LAB 14a

Working With Databases

What You Will Learn

- How to install and manage a MySQL database
- Hot to use SQL queries in your PHP code
- How to integrate user inputs into SQL queries

Approximate Time

The exercises in this lab should take approximately 90 minutes to complete.

Fundamentals of Web Development, 3rd Ed

Randy Connolly and Ricardo Hoar

Textbook by Pearson
http://www.funwebdev.com

XAMPP Version, Date Last Revised: November 22, 2020

Introducing MySQL

Preparing Directories

1 Like the previous PHP labs, this lab requires both a functioning webserver and an installation of MySQL. XAMPP currently uses MariaDB, which is a compatible fork of the open-source MySQL project that is independent of Oracle (which acquired the rights to the MySQL name when it purchased Sun in 2010).

This lab assumes you have access to MySQL/MariaDB via XAMPP.

2 Create a folder named `lab14a` to your PHP development location on your development machine (if using XAMPP, this might be `c:/xampp/htdocs`). Copy the contents of the provided lab folder (by your instructor or cloned from the GitHub repo for this lab) into the `lab14a` folder.

If using XAMPP on Windows, this will likely be `c:/xampp/htdocs`.

If using XAMPP on Mac, the `htdocs` folder is on its own mounted volume. This will require you to start XAMPP, enable localhost:8080 (via Network tab in XAMPP), mount the volume (via the Volume tab in XAMPP), then click the Explore button, which will display the Finder window for this volume. Copy the `lab14a` folder to the `htdocs` folder.

If you have finished lab 12a using XAMPPP, you will already have MySQL/MariaDB installed locally on your development machine.

Exercise 14a.1 — Running MySQL in the Shell

1 To perform the next several exercises, you will be using the MariaDB shell. This shell is available on other platforms as well.

Run the XAMPP control panel and make sure both Apache and MySQL modules are started.

2 If MySQL is running, you can run SQL queries in MySQL from within the XAMPP shell. Click the Shell button in the XAMPP Control Panel. In the shell, type the following command:

```
mysql -h localhost -u root
```

You will notice that the Shell prompt has changed to MariaDB>. You can now type in different MySQL commands.

3 Type in the following command:

```
show databases;
```

This will display the already installed databases.

4 Type in the following command:

```
use mysql;
```

This tells MySQL to use the database named mysql for subsequent query commands. This particular database was created by the install script and is used by MySQL itself. You will likely never manipulate this database. We are doing so here purely for illustration purposes.

5 Type in the following command:

```
show tables;
```

This will display the tables within the current database.

6 Type in the following commands:

```
create database travel;
use travel;
```

This will create a new database named travel and make it the current database.

7 Type in the following command (Windows):

```
source c:/xampp/htdocs/lab14a/databases/travel-3rd.sql;
```

Or, type in the following command (Mac):

```
source /opt/lampp/htdocs/lab14a/databases/travel-3rd.sql;
```

This will run the SQL statements in the specified file (if you examine this file you will notice it begins with the same two commands you just entered in step 6). If your XAMPP is installed in another location, you will have to change the path above.

8 Type in the following command:

```
show tables;
```

This should display several table names.

9 Type in the following command:

```
select * from users;
```

This should display the content of the specified table. Because MySql maps table names to file system files and because Linux-based operating systems are case sensitive, MySQL table names on Linux environments will also be case sensitive.

10 Type in the following command:

```
exit
```

This will exit the MySQL shell and return to XAMPP shell.

11 Type in the following command:

```
exit
```

This will exit the XAMPP shell.

Although you will be able to manipulate the database from your PHP code, there are some maintenance operations (such as creating tables, importing data, etc) that typically do not warrant writing custom PHP code. For these types of tasks, you will use either the MySQL command line (as shown in previous exercise) or by using some type of MySQL management tool, such as the popular web-based front-end `phpMyAdmin`.

EXERCISE 14a.2 — PHPMyAdmin

1 Start `phpAdmin`.

In XAMPP Control Panel in Windows, you do this by clicking on the Admin button for MySQL, or by entering the following URL into a browser tab: `localhost/phpmyadmin/`

In XAMPP Control Panel in Mac, you do this by entering the following URL into a browser tab: `localhost:8080/phpmyadmin/`.

Eventually you should see the phpMyAdmin panel as shown in Figure 14a.1

Because MySQL has a blank password by default and phpMyAdmin uses the same credentials as MySQL, phpMyAdmin has a blank password by default as well, which makes setup significantly easier for users. Since developers generally don't (and often shouldn't) put sensitive/important data in a development environment like XAMPP, having no password on phpMyAdmin is rarely an issue.

3 The left side of phpMyAdmin displays the existing databases in MySQL. A default installation of MySQL contains a variety of system tables used by MySQL (which depending on your installation may or may not be visible here).

4 Click on the `travel` database.

This will display the tables in the travel database.

5 Click the browse link for the `Countries` table.

This will display the first set of records in this table with edit/copy/delete links for each record.

6 Click on the Structure tab.

This will display the definitions for the fields in the current table, with links for modifying the structure.

7 Click on the SQL tab. In the SQL window type in the following query then click GO button:

```
select * from countries where Continent='NA'
```

This should display the matching records

8 Click on the phpMyAdmin logo, then click on Database tab. You can add new databases here.

9 Click on Import tab. Click on the Choose File button and navigate to the `art-3rd.sql` file.

This should create a database and populate it. You can also import data by running data definition SQL statements in the SQL tab.

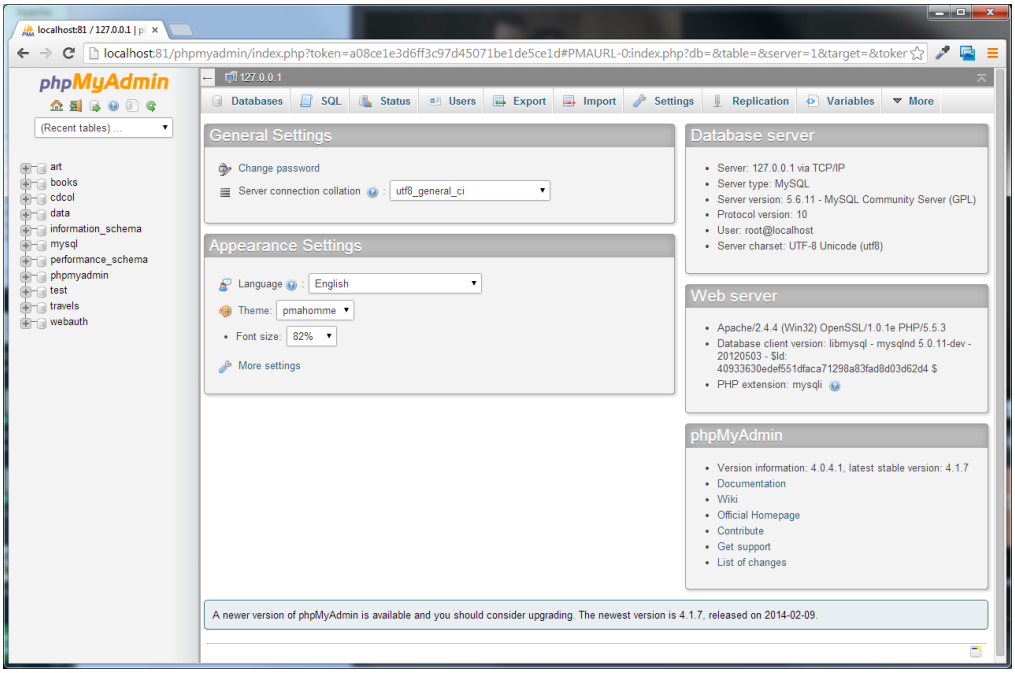

Figure 14a.1 – phpMyAdmin

TEST YOUR KNOWLEDGE #1

1 Examine the `editable.sql` file in the `database` folder.

2 Using either the MySQL/MariaDB shell (see step 7 of Exercise 14a.1) or PHPMyAdmin (see step 9 of Exercise 14a.2), perform the following action.

Populate the art database with a new table using the `editable.sql` file.

SQL

MySQL, like other relational databases, uses Structured Query Language or, as it is more commonly called, SQL (pronounced sequel) as the mechanism for storing and manipulating data. Later in the lab you will use PHP to execute SQL statements. However you will often find it helpful to run SQL statements directly in MySQL, especially when debugging.

The following exercises assume that your databases have been created and populated (i.e., you have completed Test Your Knowledge #1).

EXERCISE 14a.3 — QUERYING A DATABASE (OPTIONAL)

1 Using the MySQL/MariaDB command shell, enter the following commands.

```
use art;
select * from Artists;
```

2 Using the MySQL command shell, enter the following command.

```
select paintingid, title, yearofwork from Paintings
where yearofwork < 1600;
```

In MySQL, database names correspond to operating system directories while tables correspond to one or more operating system files. Because of this correspondence, table and database names ARE case sensitive on non-Windows operating systems (though field names are not case sensitive).

3 Modify the query as follows and test.

```
select paintingid, title, yearofwork from Paintings
where yearofwork < 1600 order by yearofwork;
```

4 Modify the query as follows and test.

```
SELECT Artists.ArtistID,Title,YearOfWork,LastName FROM Artists
INNER JOIN Paintings ON Artists.ArtistID = Paintings.ArtistID;
```

This query contains a join since it queries information from two tables. Notice that you must preface `ArtistId` with the table name since both joined tables contain a field called `ArtistId`.

Why are some of the SQL words uppercase now? No reason … just wanted to illustrate that upper or lower case can be used with the SQL keywords.

5 Modify the query as follows and test.

```
SELECT Nationality, Count(ArtistID) AS NumArtists
FROM Artists
GROUP BY Nationality;
```

This query contains an aggregate function as well as a grouping command.

EXERCISE 14a.4 — MODIFYING RECORDS (OPTIONAL)

1 Using the MySQL command shell, enter the following commands.

```
insert into Artists (firstname, lastname, nationality, yearofbirth,
yearofdeath) values ('Palma','Vecchio','Italy',1480,1528);
```

You should see message about one row being affected (i.e., one record has been inserted).

2 Examine the just-inserted record by running the following query.

```
select * from Artists where lastname = 'Vecchio';
```

Notice that `ArtistId` value has been auto-generated by MySQL. This has happened because this key field has the auto-increment property set to true.

3 Run the following new query:

```
update Artists
set details='Palmo Vecchio was a painter of the Venetian school'
where lastname = 'Vecchio';
```

4 Verify the record was updated (i.e, by running the query from step 2).

5 Run the following new query:

```
delete from Artists
where lastname = 'Vecchio';
```

6 Verify the delete worked by running the query from step 2).

One of the key benefits of databases is that the data they store can be accessed by queries. This allows us to search a database for a particular pattern and have a resulting set of matching elements returned quickly. In large sets of data, searching for a particular record can take a long time. To speed retrieval times, a special data structure called an index is used. A database table can contain one or more indexes.

EXERCISE 14a.5 — BUILD AN INDEX (OPTIONAL)

1 Using the MySQL command shell, enter the following command.

```
show indexes in Paintings;
```

You will notice that there are several indexes already defined. Where did they come from?

2 Examine the file `art-3rd.sql`. You will notice that indexes were pre-defined in the `Create Table` statements.

3 Using the MySQL command shell, enter the following command.

```
create index idxYears on Paintings (YearOfWork);
```

If you are using PHPMyAdmin, you will not need to create any separate MySQL users. If you are using a different platform (or you just want to try it), you will need to do the next exercise.

EXERCISE 14a.6 — CREATING USERS IN PHPADMIN

1. In phpMyAdmin, click on the `art` database, and then click on the **Privileges** tab.

   This will display the users who currently have access to this database. Notice the root user. This root user has special privileges within MySQL: indeed, you very well may have logged into phpMyAdmin using the root account. For development-only environments, using the root user will likely be okay. Nonetheless, we are going to create a new user which you will use for subsequent examples in this lab.

2. Click the **Add user** account link.

   This will display the Add user page (see Figure 14a.2).

3. In the Add user page, enter the following into the Login information section:

   ```
   User name (use text filed): testuser
   Host (Local):
   Password (use text filed): mypassword
   Re-Type: mypassword
   ```

   You are of course welcome to enter a different user name and password. If you do, you will need to substitute future references to testuser and password. Also, depending on the environment you are using, you may need to enter something different in the Host field (perhaps 'localhost' or '127.0.0.1')

4. In the **Database for user account** section, ensure the **Grant all privileges on database "art"** checkbox is checked.

5. In the **Global privileges** section, check the five Data privileges (select, insert, update, delete, and file).

6. Click the Go button.

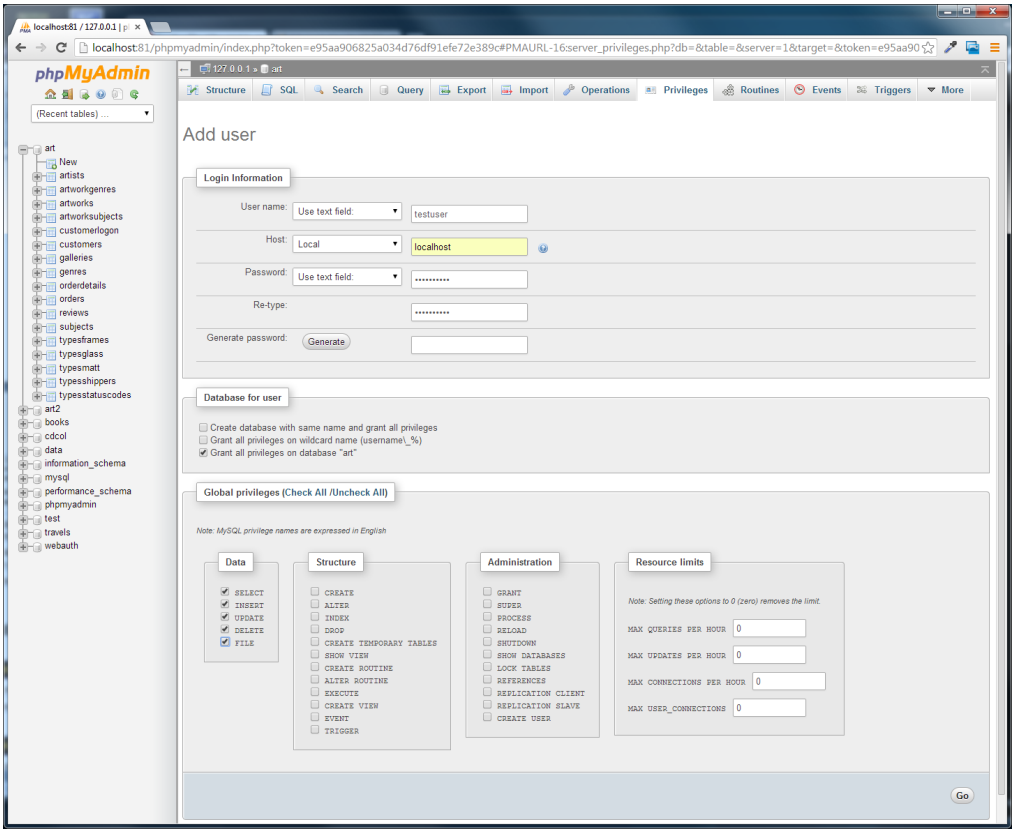

Figure 14a.2 – Creating a user

Accessing Databases in PHP

As covered in the book, there are different database APIs depending on the database you are using.

EXERCISE 14a.7 — MySQL via PHP

1 Open `config.inc.php` and modify the file as shown below.

```
<?php
define('DBHOST', 'localhost');
define('DBNAME', 'art');
define('DBUSER', 'root');
define('DBPASS', '');
define('DBCONNSTRING',"mysql:host=" . DBHOST . ";dbname=" . DBNAME .
          ";charset=utf8mb4;");
?>
```

This is the setup for XAMPP. If you are using a different environment, you will likely need to change the DBUSER, DBPASS, and DBHOST values.

The `charset` specified here in our connection string is optional. It helps ensure that the database API handles our UTF8 encoded data (i.e., foreign character sets).

2 Open `lab14a-ex07-pdo.php` and modify as follows:

```
<?php require_once('config.inc.php'); ?>
<!DOCTYPE html>
<html>
<body>
<h1>Database Tester (PDO)</h1>
<?php
try {
   $pdo = new PDO(DBCONNSTRING,DBUSER,DBPASS);
   $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
   $sql = "select * from Artists order by LastName";
   $result = $pdo->query($sql);
   // loop through the data
   while ($row = $result->fetch()) {
      echo $row['ArtistID'] . " - " . $row['LastName'] . "<br/>";
   }
   $pdo = null;
}
catch (PDOException $e) {
   die( $e->getMessage() );
}
?>
</body>
</html>
```

This uses the object-oriented PDO API for accessing databases.

3 Save and test.

This should display list of artists.

4 Open `lab14a-ex07-mysqli.php` and modify as follows:

```
<?php require_once('config.inc.php'); ?>
<!DOCTYPE html>
<html>
<body>
<h1>Database Tester (mysqli)</h1>
Genre:
<select>
<?php

$connection = mysqli_connect(DBHOST, DBUSER, DBPASS, DBNAME);
if ( mysqli_connect_errno() ) {
   die( mysqli_connect_error() );
}

$sql = "select * from Genres order by GenreName";

if ($result = mysqli_query($connection, $sql)) {
   while($row = mysqli_fetch_assoc($result))
    {
         echo '<option value="' . $row['GenreID'] . '">';
         echo $row['GenreName'];
         echo "</option>";
    }
    // release the memory used by the result set
    mysqli_free_result($result);
}

// close the database connection
mysqli_close($connection);
?>
</select>
</body>
</html>
```

This uses the procedural mysqli API for accessing databases. Older, legacy PHP code tends to use the mysqli API. For all future lab exercises, you will be using the PDO approach.

5 Go back to `lab14a-ex07-pdo.php` and modify the `while` loop as follows and test.

```
while ($row = $result->fetch()) {
   echo $row[0] . " - " . $row['LastName'] . "<br/>";
}
```

The array returned from fetch() has both an index key and a fieldname key.

6 Modify the loop as follows and test.

```
while ($row = $result->fetch(PDO::FETCH_ASSOC)) {
   echo $row['ArtistID'] . " - " . $row['LastName'] . "<br/>";
}
```

Now the array returned by fetch is indexed only by field name.

7 Comment out the `while` loop. Replace with the following foreach loop and test.

```
foreach ($result as $row) {
   echo $row[0] . " - " . $row['LastName'] . "<br/>";

}
```

With a foreach, the fetch() invocation happens implicitly within the loop. Notice that when using a foreach loop, the returned array has both an index key and a fieldname key.

8 Add a second identical loop after the first loop as follows and test.

```
$result = $pdo->query($sql);
foreach ($result as $row) {
   echo $row[0] . " - " . $row['LastName'] . "<br/>";
}
echo '<h2>-------- Second Loop --------</h2>';
foreach ($result as $row) {
   echo $row[0] . " - " . $row['LastName'] . "<br/>";
}
```

The second loop will not do anything because the resultset has come to the end of the returned data after the first loop.

9 Edit the code as follows and test.

```
$result = $pdo->query($sql);
$data = $result->fetchAll(PDO::FETCH_ASSOC);
foreach ($data as $row) {
   echo $row['ArtistID'] . " - " . $row['ArtistID'] . "<br/>";
}
echo '<h2>-------- Second Loop --------</h2>';
foreach ($data as $row) {
   echo $row['ArtistID'] . " - " . $row['ArtistID'] . "<br/>";
}
```

Now both loops work. The fetchAll method retrieves all the data at once from the table into memory. For large datasets, fetchAll should be avoided due to the amount of server memory utilized for storing all the data in a single array.

EXERCISE 14a.8 — SQLITE VIA PHP

1. Run `phpinfo.php`. Search the resulting page of information for "PDO drivers". You should see both mysql and sqlite. If your installation of PHP doesn't have SQLite enabled, you will need to either do some extra configuration steps (not covered) or skip this exercise.

2. Open `config.inc.php` and modify the file as shown below.

```
define('DBCONNSTRING','sqlite:./databases/art.db');
//define('DBCONNSTRING',"mysql:host=" …
```

Here we are commenting out the connection string to MySQL and replacing it with a connection string for the SQLlite database. Notice that SQLite only requires a file.

3. Re-request `lab14a-ex07-pdo.php`.

This should work now using the SQLite database instead of the MySQL database. This demonstrates the advantage of using PDO: one only needs to change the connection information (and possibly the SQL statements) to switch to a different database system.

4. Comment out the SQLite connection string and uncomment the MySQL connection string.

TEST YOUR KNOWLEDGE #2

1. You have been provided with the markup for the next exercise in the file `lab14a-test02.php`. You will be filling the select list with a list of sorted museums from the Galleries table.

2. At the top of the page, include or require the `config.inc.php` file.

3. Also at the top of the page, connect to the database and use `fetchAll()` to retrieve all the records from the `Galleries` table sorted by `GalleryName`.

4. Add a loop to output the rest of the `<option>` elements to the museum `<select>` list. Set the value attribute to the `GalleryID` field and the label of the option of the `GalleryName` field.

The result should look similar to that shown in Figure 14a.3

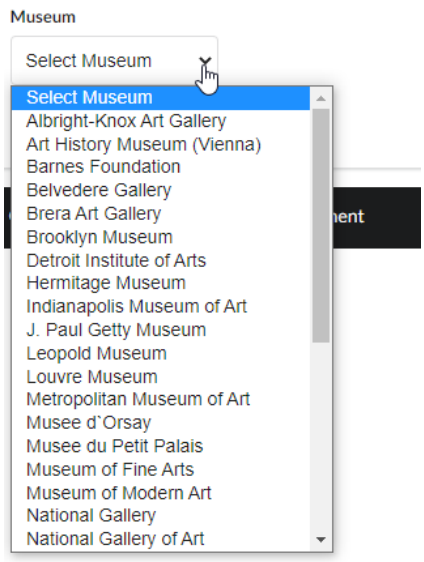

Figure 14a.3 – Completed Test Your Knowledge #2

EXERCISE 14a.9 — INTEGRATING USER INPUTS

1 Open and examine `lab14a-ex09.php`.

2 Create the following function and add to the top of the page:

```
function findPaintings($search) {
   try {
      $pdo = new PDO(DBCONNSTRING,DBUSER,DBPASS);
      $pdo->setAttribute(PDO::ATTR_ERRMODE,
                         PDO::ERRMODE_EXCEPTION);
      $sql = "SELECT Title,YearOfWork FROM Paintings";
      $sql .= " WHERE Title LIKE '%$search%'";
      $sql .= " ORDER BY YearOfWork";

      $result = $pdo->query($sql);
      $paintings = $result->fetchAll(PDO::FETCH_ASSOC);
      $pdo = null;
      return $paintings;
   }
   catch (PDOException $e) {
      die( $e->getMessage() );
   }
}
```

3 Add the following code to the top of the page:

```
if ( isset($_POST['search']) ) {
   $paintings = findPaintings($_POST['search']);
}
```

We only want to perform the search after the user submits the form.

4 In the section for the search results, add the following code:

```
<section class="twelve wide column">
<?php
   /* add your PHP code here */
   if ( isset($_POST['search']) ) {
      if ( count($paintings) > 0 ) {
         outputPaintings($paintings);
      } else {
         echo "no paintings found with search term = " .
                  $_POST['search'];
      }
   } else {
      echo "Enter a search term and press Filter";
   }
?>
</section>
```

You only want to display the paintings after the user submits the form. You also have to handle the possibility that the user hasn't yet submitted the form, or that the SQL hasn't returned any matches for the search string.

5 Create the following function and add to the top of the page:

```
function outputPaintings($paintings) {
   foreach ($paintings as $row) {
      echo $row['Title'] . " (" . $row['YearOfWork'] . ")<br/>";
    }
}
```

6 Test. Try search string of "child" or "self" or "morning".

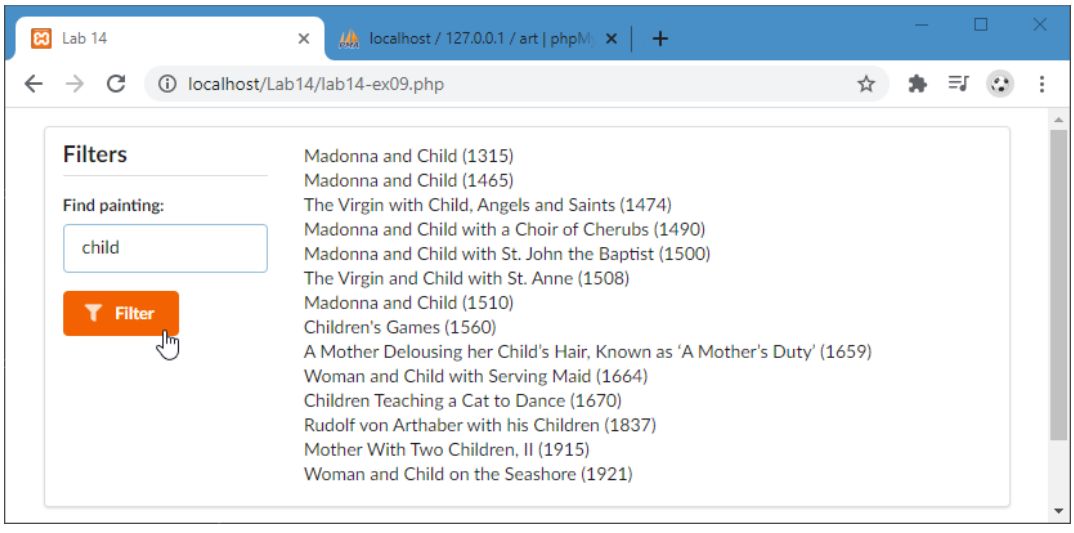

Figure 14a.4 – Completed Exercise 14a.9

EXERCISE 14a.10 — PREPARED STATEMENTS

1 This exercise modifies the previous exercise, `lab14a-ex09.php`.

2 Modify the `findPaintings` function as follows:

```
function findPaintings($search) {
   try {
      $pdo = new PDO(DBCONNSTRING,DBUSER,DBPASS);
      $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
      $sql = "SELECT Title,YearOfWork FROM Paintings";
      $sql .= " WHERE Title LIKE ?";
      $sql .= " ORDER BY YearOfWork";

      $statement = $pdo->prepare($sql);
      $statement->bindValue(1, '%' . $search . '%');
      $statement->execute();

      $paintings = $statement->fetchAll(PDO::FETCH_ASSOC);
      $pdo = null;
      return $paintings;
   }
   catch (PDOException $e) {
      die( $e->getMessage() );
   }
}
```

This is a better approach to handling user input since it protects against SQL Injection attacks.

3 Test.

When running multiple queries within a single page, we should ideally only use a single connection. This next exercise illustrates how one might do this.

EXERCISE 14a.11 — MAKING MULTIPLE QUERIES

1 Open and examine `lab14a-ex11.php`.

This page already contains functions for outputting paintings and galleries.

2 Add the top of the page, add the following.

```
try {
   $pdo = new PDO(DBCONNSTRING,DBUSER,DBPASS);
   $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

   $galleries = getGalleries($pdo);
   if ( isset($_GET['gallery']) ) {
      $paintings = getPaintings($pdo, $_GET['gallery']);
   }
   $pdo = null;
}
```

3 Add the following functions:

```
function getGalleries($pdo) {
   $sql = "SELECT GalleryID, GalleryName FROM Galleries
             ORDER BY GalleryName";
   $result = $pdo->query($sql);
   return $result->fetchAll(PDO::FETCH_ASSOC);
}

function getPaintings($pdo, $id) {
   $sql = "SELECT PaintingID, Title, YearOfWork, ImageFileName
            FROM Paintings WHERE GalleryID=?";
   $statement = $pdo->prepare($sql);
   $statement->bindValue(1, $id);
   $statement->execute();
   return $statement->fetchAll(PDO::FETCH_ASSOC);
}
```

Notice that you are passing in the PDO connection object to both functions, and that each function encapsulates a single query.

4 Test.

Notice that the gallery identifier is passed within the link URL via a query string.

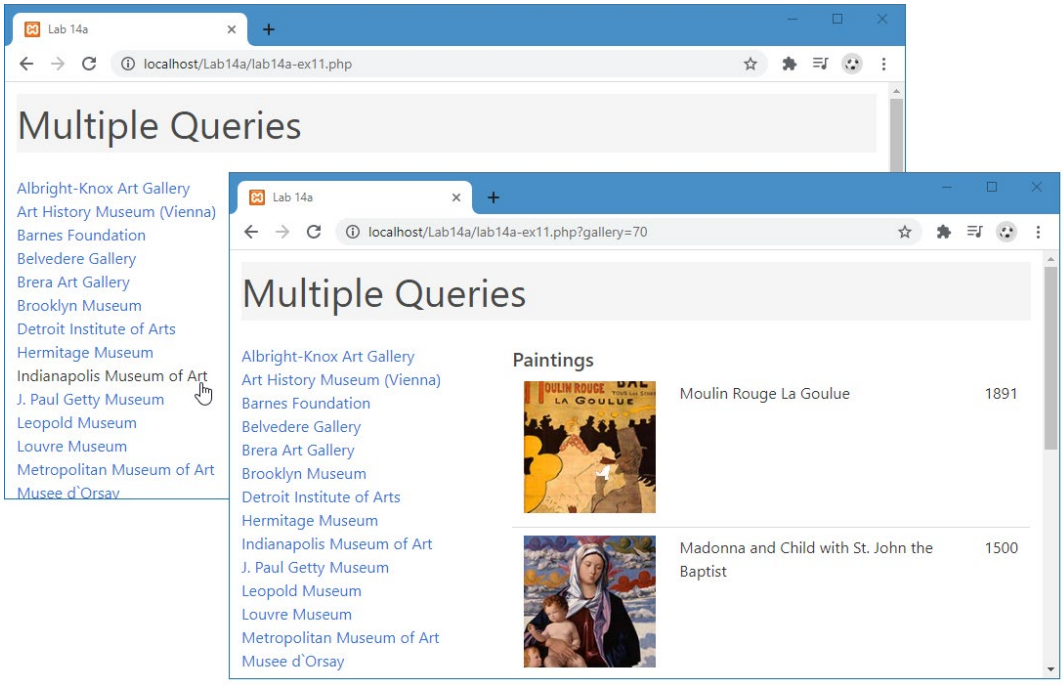

Figure 14a.5 – Completed Exercise 14a.11

Modifying Data

EXERCISE 14a.12 — Inserting Data

1 Open and examine `lab14a-ex12.php`.

This page already contains code for outputting records in the Editable table (which you added in Test Your Knowledge #1). You will be writing the code for inserting the form data into this table.

2 Add the following function.

```
function addPerson($pdo, $name, $email) {
   $sql = "INSERT INTO Editable (name, email) VALUES
            (:name,:email)";
   $statement = $pdo->prepare($sql);
   $statement->bindValue(':name', $name);
   $statement->bindValue(':email', $email);
   $statement->execute();
}
```

3 Add the following

```
try {
   $pdo = new PDO(DBCONNSTRING,DBUSER,DBPASS);
   $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

    // if it is a POST, then there is user input to save
   if ($_SERVER["REQUEST_METHOD"] == "POST") {
      addPerson($pdo, $_POST['name'], $_POST['email']);
   }

   $people = getPeople($pdo);
   $pdo = null;
}
```

4 Test by adding a new person.

5 Modify the `addPerson` function as follows and test.

```
function addPerson($pdo, $name, $email) {
   $sql = "INSERT INTO Editable (name, email) VALUES
            (:name,:email)";
   $statement = $pdo->prepare($sql);
   $statement->execute( array("name"=>$name,"email"=>$email));
}
```

This uses a shortcut approach for binding data to parameter variables.

Data Access Design and JSON

EXERCISE 14a.13 —Data Access Design

1 Create a new file named `lab14a-db-classes.inc.php`.

2 Add the following code to this file and save.

```
<?php
class DatabaseHelper {
   /*  Returns a connection object to a database  */
   public static function createConnection( $values=array() ) {
      $connString = $values[0];
      $user = $values[1];
      $password = $values[2];
      $pdo = new PDO($connString,$user,$password);
      $pdo->setAttribute(PDO::ATTR_ERRMODE,
                PDO::ERRMODE_EXCEPTION);
      $pdo->setAttribute(PDO::ATTR_DEFAULT_FETCH_MODE,
                PDO::FETCH_ASSOC);
      return $pdo;
   }
   /*
    Runs the specified SQL query using the passed connection and
    the passed array of parameters (null if none)
   */
   public static function runQuery($connection, $sql,
           $parameters=array())     {
      // Ensure parameters are in an array
      if (!is_array($parameters)) {
         $parameters = array($parameters);
      }

      $statement = null;
      if (count($parameters) > 0) {
        // Use a prepared statement if parameters
         $statement = $connection->prepare($sql);
         $executedOk = $statement->execute($parameters);
         if (! $executedOk) throw new PDOException;
      } else {
        // Execute a normal query
         $statement = $connection->query($sql);
         if (!$statement) throw new PDOException;
      }
      return $statement;
   }
}
```

2 Add the following to the top of `lab14a-ex13.php`.

```
<?php
require_once 'config.inc.php';
require_once 'lab14a-ex13-helpers.inc.php';
require_once 'lab14a-db-classes.inc.php';

try {
   $conn = DatabaseHelper::createConnection(array(DBCONNSTRING,
            DBUSER, DBPASS));
   $artSql = "SELECT ArtistID,FirstName,LastName FROM Artists
            ORDER BY LastName";
   $artists = DatabaseHelper::runQuery($conn,$artSql,null);

   if (isset($_GET['id']) && $_GET['id'] > 0) {
      $paintSql = 'select * from Paintings where ArtistId=?';
      $paintings = DatabaseHelper::runQuery($conn, $paintSql,
            Array($_GET['id']));
   } else {
      $paintResults = null;
   }
} catch (Exception $e) { die( $e->getMessage() ); }
?>
```

3 Add the following to the markup and test.

```
<div class="four wide column">
   <ul class="ui link list">
      <?php outputArtists($artists); ?>
   </ul>
</div>
<div class="twelve wide column">
   <div class="ui items">
      <?php outputPaintings($paintings); ?>
   </div>
</div>
```

4 Add the following to the following functions in `lab14a-ex13-helpers.inc.php`.

```
function outputArtists($results) {
   foreach($results as $row) {
      outputSingleArtist($row);
   }
}
function outputPaintings($results) {
   if ($results) {
      foreach($results as $row) {
         outputSinglePainting($row);
      }
   }
}
```

4 While okay, this isn't as clean as we would like. It still contains a lot of database details in the markup files. Add the following to `lab14a-db-classes.inc.php`.

```
class ArtistDB {
    private static $baseSQL = "SELECT * FROM Artists
        ORDER BY LastName";

    public function __construct($connection) {
        $this->pdo = $connection;
    }

    public function getAll() {
        $sql = self::$baseSQL;
        $statement =
           DatabaseHelper::runQuery($this->pdo, $sql, null);
        return $statement->fetchAll();
    }
}

class PaintingDB {
    private static $baseSQL = "SELECT * FROM Paintings ";

    public function __construct($connection) {
        $this->pdo = $connection;
    }

    public function getAll() {
        $sql = self::$baseSQL;
        $statement =
           DatabaseHelper::runQuery($this->pdo, $sql, null);
        return $statement->fetchAll();
    }
    public function getAllForArtist($artistID) {
        $sql = self::$baseSQL . " WHERE Paintings.ArtistID=?";
        $statement = DatabaseHelper::runQuery($this->pdo, $sql,
           Array($artistID));
        return $statement->fetchAll();
    }
}
```

This creates two table gateway classes. A gateway class ideally encapsulates all database access details within it.

5 Comment out the `try..catch` in `lab14a-ex13.php`, replace it with the following. Test.

```
try {
   $conn = DatabaseHelper::createConnection(array(DBCONNSTRING,
               DBUSER, DBPASS));
   $artGateway = new ArtistDB($conn);
   $artists = $artGateway->getAll();
   if (isset($_GET['id']) && $_GET['id'] > 0) {
      $paintGateway = new PaintingDB($conn);
      $paintings = $paintGateway->getAllForArtist($_GET['id'] );
   } else {
      $paintings = null;
   }
} catch (Exception $e) {
   die( $e->getMessage() );
}
```

This is now cleaner code in the markup page.

6 Replace the SQL in the `PaintingDB` class (`lab14a-db-classes.inc.php`) with the following:

```
private static $baseSQL = " SELECT PaintingID, Paintings.ArtistID
AS ArtistID, FirstName, LastName, ImageFileName, Title, Excerpt
FROM paintings INNER JOIN artists ON paintings.ArtistID =
artists.ArtistID";
```

In production code, we generally do not use the wildcard field specifier (`SELECT *`), so here we are explicitly indicating which fields we want. As well, notice that this SQL includes an inner join so that we have access to the artist information (`FirstName` and `LastName`).

7 Modify the `outputSinglePainting` function in `lab14a-ex13-helpers.inc.php`:

```
function outputSinglePainting($row) {
   echo '<div class="item">';
   echo '<div class="image">';
   echo '<img  src="images/art/works/square-medium/' .
          $row['ImageFileName'] .'.jpg">';
   echo '</div>';
   echo '<div class="content">';
   echo '<h4 class="ui header">';
   echo $row['Title'];
   echo '</h4>';
   echo '<div class="meta">';
   echo $row['FirstName'] .  ' ' . $row['LastName'];
   echo '</div>';
   echo '<p class="description">';
   echo $row['Excerpt'];
   echo '</p>';
   echo '</div>';
   echo '</div>';
}
```

8 Test. The result should be similar to that shown in Figure 14a.6

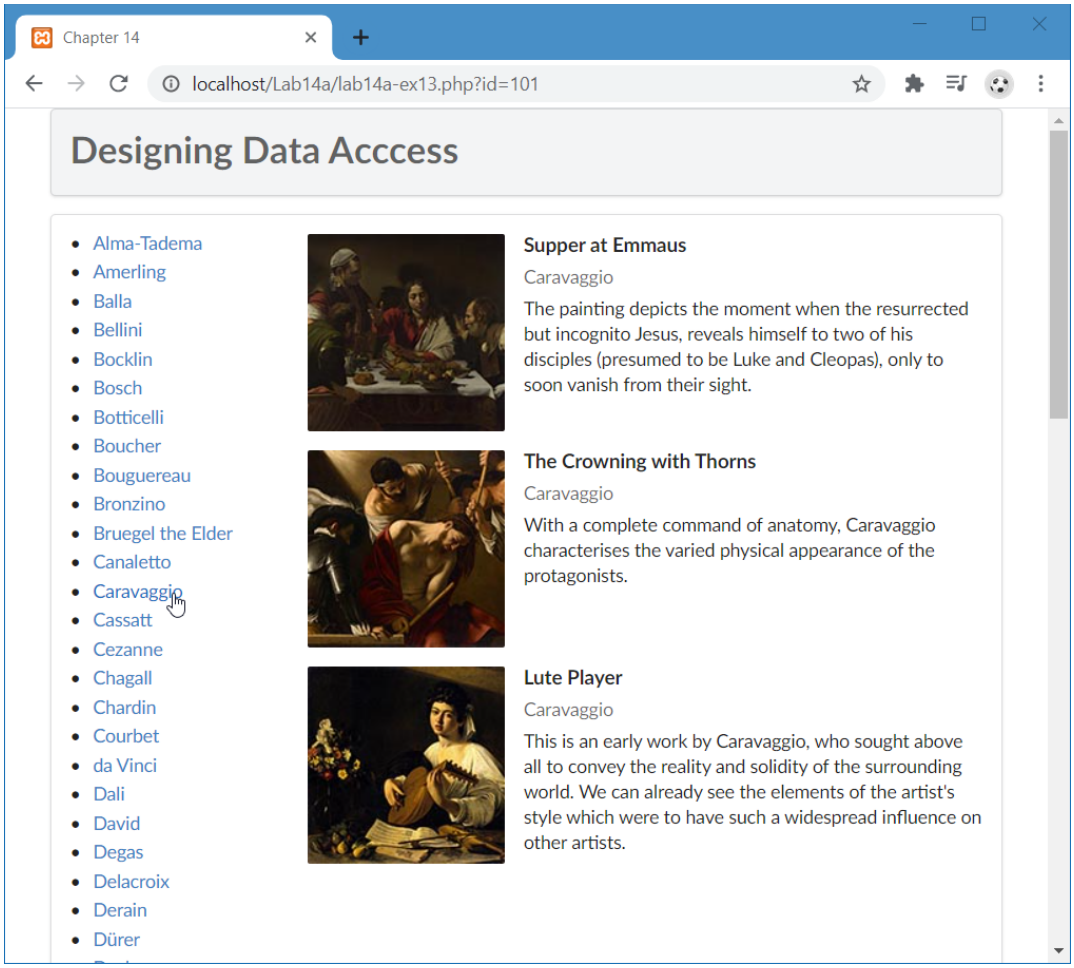

Figure 14a.6 – Completed Exercise 14a.13

EXERCISE 14a.14 — WEB API USING PHP

1 Create a new file named `painting-api.php`.

2 Add the following code.

```
<?php
require_once 'config.inc.php';
require_once 'lab14a-db-classes.inc.php';

// Tell the browser to expect JSON rather than HTML
header('Content-type: application/json');
// indicate whether other domains can use this API
header("Access-Control-Allow-Origin: *");

try {
```

```
    $conn = DatabaseHelper::createConnection(array(DBCONNSTRING,
                DBUSER, DBPASS));
    $gateway = new PaintingDB($conn);
    $paintings = $gateway->getAll();

    echo json_encode( $paintings, JSON_NUMERIC_CHECK );
} catch (Exception $e) { die( $e->getMessage() ); }
?>
```

3 Test in browser.

This returns JSON for all the paintings. In the next step, we will add the ability to return just a single painting or paintings for a specific artist.

4 Add the following function to `painting-api.php`:

```
function isCorrectQueryStringInfo($param) {
    if ( isset($_GET[$param]) && !empty($_GET[$param]) ) {
        return true;
    } else {
        return false;
    }
}
```

5 Modify `painting-api.php` as follows:

```
$gateway = new PaintingDB($conn);
if ( isCorrectQueryStringInfo("artist") )
    $paintings = $gateway->getAllForArtist($_GET["artist"]);
else if ( isCorrectQueryStringInfo("gallery") )
    $paintings = $gateway->getAllForGallery($_GET["gallery"]);
else
    $paintings = $gateway->getAll();

echo json_encode( $paintings, JSON_NUMERIC_CHECK );
```

6 Modify the SQL in the `PaintingDB` class (`lab14a-db-classes.inc.php`):

```
private static $baseSQL = "SELECT PaintingID, Paintings.ArtistID,
FirstName, LastName, Paintings.GalleryID, GalleryName,
ImageFileName, Title, Excerpt FROM Galleries INNER JOIN (Artists
INNER JOIN Paintings ON Artists.ArtistID = Paintings.ArtistID) ON
Galleries.GalleryID = Paintings.GalleryID ";
```

7 Modify the `PaintingDB` class (`lab14a-db-classes.inc.php`) by adding the following function:

```
 public function getAllForGallery($galleryID) {
    $sql = self::$baseSQL . " WHERE Paintings.GalleryID=?";
    $statement = DatabaseHelper::runQuery($this->pdo, $sql,
                    Array($galleryID));
    return $statement->fetchAll();
}
```

8 Test by using the following requests:

`http://localhost/lab14a/painting-api.php`

`http://localhost/lab14a/painting-api.php?artist=2`

`http://localhost/lab14a/painting-api.php?gallery=17`

Test Your Knowledge #3

1 You have been provided with the markup for the next exercise in the file `lab14a-test03.php`.

2 Create a new class named `GalleryDB` in `lab14a-db-classes.inc.php`. This will need a `getAll()` method that will return all the galleries. Your SQL will need to include the fields `GalleryID` and `GalleryName` from the `Galleries` table and be sorted by `GalleryName`.

3 Fill the `<select>` list with a list of gallery names using the method created in step 2. Set the `value` attribute of each `<option>` to the `GalleryID` field.

4 Add a new method to `PaintingDB` in `lab14a-db-classes.inc.php`. This method will return just the top 20 paintings, sorted by `YearOfWork`. Simply append "`LIMIT 20`" to the end of the SQL. You will need to also add `YearOfWork` and `ImageFileName` to the SQL.

5 Modify `lab14a-test03.php` so that it initially displays the top 20 paintings using the method created in Step 4. The file `lab14a-test03.php` has the sample markup for a single painting.

6 When the user selects from the museum list (remember we are not using JavaScript so the user will have to click the filter button which re-requests the page), display just the paintings from the selected museum/gallery. This will require adding a new method to your `PaintingDB` class that returns paintings with the specified `GalleryID`.

7 The result should look similar to that shown in Figure 14a.7.

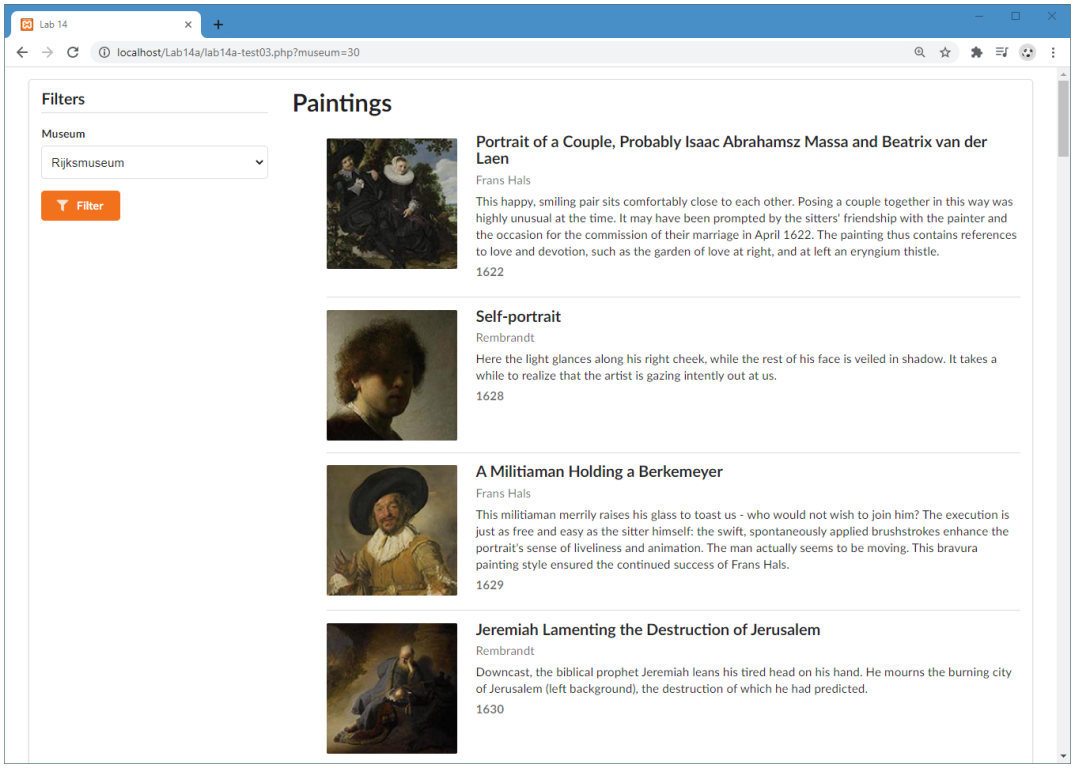

Figure 14a.7 – Test Your Knowledge #3