**Connect** C# to **MySQL**

**Connect** C# to **MySQL** using **MySQL** **Connect**or/Net, Insert, Update, Select, Delete example, Backup and restore **MySQL** database from C# or .NET application

Introduction

I will create simple examples about the DML (Insert, Update, Select, Delete) throughout the document to show how to query the database using C#, and in the end I will show you how to backup your database and save it in a *.sql* file from our application, and how to restore it back.

Getting Started

Downloading **Connect**or/Net

First make sure you have downloaded and installed the **MySQL** **Connect**or/NET from the [**MySQL** official website](http://dev.mysql.com/downloads/connector/net/6.1.html). In this article, I will use the **Connect**or/NET version 6.1.

Creating the Database

Now let's create the database, and the table that we are going to query from our application later on.

From the command line, we start by **creating the database**:

create database **Connect**CsharpTo**Mysql**;

Then we select the **database to use** before creating the table:

use **Connect**CsharpTo**Mysql**;

And we **create the table** that we will query from our application:

create table tableInfo

(

id INT NOT NULL AUTO INCREMENT,

name VARCHAR(30),

age INT,

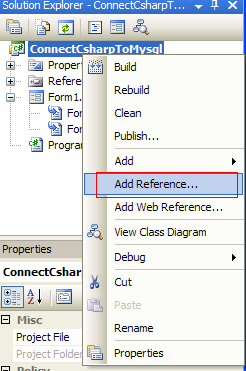
PRIMARY KEY(id)

);

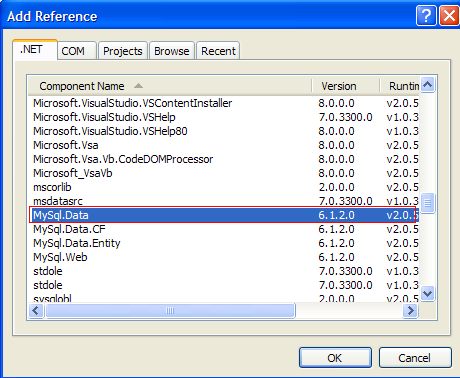
## Using the Code

### Adding Reference and Creating the MySQL Connector DLL from the Project

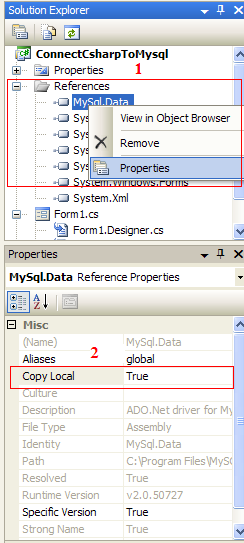
Before we start writing the code, we need to add the **mysql** Reference in our project. To do so, we right click our project name, and choose Add Reference:



Then we choose ***MySql***.Data from the list:



In order to use the application on other computers that don't have the **connect**or installed, we will have to create a DLL from the reference. To do so, we right click the reference name in our project, and set the copy local to true in its properties:



### Creating the Class

It's always a better idea to create a new class for connecting to the database and to separate the actual code from the code that will access the database. This will help keep our code neat, easier to read and more efficient.

We will start by adding the MySql Connector library:

//Add *MySql* Library

using MySql.Data.MySqlClient;

Then declaring and initializing the variables that we will use:

* connection: will be used to open a connection to the database.
* server: indicates where our server is hosted, in our case, it's localhost.
* database: is the name of the database we will use, in our case it's the database we already created earlier which is connectcsharptomysql.
* uid: is our MySQL username.
* password: is our MySQL password.
* connectionString: contains the connection string to connect to the database, and will be assigned to the connection variable.

Our class will look as follows:  
(Empty methods will be filled later on in this article.)

class DBConnect

{

private MySqlConnection connection;

private string server;

private string database;

private string uid;

private string password;

//Constructor

public DBConnect()

{

Initialize();

}

//Initialize values

private void Initialize()

{

server = "localhost";

database = "connectcsharptomysql";

uid = "root";

password = "";

string connectionString;

connectionString = "SERVER=" + server + ";" + "DATABASE=" +

database + ";" + "UID=" + uid + ";" + "PASSWORD=" + password + ";";

connection = new MySqlConnection(connectionString);

}

//open *connect*ion to database

private bool OpenConnection()

{

}

//Close *connect*ion

private bool CloseConnection()

{

}

//Insert statement

public void Insert()

{

}

//Update statement

public void Update()

{

}

//Delete statement

public void Delete()

{

}

//Select statement

public List <string> [] Select()

{

}

//Count statement

public int Count()

{

}

//Backup

public void Backup()

{

}

//Restore

public void Restore()

{

}

}

### Opening and Closing the Connection

We should always open a connection before querying our table(s), and close it right after we're done, to release the resources and indicate that this connection is no longer needed.  
Opening and closing a connection to the database is very simple, however, it's always better to use exception handling before opening a connection or closing it, to catch the errors and deal with them.

//open *connect*ion to database

private bool OpenConnection()

{

try

{

connection.Open();

return true;

}

catch (MySqlException ex)

{

//When handling errors, you can your application's response based

//on the error number.

//The two most common error numbers when *connect*ing are as follows:

//0: Cannot *connect* to server.

//1045: Invalid user name and/or password.

switch (ex.Number)

{

case 0:

MessageBox.Show("Cannot connect to server. Contact administrator");

break;

case 1045:

MessageBox.Show("Invalid username/password, please try again");

break;

}

return false;

}

}

//Close *connect*ion

private bool CloseConnection()

{

try

{

connection.Close();

return true;

}

catch (MySqlException ex)

{

MessageBox.Show(ex.Message);

return false;

}

}

### Working with DML (Insert, Update, Select, Delete)

Usually, Insert, update and delete are used to write or change data in the database, while Select is used to read data.  
For this reason, we have different types of methods to execute those queries.  
The methods are the following:

* ExecuteNonQuery: Used to execute a command that will not return any data, for example Insert, update or delete.
* ExecuteReader: Used to execute a command that will return 0 or more records, for example Select.
* ExecuteScalar: Used to execute a command that will return only 1 value, for example Select Count(\*).

I will start with Insert, update and delete, which are the easiest. The process to successfully execute a command is as follows:

1. Open connection to the database.
2. Create a MySQL command.
3. Assign a connection and a query to the command. This can be done using the constructor, or using the Connection and the CommandText methods in the MySqlCommand class.
4. Execute the command.
5. Close the connection.

//Insert statement

public void Insert()

{

string query = "INSERT INTO tableinfo (name, age) VALUES('John Smith', '33')";

//open *connect*ion

if (this.OpenConnection() == true)

{

//create command and assign the query and *connect*ion from the constructor

MySqlCommand cmd = new MySqlCommand(query, connection);

//Execute command

cmd.ExecuteNonQuery();

//close *connect*ion

this.CloseConnection();

}

}

//Update statement

public void Update()

{

string query = "UPDATE tableinfo SET name='Joe', age='22' WHERE name='John Smith'";

//Open *connect*ion

if (this.OpenConnection() == true)

{

//create *mysql* command

MySqlCommand cmd = new MySqlCommand();

//Assign the query using CommandText

cmd.CommandText = query;

//Assign the *connect*ion using *Connect*ion

cmd.Connection = connection;

//Execute query

cmd.ExecuteNonQuery();

//close *connect*ion

this.CloseConnection();

}

}

//Delete statement

public void Delete()

{

string query = "DELETE FROM tableinfo WHERE name='John Smith'";

if (this.OpenConnection() == true)

{

MySqlCommand cmd = new MySqlCommand(query, connection);

cmd.ExecuteNonQuery();

this.CloseConnection();

}

}

To execute a Select statement, we add a few more steps, and we use the ExecuteReader method that will return a dataReader object to read and store the data or records.

1. Open connection to the database.
2. Create a MySQL command.
3. Assign a connection and a query to the command. This can be done using the constructor, or using the Connection and the CommandText methods in the MySqlCommand class.
4. Create a MySqlDataReader object to read the selected records/data.
5. Execute the command.
6. Read the records and display them or store them in a list.
7. Close the data reader.
8. Close the connection.

//Select statement

public List< string >[] Select()

{

string query = "SELECT \* FROM tableinfo";

//Create a list to store the result

List< string >[] list = new List< string >[3];

list[0] = new List< string >();

list[1] = new List< string >();

list[2] = new List< string >();

//Open *connect*ion

if (this.OpenConnection() == true)

{

//Create Command

MySqlCommand cmd = new MySqlCommand(query, connection);

//Create a data reader and Execute the command

MySqlDataReader dataReader = cmd.ExecuteReader();

//Read the data and store them in the list

while (dataReader.Read())

{

list[0].Add(dataReader["id"] + "");

list[1].Add(dataReader["name"] + "");

list[2].Add(dataReader["age"] + "");

}

//close Data Reader

dataReader.Close();

//close *Connect*ion

this.CloseConnection();

//return list to be displayed

return list;

}

else

{

return list;

}

}

Sometimes, a command will always return only one value, like for example if we want to count the number of records, we have been using Select Count(\*) from tableinfo;, in this case, we will have to use the method ExecuteScalar that will return one value.

The process to successfully run and ExecuteScalar is as follows:

1. Open connection to the database.
2. Create a MySQL command.
3. Assign a connection and a query to the command. This can be done using the constructor, or using the Connection and the CommandText methods in the MySqlCommand class.
4. Execute the command.
5. Parse the result if necessary.
6. Close the connection.

//Count statement

public int Count()

{

string query = "SELECT Count(\*) FROM tableinfo";

int Count = -1;

//Open *Connect*ion

if (this.OpenConnection() == true)

{

//Create *Mysql* Command

MySqlCommand cmd = new MySqlCommand(query, connection);

//ExecuteScalar will return one value

Count = int.Parse(cmd.ExecuteScalar()+"");

//close *Connect*ion

this.CloseConnection();

return Count;

}

else

{

return Count;

}

}

### Backup and Restore the Database

Before I show you how to backup the database from our application, I will explain a little bit about processes, commands, arguments and the input and output.  
Usually, to backup a MySQL database from the command line, we write the following:

mysqldump -u username -p password -h localhost ConnectCsharpToMysql > "C:\Backup.sql"

and to restore it, we write:

mysql -u username -p password -h localhost ConnectCsharpToMysql < "C:\Backup.sql"

The following commands can be divided as such:

* *mysql* and *mysql*dump are the **filename** or the executable file.
* -u username -p password -h localhost are the **arguments**.
* > "C:\Backup.sql" is where the output is directed.
* < "C:\Backup.sql" is where the input is directed.

Now that we know how the command is divided, we can start implementing it in our application.

In C# and .NET applications, starting a process is easy. First we add the library:

using System.Diagnostics;

Then we launch an application, such as Internet Explorer:

Process.Start("IExplore.exe");

ProcessStartInfo is used in conjunction with Process, to setup the process before it starts.  
For example, to start Internet Explorer with arguments, we write the following:

ProcessStartInfo psi = new ProcessStartInfo();

psi.FileName = "IExplore.exe";

psi.Arguments = "www.codeproject.com";

Process.Start(psi);

To write our output to a file or read our input, we can use the RedirectStandardInput and RedirectStandardOutput properties in the ProcessStartInfo component to cause the process to get input from or return output to a file or other device. If we use the StandardInput or StandardOutput properties on the Process component, we must first set the corresponding value on the ProcessStartInfo property. Otherwise, the system throws an exception when we read or write to the stream.

Now back to our application, to backup the database, we will have to set the RedirectStandardOutput to true, and read the output from the process into a string and save it to a file.

//Backup

public void Backup()

{

try

{

DateTime Time = DateTime.Now;

int year = Time.Year;

int month = Time.Month;

int day = Time.Day;

int hour = Time.Hour;

int minute = Time.Minute;

int second = Time.Second;

int millisecond = Time.Millisecond;

//Save file to C:\ with the current date as a filename

string path;

path = "C:\\MySqlBackup" + year + "-" + month + "-" + day +

"-" + hour + "-" + minute + "-" + second + "-" + millisecond + ".sql";

StreamWriter file = new StreamWriter(path);

ProcessStartInfo psi = new ProcessStartInfo();

psi.FileName = "mysqldump";

psi.RedirectStandardInput = false;

psi.RedirectStandardOutput = true;

psi.Arguments = string.Format(@"-u{0} -p{1} -h{2} {3}",

uid, password, server, database);

psi.UseShellExecute = false;

Process process = Process.Start(psi);

string output;

output = process.StandardOutput.ReadToEnd();

file.WriteLine(output);

process.WaitForExit();

file.Close();

process.Close();

}

catch (IOException ex)

{

MessageBox.Show("Error , unable to backup!");

}

}

To restore the database, we read the .sql file and store it in a string, then set the RedirectStandardInput property to true, and write the input from the string to the process.

//Restore

public void Restore()

{

try

{

//Read file from C:\

string path;

path = "C:\\MySqlBackup.sql";

StreamReader file = new StreamReader(path);

string input = file.ReadToEnd();

file.Close();

ProcessStartInfo psi = new ProcessStartInfo();

psi.FileName = "mysql";

psi.RedirectStandardInput = true;

psi.RedirectStandardOutput = false;

psi.Arguments = string.Format(@"-u{0} -p{1} -h{2} {3}",

uid, password, server, database);

psi.UseShellExecute = false;

Process process = Process.Start(psi);

process.StandardInput.WriteLine(input);

process.StandardInput.Close();

process.WaitForExit();

process.Close();

}

catch (IOException ex)

{

MessageBox.Show("Error , unable to Restore!");

}

}