

# Web–Database Integration

---

## *Introduction*

Database systems enable you to create, store, organize, and manipulate data. By integrating databases and web sites you can open up new possibilities for data access and dynamic web content.

## *Objectives*

The goal of this workshop is to introduce participants to the tools and skills required for creating web–database applications. After today’s workshop, participants will be able to:

- Create a database and a table.
- Manipulate table data.
- Create web-based access to a database.
- Create database-driven web content.

## *Prerequisites*

*Web Authoring:* CGI Scripts or equivalent skills

### *Related Academic Computing Services workshops*

See the current issue of *Driver's Ed for the Information Superhighway* or go to <http://www.ukans.edu/acs/training> for the most recent information about dates, times, and places for these classes.

### **WEB AUTHORING: CGI SCRIPTS**

Use Perl to program dynamic, interactive Web sites, including support of HTML forms.

*Prerequisites:* *Web Authoring: Forms* and *Web Authoring: Introduction to Perl*.

*Class handout:* [www.ukans.edu/acs/docs/wkshop/cgi.shtml](http://www.ukans.edu/acs/docs/wkshop/cgi.shtml)

### **ACCESS: INTRODUCTION**

In this workshop, the Access modules are defined. You create a table, then use commands and menu features to create, save, index, sort, and edit a database table. Participants also filter fields and records from a table.

*Prerequisites:* Experience with the Windows environment.

This workshop requires registration. (See the registration information at [www.ukans.edu/cgiwrap/acs/training/geninfo.cgi](http://www.ukans.edu/cgiwrap/acs/training/geninfo.cgi).) There is a \$75 fee for non-KU participants.

*Class handout:* [www.ukans.edu/acs/docs/wkshop/Access-intro-handout.pdf](http://www.ukans.edu/acs/docs/wkshop/Access-intro-handout.pdf)

# Web–Database Integration

---

## DATABASES

A *database* is an organized collection of data. A *relational database* is a database in which the organization of the data incorporates the way data items are related to one another. In particular, data in a relational database is stored in one or more *tables*. A table (also called a *relation*) is a two-dimensional array of *columns* (also called *fields* or *attributes*) and *rows* (also called *records* or *tuples*). Relational databases are very versatile.

A database consisting of multiple tables can offer some advantages over a single-table database. Consider, for example, a database of student information including ID numbers, names, addresses, classes, and grades. A single-table database for this example might have these nine columns:

ID	last	first	street	city	state	zip	class	grade
:	:	:	:	:	:	:	:	:

Notice that for every class in which a student receives a grade, the **ID**, **last**, **first**, **street**, **city**, **state**, and **zip** for that student must be repeated in the table!

If the database has multiple, related tables, however, we can have one table for name and address information, and another for grade information:

ID	last	first	street	city	state	zip
:	:	:	:	:	:	:

ID	class	grade
:	:	:

Because the common **ID** column relates the two, we can use this to send a student's grade report using the name (**last**, **first**) and address (**street**, **city**, **state**, **zip**) from the first table and the **class** and **grade** data from the second. Notice that this eliminates a considerable amount of data duplication and opportunity for error. Notice also that the data can be used however we wish, without regard for the database structure; in our example, information is combined from both tables, and not all columns are utilized (only the name and address would appear on the envelope, for instance). Modification of the database structure is similarly flexible in the multiple-table relational model.

Relational databases are the predominate form of database in use today, and are the type on which this class focuses.

Of course, some ways of organizing data are better than others, and you face a number of choices when creating a database. What data is to be included? What will the columns be? How will they be divided into multiple tables? Which columns will the individual tables share with each other? These are important considerations, and, in general, it is vital that you plan your database design carefully. It should be capable of accommodating all the data you can foresee needing to store, but should not waste storage space on data you don't need. It should also be immune to errors or inconsistencies arising from changes made to data items, now or in the future.

Besides thoughtful consideration of the needs your database must meet, there are formal means of making (some) database design decisions, those of database normalization. A database that is normalized has certain structural characteristics that make it more efficient and less susceptible to data integrity problems than one that is not. There are multiple levels of normalization, the first three of which are sufficient for most common situations.

For a database to be in *first normal form*, there must be no repeating columns, each cell of the table must have only a single value, and each table row must be unique. Rows are guaranteed to be unique through the use of *primary keys*. A primary key is a column or group of columns whose values uniquely identify a row in a table.

For example, consider a database of retail clothing items with the following table:

item	colors	price	tax
T-shirt	red, blue	12.00	0.60
polo	red, yellow	12.00	0.60
T-shirt	red, blue	12.00	0.60
sweatshirt	blue, black	25.00	1.25

It is not in first normal form, both because the cells in the **colors** column have multiple items, and because there are duplicate rows, i.e., there is no primary key. Eliminating the duplicate row and splitting the colors into their own cells brings it to first normal form:

item	color	price	tax
T-shirt	red	12.00	0.60
T-shirt	blue	12.00	0.60
polo	red	12.00	0.60
polo	yellow	12.00	0.60
sweatshirt	blue	25.00	1.25
sweatshirt	black	25.00	1.25

Here, **item** and **color** together comprise the primary key. Notice that it is almost trivial to obtain first normal form.

*Second normal form* requires that, if the primary key is a combination of multiple columns, all non-key columns must depend on all components of the key. Put another way, no column can be dependent upon just one part of the primary key.

In our example, **price** and **tax** depend on **item**, but not **color**, so the table is not in second normal form. Splitting the table into two brings about second normal form:

item	color
T-shirt	red
T-shirt	blue
polo	red
polo	yellow
sweatshirt	blue
sweatshirt	black

item	price	tax
T-shirt	12.00	0.60
polo	12.00	0.60
sweatshirt	25.00	1.25

Notice that **item** and **color** still make up the primary key for the left table, and **item** is the primary key for the right table. Notice also that a table in first normal form whose primary key is a single column is automatically in second normal form as well.

*Third normal form* is attained when no non-key field depends upon a field other than the primary key.

In our example, **price** depends on **item**, the primary key, but **tax** depends on **price**, not **item**. This is called a *transitive dependency*, and must be eliminated to bring the database into third normal form. Splitting the table once more does the trick:

item	color
T-shirt	red
T-shirt	blue
polo	red
polo	yellow
sweatshirt	blue
sweatshirt	black

item	price
T-shirt	12.00
polo	12.00
sweatshirt	25.00

price	tax
12.00	0.60
25.00	1.25

---

## DATABASE MANAGEMENT SYSTEMS

The program or set of programs used to create the database and provide access to it is often called a *database management system*, or *DBMS*. This is also sometimes referred to as the *database server* or *database engine*. There is a wide variety of DBMSes available to meet a wide variety of needs.

The tool or tools used to interact with a database, e.g., for data entry or querying, may be part of the DBMS or separate from it. They may run on the same computer as the DBMS or on other computers on a network. The latter configuration is known as a *client/server system*, in which the front-end programs are the *clients*, and they communicate over the network with the database *server*. These are *two-tier* database systems.

Other terms frequently applied to databases are *database application*, which most often refers to the collection of the database, the DBMS, and the client programs. *Database system* is sometimes used to encompass the entire system, including the database application and the supporting hardware. (These terms are often used somewhat loosely, however, so they may not be used with precisely these meanings in all contexts.)

Our aim in this class is to create database applications in which the custom clients are part of web pages.

---

## **STRUCTURED QUERY LANGUAGE**

*SQL*, which stands for *Structured Query Language*, is an industry standard language for creating and working with relational databases. Because it is not a procedural language, it is used from within DBMSes or programs written in other languages. Most, but not all, DBMSes are based on SQL, or provide some level of SQL support. The implementation of the SQL standard varies from one DBMS to another, with each implementing some subset of the standard (possibly in combination with additional capabilities outside the standard). This means that moving a database from one SQL DBMS to another may require rewriting the custom software in the database application, even though both DBMSes use SQL.

---

## **MINI SQL**

Mini SQL, or mSQL, is a lightweight SQL-based DBMS for UNIX that is free for certain noncommercial users, including education institutions. We will use mSQL in this class, along with Perl as our programming language. As was the case in the Web Authoring: CGI Scripts class, the concepts and techniques introduced in this class can be applied to similar endeavors regardless of the particular tools that you use. Though the concepts presented are general, choosing a particular tool set for this class provides the necessary framework for demonstrating specific examples.

---

---

## WORKING WITH MYSQL DATABASES

In order to make use of mSQL, you need to know how databases are created, how to create and delete tables, how to enter data into tables, how to make modifications to the data, and how to selectively retrieve data from a database.

### Creating databases

Database creation is done by the mSQL server administrator using the `msqladmin` program. The following UNIX shell command creates a database called **student\_grades** on the local system:

```
msqladmin create student_grades
```

### The mSQL monitor

The `msql` command invokes the mSQL monitor, an interactive interface to the mSQL server. The command itself, `msql`, must be followed by the name of the database to be accessed, like so:

```
msql student_grades
```

The `-h host` option can also be used to access an mSQL server running on a remote host, where *host* is the remote hostname or IP address. (If no remote host is specified, the monitor connects to the server on the local host.) The mSQL monitor can be used to submit SQL commands directly to the server. After an SQL command has been entered in the mSQL monitor, entering `\g` sends the command to the server. Entering `\q` at any time quits the monitor.

Note: The `msql` command file is in the **/usr/local/Hughes/bin** directory on Eagle and Falcon, and in **/usr/local/mSQL/bin** on Lark and Raven. You may need to add the appropriate directory to your `$PATH` environment variable or use the full pathname (**/usr/local/Hughes/bin/msql** on Eagle and Falcon, **/usr/local/mSQL/bin/msql** on Lark and Raven) to use the `msql` command.

In the examples below, SQL commands are presented in all caps for clarity and emphasis. SQL is not case sensitive, however.

### Creating tables

The `CREATE` statement is used to create tables within the database. With it, you specify the table name, the column names, and the type of data each column represents. For example, to create a table of students called **students**, one could use the following command:

```
CREATE TABLE students (
    ID char(6) not null,
    last char(25),
    first char(15),
    street char(25),
    city char(15),
    state char(2),
    zip char(9)
)
```

The name of the table, **students**, follows the CREATE TABLE statement. The list in parentheses is a list of the column names, separated by commas. Each column name is followed by its type; in this example, all the columns are of type char, for strings of characters. The numbers in parentheses after char represent the length the string is allowed to have. Finally, columns can be specified with the NOT NULL qualifier, which requires all rows to have data entered into such columns. This is a good idea for primary keys.

The data types available in mSQL are summarized below.

char (len)	String of characters.
text (len)	Variable length string of characters. The defined length is used to indicate the expected average length of the data.
int	Signed integer values.
real	Decimal or scientific notation real values.
uint	Unsigned integer values.
date	Date values in the format of DD-Mon-YYYY; e.g., 1-Jan-1998.
time	Time values stored in 24-hour notation in the format of HH:MM:SS.
money	A numeric value with two fixed decimal places.

## Entering data into tables

The INSERT statement is used to add data to the table. You can insert data into a row by providing a list of column names to indicate into which columns the respective data items are to be inserted. If you exclude the column names, you must specify entries for every column in order. For example:

```
INSERT INTO students
    (ID, last, first, street, city,
     state, zip)
VALUES ('765432', 'Smith',
        'Sarah',
        '23 Massachusetts',
        'Lawrence', 'KS',
        '66044')
```



```
INSERT INTO students
VALUES ('555588', 'Jones',
       'Kaitlyn',
       '2323 West 23rd',
       'Wichita', 'KS', '67214')
```

These commands would insert rows into a table called **students**, resulting in something like this:

ID	last	first	street	city	state	zip
:	:	:	:	:	:	:
765432	Smith	Sarah	<small>23 Massachusetts</small>	Lawrence	KS	66044
555588	Jones	Kaitlyn	<small>2323 West 23<sup>rd</sup></small>	Wichita	KS	67214
:	:	:	:	:	:	:

## Retrieving data from tables

The SELECT statement is used to extract data from the database. It provides the power and versatility to retrieve exactly the information you want from the database. This may mean viewing the contents of an entire table, viewing a specific row, restricting the view to only certain columns, or viewing entries that meet a specified condition.

For example, to simply view an entire table, you can use:

```
SELECT * FROM students
```

The \* is a wildcard character that means “everything,” so in this example all columns in the table called **students** will be listed. Because no restrictions were placed on the rows to be returned, all rows will be listed.

WHERE clauses can be used to select particular rows by specifying one or more conditions that must be met. For example, the command

```
SELECT * FROM students
WHERE city = 'Wichita'
```

will display only those rows with the value "Wichita" in the **city** column. Other operators can be used to specify conditions other than equality. Available operators are <, >, =, <=, >=, <>, LIKE, RLIKE, CLIKE, and SLIKE. The last four operators are for regular expression-style pattern matching. LIKE is the standard SQL regular expression operator, which uses the following special characters:

- Matches any single character.
- % Matches 0 or more characters of any value.
- \ Escapes special characters (e.g., \% matches a literal % and \\ matches \). All other characters match themselves.

CLIKE is a standard LIKE operator that ignores case.

RLIKE is a more complete regular expression operator than that offered by standard SQL, providing more of the power of UNIX regular expression syntax. It uses the following special characters:

- . The dot character matches any single character.
- ^ When used as the first character in a regex (regular expression), the caret character forces the match to start at the first character of the string.
- \$ When used as the last character in a regex, the dollar sign forces the match to end at the last character of the string.
- [] By enclosing a group of single characters within square brackets, the regex will match a single character from the group of characters. If the ] character is one of the characters you wish to match you may specify it as the first character in the group without closing the group (e.g., [] abc] would match any single character that was ], a, b, or c). Ranges of characters can be specified within the first group using the “first-last” syntax (e.g., [a-z0-9] would match any lowercase letter or a digit). If the first character of the group is the ^ character, the regex will match any single character that is not contained within the group.
- \* If any regex element is followed by a \*, it will match zero or more instances of the regular expression. To match any string of characters you would use .\* and to match any string of digits you would use [0-9]\*.

The SLIKE operator provides a way of matching a value that sounds like the specified value.

The ORDER BY clause can be used to sort the output of a SELECT statement by the values in one or more columns. If more than one column is specified (separated by commas), the data is sorted by the first specified column, then (within that order) by the next, and so on. For example, if our student table included the names Sarah Smith, Kaitlyn Jones, and Brittney Jones,

```
SELECT last, first FROM students
       ORDER BY last, first
```

would give

```
+-----+-----+
| last  | first  |
+-----+-----+
| Jones | Brittney |
| Jones | Kaitlyn  |
| Smith | Sarah    |
+-----+-----+
```

The DESC attribute can be added after any column name to change sorting on that column from ascending to descending order, e.g.:

```
SELECT last, first FROM students
       ORDER BY last DESC, first
```

This would give

```
+-----+-----+
| last  | first  |
+-----+-----+
| Smith | Sarah  |
| Jones | Brittnay |
| Jones | Kaitlyn |
+-----+-----+
```

A list of column names can follow the SELECT statement (instead of \*) to restrict the columns returned, as seen in the last two SELECT statement examples. This can also be used to *join* information from multiple tables, by listing columns from more than one table. When this is done, each column name is preceded by the name of the table to which it belongs followed by a dot (.). With the **students** and **grades** tables of our examples, we could get a grade report for Brittnay Jones (ID number 510958) with a SELECT command like

```
SELECT students.last, students.first,
       grades.class, grades.grade
FROM students, grades
WHERE students.ID =
       grades.ID
AND students.ID = '510958'
```

This would give

```
+-----+-----+-----+-----+
| last  | first  | class  | grade |
+-----+-----+-----+-----+
| Jones | Brittnay | MJTA 410 | A      |
+-----+-----+-----+-----+
```

The SELECT statement in mSQL supports the following features, most of which have been covered in this section:

1. Relational joins among multiple tables
2. Table aliases
3. DISTINCT row selection for returning unique values
4. ORDER BY clauses for sorting
5. Normal SQL regular expression matching
6. Enhanced regular expression matching including case insensitive and soundex
7. Column-to-column comparisons in WHERE clauses
8. Complex conditions

The formal definition of the syntax for mSQL's SELECT statement is:

```
SELECT [table.] column[,
        [table.] column ]...
FROM table [ = alias] [,
        table [ = alias] ]...
[ WHERE [table.] column OPERATOR VALUE
[ AND | OR [table.] column OPERATOR
        VALUE]... ]
[ ORDER BY [table.] column [DESC] [,
        [table.] column [DESC]... ]
```

## Deleting rows

The DELETE statement is used to remove one or more rows from a table. The selection of rows to be removed is based on the same WHERE clause used by the SELECT statement. For example:

```
DELETE FROM students
        WHERE ID = '555588'
```

## Modifying table data

The UPDATE statement is used to modify data that is already in a table. New values are SET in rows meeting the condition specified in the WHERE clause. For example:

```
UPDATE grades
        SET grade = 'A'
        WHERE ID = '555588' AND
        class = 'MJTA 410'
```

This command would result in something like:

ID	class	grade
:	:	:
555588	MJTA 410	A
:	:	:

## Summary of mSQL commands

The basic SQL commands in mSQL are CREATE, INSERT, SELECT, DELETE, and UPDATE.

In the mSQL monitor, \g executes an SQL command that has been entered, and \q quits.

## Exercises

To start with the workshop examples, you'll need to log into `raven.cc.ukans.edu`.

1. Telnet to `raven.cc.ukans.edu`
2. Log into `classX`, where *X* is replaced by a number, yielding `class1`, `class2`, ..., `class15`. Number assignments and passwords will be provided by the instructor.

### *Creating and entering data into a table*

The first exercise will be to create a table in the database **workshop**.

- Create a table whose name is the username you are using, e.g., `class1`, `class2`, etc. Define it to have columns as follows:

Column name	Column type
<code>last</code>	<code>char(25)</code>
<code>first</code>	<code>char(15)</code>

- Add some of the following information to the table, as time allows. Be sure to add at least one row, but do not worry about adding them all. Feel free to add a row with your own name as well.

<b>last</b>	<b>first</b>
Jensen	Jan
Wiley	William
Bailey	Bill
Brown	Beth

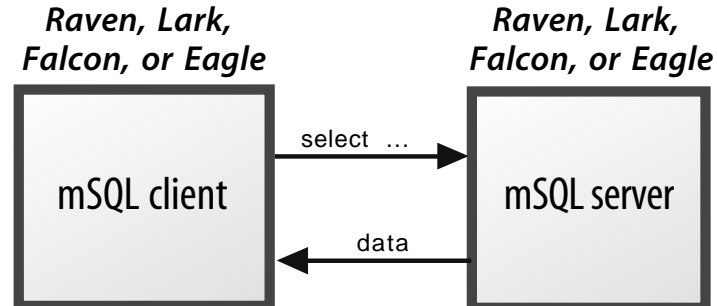
- Select the whole table.

### *Using the SELECT statement*

The second exercise demonstrates some different uses of the `SELECT` command. It uses a table called **textbook** that has already been set up. Enter each of the following commands and observe the output.

- `select * from textbook`
- `select * from textbook where title like 'A%'`
- `select * from textbook where required = 1`
- `select * from textbook where required <> 1`
- `select * from textbook where published > '1-Jan-1988'`
- `select * from textbook where price > 10.00 and required = 1`
- `select * from textbook where required = 1 and class = 101`
- `select * from textbook where class = 101 or class = 102`
- `select title from textbook`
- `select title, author, required from textbook`
- `select title from textbook where required = 1`
- `select title, price from textbook where class = 101`

The diagram below illustrates the components of the database application as it has been used up to this point.



### Importing tables

mSQL includes the `msqlimport` utility, which loads flat ASCII data files into mSQL database tables. Source files can be formatted using any character as the column separator, with line breaks delimiting the rows. Enter `msqlimport` at the UNIX shell prompt for usage information.

### Removing tables

The `DROP` statement is used to remove tables from databases. The syntax of the `DROP` statement is as follows:

```
DROP TABLE table_name
```

---

---

## ACCESSING DATABASES FROM CGI SCRIPTS

Once you're comfortable creating and managing databases with mSQL, you can integrate them with your web sites. When working with mSQL alone, you use the mSQL monitor program to enter mSQL commands and send them to the mSQL server. The basic idea behind integrating an mSQL database and a web site is to instead write a CGI script that will *construct* the mSQL commands and send them to the mSQL server itself.

This is straightforward in principle, and indeed the bulk of the work is simply implementing in the language of your CGI script (e.g., Perl) the logic required to produce the mSQL commands that you would otherwise enter into the mSQL monitor to accomplish the same things. The only outstanding requirement is the interface between your program and the mSQL server, which provides the mechanism for actually sending the mSQL commands to the server and receiving its responses.

When working with Perl, one preferred solution is the Perl *Database Interface (DBI)*. As defined by its creator, "DBI is a database access application programming interface (API) for the Perl language. The DBI API specification defines a set of functions, variables, and conventions that provide a consistent database interface independent of the actual database being used." It provides a simple means of embedding database commands in your Perl programs, and handles all the details of communicating those commands to whatever database server you are using (mSQL, in our case). The server-specific details are handled by a database driver (DBD), which DBI loads and uses to translate its server-independent functions to the necessary server-specific commands. The DBD for mSQL is *DBD::mSQL*. Both DBI and DBD::mSQL come in the form of freely available Perl modules.

### Using DBI

To enable the functionality of DBI in your script, include the `use DBI ;` command, which imports the DBI module.

### Connecting to a database

DBI is object-oriented, and provides a method called `connect`, which returns a handle to the specified database. This database handle provides the means for your program to interact with the database.

The `connect` method requires a data source string as its argument. This string must begin with `dbi :` followed by the name of the driver, without the `DBD::` portion of the module name, e.g., `dbi :mSQL`. This is followed by driver-specific information; in the case of `DBD::mSQL`, this should include a database name, hostname, and port number, delimited by colons. If the `mSQL` server is running locally on a default port, `localhost` and `1114` can be used for the hostname and port number, respectively, or these portions of the data source string can be left blank. For example, a Perl script that utilizes a database on the local host called **classX** could begin as follows:

```
#!/usr/local/bin/perl

use DBI;

$datasource = "dbi:mSQL:workshop";

$database_handle =
    DBI->connect ($datasource) ;
```

#### Issuing queries and processing results

The database handle's `prepare` method is used to prepare an SQL statement, returning a statement handle. The statement handle has an `execute` method that executes the SQL statement.

```
$sql_statement =
    "SELECT first, last FROM classX";

$statement_handle =
    $database_handle->prepare ($sql_statement) ;

$statement_handle->execute;
```

After you've executed a statement, you can use one of the statement's `fetch` methods to retrieve the rows that the SQL statement returned. There are several `fetch` methods, summarized below.

<code>fetchrow_arrayref</code>	Fetches the next row of data and returns a reference to an array holding the field values. If you assign the output of the method to the variable <code>\$ary_ref</code> , you can refer to the array with <code>@\$ary_ref</code> .
<code>fetch</code>	Alias for <code>fetchrow_arrayref</code> .
<code>fetchrow_array</code>	Fetches the next row of data and returns it as an array holding the field values.



`fetchrow_hashref` Fetches the next row of data and returns it as a reference to a hash containing field name and field value pairs. The names of the columns can be obtained by examining the `$statement_handle->{NAME}` property, where `$statement_handle` is the statement handle. A column's value can be obtained as `$hash_ref->{column_name}`.

`fetchall_arrayref` Used to fetch all the data to be returned from a prepared statement. It returns a reference to an array that contains one array reference per row (as returned by `fetchrow_arrayref`).

```
while (
    ($first, $last) =
    $statement_handle->fetchrow_array)
{
    print "$first $last \n";
}
```

When done with a statement handle, you should tell it you are finished by invoking its finish method.

```
$statement_handle->finish;
```

The database handle's `disconnect` method releases it:

```
$database_handle->disconnect;
```

## Exercise

Enter the preceding example into a file called `classX.pl` by typing

```
pico classX.pl
```

and entering the Perl code:

```
#!/usr/local/bin/perl

use DBI;

$datasource = "dbi:mSQL:workshop";

$database_handle = DBI->connect($datasource);

$sql_statement = "SELECT first, last FROM classX";

$statement_handle =
    $database_handle->prepare($sql_statement);

$statement_handle->execute;

while (($first, $last) = $statement_handle->fetchrow_array) {
    print "$first $last \n";
}

$statement_handle->finish;

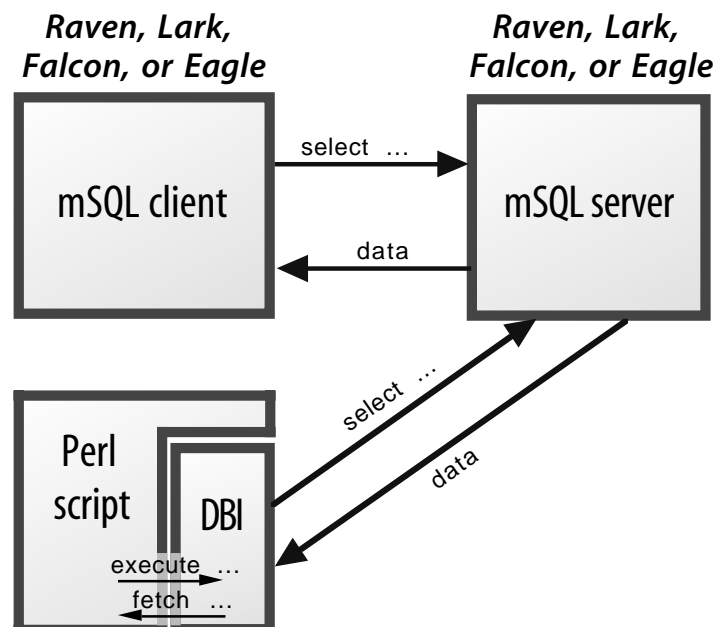
$database_handle->disconnect;
```

Save and enter

```
perl classX.pl
```

to run the program.

The diagram below illustrates the components of the database application as it now stands.



## Issuing updates

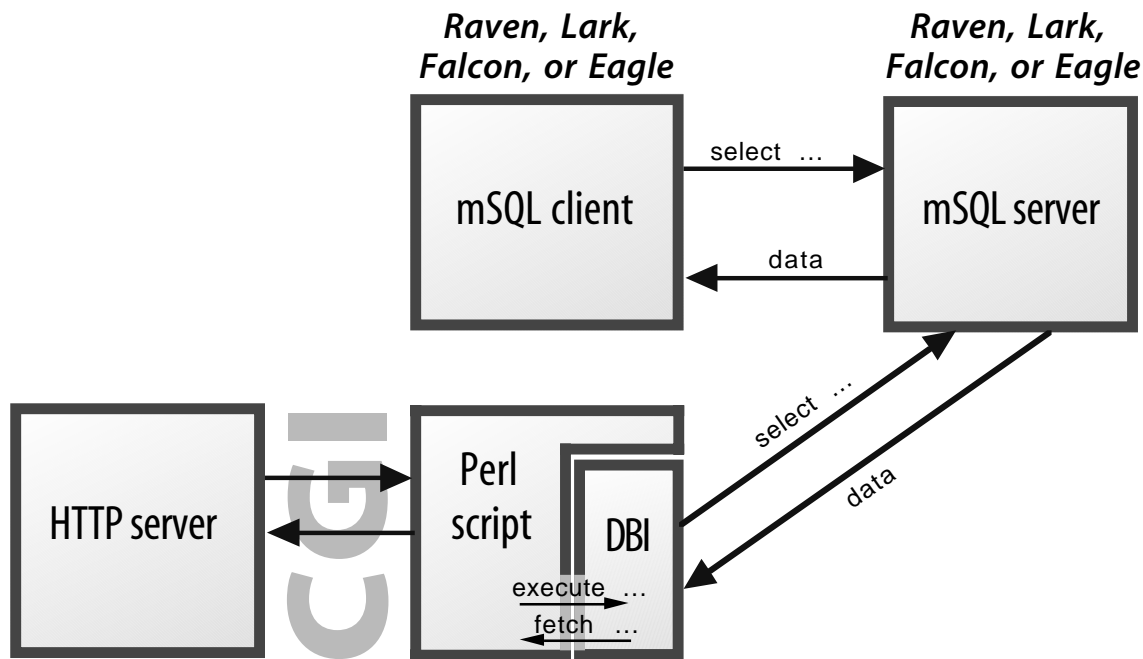
Since updates (i.e., UPDATE, INSERT, and DELETE statements) do not return any results, there is no need to prepare statement handles for them. For these, you can simply use the database handle's `do` method. For example:

```
$sql_statement =  
    "INSERT INTO classX " .  
    "VALUES ('Johnson', 'Marla')";  
  
$database_handle->do($sql_statement);
```

---

## EXERCISE

The next step is to implement a CGI script that is a client for the database, as diagrammed below.



Recall the Interest Form example from the Web Authoring: CGI Scripts class. In it an HTML form collects a name and email address from the user and passes it to a CGI script, which then generates a personalized message with the form data. We also looked at how the script could write the form data to a text-file database, and later read it back. In this exercise you'll create a similar form and use CGI scripts to store and read the form data to and from an mSQL database.

## Creating the form

The HTML document that contains the interest form is shown below:

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<HTML>
<HEAD>
<TITLE>Interest Form</TITLE>
</HEAD>
<BODY>
<H1>Interest Form</H1>
<P>
If you would like more information about the topics discussed on this
Web site, please enter your name and email address below.
<FORM action="http://raven.cc.ukans.edu/cgiwrap/classX/
myform.cgi" method="post">
<P>
<LABEL for="name">Please enter your name:</LABEL><BR>
<INPUT type="text" name="name" size="35" id="name">
<P>
<LABEL for="address">Please enter your email address:</LABEL><BR>
<INPUT type="text" name="address" size="35" id="address">
<P>
<INPUT type="submit" value="Send Address">
</FORM>
</BODY>
</HTML>

```

Notice that it is essentially the same as its counterpart from the Web Authoring: CGI Scripts class.

This document has been prepared for you and is stored as `/homea/classX/public_html/myform.html`. Please verify that it is in place, and look over it so that you understand what it does.

Adding the form data to the database

Next, create the CGI script that will service the form. This script stores the form data in the database and returns a personalized message. Enter

```

cd ~/public_html/cgi-bin
pico myform.cgi

```

Then enter this Perl script and save:

```
#!/usr/local/bin/perl

use CGI;
use DBI;

$form_data = new CGI;

$datasource = "dbi:mysql:workshop";
$database_handle = DBI->connect($datasource) or
    die "Could not connect:$DBI::errstr";

$unquoted_name = $form_data->param('name');
$name = $database_handle->quote($unquoted_name);
$address =
    $database_handle->quote($form_data->param('address'));

$insert_sql = "INSERT INTO formX " .
    "(name, address) " .
    "VALUES($name, $address)";

$database_handle->do($insert_sql) or
    die "Insert failed: " . $DBI::errstr;

$database_handle->disconnect;

print "Content-type: text/html\n\n";

print <<END_OF_MESSAGE;
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">
<TITLE>Return Page</TITLE>
<H1>Thank you for filling out my form!</H1>
<P>Thank you, $unquoted_name, for filling out my form!
<P><A href="show.cgi">See who else has signed up</A>
END_OF_MESSAGE

exit;
```

Enter

chmod 755 myform.cgi

to make it accessible.

### The database table

A table named formX, where X is the same number as in your username (e.g., user class2 has table form2) has been created for you in the database **workshop** for use with this script. It has a **name** column and an **address** column, both of type char (35).

## Testing the script

Open a web browser on your workstation and access <http://raven.cc.ukans.edu/~classX/myform.html> to test the script. You should be able to enter data into the form and see the customized thank you message. The link on the returned page won't yet function, of course, since **show.cgi** is not in place.

## Understanding **myform.cgi**

Let's go back through **myform.cgi** step by step to see how it works.

```
#!/usr/local/bin/perl
```

This is the standard Perl invocation.

```
use CGI;
use DBI;
```

These lines import the Perl modules used below. The CGI Perl module, included with most Perl distributions, uses objects to make it easy to create web forms and parse their contents. It was introduced in the Web Authoring: CGI Scripts class. DBI, of course, provides the interface to the database.

```
$form_data = new CGI;
```

This will parse the input (from both POST and GET methods) and store it into an object called `$form_data`.

```
$datasource = "dbi:mSQL:workshop";
$database_handle =
    DBI->connect($datasource) or
        die "Could not
            connect:$DBI::errstr";
```

These lines establish a connection to the database called **workshop**. `$datasource` is the data source string. `$database_handle` is the database handle. If it cannot be assigned (i.e., the assignment operation returns 0), the program exits and reports the error condition.

```
$unquoted_name =
    $form_data->param('name');
$name =
    $database_handle->quote($unquoted_name);
$address =
    $database_handle->
        quote($form_data->param('address'));
```

`$form_data->param('name')` returns the CGI query parameter called name; likewise `$form_data->param('address')` returns the address parameter. (*Object->method* is the standard Perl construction for invoking a method of an object.) In general, this is all that's needed to get the CGI parameter values. However, if a value includes a quote, e.g., O'Hara, it will present problems when used in an SQL statement later. The database handle routine `quote` takes care of this. We get the address value, prepare it with `quote`, and assign it to `$address`. Because we want to be able to use the name both as is and as cleaned up by `quote`, we first assign `$form_data->param('name')` to `$unquoted_name`, then run `quote` on that and assign the result to `$name`.

```
$insert_sql =
    "INSERT INTO formX " .
    "(name, address) " .
    "VALUES ($name, $address)";

$database_handle->do($insert_sql) or
    die "Insert failed: " .
        $DBI::errstr;

$database_handle->disconnect;
```

The SQL statement is constructed with the values obtained from the form, and executed with the `do` method. (The `.` simply represents concatenation in Perl.) If the `do` fails, the program exits and reports the error condition. The database handle is then released.

```
print "Content-type: text/html\n\n";

print <<END_OF_MESSAGE;
<!DOCTYPE HTML PUBLIC
    "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">
<TITLE>Return Page</TITLE>
<H1>Thank you for filling out my
form!</H1>
<P>Thank you, $unquoted_name, for
filling out my form!
<P><A href="show.cgi">See who else has
signed up</A>
END_OF_MESSAGE

exit;
```

An HTML document is returned, and the program exits.  
Notice that the HTML document incorporates  
\$unquoted\_name to personalize the page.

Reading information from the  
database

The script **show.cgi** completes the example. It reads the  
information from the recipients table and displays it in an  
HTML document. Enter

```
cd ~/public_html/cgi-bin  
pico show.cgi
```

Then enter this Perl script and save:

```
#!/usr/local/bin/perl  
  
use DBI;  
  
$datasource = "dbi:mysql:workshop";  
$database_handle = DBI->connect($datasource) or  
    die "Could not connect:$DBI::errstr";  
  
$sql_statement = "SELECT name, address " .  
    "FROM formX ";  
$statement_handle =  
    $database_handle->prepare($sql_statement);  
  
$statement_handle->execute or  
    die "Statement failed: " . $statement_handle->errstr;  
  
print "Content-type: text/html\n\n";  
print <<ENDSTART;  
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"  
    "http://www.w3.org/TR/html4/strict.dtd">  
<TITLE>Mailing List</TITLE>  
<H1>People who have requested information:</H1>  
<P>  
ENDSTART  
  
$row;  
while ($row = $statement_handle->fetch) {  
    print "$$row[0]" . " &lt;" . $$row[1] . "&gt;<BR>\n";  
}  
$statement_handle->finish;  
  
$database_handle->disconnect;  
  
print "<P><A href=\"/~classX/myform.html\">Return to  
    Interest Form</A>";  
exit;
```



Enter

```
chmod 755 show.cgi
```

to make it accessible.

### Testing the script

Open a local web browser and access <http://raven.cc.ukans.edu/~classX/myform.html> to test the script. After entering data into the form and viewing the customized thank you message, use the link on the returned page to test **show.cgi**.

### Understanding **show.cgi**

The program begins by invoking Perl, importing DBI, and establishing the database handle.

```
#!/usr/local/bin/perl
```

```
use DBI;
```

```
$datasource = "dbi:mysql:workshop";  
$database_handle =  
    DBI->connect($datasource) or  
        die "Could not  
            connect:$DBI::errstr";
```

The SQL statement is constructed next. Since it's a SELECT, which will return something, `$database_handle->prepare` is used on it to create a statement handle, `$statement_handle`. The statement is then executed.

```
$sql_statement =  
    "SELECT name, address " .  
    "FROM formX ";  
$statement_handle =  
    $database_handle->prepare($sql_statement);  
  
$statement_handle->execute or  
    die "Statement failed: " .  
        $statement_handle->errstr;
```

The following `print` statements get the HTML document started.

```

print "Content-type: text/html\n\n";
print <<ENDSTART;
<!DOCTYPE HTML PUBLIC
    "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">
<TITLE>Mailing List</TITLE>
<H1>People who have requested
information:</H1>
<P>
ENDSTART

```

Now the output of the SELECT statement needs to be inserted into the HTML document.

```

$row;
while ($row = $statement_handle->fetch) {
    print "$$row[0]" . " &lt;" .
        $$row[1] . "&gt;<BR>\n";
}
$statement_handle->finish;

```

```

$database_handle->disconnect;

```

The while statement loops over each row returned by the SELECT statement. Each row is fetched into \$row, and its values (the elements of the array @\$row) are printed within a line of HTML. When complete, the statement and database handles are released.

```

print "<P><A
    href=\"/~classX/myform.html\">
    Return to Interest Form</A>";
exit;

```

The remainder of the HTML document is printed, and the program exits.

---

#### EXAMPLE: STUDENTS.CGI

A more complete example of an HTML form-based database front-end can be found at <http://raven.cc.ukans.edu/cgiwrap/milo/students.cgi>. Source code for the **students.cgi** script can be viewed at <http://raven.cc.ukans.edu/~milo/cgi-bin/students.cgi>.

---

**EXAMPLE: MJTA410.CGI**

A final example can be found at <http://raven.cc.ukans.edu/cgiwrap/milo/mjta410.cgi>. A printout of the associated database's tables is provided in a separate document; a code listing for this script can be viewed at <http://raven.cc.ukans.edu/~milo/cgi-bin/mjta410.cgi>. This example highlights how databases can be used to enhance the functionality of web sites. In the previous example, the web site provided a front-end to the database—the web site added value to the database. In this example, the database powers the web site—the database adds value to the web site.

---

**OTHER TOOLS AND MODELS**

What has been presented here is but one of many means of integrating databases and web sites. It uses mSQL, Perl, and DBI (with DBD::mSQL) for the basic components of such an application: the DBMS, the CGI scripting language, and the database interface for that language. Many other combinations of tools can comprise these components, providing many possible solutions. There are also several tools whose capabilities essentially combine one or more of these components. Some HTTP servers accommodate scripts, servlets, and/or plug-ins directly, without the CGI. Others have built-in interfaces to DBMSes. Some web site design tools automatically provide database interfaces, or internalize databases. Some tools provide custom HTML tags that handle database interaction. And some DBMSes can be configured to serve their databases to the Web themselves. The options are many. Before settling on any one solution, you should shop around to be sure you find one that meets your needs.

Regardless of the solution you choose, this introduction should give you insight into how to build your own web–database application.

---

**RESOURCES****Databases**

- Codd, E. F. *The Relational Model for Database Management*. Version 2. Addison Wesley Longman, 1990. ISBN 0-201-14192-2.

<http://cseng.awl.com/bookdetail.qry?ISBN=0-201-14192-2&ptype=0>

**SQL**

- SQL FAQ

[http://epoch.CS.Berkeley.EDU:8000/sequoia/dba/montage/FAQ/SQL\\_TOC.html](http://epoch.CS.Berkeley.EDU:8000/sequoia/dba/montage/FAQ/SQL_TOC.html)

- SQL standards

[http://www.jcc.com/sql\\_stnd.html](http://www.jcc.com/sql_stnd.html)

- SQL tutorial

<http://w3.one.net/~jhoffman/sqltut.htm>

## mSQL

- mSQL home page

<http://www.hughes.com.au/>

- mSQL documentation library

<http://www.hughes.com.au/library/>

- Jepson, Brian, and David J. Hughes. *Official Guide to Mini SQL 2.0*. New York: John Wiley & Sons, Inc., 1998. ISBN 0-471-24535-6.

<http://catalog.wiley.com/ss/.272907809/title.cgi?0471245356>

<http://www.wiley.com/compbooks/catalog/24535-6.htm>

- mSQL mailing list

<http://hostingservices.net/msql/mailling-list.html>

- mSQL information/interfaces page

<http://www.hostingservices.net/msql/>

## DBD::mSQL

- DBD::mSQL readme

<http://www.perl.com/CPAN-local/modules/by-module/DBD/Msql-Mysql-modules-1.1828.readme>

- DBD::mSQL documentation

<http://www.hostingservices.net/DBI/mSQL.html>

## DBI

- DBI home page

<http://www.symbolstone.org/technology/perl/DBI/index.html>

- DBI FAQ

<http://www.symbolstone.org/technology/perl/DBI/doc/faq.html>

- DBI documentation and examples

<http://www.symbolstone.org/technology/perl/DBI/index.html#docs>

- Descartes, Alligator & Tim Bunce. *Programming the Perl DBI*. O'Reilly & Associates, Inc., 2000. ISBN 1-56592-699-4.

<http://www.oreilly.com/catalog/perl/dbi/>

- DBI mailing lists

<http://www.fugue.com/dbi>

<http://www.symbolstone.org/technology/perl/DBI/index.html#mailinglists>

## Related Perl

- Stein, Lincoln D. *Official Guide to Programming with CGI.pm*. John Wiley & Sons, Inc., 1998. ISBN 0-471-24744-8.

<http://www.wiley.com/compbooks/stein/>

<http://www.wiley.com/compbooks/catalog/24744-8.htm>

- Perl CGI.pm documentation

<http://www.perl.com/CPAN-local/doc/wwwman/CGI-pm/CGI.html>

[http://www.genome.wi.mit.edu/ftp/pub/software/WWW/cgi\\_docs.html](http://www.genome.wi.mit.edu/ftp/pub/software/WWW/cgi_docs.html)

- Perl CGI.pm examples

<http://www.genome.wi.mit.edu/WWW/examples/Ch9/>

- Object-oriented Perl documentation

<http://www.perl.com/CPAN-local/doc/manual/html/pod/perltoot.html>

---

## HELP

Academic Computing Services provides computing help in a variety of ways:

Quick phone help                      The Help Center 785/864-0200

Quick email help                      [question@ukans.edu](mailto:question@ukans.edu)

Consulting (Faculty/Staff)                      785/864-0410

Online documentation                      [www.ukans.edu/acs/docs](http://www.ukans.edu/acs/docs)

Training schedule online                      [www.ukans.edu/acs/training](http://www.ukans.edu/acs/training)

To receive automatic announcements of upcoming computer training, send the following message to the email address below:

address: [listproc@ukans.edu](mailto:listproc@ukans.edu)

message: SUB COMPUTER-TRAINING *your name*

*Note:* Substitute your real name for *your name* above, i.e., Jane Smith, not your login name.

©2000 The University of Kansas  
Academic Computing Services  
Web–Database Integration  
Prepared by Cole Robison  
785/864-0447  
[cole@ukans.edu](mailto:cole@ukans.edu)  
July 26, 2000

Comments? Evaluate this class online at <http://www.ukans.edu/acs/training/evaluation>.