Connecting The Database With The Web Page By Using Java Database Connectivity (JDBC)

Wissam Ali Hussain
MSc. Computer Science Student
Sam Higginbottom Institute of technology & Science / India

Abstract

We can design internet pages for the companies, foundations and the government offices to spreading the information & details for the offices & foundations to facilitation connecting with it by using the internet in any time, and any place. When we design this pages the customer needs to circulate some information which his needed, and store it at a data base form, for example data base contains at the tools submitted to sales, or a data base contains at the information related of the employs for the specific foundation….ect. In this case we needs to connect the data base with the internet page, this research talked about connect the data base with the internet page by using the java data base connectivity (JDBC) technique.

الملخص :

من الممكن تصميم صفحات الانترنيت للشركات، المؤسسات والدوائر الحكومية من اجل نشر المعلومات والتفاصيل عن تلك الدوائر والمؤسسات لتسهيل الاتصال بها والتعامل معها باستخدام الشبكه العنكبوتية في أي زمان ومن أي مكان ، وعند تصميم تلك الصفحات سوف يحتاج الزبون الى تداول بعض المعلومات التي يحتاجها والتي يتم خزنها على شكل قواعد بيانات مثل قاعدة بيانات تحتوي على الادوات المعروضة للبيع في شركة معينة, او قاعدة بيانات موظفي مؤسسة معينة ..... الخ. في تلك الحالة سوف نحتاج الى ربط قاعدة البيانات تلك مع صفحة الانترنت. ان هذا البحث يتحدث عن ربط قاعدة البيانات مع صفحة الانترنت باستخدام تقنية Java Data Base Connectivity (JDBC) .

3.1 Introduction

There are many applications of the computer programs, such that we can use the word processors to write documents, Web browsers to explore the Internet, and email programs to send email. These are all examples of software that runs on computers. Software is developed by using programming languages. Such that there are many programming languages to developed these software, like Cobol, Fortran, Basic, Ada, C-language, C++ , Visual basic and Java language,….ect.

Each of these languages was designed for a specific purpose. COBOL was designed for business applications and is used primarily for business data processing. FORTRAN was designed for mathematical computations and is used mainly for numeric computations. BASIC was designed to be learned and used easily. Ada was developed for the Department of Defense and is used mainly in defense projects.

C combines the power of an assembly language with the ease of use and portability of a high-level language. Visual Basic and Delphi are used in developing graphical user interfaces and in rapid application development. C++ is popular for system software projects such as writing compilers and operating systems. The Microsoft Windows operating system was coded using C++. C# (pronounced C sharp) is a new language developed by Microsoft for developing applications based on the Microsoft .NET platform. Java developed by Sun Microsystems, is widely used for developing platform - independent Internet applications.

So the programmers is preference the java language at the others programming languages, because the java language is enables users to develop and deploy applications on the Internet for servers, desktop computers, and small hand-held devices. The future of computing is being profoundly influenced by the Internet, and Java promises to remain a big part of that future. Java is *the* Internet programming language. It is a powerful programming language and helpful to review computer basics, programs, and operating systems (1 , 2) .

3.2 History of Java

In 1991, a group of Sun Microsystems engineers led by James Gosling decided to develop a language for consumer devices (cable boxes, *etc.*). They wanted the language to be small and use efficient code since these devices do not have powerful CPUs. They also wanted the language to be hardware independent since different manufacturers would use different CPUs. The project was code-named *Green*.

These conditions led them to decide to compile the code to an intermediate machine-like code for an imaginary CPU called a *virtual machine*. (Actually, there is a real CPU that implements this virtual CPU now) This intermediate code (called *byte code*) is completely hardware independent. Programs are run by an interpreter that converts the byte code to the appropriate native machine code.

Thus, once the interpreter has been ported to a computer, it can run any byte coded program. Sun uses UNIX for their computers, so the developers based their new language on C++. They picked C++ and not C because they wanted the language to be *object-oriented*. The original name of the language was *(Oak)*. However, they soon discovered that there was already a programming language called Oak, so they changed the name to *Java*.(3)

The Green project had a lot of trouble getting others interested in Java for smart devices. It was not until they decided to shift gears and market Java as a language for web applications that interest in Java took off. Many of the advantages that Java has for smart devices are even bigger advantages on the web.

Currently, there are two versions of Java. The original version of Java is 1.0. At the moment (Nov. 1997), most browsers only support this version. The newer version is 1.1 (in addition 1.2 is in beta). Only *MS Internet Explorer 4.0* and Sun's *Hot Java* browsers currently support it. The biggest differences in the two versions are in the massive Java class libraries. Unfortunately, Java 1.1 applets will not run on web browsers that do not support 1.1. , However, it is still possible to create 1.0 applets with Java 1.1 development systems. (3)

3.3 Java Applications

There are two basic types of Java applications, they are:

3.3.1 Standalone

These run as a normal program on the computer. They may be a simple console application or a windowed application. These programs have the same capabilities of any program on the system. For example, they may read and write files. Just as for other languages, it is easily to write a Java console program than a windowed program. the place to start Java programming is a standalone console program.

3.3.2 Applets

These run inside a web browser. They must be windowed and have limited power. They run in a restricted JVM (Java Virtual Machine) called the *sandbox* from which file I/O and printing are impossible. (There are ways for applets to be given more power.)

3.4 Development Tools

There are many development tools for Java, they are:

A-Sun's JDK

Sun's Java Development Kit has two big advantages, the first advantage It is the most up to date. And the second advantage It is free. As well as the main disadvantage is that it only includes command line tools, no IDE.

B-Borland's J-Builder

It contains an IDE and supports Java 1.1.

C-MS Visual J++
It contains an IDE, but to my knowledge does not yet support Java 1.1.

D-Symantec's Visual Cafe
It contains an IDE, but no Java 1.1 support yet.(3)

3.5 Java Specification
The java language is a popular language , and the programmer accepted this language very quickly because it provides with everything that is needed in modern day language, such that it has the following specification:

A- It is object oriented programming.
B- It is multithreading.
C- Automatic memory management.
D- It has a networking and security features.
E- It is platform independence .
F- Has internet / web development features.(3)

3.6 Java Database Connectivity (JDBC)
The vast majority of professional web sites today have some sort of database connectivity. Webmasters have hooked online front ends to all manner of legacy systems, including package tracking and directory databases, as well as newer systems such as e-Commerce Systems. Although database-backed systems may be more challenging to develop, the advantages of allowing a database to manage data records are many fold. Within the database, data definition and manipulation is handled through *Structured Query Language (SQL)* as was covered in the previous section. Typically, the application tier (in this module anyway!) is implemented through the Java programming language. Now, since the database doesn't "speak Java" directly and Java isn't SQL, we need to form an interface to allow the two tiers to communicate. Such that the JDBC works under two models, the two-tier model and the three-tier model, it is shows in below.(5)

3.6.1 JDBC Two Tier Model
In this model, the application communicates directly with the database as shown in the figure (Figure F 3.1) in below.(5)

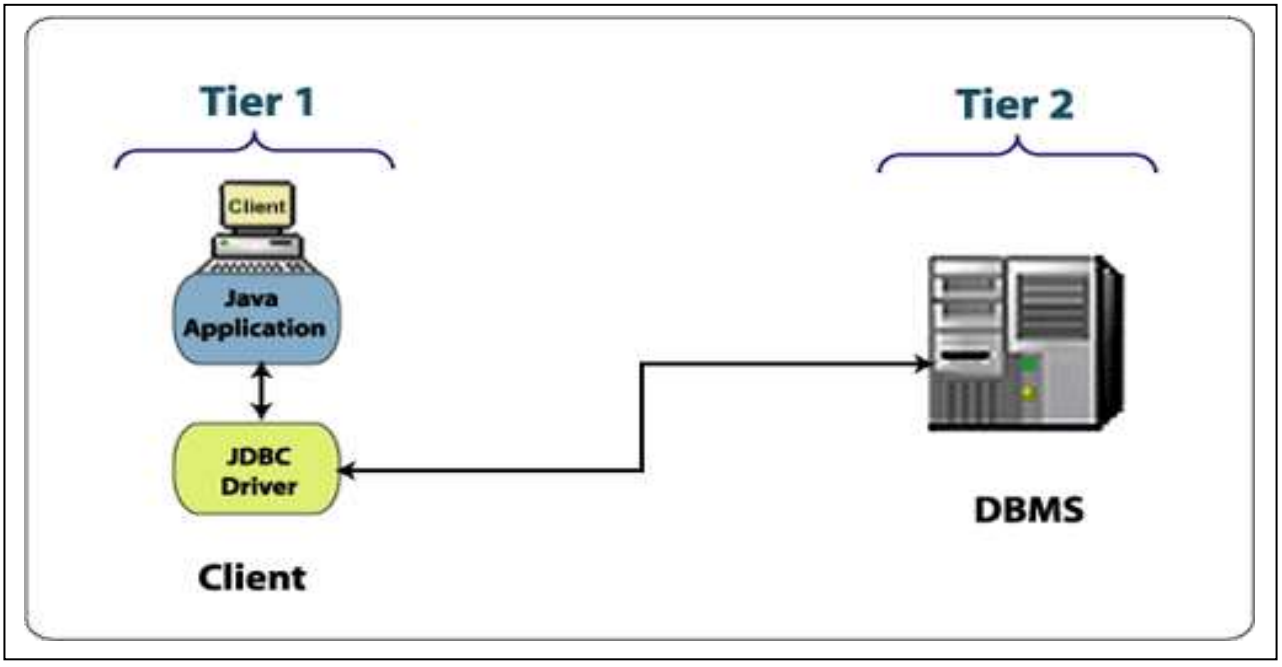


Figure (F 3.1) JDBC Two-Tier Model

The client consists of the application and the JDBC driver(s). The client takes the responsibility of presentation, business logic, transaction management and resource management. The JDBC driver also resides on the client. This driver receives the request from the application and performs the necessary transformation (i.e. transforms the request to vendor specific database calls). This transformed request is passed to the DBMS directly. Because the application and the driver reside on the same machine, the connection between the two is direct. The server is the DBMS, which receives the request from the client in a format it can understand.
The major disadvantage with this model is that there is a heavy burden on the client, as it takes care of both the presentation as well as the business logic (check back on the earlier sections of this course!). Also, the number of concurrent connections is limited because the connections are held by the application.(5)

3.6.2 JDBC Three Tier Model

This model consists of a client, a middle tier, and a server. The client holds the application front-end, which concentrates on the presentation and communicates with the middle tier requesting data. The middle tier takes care of the business logic, communicating with the data source, handling transactions and connection pooling. The middle tier typically consists of an application, an application server and the JDBC driver. The application takes care of the business logic. The application server manages transactions, handling concurrent connections and identifying the drivers. The JDBC driver provides the connectivity to the data source and implemented the JDBC API, sending the request in the form that the data source can understand. The third layer is the actual data source, which is typically a DBMS (but can be any information system with JDBC driver support, such as a spreadsheet). The advantages in using a three-tier model have already previously been discussed, and won't be repeated here. Essentially, this architecture offers advantages in scalability, usability, maintenance and performance. The following figure (F 3.2) shows JDBC Three Tier Model.(5)

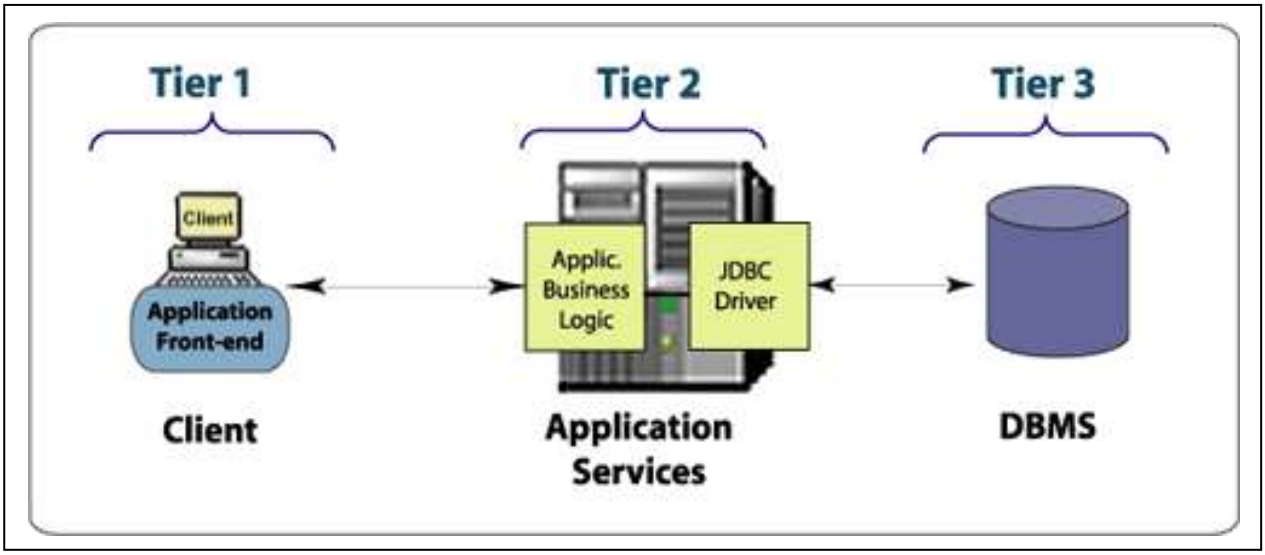


Figure (F 3.2) JDBC Three-Tier Model

In the late 90s, Sun Microsystems became known for working with vendors to define Java-based and vendor-friendly APIs for common services. Sun had a habit of adopting the best ideas in the industry and then making the Java implementation an open standard - usually successfully. The Java Database Connectivity API, called JDBC was a perfect example. An API, or ***Application Programming Interface*** is a set of classes, methods, and resources that programs can use to do their work. JDBC is a Java API for database connectivity, which is part of the overall Java API developed by Sun Microsystems.(5,6)
JDBC was based largely on Microsoft's ODBC (***Open Database Connectivity***) but has since largely surpassed it. Compared to ODBC, JDBC has more flexible APIs which programmers can use in

their applications. JDBC has all the advantages that ODBC has and caters to the needs of programmers under a variety of platforms. JDBC provides Java developers with an industry standard API for database-independent connectivity between Java Applications (Applets, Servlets, JSPs, EJBs etc.) and a wide range of relational database management systems such as Oracle, Informix, Microsoft SQL Server and Sybase. JDBC accomplishes most of what it does through a native API that translates Java methods to native calls. In its simplest form, JDBC makes it possible to do the following: (6)

- Connect to a database
- Execute SQL statements to query your database
- Generate query results
- Perform updates, inserts and deletions
- Execute Stored Procedures

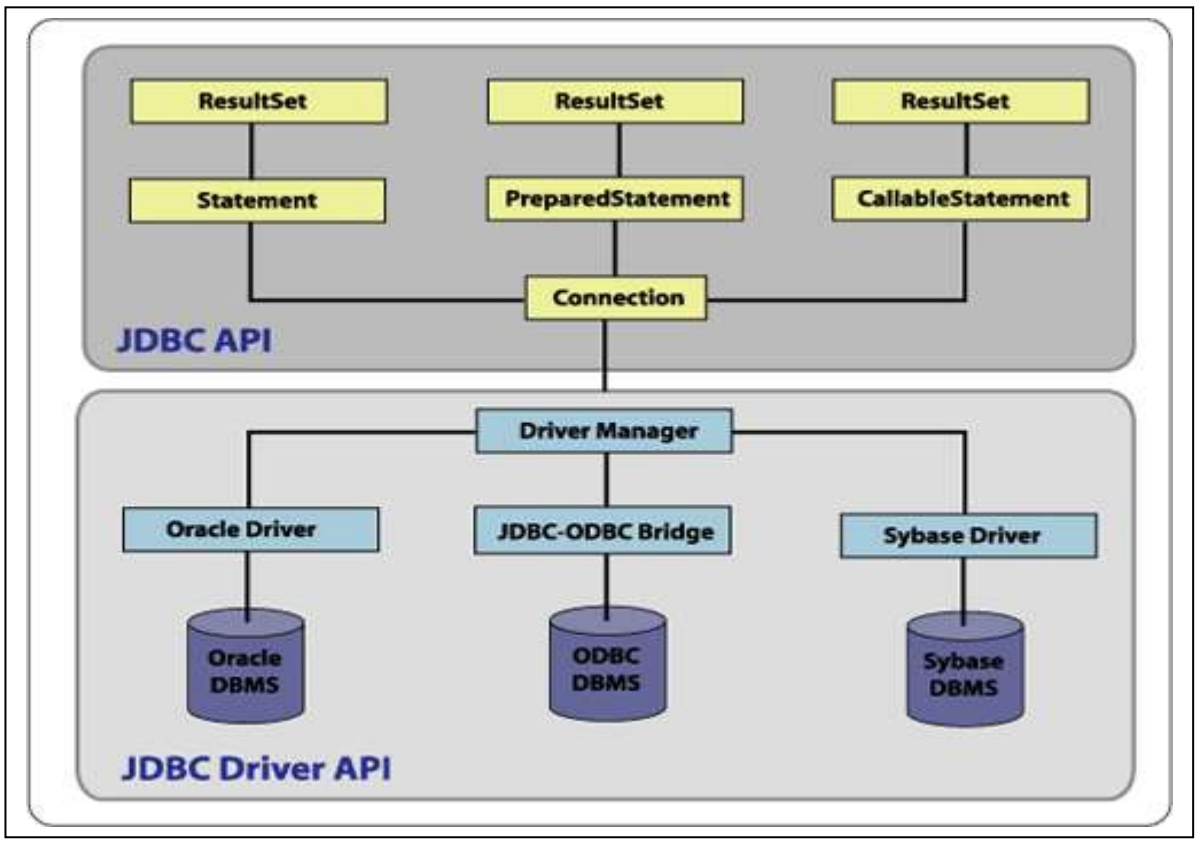
3.7 The benefits of using JDBC

There are six benefits of using JDBC, they are:

- A developer only needs to write one API to access any relational database.
- There is no need to rewrite code for different databases.
- There is no need to know the database vendor's specific APIs.
- It provides a standard API and is vendor independent.
- Almost every database vendor has some sort of JDBC driver.
- JDBC is part of the standard Java 2 platform.(4)

3.8 JDBC Architecture

The JDBC architecture consists of two layers: first, the JDBC API, which supports Java application-to-JDBC Driver Manager communications and secondly the JDBC Driver API, which is handles Driver Manager-to-Database communications. The figure (F 3.3)below shows the structure of the main interfaces and classes within JDBC and how JDBC programs interact with databases.(4)



As in ODBC, JDBC has an underlying driver manager that takes care of bridging the application with the DBMS. The driver manager supports multiple drivers connecting to many databases. The drivers themselves can be written either in Java or by using native methods. A driver written purely in Java can be used in any platform by downloading or being a part of an applet for example. A driver written using native methods gets tied up to the underlying platform in which the application runs or the applet runs.(4)

3.9 JDBC Driver

Individual databases are accessed via a specific JDBC driver that implements the ***java.sql.Driver*** interface. JDBC drivers are available for most database platforms and from a number of different vendors. There are four different driver types as follows:(4)

3.9.1 JDBC-ODBC Bridge Driver - These drivers show a JDBC face but call into an ODBC driver, which itself is normally implemented on top of a native driver written in C, C++ or another language. The actual communication with the database occurs through the ODBC drivers.(4)

3.9.2 Native-API Partly Java Driver - These drivers are implemented directly against the underlying native driver, eliminating the ODBC layer. Native drivers are still required. Typically, type 2 drivers wrap a thin layer of Java around database-specific native code libraries. As an example, with Oracle, the native code libraries are typically based on the OCI (Oracle Call Interface) libraries, which were originally designed for C/C++ programmers. Because Type 2 Drivers are implemented using native code, in some cases they can have better performance than their all-Java counterparts. (4)

3.9.3 Net-Protocol All-Java Driver - Type 3 drivers communicate via a generic network protocol to a piece of custom middleware. The client translates the database requests to a standard request format that is independent of any DBMS requests. The receiving middleware component parses this request and passes it to the specific DBMS. The middleware component might use any type of driver to provide the actual database access. With type 3 drivers, many clients can connect to many databases, and this type of driver is the most flexible one. These drivers are written entirely in Java.(4)

3.9.4 Native-Protocol/All-Java Driver - Purely Java based, type 4 drivers or *thin drivers* translate the database requests into a specific DBMS-understandable format. There is a direct call on the DBMS from the client and hence are no intervening layers. Since type 4 drivers are written completely in Java, they can run virtually unchanged on any platform.(4)

The following diagram (F3.4) shows the JDBC architecture options using the different driver types:

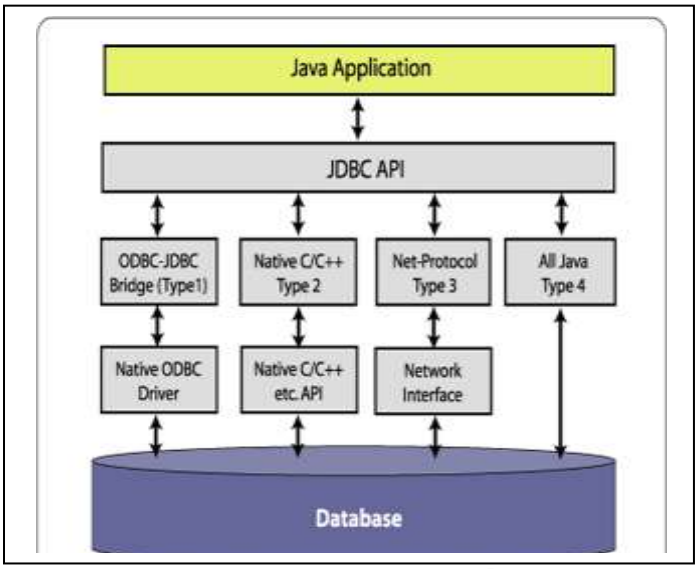


Figure (F3.4) JDBC Driver Architecture

Throughout the following instruction on how JDBC is implemented and the examples, we will encounter two driver types, namely Type 2 and Type 4.(4)

3.10 Connecting to the Database

The connecting operation to a database consists of the following steps:

1. ***Load the JDBC Driver*:** The driver class should be in the CLASSPATH environment variable when running an application, or within the relevant libraries of the application server. The specific driver class is loaded into the JVM for use later, when we go to open up a connection to the database. A class can be loaded by using the *Class object's forName() method*. When the driver is loaded into memory, it registers itself with the java.sql.DriverManager classes as an available database driver. Typically, the code is: (7)

   Class.forName (<driver class>);
   For example:
   Class.forName("oracle.jdbc.driver.OracleDriver");

2. ***Connect to the Database*** - by using the getConnection() method of the DriverManager object. The parameters to this method is the database's URL user name, and password. The database's URL contains the address of the database residing in the network and any other information such as the sub protocol and the port number the middle tier is listening to (if three-tier). JDBC URLs usually begin with (jdbc:subprotocol:subname). For example, the Oracle JDBC-Thin driver uses a URL of the form of (jdbc:oracle:thin:@database:port:sid). The JDBC-ODBC bridge uses jdbc:odbc:data-sourcename;odbcoptions). During the call to getConnection(), the DriverManager object asks each registered driver if it recognizes the URL. If a driver says yes, the driver manager uses that driver to create the Connection object. The basic construct for this stage is as follows: (7)

DriverManager.getConnection (<dburl>, userusername, password);
For example:
Connection con=null;
Con=DriverManager.getConnection("jdbc:oracle:thin:@136.206.35.131:1521:SSD", "EE_USER", "EE_PASS")

3. ***Perform Database Operations*** - the desired operations can then be executed, such as creating statements, executing statements and manipulating the ResultSet object.

All this can be done only when the connection is live (open). For example, a simple query is performed by creating a Statement object and using the executeQuery() method of Statement, and for updates etc. you should use executeUpdate().

JDBC supports three types of statements namely: *Statement* (used for executing an SQL Statement immediately), *PreparedStatement* (used for executing a compiled SQL statement) and *CallableStatement* (used for executing Stored Procedures). Examples will be provided covering the first two types of Statement, however since we did not cover Stored Procedures in detail, we will avoid more detail on the CallableStatement type. (7)

This method returns a *java.sql.ResultSet* which encapsulates the retrieved data. The ResultSet object can be seen as a representation of the query result returned one row at a time. Typically, a while loop is used to enumerate through the rows and return the data to the application. So a basic example for this stage is: (7)

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT SURNAME, FIRSTNAME FROM
EE_ADMIN.CUSTOMERS WHERE ID=2311");
while (rs.next())
{     System.out.println("Surname=" + rs.getString("SURNAME") +
            " and Firstname = " + rs.getString("FIRSTNAME")); }
```

Expected output:
Surname=Corcoran and Firstname=Sheila;

4. ***Release Resources*** **- Connections are costly**. When all necessary operations are complete, it is safe and advisable to close the Statements, ResultSets and Connections. Eg. (7)

```
if (rs != null) rs.close();
if (stmt != null) stmt.close();
if (con != null) con.close();
```

3.11 Servlets and Database

All database operations that can be performed on a stand-alone application can be performed on servlets too. Such that all the database operations are performed on the server side, and only data is passed to the client. The client is provided with no information regarding where the information has come from (ie. database details, username/password, ……. etc.). In fact, any type of Java server application (not just servlets) can connect to a database using JDBC. (3)

3.12 Servlets Connection Pooling

An application using databases as storage for information requires a lot of communication among the database engines to retrieve the information. When multiple clients access the same Web Application, the burden falls on the server to satisfy the entire client request and to dispatch the requested information. Naturally enough, the server should be efficient and powerful enough to both handle these requests and to handle the connection between itself and the DBMS. However, connecting to a database is a time-consuming activity since the database connection requires resources (communication, memory, authentication, and so on) to create each connection, maintain it and then to release it when it is no longer required. The overhead is particularly high for Web-based applications because Web users connect and disconnect more frequently. Establishing a connection once and then reusing it for subsequent requests can dramatically improve performance and response times of Web-based applications. However, allowing clients to hold on to connections can limit the number of connections which can be made to your Web Applications. For example, the default number of Connections which Oracle handles is 50 (although this can be greatly increased).there are two types of the servlets connection pooling, the first type is Non-Pooled Architecture, it is shown below. (3)

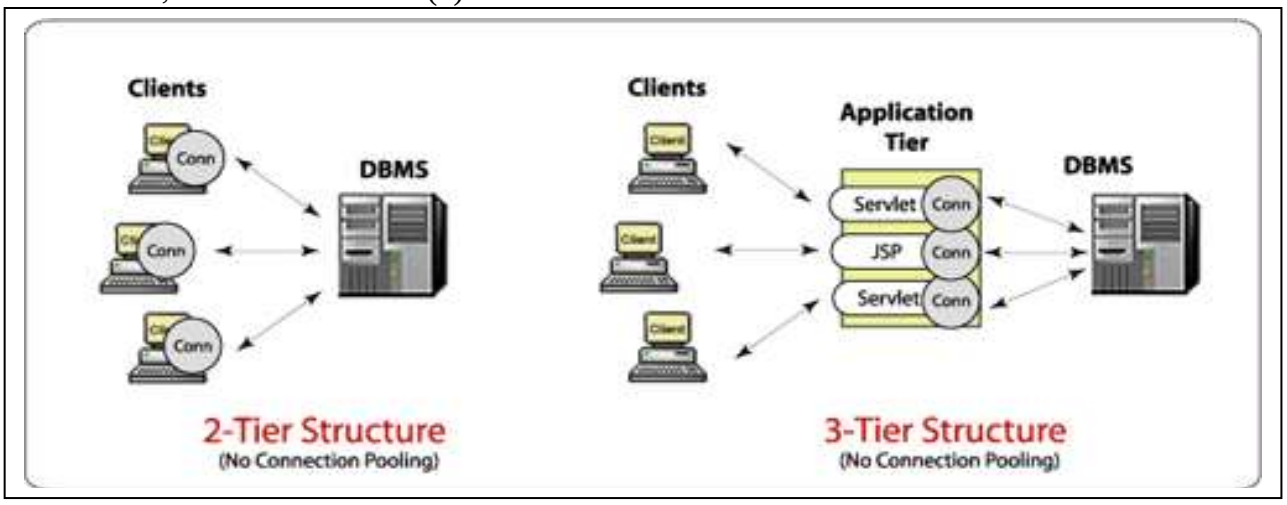


Figure (F3.5) Non-Pooled Architecture

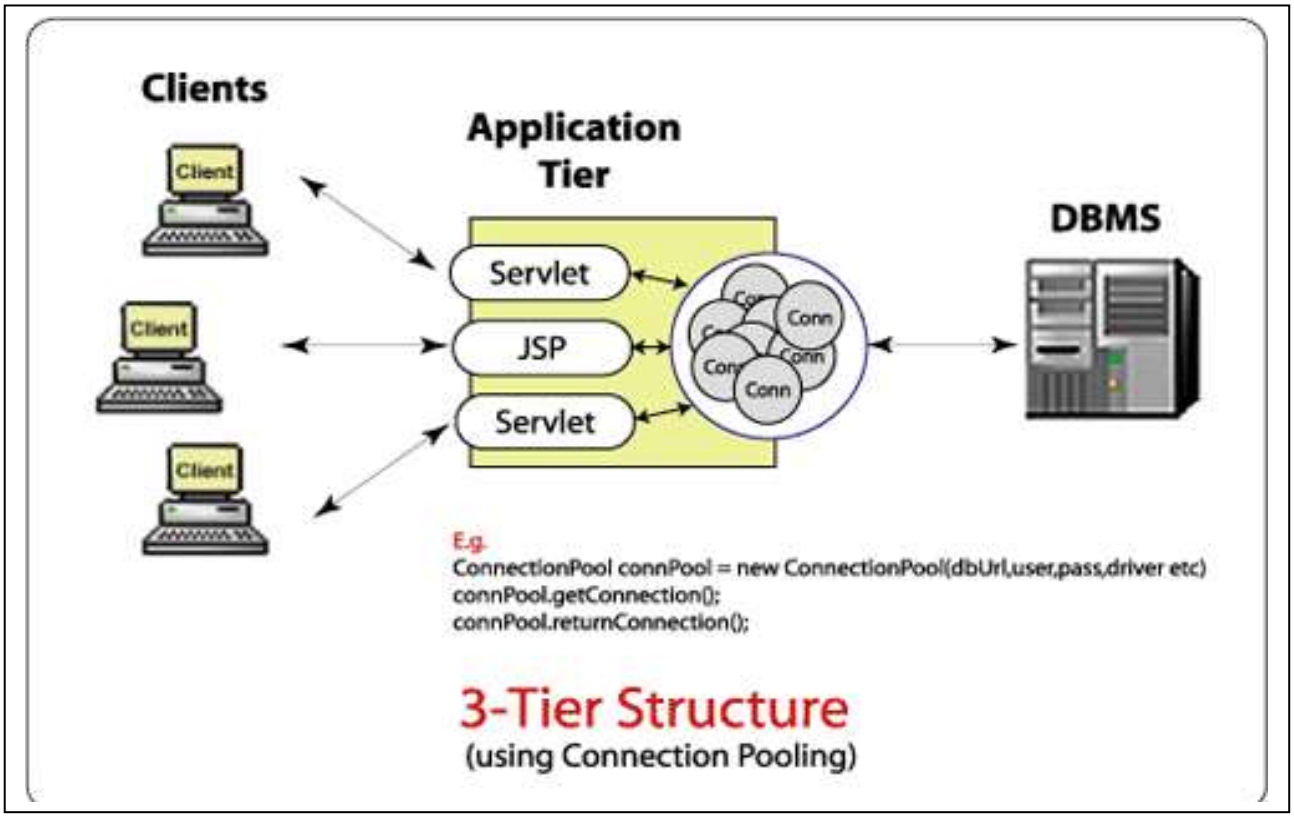
In this example, two connections are made, with a simple select statement from a small table (same one in each case). The first connection is made through standard JDBC techniques of obtaining a Connection through a JDBC driver, creating and executing a Statement. A typical output from this part of the servlet is as follows: (3)

```
JDBC Driver Version is: 8.1.7.1.0   Data = molloyda
Database Query/Connection Total Time = 20 ms
Database Connection Time Only = 16 ms
```

It can be seen how the database connection time was responsible for 16ms of the 20ms total transaction duration. The Statement and processing of the ResultSet accounted for 4ms. It should be noted, that if the query was more complex, using large tables, joins or complicated views this could be considerably larger, possibly even exceeding the connection time. However, the 16ms connection time is unnecessary with a readily pooled set of connections. The second part of the output shows the overhead in the situation where a Connection Pool is used:(3)

```
Connection using thin drivers (with connection pooling!).....
JDBC Driver Version is: 8.1.7.1.0   Data = molloyda
Database Connection/Query Time = 4 ms
Database Connection Time Only = 0 ms
```

Use of the connection pool in this circumstance results in a reduction of 75% of the entire duration. In fact the connection time is essentially reduced to 0ms, as the Connection is already waiting to be used. The most obvious benefit of Connection Pooling is in quicker response times to the end user, particularly from less powerful servers and remote databases. The 16ms in this case is a relatively powerful machine running a local database - if the database were remote the connection time could be considerably larger, and more noticable to the client. Another major benefit applies to the resources on the server and in particular connectivity with the database. If a database could simultaneously handle up to a maximum of 100 connections and assuming that all transactions were identical to that above we would have connection pooling, it is shown in below: (3)



(F3.6) connection pooling Architecture

100 * 1000/20 = 5,000 connections per second maximum (non pooled)

or

100 * 1000/4 = 20,000 connections per second maximum (connection pooled)

The obvious benefits in resources is immediately apparent. While these values both seem somewhat high, when you consider that connections in practice may take considerably longer and multiple connections per transaction may be made, the database could rapidly become the bottleneck in your Web Application.

A number of third-party connection-pooling packages are available, such as Db Connection Broker from Java Exchange. Web Logic also implement a JDBC driver that handles a pool of connections to another JDBC driver. Connection pooling takes processing and maintenance time, but it is typically worth the time lost in establishing a new connection when needed.(3)

Most decent Application Servers establish a pool of database connections that is shared by all the applications. *Connection Pools* can intelligently manage the size of the pool and make sure each connection remains valid. (3)

3.13 What is Servlets

The Servlets are protocol and platform independent server-side software components, written in Java. They run inside a Java enabled server or application server, such as the Web Sphere Application Server. Servlets are loaded and executed within the Java Virtual Machine (JVM) of the Web server or application server, in much the same way that applets are loaded and executed within the JVM of the Web client. Since servlets run inside the servers, however, they do not need a graphical user interface (GUI). In this sense, servlets are also faceless objects.

Servlets more closely resemble Common Gateway Interface (CGI) scripts or programs than applets in terms of functionality.As in CGI programs, the servlets can respond to user events from an HTML request,and then dynamically construct an HTML response that is sent back to the client. (3)

3.14 Servlet process flow

Servlets implement a common request/response paradigm for the handling of the messaging between the client and the server. The Java Servlet API defines a standard interface for the handling of these request and response messages between the client and server.

Figure (F 3.7) shows a high-level client-to-servlet process flow:

A - The client sends a request to the server.

B - The server sends the request information to the servlet.

C- The servlet builds a response and passes it to the server. That response is dynamically built, and the content of the response usually depends on the client's request. External resources may also be used.

D- The server sends the response back to the client. (3)

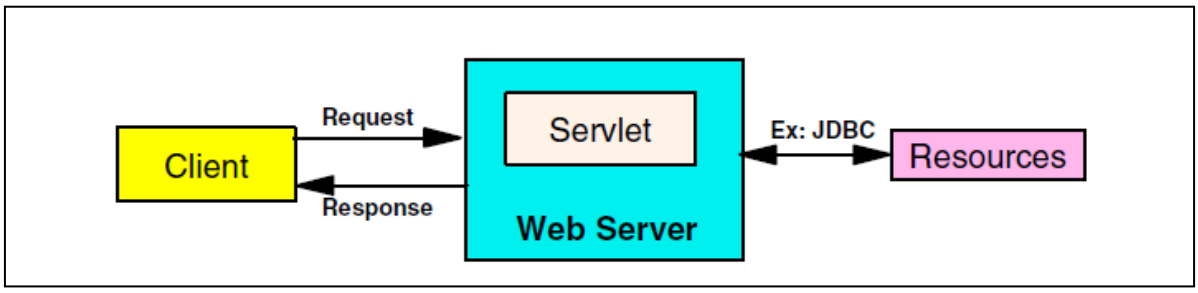


Figure (F 3.7) high-level client-to-servlet process

Servlets are powerful tools for implementing complex business application logic. Written in Java, servlets have access to the full set of Java API's, such as JDBC for accessing enterprise databases.
As mentioned above, servlets are similar to CGI in that they can produce dynamic Web content. Servlets, however, have the following advantages over traditional CGI programs:

A- ***Portability and platform independence:*** Servlets are written in Java, making them portable across platforms and across different Web servers, because the Java Servlet API defines a standard interface between a servlet and a Web server.

B- ***Persistence and performance:*** A servlet is loaded once by a Web server, and invoked for each client request. This means that the servlet can maintain system resources, like a database connection, between requests. Servlets don't incur the overhead of instantiating a new servlet with each request. CGI processes typically must be loaded with each invocation.

C- ***Java based:*** Because servlets are written in Java, they inherit all the benefits of the Java language, including a strong typed system, object-orientation, and modularity, to name a few. (3)

3.15 The Java Servlet API

The Java Servlet API is a set of Java classes which define a standard interface between a Web client and a Web servlet. Client requests are made to the Web server, which then invokes the servlet to service the request through this interface.
The Java Servlet API is a Standard Java Extension API, meaning that it is not part of the core Java framework, but rather, is available as an add-on set of packages. The API is composed of two packages:

A- *javax.servlet* .

B- *javax.servlet.http* .

The *javax.servlet* package contains classes to support generic protocol-independent servlets. This means that servlets can be used for many protocols, for example, HTTP and FTP. The *javax.servlet.http* package extends the functionality of the base package to include specific support for the HTTP protocol. we will concentrate on the classes in the javax.servlet.http package.

The *Servlet* interface class is the central abstraction of the Java Servlet API. This class defines the methods which servlets must implement, including a *service()* method for the handling of requests. The *GenericServlet* class implements this interface, and defines a generic, protocol-independent servlet. To write an HTTP servlet for use on the Web, we will use an even more specialized class of *Generic Servlet* called *Http Servlet.*
Http Servlet provides additional methods for the processing of HTTP requests such as GET (*doGet* method) and POST (*doPost* method). Although our servlets may implement a *service* method, in most cases we will implement the HTTP specific request handling methods of *doGet* and *doPost*.(7,8)

3.16 The servlet life cycle

A client of a servlet-based application does not usually communicate directly with a servlet, but requests the servlet's services through a Web server or application server that invokes the servlet through the Java Servlet API. The server's role is to manage the loading and initialization of the servlet, the servicing of the request, and the unloading or destroying of the servlet. This is generally provided by a servlet manager function of the application server.
Typically, there is one instance of a particular servlet object at a time in the Web servers' environment. This is the underlying principle to the persistence of the servlet. The Web server is responsible for handling the initialization of this servlet when the servlet is first loaded into the environment, where it remains active (or persistent) for the life of the servlet.
Each client request to the servlet is handled via a new thread against the original instance object. The Web server is responsible for creating the new threads to handle the requests. The Web server is also responsible for the unloading or reloading of the servlets. This might happen when the Web application is brought down, or the underlying class file for the servlet changes, depending on the underlying implementation of the server.
The following Figure (F 3.6) shows a basic client-to-servlet interaction such that Servlet1 is initially loaded by the Web application server. Instance variables are initialized, and remain active (persistent) for the life of the servlet.
The two Web browser clients have requested the services of Servlet1. A handler thread is spawned by the server to handle each request. Each thread has access to the originally loaded instance variables that were initialized when the servlet was loaded.
Each thread handles its own requests, and responses are sent back to the calling client.(4)

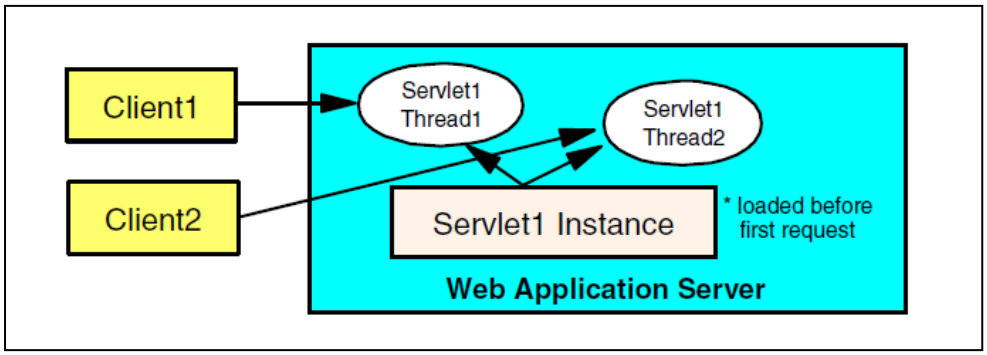


Figure (F 3.8) basic client-to-servlet interaction

References:

1- Principle of programming language, Mourise Mano, fourth edition.
2- Object oriented programming system with java, Lalit Arora, fourth edition.
3- The complete reference of java, Herbert Schildt, eight edition.
4- Introduction to the java language, Herbert Schildt, seventh edition.
5- Data base concepts, Abraham Siberschats, sixth edition.
6- Oracle data base administer, oracle university.
7- Web application development by using HTML & Java Script, Ivan Byross, fourth edition.
8- The complete reference of HTML & CSS, Thomas A. Powell, fifth edition.