

# Web Development

## Lecture 5 - Databases and PHP

# RoadMap - Next Few Sessions

- PHP and MySql
- Validating and Sanitizing form data before sending to database
- Finishing our login and registration pages
  - error message handling
  - Password hashing
- Providing user a way of logging out of a session
- Making application more dynamic using include files
- Incorporate a Products application with CRUD

# Outline

- Setting up a database table
- Making a connection from PHP to a DB
- Creating a registration and login page that uses a DB

Before we start extract the zip file on moodle and place under the htdocs directory

This should create **lecture5** folder, which is where we will place all our files for today

# HeidiSQL

When working with databases, we need have an administration client that will allow us to work with the database.

PHPMyAdmin is one client that you may have seen before as part of your XAMPP installation.

# HeidiSQL

Instead of using PHPMyAdmin, we will use a client called HeidiSQL.

Heidi can be downloaded from:

<http://www.heidisql.com>

Download and install this piece of software!

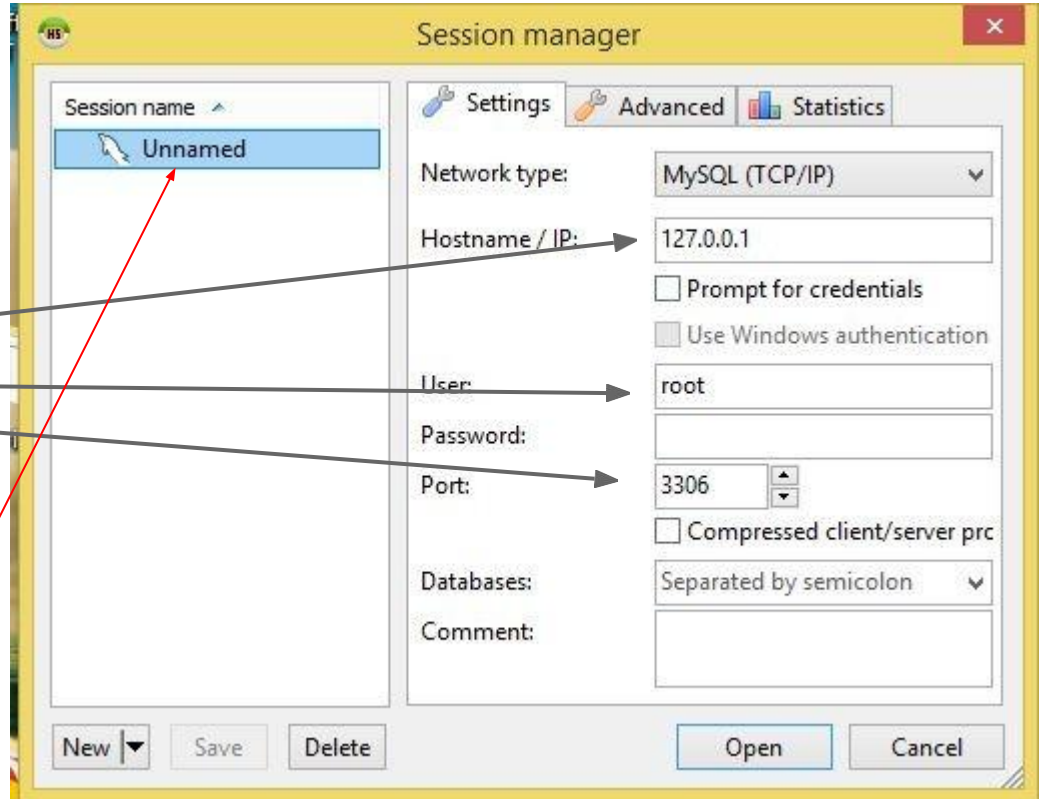
# HeidiSQL

Note localhost and 127.0.0.1 are interchangeable, the latter being the ip address of localhost

When you open Heidi, you will see that you can add the details of the database you are running with your XAMPP.

Make sure that these details are the ones for your local database! (see next slide)

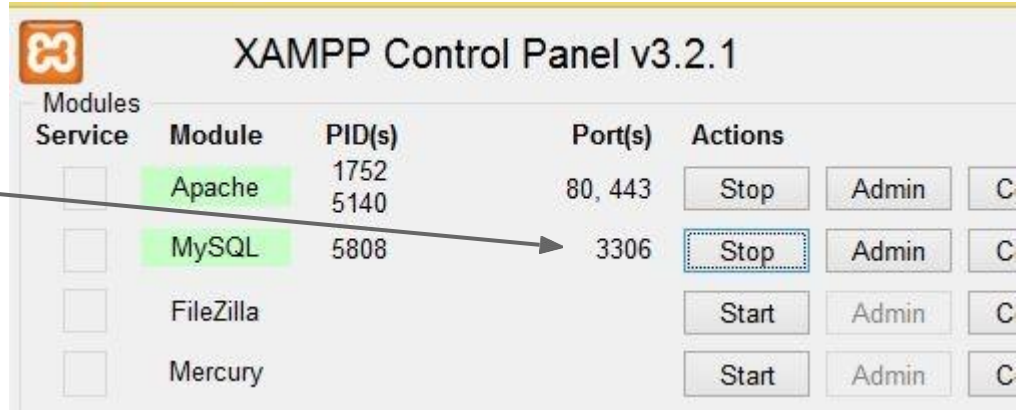
Give the **session** a name e.g. Lecture 5



# Checking the port number

If you want to know what port you need to use for MySQL, check the port number on your XAMPP console

In this case, you can see my database is running on port 3306.



Service	Module	PID(s)	Port(s)	Actions
<input type="checkbox"/>	Apache	1752 5140	80, 443	Stop Admin C
<input type="checkbox"/>	MySQL	5808	3306	Stop Admin C
<input type="checkbox"/>	FileZilla			Start Admin C
<input type="checkbox"/>	Mercury			Start Admin C

## Login Details

The username by default is root for the username, and nothing for the password, just leave it blank. We use 127.0.0.1 for the database address, because the database is running on our local machines.

# HeidiSQL

After you have entered in your details click “Open” and a new connection to the database will be opened.



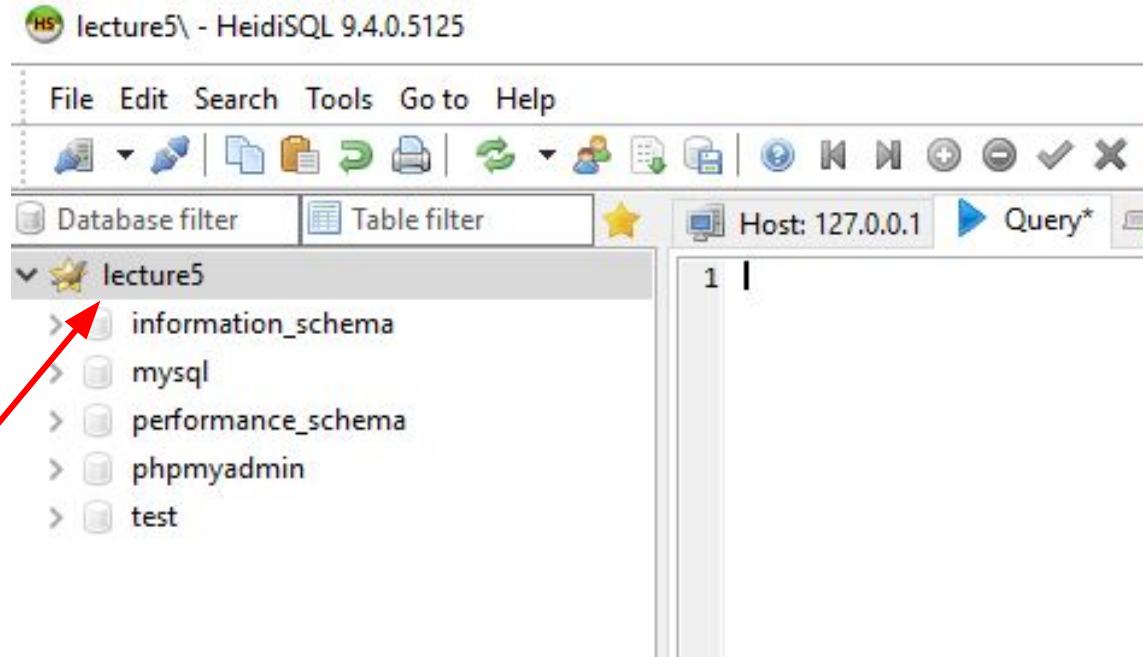
# Creating the database

When Heidi opens,  
on the left hand side, you will see  
the list of databases.

We'll start by adding a new  
database.

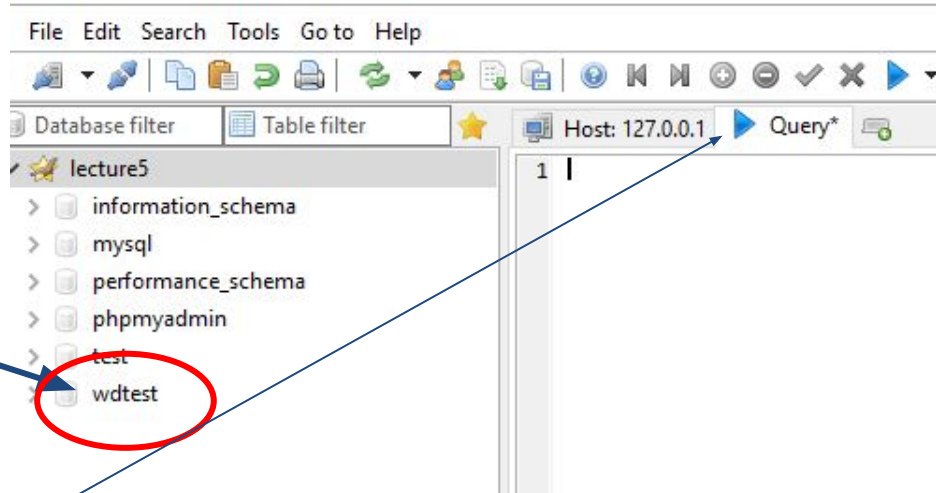
Right click on the session and  
select Create new > Database

Enter “**wdtest**” as the database  
name



# Adding the table SQL

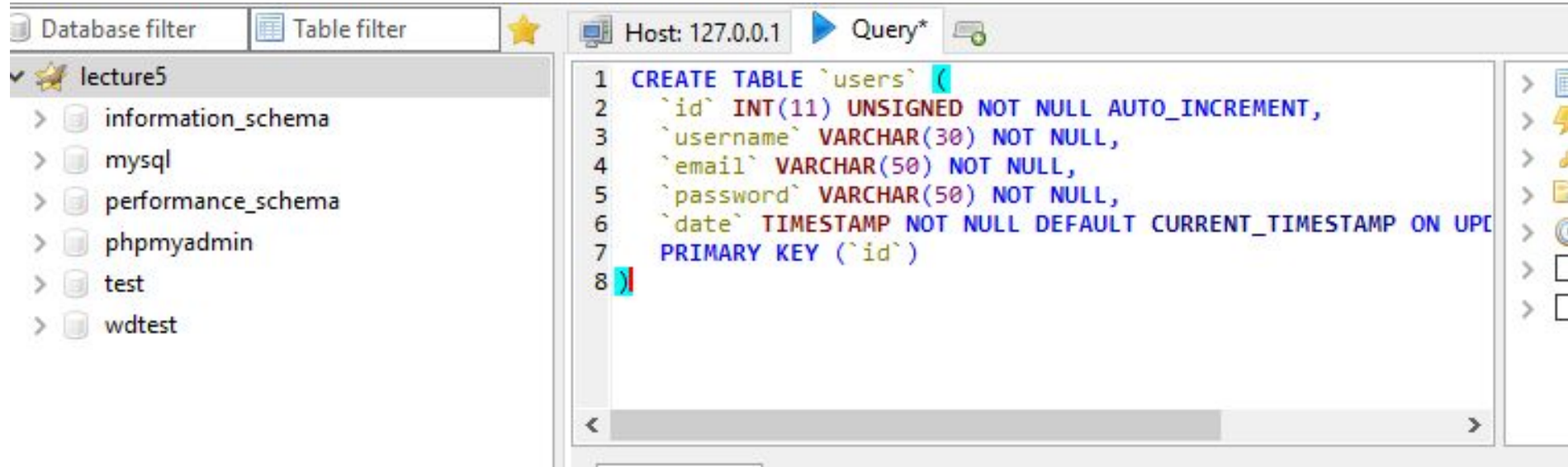
Select “wdtest” as your database by clicking on it



Then Click on the blue tab, for Query

When you are inside the query tab, copy the contents of the SQL from the **dbusers.txt** file in **lecture5** folder and paste it into the text box.

# Adding the table SQL

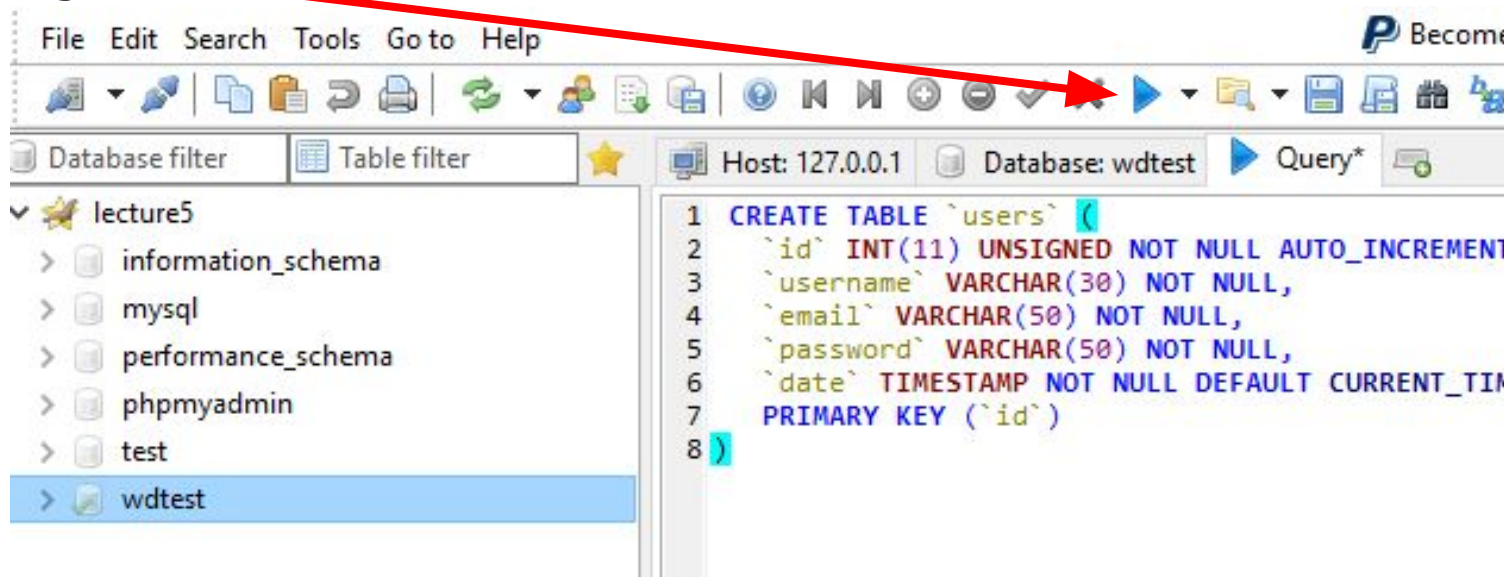


The screenshot shows a MySQL IDE interface. On the left, a 'Database filter' and 'Table filter' are visible. Below them, a tree view shows the 'lecture5' database selected, with sub-items: 'information\_schema', 'mysql', 'performance\_schema', 'phpmyadmin', 'test', and 'wdtest'. The main area displays a SQL query in a text editor, with a 'Host: 127.0.0.1' and 'Query\*' toolbar at the top. The query is as follows:

```
1 CREATE TABLE `users` (  
2   `id` INT(11) UNSIGNED NOT NULL AUTO_INCREMENT,  
3   `username` VARCHAR(30) NOT NULL,  
4   `email` VARCHAR(50) NOT NULL,  
5   `password` VARCHAR(50) NOT NULL,  
6   `date` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,  
7   PRIMARY KEY (`id`)  
8 )
```

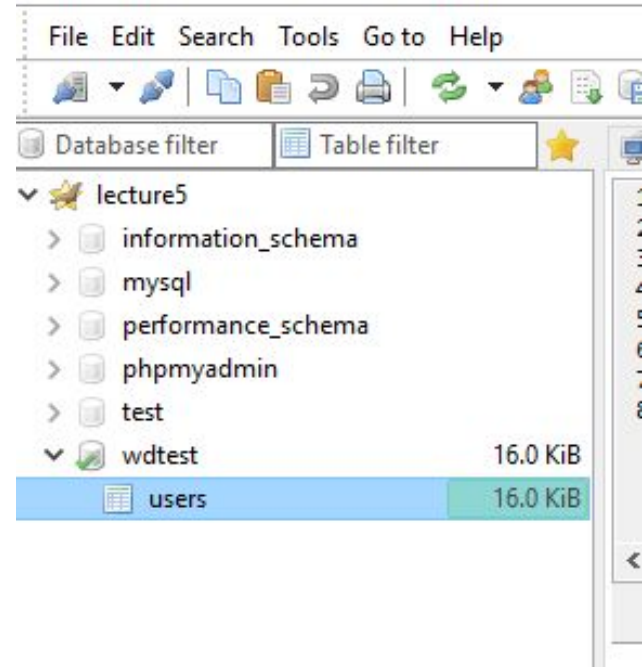
# Run the SQL

To finish, all we need to do is run this code, by clicking the Run button



# That's it!

You will see a new table has been created in your database.



# Back To PHP

Now that we have created our website, we need to add two pages:

1 page for registering   1 page for logging  
in

# Remember!

Do not cut and paste things from the lecture notes, lots of hidden characters will appear in your code causing problems.

Use the file examples on Moodle! Or type in the code yourself

# Register Page

We are going to allow the user to type in a username, email and password, and then allow them to click a register button.



# Register.php

Below is what the form looks like. We have three fields, username, email and password. One submit button is added to this form also.

```
<!DOCTYPE>
<html>
<body>
<h2> Registration Form</h2>
  <form action="register.php" method="post">
    Username <input type="text" name="username"/>
    Email <input type="text" name="email"/>
    Password <input type="password" name="password"/>
    <input type="submit"/>
  </form>
</body>
</html>
```

Create a new file and save it as register.php in the lecture5 folder

This file is **registerhtml.txt** in the lecture5 folder

# Exception Handling

We need some way of capturing errors when running PHP code. Its very similar to JAVA.

We use a try/catch statement

Inside the try statement is the code we want to execute

Inside the catch we do some error handling

1. `try{`
2. `//code that may throw exception`
3. `}catch(Exception_class_Name ref){`
4. `//error handling`
5. `}`

# Register.php

Just above the form, we will add some PHP to collect the username, email and password and then send them to the database to be saved.

**<?php**  If here is the same as `$_SERVER['REQUESTST_METHOD'] == "POST"`  
**if(\$\_POST){**

`// $_POST will have no value if not come from form post`

`$username = $_POST['username'];`

`$password = $_POST['password'];`

`$email = $_POST['email'];`

`try {`

`$host = '127.0.0.1';`

`$dbname = 'wdtest';`

`$user = 'root';`

`$pass = "";`

`# MySQL with PDO_MYSQL`

`$DBH = new PDO("mysql:host=$host;dbname=$dbname", $user, $pass);`

`} catch(PDOException $e) {echo 'Error';}`

**}**

**?>**

Code for this can be found in the registerA.txt file

Get the username, email and password and store them into variables called \$username, \$email and \$password

Make a DB connection

# Making a connection with PDO

The line that makes the DB connection is below:

```
$DBH = new PDO("mysql:host=$host;dbname=$dbname", $user, $pass);
```

We are passing the following parameters

`$host` : address for the database

`$dbname` : the name of the database, in our case the wdtest database

`$user` : the username for the account we are using

`$pass` : the password for the account.

All of these details are being passed to an instance of the **PDO** class, which stands for **Prepared Data Object**

# PDO

Once we have the PDO instance we have available a whole bunch of functions (or methods) to help us interact with the mysql database:

- `prepare()`            prepare our sql statement
- `bindParam()`        bind input variables to sql statement
- `execute()`           execute the sql statement
- `fetch()`            fetch single row from result of execute
- `query()`            run a direct sql statement in no params
- `rowCount()`        number of rows returned from execute
- `fetchAll()`        returns all rows in a query
- `fetchColumn()`    return single column from query

# PDO

**PDO stands for PHP Data Objects, an advanced approach to connect to database not only mysql but several others. Here is a list of database supported by PDO-**

**Micro Soft SQL Server**

**MySQL**

**PostgreSQL**

**Oracle**

**Sqlite**

**IBM DB2**

**Firebird**

**INFORMIX etc.**

# PDO

The real PDO benefits are:

- security (*usable* prepared statements)
- usability (many helper functions to automate routine operations)
- reusability (unified API to access multitude of databases, from SQLite to Oracle)

The PDO extension defines a lightweight, consistent interface for accessing databases in PHP. It means that, regardless of which database you're using, use the same functions to issue queries and fetch data. There is no need to rewrite your queries again when you change your database.



# Insert statement preparation

```
$sql = "INSERT INTO users (username, password,email) VALUES (?, ?, ?);";  
$sth = $DBH->prepare($sql);
```

```
$sth->bindParam(1, $username);  
$sth->bindParam(2, $password);  
$sth->bindParam(3, $email);  
  
$sth->execute();  
echo 'You are now registered!';
```

Insert code just under the \$DBH statement. This code can be found in the **registerB.txt** file

Standard SQL insert statement. We are inserting into the **users** table.

Notice how we are not passing the variables, we are passing a question mark. This is because, on the line below, we are passing variables 1, 2 and 3 later in the bind statement

# bindParam()

We use the bindParam function to pass the variables into the statement. We do this because we don't want to allow people to put their own content into the SQL statement because it can lead to SQL injection! (more later)

We **prepare** the statement, and

then **bind** the parameters to the statement.

# Register.php

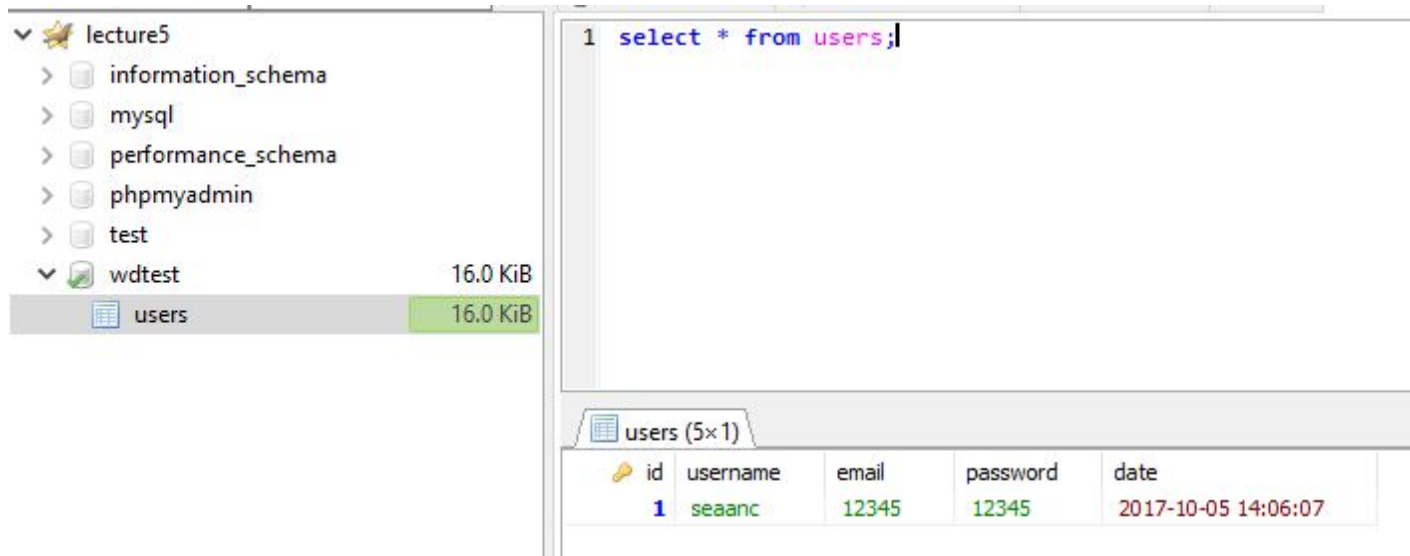
When the user enters in their username, email and password and clicks the button, a new record will then be inserted into the database table.

Lets try it. Bring up the register.php file in the browser i.e.

`localhost/lecture5/register.php`

# Register.php

If successful go back to Heidi and see if your record had been added



The screenshot shows the HeidiSQL interface. On the left, a tree view displays the database structure under 'lecture5', including 'information\_schema', 'mysql', 'performance\_schema', 'phpmyadmin', 'test', and 'wdtest'. The 'users' table is selected under 'wdtest'. The main query editor shows the SQL command: `1 select * from users;`. Below the editor, a tab labeled 'users (5x1)' displays the query results in a table format.

id	username	email	password	date
1	seaanc	12345	12345	2017-10-05 14:06:07

# Logging In

Now that we have a record inserted into the database, we can now look at the login.php page.

This page will allow the user to enter in their username and password and click Login.

# Login.php

When they click login, we want to check the database to see if what the user entered is what we have stored in the database.

# Login.php

Create a new file and add this content. Save as **login.php** in the lecture5 folder. Can see in **loginhtml.txt**

Checks if a message string is not empty. Used to display message to the user. Set in database php code to follow


```
1  <!DOCTYPE>
2  <html>
3  <body>
4    <h2>Login</h2><br></br>
5    <form action="login.php" method="post">
6      Username <input type="text" name="username"/>
7      Password <input type="password" name="password"/>
8      <input type="submit" name="submit" value="Login"/>
9    <?php
10     if(!empty($message)){ echo '<br>';
11     echo $message;
12   }
13   ?>
14 </form>
15 </body>
16 </html>
```

# Just above the form we add this..

<?php

```
if($_POST){  
    $username = $_POST['username'];  
    $password = $_POST['password'];  
    try {  
        $host = '127.0.0.1';  
        $dbname = 'wdtest';  
        $user = 'root';  
        $pass = "";  
        # MySQL with PDO_MYSQL  
        $DBH = new PDO("mysql:host=$host;dbname=$dbname", $user,  
            $pass);  
    } catch(PDOException $e) {echo 'Error';}  
}
```

?>



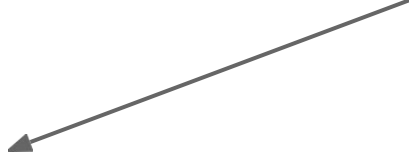
This code is the same as  
the register.php code.  
Can copy from  
**registerA.txt**, just  
remove the email  
reference



# Named Placeholders

This next example uses what we call named placeholders. We give a name, and then give the value later on. Named placeholders look like this

:name



```
$q = $DBH->prepare("select * from user where username = :username and password = :password  
LIMIT 1");
```

```
$q->bindValue(':username', $username);
```

```
$q->bindValue(':password', $password);
```

```
$q->execute();
```

```
$row = $q->fetch(PDO::FETCH_ASSOC);
```

```
//returns table row as an associative array
```

```
//of values column names to data values
```

```
//Array ( [id] => 1 [username] => seaanc
```

```
//      [email] => 12345 [password] => 12345 [date] => 2017-10-05 14:06:07 )
```

```
$message = "";
```

```
if (!empty($row)){
```

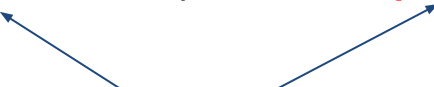
```
    $username = $row['username'];
```

```
    $message = 'Loggin in!';
```

```
} else {
```

```
    $message= 'Sorry your log in details are not correct';
```

```
}
```



This uses “named” place holders, we just give a little nickname to the variables

This file is **loginB.txt** in the lecture5 folder. Add code after the \$DBH statement

# What happens?

If everything goes ok, the user will login.

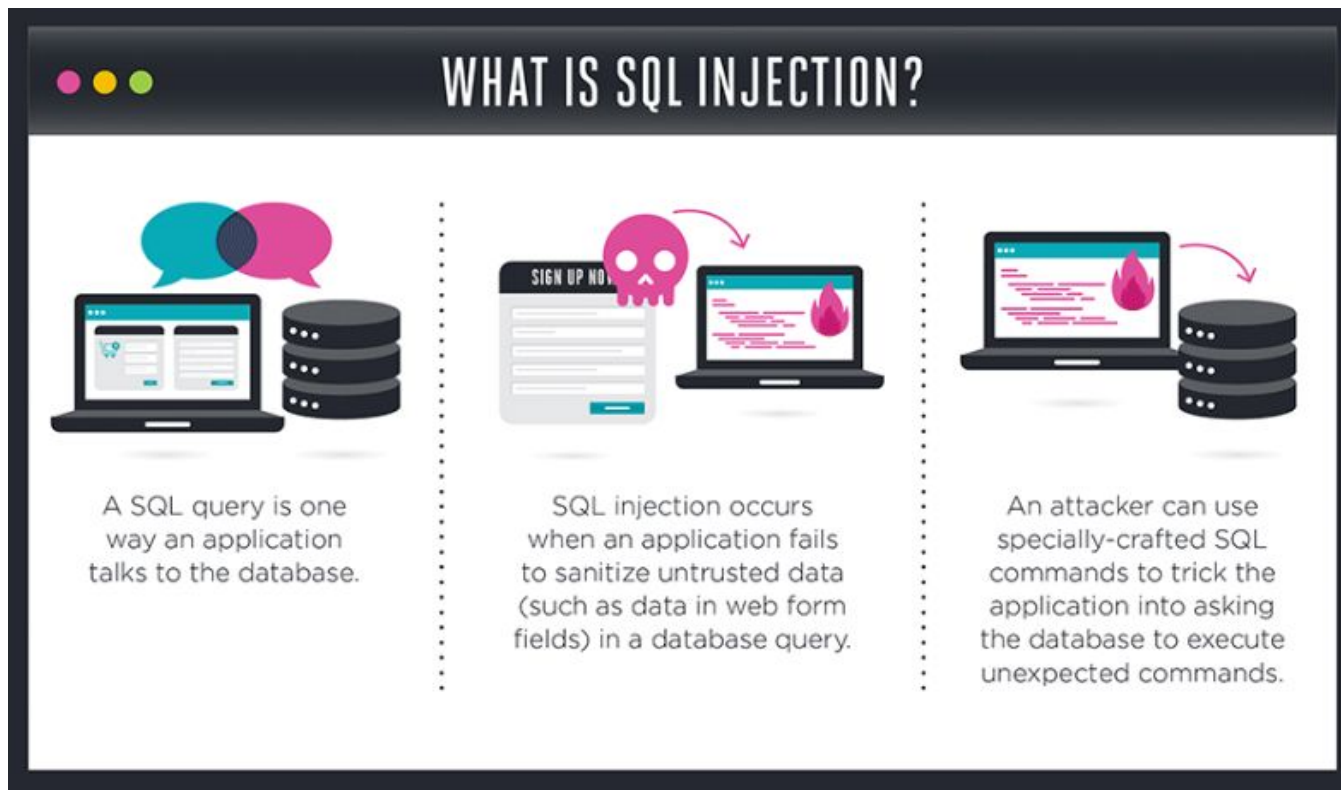
If the details are not correct, it will print an error message to the screen.

# Security

One of the big issues with allowing people to insert or select records from forms directly to the database is SQL injection.

SQL injection is when someone adds some SQL into a form field, e.g. the username field with malicious intent to break a database.

# SQL Injection



# Security

When the insert statement goes to add the username and password into the database, a person may inject some SQL themselves such as:

`'DROP TABLE users;'`

Via an input element on the form

# SQL Injection

**SQL injection** is a code **injection** technique, used to attack data-driven applications, in which nefarious **SQL** statements are inserted into an entry field for execution (e.g. to dump the database contents to the attacker).

```
$name = "Bobby';DROP TABLE users;";
```

Entered via form

```
$query = "SELECT * FROM users WHERE name='$name'";
```

which compiles into malicious sequence

```
SELECT * FROM users WHERE name='Bobby';DROP TABLE users;
```

Previous versions of php mysql libraries would allow you to set up a query like this

# Named Placeholders

If we use PDOs to do our insert statements, and our select statements with named placeholders, or positional(?) place holders we can help prevent SQL Injection

If we don't use prepared statements like in the previous slide example we would be required to do more work to sanitize the data before issuing the query



**PDO::bindParam()** //returns true or false

Binds a PHP variable to a corresponding named or question mark placeholder in the SQL statement that was used to prepare the statement.

Note that PDO supports positional (?) and named (:email) placeholders, the latter always begins from a colon and can be written using letters, digits and underscores only. Also note that no quotes have to be ever used around placeholders.

# Named Placeholder

```
<?php
```

```
/* Execute a prepared statement by binding PHP variables */
```

```
$calories = 150;
```

```
$colour = 'red';
```

```
$sth = $dbh->prepare('SELECT name, colour, calories  
FROM fruit
```

```
WHERE calories < :calories AND colour = :colour');
```

```
$sth->bindParam(':calories', $calories);
```

```
$sth->bindParam(':colour', $colour);
```

```
$sth->execute();
```

```
?>
```

# Positional Placeholder


```
<?php
/* Execute a prepared statement by binding PHP variables */
$calories = 150;
$colour = 'red';
$sth = $dbh->prepare('SELECT name, colour, calories
    FROM fruit
    WHERE calories < ? AND colour = ?');
$sth->bindParam(1, $calories);
$sth->bindParam(2, $colour);
$sth->execute();
```

"bindParam" comes into action, binding placeholders to user entered data. Notice that **bindParam only binds data to placeholders leaving unchanged the query**. So there is no way to change the original SQL query, because it has already sent to the server by means of the **prepare** method and because you are sending **SQL queries and input data separately** so user entered data can't interfere with queries.

This allows SQL to insert the values into the expression, without modifying the expression itself.

So you would not be able to do something like

```
$name = "Bobby';DROP TABLE users; ";
```



It simply becomes a search string rather than becoming part of the query i.e we would be searching for a name of

```
"Bobby';DROP TABLE users; "
```

Which would be meaningless but harmless

# Add Some style

There is a css file on the moodle page called **style.css**. Download and place in your folder

We will add it to the html of both pages. Add the html to load the css

```
<head>  
  <link rel="stylesheet" href="style.css">  
</head>
```

After the `<html>` tag in both files

We now need to add the class attributes from the css to the `form` and `input` submit button

## Add Some style

To the form tag we add the class attribute

`<form` **`class='form-style'`** .....

Add to both files

**register.php**

And

**login.php**

To the submit button we add the class attribute so that

`<input type="submit"` **`class='button'`** .....

# Homework

Using the application we've worked on add the following functionality

Add a confirmation page to the application, say confirm.php

From the register.php file and the login.php files if we successfully store and retrieve a record to/from the database re-route to this new confirmation page using the header function

Store session data before rerouting for the **username** and email and display it in the new confirmation page.

Store a message via session variable "thank you for registering".\$username if coming from register.php

Don't forget to add start\_session() to the top of each page.

# Homework

Check that a session variable exists for **username** before printing the session variables in confirm.php else re-route to login.php

Print register message

```
if appropriate session variable set
    print register message
else
    print "Logged in as: ".$username
```