Chapter

8

SECTION II: JAVA SERVLETS

Working With Databases

A DBMS forms an integral part of any application. It is hard to find a web or enterprise application today that does not have some sort of database connectivity. A lot of applications developed in Java EE environment are dependent on a database that stores information that the application uses. For example**:**

- Search engine use databases to store information, which they have extracted from web pages
- Websites that earn their cash flows from E-commerce use databases that store information about their products, customers and orders
- Geo-imaging sites provide those who access them, photographic images of the world from space, use databases to store these images captured by a satellite using high definition cameras

Working with database using Java is very simple as Java supports various database systems. Java has an API called JDBC API which allows working with databases. The JDBC API is industrially accepted for database-independent connectivity between the Java programming language and a wide variety of databases and other tabular data sources.

JDBC API technology defines the **Write Once, Run Anywhere** paradigm for applications that require access to enterprise data.

What Is JDBC?

In 1996, JavaSoft, released its first version of the JDBC kit. JDBC stands for **J**ava **D**ata**B**ase **C**onnectivity. This is actually an API, which consists of a set of Java classes, interfaces and exceptions designed to perform actions against any database.

Applications developed with Java and JDBC are platform and database vendor independent i.e. the same Java program can run on a PC, a workstation or a network computer and can connect to any vendor's DBMS simply by changing the JDBC middleware.

JDBC is today a mature and well-accepted table data, access and standard.

JDBC Drivers

Applications written using the JDBC API communicate with a JDBC driver manager, which uses driver specifically loaded to communicate with the DB engine.

RDBMS [**R**elational **D**ata**B**ase **M**anagement **S**ystems] or third-party vendors develop database specific drivers. Application developers use these drivers in their applications, to access appropriate database tables.

REMINDER

This book focuses on MySQL as the database of choice hence MySQL Connector/J will be used as the JDBC driver to access the MySQL database.

The latest release of JDBC technology i.e. JDBC 4.0 API provides access to non-database tabular sources of data such as spreadsheets and flat files.

Developers use these drivers to develop applications, which access the respective databases. The JDBC application developers can easily replace one driver for their application with another better one without having to alter the application as the drivers are adhered to JDBC specification. If some proprietary API provided by some RDBMS vendor is used for developing Java applications, it becomes mandatory to modify a substantial amount of application in order to switch to other driver and/or database.

With JDBC in place the developers can develop Java data access applications without having to learn and use proprietary APIs provided by different RDBMS. The developer only needs to learn JDBC and then data access applications can be developed conveniently that can access different RDBMS and/or using different JDBC drivers.

Although relational databases are most common databases, JDBC can be used with any kind of database. The main reason behind this is JDBC concepts common database functions into a set of common classes and methods. Database specific code is contained in a code library, which is called as a driver library. With the driver library for a database, JDBC API can be used to send commands to the database and extract data from the database.

Types Of JDBC Drivers

Connection with an application to a database server using a database driver can be done in the following four ways**:**

JDBC Type 1 - JDBC-ODBC Driver

Type 1 drivers act as a **bridge** between JDBC and the ODBC database connectivity mechanism.

The JDBC-ODBC bridge uses standard ODBC drivers to provide JDBC access to database tables. The JDBC-ODBC bridge requires that native ODBC libraries, drivers and their required support files be installed and configured on each client machine which employs this driver.

This driver delegates the work of data access to ODBC API. They are the slowest database access API of all, due to the multiple levels of translation that have to occur.

JDBC Type 2 - Java Native Interface Driver

They mainly use the **J**ava **N**ative **I**nterface [JNI] to translate calls to the local database API.

They also provide Java wrapper classes that are invoked using JDBC drivers.

Type 2 drivers are usually faster than the Type 1 drivers. Like Type 1 drivers, Type 2 drivers also require native database client libraries to be installed and configured on all client machines.

JDBC Type 3 - Java Network Protocol Driver

They are written in pure Java and use a vendor independent network protocol to communicate with JDBC middleware placed on a server on the network. Middleware placed on the server then translates the network database requests to database specific function calls.

Type 3 drivers are a more flexible JDBC solution as they do not require any native database libraries on the client and can connect to many different databases on a server placed on a network.

Type 3 drivers do not require any installation on the client side for deployment purpose over the Internet. Type 3 drivers offer an application the ability to transparently access different types of databases.

JDBC Type 4 - Java Database Protocol Driver

They are also written in pure Java and implement a database protocol such as Oracle's SQL*NET, to communicate directly with the Oracle. They are the most efficient among all driver types and are most commonly used.

SUN encourages developing and using type 4 drivers in Java for data access applications.

Type 4 drivers do not require any native database libraries to be loaded on the client and can be deployed over the Internet.

Unlike Type 3 drivers, if the backend database changes, a new Type driver must be purchased and deployed. However, as Type 4 drivers communicate directly with the database engine and do not use middleware or a native library, they are the fastest JDBC drivers available.

One drawback of Type 4 drivers is that they are database specific.

Type 4 drivers usually exhibit the best data access performance, being bound to a specifc database.

Which Driver To Choose From?

A question arises as to which is the right type of driver to be used for an application?

Among all the above listed drivers, selection of the right type of driver depends on the requirement of a specific project.

In **terms of cost** if Type 3 or Type 4 drivers are expensive then Type 1 and Type 2 drivers are most suitable as they are usually available free but if the mechanisms for installing and configuring software on each client machine is not available then Type 1 and Type 2 drivers can be opted out.

If price is not a matter, then there is an issue of which driver i.e. the Type 3 driver or the Type 4 driver to be used. So, evaluate the **benefits of the flexibility and the interoperability against each driver's performance**. Type 3 drivers usually offers an application the ability to clearly access different types of databases, where as Type 4 drivers usually excel in performance if bound to a specific database.

Advantages Of JDBC

The following are the advantages of JDBC:

- While using JDBC, businesses are not limited to any proprietary architecture and can proceed to use their installed databases and to access information, which may be stored on the same or different DBMS
- The combination of Java API and the JDBC API results into the easy and economically more efficient application development. JDBC overcomes with complexity of many data access tasks. JDBC API is simple to learn, easy to deploy and inexpensive to maintain as it simplifies enterprise development
- With the JDBC API, no configuration is required on the client side. It is a JDBC URL or a DataSource object registered with a **J**ava **N**aming and **D**irectory **I**nterface [JNDI] naming service which defines all the information required to establish a connection with a driver written in Java programming language
- JDBC API is available everywhere on the Java EE platform, which provides support for **Write Once**, **Run Anywhere** paradigm as the developers can actually write database application once and access data anywhere
- The JDBC API facilitates the development of sophisticated applications by providing metadata access. This sophisticated applications are needed to recognize essential facilities and capabilities of a database connection

JDBC Architecture

JDBC is an API specification developed by Sun Microsystems, which defines a uniform set of rule using an interface for accessing different relational databases. JDBC forms a part of the core of the Java platform. It is included in the Java SDK distribution.

The singular purpose of the JDBC API is to provide a resource to developers through which they can issue SQL statements and process their results in a consistent, database-independent manner.

JDBC defines classes and interfaces, which provides rich, object-oriented access to databases and represent objects such as**:**

- Database connections
- SQL statements
- Result sets
- Database metadata
- Prepared statements
- **B**inary **L**arge **Ob**ject**s** [BLOBS]
- **C**haracter **L**arge **Ob**ject**s** [CLOBS]
- Callable statements
- Database drivers
- Driver manager

The JDBC API makes use of a driver manager, which is bound to database specific drivers. This combination provides consistent, transparent, Db connectivity to all databases.

The JDBC driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases and also verifies whether the driver used for accessing each data source is correct or not.

A JDBC driver translates standard JDBC calls into a call native to a database, which enables an application module to communicate with the database. It is this interpretation layer [mapped to the appropriate driver] that provides database independent JDBC applications. If the backend, database changes, then very little code modification is required along with the replacement of the JDBC driver with the new Db driver.

The JDBC API is available in two packages namely**:**

- java**.**sql Package**:** This API is a core API that is compatible with any driver that uses JDBC technology
- javax**.**sql Package**:** This package is an Optional Package API, which extends the functionality of the JDBC API from a client-side API to a server-side API. It provides scrollable result sets and cursor support

The following are important JDBC classes, interfaces and exceptions in the java.sql package:

- **Driver:** Driver interface gives JDBC a launching point for database connectivity by responding to DrvierManager connection requests and providing information about the implementation in question
- **DriverManager:** DriverManager class actually keeps a list of classes that implement the Driver interface. When an application is run, DriverManager loads all the drivers found in the memory. When opening a connection to a database DriverManager selects the most appropriate driver from the previously loaded drivers
- **Connection:** Connection interface represents a connection with a data source. This interface can be used to retrieve information regarding the tables in the database to which connection is opened
- **Statement:** Statement interface represents static SQL statement that can be used to retrieve ResultSet object(s). The objective of Statement interface is to pass to the database the SQL command for execution and to retrieve output results from the database in the form of a ResultSet

REMINDER

Only one ResultSet can be open per statement at a time.

- **ResultSet:** ResultSet is a database result set generated from a currently executed SQL statement. The data from the query is delivered in the form of a table. The rows of the table are returned to the program in sequence
- **RowSet:** RowSet object extends ResultSet interface to add support for disconnected result sets and thereby helps in retrieval of data completely
- **PreparedStatement:** PreparedStatement object is an SQL statement that is pre-compiled and stored. This object can then be executed multiple times much more efficiently than preparing and issuing the same statement each time it is needed. Therefore, it is a higher performance alternative to Statement object
- **CallableStatement:** CallableStatement represents a stored procedure. It can be used to execute stored procedures in a RDBMS that supports them
- **DataSource:** DataSource object abstracts a data source. This object can be used in place of DriverManager to efficiently obtain data source connections

Accessing Database

To access and work with a database using JDBC, the following are the steps involved:

- Configuring JDBC Driver
- Creating A Database Connection

- Executing Queries
- Processing The Results
- Closing The Database Connection

Configuring JDBC Driver

The first step to establish a database connection using a JDBC driver involves loading the specific driver class into the application's JVM. This makes the driver available later, when required for opening the connection.

Class.forName(String**).newInstance()** is used to load the JDBC driver class**:**

```
Class.forName("com.mysql.jdbc.Driver").newInstance();
```

The above code spec indicates that the JDBC driver from some JDBC vendor has to be loaded into the application.

Class.forName() is a **static** method. This instructs the JVM to dynamically locate, load and link the class specified to it as a parameter. **newInstance()** indicates that a new instance of the current class should be created.

When the driver class is loaded into memory, it creates an instance of itself and registers with java**.**sql**.**DriverManager class as an available database driver.

Creating A Database Connection

Once the driver is loaded, a database connection needs to be established. A database URL identifies a database connection and notifies the driver manager about which driver and data source is used.

Syntax: [For Database URL]

```
jdbc:<SubProtocol>:<SubName>
```

Here,

- **jdbc** indicates that JDBC is being used to establish the database connection
- **SubProtocol** is the name of the database the developer wants to connect to. Example**:** mysql, oracle, odbc and so on
- **SubName** is typically a logical name or alias, which provides additional information on how and where to connect

HINT

Most database URLs closely follow standard syntax. However, the JDBC database URL conventions are very flexible. They allow each driver to define the information that should be included in its URL.

The following list represents the syntax for three common JDBC database URLs. Each type of database driver requires different information within its URL:

JDBC	Database URL	Driver Used
MySQL	jdbc:mysql://Server[:Port]/Database_Name	MySQL Connector/J JDBC Driver
Oracle	jdbc:oracle:thin:@Server:Port:Instance_Name	Oracle Type 4 JDBC Driver
ODBC	jdbc:odbc:Data_Source_Name	JDBC-ODBC Bridge Driver

To create a database connection, the JDBC connection method **getConnection()** of DriverManager is used as follows:

```
Connection con = DriverManager.getConnection("jdbc:mysql://localhost/test ", "user", "passwd");
```

The method is passed a specially formatted URL that specifies the database. The URL used is dependent upon the JDBC driver implemented. It always begins with **jdbc:** protocol, but the rest depends upon the particular vendor. It returns a class that implements java.sql.Connection interface.

Within the getConnection() method, DriverManager queries each registered driver until it locates one that recognizes the specified database URL. Once the correct driver is located, DriverManager uses it to create Connection object. While using JDBC, it is required to import **java.sql** package as the driver manager, the connection objects and the other JDBC objects are contained in this package.

REMINDER

To improve the performance of JDBC, define the database connection as an instance variable then open the Db connection within the Servlet's **init()** method. Then the database connection will be established only once i.e. when the Servlet is first loaded and will be shared across all Db requests thereafter.

Executing Queries

After establishing database connection there should be some way to execute queries. There are three ways of executing a query:

- ❑ Standard Statement
- ❑ Prepared Statement

- Callable Statement

Standard Statement

The simplest way to execute a query is to use java.sql.Statement class. To obtain a new Statement object createStatement() of Connection object is used which is written as follows:

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

Statement objects are never instantiated directly.

A query that returns data can be executed using the **executeQuery()** method of Statement. This method executes the statement and returns java.sql.ResultSet class that encapsulates the retrieved data. The following code spec defines ResultSet object that encapsulates the retrieved data:

```
ResultSet rs = stmt.executeQuery("SELECT * FROM Books");
```

HINT

For inserts, updates or deletes, use the executeUpdate() method. The executeUpdate() method accepts an SQL statement that contains user instructions to insert, update or delete table data.

Prepared Statement

A Prepared Statement is used for an SQL statement, which must be executed multiple times. When a prepared statement is created, the SQL statement is sent to the database for pre-compilation [if this is supported by the JDBC driver].

As they are precompiled, Prepared Statement executes much faster than standard SQL statements.

Syntax:

```
PreparedStatement pstmt = con.prepareStatement("INSERT INTO Books(BookNo, Name)
VALUES(?, ?)");
```

HINT

The question marks in the PreparedStatement syntax represent dynamic query parameters. These parameters **can be changed** each time the prepared statement is called.

Callable Statement

Callable Statement is used to execute SQL stored procedures. Methods are provided to specify input parameters and retrieve return values.

Callable Statement object extends PreparedStatement and therefore, inherits its methods.

Syntax:

```
CallableStatement cstmt = con.prepareCall("{call getBooks(?, ?)}");
```

Processing The Results

ResultSet object is a cursor [i.e. a specific place in memory], which holds SQL query output. **next()**, a method that belongs to ResultSet object, can be used to navigate across data rows in the opened cursor one row at a time, starting from the topmost row. ResultSet object also has many other methods for retrieving data from the current row. The getString() method and the getObject() method are among the most frequently used for retrieving specific column values from a data row.

Syntax:

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

REMINDER

ResultSet is linked to its parent Statement. Therefore, if a Statement is closed or used to execute another query, any related ResultSet objects are closed automatically.

When a call is made to the getConnection() method, DriverManager object queries each registered driver, if it recognizes the URL. If a driver agrees, the driver manager uses that driver to create Connection object.

Closing The Database Connection

Since database connections are a valuable application resource and consume a lot of system resources to create, the DB connection should be closed only when all table data processing is complete. Connection object has a built-in method, the **close()** methodfor this purpose.

In addition to closing the database connection, application code spec, should explicitly close all Statement and ResultSet objects using their **close()** methods.

While it is true that the JVM's built-in garbage collection processes will eventually release resources that are no longer active, it is always a good practice to manually release these resources as soon as they are no longer useful.

Syntax:

```
rs.close();
stmt.close();
conn.close();
```

The Customers GUI Example

This chapter demonstrates JDBC using the MySQL Database 5.1.44. Ensure that MySQL database is downloaded and installed on the machine prior running the example.

MySQL provides connectivity to client applications developed in the Java EE 6 using a JDBC driver named **MySQL Connector/J**.

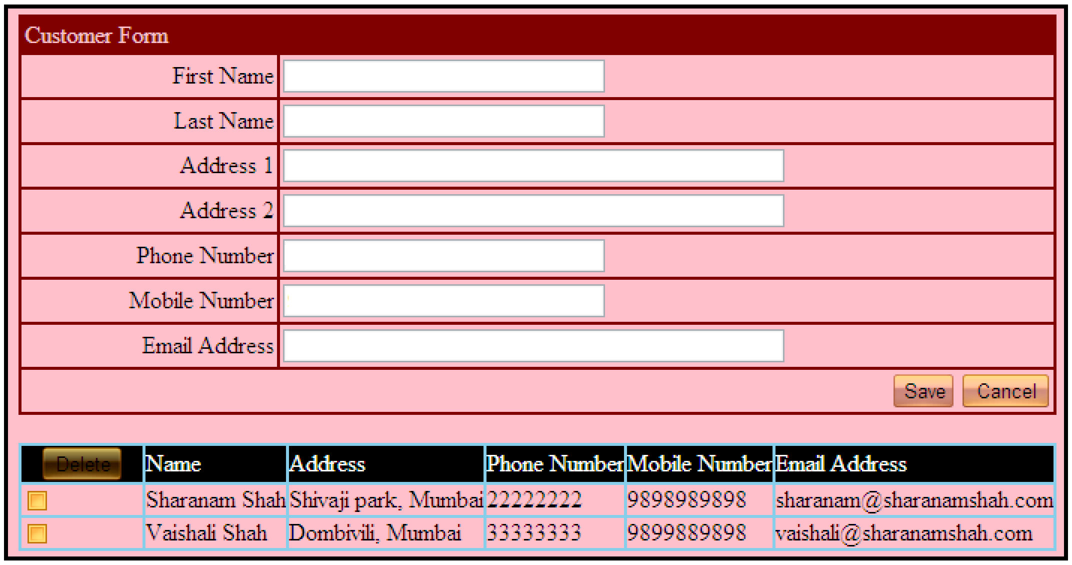
MySQL Connector/J is a native Java driver that converts JDBC calls into the network protocol used by the MySQL database. MySQL Connector/J is a Type 4 driver, which means that MySQL Connector is pure Java code spec and communicates directly with the MySQL server using the MySQL protocol.

MySQL Connector/J allows the developers working with Java EE 6, to build applications, which interact with MySQL and connect all corporate data even in a heterogeneous environment.

Download the MySQL Connector/J JDBC Driver from the website http://www.mysql.com. At the time of writing this book, the latest version was MySQL Connector/J 5.1.10 [Available in the Book CDROM].

The following example is the Customers G.U.I, which interacts with data stored in the Customers table under the MySQL Db engine.

After the form is crafted it appears as shown in diagram 8.1.



Customer Form	
First Name	
Last Name	
Address 1	
Address 2	
Phone Number	
Mobile Number	
Email Address	
	Save Cancel

Delete	Name	Address	Phone Number	Mobile Number	Email Address
☐	Sharanam Shah	Shivaji park, Mumbai	22222222	9898989898	sharanam@sharanamshah.com
☐	Vaishali Shah	Dombivili, Mumbai	33333333	9899889898	vaishali@sharanamshah.com

Diagram 8.1: Customers G.U.I

The form allows:

- Inserting data into the Customers table
- Updating data already existing in the Customers table
- Viewing of data available in the Customers table
- Deleting data from the Customers table

Functions declared in CustomerServlet.java:

JavaScript	setMode()
	setDelMode()
	formDeleteValues()
	setEditMode()

Form Details:

Form Name	frmCustomers
Form Title	Customer Form
Bound To	bookshop.Customers

Data Fields:

Object	Label	Name	Bound To
Hidden	- -	hidMode	- -
Hidden	- -	hidSelDel	- -
Hidden	- -	hidCustomerNo	Customers.CustomerNo
TextBox	First Name	txtFirstName	Customers.FirstName
TextBox	Last Name	txtLastName	Customers.LastName
TextBox	Address 1	txtAddress1	Customers.AddressLine1
TextBox	Address 2	txtAddress2	Customers.AddressLine2
TextBox	Phone Number	txtPhoneNumber	Customers.PhoneNumber
TextBox	Mobile Number	txtMobileNumber	Customers.MobileNumber
TextBox	Email Address	txtEmailAddress	Customers.EmailAddress
Checkbox	- -	chk [For CustomerNo]	(Customers.CustomerNo)'s value

Data Controls:

Object	Label	Name	Action
Button	Save	cmdSubmit	- -
Button	Cancel	cmdCancel	JavaScript:setMode()
Button	Delete	cmdDelete	JavaScript:setDelMode()

First create a table named **Customers** that stores valid customer information.

The structure of the MySQL table is as follows:

Column Definition:

Column Name	Data Type	Width	Description
CustomerNo	Integer	11	Identity number of the customer. Is the primary key. Is auto incremented
FirstName	varchar	30	The first name of the customer
LastName	varchar	30	The last name of the customer
AddressLine1	varchar	100	The address of the customer where it resides
AddressLine2	varchar	100	The address of the customer where it resides
PhoneNumber	varchar	30	The phone number of the customer
MobileNumber	varchar	30	The mobile number of the customer
EmailAddress	varchar	100	The email address of the customer

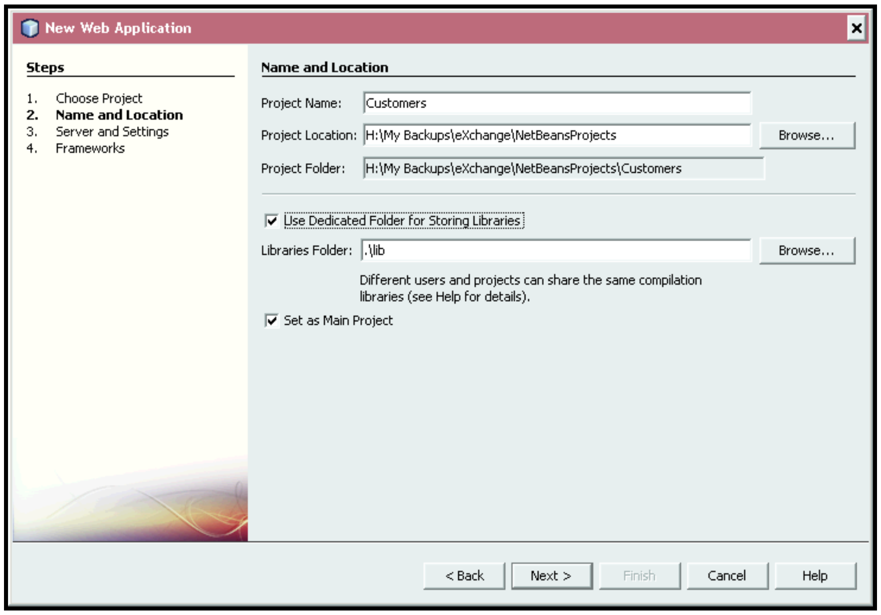
The SQL statement used to create the Customers table is as follows:

```
CREATE TABLE Customers(
   CustomerNo int(11) PRIMARY KEY AUTO_INCREMENT,
   FirstName varchar(30),
   LastName varchar(30),
AddressLine1 varchar(100),
AddressLine2 varchar(100),
PhoneNumber varchar(30),
MobileNumber varchar(30),
EmailAddress varchar(100));
```

Before starting with the code spec of Customers GUI, let's create a Web application named **Customers**. Follow the steps explained in *Chapter 07: Working With Servlets* to create a web application.

It's a good practice to create a dedicated ***lib*** *directory to hold all the required library files in the project directory.*

When **New Project** dialog box prompts for the Web Application name, enter the name and also select the option **Use Dedicated Folder for Storing Libraries** as shown in diagram 8.2.1.



Project Name: Customers
Project Location: H:\My Backups\eXchange\NetBeansProjects
Project Folder: H:\My Backups\eXchange\NetBeansProjects\Customers
[✓] Use Dedicated Folder for Storing Libraries
Libraries Folder: .\lib
Different users and projects can share the same compilation libraries (see Help for details).
[✓] Set as Main Project

Diagram 8.2.1: Selecting the option for Dedicated folder for storing libraries

This creates a **lib** directory under the web application named **Customers** as shown in diagram 8.2.2. The **lib** folder holds a few pre-defined library files created by NetBeans IDE required that support Web Application development.

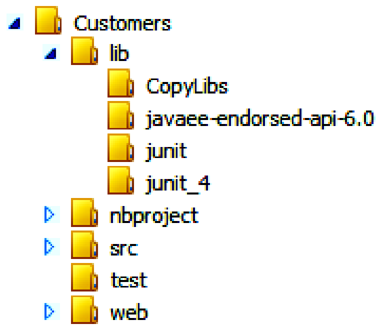


Diagram 8.2.2: A dedicated lib directory created for storing libraries

Adding MySQL Connector/J JAR File To The Web Application

Extract the contents of the downloaded **MySQl Connector/J** driver to a folder of choice.

Right-click **Libraries** directory and select **Add JAR/Folder...** as shown in diagram 8.3.1.

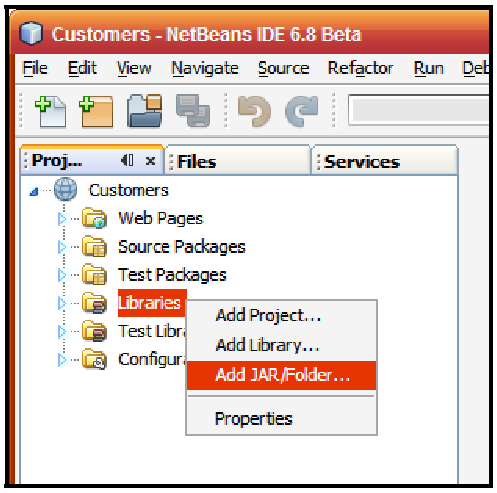


Diagram 8.3.1: Selecting Add Jar/Folder

Add JAR/Folder dialog box to choose the JAR files appears as shown in diagram 8.3.2.

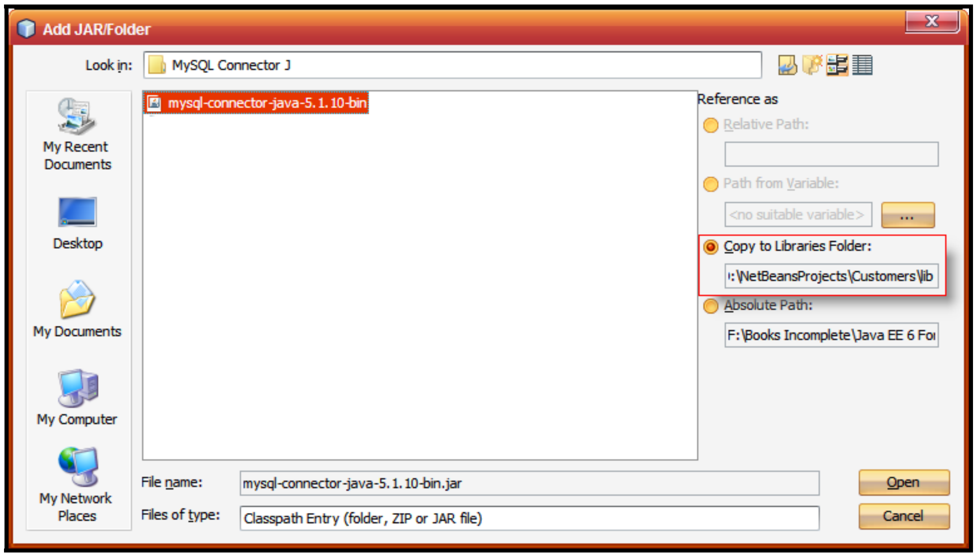

Add JAR/Folder
Look in: MySQL Connector J
mysql-connector-java-5.1.10-bin
My Recent Documents
Desktop
My Documents
My Computer
My Network Places
Reference as
Relative Path:
Path from Variable:
<no suitable variable>
...
Copy to Libraries Folder:
:\NetBeansProjects\Customers\lib
Absolute Path:
F:\Books Incomplete\Java EE 6 Fo
File name: mysql-connector-java-5.1.10-bin.jar
Files of type: Classpath Entry (folder, ZIP or JAR file)
Open
Cancel

Diagram 8.3.2: Selecting MySQL Connector/J JAR file

Select **mysql-connector-java-5.1.10-bin.jar** file [from the directory, that holds the extracted contents of the MySQL Connector/J driver] to add it to the project and choose the option **Copy to Libraries Folder** as shown in diagram 8.3.2. Click Open.

The **mysql-connector-java-5.1.10-bin.jar** file is available in NetBeans IDE under the web application's **Libraries** section as shown in diagram 8.3.3.1 as well as in the **lib** directory of the web application as shown in diagram 8.3.3.2.

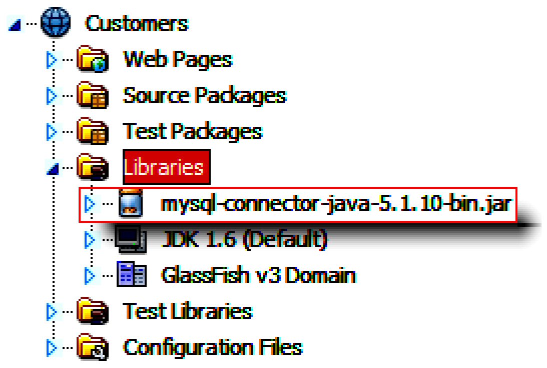


Diagram 8.3.3.1: JAR added in the Libraries directory of the web application in NetBeans

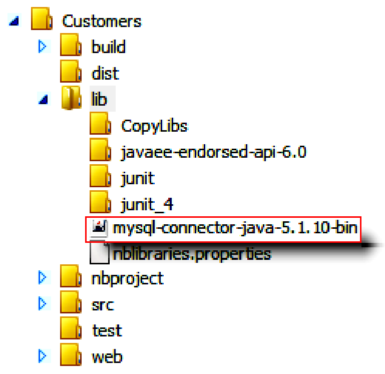


Diagram 8.3.3.2: JAR added in the lib [dedicated] directory of the web application

Next create a Servlet named **CustomerServlet**. Follow the steps as explained in *Chapter 07: Working With Servlets* to create an annotated servlet.

Code spec: [CustomerServlet.java]

```java
package servlet;

import java.io.IOException;
import java.io.PrintWriter;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(name="CustomerServlet", urlPatterns={"/CustomerServlet"})
public class CustomerServlet extends HttpServlet {
    @Override
    protected void service(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        Connection dbcon = null;
        Statement stmt = null;
        ResultSet rs;
        String query = null;
        try {
            Class.forName("com.mysql.jdbc.Driver").newInstance();
            dbcon = DriverManager.getConnection("jdbc:mysql://localhost/bookshop", "root",
            "123456");
        }
        catch(Exception e) {
            out.println("Sorry failed to connect to the Database. " + e.getMessage());
        }

        if("D".equals(request.getParameter("hidMode")) && dbcon != null) {
            try {
                stmt = dbcon.createStatement();
                query = "DELETE FROM Customers WHERE CustomerNo IN(" +
                request.getParameter("hidSelDel") + ")";
                stmt.executeUpdate(query);
                response.sendRedirect("CustomerServlet");
            }
            catch(Exception e) {
                out.println("Sorry failed to delete values from the database table. " + e.getMessage());
            }
        }

        if("U".equals(request.getParameter("hidMode")) && dbcon != null) {
            try {
                stmt = dbcon.createStatement();
                query = "UPDATE Customers SET FirstName = '" + request.getParameter("txtFirstName")
        + "', LastName = '" + request.getParameter("txtLastName") + "', AddressLine1 = '" +
        request.getParameter("txtAddress1") + "', AddressLine2 = '" +
        request.getParameter("txtAddress2") + "', PhoneNumber = '" +
        request.getParameter("txtPhoneNumber") + "', MobileNumber = '" +
        request.getParameter("txtMobileNumber") + "', EmailAddress = '" +
        request.getParameter("txtEmailAddress") + "' WHERE CustomerNo = '" +
        request.getParameter("hidCustomerNo") + "'";
        stmt.executeUpdate(query);
        response.sendRedirect("CustomerServlet");
    }
    catch(Exception e) {
        out.println("Sorry failed to update values from the database table. " + e.getMessage());
    }
}

if("I".equals(request.getParameter("hidMode")) && dbcon != null) {
    String firstName = request.getParameter("txtFirstName");
    String lastName = request.getParameter("txtLastName");
    String address1 = request.getParameter("txtAddress1");
    String address2 = request.getParameter("txtAddress2");
    String phone = request.getParameter("txtPhoneNumber");
    String mobile = request.getParameter("txtMobileNumber");
    String email = request.getParameter("txtEmailAddress");
    try {
        if(firstName.length() > 0 && lastName.length() > 0 && address1.length() > 0 &&
        address2.length() > 0 && phone.length() > 0 && mobile.length() > 0 && email.length() >
        0) {
            stmt = dbcon.createStatement();
            query = "INSERT INTO Customers (FirstName, LastName, AddressLine1, AddressLine2,
            PhoneNumber, MobileNumber, EmailAddress) VALUES ('" + firstName + "', '" +
            lastName + "', '" + address1 + "', '" + address2 + "', '" + phone + "', '" + mobile + "', '"
            + email + "')";
            stmt.executeUpdate(query);
            response.sendRedirect("CustomerServlet");
        }
        else {
            out.println("Customer details cannot be left blank.");
        }
    }
    catch(Exception e) {
        out.println("Sorry failed to insert values into the Database table. " + e.getMessage());
    }
}
out.println("<html>");
out.println("<head>");
out.println("<script language='JavaScript'>");
out.println("function setMode() {");
out.println("document.frmCustomers.txtFirstName.value = '';");
out.println("document.frmCustomers.txtLastName.value = '';");
out.println("document.frmCustomers.txtAddress1.value = '';");
out.println("document.frmCustomers.txtAddress2.value = '';");
out.println("document.frmCustomers.txtPhoneNumber.value = '';");
out.println("document.frmCustomers.txtMobileNumber.value = '';");
out.println("document.frmCustomers.txtEmailAddress.value = '';");
out.println("}");
out.println("function setDelMode() {");
out.println("document.frmCustomers.hidMode.value = 'D';");
out.println("formDeleteValues('hidSelDel');");
out.println("}");
out.println("function formDeleteValues(hidden) {");
out.println("var selValues = '';");
out.println("for (i=0;i<document.forms[0].elements.length;i++) {");
out.println("if(document.forms[0].elements[i].type == \"checkbox\") {");
out.println("if (document.forms[0].elements[i].checked == true) {");
out.println("selValues = selValues + document.forms[0].elements[i].value + \",\";");
out.println("}");
out.println("}");
out.println("}");
out.println("if (selValues.length < 1) {");
out.println("alert(\"Please choose records you wish to delete.\");");
out.println("}");
out.println("else {");
out.println("selValues = selValues.substring(0,selValues.length-1);");
out.println("eval(\"document.forms[0].\"+hidden+\".value = '\" + selValues + \"';\");");
out.println("document.forms[0].submit();");
out.println("}");
out.println("}");
out.println("function setEditMode(CustomerNo, FirstName, LastName, AddressLine1,
AddressLine2, PhoneNumber, MobileNumber, EmailAddress) {");
out.println("document.frmCustomers.hidCustomerNo.value = CustomerNo;");
out.println("document.frmCustomers.txtFirstName.value = FirstName;");
out.println("document.frmCustomers.txtLastName.value = LastName;");
out.println("document.frmCustomers.txtAddress1.value = AddressLine1;");
out.println("document.frmCustomers.txtAddress2.value = AddressLine2;");
out.println("document.frmCustomers.txtPhoneNumber.value = PhoneNumber;");
out.println("document.frmCustomers.txtMobileNumber.value = MobileNumber;");
out.println("document.frmCustomers.txtEmailAddress.value = EmailAddress;");
out.println("document.frmCustomers.hidMode.value = 'U';");
out.println("}");
out.println("</script>");
out.println("<title>Customer Form</title>");
out.println("</head>");
out.println("<body bgcolor='pink'>");
out.println("<form action='CustomerServlet' method='POST' name='frmCustomers'>");
out.println("<input name='hidMode' type='hidden' value='I'>");
out.println("<input name='hidSelDel' type='hidden'>");
out.println("<input name='hidCustomerNo' type='hidden'>");
out.println("<table align='center' bgcolor='pink' cellpadding='0' cellspacing='0' name='tblouter'
width='85%'>");
out.println("<tr height='200' valign='top'>");
out.println("<td align='center' colspan='10'>");
out.println("<table align='center' bgcolor='pink' border='1' bordercolor='maroon' cellpadding='2'
cellspacing='0' name='tblFirstChild' width='100%'>");
out.println("<tr>");
out.println("<td align='left' colspan='2' bgcolor='maroon'>");
out.println("<font color='pink'>Customer Form</font>");
out.println("</td>");
out.println("</tr>");
out.println("<tr>");
out.println("<td align='right' width='25%'>First Name</td>");
out.println("<td align='left'>");
out.println("<input maxlength='35' name='txtFirstName' type='text' size='30'>");
out.println("</td>");
out.println("</tr>");
out.println("<tr>");
out.println("<td align='right' width='25%'>Last Name</td>");
out.println("<td align='left'>");
out.println("<input maxlength='35' name='txtLastName' type='text' size='30'>");
out.println("</td>");
out.println("</tr>");
out.println("<tr>");
out.println("<td align='right' width='25%'>Address 1</td>");
out.println("<td align='left'>");
out.println("<input maxlength='75' name='txtAddress1' type='text' size='50'>");
out.println("</td>");
out.println("</tr>");
out.println("<tr>");
out.println("<td align='right' width='25%'>Address 2</td>");
out.println("<td align='left'>");
out.println("<input maxlength='75' name='txtAddress2' type='text' size='50'>");
out.println("</td>");
out.println("</tr>");
out.println("<tr>");
out.println("<td align='right' width='25%'>Phone Number</td>");
out.println("<td align='left'>");
out.println("<input maxlength='35' name='txtPhoneNumber' type='text' size='30'>");
out.println("</td>");
out.println("</tr>");
out.println("<tr>");
out.println("<td align='right' width='25%'>Mobile Number</td>");
out.println("<td align='left'>");
out.println("<input maxlength='35' name='txtMobileNumber' type='text' size='30'>");
out.println("</td>");
out.println("</tr>");
out.println("<tr>");
out.println("<td align='right' width='25%'>Email Address</td>");
out.println("<td align='left'>");
out.println("<input maxlength='75' name='txtEmailAddress' type='text' size='50'>");
out.println("</td>");
out.println("</tr>");
out.println("<tr>");
out.println("<td colspan='2' align='right'>");
out.println("<input name='cmdSubmit' type='submit' value='Save'>");
out.println("<input name='cmdCancel' onClick='setMode();' type='button' value='Cancel'>");
out.println("</td>");
out.println("</tr>");
out.println("</table>");
out.println("</td>");
out.println("</tr>");
out.println("</table><br>");
if(dbcon != null) {
   try {
      stmt = dbcon.createStatement();
      query = "SELECT * FROM Customers";
      rs = stmt.executeQuery(query);
      out.println("<table align='center' border='1' width='85%' bordercolor='skyblue'
      cellpadding='0' cellspacing='0' name='tblSecondChild'>");
      out.println("<tr bgcolor='black'>");
      out.println("<td width='12%' align='center'><input name='cmdDelete' type='button'
      value='Delete' onClick='setDelMode();'></td>");
      out.println("<td><font color='#FFFFFF'>Name</font></td>");
```

```
                out.println("<td><font color='#FFFFFF'>Address</font></td>");
                out.println("<td><font color='#FFFFFF'>Phone Number</font></td>");
                out.println("<td><font color='#FFFFFF'>Mobile Number</font></td>");
                out.println("<td><font color='#FFFFFF'>Email Address</font></td>");
                out.println("</tr>");
                if(rs != null) {
                    while(rs.next()) {
                        out.println("<tr>");
                        out.println("<td><input type='checkbox' name='chk" + rs.getString("CustomerNo")
                        + "' VALUE='" + rs.getString("CustomerNo") + "'></td>");
                        out.println("<td style=\"cursor:pointer\" onMouseDown=\"setEditMode('" + rs.
                        getString("CustomerNo") + "', '" + rs.getString("FirstName") + "', '" +
                        rs.getString("LastName") + "', '" + rs.getString("AddressLine1") + "', '" +
                        rs.getString("AddressLine2") + "', '" + rs.getString("PhoneNumber") + "', '" + rs.
                        getString("MobileNumber") + "', '" + rs.getString("EmailAddress") + "');\">" +
                        rs.getString("FirstName") + " " + rs.getString("LastName") + "</td>");
                        out.println("<td style=\"cursor:pointer\" onMouseDown=\"setEditMode('" + rs.
                        getString("CustomerNo") + "', '" + rs.getString("FirstName") + "', '" +
                        rs.getString("LastName") + "', '" + rs.getString("AddressLine1") + "', '" +
                        rs.getString("AddressLine2") + "', '" + rs.getString("PhoneNumber") + "', '" + rs.
                        getString("MobileNumber") + "', '" + rs.getString("EmailAddress") + "');\">" +
                        rs.getString("AddressLine1") + ", " + rs.getString("AddressLine2") + "</td>");
                        out.println("<td style=\"cursor:pointer\" onMouseDown=\"setEditMode('" + rs.
                        getString("CustomerNo") + "', '" + rs.getString("FirstName") + "', '" +
                        rs.getString("LastName") + "', '" + rs.getString("AddressLine1") + "', '" +
                        rs.getString("AddressLine2") + "', '" + rs.getString("PhoneNumber") + "', '" + rs.
                        getString("MobileNumber") + "', '" + rs.getString("EmailAddress") + "');\">" +
                        rs.getString("PhoneNumber") + "</td>");
                        out.println("<td style=\"cursor:pointer\" onMouseDown=\"setEditMode('" + rs.
                        getString("CustomerNo") + "', '" + rs.getString("FirstName") + "', '" +
                        rs.getString("LastName") + "', '" + rs.getString("AddressLine1") + "', '" +
                        rs.getString("AddressLine2") + "', '" + rs.getString("PhoneNumber") + "', '" + rs.
                        getString("MobileNumber") + "', '" + rs.getString("EmailAddress") + "');\">" +
                        rs.getString("MobileNumber") + "</td>");
                        out.println("<td style=\"cursor:pointer\" onMouseDown=\"setEditMode('" + rs.
                        getString("CustomerNo") + "', '" + rs.getString("FirstName") + "', '" +
                        rs.getString("LastName") + "', '" + rs.getString("AddressLine1") + "', '" +
                        rs.getString("AddressLine2") + "', '" + rs.getString("PhoneNumber") + "', '" + rs.
                        getString("MobileNumber") + "', '" + rs.getString("EmailAddress") + "');\">" +
                        rs.getString("EmailAddress") + "</td>");
                        out.println("</tr>");
                    }
                }
                out.println("</table>");
                dbcon.close();
            }
            catch(Exception e) {
                out.println("Sorry Failed to execute the query. " + e.getMessage());
            }
        }
        out.println("</form>");
        out.println("</body>");
        out.println("</html>");
        out.close();
    }
}
```

Explanation:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
```

java.sql package is used to connect to the MySQL database. java.sql package contains majority of class objects used for database access.

The following interfaces or classes are imported**:**

- DriverManager
- Connection
- ResultSet
- Statement

Almost every database driver supports the components of this package.

```
Connection dbcon = null;
Statement stmt = null;
ResultSet rs;
```

Connection, ResultSet and Statement objects are declared, with the Connection and Statement objects set to null.

The **Connection** object is an interface which defines a link to a database. The Connection object is essentially a pipeline between the Java code spec and the database engine, which exposes the database tables and their data to the Java code spec.

The **Statement** object is an interface that represents how application data requests are sent from Java code spec to the database engine. Statement objects can hold ANSI SQL statements compatible across all database systems.

The **ResultSet** object is an interface that represents a set of records retrieved from the database. Different SQL statements including stored procedures, may return one or more ResultSet objects.

```
String query = null;
```

A variable named query of type String is declared and its value is set to null.

```
Class.forName("com.mysql.jdbc.Driver").newInstance();
dbcon = DriverManager.getConnection("jdbc:mysql://localhost/bookshop", "root", "123456");
```

Class.forName() is used to load the class into memory. It indicates that the JDBC driver from some JDBC vendor has to be loaded into the application.

Class.forName() is a **static** method, which instructs the JVM to dynamically locate, load and link to the class specified to it as a parameter. **newInstance()** indicates that a new instance of the current class should be created.

Next, to connect to a database, create a JDBC Connection object. This acts as a factory for Statement objects that provide the Java application the ability to submit SQL commands to the database.

Creating a Connection involves a single call to the DriverManager object.

getConnection() is overloaded to accept a URL that encodes the username and password**:**

- The database url i.e. **jdbc:mysql://localhost/bookshop**
  - **jdbc** indicates that JDBC is being used to establish the database connection
  - **mysql** is the name of the MySQL database
  - **//localhost/bookshop** is the path to the database
    - **localhost** is the name of the host where MySQL resides
    - **bookshop** is the name of the database, which holds tables, views and so on
- The username of the MySQL user i.e. **root**
- The password of the MySQL user i.e. **123456**

REMINDER

DriverManager is a container for all registered drivers. In order to obtain the correct Connection object, DriverManager needs to be informed about which driver it should use and to which database it should connect. This information is encapsulated in a Database URL.

This application allows deleting, updating and inserting customer details. Hence to identify the mode of operation, a hidden form field called **hidMode** is used through out to determine the operation mode i.e. **D** for **D**elete, **U** for **U**pdate and **I** for **I**nsert.

```
if("D".equals(request.getParameter("hidMode")) && dbcon != null) {
   try {
      stmt = dbcon.createStatement();
      query = "DELETE FROM Customers WHERE CustomerNo IN(" +
      request.getParameter("hidSelDel") + ")";
      stmt.executeUpdate(query);
      response.sendRedirect("CustomerServlet");
   }
   catch(Exception e) {
      out.println("Sorry failed to delete values from the database table. " + e.getMessage());
   }
}
```

It is determined if the hidden data field **hidMode** holds the value **D** to ensure that the form is in delete mode, which happens when the user has chosen the desired records for deletion using checkboxes and clicked Delete.

A Statement object is spawned using a valid Connection object by invoking createStatement().

Once a valid Statement object is spawned, execute(), executeQuery() or executeUpdate() can be used to perform SQL actions.

executeUpdate() is normally used for SQL statements that do not return data from the database. This includes UPDATE, DELETE, INSERT and ALTER statements.

executeUpdate() will typically return an integer which indicates the number of rows affected by the UPDATE, DELETE and INSERT commands. For other SQL statements, executeUpdate() returns the value **0** which indicates successful execution of the SQL statement.

The DELETE SQL query is passed to executeUpdate() for execution.

The records are deleted based on the following WHERE clause:

```
WHERE CustomerNo IN(" + request.getParameter("hidSelDel") + ")
```

hidSelDel is populated in the HTML form using JavaScript. This hidden variable holds a list of comma-separated values [i.e. one or more CustomerNo's]. This is therefore passed to the WHERE condition using the IN operator which accepts multiple comma-separated values for deletion.

sendRedirect() of the HttpServletResponse interface is used to send a temporary redirect response to the user using the specified redirect location URL. Here the same form is served again to reflect the record deletion.

If the delete operation fails, then the catch block takes care of it by displaying a user-generated message along with the server-generated error message.

```
if("U".equals(request.getParameter("hidMode")) && dbcon != null) {
   try {
      stmt = dbcon.createStatement();
      query = "UPDATE Customers SET FirstName = '" + request.getParameter("txtFirstName")
      + "', LastName = '" + request.getParameter("txtLastName") + "', AddressLine1 = '" +
      request.getParameter("txtAddress1") + "', AddressLine2 = '" +
      request.getParameter("txtAddress2") + "', PhoneNumber = '" +
          request.getParameter("txtPhoneNumber") + "', MobileNumber = '" +
          request.getParameter("txtMobileNumber") + "', EmailAddress = '" +
          request.getParameter("txtEmailAddress") + "' WHERE CustomerNo = '" +
          request.getParameter("hidCustomerNo") + "'";
          stmt.executeUpdate(query);
          response.sendRedirect("CustomerServlet");
     }
     catch(Exception e) {
          out.println("Sorry failed to update values from the database table. " + e.getMessage());
     }
}
```

It is determined if the hidden data field **hidMode** holds the value **U** to ensure that the form is in update mode, which happens when the user has edited a desired record and clicked Save.

The UPDATE SQL query is passed to executeUpdate() for execution.

Finally, the same form is served again to reflect the updated records.

If the update operation fails, then the catch block takes care of it by displaying a user-generated message along with the server-generated error message.

```
if("I".equals(request.getParameter("hidMode")) && dbcon != null) {
     String firstName = request.getParameter("txtFirstName");
     String lastName = request.getParameter("txtLastName");
     String address1 = request.getParameter("txtAddress1");
     String address2 = request.getParameter("txtAddress2");
     String phone = request.getParameter("txtPhoneNumber");
     String mobile = request.getParameter("txtMobileNumber");
     String email = request.getParameter("txtEmailAddress");
     try {
          if(firstName.length() > 0 && lastName.length() > 0 && address1.length() > 0 &&
          address2.length() > 0 && phone.length() > 0 && mobile.length() > 0 && email.length() >
          0) {
               stmt = dbcon.createStatement();
               query = "INSERT INTO Customers (FirstName, LastName, AddressLine1, AddressLine2,
               PhoneNumber, MobileNumber, EmailAddress) VALUES ('" + firstName + "', '" +
               lastName + "', '" + address1 + "', '" + address2 + "', '" + phone + "', '" + mobile + "', '"
               + email + "')";
               stmt.executeUpdate(query);
               response.sendRedirect("CustomerServlet");
          }
          else {
               out.println("Customer details cannot be left blank.");
          }
     }
     catch(Exception e) {
          out.println("Sorry failed to insert values into the Database table. " + e.getMessage());
```

```
    }
}
```

It is determined if the hidden data field **hidMode** holds the value **I** to ensure that the form is in insert mode, which happens when the user has added a new record and clicked Save.

The variables named firstName, lastName, address1, address2, phone, mobile and email of type String are declared and populated with the values entered by the user in the form.

It is determined if the length of the values held by the variables i.e. firstName, lastName, address1, address2, phone, mobile and email is more than zero i.e. the textboxes are not left empty.

If this is true, then a valid INSERT Statement is passed to executeUpdate() for execution.

If this is false, then a user defined error message is displayed stating that the customer details cannot be left blank.

Finally, the same form is served again to reflect the updated records.

If the insert operation fails, then the catch block takes care of it by displaying a user-generated message along with the server-generated error message.

This application uses JavaScript for clearing the contents of the form and to allow the edit and delete operations. In case of the **edit** operation, the values are picked up from the HTML table and populated in the appropriate textboxes. In case of the **delete** operation, a comma-separated string holding the Customer numbers of the records chosen for deletion are made available to the Servlet for the actual record deletion.

```
out.println("function setMode() {");
out.println("document.frmCustomers.txtFirstName.value = ";");
out.println("document.frmCustomers.txtLastName.value = ";");
out.println("document.frmCustomers.txtAddress1.value = ";");
out.println("document.frmCustomers.txtAddress2.value = ";");
out.println("document.frmCustomers.txtPhoneNumber.value = ";");
out.println("document.frmCustomers.txtMobileNumber.value = ";");
out.println("document.frmCustomers.txtEmailAddress.value = ";");
out.println("}");
```

setMode() of JavaScript is called when Cancel is clicked. setMode() clears the form fields.

```
out.println("function setDelMode() {");
out.println("document.frmCustomers.hidMode.value = 'D';");
out.println("formDeleteValues('hidSelDel');");
out.println("}");
```

setDelMode() of JavaScript is called when Delete is clicked. setMode() performs the following operations:

- Switches the form mode to **D**elete. This is done, by setting the value of hidden variable named **hidMode** to **D**. This variable will be used to understand the mode in which the form is and accordingly perform appropriate database operation
- Calls a user-defined function **formDeleteValues()** with a parameter that when returned holds a list of CustomerNos for deletion. This function generates a string of comma-separated Identities of the records selected for deletion. If no records are selected, a message indicates the same

```
out.println("function formDeleteValues(hidden) {");
out.println("var selValues = '';");
out.println("for (i=0;i<document.forms[0].elements.length;i++) {");
out.println("if(document.forms[0].elements[i].type == \"checkbox\") {");
out.println("if (document.forms[0].elements[i].checked == true) {");
out.println("selValues = selValues + document.forms[0].elements[i].value + \",\";");
out.println("}");
out.println("}");
out.println("}");
out.println("if (selValues.length < 1) {");
out.println("alert(\"Please choose records you wish to delete.\");");
out.println("}");
out.println("else {");
out.println("selValues = selValues.substring(0,selValues.length-1);");
out.println("eval(\"document.forms[0].\"+hidden+\".value = '\" + selValues + \"';\");");
out.println("document.forms[0].submit();");
out.println("}");
out.println("}");
```

formDeleteValues() is called to create a list of records that are selected for being deleted. This list holds the Identity numbers of the records in a comma separated format. This function:

- Declares a string variable **selValues** and initializes it with an empty string
- Traverses through all the form elements
- On every iteration:
  - Verifies whether the form element is a checkbox
  - Verifies whether the checkbox is checked
  - Generates [and appends on every iteration] a string to hold the records [in form of Customer IDentities] selected for deletion in a comma-separated fashion in a string variable **selValues**
- Verifies whether the variable **selValues** [after the traversing is done] holds any value [the comma-separated string], if false:
  - Displays a message informing the user to choose records

  - If true, removes the last comma from the string which contains comma-separated identities for deletion
  - Submits the form

```
out.println("function setEditMode(CustomerNo, FirstName, LastName, AddressLine1,
AddressLine2, PhoneNumber, MobileNumber, EmailAddress) {");
out.println("document.frmCustomers.hidCustomerNo.value = CustomerNo;");
out.println("document.frmCustomers.txtFirstName.value = FirstName;");
out.println("document.frmCustomers.txtLastName.value = LastName;");
out.println("document.frmCustomers.txtAddress1.value = AddressLine1;");
out.println("document.frmCustomers.txtAddress2.value = AddressLine2;");
out.println("document.frmCustomers.txtPhoneNumber.value = PhoneNumber;");
out.println("document.frmCustomers.txtMobileNumber.value = MobileNumber;");
out.println("document.frmCustomers.txtEmailAddress.value = EmailAddress;");
out.println("document.frmCustomers.hidMode.value = 'U';");
out.println("}");
```

setEditMode() accepts parameters [The column values available on the Customers GRID] and transfers the same to the form fields [usually textboxes] on the Customer Form for editing. It is called when a record is selected for modification and performs the following operations:

- Transfers the value held by the first parameter to the hidden variable [hidCustomerNo]
- Transfers the value held by the second, third, fourth, fifth, sixth and seventh parameter to appropriate form fields thus making it available for modification
- Switches the form mode to **U**pdate. This is done, by setting the value of hidden variable named **hidMode** to **U**. This variable will be used to understand the mode in which the form is and accordingly perform appropriate database operation

```
out.println("<form action='CustomerServlet' method='POST' name='frmCustomers'>");
```

An HTML form is initialized, which submits the data captured by the form for the processing. Since the same file processes the data captured by the form, the ACTION attribute of the HTML <FORM> element points to it.

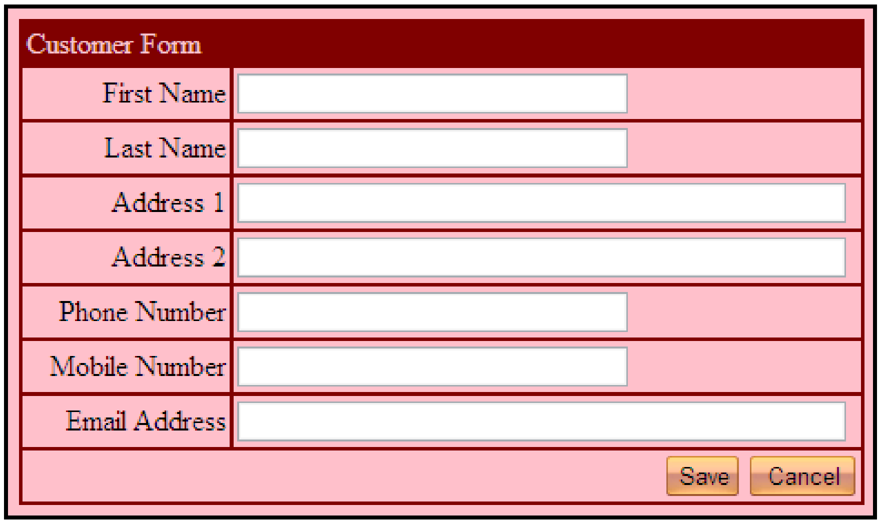
```
out.println("<input name='hidMode' type='hidden' value='I'>");
out.println("<input name='hidSelDel' type='hidden'>");
out.println("<input name='hidCustomerNo' type='hidden'>");
```

The following hidden variables are declared using the HTML <INPUT> element**:**

- hidMode**:** Is used to determine the form mode. Holds 'I' for **I**nsert, 'U' for **U**pdate and 'D' for **D**elete. It holds the default value as 'I' when the page is rendered for the first time
- hidSelDel**:** Holds the customer identities for identifying records which have been selected for deletion
- hidCustomerNo**:** Holds the Customer Identity for the update operation

```
out.println("<table align='center' bgcolor='pink' cellpadding='0' cellspacing='0' name='tblouter'
width='85%'>");
out.println("<tr height='200' valign='top'>");
out.println("<td align='center' colspan='10'>");
. . .
. . .
. . .
out.println("</td>");
out.println("</tr>");
out.println("</table><br>");
```

HTML elements follow for generating the form fields such as First Name, Last name, Address 1, Address 2, Phone Number, Mobile Number and Email Address and buttons such as Save and Cancel as shown in diagram 8.4.1.



Customer Form

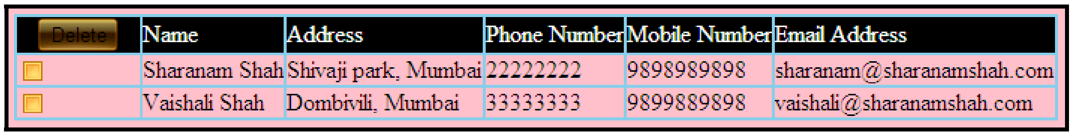
First Name	
Last Name	
Address 1	
Address 2	
Phone Number	
Mobile Number	
Email Address	

Diagram 8.4.1: HTML elements generating form fields

```
if(dbcon != null) {
   try {
      stmt = dbcon.createStatement();
      query = "SELECT * FROM Customers";
      rs = stmt.executeQuery(query);
      out.println("<table align='center' border='1' width='85%' bordercolor='skyblue'
      cellpadding='0' cellspacing='0' name='tblSecondChild'>");
      out.println("<tr bgcolor='black'>");
      out.println("<td width='12%' align='center'><input name='cmdDelete' type='button'
      value='Delete' onClick='setDelMode();'></td>");
      out.println("<td><font color='#FFFFFF'>Name</font></td>");
      . . .
      . . .
      out.println("</tr>");
```

```
        if(rs != null) {
            while(rs.next()) {
                out.println("<tr>");
                out.println("<td><input type='checkbox' name='chk" + rs.getString("CustomerNo")
                + "' VALUE='" + rs.getString("CustomerNo") + "'></td>");
                out.println("<td style=\"cursor:pointer\" onMouseDown=\"setEditMode('" + rs.
                getString("CustomerNo") + "', '" + rs.getString("FirstName") + "', '" +
                rs.getString("LastName") + "', '" + rs.getString("AddressLine1") + "', '" +
                rs.getString("AddressLine2") + "', '" + rs.getString("PhoneNumber") + "', '" + rs.
                getString("MobileNumber") + "', '" + rs.getString("EmailAddress") + "');\">" +
                rs.getString("FirstName") + " " + rs.getString("LastName") + "</td>");
                . . .
                . . .
                out.println("</tr>");
            }
        }
        out.println("</table>");
        dbcon.close();
    }
    catch(Exception e) {
            out.println("Sorry Failed to execute the query. " + e.getMessage());
    }
}
```

A ResultSet named rs is declared, which holds the value returned by **executeQuery()**, which in turn holds the SELECT SQL statement. It provides access to the records available in the **Customers** table that have been extracted from the database. These records are displayed in a tabular form [GRID] as shown in diagram 8.4.2.



Delete	Name	Address	Phone Number	Mobile Number	Email Address
☐	Sharanam Shah	Shivaji park, Mumbai	22222222	9898989898	sharanam@sharanamshah.com
☐	Vaishali Shah	Dombivili, Mumbai	33333333	9899889898	vaishali@sharanamshah.com

Diagram 8.4.2

If the SQL query retrieves any records the following operations are performed:

- ❑ HTML code spec follows for the creation of Delete button with its **onClick** event set to call a function named **setDelMode()**
- ❑ A tabular layout is created using pure HTML <TABLE> element to hold the records retrieved. This is done using a **WHILE** loop, which traverses through the records retrieved from the table via the SQL query created earlier. ResultSet reads one row at a time, beginning from the first row to the last row. next() attempts to iterate to the next row in the ResultSet and returns false if the end of the ResultSet is encountered

WARNING

A ResultSet will always point to **before the first row** and therefore, next() has to be used to access data.

- Form fields as shown in diagram 8.4.2 are linked to the function **setEditMode()**. This function is responsible for populating the form fields when a record is clicked from the tabular layout [GRID]. The data is retrieved using **getString()** from the database. **getString()** extracts the data using column names

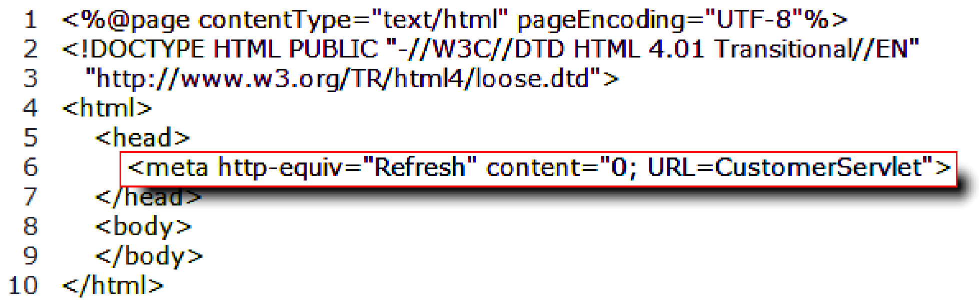
REMINDER

The ResultSet object has **getter** for all Java data types. The database driver tries to translate the SQL data type to the requested Java data type. Each of these methods is overloaded to accept either a column index or a column name. Columns are numbered from left to right and the first column is always number 1. If the column name is used, then the name is case insensitive regardless of the underlying database. Extracting data by column name makes code easier to read.

Modifying index.jsp

Modify the default index.jsp [to invoke the servlet CustomerServlet] which NetBeans creates as a part of the web application.

Code spec: [index.jsp]



```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">
<html>
   <head>
      <meta http-equiv="Refresh" content="0; URL=CustomerServlet">
   </head>
   <body>
   </body>
</html>
```

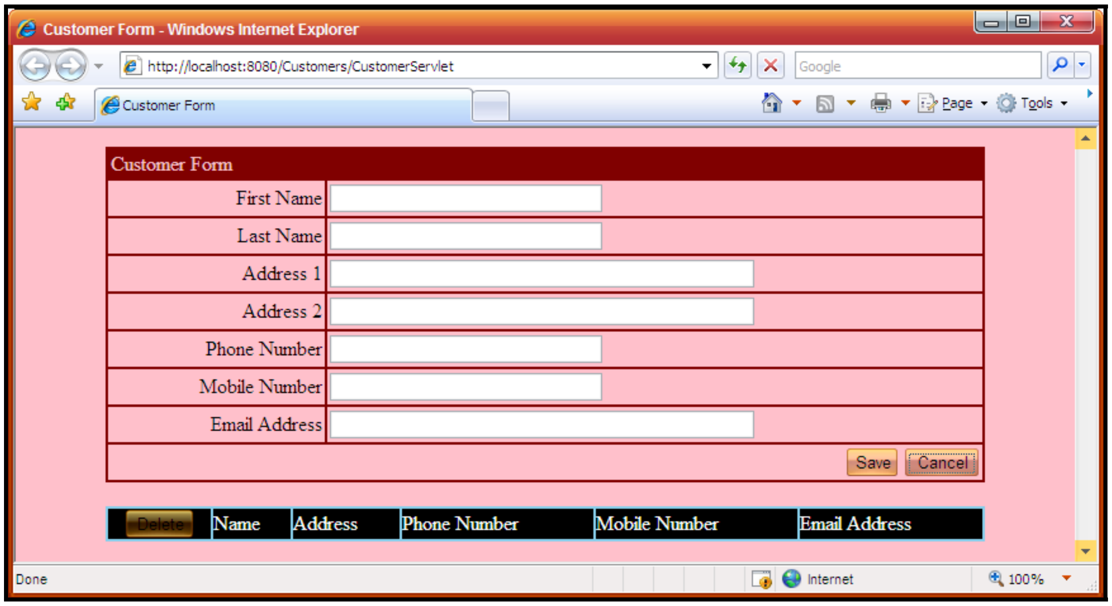
Once the files are ready and placed appropriately the deployment can begin. Compile and build the Web application.

Run the compiled web application.

WARNING

In this example, MySQL is assumed to reside on the same machine, which holds the JVM. Hence, the URL of the connection holds **localhost**.

If the connection fails, then ensure that the host file holds a mapping entry between localhost and 127.0.0.1.


Customer Form - Windows Internet Explorer
http://localhost:8080/Customers/CustomerServlet
Google
Customer Form
Page
Tools
Done
Internet
100%

Customer Form

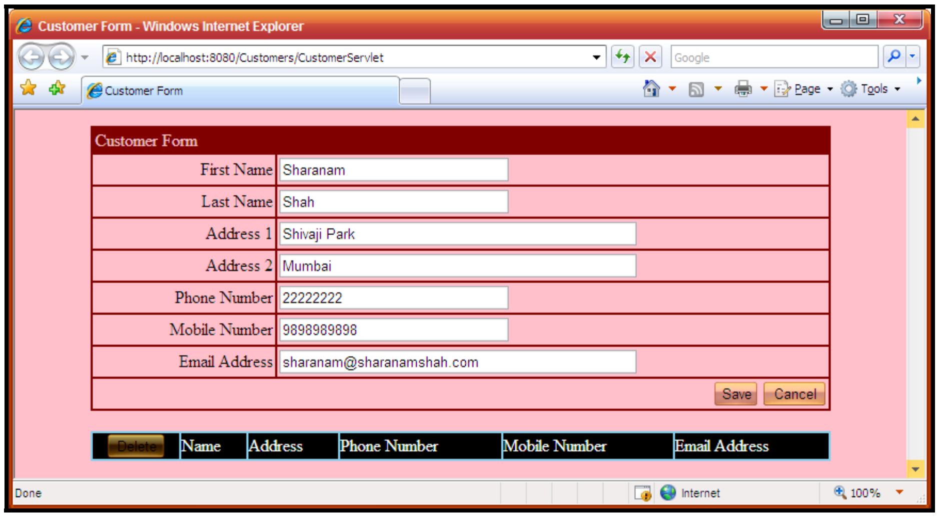
First Name	
Last Name	
Address 1	
Address 2	
Phone Number	
Mobile Number	
Email Address	
	Save Cancel

Delete	Name	Address	Phone Number	Mobile Number	Email Address

Diagram 8.4.3: Running the compile web application

By default, the Customer form is in Insert Mode.

Populate the textboxes in the Customer form as shown in diagram 8.4.4.


Customer Form - Windows Internet Explorer
http://localhost:8080/Customers/CustomerServlet
Customer Form
Page
Tools
Save
Cancel
Delete
Done
Internet
100%

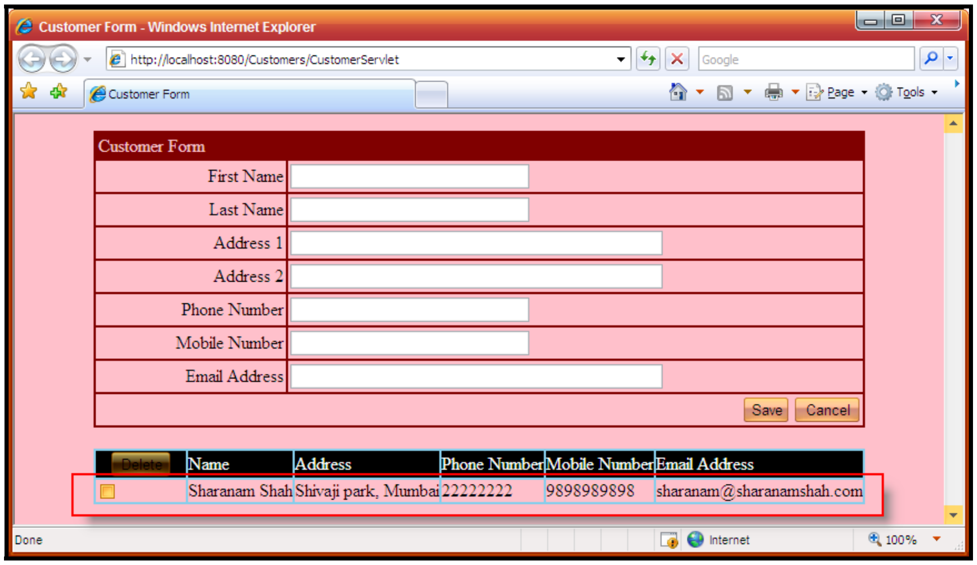
Customer Form

First Name	Sharanam
Last Name	Shah
Address 1	Shivaji Park
Address 2	Mumbai
Phone Number	22222222
Mobile Number	9898989898
Email Address	sharanam@sharanamshah.com

Delete	Name	Address	Phone Number	Mobile Number	Email Address

Diagram 8.4.4: Inserting records in the Customer Form

Click Save. The record is added to the database and the same is displayed in the tabular GRID below as shown in diagram 8.4.5.

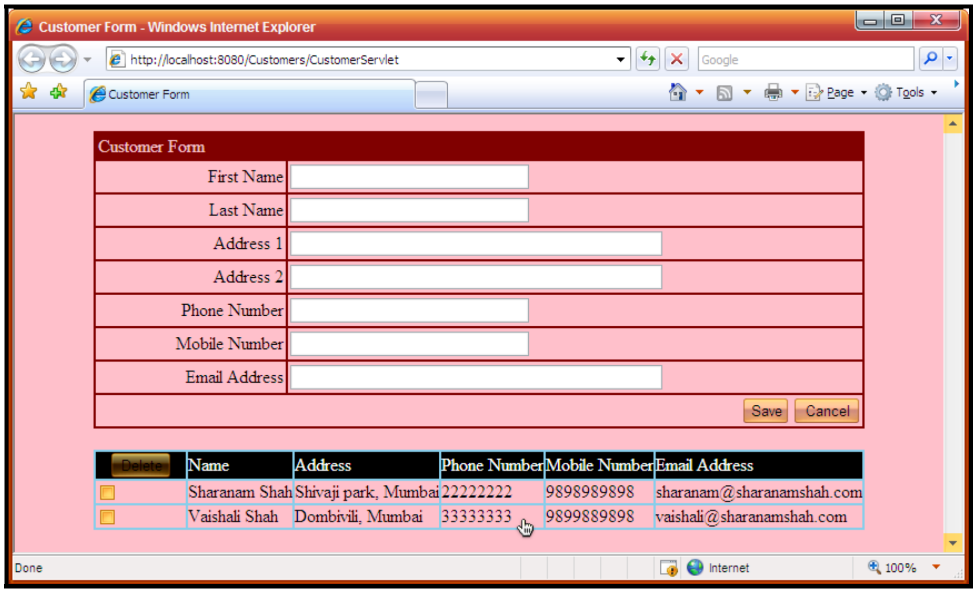


Customer Form

Delete	Name	Address	Phone Number	Mobile Number	Email Address
☐	Sharanam Shah	Shivaji park, Mumbai	22222222	9898989898	sharanam@sharanamshah.com

Diagram 8.4.5: Records added and the same viewed in the GRID

To update a particular record, take the mouse over that record and click on it as shown in diagram 8.4.6.


Customer Form - Windows Internet Explorer
http://localhost:8080/Customers/CustomerServlet
Customer Form
Save
Cancel
Delete
Done
Internet
100%

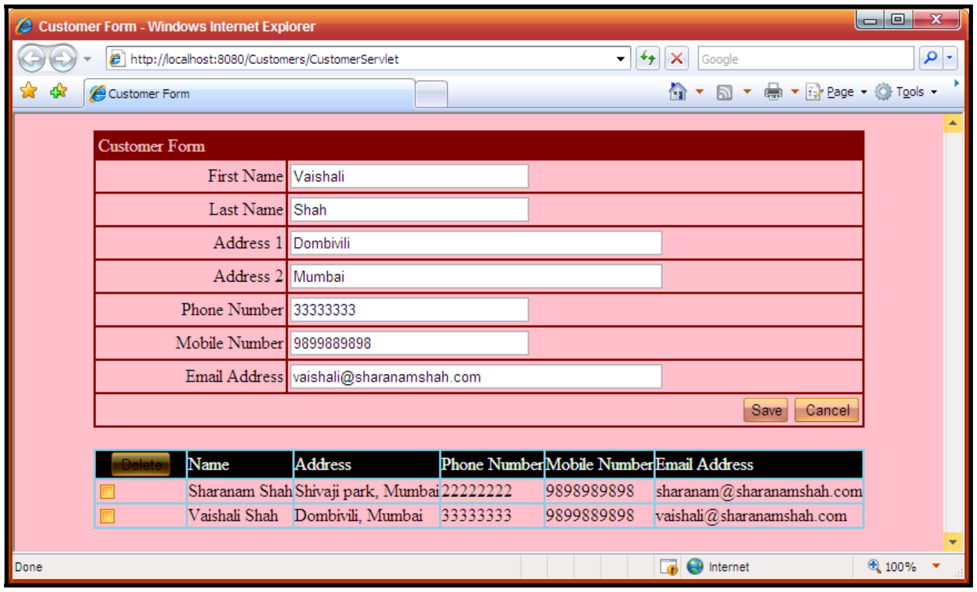
Customer Form

First Name	
Last Name	
Address 1	
Address 2	
Phone Number	
Mobile Number	
Email Address	

Delete	Name	Address	Phone Number	Mobile Number	Email Address
☐	Sharanam Shah	Shivaji park, Mumbai	22222222	9898989898	sharanam@sharanamshah.com
☐	Vaishali Shah	Dombivili, Mumbai	33333333	9899889898	vaishali@sharanamshah.com

Diagram 8.4.6: Mouse over the record to update the record

The pointer of the mouse changes if moved over the records. Click once and the textboxes in the Customer form are populated with that particular record as shown in diagram 8.4.7.


Customer Form - Windows Internet Explorer
http://localhost:8080/Customers/CustomerServlet
Customer Form
Save
Cancel
Delete
Done
Internet
100%

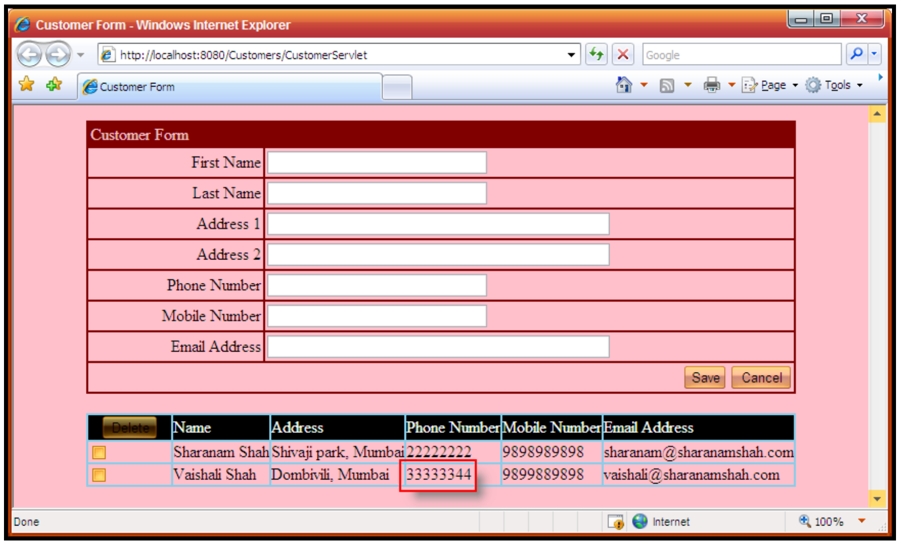
Customer Form

First Name	Vaishali
Last Name	Shah
Address 1	Dombivili
Address 2	Mumbai
Phone Number	33333333
Mobile Number	9899889898
Email Address	vaishali@sharanamshah.com

Delete	Name	Address	Phone Number	Mobile Number	Email Address
☐	Sharanam Shah	Shivaji park, Mumbai	22222222	9898989898	sharanam@sharanamshah.com
☐	Vaishali Shah	Dombivili, Mumbai	33333333	9899889898	vaishali@sharanamshah.com

Diagram 8.4.7: Record in updation mode in the Customer Form

Make the required changes and click Save. The record is updated in the database and the same is reflected in the tabular GRID below as shown in diagram 8.4.8.


Customer Form - Windows Internet Explorer

http://localhost:8080/Customers/CustomerServlet

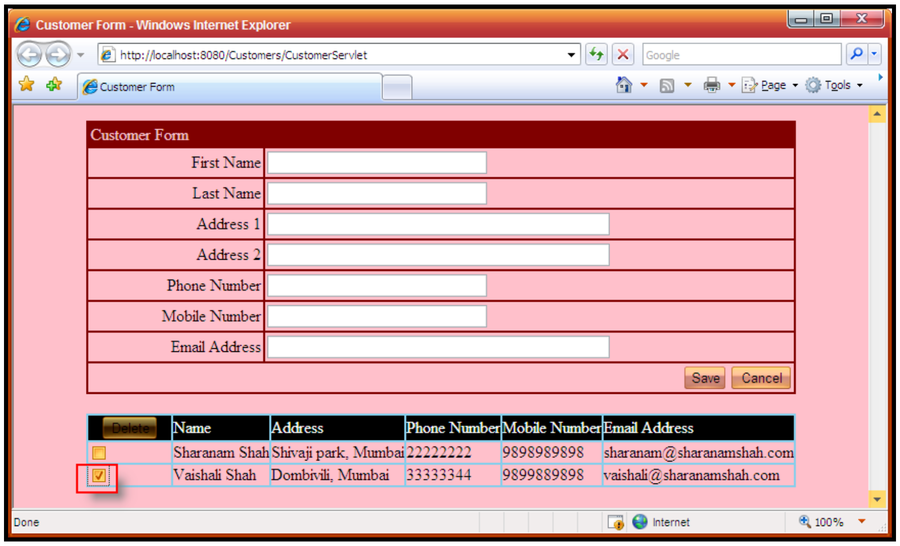
Customer Form

Customer Form	
First Name	
Last Name	
Address 1	
Address 2	
Phone Number	
Mobile Number	
Email Address	

Delete	Name	Address	Phone Number	Mobile Number	Email Address
☐	Sharanam Shah	Shivaji park, Mumbai	22222222	9898989898	sharanam@sharanamshah.com
☐	Vaishali Shah	Dombivili, Mumbai	33333344	9899889898	vaishali@sharanamshah.com

Diagram 8.4.8: Updated record in the Customer Form

To delete a particular record(s), switch on the checkbox available next to the record to be deleted as shown in diagram 8.4.9.


Customer Form - Windows Internet Explorer

http://localhost:8080/Customers/CustomerServlet

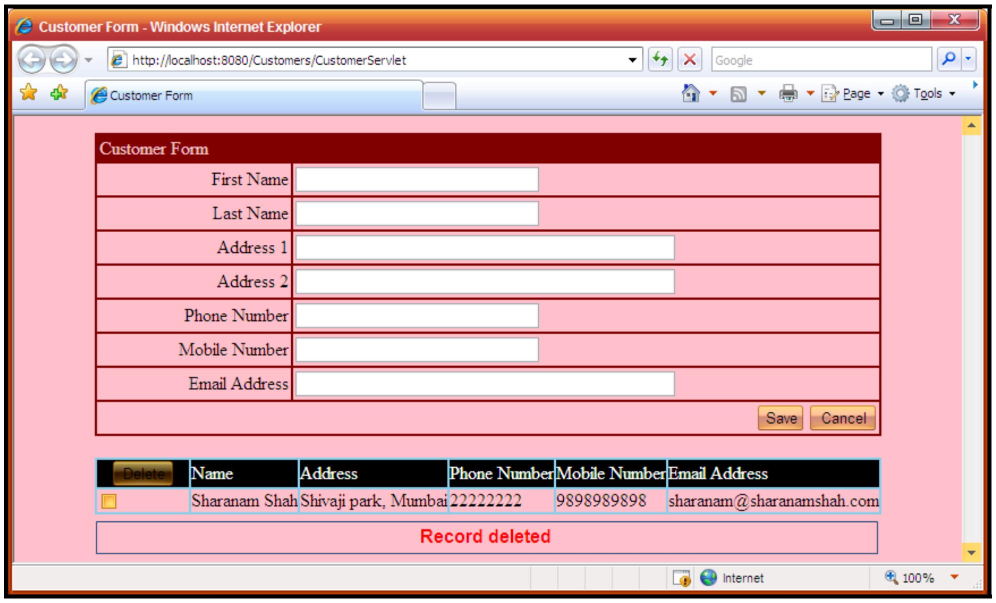
Customer Form

Customer Form	
First Name	
Last Name	
Address 1	
Address 2	
Phone Number	
Mobile Number	
Email Address	

Delete	Name	Address	Phone Number	Mobile Number	Email Address
☐	Sharanam Shah	Shivaji park, Mumbai	22222222	9898989898	sharanam@sharanamshah.com
✓	Vaishali Shah	Dombivili, Mumbai	33333344	9899889898	vaishali@sharanamshah.com

Diagram 8.4.9: Check box is ticked for deletion

Click Delete. The record is deleted from the database and the same is also reflected in the tabular GRID below as shown in diagram 8.4.10.


Customer Form - Windows Internet Explorer
http://localhost:8080/Customers/CustomerServlet
Google
Customer Form
Page
Tools
Save
Cancel
Delete
Internet
100%

Customer Form

First Name	
Last Name	
Address 1	
Address 2	
Phone Number	
Mobile Number	
Email Address	

Delete	Name	Address	Phone Number	Mobile Number	Email Address
☐	Sharanam Shah	Shivaji park, Mumbai	22222222	9898989898	sharanam@sharanamshah.com

Record deleted

Diagram 8.4.10: Record deleted from the Customer Form

The Book CDROM holds the complete application source code built using the NetBeans IDE for the following applications**:**

- Codespecs / Section 2 / Chapter08_Cds / **Customers**

The web application can be directly used by making appropriate changes [MySQL - username/password in the **Servlet**] and then compiling, building and executing it.