

PHP Login script tutorial

Learn to create a simple login system with PHP and a MariaDB script, this tutorial is easy to follow, teach you step by step.

Overview

In this tutorial, we create 6 php files for testing our code.

- main_login.php
- checklogin.php
- login_success.php
- logout.php
- register.php <- we will look at this later
- process_register.php <- we will look at this later

Steps

1. Modify database connection, and create database and user. Should already be in place.
2. Create table "members" in database "blog".
3. Create file main_login.php.
4. Create file checklogin.php.
5. Create file login_success.php.
6. Create file logout.php.
7. Create file register.php.
8. Create file process_register.php.

Step 1: Configure Database and Create Database and User If Necessary

NOTE: If you haven't done so, ensure you have installed the php-mysqli drivers via:

```
apt install php-mysqli
```

Historically, and on less secure systems like ApacheFirends XAMPP we could use the root access via PHPMYAdmin to create a database and a user with full privileges. That option is not available in a more hardened version of a LAMP stack running on recent versions of Debian. As such, we need to log into our VM via a command line (consider putty so you can copy/paste) and execute the following. **Don't forget the semi colons at the end of the SQL statements, it's how MySQL executes.**

First, we need to modify MariaDB to allow remote connections. Rather than deploying a package like PHPMYAdmin to your web server for database management, we are going to create a remote database connection and database users that allows remote management. First thing, we need to modify our database server to allow this connection.

We need to log in as **root** to make the following changes. Modify the server config as below:

```
nano /etc/mysql/mariadb.conf.d/50-server.cnf
```

You need to change the address you support to something that makes sense. Most will set it to

```
bind-address = 0.0.0.0
```

This can introduce a vulnerability that allows anyone to connect, something a secure server doesn't need. We will address this with network connections in the real world, blocking traffic at the router level, and is out of scope to this course. We will help with this by allowing specific access to databases via the built in database management tools.

This allows both localhost access and our host only network. Should look like the following:

```
GNU nano 3.2 50-server.cnf
#
# These groups are read by MariaDB server.
# Use it for options that only the server (but not clients) should see
#
# See the examples of server my.cnf files in /usr/share/mysql

# this is read by the standalone daemon and embedded servers
[server]

# this is only for the mysqld standalone daemon
[mysqld]

#
# * Basic Settings
#
user                = mysql
pid-file            = /run/mysqld/mysqld.pid
socket              = /run/mysqld/mysqld.sock
#port               = 3306
basedir             = /usr
datadir             = /var/lib/mysql
tmpdir              = /tmp
lc-messages-dir     = /usr/share/mysql
#skip-external-locking

# Instead of skip-networking the default is now to listen only on
# localhost which is more compatible and is not less secure.
bind-address        = 0.0.0.0_

#
# * Fine Tuning
#
[ Wrote 133 lines ]
^G Get Help  ^O Write Out  ^W Where Is   ^K Cut Text   ^J Justify    ^C Cur Pos    M-U Undo
^X Exit      ^R Read File  ^N Replace    ^U Uncut Text ^T To Spell   ^G Go To Line M-E Redo
```

Ctrl + O to save (write out) and Ctrl + X to exit. Left control key.

Now restart MariaDB server service with the following:

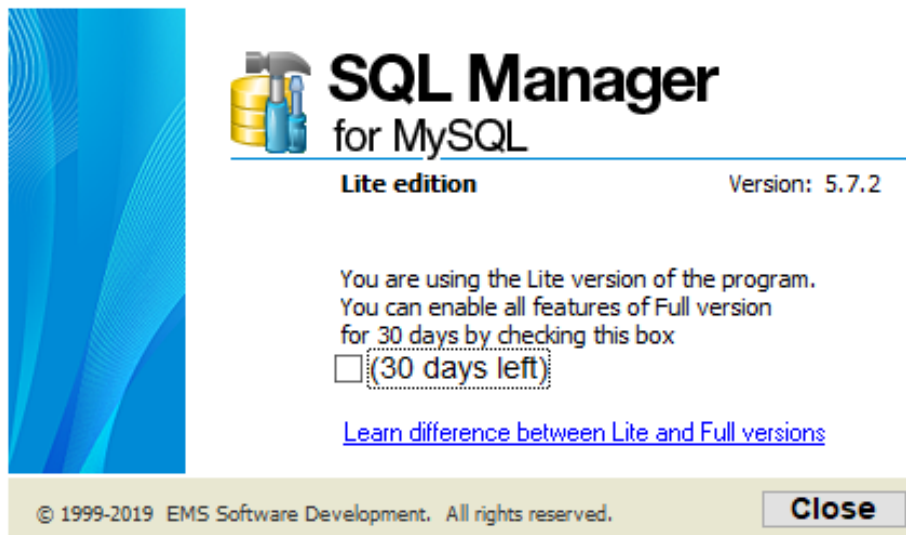
```
systemctl restart mysql.service
systemctl restart mariadb.service
```

We now need to install a database management tool to replace web-based tools like PHPMyAdmin, known vulnerabilities. There are many, however, consider the free version of tools from SQLManager for MySQL, which will work for MariaDB. Link can be found in Learn, but is here as well:

<https://www.sqlmanager.net/en/tools/free>

Defaults for installation and initial launch should be fine, but for the first month of use, every time you will be prompted to try the full version. **I strongly recommend you always say No, as you cannot go back to the free one, and will have to pay after the trial is over, or try something else.**

Just click **Close** to the dialog box below



Now that our connection is ready, we need to create a database to connect to, and accounts to connect with. We should not use root for this, we should use accounts related to the application and database that goes with it.

Log into the command line version of mysql:

```
mysql -u root
```

Once in MySQL command line utility (or more specifically, MariaDB), create a database with:

```
create database blog;
```

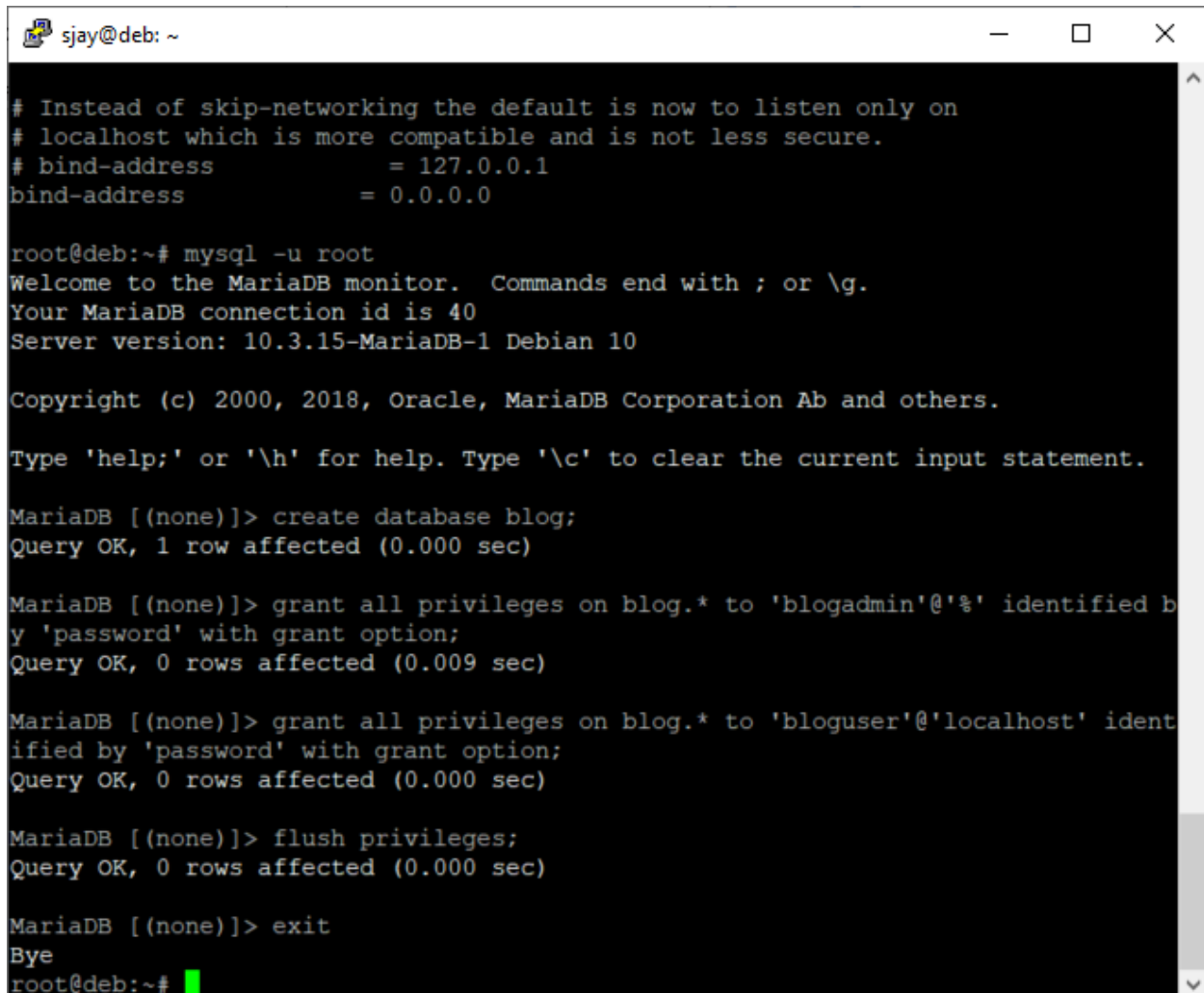
Next we need to create two users, a localhost account for database connections within the application, and one to allow network connectivity to manage through something like SQLManager

```
grant all privileges on blog.* to 'bloguser'@'localhost' identified by 'password';
```

```
grant all privileges on blog.* to 'blogadmin'@'%' identified by 'password';
```

Finally, flush privileges to apply the permissions:

```
flush privileges;
```

A terminal window titled 'sjay@deb: ~' with standard window controls. It shows a series of MySQL commands and their outputs. The commands include setting bind-address, logging in as root, creating a database named 'blog', granting privileges to 'blogadmin' and 'bloguser', flushing privileges, and exiting. The output shows the MariaDB monitor interface and confirmation messages for each command.

```
sjay@deb: ~  
# Instead of skip-networking the default is now to listen only on  
# localhost which is more compatible and is not less secure.  
# bind-address          = 127.0.0.1  
bind-address           = 0.0.0.0  
  
root@deb:~# mysql -u root  
Welcome to the MariaDB monitor.  Commands end with ; or \g.  
Your MariaDB connection id is 40  
Server version: 10.3.15-MariaDB-1 Debian 10  
  
Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.  
  
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.  
  
MariaDB [(none)]> create database blog;  
Query OK, 1 row affected (0.000 sec)  
  
MariaDB [(none)]> grant all privileges on blog.* to 'blogadmin'@'%' identified by 'password' with grant option;  
Query OK, 0 rows affected (0.009 sec)  
  
MariaDB [(none)]> grant all privileges on blog.* to 'bloguser'@'localhost' identified by 'password' with grant option;  
Query OK, 0 rows affected (0.000 sec)  
  
MariaDB [(none)]> flush privileges;  
Query OK, 0 rows affected (0.000 sec)  
  
MariaDB [(none)]> exit  
Bye  
root@deb:~#
```


Now we can connect with our database management tool:

Launch SQL Manager for MySQL, if not already running. Select menu item Database → Register Database, and set the credentials to connect to your database (for me, the database IP is 192.168.56.110, database is blog, the user is blogadmin, and password is password):

Register Database Wizard

Register Database

Specify the connection parameters



SQL
Manager
for
MySQL

Welcome to the Register Database Wizard!
This wizard allows you to set the connection parameters for the selected databases only once, giving you the possibility to connect them quickly afterwards.

This wizard will guide you through the process of setting the connection parameters, selecting databases, and customizing their specific options.

Host name

192.168.56.110

Port

3306

User name

blogadmin

Password

••••••••

Named pipe

☐ Use HTTP tunneling

Help

< Back


Next >

Cancel

Register Database Wizard

Register Database

Set some specific options for registered database(s) and click the Finish button



SQL
Manager
for
MySQL

Database name

blog

Database alias

information_schema

Client charset

DEFAULT

Font charset

DEFAULT_CHARSET

☒ Refresh objects on connection

☐ Interactive mode

☐ Login prompt before connection

☒ Quote identifiers

☐ Use compression protocol

☐ Autoconnect at startup

☐ Create new tab for this database

Help

< Back

Finish

Cancel

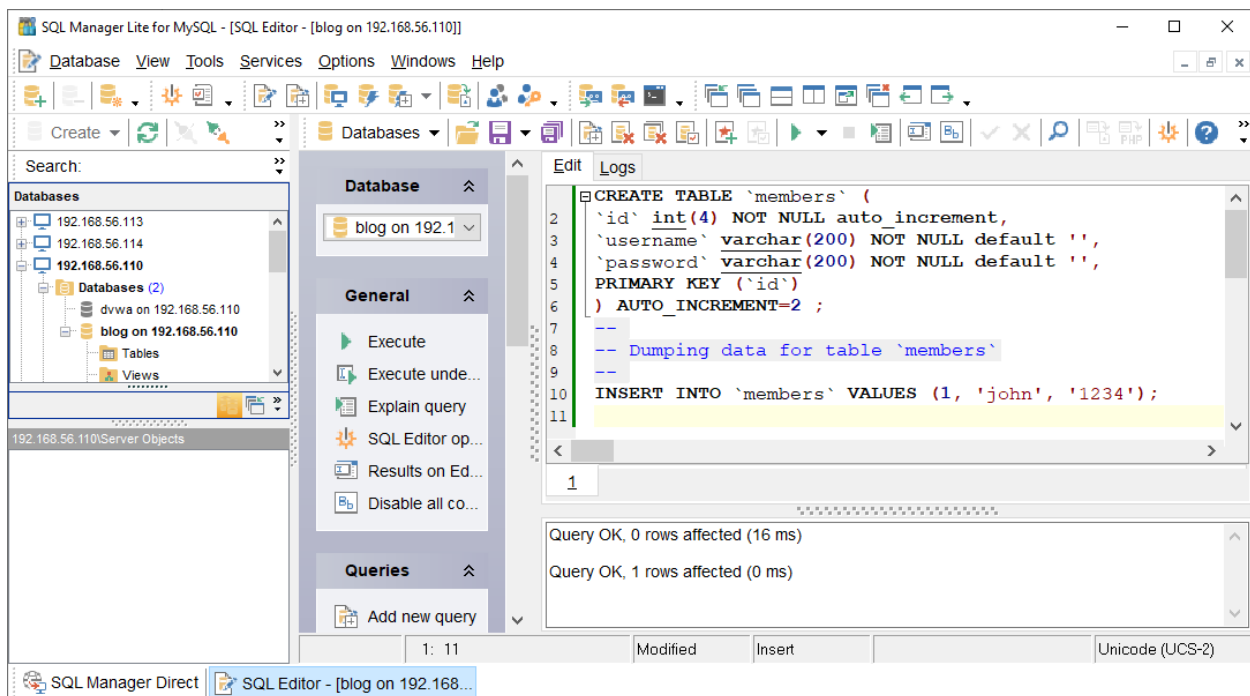
STEP 2: Create table "members"

Test your above and log into your database with the account created above. Use the SQL below for testing this code. It will create a table called members in your blog database.

Database		
Table "members"		
id	username	password
1	john	1234

```
CREATE TABLE `members` (  
  `id` int(4) NOT NULL auto_increment,  
  `username` varchar(255) NOT NULL default '',  
  `password` varchar(255) NOT NULL default '',  
  PRIMARY KEY (`id`)  
) AUTO_INCREMENT=2 ;  
--  
-- Dumping data for table `members`  
--  
INSERT INTO `members` VALUES (1, 'john', '1234');
```

Your output should look like the following:



Again, the above is used on systems that don't give us PHPMyAdmin, something that shouldn't be used anyway given the vulnerabilities it causes, and given you shouldn't deploy extra services on a production environment.

STEP 3: Create file main_login.php

The first file we need to create is "main_login.php" which is a login form.

Member Login

Username :

Password :

action='checklogin.php'

Code

```
<table width="300" border="0" align="center" cellpadding="0" cellspacing="1"
bgcolor="#CCCCCC">
<tr>
<form name="form1" method="post" action="checklogin.php">
<td>
<table width="100%" border="0" cellpadding="3" cellspacing="1" bgcolor="#FFFFFF">
<tr>
<td colspan="3"><strong>Member Login </strong></td>
</tr>
<tr>
<td width="78">Username</td>
<td width="6">:</td>
<td width="294"><input name="username" id="username" required></td>
</tr>
<tr>
<td>Password</td>
<td>:</td>
<td><input name="password" id="password" required></td>
</tr>
<tr>
<td>&nbsp;</td>
<td>&nbsp;</td>
<td><input type="submit" value="Login"></td>
</tr>
</table>
</td>
</form>
</tr>
</table>
```

STEP 4: Create file databaseconnection.php

It is often considered a good idea to create a separate database connection file to be used with your application. As we have seen, however, you can have different connection privileges through SQL to mitigate against injections. If you choose to do this, you should use separate connection strings for activities like registration and logging in.

For now, we will create a single SQL connection, using the following and save as databaseconnection.php:

```
<?php
//Define variables needed to connect to the MySQL database
define('DB_DSN', 'mysql:host=localhost;dbname=blog;charset=utf8');
define('DB_USER', 'bloguser');
define('DB_PASS', 'password');

//Connect to the database. If the connection fails the webapp exits
try {
    $db = new PDO(DB_DSN, DB_USER, DB_PASS);
} catch (PDOException $e) {
    echo 'Error: '.$e->getMessage();
    die(); // Force execution to stop on errors.
}
?>
```

Of course, you will need to adjust username, password, and maybe database name to what you set in your instance.

STEP 5: Create file checklogin.php

We have a login form in step 2, when a user submit their username and password, PHP code in checklogin.php will check that this user exist in our database or not.

If user has the right username and password, then the code will register username and password in the session and redirect to "login_success.php". If username or password is wrong the system will show "Wrong Username or Password".

Code

```
<?php
ob_start(); // session management

require('databaseconnection.php');

$tbl_name="members"; // Table name if you wish to use a variable

$myusername=$_POST['username'];
$mypassword=$_POST['password'];
// To protect MySQL injection, we use PDO
$select_sql = "SELECT username FROM members WHERE username='" . $myusername . "' AND
    password='" . $mypassword . "'";
$statement = $db->prepare($select_sql);
$statement->execute();
$user = $statement->fetch();

// If returned password matches entered password, valid login

if($user){
```



```

        // Register $myusername and redirect to file "login_success.php"
        session_start();
        $_SESSION['username'] = $user['username'];
        header("location:login_success.php");
    }
    else {
        echo "Wrong Username or Password";
        echo "<pre>$select_sql</pre>";
    }
    ob_end_flush();
?>

```

STEP 6: Create file login_success.php

User can't view this page if the session is not registered.

Code

```

<?php
// Check if session is not registered, redirect back to main page.
// Put this code in first line of web page.
session_start();

if(!isset($_SESSION['username'])){
    header("location:main_login.php");
}
?>
Login Successful <?= $_SESSION['username'] ?>

```

STEP 7: Create file logout.php

For any application, you will need a logout file. The code in this file will destroy the session.

```

<?php
// Put this code in first line of web page.
session_start();
session_destroy();

echo "Session Username is ".$_SESSION['username'];
?>

```

STEP 8: Harden the Login Script

If you try a SQL injection to the above, we see our injection will allow us to log in. We need to mitigate this, and as mentioned, the more tactics the better. Please modify your login to use PDO, as below:

```

if ($_POST) {
    $select_sql = "SELECT username FROM members WHERE username=:username and
        password=:password;";
    $statement = $db->prepare($select_sql);
    $statement->execute($_POST);
    $statement->bindParam(':username', $_POST['username']);
    $statement->bindParam(':password', $_POST['password']);
    $user = $statement->fetch();
}

```

```
}
```

With the above, we will cover most problems. If someone is able to still pass an injection into our code, we should take our mitigation to another level. Rather than select a username from the users table where the username and password match, it is always better to take a password from the database based on a username supplied, and compare passwords. This also makes for easier hash implementation, as we will see in the future. Consider the following:

```
if ($_POST) {
    $select_sql = "SELECT password FROM members WHERE username=:username;";
    $statement = $db->prepare($select_sql);
    $statement->bindParam(':username', $_POST['username']);
    $statement->execute();
    $pass = $statement->fetch();
}

// If returned password matches entered password, valid login

if ($pass['password']==$_POST['password'] && $_POST['password']<>''){
    // Register $myusername and redirect to file "login_success.php"
    session_start();
    $_SESSION['username'] = $_POST['username'];
    header("location:login_success.php");
}
```

Now would be a good time for a milestone backup, now that you have things working well. In terminal, change directory to /var/www/html and type:

```
cp -r login login_backup
```

The above assumes the directory you were working from is called “login”. This will create a backup of the directory you are already working in

STEP 9: Encrypt your work

To ensure proper protection, we want to store our passwords encrypted. Modify your checklogin.php to use encryption as below, or create a new form file that calls a new checklogin.php.

It is very important to note that the below script will not work without registering new users to our new system, as we need to create a proper hash. Easiest way to do this is by creating a registration page, and using it (step 10 below).

```
<?php
ob_start(); // session management

require('databaseconnection.php');

$mypassword=$_POST['password'];

$select_sql = "SELECT password FROM members WHERE username=:username;";
$statement = $db->prepare($select_sql);
```

```

$statement->bindParam(':username',$_POST['username']);
$statement->execute();
$pass = $statement->fetch();

$returnedpassword=$pass['password'];

$checkpassword = password_verify($mypassword, $returnedpassword);

// If returned password matches entered password, valid login
if($checkpassword){
    // Register $myusername and redirect to file "login_success.php"
    session_start();
    $_SESSION['username'] = $_POST['username'];

    // the following will be used for brute force attacks in the future
    echo "Successful login as: " . $_SESSION['username'];
    header("refresh:3; url=login_success.php");
}
else {
    echo "Wrong Username or Password";
    // the following code should never be seen in a production website
    echo "<pre>$select_sql</pre>";
    echo "<pre>";
    print_r($pass);
    echo "<br /> password based on form: " . $mypassword;
    echo "<br /> password from database: " . $returnedpassword;
    // These are the hashed password's components
    // password_verify will use this info to recreate the hash created by
    // password_hash(). This works because we know it uses bcrypt
    $algo = substr($returnedpassword, 0, 4); // $2y$ == Blowfish/bcrypt
    echo "<br /> password algo: " . $algo;
    $cost = substr($returnedpassword, 4, 2);
    echo "<br /> password cost: " . $cost;
    $salt = substr($returnedpassword, 7, 22);
    echo "<br /> password salt: " . $salt;
    $hash = substr($returnedpassword, 29);
    echo "<br /> password hash: " . $hash;

    // recreate hash from the form password and stored hash components
    $rehash_args=$algo.$cost."$".$salt;
    echo "<br /> form password hashed: " . crypt($mypassword,
    $rehash_args);
    echo "</pre>";
}
ob_end_flush();
?>

```

When we look at this, we see that we need to update the table with the correct, encrypted string. Real world, we would never echo back the encrypted password data to the user, but we can use this to update our table.

STEP 10: Register to database

We need to create a form that accepts registration information (username and password), and store them to the database. Of course, now we need to ensure the password is encrypted, and whatever we use for the encryption comparison is the same we use for storing the password in the database. Fortunately the PHP functions `password_hash()` and `password_verify()` manage this for us.

First the form. Add the following to `register.php`:

```
<table width="300" border="0" align="center" cellpadding="0" cellspacing="1"
bgcolor="#CCCCCC">
<tr>
<form name="form1" method="post" action="process_register.php">
<td>
<table width="100%" border="0" cellpadding="3" cellspacing="1"
bgcolor="#FFFFFF">
<tr>
<td colspan="3"><strong>Member Register </strong></td>
</tr>
<tr>
<td width="78">Username</td>
<td width="6">:</td>
<td width="294"><input name="username" type="text" id="username"></td>
</tr>
<tr>
<td>Password</td>
<td>:</td>
<td><input name="password" type="text" id="password"></td>
</tr>
<tr>
<td>&nbsp;</td>
<td>&nbsp;</td>
<td><input type="submit" name="Submit" value="Register"></td>
</tr>
</table>
</td>
</form>
</tr>
</table>
```

Next, we need the PHP to write this to the database. Add the following to a new file called `process_register.php`:

```
<?php

ob_start(); // session management

require('databaseconnection.php');

$tbl_name="members"; // Table name if you wish to use a variable
```

```

$myusername=$_POST['username'];
$mypassword=$_POST['password'];

$encrypted_password = password_hash($mypassword, PASSWORD_DEFAULT);

$insert_sql="insert into members (username,password) values
(:myusername,:encrypted_password)";
$stmt = $db->prepare($insert_sql);
$stmt->bindParam(':myusername',$myusername);
$stmt->bindParam(':encrypted_password',$encrypted_password);
$stmt->execute() or die(print_r($stmt->errorInfo(), true));
$pass = $stmt->fetch();

echo "Registered";
header("refresh:3; url=main_login.php");

// Again, we should never see this in a production environment
printf("<br />SQL statement is $insert_sql");
ob_end_flush();
?>

```

The above function `password_hash()` generates a salt automatically and saves it with the password hash (as well as the encryption algorithm).

With the above, when we successfully create a record, we are then taken to the login page. We will build on this when it is incorporated into our blog website.

Final Notes

With `password_hash()` and `password_verify()` functions, we effectively have functions that wrap the PHP `crypt()` function with strong parameters like Blowfish/bcrypt and a nice long salt. The nice thing is that it stores its encryption parameters with the password hash so it should be somewhat future proof; this is why we were able to recreate the process to compare incorrectly supplied passwords.

In future releases, the `password_hash()` function might choose to use a more advanced hashing algorithm, but it shouldn't break existing hashes because this info is stored with the hash.