Accessing Your Pervasive.SQL Databases Online with J2EE

A beginner's guide to utilizing the power of the
Java 2 Enterprise Edition and Pervasive.SQL 2000i

Revision 1

April 2001

Pervasive Software Inc.

Pervasive Software Inc.
12365 Riata Trace Parkway II
Austin, Texas 78727
Public Relations Contact: Marian Kelley
Telephone: 800-287-4383/ 512-231-6000
Fax: 512-231-6010
Internet: http://www.pervasive.com

TABLE OF CONTENTS

INTRODUCTION

What is a Web Application?

A Web application is a piece of software which offers services to users via Web browsers. The interfaces to Web applications are written in a document markup language called HTML while the applications themselves are deployed on servers connected either to the Internet or local intranets. Web applications are capable of processing information entered by users, presenting information dynamically from databases, and communicating with other enterprise systems to automate business processes. Online stores, stock tickers, weather sites, and online games are all examples of Web applications.

The primary advantage of Web applications over traditional client/server applications is accessibility. Client/server apps require a special piece of software on each client to communicate with the server software. In addition, client/server applications are usually only accessible across individually built and maintained networks. In other words, you cannot access a traditional client/server application from a publicly available network such as the Internet. Since an overwhelming number of people already have access to the Internet, this is a major disadvantage. To utilize a Web application, all you need is a standard Web browser and an Internet connection.

What is Three-Tier Architecture?

Web applications generally consist of three tiers: a Web server, an application server, and a database server.

The Web server is responsible for communicating directly with the user's Web browser via a network protocol named HTTP. In general, HTML pages are sent from the Web server to the Web browser. However, any information may be sent such as software updates, graphics, or streaming video. Basically, the Web server distributes files from its own file system to the Web browsers requesting them. These files may be HTML files, movies, executables, etc. Examples of popular Web servers include Apache, Microsoft IIS, and Netscape Enterprise Server.

The application server handles requests too sophisticated for the Web server. These requests generally involve communication with other servers such as database servers or mail servers and may include traditional program logic such as if statements and iterative loops. Each application server operates a little differently, but in general, they execute special program files which provide instructions for outputting HTML, accessing database tables, sending e-mail, etc. Examples of popular application servers include Jakarta Tomcat, Microsoft IIS, and BEA WebLogic. Tomcat and WebLogic process servlets and Java server pages (JSPs), while IIS processes application server pages (ASPs). JSPs and ASPs look like HTML pages with embedded instructions to enable dynamic content, while servlets are written directly as Java classes.

Database servers are the foundation of almost all business applications. Legacy databases developed in the client/server and mainframe eras are often accessed and modified by much newer Web applications. Technologies such as structured query language (SQL), open database connectivity (ODBC), and Java database connectivity (JDBC) have greatly standardized the access methods for most database products. With careful planning, applications can be written which are independent of any particular database product. This allows organizations to migrate between databases over time, if necessary.

How is an HTTP Request Serviced?

To illustrate how all of the pieces work together, let's consider a feature found in many online applications, the login option. To set the scenario, imagine you're browsing your favorite online store when you decide to look at your shopping cart from last visit. To do this, you'll need to identify yourself to the system by logging in.

When you click the 'login' link, an HTTP request is sent to the server which contains the location, or URL, of the login HTML page. The Web server maps the URL to the correct HTML page in its file system and returns the

page to the user's Web browser. Because the login page itself never changes, the application server is not necessary to process the request.

Continuing with the scenario, let's say you fill out your username and password in the form and click *submit*. A second HTTP request is sent to the server with the URL of the application file needed to process your request. This time, the Web server realizes it cannot service your request and forwards it to the application server. The application server executes the commands necessary to:

- connect to the online store's database server
- check your login credentials against the database
- create a session and output a confirmation HTML page if your login succeeded
- output an HTML error page if your login failed

The login confirmation or denial is returned to the Web server where it is forwarded to the user's Web browser. Different architectures vary, but in general, the Web server communicates with the application server via a special Web server plug-in.

What are Servlets and JSPs?

A servlet is a series of commands written in Java which are executed in response to a user's request. You may use servlets to perform a wide variety of tasks, including database searches and updates, dynamic content display, file modifications, etc. Because they are written in Java, you may use servlets to exploit the full functionality of the Java platform.

Java server pages (JSPs) are compiled into servlets upon their first execution. The advantage of writing JSPs is that they look and feel more like standard HTML pages. For those familiar with creating static Websites, the familiarity of HTML is a real plus. In addition, Java code may be embedded within JSPs allowing you all of the capabilities of servlets. The performance differences between servlets and JSPs are minimal once the JSPs have been compiled. What is JDBC?

JDBC is a standard method for accessing databases through Java. Several database vendors have written their own JDBC drivers to support access to their data files through the standard JDBC classes and methods specified by Sun. Initially, the primary purpose of the JDBC driver was to pass SQL statements from Java applications to database engines and return the results in usable formats. Meta data could also be retrieved describing aspects of the database provider and returned result sets. With the advent of the JDBC 2.0 standard, however, a great deal more is possible. Result sets may be scrolled and updated, several different types of transactions may be carried out, and the meta data available is much more complete. The Pervasive.SQL JDBC driver supports most of the JDBC 2.0 standard. Not included is support for complex data types and methods not applicable to the Pervasive.SQL engine. For a complete listing of the supported JDBC features, please see your Pervasive.SQL product documentation.

What is JNDI and How Can It Help Me?

JNDI stands for Java Naming and Directory Interface. It allows you to reference certain system resources by a name which is resolved into an actual location at run-time. One of the resources which may be located via JNDI is a JDBC data source. To connect to a JDBC data source, you need the name of the JDBC driver class and a URL which defines the location of that data source. These data may be stored using JNDI so that if you want to change them in the future, you need only update the information in one place. This gives IT managers added flexibility in migrating between database providers because, in theory, no code would need to be updated to support the new database product.

Which Software Components Can I Download to Get Started?

Everything you need to begin developing servlets and JSPs is freely available from either Sun Microsystems or the Apache Software Foundation. At a minimum, you'll need to download the latest version of the Java Software Development Kit (SDK) and Jakarta Tomcat. The SDK is available from http://java.sun.com/j2se, while Tomcat is available from http://jakarta.apache.org/tomcat.

For a more robust configuration, you'll also need to download the Apache Web server and mod_jk plug-in. An installer for the Apache Web server may be found at http://httpd.apache.org while mod_jk is located with the Tomcat binaries.

Finally, you may purchase a copy of Pervasive.SQL from Pervasive's Website, http://www.pervasive.com, if you have not already done so. Pervasive.SQL is an ideal foundation for any Web application, as it is scalable, cross-platform, and accessible via ODBC, OLE-DB, JDBC, ActiveX, and more.

For more information on installing and configuring these software products, see the next section.

PART I - INSTALLATION & INITIAL CONFIGURATION

Overview

This section details the installation of the software that is needed to get your Pervasive.SQL databases to the Web. This software is available, widely used, standards-based, and supported by large communities such as the World Wide Web Consortium (W3C) and Sun Microsystems. Developing with these software components should afford your application a long life span.

There are innumerous ways to configure your system. Platform, desired use, and time will eventually determine your server configurations. We have outlined two different set ups—Minimal and Complete. The minimal configuration gets you a development environment, fast and easy. The complete configuration is more robust, and may be used when deploying your system to the Internet.

Both of these configurations are specific to the development and deployment of Java Server Pages (JSPs), and Java Servlets. Once an understanding of these technologies is in place, you can easily integrate a database by using Pervasive.SQL's JDBC driver (See *Part III*). For more information about configuring and enhancing your Pervasive.SQL database, please refer to the Pervasive.SQL product documentation. This document assumes Pervasive.SQL is already installed and running.

This document also assumes the installation and configuration mentioned herein will relate to the Microsoft Windows NT/2000 platform. The installation and configuration steps are extremely similar and have been tested on the Linux Operating System. The only differences involved are the ones to be expected, such as file structure and file type differences. For example, it is necessary to get the tarball of the Java Development Kit for a Linux machine, as opposed to the executable installer for Windows.

The final assumption we will make, as this is not a living document, is that the software versions are subject to change. Feel free to download the latest versions as they are probably more stable and contain new features. For any discrepancies found in our guide, consult the official product documentation. This guide is designed to provide a Web development mindset, not necessarily complete details on a product-by-product basis. The processes included in this paper typically require the smallest number of steps but are no substitute for the instructions found in the official product documentation. Instead, reading this document in conjunction with the product documentation should yield the best results.

Minimal Configuration

The minimal configuration utilizes the Apache Software Foundation's Jakarta Project. The servlet and JSP engine in the Jakarta Project is called Tomcat. It is necessary to obtain and install Tomcat, as well as the Java environment in which Tomcat must run.

JDK: Java Development Kit

The following steps can all be found on Sun's Web-site, http://java.sun.com. However, some direct paths are mentioned in the following steps. To install the Java SDK:

1. Point your Web browser to http://java.sun.com/j2se/1.3/download-windows.html and follow the steps to download j2sdk1_3_0-win.exe.
2. Run the installer and note where the root directory of the Java tree is placed, as this information is needed to configure Tomcat. The default root is typically C:\jdk1.3\.

After completing these steps, Java should be installed.

Tomcat: Servlet and JSP Engine

Just as with Java, most of the information here was learned and obtained from the Tomcat Web-site, http://jakarta.apache.org/tomcat/index.html. However, these comprehensive steps should provide some description as to what, not only how. The following steps will install the minimal Tomcat Servlet and JSP Engine:

1. At the writing of this document, the latest release of Tomcat could be found here: http://jakarta.apache.org/builds/tomcat/release/v3.1/bin/jakarta-tomcat.zip. Download this zip file from the Tomcat Website.
2. There is no installer, so simply unzip/extract the file. The default root directory is named C:\jakarta-tomcat\, yet this document refers to Tomcat's root directory as C:\tomcat\. If you wish to avoid confusion, please rename your directory structure to match. If, on the other hand, you feel you can handle this anomaly, leave your directory structure as is.
3. The tomcat user's guide, C:\tomcat\doc\uguide\tomcat_ug.html, states that:

   > Tomcat can be used as either a stand-alone container (mainly for development and debugging) or as an add-on to an existing Web server (currently Apache, IIS and Netscape servers are supported). This means that whenever you are deploying Tomcat you will have to decide how to use it.

   Read through the user's guide for more information, or follow the next steps to set up the stand-alone container.

   In order for the batch file that is used to start Tomcat to know where Tomcat and Java are located, a few environment variables must be created or appended. To access the environment variables in Windows 2000, right click on "My Computer" on the "Desktop". From the pop-up menu, click on "Properties" and you will be directed to the "System Properties" dialog box. Click on the "Advanced" tab, and then on the "Environment Variables..." button. For NT, right click on my computer, choose the “Properties” option, and select the “Environment” tab.

4. You will see *system*, and *user* environment variables. Using the interface, add the following two variables under the *system* section.

Variable	Value
JAVA_HOME	C:\jdk1.3
TOMCAT_HOME	C:\tomcat

5. Edit the PATH environment variable, and add to it "C:\jdk1.3\jre\bin". This puts the Java Run-time Environment in your system's path so that Tomcat can be executed.

Tomcat should now be installed and ready to run.

Starting and Stopping Tomcat

Once the configuration steps have been completed, as discussed in *Part I*, Tomcat can be run from a batch (.bat) file that has been included with the Tomcat package. Simply open up a command prompt and run C:\tomcat\bin\startup.bat. This batch file opens up a command prompt of its own, and a Java environment in which Tomcat executes. To test that Tomcat is now running, point your Web browser to http://localhost:8080/. There, you should find the Tomcat home page.

To stop Tomcat, simply close the command prompt that was opened by the startup batch file.

This system of starting and stopping Tomcat works, but is quite messy. It would make sense to have the Tomcat Server running as an NT Service (daemon). However, setting up Tomcat as an NT Service is outside the current scope of this document, as the command prompt utilization works fine.

The Jakarta project is apparently in the process of creating the ability to setup Tomcat as an NT Service. There is an executable available in the CVS repository at http://jakarta.apache.org/cvsWeb/index.cgi/jakarta-tomcat/bin/nt_service/Attic/

For information on installation and use, query the Jakarta newsgroups and the Web, where an abundant amount of information is available.

Complete Configuration

The complete configuration details the configuring of Tomcat to be run in conjunction with Apache, the world's leading Web server. At this point, Tomcat should be working on your machine as described in the previous section. Unfortunately, this minimal configuration uses a small HTTP server. When you deploy your Web application for enterprise use, you should use the Apache Web server and Tomcat servlet container in tandem. This can be done through the use of the mod_jk plug-in.

Apache: Web Server

Apache is simple to install on the Windows platform. The basic integration of Tomcat is also very straightforward. The power of these tools lies in the configuration files modifiable after installation. For more information on this advanced configuration, consult the Apache product documentation. To get started, follow the following steps:

1. Obtain the Apache installer from http://httpd.apache.org/dist/binaries/win32/
2. Run the installer
3. The typical install path is C:\Program Files\Apache Group\Apache\, but it expedites further configuration to use C:\Apache\ as the root.

Apache should now be installed and ready to serve Web pages.

Starting Apache

There are two ways to run Apache on the Windows NT/2000 Operating System; both are accessible from the Apache Program group in the Start menu.

1. "Start Apache" from the Start menu—This executes a .bat file within the context of a command prompt that appears on the desktop. Whenever you start Apache this way, you must leave this prompt open.
2. If you wish to install Apache as a service, choose the "Apache Install as Service" option from the Apache Program group. This creates a service allowing Apache to start automatically upon boot.

To test, start Apache using either method above, and point your Web browser to http://127.0.0.1/. You should see a Web page that says "It Worked".

mod_jk: Application Server Plug-in

mod_jk is a module that can be loaded at run time as a plug-in to the Apache Web server. By using mod_jk, your Website will gain the benefits and flexibility of Apache, while also providing the dynamic content associated with Tomcat. At the time of writing, mod_jk.dll was located at http://jakarta.apache.org/builds/jakarta-tomcat/release/v3.2.1/bin/win32/i386/.

Once you've downloaded the DLL, place it in C:\Apache\modules\, or the equivalent if you have a different Apache root. This places the module in a place where Apache can find it. To tell Apache to look for it, you must update the httpd.conf file located in C:\Apache\conf\.

Simply append the following line to the end of the httpd.conf file:

```
Include C:/tomcat/conf/mod_jk.conf-auto
```

This line includes the default mod_jk configuration file generated by Tomcat. This file is updated every time Tomcat is started. If you wish to enhance the configuration to suit the needs of your site, do not include the automatic file. Instead, use your own configuration. More information can be found in the mod_jk HOWTO file, mod_jk-howto.html, located in the doc directory of the Tomcat installation.

Whether you have installed the minimal or complete configuration, your system is now ready to execute servlets and JSPs.

PART II – SETTING UP A DEVELOPMENT ENVIRONMENT

Overview

Once Tomcat is installed, you should get a feel for what a servlet is, as well as how to create one in the Tomcat environment. Setting up an easy-to-use development environment has been made efficient by a tool called Ant, Jakarta's XML-based build tool. In the context of Tomcat, Ant is used to compile and deploy servlets and JSPs. Ant takes as input an XML file, build.xml, which delineates the deployment procedure.

Application Directory Structure

When creating a Tomcat Web application, there are two directory structures to keep in mind—development and deployment. The development directory is where you will create, delete, and modify source files, as well as place images, database files, and static HTML files. The deployment directory system is where Tomcat looks to run your application, and contains copies of most of the contents of the development directory in addition to the compiled servlets (.class files). It is this compiled Java code that Tomcat uses when executing servlets.

Table 1: Development Directory Structure

Directory	File(s)
Root	build.xml
Db	Database files
Etc	Web.xml
Images	GIFs and JPGs
Jsp	Java Server Pages
Lib	Java Archives (JARs)
Src	Servlet Source Code
Web	Static HTML Pages

Table 2: Deployment Directory Structure

Directory	File(s)
Root	Static HTML Pages & JSPs
Db	Database files
Images	GIFs and JPGs
Javadoc	Generated Java Documentation
WEB-INF	Web.xml
Classes	Servlet Classes
Lib	Java Archives (JARs)

Application Build Process

Table 3 shows the actions that the build tool performs to deploy a typical Web application.

Table 3: Web Application Build Process

Development C:\MyServlet	Action	Deployment C:\tomcat\Webapps\MyServlet\
Development root	copied to	deployment root
Db	copied to	Db
Images	copied to	Images
Jsp	copied to	deployment root
lib	copied to	WEB-INF\lib
src	compiled to	WEB-INF\classes
Web	copied to	deployment root

To get a feel for the development environment and the build process, let's take a look at the sample that Tomcat provides. To do so, open up a command prompt and your favorite text editor. In the command prompt, change the directory to C:\tomcat\doc\appdev\sample\ and perform a directory listing (dir). You should see two important files and three or four directories corresponding to *Table 1*. The files of importance are build.bat and build.xml. Build.bat is the script that executes Ant, which builds your Web application. Ant reads build.xml for instructions on building your Web application. Typing build on the command line causes the deployment directory structure to be created and populated according to build.xml. Any servlet source code found in the src directory will be compiled to the classes deployment directory.

If you look in the etc directory, you will see another XML file, Web.xml. Web.xml specifies mappings to your servlets, as well as other similar settings. More information on Web.xml and build.xml is available from the Jakarta product documentation, as well as from *Part V* of this document.

In the next section of this document, we will begin writing code. I would suggest either copying or using the sample directory directly as you proceed through the lessons ahead. The only file that you really need to touch would be the src/Hello.java file. All of the source code in *Part III* will work fine from there.

As you build your fundamental understanding of Jakarta and the servlet environment, you will be able to make configuration changes in your development environment which better suit your needs. The majority of these changes will be done within the build.xml and Web.xml configuration files.

PART III – WRITING JAVA SERVLETS

Installing the Samples Included with this Paper

There are two example applications included with this paper to help you get started. One is a collection of servlets which demonstrate individual JDBC and servlet features. The other is a simple shipping application named ABCExpress.com. Their development directories and source code are a part of the Webdev.zip file file available for download from the Pervasive Software Component Zone located at http://www.pervasive.com/componentzone (in the Developer Articles and Tips section). There, you will also find more complete build.xml files than are included with Tomcat.

Installing and Running the Servlet Samples

Copy samples.war included with this article into your Webapps directory under the Tomcat root. On my machine, this folder is simply c:\tomcat\Webapps. Start or restart your Tomcat server.

Assuming Pervasive.SQL is installed and running, you should be able to access the servlets from the following URL: http://localhost:8080/samples/. This application accesses Pervasive.SQL's sample database DEMODATA. If you have deleted this database since installing Pervasive.SQL, the samples will not function properly.

Installing and Running ABCExpress.com

Copy ABCExpress.war included with this article into your Webapps directory under the Tomcat root. On my machine, this folder is simply c:\tomcat\Webapps. Start or restart your Tomcat server. Since ABCExpress.com has its own database, you'll need to create an ODBC reference to it. From the control panel in Windows, add a system data source name (DSN) with the name 'ABCExpress Data'. The Pervasive.SQL named database should be named 'ABCEXPRESSDATA' and should reference a directory similar to mine: c:\tomcat\Webapps\ABCExpress\db. For more information on creating Pervasive.SQL ODBC data sources, please see the Pervasive.SQL product documentation.

Assuming Pervasive.SQL is installed and running, you should be able to access your ABCExpress application from the following URL: http://localhost:8080/ABCExpress/.

Writing a Hello World Servlet

To understand servlet development, it helps to understand how they are called from Tomcat. When Tomcat executes a servlet, it calls a service routine and passes in as arguments a request object and a response object. Additional information about the user's request can be retrieved from the request object, while the response object is used to return information to the user. For example, the user name and password you entered from a login page would be stored in the request object, while the print writer used to return HTML to the user's Web browser would be a part of the response object.

When executed, the following code displays 'Hello World' in a Web browser:

```
/* Hello World Servlet Sample */

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWorld extends HttpServlet {

 public void doGet(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, IOException
 {
  PrintWriter out = response.getWriter();
  response.setContentType("text/plain");

  out.println("Hello World");
 }
}
```

First, note that the servlet above, HelloWorld, extends the HttpServlet class. The HTTPServlet class contains a variety of service methods which may be defined in your own servlets. Unless you are passing post parameters, you will probably only need to implement the doGet method in your servlets (post parameters will be discussed shortly). As stated above, the service method takes as input the request and response objects passed in by Tomcat. This servlet does not use any information from the request object. However, it does utilize the print writer from the response object to output 'Hello World'. It also sets the content type of the HTTP response to *text/plain*. Although this is by far the most common type of HTTP response, you could also return data of other MIME types such as zip files.

Passing Servlet Parameters

To create interactive Web applications, it is often necessary to retrieve data from the user for processing. For example, when you shop online you enter your credit card information in an HTML form and submit the data to the server. This data is passed to the servlet processing your order as a part of the request object mentioned above.

There are two types of servlet parameters: get parameters and post parameters. Get parameters are appended to the end of the requested URL. For example, entering the following URL would cause two get parameters (firstname and lastname) to be passed to the fictitious HelloWorld servlet:

http://localhost/HelloWorld?firstname=Matt&lastname=Johnson

Post parameters are typically passed to servlets when a user submits an HTML form. In this case, it is advantageous to pass post parameters because, unlike get parameters, they are not displayed on the URL line of the browser. The following HTML generates a form which submits post arguments to the DisplayParameter servlet:

<FORM ACTION="http://localhost/samples/DisplayParameter" METHOD="post">

Get Parameters

The following line of HTML displays a link which submits two get parameters to the DisplayParameter servlet:

<A HREF="http://localhost/DisplayParameter?param1=value1&param2=value2">Click Me</A>

The following servlet code displays the values of the two get parameters:

```
/* Get Parameter Sample */

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.*;
import javax.servlet.http.*;

public class DisplayParameter extends HttpServlet {

 public void doGet(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, IOException
 {
  PrintWriter out = response.getWriter();
  response.setContentType("text/plain");

  out.println("Parameter 1: " + request.getParameter("param1"));
  out.println("Parameter 2: " + request.getParameter("param2"));
 }
}
```

The method getParameter() takes as input the name of the parameter and returns its value as a string. For parameters with more than one value, you will need to use getParameterValues() which returns an array.

Post Parameters

Post parameters may be retrieved in much the same way as get parameters. The getParameter() and getParameterValues() methods operate the same in both cases. However, post parameters are only available through a special service routine, doPost(). The following HTML form prompts the user to enter their first and last names and submits their values as post arguments:

```
<FORM ACTION="http://localhost/PostParameter" METHOD="POST">
 First Name: <INPUT TYPE="text" NAME="firstname">
 Last Name: <INPUT TYPE="text" NAME="lastname">
     <INPUT TYPE="submit">
</FORM>
```

The action attribute of the form specifies the servlet whose doPost() method will be executed upon submission. Following is a simple servlet which prints the values of the two parameters. Notice, the service method is doPost(), not doGet().

```
/* Post Parameter Sample */

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.*;
import javax.servlet.http.*;

public class PostParameter extends HttpServlet {

 public void doPost(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, IOException
 {
  PrintWriter out = response.getWriter();
  response.setContentType("text/plain");

  out.println("First Name: " + request.getParameter("firstname"));
  out.println("Last Name: " + request.getParameter("lastname"));
 }
}
```

Connecting to Pervasive.SQL Through JDBC

When connecting to any database via JDBC, two steps are required. First, you must initialize the JDBC driver. With Pervasive.SQL, you would use the following code:

```
class.forName("com.pervasive.jdbc.v2.Driver");
```

Next, you must create a connection to the database you would like to access. This requires that you know the database's name and location. The following code accesses the DEMODATA database on the local machine:

```
Connection conn = DriverManager.getConnection("jdbc:pervasive://127.0.0.1:1583/DEMODATA");
```

By convention, 127.0.0.1 is the IP address of the local machine. This can, of course, be replaced with another machine name if your database server and application server are on different machines. 1583 denotes the port on which Pervasive.SQL listens, and DEMODATA is the ODBC name of the database being accessed. The following code is a complete servlet which connects to the DEMODATA database:

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

public class MakeConnection extends HttpServlet {

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

  try {
   /* Initialize JDBC Driver */
   Class.forName("com.pervasive.jdbc.v2.Driver");

   /* Create Connection to DEMODATA */
   Connection conn;
   conn = DriverManager.getConnection("jdbc:pervasive://127.0.0.1:1583/DEMODATA");

   /* Access Database... */

   /* Close Database Connection */
   conn.close();
  }
  catch (ClassNotFoundException eCNF) {
   out.println("caught exception: " + eCNF);
  }
  catch (SQLException eSQL) {
   out.println("SQL Exception: " + eSQL);
  }
 }
}
```

Finally, if your database has login security, you may pass a user name and password into the getConnection() method as follows:

```
String url = "jdbc:pervasive://127.0.0.1:1583/DEMODATA";
Connection conn = DriverManager.getConnection(url, "username", "password");
```

Retrieving JDBC ResultSets

A JDBC result set can be thought of as a table of data which is read and manipulated one row at a time. This row is determined by the current position of the result set and is the target of all available read and write operations. The position of the result set may be modified using navigation commands such as previous(), next(), first(), and last(). In addition, the result set position may be either before the first row or after the last row. These two special positions may at first seem unnecessary. However, they turn out to be useful in systematically working with data.

ResultSet Types

There are two attributes of result sets which may be modified to customize their behavior, concurrency and scrolling type. The two possible values for concurrency are *read-only* and *updateable*. As the terms imply, read-only result sets cannot be modified while updateable result sets can.

The scrolling type attribute deals with how the result set behaves while you are scrolling through its records. *Forward-only* result sets are efficient, but somewhat inflexible. Of all of the methods available for moving between records in a result set, only the next() method is valid for forward-only result sets. *Scrollable-insensitive* and *scrollable-sensitive* result sets allow you to use the full array of navigation methods. Scrollable-insensitive result sets are not continually updated against the database. In other words, if new records are added to the database or existing records are updated after you retrieve a result set, these changes will not be visible to you. In other words, you have a snapshot of the database at the time you queried it. As expected, scrollable-sensitive result sets exhibit the opposite behavior. Although concurrency and scrolling type are attributes of result sets, you must set these attributes in the statements that create those result sets. Below are examples which demonstrate this:

```
/* Create statement to generate read-only, forward-only result sets */
Statement stmt;
stmt = conn.createStatement();

/* Create statement to generate updateable, scrollable-insensitive result sets */
Statement stmt;
stmt = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATEABLE);

/* Create statement to generate read-only, scrollable-sensitive result sets */
Statement stmt;
stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_READ_ONLY);
```

Executing SQL SELECT Statements

SQL is a standardized query language designed to search, update, insert, and delete data in a database. SQL SELECT statements retrieve a relation, or table of data, drawn from one or more database tables. Once retrieved, the relation is stored as a ResultSet in Java. Learning SQL is outside the scope of this article. For more information, see the Pervasive.SQL documentation or a book on SQL.

Following are several examples showing how result sets may be retrieved:

```
/* Return all records from the 'department' table */
ResultSet rs = stmt.executeQuery("SELECT * FROM Department");

/* Return 'Art' department data from the department table */
ResultSet rs = stmt.executeQuery("SELECT * FROM Department WHERE Name = 'Art'");

/* Execute a prepared statement with one parameter passed into the servlet */
PreparedStatement pstmt;
pstmt = conn.prepareStatement("SELECT * FROM Department WHERE Name = ?");
pstmt.setString(1,request.getParameter("deptName"));
ResultSet rs = pstmt.executeQuery();
```

The last example uses a prepared statement with one parameter. The parameter value is set equal to the servlet parameter, deptName.

Navigating and Displaying ResultSets

Result sets may only be accessed one row at a time. Thus, to manipulate an entire result set, you must be able to move between its rows. Numerous methods have been created for this purpose. With a forward-only result set, you may only use the next() method. However, with scrollable result sets, you may use:

Navigation Method
void beforeFirst()
void afterLast()
boolean first()
boolean last()
boolean previous()
boolean next()
boolean absolute(int row)
boolean relative(int displacement)

Several of the methods return boolean values. This value denotes whether or not a record was successfully found at that location. This information may be used in a while loop to discover the end of a result set.

Additional methods have been provided to test the current position of the result set. You may test for the special positions *before first* and *after last* as well as for the first and last record. In summary:

Position Testing Method
boolean isBeforeFirst()
boolean isAfterFirst()
boolean isFirst()
boolean isLast()

Below is a complete servlet demonstrating the next() method:

```
/* Result set servlet example */

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

public class DisplayDepts extends HttpServlet {
 public void doGet(HttpServletRequest request, HttpServletResponse response)
  throws IOException, ServletException
 {
  response.setContentType("text/html");
  PrintWriter out = response.getWriter();

  try {
   /* Establish database connection */
   Class.forName("com.pervasive.jdbc.v2.Driver");
   Connection conn;
   conn = DriverManager.getConnection("jdbc:pervasive://127.0.0.1:1583/DEMODATA");

   out.println("<HTML>");
   out.println("<BODY>");

   /* Retrieve departments */
   Statement stmt = conn.createStatement();
   ResultSet rs = stmt.executeQuery("SELECT * FROM Department");

   out.println(" <TABLE>");
   out.println("  <TR>");
   out.println("   <TH>Name</TH>");
   out.println("   <TH>Building</TH>");
   out.println("   <TH>Room Number</TH>");
   out.println("   <TH>Phone Number</TH>");
   out.println("  </TR>");

   while (rs.next()) {
    out.println("  <TR>");
    out.println("   <TD>" + rs.getString("Name") + "</TD>");
    out.println("   <TD>" + rs.getString("Building_Name") + "</TD>");
    out.println("   <TD>" + rs.getInt("Room_Number") +"</TD>");
    out.println("   <TD>" + rs.getLong("Phone_Number") + "</TD>");
    out.println("  </TR>");
   }

   out.println(" </TABLE>");
   out.println("</BODY>");
   out.println("</HTML>");

   conn.close();
  }
  catch (ClassNotFoundException eCNF) { out.println("caught exception: " + eCNF); }
  catch (SQLException eSQL) { out.println("SQL Exception: " + eSQL); }
 }
}
```

There are a few things worth mentioning about the sample servlet above. First, the servlet uses a while loop to iterate through the result set. The loop terminates when the next() method returns false. Next, the getXXX() methods are used to display the result set data. These methods may take either a string denoting the column name or an integer denoting the column number (starting from 1). Each method returns its respective primitive type. In other words, getString() returns a string, getInt() returns an int, etc.

Updating Data via SQL

SQL statements were the only means for updating data under the JDBC 1.0 standard and will continue to play a major role in JDBC development. The JDBC executeUpdate() method allows you to execute INSERT, UPDATE, and DELETE SQL statements in exactly the same manner. Below are examples which should get you started writing your own JDBC code:

```
/* Insert record #1253, 'Matt Williams', into database */
stmt.executeUpdate("INSERT INTO Person VALUES ('1253', 'Matt', 'Williams')");

/* Exchange the first name 'John' for every 'Matt' */
stmt.executeUpdate("UPDATE Person SET FName = 'John' WHERE FName = 'Matt'");

/* Change the last name of every 'John' to 'Smith' */
PreparedStatement pstmt;
pstmt = conn.prepareStatement("UPDATE Person SET LName = ? WHERE FName = ?");
pstmt.setString(1, "Smith");
pstmt.setString(2, "John");
pstmt.executeUpdate();

/* Remove the people named 'John Smith' from the database */
stmt.executeUpdate("DELETE FROM Person WHERE Name = 'John Smith'");
```

Updating Data via Updateable ResultSets

Remember, result sets can be thought of as tables of data you can view and update one row at a time. To extend the analogy, imagine that a result set had a blank row at the bottom for a new record. With this state of mind, updating data with updateable result sets is simple.

Inserting Records

To insert records, you'll need to:

1. Move to the special *insert row*.
2. Set the values by column for the new record.
3. Insert the record into the database.

The following code snippet demonstrates these steps:

```
/* Step 1: Move to 'insert row" */
rs.moveToInsertRow();

/* Step 2: Set data values of new record */
rs.updateInt("PersonID", 1253);
rs.updateString("FName", "Matt");
rs.updateString("LName", "Williams");

/* Step 3: Insert record into database */
rs.insertRow();
```

Updating Records

The process for updating records is very similar to inserting records. Once you have positioned the result set at the record you'd like to update, perform the following steps:

1. Set the values by column for the new record
2. Update the record in the database

Below is a code snippet demonstrating a result set update:

```
/* Step 1: Change first name in current row to 'John' */
rs.updateString("FName", "John");

/* Step 2; Update row in database */
rs.updateRow();
```

Deleting Records

The procedure for deleting records using result sets is simple. For a result set object named rs already positioned at the record to be deleted, the code would be:

```
rs.deleteRow();
```

Maintaining User Login Sessions

By design, HTTP is a stateless network protocol in which individual requests are independent of one another. As a designer, it is your responsibility to relate these requests into a continuous user session, if necessary. Sessions are important because most business systems are role-based. Role-based systems grant special privileges to different types of users. Without sessions, users would have to identify themselves to the system each time they accessed these special privileges. Sessions allow Web applications to "remember" who you are between requests so you only have to login once.

Most application servers provide mechanisms for tracking user sessions. In general, they store information on the server which is tagged with a special session identifier. Given the session identifier, the application server can return all of the information related to that session. At a minimum, the person's username must be stored as part of the session so the application can determine who is using the system. Other information such as security level, name, and address may be cached for performance reasons.

There are two commonly-used methods for passing a user's session identifier to the server. The most elegant solution is to write a cookie on the user's machine which stores the identifier when a session is created. Subsequent HTTP requests include the contents of the cookie, allowing the system to bind the requests together into a session. Unfortunately, not all Web browsers are set to accept cookies. The other method works in much the same way except the session identifier is passed as a part of the URL. In Java, you would link to a URL similar to the following:

http://localhost/samples/HelloWorld?JSessionID=13421

The disadvantage of this method is that each link must be explicitly coded to pass this parameter. Most, if not all, servlet and JSP containers handle the details of session tracking for you. Tomcat, for example, silently writes a cookie if possible. However, it's good practice to pass the JSessionID parameter as part of your URLs just in case cookies are not accepted.

If the application server cannot find a valid session identifier in the user's HTTP request, a new session identifier is generated, effectively starting a new session. Information may be attached to this session and retrieved in response to a later HTTP request. In a stateless system, a new session is created each time a servlet is executed, giving the appearance of no sessions at all.

Adding Session Attributes

To add information, or attributes, to a session, you must first create an object which references the current session. This may be done with the following code:

```
HttpSession session = request.getSession();
```

Session attributes are value-key pairs in which the key is a textual description and the value is an object. The following code stores the current user's username (as entered from a login form) as a session attribute named currentUser:

```
session.setAttribute("currentUser", request.getParameter("username"));
```

Although the getParameter() method returns a string, any type of object could have been attached to the session as an attribute.

Terminating Sessions

When an application server terminates a user session, it:

1. Deletes the current session identifier from the list of valid sessions
2. Frees the attributes associated with the old session
3. Generates a new session identifier

The only code required from the developer's perspective is:

```
session.invalidate();
```

where session is an object referencing the current session.

Modifying Session Time-Outs

One of the issues in creating session-oriented Web applications is determining when the user has quit accessing the system. When a user browses to another Website, no message is sent to the original Website. In other words, from the application server's perspective, there's no way to tell whether the user has:

a) closed the browser
b) browsed to another site
c) just been taking their time at your site
d) left their computer completely and gone to play golf

If sessions were never terminated, the server would eventually become overloaded with session variables and crash. To handle this problem, application servers periodically terminate inactive sessions. By default, Tomcat uses a thirty-minute time-out. However, this value may be updated in Tomcat's primary Web.xml file located in its conf directory.

PART IV - ADVANCED WEB APPLICATION FEATURES

Using an HTML Popup Calendar

One of the difficult issues in designing Web applications is inputting dates. There's no standard date format, and the user usually has to refer to a calendar to select the correct date. To help address these issues, a popup calendar has been included with this article, courtesy of Pervasive. It's written as a Java servlet and handles all dates available within Java's date constraints. There are no restrictions on how you may use either the popup calendar source code or class file.

Adding the Calendar to an HTML Page

First, you must include PopupCalendar.class with your other servlet class files in your deployment directory. In your HTML page, you'll need to include two separate pieces of code. First, place the following JavaScript either in the HTML header or at the beginning of the HTML body:

```
<SCRIPT LANGUAGE="JavaScript">
 function displayCalendar(inputField) {
  window.dateField = inputField;
  window.open('PopupCalendar','cal','WIDTH=200,HEIGHT=260');
 }
</SCRIPT>
```

In the above code, PopupCalendar serves as the relative URL to the PopupCalendar servlet. If this servlet is located elsewhere, this value will need to be changed. Next, create a link to the calendar next to a text field. For example:

```
<FORM METHOD='POST' ACTION='ShowDate'>
 <!-- Date text field -->
 Birthday: <INPUT TYPE="text" NAME="birthday">

 <!-- Popup calendar link, must reference text field, 'birthday' -->
 <A HREF='#' onClick='displayCalendar(birthday)'>Calendar</A>

 <!-- Form submit button -->
 <INPUT TYPE=SUBMIT VALUE=Submit>
</FORM>
```

For complete samples demonstrating the calendar's usage, refer to the source code of the included sample servlets.

Sending E-mail

Writing servlets which send e-mail is not elementary. It requires using the classes in the javax.mail packages as well as a more detailed understanding of HTTP. There are, however, several well-written articles available on the Web which can help. I found a particularly good article on the topic at http://www.earthWeb.com. Search for the string 'Web Mail'.

Uploading Files to a Server

Like sending e-mail, uploading files is a more complex task than the general information presented in this article. After a brief search, though, I found a number of pre-packaged solutions on the Web available for download. Perhaps the best was the solution available at http://www.servlets.com. To use the solution, you'll need to download a JAR which abstracts some of the HTTP details as well as the source code example.

Performing Database Transactions

By default in JDBC, data changes are reflected immediately in the database. To show how this might cause problems, imagine a complex procedure which required thousands of database operations. If each change was committed immediately, what would happen if the software suddenly failed halfway through? The database could be left in an unknown state with no means for recovery. In this case, we would rather execute the operations in an "all-or-none" fashion. In JDBC, you can group database operations into transactions that are either executed in full or rolled back, leaving no changes.

Transaction Isolation Levels

The transaction isolation level is a property of the database connection, and once a transaction has begun, it specifies how sensitive the connection will be to data changes made by other processes. There are two different levels supported by Pervasive.SQL. TRANSACTION_READ_COMMITTED means that changes made to the database after a transaction has begun are visible to the connection. In other words, with this isolation level, your view of the database could be changing as you process your transaction. To maintain a static view of the data during a transaction, you should set the transaction isolation level to TRANSACTION_SERIALIZABLE.

Executing Transactions

By default, each database operation in Java is its own transaction. Thus, you start and finish a separate transaction each time you execute a statement. To extend your transactions to span multiple statement executions, you must set the autoCommit property of the database connection to *false* using Connection.setAutoCommit(). This will cause you to manually commit your changes to the database using the Connection.commit() method. You may throw your changes away at any time using the Connection.rollback() method. Below is a snippet of example code demonstrating transactions:

```
try {
 /* Create connection */
 conn = DriverManager.getConnection(url, "username", "password");

 /* Set to manually transaction handling */
 conn.setAutoCommit(false);

 /* Create JDBC statement and SQL batch */
 stmt = conn.createStatement();

 /* Add SQL statements to statement batch */
 stmt.addBatch("INSERT INTO Person VALUES(98423, 'Mark', 'Bolton')");
 stmt.addBatch("INSERT INTO Person VALUES(48932, 'Matt', 'Patel')");
 stmt.addBatch("INSERT INTO Person VALUES(43221, 'Yulia', 'Davis')");

 /* Execute SQL statements in batch */
 stmt.executeBatch();

 /* Commit changes */
 conn.commit();

 /* Clean up */
 stmt.close();
 conn.close();

} catch(BatchUpdateException b) {
 /* Process batch error */

} catch(SQLException ex) {
 /* Process SQL error */

}
```

PART V - ADVANCED DEPLOYMENT OPTIONS – DEPLOYING SERVLETS TO TOMCAT

Overview

This section will answer some of the questions involved with deploying servlets to the Tomcat environment. This is intended more to make you aware of the various options that you have than to give in-depth detail. The topics covered give the premises behind utilizing the configuration options, but the product documentation referenced will be the definitive source for step-by-step instructions.

Writing build.xml Files

The build.xml files used by Web applications in the Jakarta Tomcat environment can be very powerful. The following question-answer set should provide you with some helpful information.

What are targets?

Targets tell the build system, Ant, what type of build you would like to perform. Typical targets include compile, clean, and prepare. In the initial section of this document, we specified the build tool be run using solely the batch file named build.bat with no arguments. This runs the compile target by default. The targets allow you to start fresh and delete all of the files in the deployment directory by issuing the "build.bat clean" command. Defining targets for you own particular build needs can obviously be very powerful.

What are some typical operations?

The operations available through Ant, via the build.xml file, are numerous. The ones that are more commonly used consist of mkdir, copydir, deltree and javac (compiler). Use these operations within a specified target to customize your build process.

Where can I find a reusable example?

The file C:\tomcat\doc\appdev\build.xml.txt has a generic build.xml file that you can use as a starting point for new Web applications. To begin, you will only need to change the project name from *MyProject* to whatever your project is named. You may also look in the etc directories of the Webdev.zip file available for download from the Pervasive Software Component Zone located at http://www.pervasive.com/componentzone (in the Developer Articles and Tips section) for additional build.xml files.

Where can I get more information?

The Tomcat Web site, http://jakarta.apache.org/tomcat/, will always have the latest documentation on the currently supported build.xml syntax. In your local documentation, look at the user's guide for more detailed information.

Writing Web.xml Files

The Web.xml file is a servlet standard enforced by Sun and available at http://java.sun.com/products/servlet/. Tomcat uses a portion of that standard in allowing you to overwrite the default Web.xml context found in the Tomcat conf directory with the one in your individual application's directory. If there is no Web.xml, then the one from the conf directory is used.

Deploying Web Applications as Web ARchives (WARs)

What is a WAR?

A WAR is a Web Application Archive, used when deploying a Web application to a remote server. The contents of a WAR are the entire contents of your Web application. You need only provide a WAR to a system administrator to deploy your application to a remote site.

How Do I Create a WAR?

The Jakarta group has integrated the creation of WARs neatly into the build system. WARs are created with exactly the same format and tools and the JAR format. In the latest version of Tomcat, we had to add some

commands under the dist target to build WAR files. To see the build.xml file, refer to the Webdev.zip file available for download from the Pervasive Software Component Zone located at http://www.pervasive.com/componentzone (in the Developer Articles and Tips section).

After making these modifications, you should be able to run "build dist" to create WARs.

How Do I Deploy a WAR?

On the deployment server, place the WAR in the Webapps folder. That's it! Tomcat extracts the WARs when it is started, and creates the context for which to access them. This makes for easy deployment on remote machines.

PART VI - ADVANCED DEPLOYMENT OPTIONS – CONFIGURING TOMCAT

Overview

Advanced application and Web server configuration is outside the scope of this document. However, the following section should provide a framework for understanding what some of the additional configuration features are, as well as provide sources for more information.

Configuring Tomcat

In a robust, enterprise environment, Tomcat should not be configured to handle HTTP requests. Another Web server, such as Apache, should be installed and configured for the task. That being said, many of the Tomcat configuration options are Web server specific and will not be discussed.

server.xml

server.xml serves as the primary configuration file for the Tomcat server. In it, you'll find general configuration options which apply to the server as a whole. For those familiar with Apache, server.xml is Tomcat's version of httpd.conf.

Some simple items included in server.xml include:

1. HTTP listening port - port Tomcat Web server listens on
2. Logging Levels - How much output is written to the JSP and servlet log files
3. Contexts - Mappings between physical directories and server URLs

Web.xml

Web.xml defines the default properties that all Tomcat-deployed Web applications possess. These defaults may be overridden by the Web.xml files written for individual applications. There are a number of useful configuration options available in the Tomcat Web.xml file including:

1. Servlet URL pattern mapping - By default, any URL that includes /servlet/ will be mapped to the .class files located in the classes directory for that application. For example, a request to http://localhost/samples/servlet/HelloWorld/ would be mapped to the following file on my machine: c:\tomcat\Webapps\samples\WEB-INF\classes\HelloWorld.class.

2. JSP URL pattern mapping - By default, all URLs that end with .jsp are mapped to be handled by the Jasper (JSP) engine.
3. Session timeout length - Determines how long a user session may remain inactive before being terminated.
4. MIME mappings - Maps content types to Web browser plug-ins capable of servicing the requests.

Placing .class and .jar Files

In Tomcat, you may place class files and Java archives (JARs) used often in your applications in two special folders under the Tomcat root, lib and classes. Placing JARs in the lib directory causes them to be added to the class path automatically when Tomcat starts. Similarly, the entire classes directory under the Tomcat root is placed in the server's class path. Without these mechanisms, you would have to manually update the class path on the server every time your application called JAR or class file methods. A similar mechanism has been created for individual Web applications as well.

worker.properties

The worker.properties file defines a set of Tomcat processes capable of dividing the application server requests. This feature, often termed 'load-balancing', greatly expands the scalability of the Tomcat application server.

tomcat.policy

tomcat.policy defines the security settings for the application server as a whole. Additional security configurations may be specified on an application-by-application basis in the application's Web.xml file.

Additional coverage of all of these topics may be found in Tomcat's user manual located under the Tomcat docs directory.

Configuring mod_jk

mod_jk is a replacement for the JServ plug-in which offers greater functionality and flexibility in mapping requests between Web servers and Tomcat. Ideally, all HTTP requests will first be handled by a Web server and then forwarded to Tomcat if necessary. Again, mod_jk is the mechanism that handles the communication between Web servers and Tomcat.

A number of important items may be configured in mod_jk.conf-auto, including Web server-to-application server protocols, worker request mappings, etc. For more information about configuring mod_jk, see the mod_jk HOWTO in either the Tomcat docs directory or the Jakarta homepage, http://jakarta.apache.org/.

For more information about Pervasive.SQL 2000*i*, pricing or support, please contact one of our sales offices or visit our web site at http://www.pervasive.com .

Corporate Headquarters

Pervasive Software Ltd.
12365 Riata Trace Parkway, Building II
Austin, TX 78727
Phone: 800-287-4383 or 512-231-6000
Fax: 512-231-6010

International Offices

Pervasive Software Ltd.
Regus House
Highbridge, Oxford Road
Uxbridge
Middlesex, England UB8 1HR
Phone: +44-1895-876331
Fax: +44-1895-876331

Pervasive Software European
Service and Support
Bessenveldstraat 25A
B-1831 Diegem, Belgium
Phone: +32-2-710-1660
Fax: +32-2-718-0331

Pervasive Software SARL
Immueble Atria
21, Avenue Edouard Belin
F-92 500 Rueil Malmaison,
France
Phone: +33-1-55-47-17-00
Fax: +33-1-55-47-17-07

Pervasive Software GmbH
Frankfurter Strasse 151d
D-63303 Dreieich, Germany
Phone: +49-6103-9622-00
Fax: +49-6103-9622-05

Pervasive Software, N.V.
Airport Boulevard Office Park
Bessenveldstraat 25A
B-1831 Diegem, Belgium
Phone: +32-2-718-0330
Fax: +32-2-718-0331

Pervasive Software Co., Ltd.
World Trade Center, 33F
2-4-1 Hamamatsu-cho
Minato-ku, Tokyo 105-6124
Japan
Phone: +81-3-5405-2261
Fax: +81-3-5405-2269

http://www.pervasive.com
info@pervasive.com
salessupport@pervasive.com
http://www.pervasive.com/support/Email_Support.taf

Disclaimer: The performance demonstrated in this document is for reference purposes only and does not constitute a performance guarantee even if under similar configurations.