PHP and SQLite

CS106E, Young

In this handout, we learn how to connect our PHP webpages to a SQLite database.

Creating and Setting Up the PDO

The key to connecting our PHP program to a database is an object called the PDO (this is an abbreviation for PHP Data Objects). The PDO can be used to connect to a wide range of databases including MySQL, Oracle, IBM, and Postgres databases, if the server is properly configured to use them.

We create a new PDO object and pass in information on the type of database and the actual database location like this:

```php
$db = new PDO("sqlite:cities.db");
$db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_WARNING);
```

Our first line connects to the “cities.db” SQLite database file, assuming that the cities.db file is in the same directory as our PHP file. The second line sets up error message settings. Without this, if a SQL call to the database fails, you’ll get no warning – your program will simply fail to work. With this line included, SQL error messages will appear on the webpage, the same way using the “php.ini” file causes PHP errors to appear on a webpage.

Query and Exec

Once we’ve created our PDO object, we’ll be able to perform SQL operations on the database the PDO object is associated with. There are two different methods we’ll be using on the PDO object – query and exec.

query – We use query when we expect to get database information back from the call. In CS106E we’ll be using this exclusively when we want to do a “SELECT” to get data out of the database. It should also be used with the “JOIN” SQL command, which can combine data from multiple tables as briefly discussed in the SQL handout.

exec – We use exec for SQL commands which do not return data from the database. This includes insert, update, and delete.\(^1\)

Exec

Let’s start off with the exec method, as it’s easier to use than query. Simply take the text that you would have typed into the SQLite shell and pass it into the exec command as a string. For example, if I were working from the SQLite shell with the Cities Database described in the SQL handout, and I wanted to add another city, I could type:

```
```

\(^1\) You can also use exec for “create table” commands, although I don’t recommend creating your tables from PHP. Remember you’ll only want to create your table once, not every time your webpage loads. So create it manually through the SQLite shell.
From PHP, I enter the same thing, minus the semicolon. Here’s my exec call on the PDO.

```
$db->exec("INSERT INTO cities VALUES('Lagos','Africa',8048000);"
```

Notice the use of quotes here. The double quotes denote a string for PHP, and the single quotes denote strings for SQL. If I accidentally use single quotes for PHP or double quotes for SQL, this line will give me a syntax error. If I were to replace the single quotes around Lagos with double quotes, SQL would see the first thing surrounded by double quotes, which would be the string:

```
"INSERT INTO cities VALUES(" 
```

and wouldn’t get past that second double quote to any of the rest of the statement.

If I have a webpage with an HTML form like this:

```
<form action="add-city.php" method="post">
    Name: <input type="text" name="city" /><br />
    Continent: <input type="text" name="continent" /><br />
    Population: <input type="text" name="population" />
    <input type="submit">
</form>
```

My PHP webpage can retrieve the data from the form and enter it in the cities database using the following commands.

```
$city = $_POST["city"]; 
$cont = $_POST["continent"]; 
$pop = $_POST["population"]; 
$db->exec("INSERT INTO cities VALUES('$city','$cont',$pop);"
```

Important: notice the use of both double and single quotes in the exec method call. For SQL, the city name is a string as is the continent name. Therefore, I need to put single quotes around them. **If you don’t put those single-quotes around strings you pass in to your SQL calls, you will get a SQL error.**

Delete and update work similarly. For example, I can delete a city from my database using:

```
$db->exec("DELETE FROM cities WHERE city = '$city';"
```

Again, note the use of single and double quotes. A pair of double quotes goes around the entire SQL command, since that’s a string from PHP’s perspective. Looking inside of the double quoted PHP string, the word cities is a table name and the first instance of city is a column name, neither of these are quoted in SQL. On the other hand, the actual city name that we are looking up does go in quotes because it’s a SQL string, and its value is retrieved using PHP’s variable substitution mechanism.

### Query

Query works similar to exec in that we can just pass in the string that we would have typed into the SQLite shell when using a SELECT to retrieve data. However, in this case, we need to
get data back from the query call. The data comes back in a data object called a PDOStatement. This has a number of different methods on it, but for our purposes, the important thing is that we can iterate through the results using PHP’s foreach control structure.

Let’s see how we would iterate through our list of cities. First, we call query on the PDO object. In this case we want all the cities. If we were typing in the SQLite shell we would enter:

```
SELECT * FROM cities;
```

So we’re going to enter the same thing as a parameter to our exec call, we’re going to store the PDOStatement returned in a variable:

```
$results = $db->query("SELECT * FROM cities");
```

In this example, I’m retrieving all the cities in the database, but we could restrict the cities returned using a WHERE clause as show in the SQL handout. Just enter what you would have typed in the SQLite shell as the parameter to exec method.

Now we just iterate through all the $results using foreach. Each iteration in the loop we will get a row. The row acts as an associative array, allowing us to retrieve the contents of each column within the row, via the column name:

```
foreach ($results as $row) {
  $currCity = $row["city"];  
  $currContinent = $row["continent"]; 
  $currPopulation = $row["population"]; 
  echo "$currCity in $currContinent with $currPopulation population. <br />
```

If our query returns no results – for example, if the table is currently empty – we can still use the foreach on it, we simply won’t have any items to go over, so the loop statements will be skipped over and nothing will get printed out.

Note: you cannot tell if there are no results returned by simply checking to see whether or not $results exists. Even if no results are returned query will still return a PDOStatement object.

---

**Debugging**

If your webpage isn’t working correctly, here are some helpful techniques for debugging.

**Access the Database via the SQLite Shell**

You can’t easily tell what’s in the database from PHP. Log into the Stanford Unix machines and access the database directly with the SQLite shell. You can perform SELECT’s using the SQLite Shell in order to find out exactly what’s in the database.

As databases are setup for simultaneous access, there’s no problem accessing the database with the SQLite Shell at the same time you’re accessing the database from your PHP webpage.

**Printout and Execute Your Commands**

Trying to figure out why SQL instructions entered via PHP isn’t working can be frustrating. Even with the error messages turned on using the $db->setAttribute(...) call, error messages from the SQLite Shell are easier to understand than PHP error messages.
You can print out your SQL commands from PHP and then enter them into the SQLite shell directly. For example:

```php
$cmd = "INSERT INTO cities VALUES('$city','$cont','$pop');
echo "<p>$cmd</p>";
$db->exec($cmd);
```

will both printout the command string to the webpage and will execute it. I can then look at the webpage, see exactly what my exec call attempted to execute on the database. I can also copy that command string from the webpage and paste it directly into the SQLite shell and see what the SQLite shell does with it.

**print_r**

Don’t forget about the print_r mentioned in the previous handout. If you are not sure, for example, what iterating through the $result is getting you, try a print_r on each row:

```php
foreach ($result as $row) {
    echo "<p>
    print_r($row);
    echo "</p>";
}
```

### SQL Injection Attacks

Don’t use the SQL I’ve shown you here on actual production websites. There’s an extra step you’ll want to take for actual use. Using a prepared statement will prevent SQL Injection attacks. Prepared statements build on what I’ve just showed you, and aren’t particularly hard to use, but they do add some extra complexity to the process. You can learn about how to use them here.


So what’s a SQL injection attack? I think the following XKCD cartoon provides the best (and most humorous explanation):

**Exploits of a Mom**

---

Putting the string inside of a set of paragraph tags as I’ve done here isn’t strictly necessary. However, it will make your SQL command standout from the rest of the webpage results.
A SQL injection attack inserts SQL commands where the program is only expecting text. If we simply take the user’s inputs and insert them into a string that we then call exec or query on, someone can take advantage of this to execute their own SQL commands.