Transactions, prepared statements, and PDO

INFO/CS 2300:
Intermediate Web Design and Programming

Assignments

HW2 – A TA will review the 3 Piazza posts and regrade submissions – not immediately

P3M1 – should be graded.

P3M2 – If you submitted an extension request, we're using that If not graded within 48 hours of your due date, email Hajin HL934@cornell.edu.

HW3 – coming soon – optional

Final Project – coming soon

List - detail

rentals.php – PHP loop

```
while ( $row = $result->fetch_assoc() ) {
    $rental_date = $row[ 'rental_date' ];
    $last_name = $row[ 'last_name' ];
    $title = $row[ 'title' ];
    $customer_id = $row[ 'customer_id' ];
    $customer_url = "customer.php?customer_id=$customer_id";

    print('<tr>');
        print "<td>$rental_date</td>";
        print "<td><a href='$customer_url'>$last_name</a></td>";
        print "<td>$title</td>";
    print('</tr>');
}
```

Link to a specific customer using the id

FILTER_SANITIZE_NUMBER_INT

customer.php

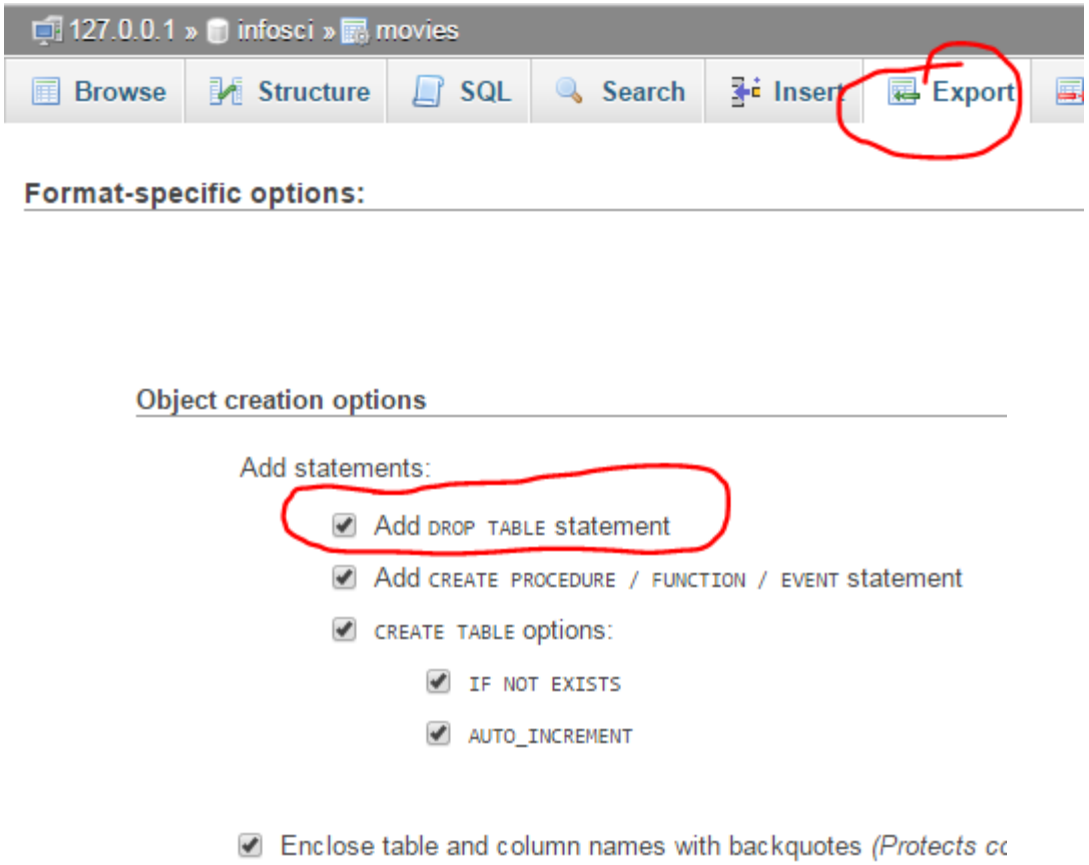
```
$input_customer_id = filter_input( INPUT_GET, 'customer_id', … );
if( empty ( $input_customer_id ) ) {
    echo '<p>Sorry but no customer was found.</p>';
} else {
    require_once 'includes/functions.php';
    $customer = get_customer( $input_customer_id );

    $first_name = $customer[ 'first_name' ];
    $last_name = $customer[ 'last_name' ];
    $email = $customer[ 'email' ];

    echo "<h1>$first_name $last_name</h1>";
    echo "<p>Email: $email</p>";
}
```

List - detail

For a working example, see the movies demo from lecture 12

Copying MySQL Dbs

phpMyAdmin


127.0.0.1 » infosci » movies
Browse
Structure
SQL
Search
Insert
Export
Format-specific options:
Object creation options
Add statements:
Add DROP TABLE statement
Add CREATE PROCEDURE / FUNCTION / EVENT statement
CREATE TABLE options:
IF NOT EXISTS
AUTO_INCREMENT
Enclose table and column names with backquotes (Protects co

DROP TABLE

Deletes the table

Prevents overwrite

```
DROP TABLE IF EXISTS `movies`;
CREATE TABLE IF NOT EXISTS `movies` (
  `movie_id` int(11) NOT NULL AUTO_INCREMENT,
  `title` varchar(255) COLLATE latin1_general_ci NOT NULL,
  `year` varchar(255) COLLATE latin1_general_ci NOT NULL,
  `length` int(11) DEFAULT NULL,
  PRIMARY KEY (`movie_id`)
) ENGINE=InnoDB  DEFAULT CHARSET=latin1
    COLLATE=latin1_general_ci AUTO_INCREMENT=17 ;
```

You would normally use either DROP TABLE or IF NOT EXISTS

Click In!

Click In!

Server on your local computer connecting with MySQL

A. MAMP and have had problems

B. MAMP no problems

C. XAMPP and problems

D. XAMPP no problems

E. I'm not running MAMP or XAMPP

Transactions

Transactions

So far we've assumed that one SQL statement at a time is OK. But sometimes this could lead to problems.

Example: Ticket sales

Suppose we use a database to maintain the number of available tickets for an event.

```
<form method="post">

    <p>How many tickets would you like?
        <select name="amt">
            <?php
                for ($i = 1; $i <= 5 && $i <= $amt; $i++) {
                    print( "<option value= '$i' >$i</option>" );
                }
            ?>
        </select>
    </p>

<input type="submit" name="trans" value="Buy tickets">
</form>
```

```
if (isset($_POST['trans'])) {
    $purchased = filter_input( INPUT_POST, 'amt', FILTER_SANITIZE_NUMBER_INT );
    $query = "UPDATE tix SET count = count - $purchased";
    $mysqli->query($query);
    print("<p>Thank you for your purchase of
                                        $purchased tickets</p>");
}


$result = $mysqli->query("SELECT count FROM tix");
$row = $result->fetch_assoc();
$amt = $row['count'];

print( "<p>Number of tickets left: $amt</p>" );
$mysqli->close();
```

Now suppose…

We have two users trying to get tickets at the same time.

Click In!

Click In!

How can we solve the potential overselling problem introduced by this SQL statement?

```
$query = "UPDATE tix SET count = count - $purchased";
```

A. Use SQL to check the DB before this update
B. Use SQL to check the DB after this update
C. Keep 5 tickets in reserve
D. Use a SQL transaction
E. None of the above

Click In!

How can we solve the potential overselling problem introduced by this SQL statement?

```
$query = "UPDATE tix SET count = count - $purchased";
```

A. Use SQL to check the DB before this update
B. Use SQL to check the DB after this update
C. Keep 5 tickets in reserve
D. Use a SQL transaction
E. None of the above

Involves guessing and is vague

A & B: DB can change by another user in between queries

What’s the problem?

Under normal circumstances, when a web page runs 2 SQL queries, there is always a possibility that another user / page will run a query in between.

What would we like to happen?

We'd like to check the status of the database and make a change without anything else getting in between.

ACID transactions

We like the steps of the update to be grouped into one single transaction. We'd like transactions to be:

- Atomic: Either all the steps of a transaction happen or none of them do.
- Consistent: Transactions leave the DB in a consistent state.
- Isolated: Transactions execute independently of one another.
- Durable: A successfully completed transaction is permanently recorded in the DB.

Transactions in SQL

To group a set of SQL statements into a transaction, we first issue a command ‘START TRANSACTION’.

At the end of the group, if we are ready to have all the steps take effect, we issue a command ‘COMMIT’.

If at any point prior to the commit, we want to abort the transaction, we issue a command ‘ROLLBACK’.

Transaction Requirement

For tables in MySQL for which you will want to do transactions, need to set the type of the table to be 'InnoDB'.

E.g.

```
CREATE TABLE `tix` (  `event` varchar(20) NOT NULL,
  `count` int(5) default NULL,  PRIMARY KEY
  (`event`)) ENGINE=InnoDB DEFAULT
  CHARSET=latin1;
```

The default table engine, MyISAM, doesn’t allow for standard SQL transactions.

Transactions in MySQL

We can start a transaction with:

//$mysqli->begin_transaction(); //Doesn't work

//$mysqli->autocommit(FALSE); //Might be true or false - set it back?

```
$mysqli->query( "START TRANSACTION" );
```

Then carry out the steps of the transaction, and either commit or rollback.

```
$mysqli->commit();  or
$mysqli->rollback();
```

```
if (isset($_POST['trans'])) {
  $purchased =
  $mysqli->query("START TRANSACTION");
  $query = "UPDATE tix SET count = count - $purchased";
  $mysqli->query($query);

  $result = $mysqli->query("SELECT count FROM tix");
  $row = $result->fetch_row();
  $amt = $row[0];
  if ($amt < 0) {
     $mysqli->rollback();
    print("Your transaction did not go through");
   } else {
     $mysqli->commit();
     print("<p>Thank you for your purchase of $purchased
          tickets</p>");
  }
}
```

Prepared statements

Its not hard to imagine writing code like this:

```
$username = $_POST[ 'username' ];
$password = hash( 'sha256', $_POST[ 'password' ] );
$query = "SELECT * FROM users WHERE username =
    '$username' AND password = '$password'; ";
```

But in fact this is pretty dangerous. Why?

Click In!

Click In – What's wrong?

```
$post_username = $_POST[ 'username' ];
$post_password = $_POST[ 'password' ];
$hashed_password = hash[ "sha256", $post_password ];

$query = "SELECT * FROM users
    WHERE username = '$post_username'
    AND password = '$hashed_password';";
```

A. User input not sanitized
B. Password is displayed in plain text
C. username passed directly to database
D. A and C
E. None of the above

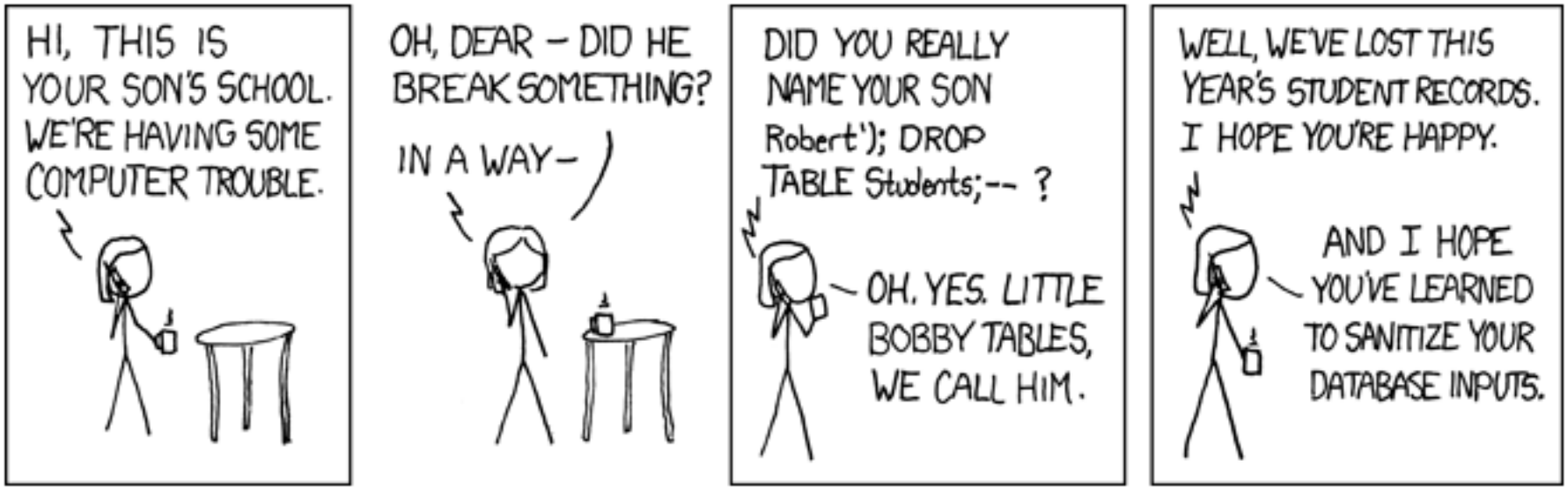
Click In – What's wrong?

```
$post_username = $_POST[ 'username' ];
$post_password = $_POST[ 'password' ];
$hashed_password = hash[ "sha256", $post_password ];

$query = "SELECT * FROM users
    WHERE username = '$post_username'
    AND password = '$hashed_password';";
```

A. User input not sanitized
B. Password is displayed in plain text
C. username passed directly to database
D. A and C
E. None of the above

```
$query = "SELECT *
                FROM users
                WHERE username = '$username'
                AND password = '$password'; ";

//If exactly one record is returned, call it a success
```

SQL Injection



Prepared statements

```
$username = $_POST['username'];
$password = hash( "sha256", $_POST[ 'password‘ ] );

$query = "SELECT * FROM users
    WHERE username = ? AND hashpassword = ?";

$stmt = $mysqli->stmt_init();

if ($stmt->prepare($query)) {
    $stmt->bind_param( ‘ss’ , $username, $password );
    $stmt->execute();
    $result = $stmt->get_result();
}
```

Parameters for bind_param:

- s – string
- i – integer
- d – double
- b – binary

Prepared statements ensure that

- inputs are properly quoted
- only the intended SQL is executed

Can also use $mysqli->real_escape_string():
e.g.

```
$user =
  $mysqli->real_escape_string($_POST['user']);
```

Analogous to htmlentities().

PDO: Database abstraction

PHP Data Objects

PHP 5 has a built-in class to handle calls to DBs called PDO.

Analagous to mysqli but more general

PDO Methods

PDO(“mysql:host=hostname;dbname=dbname”, username, password)

This is a constructor method that returns a PDO object connecting to a MySQL DB.

E.g.

```
$db = new PDO( 'mysql:host=' . DB_HOST . ';dbname=' . DB_NAME, DB_USER, DB_PASSWORD );
```

$db->exec(*string*)

Executes SQL query string and returns number of affected rows. But you would usually use a prepared statement

$db->query(*string*)

Executes SQL query string and returns a “PDO Statement”

E.g.

$stmt = $db->query(“SELECT * FROM Movies”);

$*PDOstatement*->fetch($fetch_style)

Returns next row of the table as an array according to fetch_style. PDO::FETCH_NUM, PDO::FETCH_ASSOC

E.g.

```
while ($row = $stmt->fetch( PDO::FETCH_ASSOC )) {
        print("<p>".$row['Title']."</p>");
        …
}
```

A shortcut

Can actually iterate through the results as follows.

```
$query = “SELECT * FROM Movies”;
foreach ($db->query($query) as $row) {
        print(“<p>”.$row[ ‘Title’ ].“</p>”);
        …
}
```

Prepared statements

PDO also has prepared statements.

```
$query = "SELECT * FROM users
   WHERE username = ? AND hashpassword = ?";
$stmt = $db->prepare($query);
$stmt->bindParam(1,$_POST[ ‘username’ ]);
$stmt->bindParam(2,hash(‘sha256’,$_POST['password']));
$stmt->execute();
while ($row = $stmt->fetch()) {
…
}
```

Named parameters

Named parameters make code easier to read

```
$query = "SELECT * FROM users
    WHERE username = :username
    AND hashpassword = :password";
$stmt = $mypdo->prepare($query);
$stmt->bindParam( ':username', $_POST[ ‘username’ ] );
$password = hash("sha256",$_POST['password']);
$stmt->bindParam( ':password' , $password );
$stmt->execute();
while ($row = $stmt->fetch()) {
…
}
```

Transactions

Transactions in PDO are cleaner than in mysqli.

```
$mypdo->beginTransaction();
```

to start the transaction

```
$mypdo->commit();
```

to commit the current transaction

```
$mypdo->rollback();
```

to roll back the current transaction

But why?

One reason: Then we can easily change the database we're working with.

SQLite

SQLite is another DBMS (like MySQL), except that the DB gets stored in a file (which is then easy to copy from one machine to another).

Installing/configuring PDO/SQLite

(XAMPP) Need to uncomment in php.ini:

extension=php_pdo_sqlite.dll.

Can download a simple command line interface: sqlite3.exe from ww.sqlite.org.

Firefox extension: SQLite Manager

The change

The only change to the code you would need to make to use SQLite instead is the initial connection

```
$db = new PDO('sqlite:infosci.sqlite' );
$query = 'SELECT * FROM movies';
foreach ($db->query($query) as $row) {
    print('<p>'.$row['Title'].'</p>');
}
```

Why might this be useful?

- Have PHP installed but not MySQL
- Makes site easier to move

However

- Not as robust – no passwords
- size

Other DBMS

PDO can also work with:

- PostgreSQL
- MS SQL Server
- … and many others.

Review

- When writing database code than can be accessed by multiple users simultaneously, need to think through issues of what can happen; SQL provides means for atomic *transactions* to deal with this.
- Prepared statements help us avoid SQL injection attacks.
- PDO lets us abstract away the DB being used.

Reminders…

P3 M3 due Tuesday