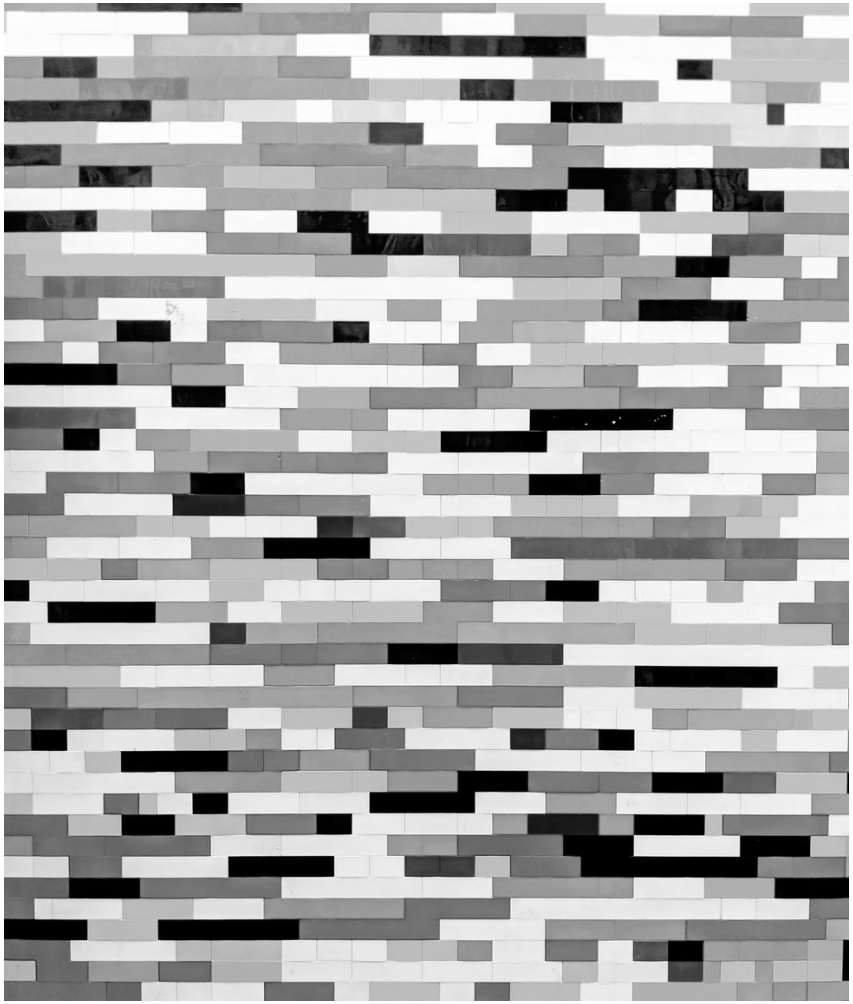
DEEPer

Databases

Week 5 Session 1

TODAY'S SESSION



01 MySQL

02 Databases in PHP

MySQL

01

What Is A Database?

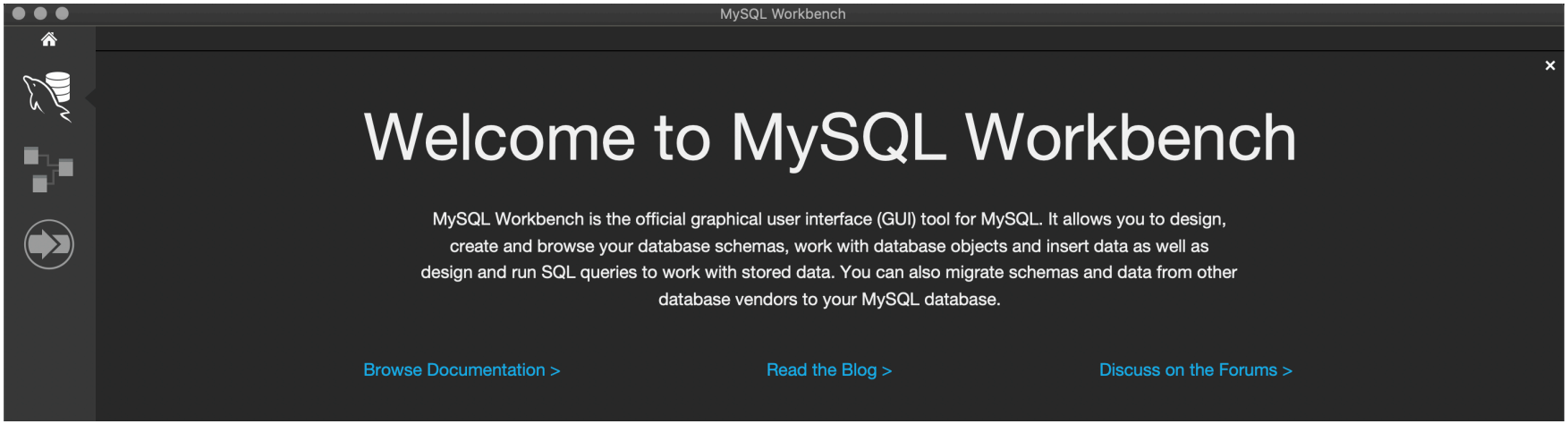
- A database is a storage mechanism for data, generally with some pre-defined structure
- A single **database** can contain multiple **tables**, each of which has its own structure
- Each **table** contains multiple **columns**
- Once defined, each **table** can then contain multiple **data rows**
- For example a shop may have a **shop** database, which contains one table for **products** and another for **reviews**

What Is A Database Table?

- A table stores rows of data separated into multiple columns
- Each column is defined up front, and only created columns can be populated
- Each column has some metadata, including what type of data it stores and its maximum length
- A single table in MySQL can technically support up to 4096 columns, but they should never reach even close to that number!

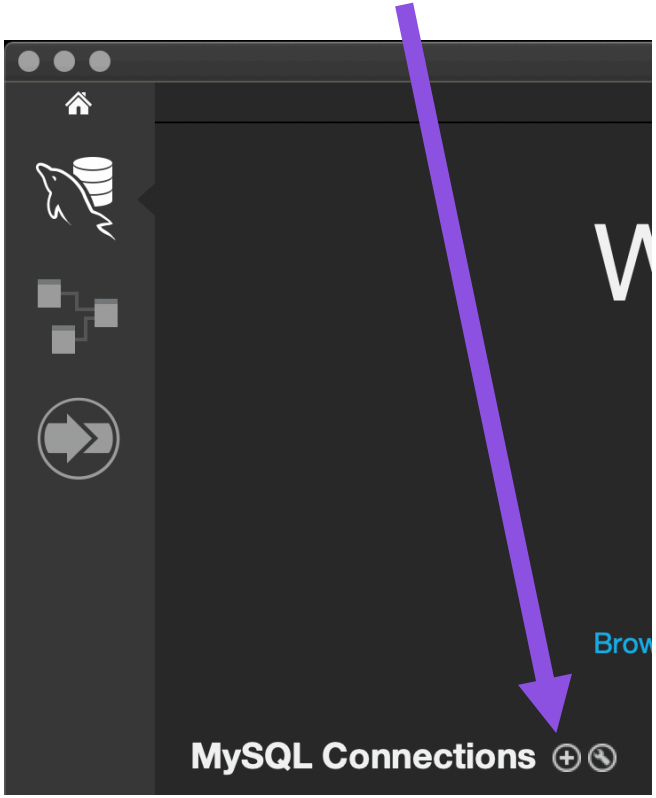
Connecting To Our DB – Code Along

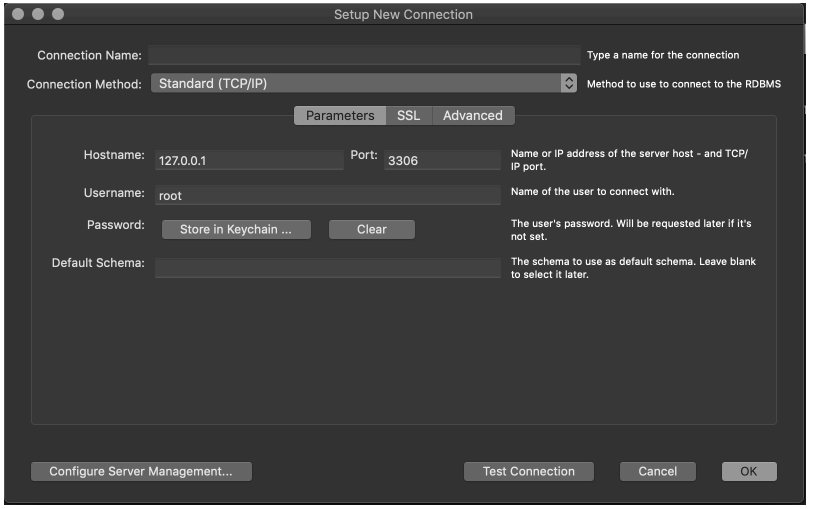
Using command + space, open “MySQLWorkbench”



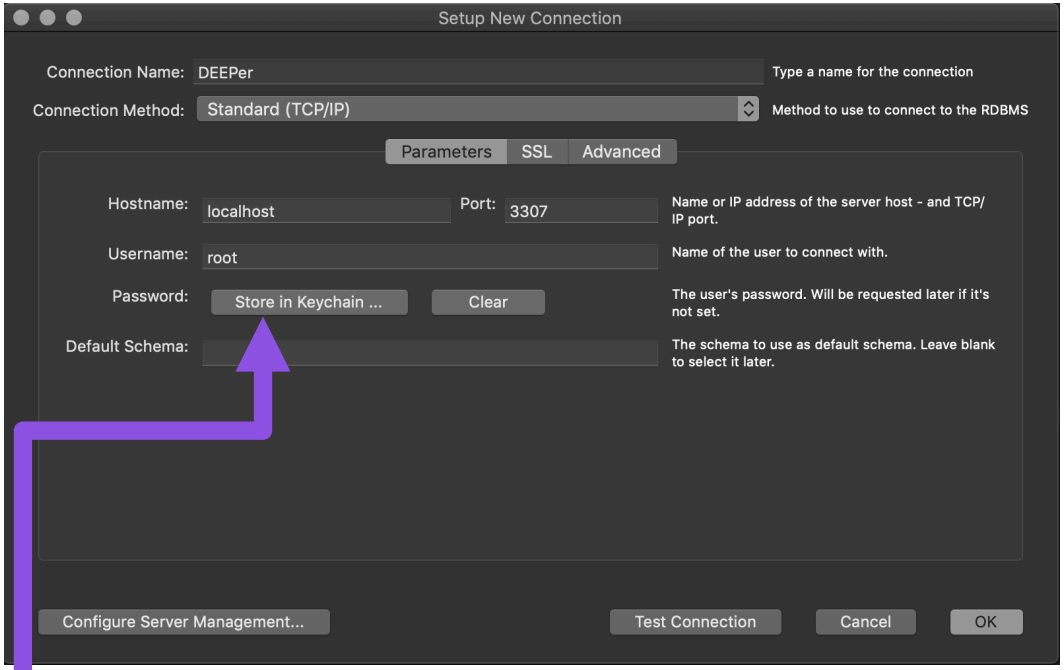
Connecting To Our DB – Code Along

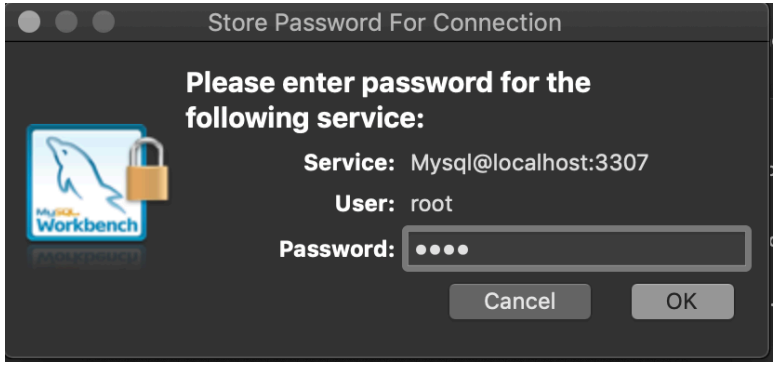
Click on the + next to MySQL Connections





Connecting To Our DB – Code Along

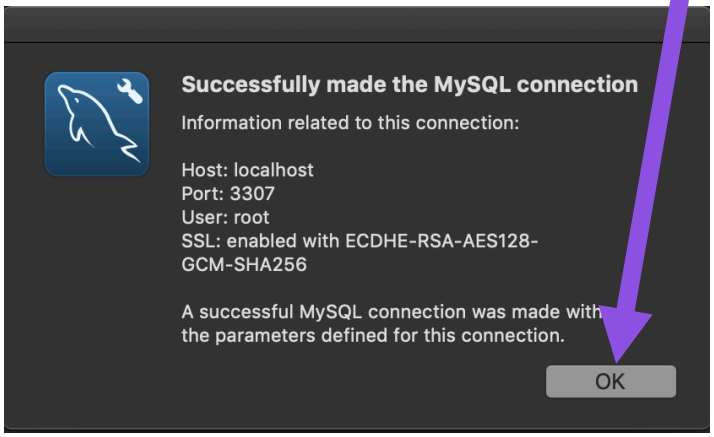


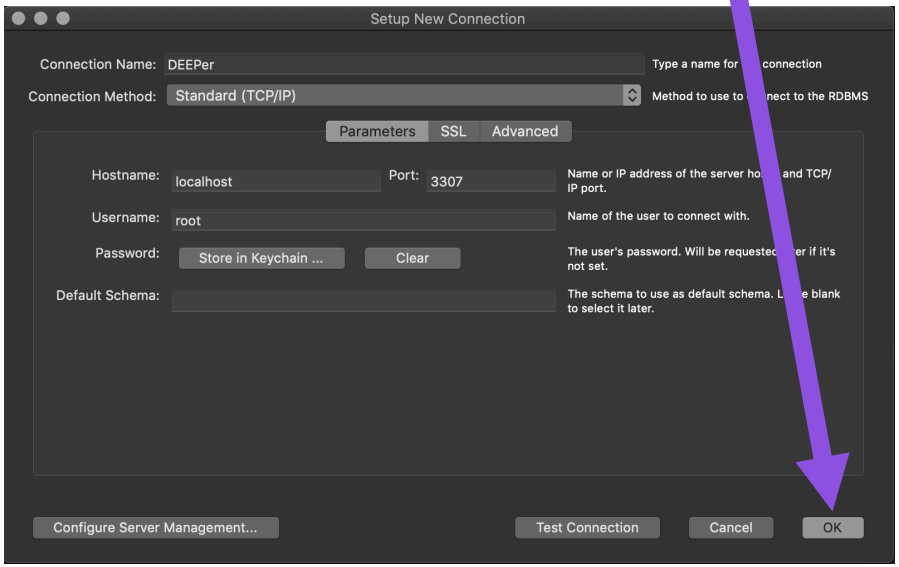


1. Enter Connection Name: `DEEPer`, Hostname: `localhost`, Port: `3307`
2. Next to Password, Click “Store in Keychain …” and enter `root` (all lower-cased) and click “OK”

Connecting To Our DB – Code Along

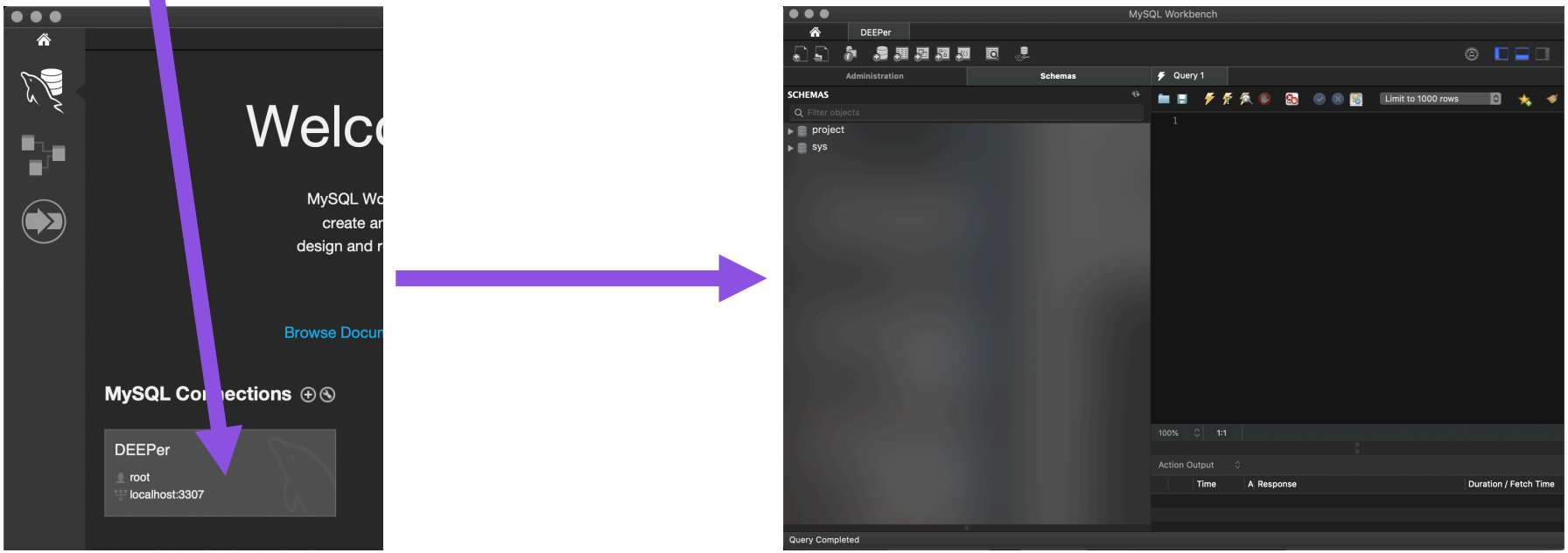
Click “Test Connection” at the bottom. You should see a successful popup.
Click OK to close this popup, then click OK





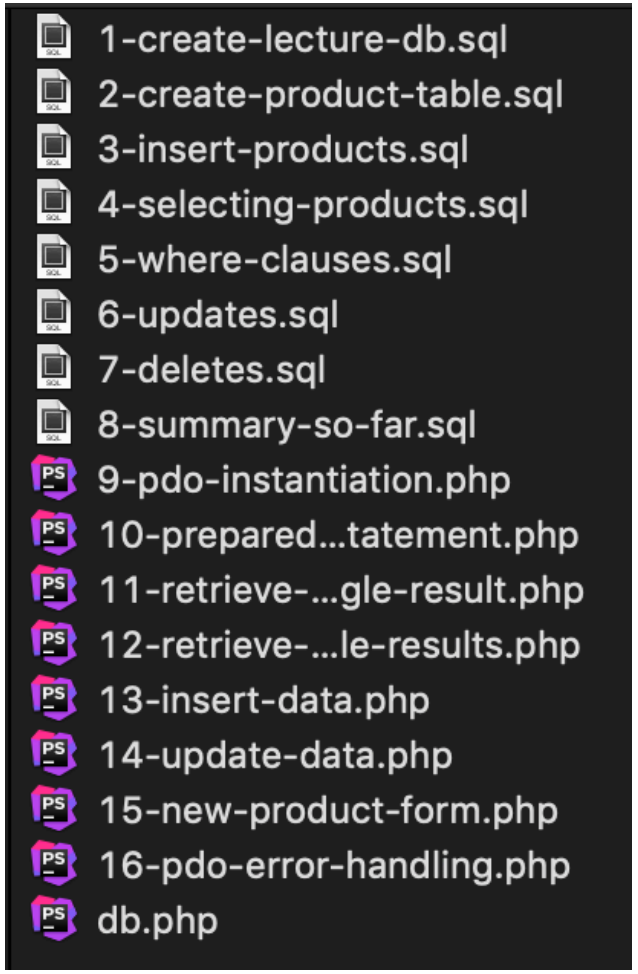
Connecting To Our DB – Code Along

Double-click the new tile at the bottom of the window to connect and open the database connection



Code Alongs

- For this lecture we will provide the code required for each step, but some steps we may recommend typing out
- Download the W4S2-SQL-statements.zip file from the #resources channel in Slack
- Double-click the downloaded zip to extract it as normal
- Open the extracted folder in PHPStorm via `File > Open`, then selecting the the `W4S2-SQL-statements` folder in `Downloads`

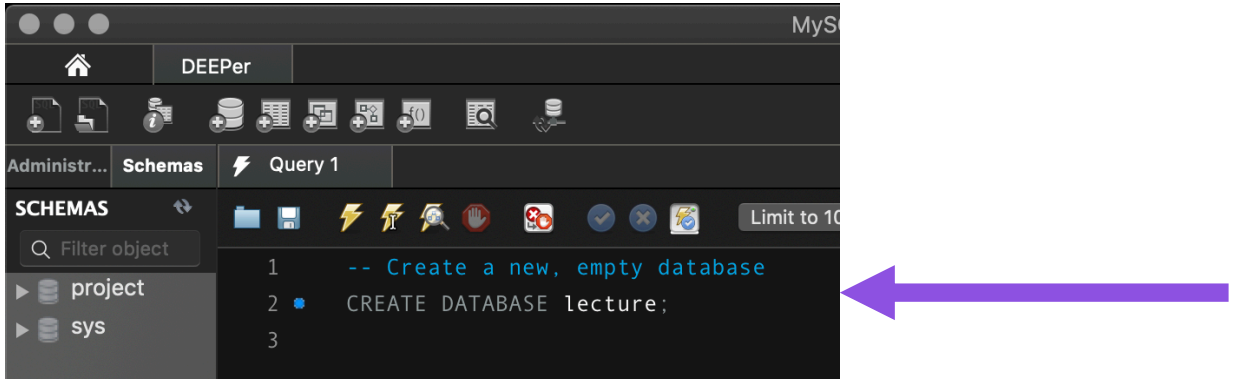


Creating A New Schema – Code Along

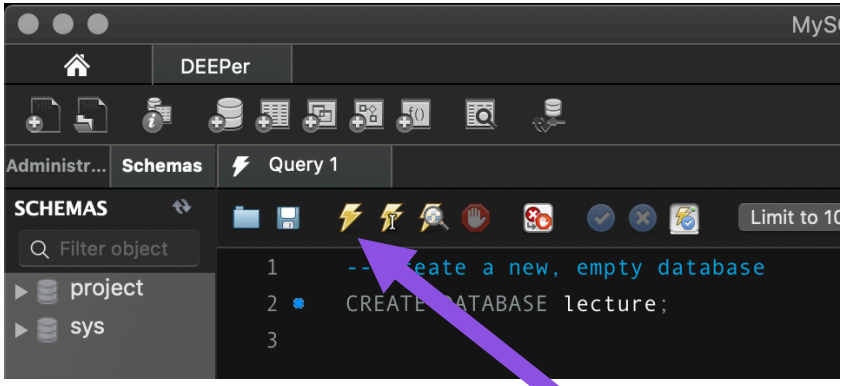
- Creating a new empty database is simple

```
1   -- Create a new, empty database
2   CREATE DATABASE lecture;
```

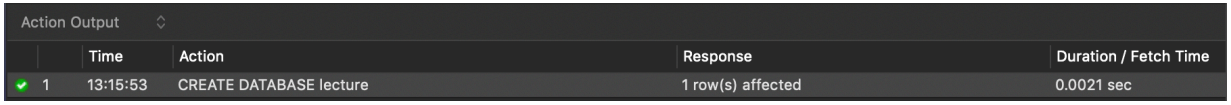
- Open the first file 1-create-lecture-db.sql
- Copy the contents of the file and paste into MySQL Workbench



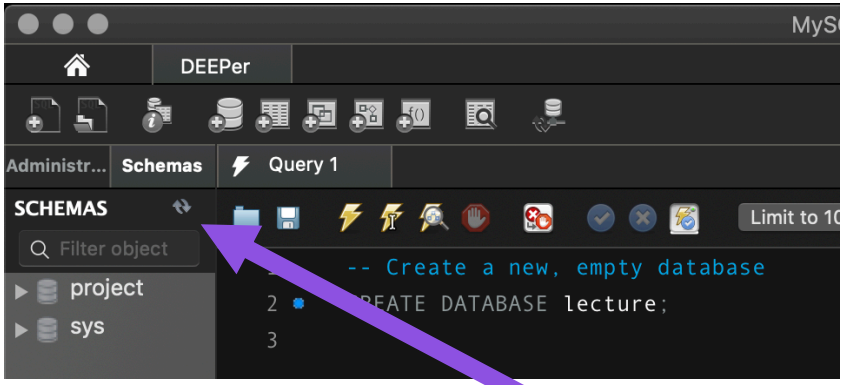
Creating A New Schema – Code Along



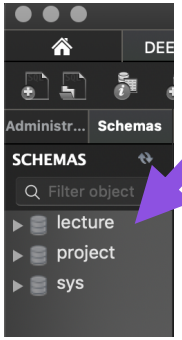
- Click the left-most lightning icon to execute the SQL in the editor
- A successful message should display in the bottom panel



Creating A New Schema – Code Along



- Click the refresh button next to SCHEMAS
- Our new schema will then appear



Creating A Table

- Creating a new table is slightly more complex than creating a schema
- Remembering the syntax is not important, understanding it however is
- You will generally not write this code yourself but use a generator to do so

Creating A Table


```
CREATE TABLE `product_review` (
  -- Columns defined here...
);
```

- The syntax to define the table itself is straightforward
- Table names can include spaces, but **should not**
- Underscores are conventionally used to separate words

Creating A Table – Columns

- Within the create table statement, a comma-separated list of column definitions is included
- In a single column's simplest form, the only parameters required are the name of the column and its data type

```
`columnName` DATATYPE(length)
```

- There are many data types, but the most important ones for today are;
  - `VARCHAR` – variable length string, most common string storage (e.g. for a product's title)
  - `INT` – an integer value (e.g. for a product's ID)

Creating A Table – Columns

- Columns have a few flags that can be set against them, one important flag being whether the column allows null
- By default a column **will** allow null

```
`columnName` DATATYPE(length)
```

- To prevent null values, `NOT NULL` is added after the type

```
`columnName` DATATYPE(length) NOT NULL
```

Creating A Table – Columns


```
`columnName` INT(11) NOT NULL AUTO_INCREMENT
```

- One last important flag is `AUTO_INCREMENT`
- If this is set against a column, its value will be automatically calculated by the database
- This is especially useful for ID columns
- The database will automatically store the next available integer in that column when a row is added
- The column should therefore be an INT column – the length of 11 is a convention, but not mandatory
- An AI column must be defined as a key – more on this later

Creating A Table – Code Along


```
CREATE TABLE `product` (
  `id` INT(11) NOT NULL AUTO_INCREMENT,
  `title` VARCHAR(255) NOT NULL,
  PRIMARY KEY (`id`)
);
```

- Open the second SQL file – `2-create-product-table.sql`
- Run the code in MySQL Workbench as before
- Here we create a product table with two columns;
  - `id` - an integer column, not allowing null and automatically incrementing
  - `title`, a string column, also not allowing null
- Refreshing the schema list and expanding the `lecture` DB will show our new table

Inserting Data

- Now that we have a table, we can insert some data
- This is achieved using an `INSERT` statement

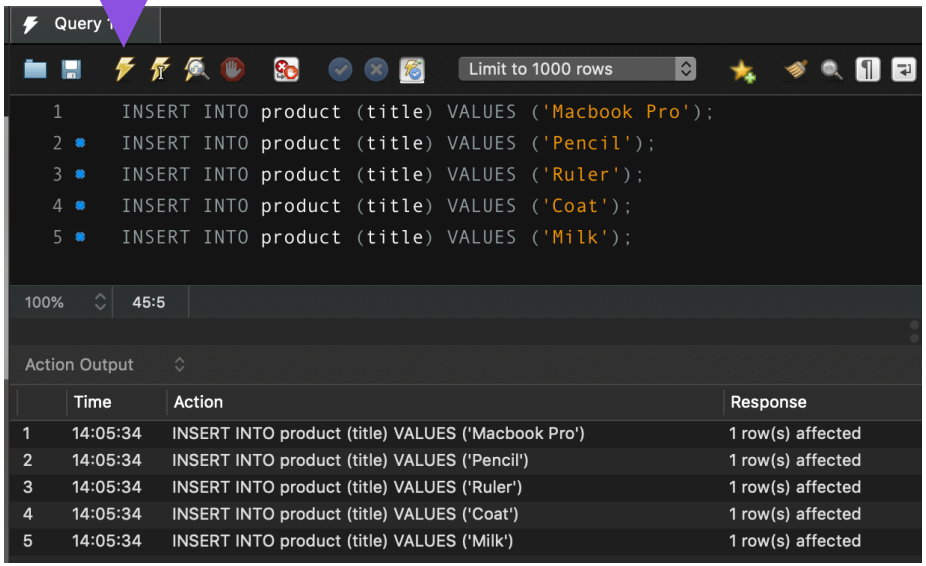
```
INSERT INTO table_name (columnA, columnB) VALUES (valueA, valueB);
```

- For example, using our product table

```
INSERT INTO product (title) VALUES ('My Awesome Product');
```

Inserting Data – Code Along

- Copy and run the SQL from `3-insert-products.sql`
- Remember to use the left-most lightning button to execute
- Five products will be created, which we will confirm next



Selecting Data

- Rows in a table can be pulled out using `SELECT` statements

```
SELECT columnA, columnB FROM table_name;
```

- So using our product table we can write;

```
SELECT id, title FROM product;
```

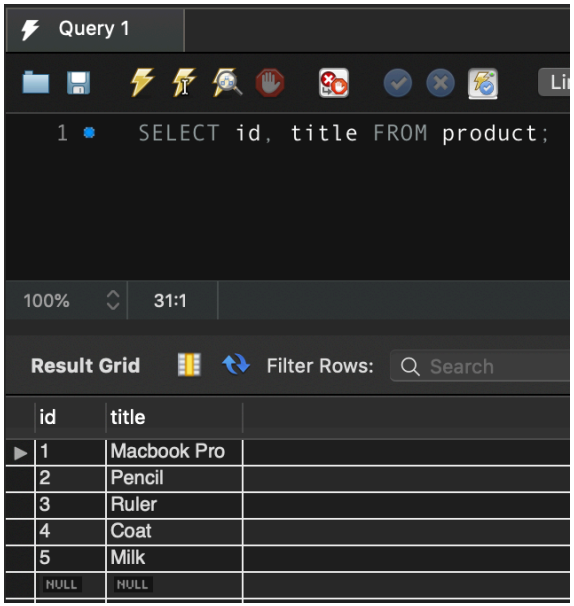
- A * can be used in place of columns if **all** columns are desired

```
SELECT * FROM product;
```

Selecting Data – Code Along

- Run the following SQL statement, or copy from the file `4-selecting-products.sql`

```
SELECT id, title FROM product;
```



We can now see the IDs that were automatically generated during the inserts

Selecting Data – Filtering Results

- In the previous code along we selected all products, which returned all rows in the table
- This is rarely desirable, especially in tables where there may be many thousands of rows
- We can apply `WHERE` clauses to SQL queries to filter the results we wish to return

```
SELECT columnA, columnB FROM table_name WHERE conditions;
```

Filtering Results – Code Along

- Try running each of the `SELECT` statements in `5-where-clauses.sql`

```sql
SELECT id, title FROM product WHERE id = 2;
SELECT id, title FROM product WHERE id > 3;
SELECT id, title FROM product WHERE id IN (2, 4);

SELECT id, title FROM product WHERE title = 'Milk';
SELECT id, title FROM product WHERE title IN ('Ruler', 'Pencil');
```

Updating Rows

- Existing rows can be amended using an `UPDATE` statement

```
UPDATE table_name SET columnA = 'new value';
```

- An `UPDATE` with no `WHERE` clause will apply to **all rows in the table**

```
UPDATE product SET title = 'Protractor';
```

- An `UPDATE` with a `WHERE` clause will only apply to the rows meeting its criteria

```
UPDATE product SET title = 'Protractor' WHERE title = 'Ruler';
```

Deleting Rows

- Deleting rows is, naturally, a dangerous operation
- SQL believes you know what you are doing when running queries
- A `DELETE` statement with no conditions will try to delete **every row in the table**
- Almost always, a `DELETE` should therefore have at least one WHERE clause
- The structure of the `WHERE` clause is identical to selecting or updating

```
DELETE FROM product WHERE id = 4;
```

Safely Updating & Deleting Rows

- If manually running `UPDATE` or `DELETE` statements against a database there is a process to follow to help stay safe;

1. Execute a `SELECT` statement first – `SELECT`s are safe, and you can confirm you have the correct WHERE clauses first
2. Once you are comfortable that only the expected rows are returned, update the statement to a `DELETE`
3. Where possible, especially in sensitive locations, get a colleague to double-check the query before you run it

Summary So Far


```sql
CREATE DATABASE lecture;

CREATE TABLE `product` (
  `id` INT(11) NOT NULL AUTO_INCREMENT,
  `title` VARCHAR(255) NOT NULL,
  PRIMARY KEY (`id`)
);

SELECT id, title FROM product WHERE id = 2;
SELECT id, title FROM product WHERE id > 3;
SELECT id, title FROM product WHERE id IN (2, 4);

UPDATE product SET title = 'Protractor' WHERE title = 'Ruler';

DELETE FROM product WHERE id = 4;
```

02

Databases In PHP

Databases In PHP

- Everything we have done so far has been directly in MySQL Workbench
- When building web applications we want to be able to perform the operations we have seen dynamically
- PHP provides a range of functionality for interacting with databases
- For the purposes of this lecture we will be using PHP Data Objects (PDO)

PDO

- PDO is a class provided by PHP which provides various methods to interact with a database
- In its simplest form, its instantiation requires 3 parameters

```php
<?php

$databaseUserUsername = '...';
$databaseUserPassword = '...';

$dbh = new PDO(
  'mysql:host=localhost;dbname=yourDatabaseName',
  $databaseUserUsername,
  $databaseUserPassword
);
```

PDO – Code Along

- Create a new file - `exercises/week-04/lecture2/db.php`
- Copy the following code from `9-pdo-instantiation.php` or code along
- Test via http://localhost/exercises/week-04/lecture2/db.php

```php
<?php

$username = 'root';
$password = 'root';

try {
    $dbh = new PDO(
        'mysql:host=mysql;dbname=lecture',
        $username,
        $password
    );
} catch (PDOException $e) {
    // We could log this!
    die('Unable to establish a database connection');
}
```

Parameterised Queries

- When writing raw SQL in Workbench, we formed one single statement with values inline
- In PHP however, we **must not** build an SQL query by concatenating it together, **especially if user input is involved**
- Concatenating user input into a query makes the application vulnerable to **SQL Injection Attacks**
- PDO allows us to use placeholders in queries, and provide the list of parameters separately
- User input should still be validated, especially to confirm to business logic, but this helps protect us

Parameterised Queries - Example


```php
<?php

require_once 'db.php';

// We're pretending this comes from user input, e.g. $_GET['productId']
$productId = 3;

// The BAD way
$stmt = $dbh->query('SELECT id, title FROM product WHERE id = ' . $productId);
$product = $stmt->fetch();
```

Parameterised Queries - Example


```php
<?php

require_once 'db.php';

// We're pretending this comes from user input, e.g. $_GET['productId']
$productId = 3;

// The GOOD way
$stmt = $dbh->prepare('SELECT id, title FROM product WHERE id = :productId');
$stmt->execute([
  'productId' => $productId
]);
$product = $stmt->fetch();

var_dump($product);
```

```
array (size=4)
  'id' => string '3' (length=1)
  0 => string '3' (length=1)
  'title' => string 'Ruler' (length=5)
  1 => string 'Ruler' (length=5)
```

Parameterised Queries – Fetch Modes


```
array (size=4)
  'id' => string '3' (length=1)
  0 => string '3' (length=1)
  'title' => string 'Ruler' (length=5)
  1 => string 'Ruler' (length=5)
```

- We can tell PDO which format to return the query result in by providing a **fetch mode**.
- Each fetch mode is a constant on the PDO object. The main ones are;
  - `PDO::FETCH_NUM` – enumerated array
  - `PDO::FETCH_ASSOC` – associative array, keyed by column
  - `PDO::FETCH_BOTH` – the default, returns both together

Parameterised Query – FETCH_ASSOC


```php
// ...
$stmt = $dbh->prepare('SELECT id, title FROM product WHERE id = :productId');
$stmt->execute([
  'productId' => $productId
]);
$product = $stmt->fetch(PDO::FETCH_ASSOC);

var_dump($product);
```

```
array (size=2)
  'id' => string '3' (length=1)
  'title' => string 'Ruler' (length=5)
```

Retrieving A Single Result


```php
// ...
$stmt = $dbh->prepare('SELECT id, title FROM product WHERE id = :productId');
$stmt->execute([
  'productId' => $productId
]);
$product = $stmt->fetch(PDO::FETCH_ASSOC);

var_dump($product);
```

```
array (size=2)
  'id' => string '3' (length=1)
  'title' => string 'Ruler' (length=5)
```

- We saw this example in the Parameterised Query section
- When only a single result is expected, use `$stmt->fetch();`

Retrieving Multiple Results


```php
<?php

require_once 'db.php';

$stmt = $dbh->prepare(
  'SELECT id, title FROM product WHERE id > 2'
);
$stmt->execute();
$products = $stmt->fetchAll(PDO::FETCH_ASSOC);

var_dump($products);
```

```
array (size=3)
  0 =>
    array (size=2)
      'id' => string '3' (length=1)
      'title' => string 'Ruler' (length=5)
  1 =>
    array (size=2)
      'id' => string '4' (length=1)
      'title' => string 'Coat' (length=4)
  2 =>
    array (size=2)
      'id' => string '5' (length=1)
      'title' => string 'Milk' (length=4)
```

- When a query can return multiple results, use `$stmt->fetchAll();`

Inserting, Updating And Deleting Data

- We have already seen all we need to know about Prepared Queries to run `INSERT`, `UPDATE` and `DELETE` queries
- First, prepare a query

```
$stmt = $dbh->prepare('INSERT INTO product (title) VALUES (:title)');
```

- Second, execute the query with any parameters

```
$stmt->execute([
  'title' => $newProductTitle,
]);
```

Inserting Data - Example


```php
<?php

require_once 'db.php';

// We're pretending this comes from user input, e.g. $_POST['title']
$newProductTitle = 'New Product Title';

$stmt = $dbh->prepare('INSERT INTO product (title) VALUES (:title)');

$stmt->execute([
  'title' => $newProductTitle,
]);

// We can optionally check how many rows were affected by the statement
echo '# Rows affected: ' . $stmt->rowCount(); // 1
```

Updating Data - Example

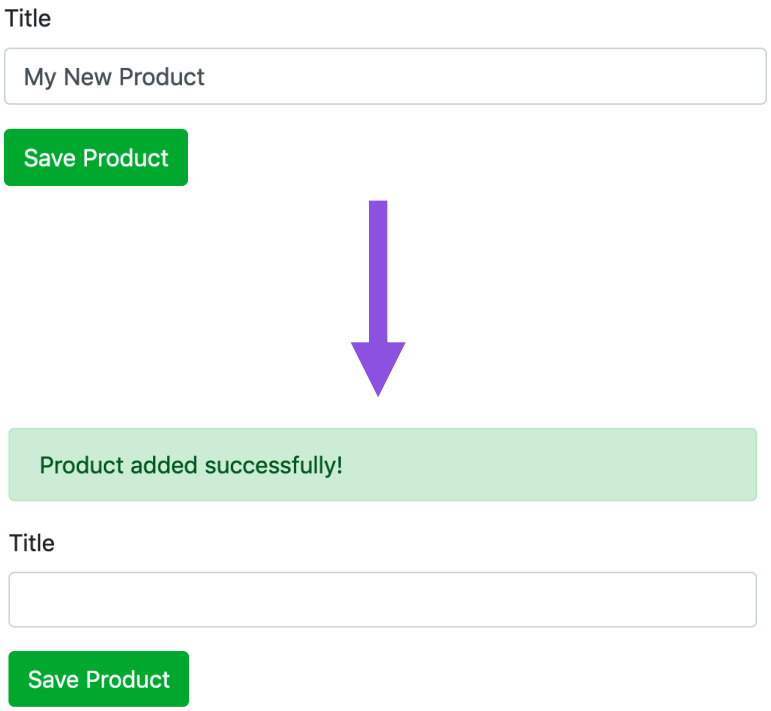

```php
<?php

require_once 'db.php';

// We're pretending this comes from user input, e.g. $_POST['id'], $_POST['title']
$productId = 3;
$productTitle = 'Edited Product Title';

$stmt = $dbh->prepare('UPDATE product SET title = :title WHERE id = :id');

$stmt->execute([
    'id' => $productId,
    'title' => $productTitle,
]);

// We can optionally check how many rows were affected by the statement
echo '# Rows affected: ' . $stmt->rowCount(); // 1
```

Deleting Data - Example


```php
<?php

require_once 'db.php';

// We're pretending this comes from user input, e.g. $_POST['id']
$productId = 3;

$stmt = $dbh->prepare('DELETE FROM product WHERE id = :id');

$stmt->execute([
    'id' => $productId,
]);

// We can optionally check how many rows were affected by the statement
echo '# Rows affected: ' . $stmt->rowCount(); // 1
```

Inserting Data – A Full Example


```php
<?php

require_once 'db.php';

$productAddedSuccess = false;

if (!empty($_POST)) {
    $newProductTitle = $_POST['title'];

    $stmt = $dbh->prepare('INSERT INTO product (title) VALUES (:title)');

    $stmt->execute([
        'title' => $newProductTitle,
    ]);

    $productAddedSuccess = true;
}
?>

...
<form action="" method="post">

    <?php if ($productAddedSuccess): ?>
    <p class="alert alert-success">Product added successfully!</p>
    <?php endif; ?>

    <div class="form-group">
        <label for="product-title">Title</label>
        <input type="text" name="title" id="product-title" class="form-control">
    </div>

    <button type="submit" class="btn btn-success">Save Product</button>
</form>
...
```



Handling Query Errors

- By default, PDO hides a lot of errors from the developer and user
- For example, the following code does not trigger any kind of error or exception

```php
<?php

$stmt = $dbh->prepare('This is not valid SQL');
$stmt->execute();
```

Handling Query Errors

We can set a flag against the PDO instance to force it to throw exceptions when query errors occur

```php
// In db.php
$dbh = new PDO(
    'mysql:host=mysql;dbname=lecture',
    $username,
    $password
);

// This line enables exceptions on error
$dbh->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
```

Handling Query Errors

Now when we rerun the invalid query, we get an exception

Fatal error: Uncaught PDOException: SQLSTATE[42000]: Syntax error or access violation: 1064 You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'This is not valid SQL' at line 1 in /var/www/sites /deeper.local/exercises/week-04/lecture2/prepared-statements.php on line *13*

PDOException: SQLSTATE[42000]: Syntax error or access violation: 1064 You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'This is not valid SQL' at line 1 in /var/www/sites/deeper.local/exercises/week-04/lecture2/prepared-statements.php on line *13*

Call Stack

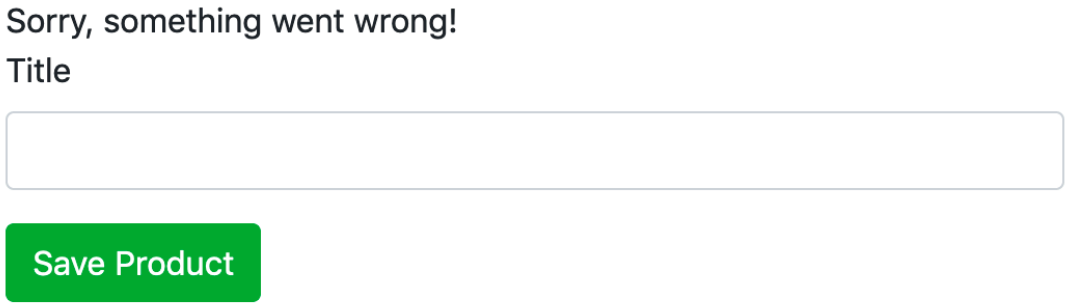
#	Time	Memory	Function	Location
1	0.0078	365472	{main}( )	.../prepared-statements.php:0
2	0.0160	414824	execute ( )	.../prepared-statements.php:13

This however introduces another issue – our SQL errors are being outputted to the user, very bad news!

Handling Query Errors

To resolve this, for now, all queries should **always** be wrapped in a try/catch block, handling any PDO exceptions

```php
try {
  $stmt = $dbh->prepare('This is not valid SQL');
  $stmt->execute();
} catch (PDOException $e) {
  // Appropriate error handling, logging etc
  echo 'Sorry, something went wrong!';
}
```



PDO Summary

Establish a database connection

```php
$username = 'root';
$password = 'root';

try {
    $dbh = new PDO(
        'mysql:host=mysql;dbname=lecture',
        $username,
        $password
    );
} catch (PDOException $e) {
    // We could log this!
    die('Unable to establish a database connection');
}
```

PDO Summary

Prepare a query, execute it, and if it is retrieving data, use `fetch/fetchAll`

```
$stmt = $dbh->prepare('SELECT id, title FROM product WHERE id = :productId');
$stmt->execute([
  'productId' => 3,
]);
$product = $stmt->fetch(PDO::FETCH_ASSOC);
```