements SingleThreadModel{
   Connection con=null;   //database connection, one per pooled instance

   public void init() throws ServletException{
         //Establish the connection for this instance
         try{
             con = establishConnection();

            con.setAutoCommit(false);
         }
         catch(SQLException e){
            throw new ServletException(e.getMessage());
         }
   }

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

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

         try{
            //Use the connection uniqely assigned to this instance
```

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

            //Update the database any number of ways

            //Commit the transaction
            con.commit();
          }
          catch(SQLException e){
           try {con.rollback();}catch(SQLException ignored){}
          }
  }

public void destroy(){
   if (con !=null){
         try{con.close();} catch(SQLException ignored){}
   }
}

private Connection establishConnection() throws SQLException{
   //Not implemented.
   return con;
}
}
```

A far better approach would be for the servlet to use a dedicated ConnectionPool object, held as an instance of class variable, from which it could “check out” and “check in” connections. The “check out” connection can be held as local variable, ensuring dedicated access. An external pool provides the servlet more control over the connection management. The pool can also verify the health of each connection, and the pool can be configured to always create some minimum number of connections but never create more than some maximum. When using SingleThreadModel, the sevrer might create many more instances (and thus more connections) than the databasecan handle.

2.9 Background Processing

Servlets can do more than simply persist between accesses. They can also execute between accesses. Any thread started by a servlet can continue executing even after the response has been sent. This ability proves most useful for long-running tasks whose incremental results should be made available to multiple clients. A background thread started in init() performs continuous work while request-handling threads display the current status with doGet(). It’s similar technique to that used in animation applets, where one thread changes the picture and another paints the display.

A servlet that searches for prime numbers above one quadrillion. It starts with such a large number to make the calculation slow enough to adequately demonstrated caching effects. The algorithm it uses couldn’t be simpler: it selects odd-numbered candidates and attempts to divide them by every odd integer between 3 and their square root. If none of the integers evenly divides the candidate, it is declared prime.

```
import java.io.*;
import java.util.*;
```

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

public class PrimeSearcher extends HttpServlet implements Runnable{
   long lastprime=0;    //Last prime found
   Date lastprimeModified= new Date();     // When it was found
   Thread searcher;         //Background search thread

   public void init() throws ServletException{
         searcher = new Thread(this);
         searcher.setPriority(Thread.MIN_PRIORITY); //be a good citizen
         searcher.start();
   }

   public void run(){
         // QTTBBBMMMTTTOOO
         long candidate = 1000000000000001L; //one quadrillion and one
         //Begin loop searching for primes
         while(true){                     //Search forever
            if(isPrime(candidate)){
                  lastprime = candidate;       // New prime
                  lastprimeModified= new Date();   // New "prime time"
            }
            candidate+=2;              //evens aren't prime

            // Between candidates take a 0.2 second break
            // Another way to be a good citizen with system resources
            try{
                  searcher.sleep(200);
            }catch(InterruptedException ignored) {}
         }
   }

   private static boolean isPrime(long candidate){
         // Try dividing the number by all odd numbers between 3 and its sqrt
         long sqrt = (long) Math.sqrt(candidate);
         for (long i=3;i<=sqrt;i+=2){
            if (candidate % 1 == 0) return false; //found a factor
         }
         // Wasn't evenley divisible, so it's prime
         return true;
   }

   public void doGet(HttpServletRequest req, HttpServletResponse res)
         throws ServletException, IOException{
         res.setContentType("text/plain");
         PrintWriter out = res.getWriter();
         if(lastprime == 0){
            out.println("Still searching for first prime ...");
            out.println("at "+ lastprimeModified);
         }
```

```
    }

    public void destroy(){
        searcher.stop();
    }
}
```

The searcher thread begins its search in the init() method. Its latest find is saved in last prime, along with the time it was found in lastPrimeModified. Each time a client accesses the servlet, the doGet() method reports the largest prime found so far and the time it was found. The searcher runs independently of client accesses; even if no one accesses the servlet it continues to find primes silently. If several clients access the servlet at the same time, they all see the same current status.

Notice that the destroy() method stops the searcher thread. This is very important. If a servlet doesn't stop its background threads, they continue to run until the virtual machine exits. Even when a servlet is reloaded (either explicitly or because its class file changed), its threads won't be stopped. Instead, it's likely that the new servlet will create extra copies of the background threads.

2.10 Load on Startup

To have the PrimeSearcher start searching for primes as quickly as possible, we can configure the servlet's web application to load the servlet at server start. This is accomplished by adding the <load-on-startup> tag to the <servlet> entry of the development descriptor as follow:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<web-app>
        <servlet>
          <servlet-name>
                ps
           </servlet-name>
          <servlet-class>
                PrimeSearcher
        </servlet-class>
        <load-on-startup/>
</servlet>
</web-app>
```

Note that the servlet instance handling the URL */servlet/PrimeSearcher* isn't loaded at startup. The <load-on-satrtup> tag shown above is empty. The tag can also contain a positive integer indicating the order in which the servlet should be loaded relative to other servlets in the context. Servlets with lower numbers are loaded before those with higher numbers. Servlets with negative values or noninteger values may be loaded at any time in the startup sequence, with the exact order depending on the server. For example, the web.xml shown below guarantees ***first*** is loaded before ***second***, while ***anytime*** could be loaded anytime during the server startup.

```
<web-app>
  <servlet>
```

```
  <servlet-name>
        first
</servlet-name>
<servlet-class>
        First
</servlet-class>
<load-on-startup>10</laod-on-startup>
</servlet>
 <servlet>
  <servlet-name>
        second
</servlet-name>
<servlet-class>
        Second
</servlet-class>
<load-on-startup>20</laod-on-startup>
</servlet>
<servlet>
  <servlet-name>
        anytime
</servlet-name>
<servlet-class>
        Anytime
</servlet-class>
<laod-on-startup/>
</servlet>
</web-app>
```

2.11 Client-Side Caching

Servlets handle GET requests with doGet() method. And that's almost true. The full truth is that not every request really needs to invoke doGet(). For example a web browser that repeatedly accesses PrimeSearcher should need to call a doGet() only after the searcher thread has found anew prime. Until that time any call to doGet() just generates the same page the user has already seen, a page probably stored in the browser's cache. What's really needed is a way for a servlet to report when its output has changed. That's where the getLastModified() method comes in.

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 the 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. Its 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 servlet can read this header and determine if the file has changed since the given time. 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 respond is the 304 Not Modified status code.

The extra help a servlet can provide is implementing the getLastModified() method. A servlet should implement this method to return the time it last changed its output. Servers call this method at two times. The first time the server calls the method is when the server returns a response, so that server 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 the server can intelligently determine how to respond. 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 returns the servlet's output.

Here's a getLastModified() method for our PrimeSearcher example that returns when the last prime was found:

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

Notice that this method returns a long value that represents the time as a number of milliseconds since midnight, January 1, 1970, GMT.

2.12 Server-Side Caching

The getLastModified() method can be used, with a little trickery, to help manage a server-side cache of the servlet's output. Servlets implementing this trick can have their output caught and catched on the server side, then automatically resent to clients as appropriate according to the servlet's getLastModified() method. This can greatly speed servlet page generation, especially for servlets whose output takes a significant time to produce but changes only rarely.

2.13 Getting a servlet init parameter

A servlet uses the getInitParameter() method for 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 doesn't exist. The return value is always a single String. It is up to the servlet to interpret the value.

The GenericServlet class implements the ServerConfig interface and thus provides direct access to the getInitOarameter() method. This means the method can be called like this:

```
Public void init() throws ServletException{
        String greeting = getInitParameters("greeting");
        }
```

We can assume a custom establishConnection() method to abstract away the details of JDBC. Using init parameters to establish a Database Connection :

```
java.sql.Connection con=null;
public void init() throws ServletException{
        String host = getInitParameter("host");
        int port =Integer.ParseInt(getInitParameter("port");
        String db = getInitParameter("db");
        String user = getInitParameter("user");
        String password = getInitParameter("password");
        String proxy = getInitParameter("proxy");

con = establishConnection(host,port,db,user,password,proxy);
}
```

2.14 Getting a Servlet's Name

In ServletConfig interface there's a method that returns the servlet's registered name:

```
Public String ServletConfig.getServerName();
```

Following code demonstrates how to use the servlet's name to retrieve a value from the servlet context, using the name as part of the lookup key:

```
        Public void doGet(HttpServletRequest req, HttpServletResponse res) throws
                                        ServletException, IOException{
                String name = getServletName();
                ServletContext context = getServletContext();
                Object value = context.getAttribute(name+".state);
```

Using the servlet name in the key, each servlet instance can easily keep a separate attribute value within the shared context.

2.15 Getting Information about the server

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

```
Public String ServletRequest.getServerName()
Public int ServletRequest.getServerPort()
```

Getting information about the server is an "About this server" servlet. Example:

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

public class ServerSnoop extends GenericServlet{

   public void service(ServletRequest req,ServletResponse res)
```

```
                    throws ServletException,IOException{
        res.setContentType("text/plain");
        PrintWriter out = res.getWriter();
        ServletContext context = getServletContext();
        out.println("req.getServerName(): "+req.getServerName());
        out.println("req.getServerPort(): "+req.getServerPort());
        out.println("context.getServerInfo(): "+context.getServerInfo());
        out.println("getServerInfo() name: "+
                getServerInfoName(context.getServerInfo()));
        out.println("getServerInfo() version: "+
                getServerInfoVersion(context.getServerInfo()));
      out.println("context.getAttributeNames():");
      Enumeration enumeration = context.getAttributeNames();
        while (enumeration.hasMoreElements()){
          String name = (String) enumeration.nextElement();
          out.println(" context.getAttribute(\""+name+"\"): "+
                    context.getAttribute(name));
        }
    }

    private String getServerInfoName(String serverInfo){
        int slash = serverInfo.indexOf('/');
        if (slash == -1) return serverInfo;
        else return serverInfo.substring(0,slash);
    }

    private String getServerInfoVersion(String serverInfo){
        // Version info is everything between the slash and the space
        int slash = serverInfo.indexOf('/');
        if (slash == -1) return null;
        int space = serverInfo.indexOf(' ',slash);
        if (space == -1) space = serverInfo.length();
        return serverInfo.substring(slash+1,space);
    }
}
```

Using URL http://localhost:8080/servlets-examples/servlet/ServerSnoop I get following output:

```
req.getServerName(): localhost
req.getServerPort(): 8080
context.getServerInfo(): Apache Tomcat/5.5.11
getServerInfo() name: Apache Tomcat
getServerInfo() version: 5.5.11
context.getAttributeNames():
context.getAttribute("org.apache.catalina.jsp_classpath"):
/C:/tomcat/webapps/servlets-examples/WEB-
INF/classes/;/C:/tomcat/shared/classes/;/C:/tomcat/common/classes/;/C:/tomc
at/common/i18n/tomcat-i18n-en.jar;/C:/tomcat/common/i18n/tomcat-i18n-
es.jar;/C:/tomcat/common/i18n/tomcat-i18n-
fr.jar;/C:/tomcat/common/i18n/tomcat-i18n-
ja.jar;/C:/tomcat/common/lib/commons-el.jar;/C:/tomcat/common/lib/jasper-
compiler-jdt.jar;/C:/tomcat/common/lib/jasper-
compiler.jar;/C:/tomcat/common/lib/jasper-
```

```
runtime.jar;/C:/tomcat/common/lib/jsp-api.jar;/C:/tomcat/common/lib/naming-
factory-dbcp.jar;/C:/tomcat/common/lib/naming-
factory.jar;/C:/tomcat/common/lib/naming-
resources.jar;/C:/tomcat/common/lib/servlet-
api.jar;/C:/tomcat/bin/bootstrap.jar;/C:/Programme/Java/jre1.5.0_04/lib/ext
/dnsns.jar;/C:/Programme/Java/jre1.5.0_04/lib/ext/localedata.jar;/C:/Progra
mme/Java/jre1.5.0_04/lib/ext/sunjce_provider.jar;/C:/Programme/Java/jre1.5.
0_04/lib/ext/sunpkcs11.jar
```
context.getAttribute("javax.servlet.context.tempdir"):
`C:\tomcat\work\Catalina\localhost\servlets-examples`
context.getAttribute("org.apache.catalina.resources"):
`org.apache.naming.resources.ProxyDirContext@933bcb`
context.getAttribute("org.apache.catalina.WELCOME_FILES"):
[Ljava.lang.String;@3ac93e

2.16 Retrieving information about the Client

For each request, a servlet has the ability to find out about the client machines and for pages requiring authentication, about the user.

2.16.1 CGI environment variables and corresponding Servlet methods

CGI	Servlet
SERVER_NAME	reg.getServerName()
SERVER_SOFTWARE	reg.getServletContext()
SERVER_PROTOCOL	reg.getProtocol()
SERVER_PORT	reg.getServerPort()
REQUEST_METHOD	reg.getMethod()
PATH_INFO	reg.getPathInfo()
PATH_TRANSLATED	reg.getPathTranslated()
SCRIPT_NAME	Reg.getServletPath()
DOCUMENT_ROOT	GetServletContext().getRealPath(“/”)
QUERY_STRING	Reg.getQueryString()
REMOTE_HOST	Reg.getRemoteHost()
REMOTE_ADDR	Reg.getRemoteAddr()
AUTH_TYPE	Reg.getAuthType()
REMOTE_USER	Reg.getRemoteUser()
CONTENT_TYPE	Reg.getContentType()
CONTENT_LENGTH	Reg.getContentLength()
HTTP_ACCEPT	Reg.getHeader(“Accept”)
HTTP_USER_AGENT	Reg.getHeader(“User-Agent”)
HTTP_REFERER	Reg.getHeader(“Referer”)

2.16.2 Printing information about HTTP request header

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

public class HeaderSnoop extends HttpServlet{
   public void doGet(HttpServletRequest req,HttpServletResponse res)
         throws ServletException,IOException{
```

```
        res.setContentType("text/plain");
        PrintWriter out = res.getWriter();
        out.println("Request Headers: ");
        out.println();
        Enumeration names = req.getHeaderNames();
        while(names.hasMoreElements()){
           String name =(String) names.nextElement();
           Enumeration values = req.getHeaders(name); // Support multiple values
           if(values != null){
                 while(values.hasMoreElements()){
                    String value= (String) values.nextElement();
                    out.println(name+ ": " +value);
                 }
           }
        }
   }
}
```

Output:

```
Request Headers:

accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/x-
shockwave-flash, application/vnd.ms-powerpoint, application/vnd.ms-excel,
application/msword, */*
accept-language: de
accept-encoding: gzip, deflate
user-agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; .NET CLR
1.1.4322)
host: localhost:8080
connection: Keep-Alive
```

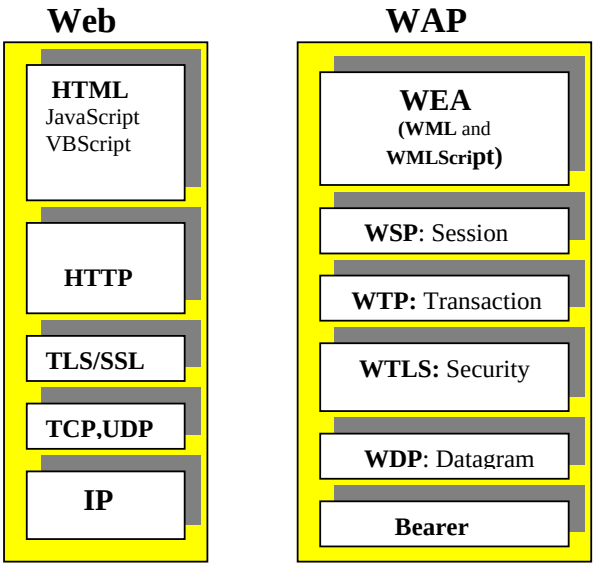
2.17 Sending Multimedia Content

The web consists of more than HTML

2.17.1 WAP and HTML

The Wireless Application Protocol (WAP) is a de facto standard for providing Internet communications to mobile phones, pagers, and personal digital assistants (PDAa) on Wireless networks across the world. It was created in 1998 by Ericsson, Nokia, Motorola, and phone.com, i.e. *www.wapforum.org*.

WAP consists of a set of specifications for developing applications to run on wireless networks. The WAP Protocols covers both the application level (the WML markup language and WMLScript scripting language, collectively known as the Web Application Environment or WAE) and the underlying network transport layers (the WDP, WTLS, WTP protocols). The WAP stack parallels the web protocol stack as following picture shows:

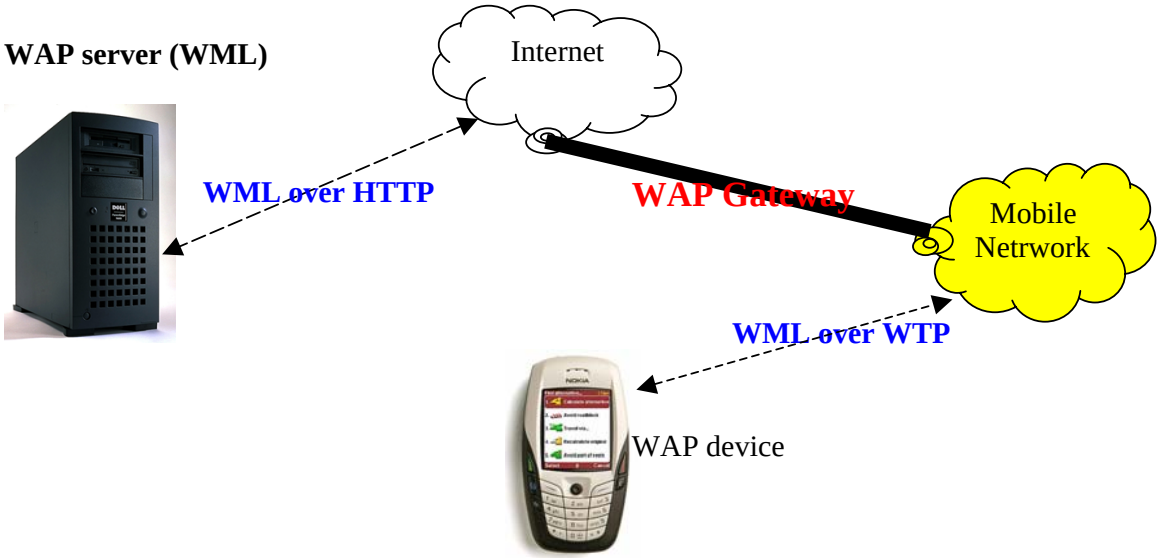


A WAP Gateway acts as an intermediary between the WAP network and the web. A gateway converts a WAP request into a web request, and the following web response into a WAP response. WAP Gateways give WAP devices access to standard web servers.

2.17.2 WML

WAP devices don't interact with normal HTML and image content. Instead, they use the Wireless Markup Language (WML) for text content, the WMLScript language for scripting, and the Wireless Bitmap (WBMP) monochromatic image format for graphics.

WML is an application of XML, similar to HTML but with far fewer tags.



Example below shows a static WML document that acts as a mini-bartender. It provides a list of drinks to choose from, then displays the ingredients of the selected drink. * Notice the document is XML with a well-known DTD.

```
<?xml version="1.0"?>

<!DOCTYPE wml PUBLIC
```

```
  "-//WAPFORUM//DTD WML 1.1/EN"
  "http://www.wapforum.org/DTD/wml_1.1.xml">

<wml>
<card id="Bartender" title="Select a Drink">
<p>
Select a Drink:

<anchor>
Kamikaze <go href="#Kamikaze" />
</anchor><br/>
<anchor>
 Margarita <go href="#Margarita" />
</anchor><br/>
<anchor>
Boilermaker <go href="#Boilermaker" />
</anchor><br/>
</p>
</card>

<card id="Kamikaze" title="Kamikaze">
<p>
To make a Kamikaze:<br/>
1 part Vodka<br/>
1 part Triple Sec<br/>
1 part Lime Juice<br/>
</p>
</card>

<card id="Margarita" title="Margarita">
<p>
To make a Margarita:<br/>
1 1/2 oz Tequila<br/>
1/2 oz Triple<br/>
1 ozLime Juice<br/>
Salt<br/>
</p>
</card>

<card id="Boilermaker" title="Boilermaker">
<p>
To make a Boilermaker:<br/>
2 oz Whiskey<br/>
10 oz Beer<br/>
1 ozLime Juice<br/>
Salt<br/>
</p>
</card>
</wml>
```

The document contains four cards. The first card, displayed by default, provides a short list of drinks. Each drink name is a hyperlink to another card in the document.

2.17.3 WAP Device Simulators

WAP device Simulators can be launched and be downloaded, among other links, at:

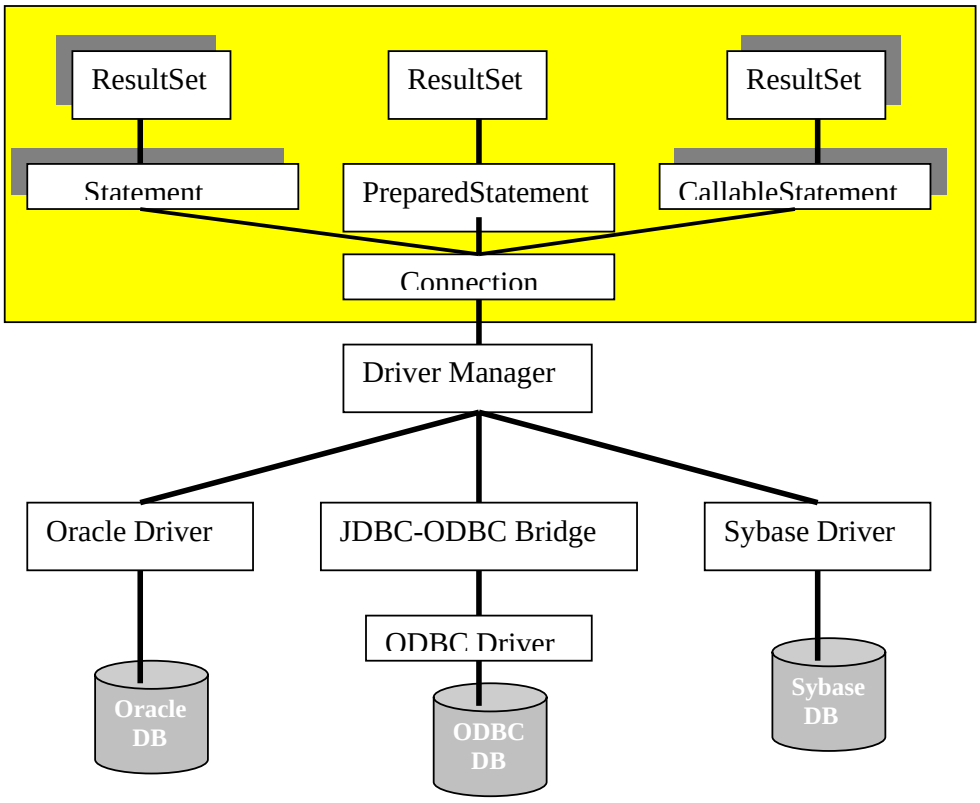
- http://www.yospace.com/

2.18 Database Connectivity

The biggest advantage for servlets with regard to database connectivity is that the servlet lifecycle allows servlets to maintain pools to open database connections. An existing connection can trim several seconds from a response time, compared to a CGI script that has to re-establish its connection for every invocation. Another advantage of servlets over CGI and many other technologies is that JDBC is database independent.

2.18.1 The JDBC API

JDBC is a SQL-level API – one that allows us 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.



2.18.2 Getting Connection from a Servlet

A servlet can use the same approach to load database connection information from a properties file stored under the web application's WEB-INF directory. The getResourceAsStream() method can retrieve the file's contents:

Properties props = new Properties();
InputStream in = getServletContext().getResourceAsStream("/WEB-INF/sql.properties");
props.load(in);
in.close();

The ContextProperties class below makes the context init parameters available as Properties Object. This allows all the init parameter name/value pairs to be passed to the DriverManager.getConnection() method.

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

public classProperties extends Properties{
   public ContextProperties(ServletContext context){
         Enumeration props = context.getInitParameterNames();
         while (props.hasMoreElements()){
            String name=(String) props.nextElement();
            String value = (String) context.getInitParameter(name);
            put(name,value);
         }
   }
}
```

A servlet uses this class instead of loading data from sql.properties, as shown below:

```
// Get the context init params as a properties object
ContextProperties props = new ContextProperties(getServletContext());

// load the driver
class.forName(props.getProperty("connection.driver"));
```

2.18.3 A JDBC-Enabled Servlet

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

public class DBPhoneLookup extends HttpServlet{
   public void doGet(HttpServletRequest req, HTTPServletResponse res){
         Connection con = null;
      Statement stmt= null;
      ResultSet rs = null;

   res.setContentType("text/html");
   PrintWriter out=res.getWriter();
   try{
         // Load (and therefore register) the Oracle Driver
         Class.forName("oracle.jdbc.driver.OracleDriver");
```

```
        // Get a Connection to the database
        con = DriverManager.getConnection(
              "jdbc:oracle:thin@dbhost:1528:ORCL","user","passwd");

        //Create a Statement object
        stmt = con.createStatement();

        //Execute an SQL query, get a ResultSet
        rs = stmt.executeQuery("SELECT NAME, PHONE FROM EMPLOYEES");

        //Display the result set as a list
        out.println("<HTML><HEAD><TITLE> Phonebook</TITLE></HEAD>");
        out.println("BODY");
        out.println("<UL>");
        while(rs.next()){
         out.println("<L>"+ rs.getString("name")+ " "+res.getString("phone"));
        }
        out.println("<UL>");
        out.println("</body></html>");
   }
   catch(ClassNoFoundException e){
        out.println("Couldn't load database driver: "+e.getMessage());
   }
   catch(SQLException e){
        out.println("SQLException caught: "+ e.getMessage());
   }
   finally{
        // Always close the database connection
        try{
           if (con != null) con.close();
        }
        catch(SqlException ignored) {}
   }
  }
}
```

All DBPhoneLookup does is connect to the database, run a query that retrieves the names and phone numbers of everyone in the employee table, and display the list.

2.18.4 Result set in detail

If we want to display the result of a query in an HTML table, it would be nice to have some Java code that builds the table automatically from the ResultSet rather than having to write the same loop-and-display code over and over.

The **ResultSetMetaData** interface provides a way for a program to learn about the underlying structure of a query result on the fly. We can use it to build an object that dynamically generates an HTML table from a ResultSet.

```
import java.sql.*;

public classHtmlResultSet{
```

```
    private ResultSet(ResultSet rs){
            this.rs=rs;
    }

    public String toString(){
      // Can be called at most once
    StringBuffer out = new StringBuffer();

    //Start a table to display the result set
    out.appened("<TABLE>\n");

  try{
            ResultSetMetaData rsmd = rs.getMetaData();
            int numcols = rsmd.getColumnCount();

            //Title the table with the result set's column labels
            out.append("<tr">");
            for(int i=1;i<=numcols;i++){
                    out.appened("<th>"+rsmd.getColumnLabel(i));
            }
     out.append("</tr>\n");

       while(rs.next()){
            out.append("<tr>"); //start a new row
            for(int i=1;i<=numcols;i++){
             out.append("<TD>");//Start a new data element
             Object obj = rs.getObject(i);
                    if(obj != null)
                      out.append(obj.toString());
                    else
                   out.println(" ");
                    }
             out.append ("</tr>\n");
            }
            //End the table
            out.append("</table>\n");
       }
    catch(SQLException e){
      out.appened("<table><H1>Error:</H1>"+e.getMessage()+ "\n");
    }
    return out.toString();
    }
}
```

2.18.5 Reusing Database Connections

A servlet can create one or more Connection objects in its init()method and reuse them in its service(), doGet(), and doPost()methods(). Example below shows a servlet, which creates its connection in advance. It uses the HtmlSQLResult class from previous chapter to display the rsuults:

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

public DBPhoneLookupReuse extends HttpServlet{
        private Connection con = null;

        public void init() throws ServletException{
                try{
                //Load (and therefter register) the sybase driver
                class.forName("com.sybase.jdbc.SybDriver");
                con=DriverManager.getConnection(
                        "jdbc:sybase:Tds:dbhost:7678","user","passwd");
           }
           catch(ClassNotFoundException e){
                throw new UnavailableException("Couldn't load database driver");
           }
           catch(SQLException e){
                throw new UnavailableException("Couldn't get db connection");
          }
        }

  public void doGet(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException{
          res.setContentType("text/html");
          PrintWriter out = res.getWriter();
        out.println("<HTML><HEAD><TITLE>Phonebook</title></head>");
        out.println("<BODY">);

        HtmlSQLResult result = new HtmlSQLResult("SELECT NAME, PHONE FROM
EMPLOYERS", con);

        //Display the resulting output
        out.println("<H2>Employees:</h2>");
        out.println("result);
        out.println("</BODY></html>");
        }

        public void destroy(){
                //Clean up
                try{
                if (con != null) con.close();
                }
                catch(SQLException ignored){}
        }
}
```