

# Working with Databases

## Chapter 11

Section 1 of 7

# **DATABASES AND WEB DEVELOPMENT**

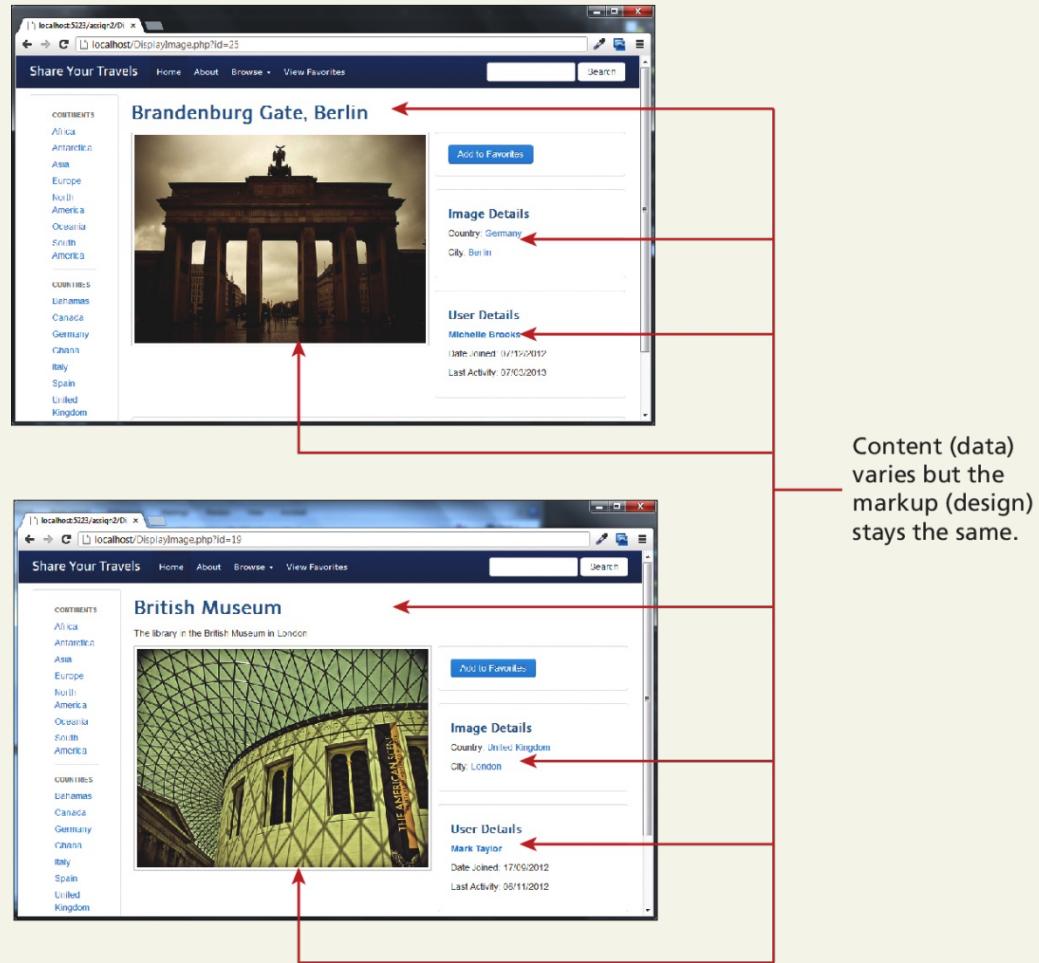
# The Role of Databases

In Web Development

Databases provide a way to implement one of the most important software design principles:

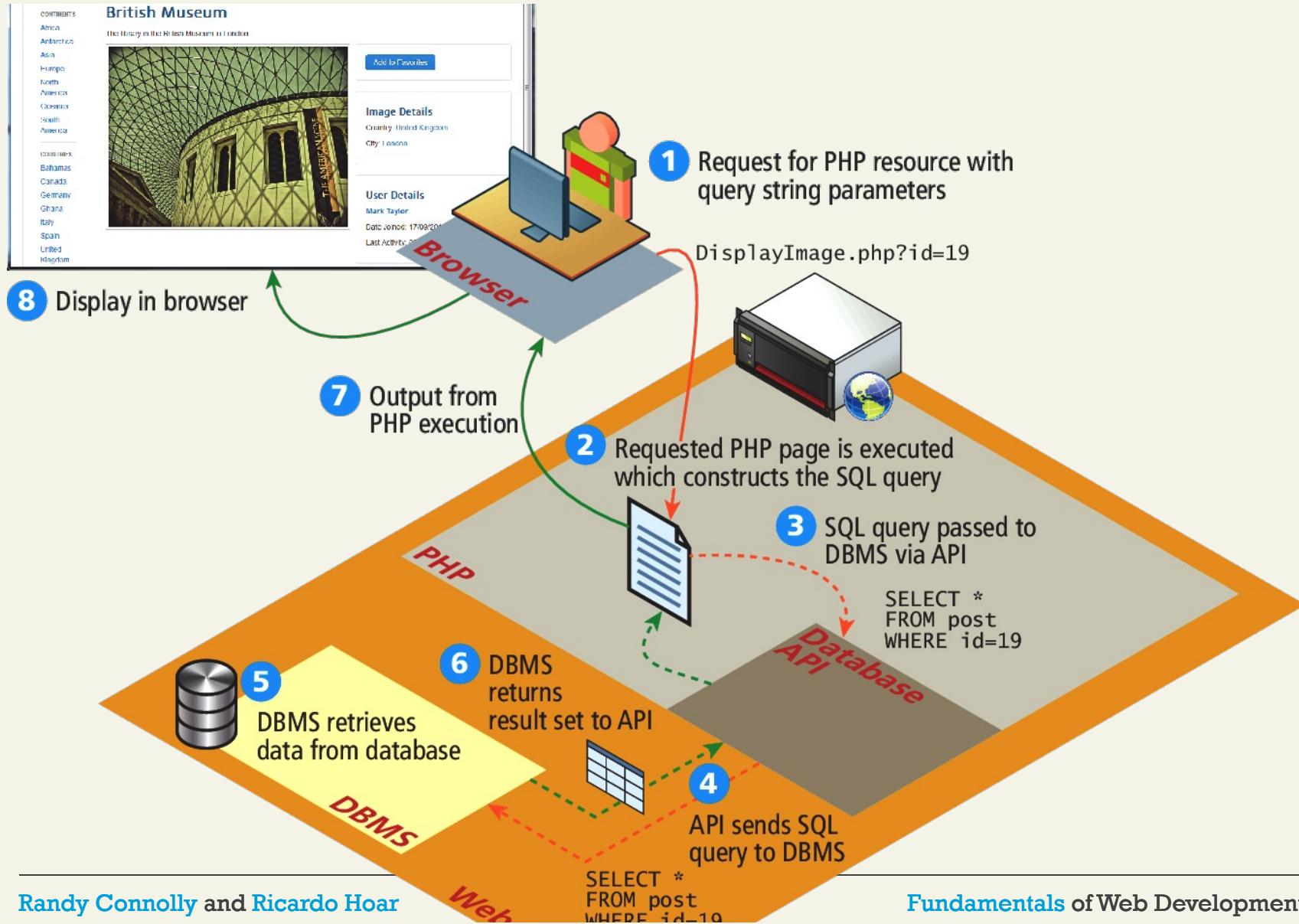
*one should separate that which varies from that which stays the same*

The program (PHP) determines which data to display, often from information in the GET or POST query string, and then uses a database API to interact with the database



# That Which Changes

Can be stored in the DB



# Transactions

An Advanced Topic.

Anytime one of your PHP pages makes changes to the database via an UPDATE, INSERT, or DELETE statement, you also need to be concerned with the possibility of failure.

A **transaction** refers to a sequence of steps that are treated as a single unit, and provide a way to gracefully handle errors and keep your data properly consistent when errors do occur.

# Transactions

## An Example

Imagine how a purchase would work in a web storefront. After the user has verified the shipping address, entered a credit card, and selected a shipping option and clicks the final *Pay for Order* button? Imagine that the following steps need to happen.

1. Write order records to the website database.
2. Check credit card service to see if payment is accepted.
3. If payment is accepted, send message to legacy ordering system.
4. Remove purchased item from warehouse inventory table.
5. Send message to shipping provider.

# Transactions

## An Example

At any step in this process, errors could occur. For instance:

- The DBMS system could crash after writing the first order record but before the second order record could be written.
- The credit card service could be unresponsive, or the credit card payment declined.
- The legacy ordering system or inventory system or shipping provider system could be down.

# Transactions

Multiple types

**Local Transactions** can be handled by the DBMS.

**Distributed Transactions** involve multiple hosts, several of which we may have no control over.

Distributed transactions are much more complicated than local transactions

# Local Transactions

The easy transactions

The SQL for transactions use the START TRANSACTION, COMMIT, and ROLLBACK commands

```
/* By starting the transaction, all database modifications within  
the transaction will only be permanently saved in the database  
if they all work */  
  
START TRANSACTION  
  
INSERT INTO orders ...  
INSERT INTO orderDetails ...  
UPDATE inventory ...  
  
/* if we have made it here everything has worked so commit changes */  
COMMIT  
  
/* if we replace COMMIT with ROLLBACK then the three database  
changes would be "undone" */
```

**LISTING 11.1** SQL commands for transaction processing

# Data Definition Statements

**Data Manipulation Language** features of SQL are what we typically describe in web development and include the SELECT, UPDATE, INSERT, and DELETE.

There is also a **Data Definition Language (DDL)** in SQL, which is used for creating tables, modifying the structure of a table, deleting tables, and creating and deleting databases. You may find yourself using them indirectly within something like the **phpMyAdmin** management tool, although a text syntax does exist for those interested.

Section 3 of 7

# DATABASE APIS

# API

Application Programming Interface

**API** stands for application programming interface and in general refers to the classes, methods, functions, and variables that your application uses to perform some task.

Some database APIs work only with a specific type of database; others are cross-platform and can work with multiple databases.

# PHP MySQL APIs

There is more than 1

- **MySQL extension.** This was the original extension to PHP for working with MySQL and has been replaced with the newer mysqli extension. This procedural API should now only be used with versions of MySQL older than 4.1.3.
- **mysqli extension.** The MySQL Improved extension takes advantage of features of versions of MySQL after 4.1.3. This extension provides **both a procedural and an object-oriented approach**. This extension also supports most of the latest features of MySQL.
- **PHP data objects (PDOs).** This object-oriented API has been available since PHP 5.1 and provides an **abstraction layer** (i.e., a set of classes that hide the implementation details for some set of functionality) that with the appropriate drivers can be used with *any* database, and not just MySQL databases. However, it is not able to make use of all the latest features of MySQL.

We will show how to do some of the most common database operations using the **procedural mysqli** extension as well as the **object-oriented PDO**.

Section 4 of 7

# MANAGING A MYSQL DATABASE

# How do you access the DBMS

So, so many ways

Although you will eventually be able to manipulate the database from your PHP code, there are some routine maintenance operations that do not warrant custom code. Tools include:

- Command Line Interface
- phpMyAdmin
- MySQLWorkbench

# Command Line Interface

Oldie but a goodie.

To launch an interactive MySQL command-line session, you must specify the host, username, and database name to connect to as shown below:

```
mysql -h 192.168.1.14 -u bookUser -p
```

Once inside of a session, you may enter any SQL query, terminated with a semicolon (;

# Command Line Interface

Oldie but a goodie.

```
Database changed
mysql> SHOW TABLES;
+-----+
| Tables_in_book_database |
+-----+
| authors
| bindingtypes
| bookauthors
| books
| categories
| disciplines
| imprints
| productionstatuses
| subcategories
+-----+
9 rows in set (0.00 sec)

mysql> SHOW COLUMNS IN authors;
+-----+-----+-----+-----+-----+-----+
| Field | Type   | Null | Key | Default | Extra       |
+-----+-----+-----+-----+-----+-----+
| ID    | int(11) | NO   | PRI | NULL    | auto_increment |
| FirstName | varchar(255) | YES  |     | NULL    |               |
| LastName | varchar(255) | YES  |     | NULL    |               |
| Institution | varchar(255) | YES  |     | NULL    |               |
+-----+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)

mysql> SELECT * FROM authors WHERE FirstName LIKE "A%";
+-----+-----+-----+-----+
| ID  | FirstName | LastName | Institution          |
+-----+-----+-----+-----+
| 2   | Andrew    | Abel     | Wharton School of the University of Pennsylvania
| 25  | Allen     | Center   | NULL
| 37  | Allen     | Dooley   | Santa Ana College
| 40  | Andrew    | DuBrin   | Rochester Institute of Technology
| 56  | Allan     | Hambley  | NULL
| 57  | Arden     | Hamer    | Indiana University of Pennsylvania
| 82  | Arthur    | Keown    | Virginia Polytechnic Instit. and State University
| 102 | Annie     | McKee    | NULL
| 119 | Arthur    | O'Sullivan | NULL
| 172 | Allyn     | Washington | Dutchess Community College
| 194 | Anne Frances | Wysocki  | University of Wisconsin, Milwaukee
| 198 | Alice M.   | Gillam    | University of Wisconsin-Milwaukee
| 214 | Anthony P. | O'Brien   | Lehigh University
| 216 | Alvin C.   | Burns     | NULL
| 225 | Abbey      | Deitel    | NULL
| 252 | Alvin      | Arens     | Michigan State University
| 258 | Ali         | Owlia     | NULL
| 270 | Anne        | Winkler   | NULL
| 275 | Alan        | Marks     | DeVry University
+-----+-----+-----+-----+
19 rows in set (0.00 sec)
```

# Command Line Interface

I bet you'll use it oneday...

In addition to the interactive prompt, the command line can be used to import and export entire databases or run a batch of SQL commands from a file.

To import commands from a file called *commands.sql*, for example, we would use the < redirection:

```
mysql -h 192.168.1.14 -u bookUser -p < commands.sql
```

Although GUI tools allow this as well, the CLI can be integrated into automated scripts to aid with mirroring, backup, and recovery.

# phpMyAdmin

Will even generate PHP code

A popular web-based front-end (written in PHP) called **phpMyAdmin** allows developers to access management tools through a web portal.

MySQL has a number of predefined databases it uses for its own operation.

phpMyAdmin allows you to view and manipulate any table in a database.

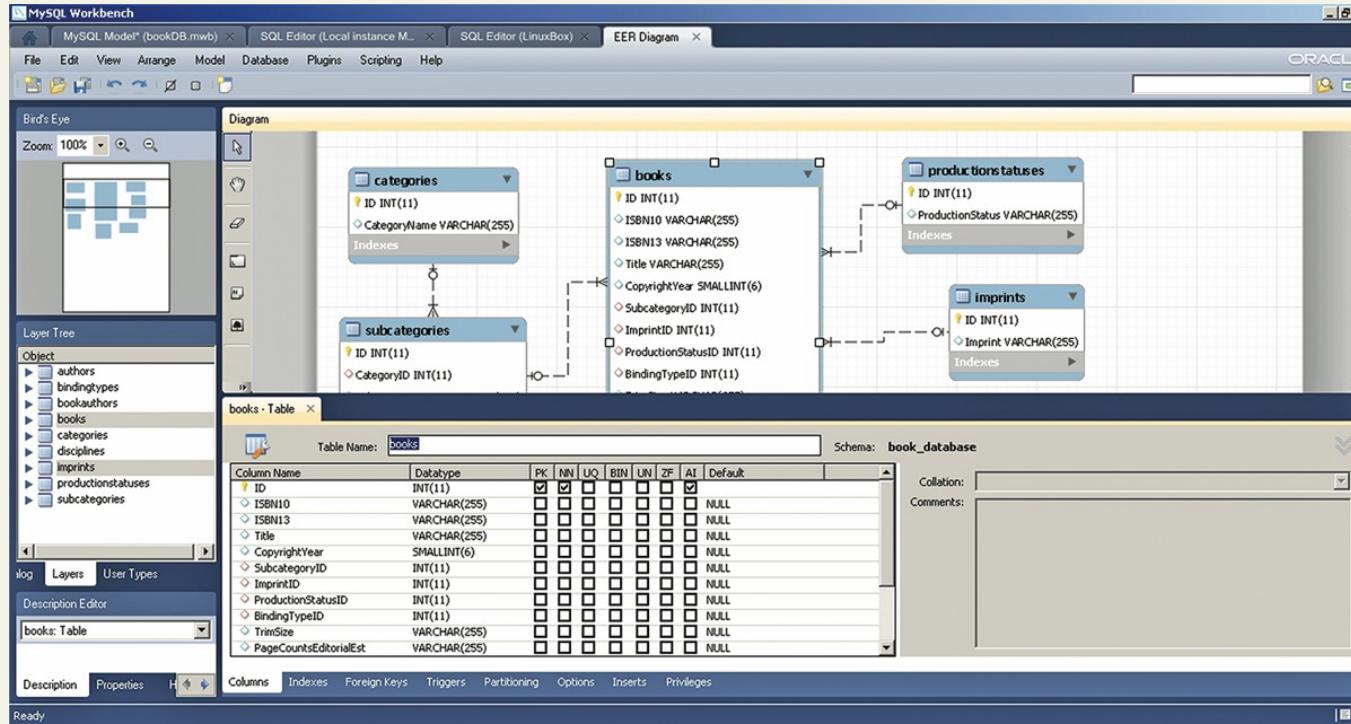
The screenshot displays the phpMyAdmin interface with two windows open. The top window shows the 'General Settings' and 'Database server' panes. The 'General Settings' pane includes fields for 'Change password' and 'General connection collation' set to 'utf8\_general\_ci'. The 'Database server' pane lists MySQL server details: Server: 127.0.0.1 via TCP/IP, Software: MySQL, Software version: 5.5.27 - MySQL Community Server (GPL), Protocol version: 10, User: root@localhost, and Server charset: UTF-8 Unicode (utf8). The bottom window shows the 'Structure' and 'SQL' tabs selected for the 'books' database. The table structure for 'books' is displayed, showing columns like id, title, author\_id, and category\_id. A large table below lists 1,000 rows from the 'books' table, with columns for id, title, author\_id, category\_id, and size.

id	title	author_id	category_id	size
1	Book 1	1	1	1000
2	Book 2	2	2	1000
3	Book 3	3	3	1000
4	Book 4	4	4	1000
5	Book 5	5	5	1000
6	Book 6	6	6	1000
7	Book 7	7	7	1000
8	Book 8	8	8	1000
9	Book 9	9	9	1000
10	Book 10	10	10	1000
11	Book 11	11	11	1000
12	Book 12	12	12	1000
13	Book 13	13	13	1000
14	Book 14	14	14	1000
15	Book 15	15	15	1000
16	Book 16	16	16	1000
17	Book 17	17	17	1000
18	Book 18	18	18	1000
19	Book 19	19	19	1000
20	Book 20	20	20	1000
21	Book 21	21	21	1000
22	Book 22	22	22	1000
23	Book 23	23	23	1000
24	Book 24	24	24	1000
25	Book 25	25	25	1000
26	Book 26	26	26	1000
27	Book 27	27	27	1000
28	Book 28	28	28	1000
29	Book 29	29	29	1000
30	Book 30	30	30	1000
31	Book 31	31	31	1000
32	Book 32	32	32	1000
33	Book 33	33	33	1000
34	Book 34	34	34	1000
35	Book 35	35	35	1000
36	Book 36	36	36	1000
37	Book 37	37	37	1000
38	Book 38	38	38	1000
39	Book 39	39	39	1000
40	Book 40	40	40	1000
41	Book 41	41	41	1000
42	Book 42	42	42	1000
43	Book 43	43	43	1000
44	Book 44	44	44	1000
45	Book 45	45	45	1000
46	Book 46	46	46	1000
47	Book 47	47	47	1000
48	Book 48	48	48	1000
49	Book 49	49	49	1000
50	Book 50	50	50	1000
51	Book 51	51	51	1000
52	Book 52	52	52	1000
53	Book 53	53	53	1000
54	Book 54	54	54	1000
55	Book 55	55	55	1000
56	Book 56	56	56	1000
57	Book 57	57	57	1000
58	Book 58	58	58	1000
59	Book 59	59	59	1000
60	Book 60	60	60	1000
61	Book 61	61	61	1000
62	Book 62	62	62	1000
63	Book 63	63	63	1000
64	Book 64	64	64	1000
65	Book 65	65	65	1000
66	Book 66	66	66	1000
67	Book 67	67	67	1000
68	Book 68	68	68	1000
69	Book 69	69	69	1000
70	Book 70	70	70	1000
71	Book 71	71	71	1000
72	Book 72	72	72	1000
73	Book 73	73	73	1000
74	Book 74	74	74	1000
75	Book 75	75	75	1000
76	Book 76	76	76	1000
77	Book 77	77	77	1000
78	Book 78	78	78	1000
79	Book 79	79	79	1000
80	Book 80	80	80	1000
81	Book 81	81	81	1000
82	Book 82	82	82	1000
83	Book 83	83	83	1000
84	Book 84	84	84	1000
85	Book 85	85	85	1000
86	Book 86	86	86	1000
87	Book 87	87	87	1000
88	Book 88	88	88	1000
89	Book 89	89	89	1000
90	Book 90	90	90	1000
91	Book 91	91	91	1000
92	Book 92	92	92	1000
93	Book 93	93	93	1000
94	Book 94	94	94	1000
95	Book 95	95	95	1000
96	Book 96	96	96	1000
97	Book 97	97	97	1000
98	Book 98	98	98	1000
99	Book 99	99	99	1000
100	Book 100	100	100	1000

# MySQL Workbench

Powerful design tools

The MySQL Workbench is a free tool from Oracle to work with MySQL databases.



Section 5 of 7

# ACCESSING MYSQL IN PHP

# Database Connection Algorithm

No matter what API you use, the basic interaction with a database is the same:

1. Connect to the database.
2. Handle connection errors.
3. Execute the SQL query.
4. Process the results.
5. Free resources and close connection.

# Database Connection Algorithm

```
<?php

try {
    $connString = "mysql:host=localhost;dbname=bookcrm";
    $user = "testuser";
    $pass = "mypassword";

    $pdo = new PDO($connString, $user, $pass);
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

    $sql = "select * from Categories order by CategoryName";
    $result = $pdo->query($sql);

    while ($row = $result->fetch()) {
        echo $row['ID'] . " - " . $row['CategoryName'] . "<br/>";
    }
    $pdo = null;
}

catch (PDOException $e) {
    die( $e->getMessage() );
}

?>
```

# Database Connection Algorithm

An illustration through example

```
// modify these variables for your installation
$host = "localhost";
$database = "bookcrm";
$user = "testuser";
$pass = "mypassword";

$connection = mysqli_connect($host, $user, $pass, $database);
```

**LISTING 11.3** Connecting to a database with mysqli (procedural)

```
// modify these variables for your installation
$connectionString = "mysql:host=localhost;dbname=bookcrm";
$user = "testuser";
$pass = "mypassword";

$pdo = new PDO($connectionString, $user, $pass);
```

**LISTING 11.4** Connecting to a database with PDO (object-oriented)

# Storing Connection Details

Hard-coding the database connection details in your code is not ideal.

Connection details almost always change as a site moves from development, to testing, to production.

We should move these connection details out of our connection code and place it in some central location.

```
<?php  
define('DBHOST', 'localhost');  
define('DBNAME', 'bookcrm');  
define('DBUSER', 'testuser');  
define('DBPASS', 'mypassword');  
?>
```

**LISTING 11.5** Defining connection details via constants in a separate file (config.php)

# Handling Connection Errors

We need to handle potential connection errors in our code.

- Procedural **mysqli** techniques use conditional (if...else) statements on the returned object from the connection attempt.
- The **PDO** technique uses try-catch which relies on thrown exceptions when an error occurs.

# Handling Connection Errors

```
$connection = mysqli_connect(DBHOST, DBUSER, DBPASS, DBNAME);

// mysqli_connect_errno returns the last error code
if ( mysqli_connect_errno() ) {
    die( mysqli_connect_error() ); // die() is equivalent to exit()
}
```

**LISTING 11.8** Handling connection errors with mysqli (version 2)

# Handling Connection Errors

Object-Oriented PDO with try-catch

```
try {
    $connString = "mysql:host=localhost;dbname=bookcrm";
    $user = "DBUSER";
    $pass = "DBPASS";

    $pdo = new PDO($connString,$user,$pass);
    ...
}
catch (PDOException $e) {
    die( $e->getMessage() );
}
```

**LISTING 11.9** Handling connection errors with PDO

In addition, PDO has 3 different error modes, that allow you to control when errors are thrown.

# Execute the Query

Procedural and Object-Oriented

```
$sql = "SELECT * FROM Categories ORDER BY CategoryName";  
  
// returns a mysqli_result object  
$result = mysqli_query($connection, $sql);
```

**LISTING 11.11** Executing a SELECT query (mysqli)

```
$sql = "SELECT * FROM Categories ORDER BY CategoryName";  
  
// returns a PDOStatement object  
$result = $pdo->query($sql);
```

**LISTING 11.12** Executing a SELECT query (pdo)

Both return a **result set**, which is a type of cursor or pointer to the returned data

# Queries that don't return data

Procedural and Object-Oriented

```
$sql = "UPDATE Categories SET CategoryName='Web' WHERE  
    CategoryName='Business'";  
  
if ( mysqli_query($connection, $sql) ) {  
    $count = mysqli_affected_rows($connection);  
    echo "<p>Updated " . $count . " rows</p>";  
}
```

**LISTING 11.13** Executing a query that doesn't return data (mysqli)

```
$sql = "UPDATE Categories SET CategoryName='Web' WHERE  
    CategoryName='Business"';  
$count = $pdo->exec($sql);  
echo "<p>Updated " . $count . " rows</p>";
```

**LISTING 11.14** Executing a query that doesn't return data (PDO)

# Integrating User Data

Say, using an HTML form posted to the PHP script

Browser – Rename Category Form

Category to change: English

New category name: Communications

Save

```
<form method="post" action="rename.php">
  <input type="text" name="old" /><br/>
  <input type="text" name="new" /><br/>
  <input type="submit" />
</form>
```

`$_POST['old']`

English

`$_POST['new']`

Communications

UPDATE Categories SET CategoryName='English' WHERE CategoryName='Communications'

# Integrating User Data

Not everyone is nice.

```
$from = $_POST['old'];
$to = $_POST['new'];
$sql = "UPDATE Categories SET CategoryName='$to' WHERE
        CategoryName='$from'";

$count = $pdo->exec($sql);
```

**LISTING 11.15** Integrating user input into a query (first attempt)

While this does work, it opens our site to one of the most common web security attacks, the **SQL injection attack**.

# SQL Injection Illustration



- 0 A vulnerable form passes unsanitized user input directly into SQL queries.

User   
Password

POST



- 1 Hacker inputs SQL code into a text field and submits the form.

User   
Password

POST

```
...  
$user = $_POST['username'];  
$pass = $_POST['pass'];  
$sql = "SELECT * FROM Users WHERE  
    uname='$user' AND passwd=MD5('$pass')";  
sql_query($sql);  
...
```

- 2 PHP script puts the raw fields directly into the SQL query.

```
SELECT * FROM Users WHERE  
    uname='alice' AND  
    passwd=MD5('abcd')
```

Users table



```
SELECT * FROM Users WHERE uname='';  
TRUNCATE TABLE Users;  
# ' AND passwd=MD5('')
```

- 3 The resulting query is actually two queries.

Rest of query  
is commented  
out

Users table



- 4 All records in Users table are deleted.

# Defend against attack

Distrust user input

The SQL injection class of attack can be protected against by

- **Sanitizing user input**
- **Using Prepared Statements**

# Sanitize User Input

Quick and easy

Each database system has functions to remove any special characters from a desired piece of text. User input can be sanitized in PHP using the **mysqli\_real\_escape\_string()** method or, if using PDO, the **quote()** method

```
$from = $pdo->quote($from);
$to = $pdo->quote($to);
$sql = "UPDATE Categories SET CategoryName=$to WHERE
        CategoryName=$from";

$count = $pdo->exec($sql);
```

**LISTING 11.16** Sanitizing user input before use in an SQL query

# Prepared Statements

Better in general

A **prepared statement** is actually a way to improve performance for queries that need to be executed multiple times.

When MySQL creates a prepared statement, it does something akin to a compiler in that it optimizes it so that it has superior performance for multiple requests.

It also integrates sanitization into each user input automatically, thereby protecting us from SQL injection.

# Prepared Statements

mysqli

non bindm ma bind\_param

```
// retrieve parameter value from query string
$id = $_GET['id'];

// construct parameterized query - notice the ? parameter
$sql = "SELECT Title, CopyrightYear FROM Books WHERE ID=?";

// create a prepared statement

if ($statement = mysqli_prepare($connection, $sql)) {
    // Bind parameters s - string, b - blob, i - int, etc
    mysqli_stmt_bindm($statement, 'i', $id);

    // execute query
    mysqli_stmt_execute($statement);

    // Learn in next section how to access the returned data
    ...
}
```

LISTING 11.17 Using a prepared statement (mysqli)

# Prepared Statements

PDO

```
// retrieve parameter value from query string
$id = $_GET['id'];

/* method 1 */
$sql = "SELECT Title, CopyrightYear FROM Books WHERE ID = ?";
$statement = $pdo->prepare($sql);
$statement->bindValue(1, $id);
$statement->execute();

/* method 2 */
$sql = "SELECT Title, CopyrightYear FROM Books WHERE ID = :id";
$statement = $pdo->prepare($sql);
$statement->bindValue(':id', $id);
$statement->execute();
```

**LISTING 11.18** Using a prepared statement (PDO)

# Prepared Statements

Comparison of two techniques

```
/* technique 1 - question mark placeholders */
$sql = "INSERT INTO books (ISBN10, Title, CopyrightYear, ImprintId,
    ProductionStatusId, TrimSize, Description) VALUES
        (?, ?, ?, ?, ?, ?, ?)";
$statement = $pdo->prepare($sql);
$statement->bindValue(1, $_POST['isbn']);
$statement->bindValue(2, $_POST['title']);
$statement->bindValue(3, $_POST['year']);
$statement->bindValue(4, $_POST['imprint']);
$statement->bindValue(5, $_POST['status']);
$statement->bindValue(6, $_POST['size']);
$statement->bindValue(7, $_POST['desc']);
$statement->execute();

/* technique 2 - named parameters */
$sql = "INSERT INTO books (ISBN10, Title, CopyrightYear, ImprintId,
    ProductionStatusId, TrimSize, Description) VALUES (:isbn,
        :title, :year, :imprint, :status, :size, :desc) ";
$statement = $pdo->prepare($sql);
$statement->bindValue(':isbn', $_POST['isbn']);
$statement->bindValue(':title', $_POST['title']);
$statement->bindValue(':year', $_POST['year']);
$statement->bindValue(':imprint', $_POST['imprint']);
$statement->bindValue(':status', $_POST['status']);
$statement->bindValue(':size', $_POST['size']);
$statement->bindValue(':desc', $_POST['desc']);
$statement->execute();
```

# Process Query Results

```
$sql = "SELECT * FROM Categories ORDER BY CategoryName";
// run the query
if ($result = mysqli_query($connection, $sql)) {
    // fetch a record from result set into an associative array
    while($row = mysqli_fetch_assoc($result))
    {
        // the keys match the field names from the table
        echo $row['ID'] . " - " . $row['CategoryName'];
        echo "<br/>";
    }
}
```

**LISTING 11.20** Looping through the result set (mysqli—not prepared statements)

```
$sql = "SELECT * FROM Categories ORDER BY CategoryName";
$result = $pdo->query($sql);

while ( $row = $result->fetch() ) {
    echo $row['ID'] . " - " . $row['CategoryName'] . "<br/>";
}
```

**LISTING 11.22** Looping through the result set (PDO)

# Process Query Results

Mysqli – using prepared statements

```
$sql = "SELECT Title, CopyrightYear FROM Books WHERE ID=?";
if ($statement = mysqli_prepare($connection, $sql)) {

    mysqli_stmt_bindm($statement, 'i', $id);
    mysqli_stmt_execute($statement);

    // bind result variables
    mysqli_stmt_bind_result($statement, $title, $year);

    // loop through the data
    while (mysqli_stmt_fetch($statement)) {
        echo $title . '-' . $year . '<br/>';
    }
}
```

**LISTING 11.21** Looping through the result set (mysqli—using prepared statements)

# Fetch into an object

Instead of an array

Consider the following (very simplified) class:

```
class Book {  
  
    public $id;  
    public $title;  
    public $copyrightYear;  
    public $description;  
}
```

# Fetch into an object

Instead of an array

```
$id = $_GET['id'];
$sql = "SELECT id, title, copyrightYear, description FROM Books
        WHERE id= ?";
$statement = $pdo->prepare($sql);
$statement->bindValue(1, $id);
$statement->execute();

$b = $statement->fetchObject('Book');
echo 'ID: ' . $b->id . '<br/>';
echo 'Title: ' . $b->title . '<br/>';
echo 'Year: ' . $b->copyrightYear . '<br/>';
echo 'Description: ' . $b->description . '<br/>';
```

**LISTING 11.23** Populating an object from a result set (PDO)

The property names must match **exactly** (including the case) the field names in the table(s) in the query

# Fetch into an object

A more flexible example, where class names needn't match DB fields

```
class Book {
    public $id;
    public $title;
    public $copyrightYear;
    public $description;

    function __construct($record)
    {
        // the references to the field names in associative array must
        // match the case in the table
        $this->id = $record['ID'];
        $this->title = $record['Title'];
        $this->copyrightYear = $record['CopyrightYear'];
        $this->description = $record['Description'];
    }
}

...
// in some other page or class
$statement->execute();

// using the Book class
$b = new Book($statement->fetch());
echo 'ID: ' . $b->id . '<br/>';
echo 'Title: ' . $b->title . '<br/>';
echo 'Copyright Year: ' . $b->copyrightYear . '<br/>';
echo 'Description: ' . $b->description . '<br/>';
```

# Freeing Resources

And closing the connection

```
// mysqli approach
$connection = mysqli_connect($host, $user, $pass, $database);
...
// release the memory used by the result set. This is necessary if
// you are going to run another query on this connection
mysqli_free_result($result);
...
// close the database connection
mysqli_close($connection);
```

```
// PDO approach
$pdo = new PDO($connString,$user,$pass);
...
// closes connection and frees the resources used by the PDO object
$pdo = null;
```

LISTING 11.25 Closing the connection

# Using Transactions

mysqli

```
mysqli_begin_transaction(mysqli $mysql, int $flags = 0, ?string $name = null): bool
```

```
$connection = mysqli_connect($host, $user, $pass, $database);  
...  
/* set autocommit to off. If autocommit is on, then mysql will  
commit (i.e., make the data change permanent) each command after  
it is executed */  
mysqli_autocommit($connection, FALSE);  
  
/* insert some values */  
$result1 = mysqli_query($connection,  
    "INSERT INTO Categories (CategoryName) VALUES ('Philosophy')");  
$result2 = mysqli_query($connection,  
    "INSERT INTO Categories (CategoryName) VALUES ('Art')");  
  
if ($result1 && $result2) {  
    /* commit transaction */  
    mysqli_commit($connection);  
}  
else {  
    /* rollback transaction */  
    mysqli_rollback($connection);  
}
```

# Using Transactions

## PDO

```
$pdo = new PDO($connString,$user,$pass);
// turn on exceptions so that exception is thrown if error occurs
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

...
try {
    // begin a transaction
    $pdo->beginTransaction();

    // a set of queries: if one fails, an exception will be thrown
    $pdo->query("INSERT INTO Categories (CategoryName) VALUES
        ('Philosophy')");
    $pdo->query("INSERT INTO Categories (CategoryName) VALUES
        ('Art')");

    // if we arrive here, it means that no exception was thrown
    // which means no query has failed, so we can commit the
    // transaction
    $pdo->commit();
} catch (Exception $e) {
    // we must rollback the transaction since an error occurred
    // with insert
    $pdo->rollback();
}
```

Section 7 of 7

# SAMPLE DATABASE TECHNIQUES

# Display a list of Links

One of the most common database tasks in PHP is to display a list of links (i.e., a series of `<li>` elements within a `<ul>`).

```
<ul>
    <li><a href="list.php?category=7">Business</a></li>
    <li><a href="list.php?category=2">Computer Science</a></li>
    <li><a href="list.php?category=3">Economics</a></li>
    <li><a href="list.php?category=9">Engineering</a></li>
    <li><a href="list.php?category=4">English</a></li>
    <li><a href="list.php?category=6">Mathematics</a></li>
    <li><a href="list.php?category=8">Statistics</a></li>
    <li><a href="list.php?category=5">Student Success</a></li>
</ul>
```

# Display a list of Links

The code would look something like the following:

```
$sql = "SELECT * FROM Categories ORDER BY CategoryName";
$result = $pdo->query($sql);
while ($row = $result->fetch()) {
    echo '<li>';
    echo '<a href="list.php?category=' . $row['ID'] . '">';
    echo $row['CategoryName'];
    echo '</a>';
    echo '</li>';
}
```

# Search and Results Page

Visual of search box, results page, and no results page

The figure consists of three vertically stacked screenshots of a web browser window. The top screenshot shows a search form with a placeholder 'Enter search string' and a 'Submit' button. The middle screenshot shows the same form with the word 'business' typed into the input field, and below it, a table of search results. The bottom screenshot shows the same search form with the placeholder 'Enter search string' and a 'Submit' button.

Display the user's search term in the text box

ISBN	Title	Year
0321838696	Business Statistics: A First Course	2014
0321836510	Statistics for Business: Decision Making and Analysis	2014
032182623X	Statistics for Business and Economics	2014
0132898357	Mathematics for Business	2014
0133011208	Business Math	2014
0133140423	Business Math Brief	2014
0132666790	Essentials of Entrepreneurship and Small Business Management	2014
013261930X	English for Careers: Business, Professional, and Technical	2014
0132971275	Intercultural Business Communication	2014
0133059049	Better Business	2014
0133063003	International Business	2014
0132971321	Business Communication Essentials	2014
0132846918	Digital Business Networks	2014
0133059510	Business Communication	2014
013610066X	Business Intelligence	2011

1 What is displayed when the page is first requested

2 Search results displayed in simple HTML table

To aid in debugging, we will use HTTP GET.

3 If there are no matches, won't display anything (later we can add error messages)

# Search and Results Page

In this example, we will assume that there is a text box with the name *txtSearch* in which the user enters a search string along with a *Submit* button.

The data that we will filter is the *Book* table; we will display any book records that contain the user-entered text in the *Title* field.

```
// add SQL wildcard characters to search term
$searchFor = '%' . $_GET['txtSearch'] . '%';
$sql = "SELECT * FROM Books WHERE Title Like ?";
$statement = $pdo->prepare($sql);
$statement->bindValue(1, $searchFor);
$statement->execute();
```

# Search and Results Page

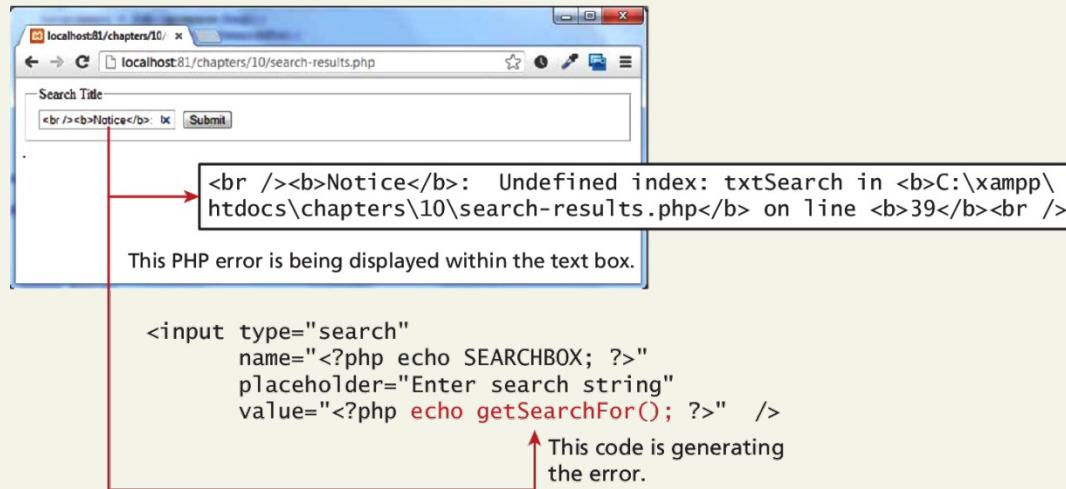
To redisplay the search term we will add code like:

```
<input  
    type="search"  
    name="txtSearch"  
    placeholder="Enter search string"  
    value="php echo $_GET['txtSearch']; ?&gt;" /&gt;</pre
```

To where we generate the form. Unfortunately...

# Search and Results Page

Problem to be solved



```
function getSearchFor()
{
    $value = "";
    if (isset($_GET[SEARCHBOX])) {
        $value = $_GET[SEARCHBOX];
    }
    return $value;
}
```

**LISTING 11.30** Solution to search results page problem

# Files in the Database

There are two ways of storing files in a database:

1. Storing file location in the database, and storing the file on the server's filesystem
2. Storing the file itself in the database in the form of a binary BLOB

# Files in the Database

As a file Location

Some page in the browser



```
<form enctype='multipart/form-data' method='post' action='upFile.php'>
  <input type='file' name='file1'></input>
  <input type='submit'></input>
</form>
```

1 User uploads file

C:\Users\ricardo\Pictures\Sample1.png

upFile.php

2 PHP script retrieves uploaded file from  
\$\_FILES array, gives it a unique file name,  
and then moves it to special location.



ID	UID	Path	ImageContent
..	...	...	...
280	35	/images/983412824.jpg	...

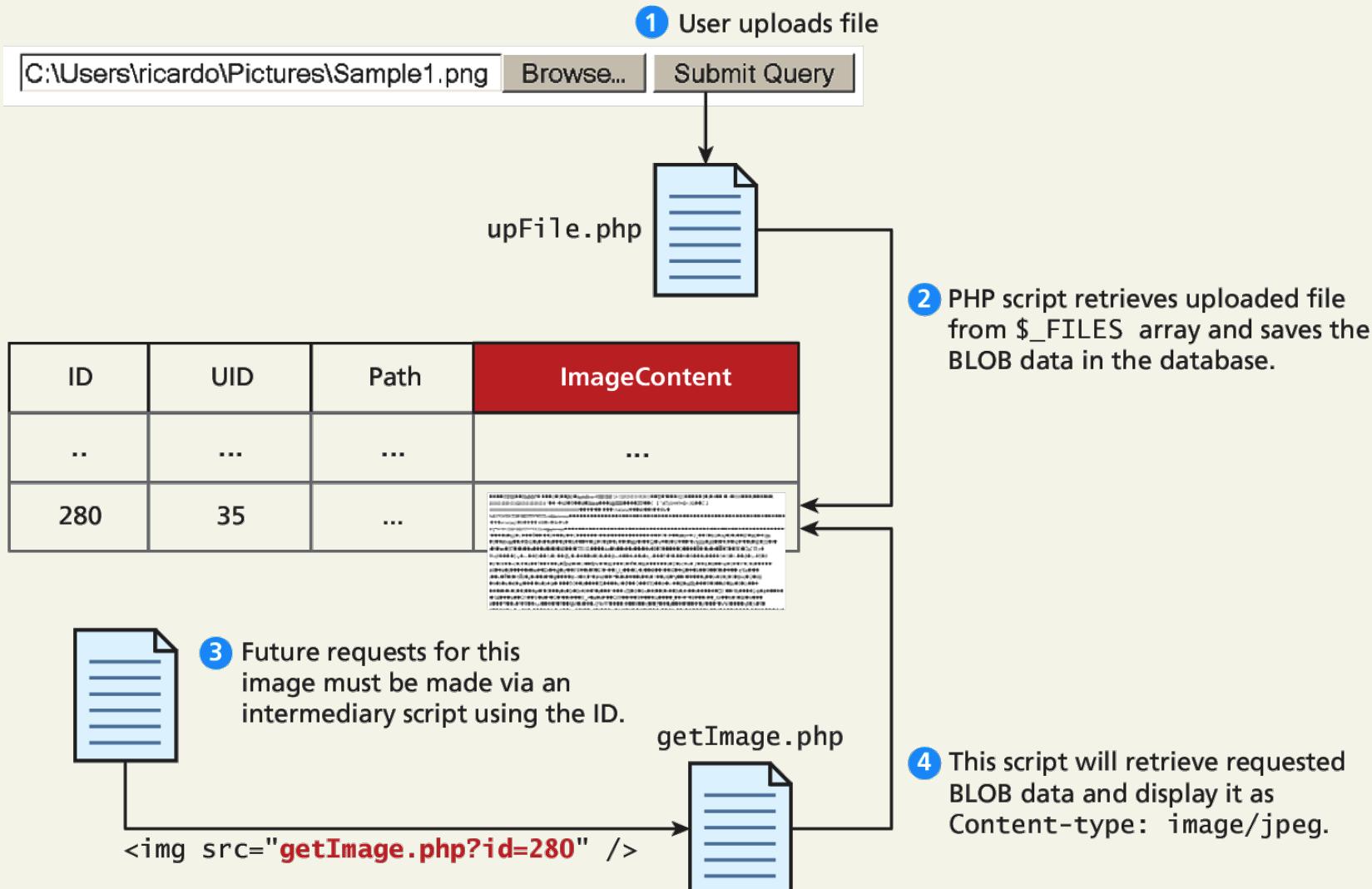
3 PHP script then saves  
this information in  
database table.



4 Future requests for this  
image can be made by any  
page by using the path of  
the file.

# Files in the Database

As a BLOB



# Files in the Database

Storing and retrieving BLOBs

```
$fileContent = file_get_contents("someImage.jpg");
$sql = "INSERT INTO TravelImage (ImageContent) VALUES(:data)";

$statement = $pdo->prepare($sql);
$statement->bindParam(':data', $fileContent, PDO::PARAM_LOB);
$statement->execute();
```

**LISTING 11.35** Code to save file contents in a BLOB field

```
// retrieve blob content from database
$sql = "SELECT * FROM TravelImage WHERE ImageID=:id";
$statement = $pdo->prepare($sql);
$statement->bindParam(':id', $_GET['id']);
$statement->execute();

$result = $statement->fetch(PDO::FETCH_ASSOC);
if ($result) {
    // Output the MIME header
    header("Content-type: image/jpeg");
    // Output the image
    echo ($result["ImageContent"]);
}
```

**LISTING 11.36** Code to fetch and echo BLOB image

# Files in the Database

Storing and retrieving BLOBs

Listing 11.36 can then be integrated into HTML image tags by pointing the src attribute to our script

```

```

Would now reference a dynamic PHP script like:

```

```

# Files in the Database

HTTP headers matter

The same file output with correct and incorrect headers is interpreted differently by the browser

