

# **Jupiter**

## ***Information Technology***

### **Tutorial**

First steps in setting up tables in mySQL

**21 February 2003**

18 Church St  
Swansea. SA4 3EA  
United Kingdom

+44 (0)1792 875340  
info@jup-it-er.co.uk  
www.jup-it-er.co.uk

Copyright © 2003 – Jupiter Informaion Technology – All rights reserved

## Introduction

In this tutorial we will find out how to create a database, then create a table and populate it with data. We will also look at one or two issues that may occur, and look at ways of solving them.

The brief is to create a user database that can be accessed via a web browser for user authentication. This is effectively the means to provide a gateway into a secure or subscription only web site.

The site owner wants the user to enter his name (8 characters), a password (up to 12 characters), his social security number, and an e-mail address.

Since we can reasonably assume that the social security number will be unique, we will use this as the primary key field for searching, sorting and indexing. Also, since an e-mail address can be variable length, we will allow 255 characters. It is probably a lot more than we want, but we can always reduce this to a more sensible figure (particularly on a large database with many records - we would be much more strict with usage of our storage space).

## Create a database

So, let us create a new database, `jtest`, and see what happens.

We could just issue the create database command, and everything would work fine. A new database will be created, and we are all set to create tables and populate the database and link up to our browser and invite users to log on to our site and .

Whoa! Let's just slow down a bit. OK. We could just issue the create database command, but there may already be a database with the same name on our server. What would happen then? Well, we could easily overwrite the existing database, if we are not careful. If it contains payroll data, for example, you and your colleagues won't receive your pay cheque this month, and it will all be your fault when all those families get evicted from their homes . well, actually, it's not as bad as that. MySQL won't allow us to do it, so we are covered. However, good practice suggests that we cannot rely on the DBMS (Data Base Management System) to keep us out of trouble. MySQL may protect us, but you could easily come across another DBMS using the SQL language that doesn't

offer protection. In any case, if you are issuing the command in a script, when MySQL traps an error the script will stop dead in its tracks.

Here is output from a session that shows what happens:

```
mysql> create database jtest;  
ERROR 1007: Can't create database 'jtest'. Database exists
```

Note the error message generated by MySQL.

So, how can we ensure that we are careful? Well, we check to see if a database of the same name exists, and then only if one does not exist do we create our database.

If it does exist already, then MySQL just ignores the command without any problems or any fuss (which is what we want).

So, now let's issue the command, below. You can change `jtest` to whatever you like. Note the semi-colon at the end of the line. This tells MySQL that the command can now be executed. If you omit the semi-colon, MySQL will display a further prompt, while expecting further input.

```
create database if not exists jtest;
```

No tables are created yet. Creating a database just prepares the ground.

Here are the results of issuing this command. Contrast it with the previous command. MySQL's response is now calm and controlled. If this is in a script, the script will continue executing normally.

```
mysql> create database if not exists jtest ;  
Query OK, 0 rows affected (0.01 sec)
```

## Create a table

So, now we can create a table with fields. (Primary key field is `ss_num`). Note: `ss_num` is 11 characters wide. This allows for the dash between the groups of numbers, and is also big enough (with space to spare) for UK Social Security numbers. If this database was not international, we might consider removing the dashes (and reducing the size of the field) and use formatting commands on a report. However, with international variances on the data format, it would be an onerous task to allow for all possible variations.

You might consider adding an additional field for country, so that data such as USA, UK, CAN, AUS, etc can be entered. This can then be combined with `ss_num` to provide a composite key, such that you search, sort and index on something like:

**UKXX999999X** or **USAXXX-XX-XXXX** or whatever.

NOTE: The country code could be suffixed or prefixed. It does not matter, so long as the resulting key is unique, and the scheme is consistent.

This is because it is conceivable that a US social security number could be identical to a SS number from some other country. We cannot now guarantee uniqueness on the SS number alone, and a primary key value must always be unique.

```
create table jtest.user (name varchar(8), password varchar(12),
ss_num varchar(11) not null, email varchar(255), primary key
ss_num);
```

Our table description now looks like the following:

Field	Type	Null	Key	Default	Extra
name	varchar(8)	YES		NULL	
password	varchar(12)	YES		NULL	
ss_num	varchar(11)		PRI		
email	varchar(255)	YES		NULL	

Notice that 3 fields are NULL, while the Primary Key is NOT NULL. You cannot index columns that have NULL values (Yes. NULL is a value. See the MySQL manual - [manual.html#NULL\\_values](#)), so we have to declare `ss_num` NOT NULL, since this is our primary key field and we will want, eventually, to index on this field. You also cannot insert NULL values into an indexed column. More on this later.

### Entering Data

There are two alternative methods for entering data

Enter a single row of data with `INSERT INTO tablename (col1, col2, ..., colN) VALUES ( data, data, ..., data);` or import a lot of data from a file using `LOAD DATA INFILE 'filename' INTO tablename` etc;

If you have a lot of data to enter you could create a script, like `ADD_DATA.SQL`, with multiple `INSERT` commands ...

Enter the following command:

```
insert into user (name, password, ss_num, email)
values ("fred", "pword", "123-34-5678", "fred@fanakerpan.com");
```

which gives the following output :

```
+-----+-----+-----+-----+
| name | password | ss_num      | email                |
+-----+-----+-----+-----+
| fred | pword    | 123-34-5678 | fred@fanakerpan.com |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

However, use of LOAD DATA INFILE command is recommended for inputting a large number of records into a MySQL Database table. For a start, there is less typing involved. You only need to type out the data, rather than having to type 'insert into etc, etc .'.

Also, it is easier to validate data before data is entered into the database. Any errors can be discovered and corrected before entry. This can be done manually by an operator / data entry clerk (impractical for a large number of records, and error-prone), or it could be done automatically with a dedicated program.

In the context that many mySQL databases are used, it might be appropriate to create a PHP or PERL script for validation, or write a C program. Whichever language is used, it is a trivial exercise to do this which would pay large dividends in:

- a. increased accuracy,
- b. data integrity,
- c. less time spent recovering and correcting data,
- d. increased customer satisfaction,
- e. greater efficiency, and
- f. greater professionalism

By studying the data example below (contents of userdata.txt) it is important to realise the default field delimiter is the TAB character, and the default record delimiter is the NewLine character. So, it is straightforward to enter data into the data file.

It is possible to change the default delimiters by using various parameters in the LOAD DATA INFILE command. See the MySQL manual, [manual.html#LOAD\\_DATA](#) for a fuller description of the syntax.

Here are the contents of userdata.txt:

```
Fred password 123-23-2345 fred.fanakerpan@fishworld.com
Joey secure 234-12-1234 joey@budgieworld.com
```

When using the command, the author found that he needed to copy the userdata.txt data file into the mysql/data/jtest directory before the command would execute. On a production system, the datafile should be saved to a directory in the path. See your sysadmin for information on your systems path.

Issue the following MySQL command:

```
load data infile "userdata.txt" into table user;
```

The output from an example session is shown below.

```
mysql> load data infile "userdata.txt" into table user;
Query OK, 2 rows affected (0.01 sec)
Records: 2 Deleted: 0 Skipped: 0 Warnings: 0
mysql> select * from user
-> ;
+-----+-----+-----+-----+
| name | password | ss_num      | email                                |
+-----+-----+-----+-----+
| fred | pword     | 123-34-5678 | fred@fanakerpan.com                 |
| fred | password  | 123-23-2345 | fred.fanakerpan@fishworld.com       |
| joey | secure    | 234-12-1234 | joey@budgieworld.com                 |
+-----+-----+-----+-----+
3 rows in set (0.00 sec)
```

### Doing it

So, putting it all together, the MAKETABLE.SQL script looks like the following:

```
create database if not exists jtest;
create table jtest.user (name varchar(8), password varchar(12), ss_num
varchar(11) not null, email varchar(255), primary key (ss_num));
```

and the ADD\_DATA.SQL script looks like this:

```
insert into user (name, password, ss_num, email) values ("fred", "pword",
"123-34-5678", "fred@fanakerpan.com");
insert into user (name, password, ss_num, email) values ("joey", "secret",
"234-56-6789", "joey@budgieworld.com");
```

Incidentally, you do not have to put data into all the fields in one session. Sometimes, we only have partial information. As an example, look at this session:

```
mysql> insert into user (name, password) values ("laurence","letmein");
Query OK, 1 row affected (0.01 sec)

mysql> select * from user;
+-----+-----+-----+-----+
| name      | password | ss_num      | email                                |
+-----+-----+-----+-----+
| fred      | pword    | 123-34-5678 | fred@fanakerpan.com                 |
| fred      | password | 123-23-2345 | fred.fanakerpan@fishworld.com       |
| joey      | secure   | 234-12-1234 | joey@budgieworld.com                 |
| laurence  | letmein  |              | NULL                                  |
+-----+-----+-----+-----+
4 rows in set (0.01 sec)
```

Notice the NULL value in the email field (and the absence of NULL in ss\_num).

To run the scripts, start mySQL with something like:

```
mysql -h host -u user databasename
```

On some systems, you may need to include the path , as in:

```
/usr/mysql/bin/mysql/mysql -h host -u user databasename
```

for example. You should ask your sysadmin for the actual path.

From the list of commands (below), we can see that if we type

```
source or \. Scriptname
```

then we can run the scripts whenever we want.

MySQL commands:

Note that all text commands must be first on line and end with ';'.

help	(\h)	Display this help.
?	(\?)	Synonym for `help`.
clear	(\c)	Clear command.
connect	(\r)	Reconnect to the server. Optional arguments are db and host.
ego	(\G)	Send command to mysql server, display result vertically.
exit	(\q)	Exit mysql. Same as quit.
go	(\g)	Send command to mysql server.
notee	(\t)	Don't write into outfile.
print	(\p)	Print current command.
quit	(\q)	Quit mysql.
rehash	(\#)	Rebuild completion hash.
source	(\.)	Execute a SQL script file. Takes a file name as an argument.
status	(\s)	Get status information from the server.
tee	(\T)	Set outfile [to_outfile]. Append everything into given outfile.
use	(\u)	Use another database. Takes database name as argument.