Working with Databases

Chapter 11

Randy Connolly and Ricardo Hoar

Section 1 of 7

DATABASES AND WEB DEVELOPMENT

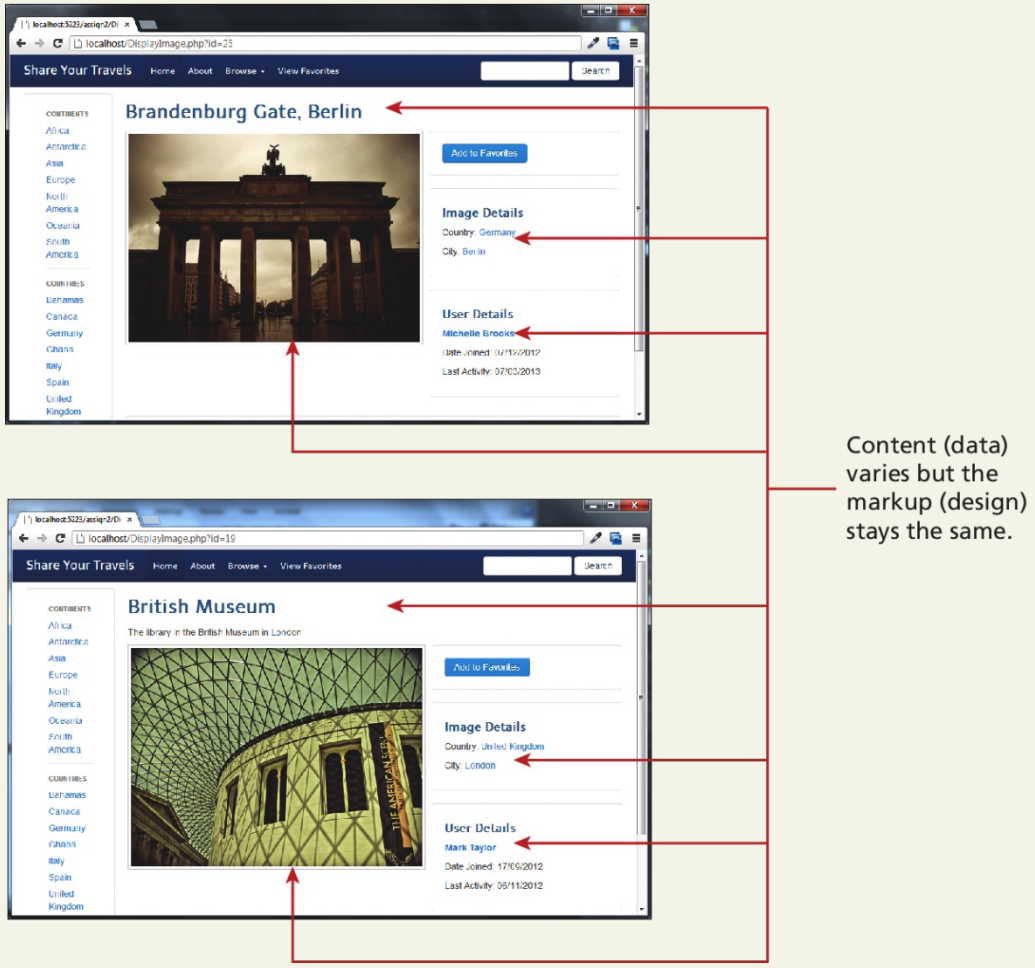
The Role of Databases

In Web Development

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

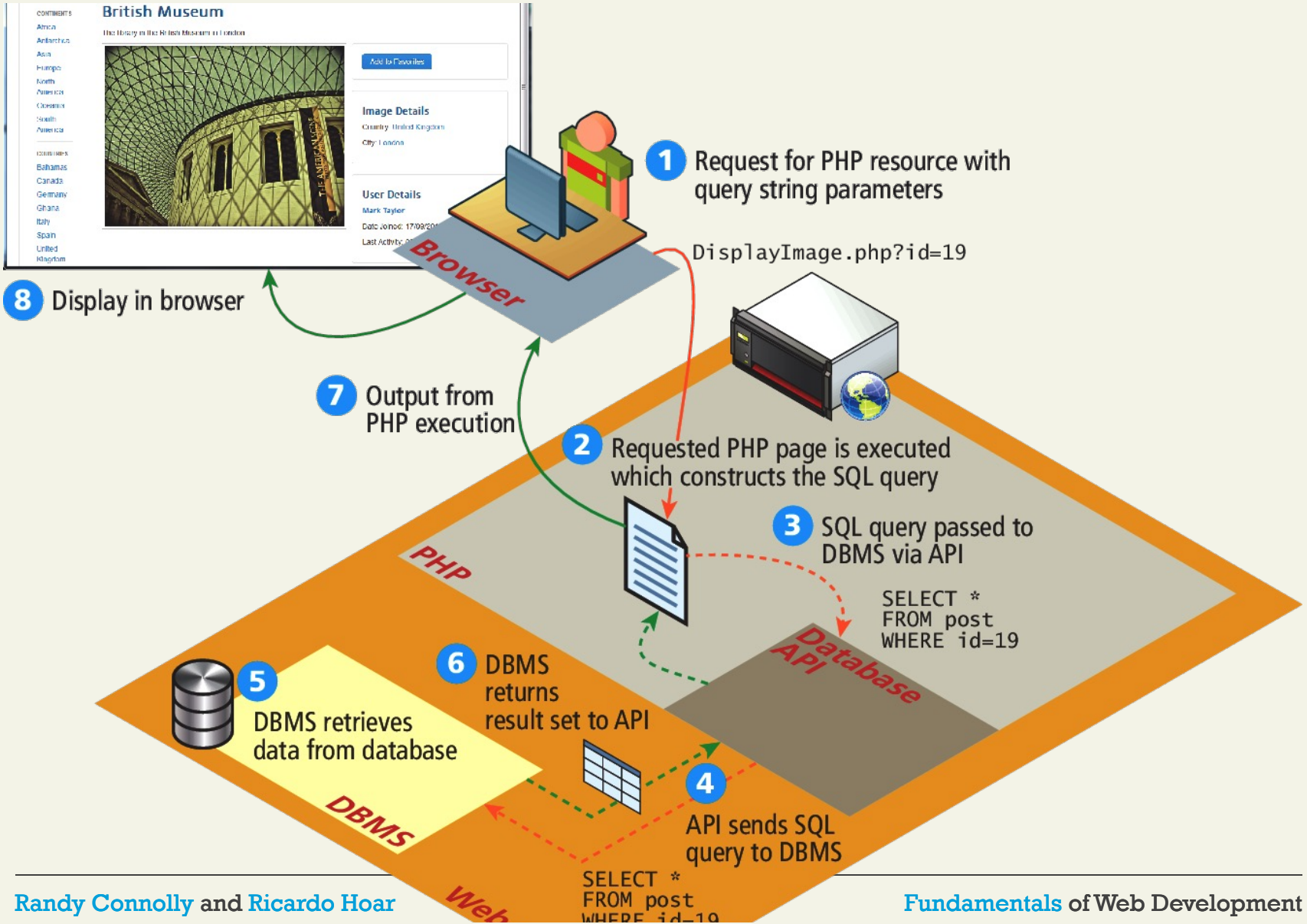
> *one should separate that which varies from that which stays the same*

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.

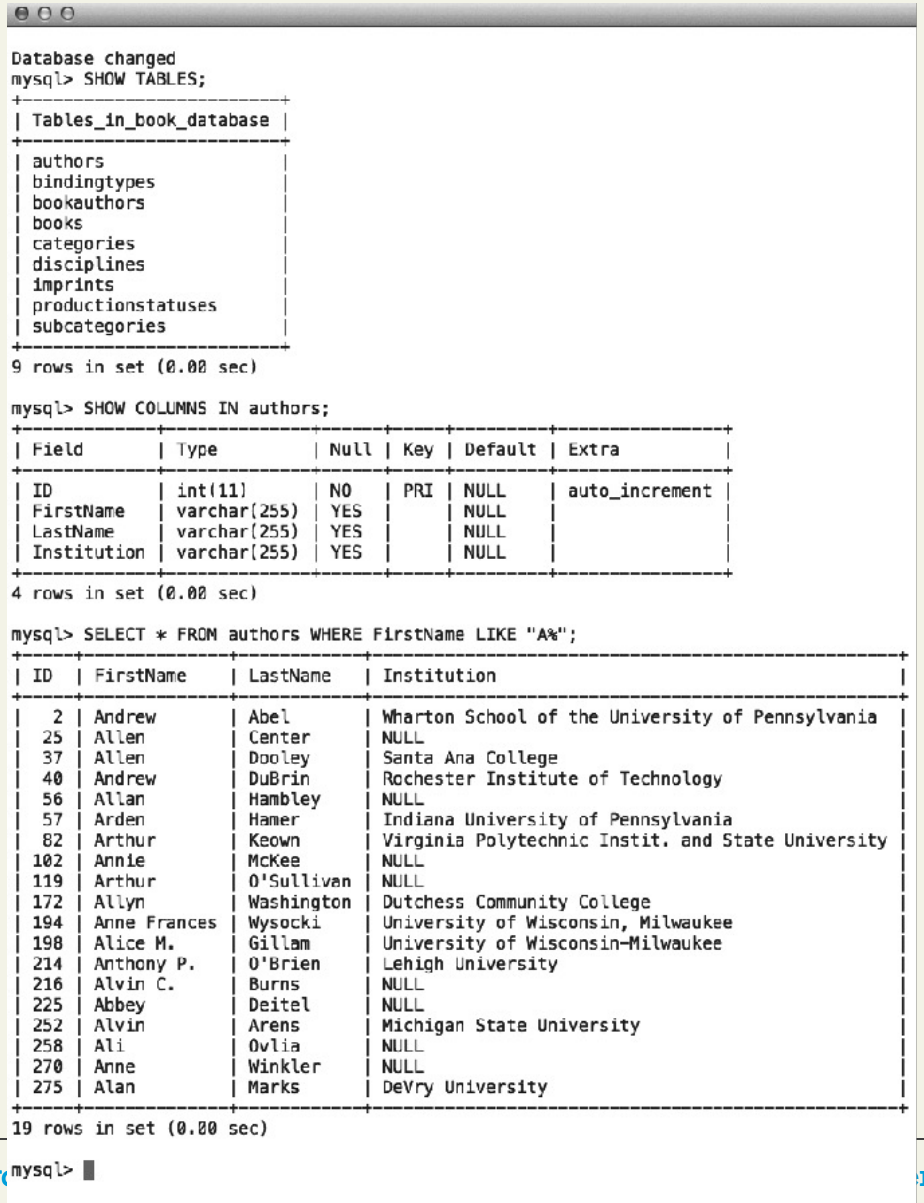
To launch an interactive MySQL command-line session, you must specify the host, username, and database name to connect to as shown below:

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         | Ovlia      | NULL                                          |
| 270 | Anne        | Winkler    | NULL                                          |
| 275 | Alan        | Marks      | DeVry University                              |
+-----+-------------+------------+-----------------------------------------------+
19 rows in set (0.00 sec)

mysql>
```

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

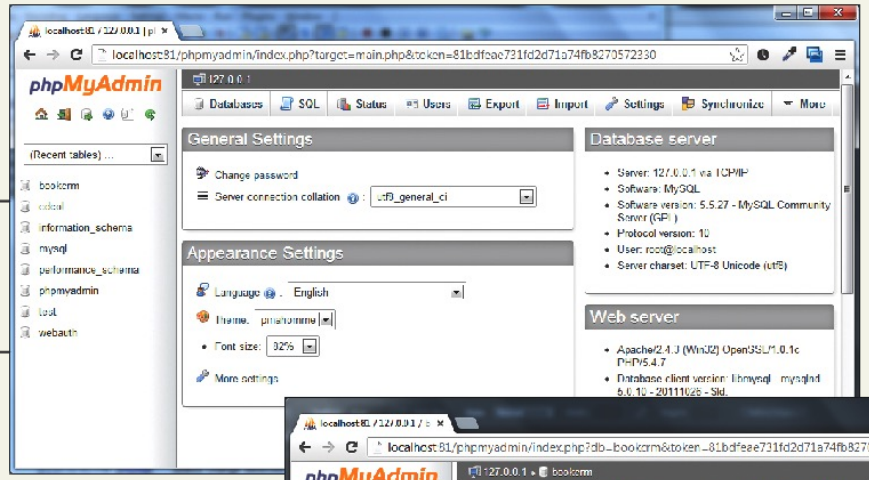
Although GUI tools allow this as well, the CLI can be integrated into automated scripts to aid with mirroring, backup, and recovery.

phpMyAdmin

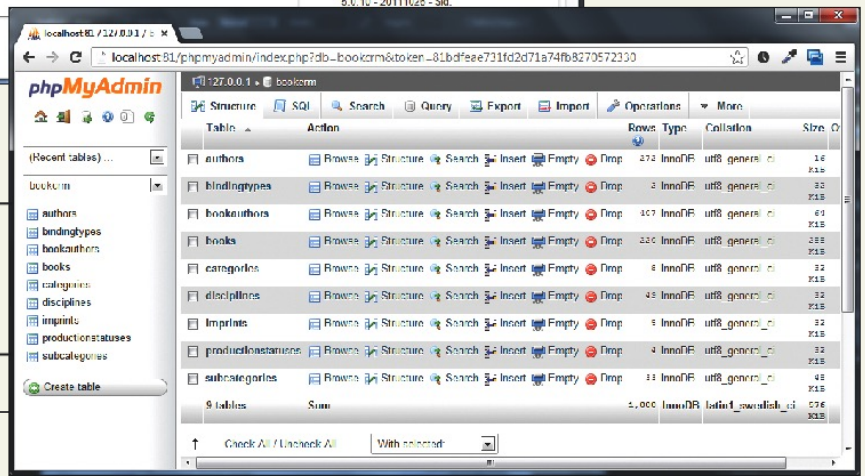
Will even generate PHP code

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.



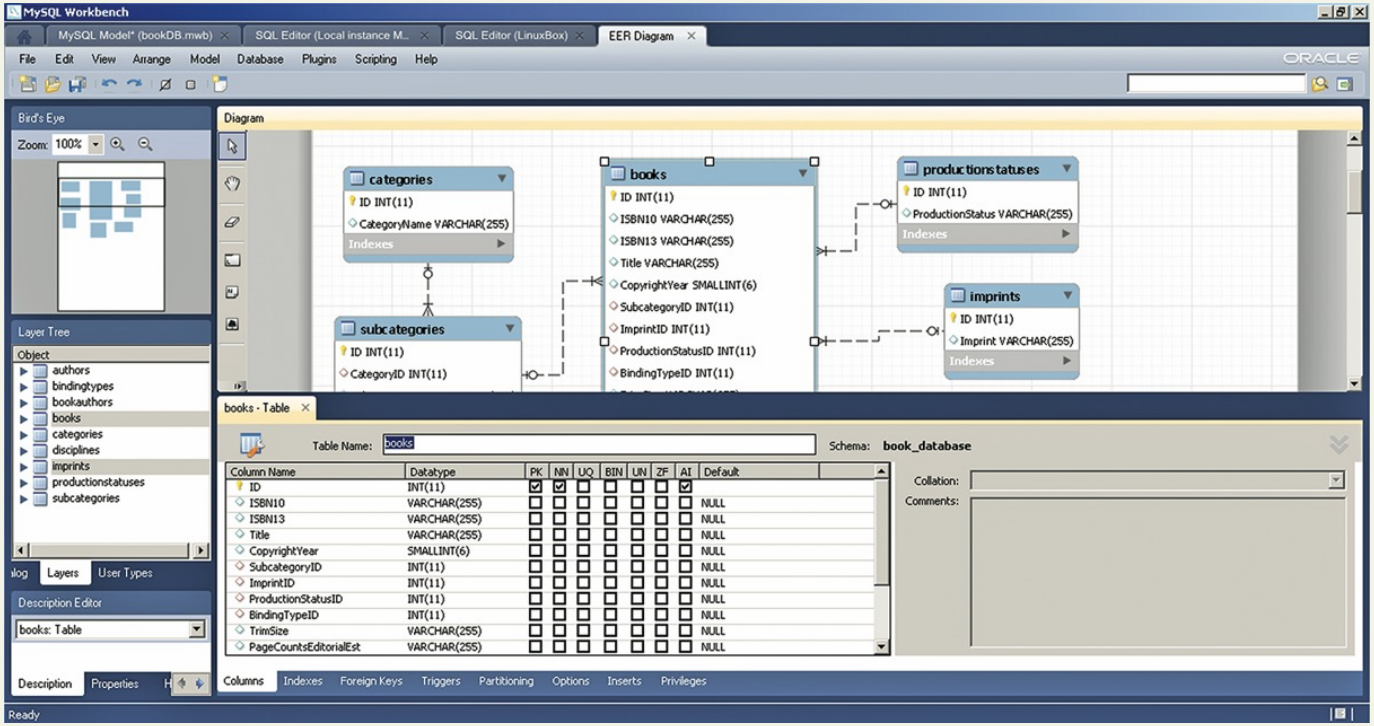
phpMyAdmin allows you to view and manipulate any table in a database.



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() );
}

?>
```

1
2
3
4
5

Database Connection Algorithm

An illustration through example

```php
// 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)

```php
// 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

```php
$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

```php
$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)

```php
$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>";
}
```

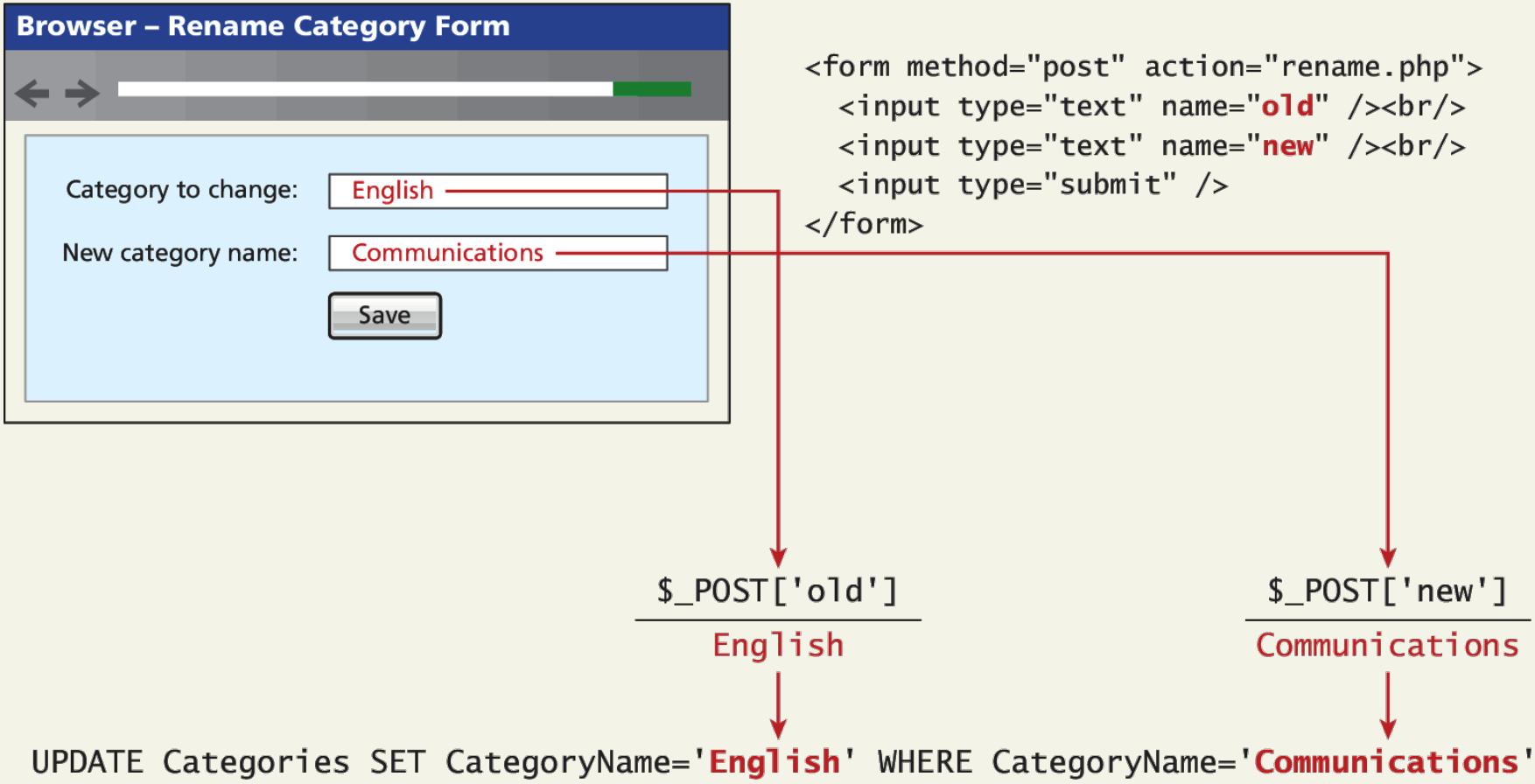
LISTING 11.13 Executing a query that doesn't return data (mysqli)

```
$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



Integrating User Data

Not everyone is nice.

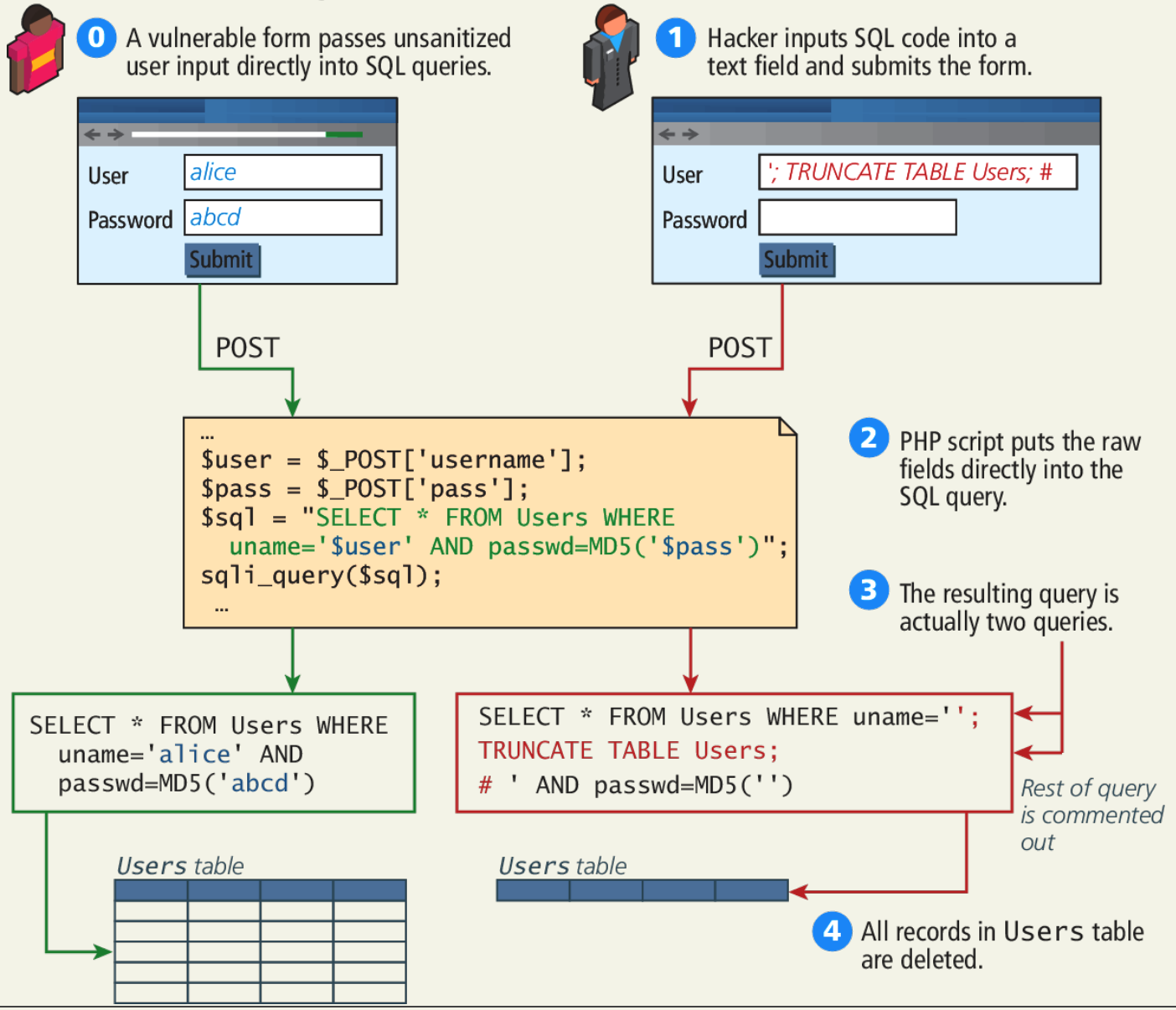
```
$from = $_POST['old'];
$to = $_POST['new'];
$sql = "UPDATE Categories SET CategoryName='$to' WHERE
        CategoryName='$from'";

$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



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

```php
// 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(4, $_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

```php
// 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);
```

```php
// 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);
}
```

LISTING 11.26 Using transactions (mysqi extension)

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();
}
```

LISTING 11.27 Using transactions (PDO)

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>
```

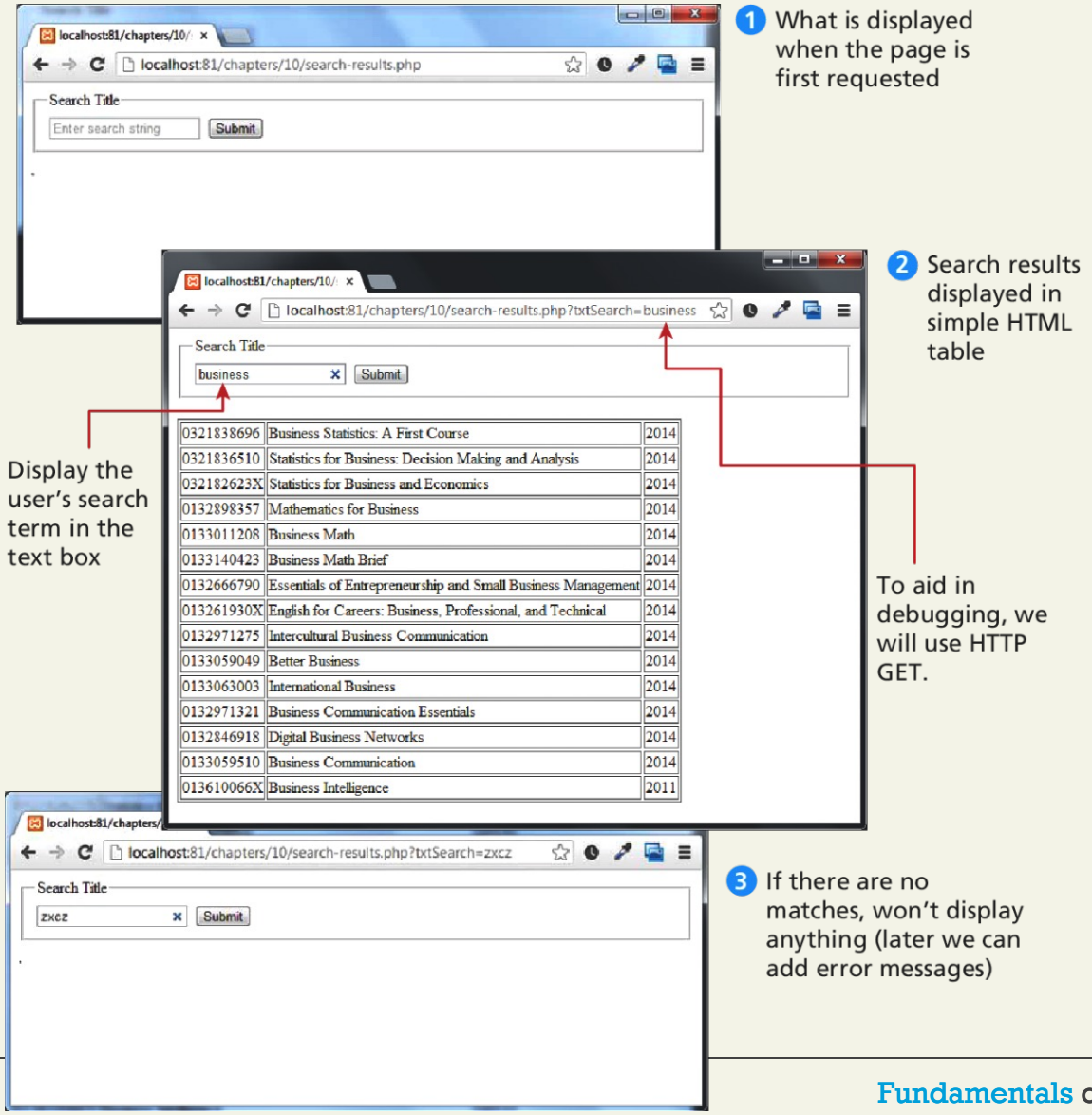
Display a list of Links

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



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

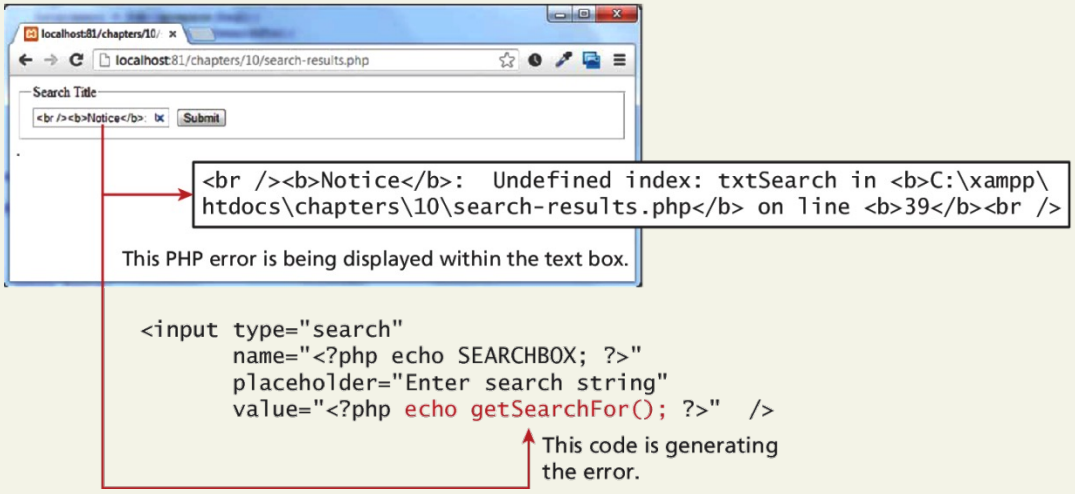
To redisplay the search term we will add code like:

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

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

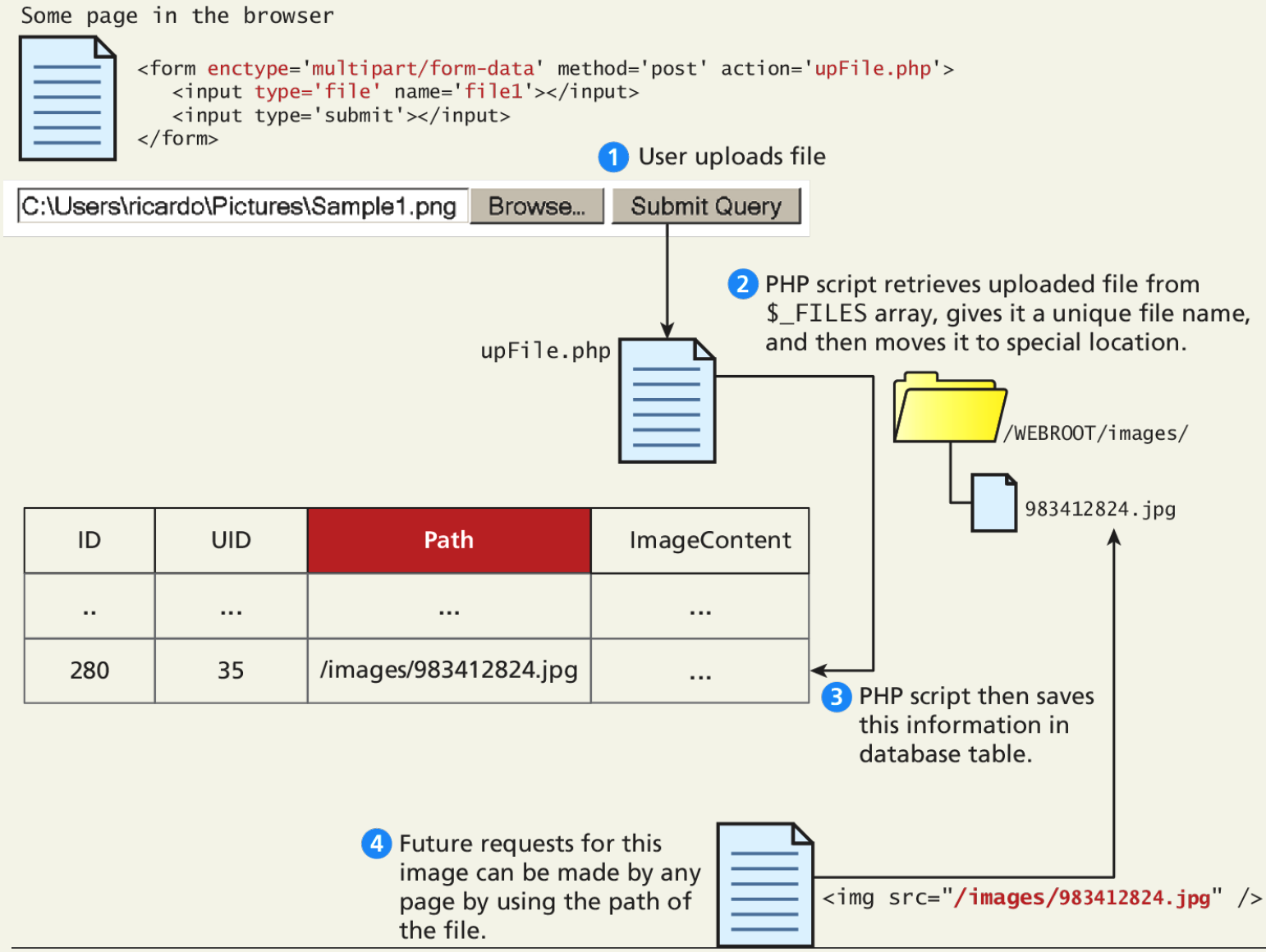
Files in the Database

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

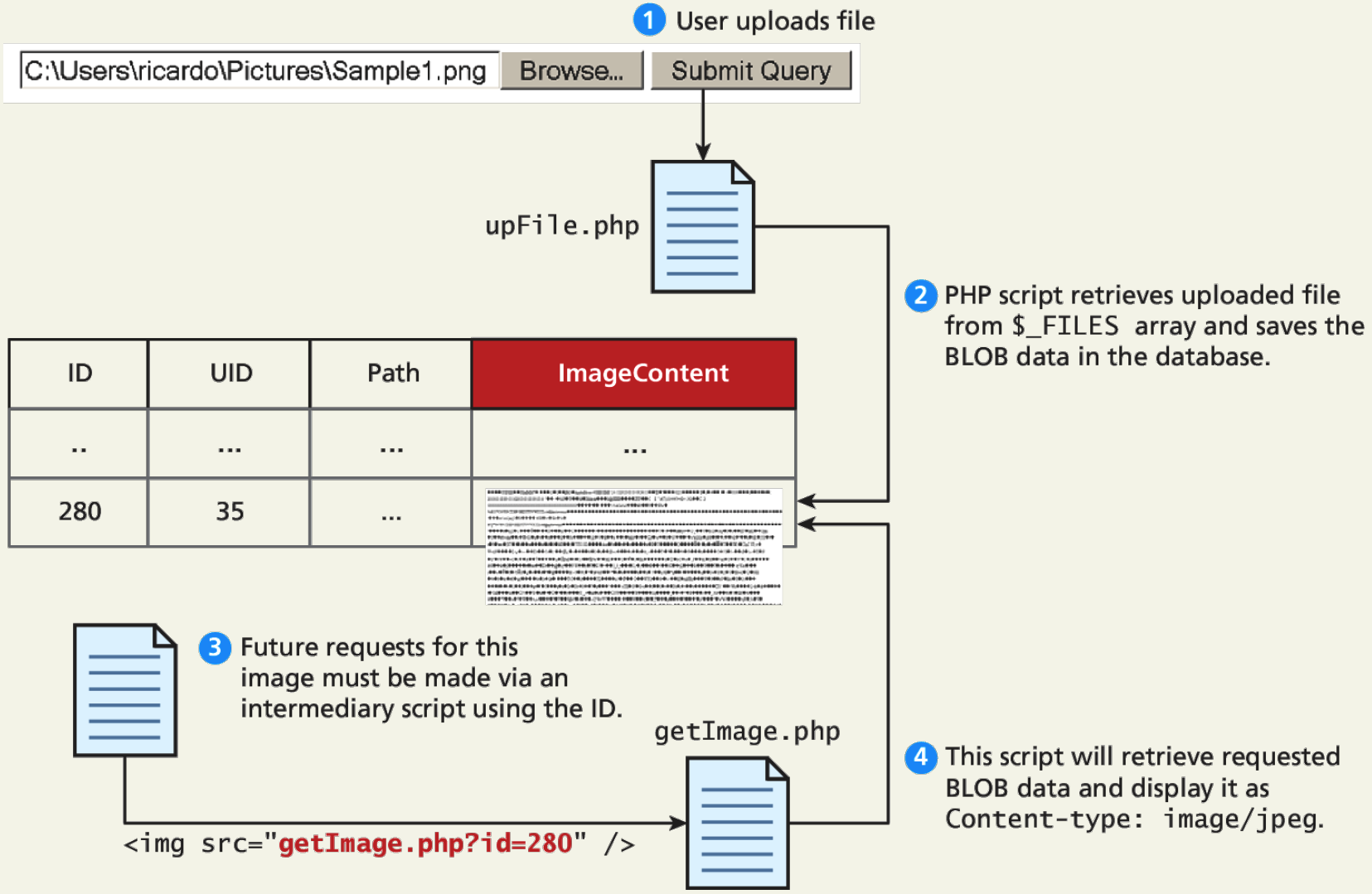
Files in the Database

As a file Location



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