Computer Science 75
Spring 2009
Scribe Notes

Lecture 5: March 9, 2009
Andrew Sellergren

Contents

Computer Science 75 Lecture 5: March 9, 2009
Spring 2009 Andrew Sellergren
Scribe Notes

1 Announcements (0:00–2:00)

- Project 1 is due on March 16, 2009 at noon!
- When using the Bulletin Board (please do!), be sure to search for previous posts pertaining to your question before posting your own.
- Avail yourself of the scribe notes[1] so that you don't have to suffer the tedium of scribbling down every little tidbit during lecture.[2]

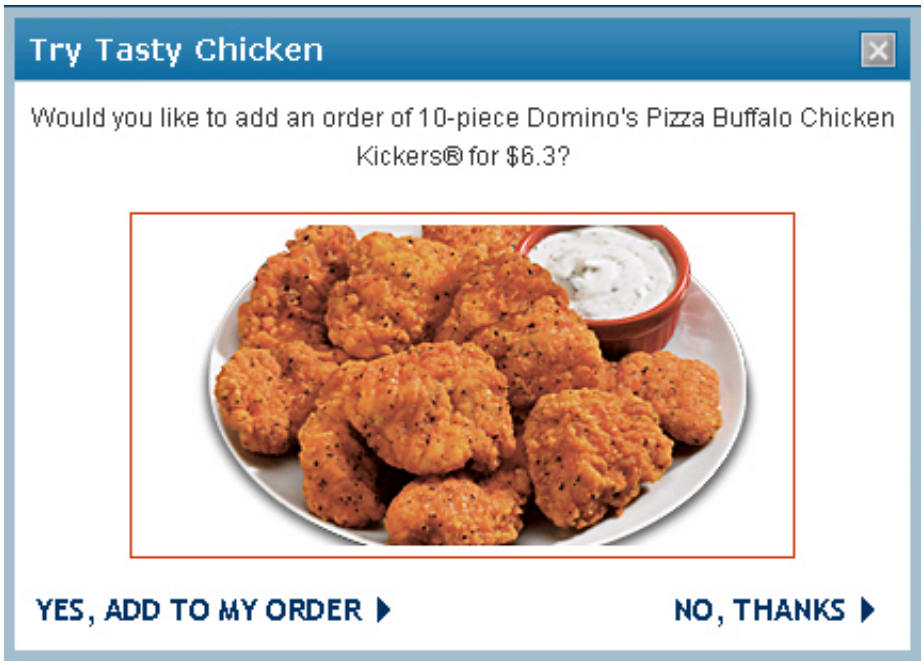
2 Quick Examples (2:00–8:00)

- What can we learn from the example of Domino's online ordering system? When we access it, the URL in our browser looks something like the following:

  `http://www28.order.dominos.com/.../login.jsp;jsessionid=13659CF85996...`

  First, we see that the URL doesn't begin with `https`, so we know that they're not using SSL for possibly sensitive information. Second, we see that the extension of their webpage is `.jsp`, so we know they're using a Java environment. Third, we see that a long alphanumeric string is being passed, so it seems that Domino's is opting to send cookies via the URL rather than the headers. Finally, (and perhaps David's original point), the URL begins with `www28`, which indicates that Domino's is doing load-balancing in a potentially troublesome way, by hardcoding the specific server to which traffic is being directed. What happens, for example, if that particular server goes down and the user tries to access the same URL? Dead end. The better solution is to load balance behind the scenes. In this way, the URL will appear to users the same no matter which server their traffic is directed to.
- At the risk of picking on Domino's too much,[3] we have another fun example:

[1] Wow, so meta.
[2] Leave that tedium to me.
[3] Really, though, David should realize that the sword is double-edged. How often *are* you ordering from Domino's in order to notice all these bugs?



Notice that some kind of formatting error has occurred such that the price of the Buffalo Chicken Kickers is not being displayed with two digits after the decimal point.

- Moving on to pick on The Wall Street Journal, we can see here that the form requires that the user input his account number without spaces, hyphens, or slashes. Frankly, this is just lazy. It's very easy to take care of this parsing server-side or even client-side, so why trouble the user with it? Can you think of a way in PHP to remove spaces, hyphens, and slashes from a user's input?

3 A Quick Word about Grading (8:00–10:00)

- As you can see from page 1<6 of the Project 1 Specification, grading will be done along three axes:
  - Correctness
  - Design
  - Style
- A further note, Correctness, Design, and Style will be weighted in the ration of 3:2:1 and will each be measured along a five-point scale as follows:
  - poor (1)
  - fair (2)

  - good (3)
  - better (4)
  - best (5)

- Know that we'll normalize grades across sections and ultimately we'll take into account improvement throughout the semester. That is, we'll be much more inclined to give breaks if we can see that you're trending upward from Project 1 to Project 3.
- Overall, the focus of grades is merely to make you realize that no sample of code is perfect (even that generated by the staff), so there's always room for improvement.
- Don't hesitate to reach out to your assigned TF or to `help@cs75.net` if you have more specific or private questions about grading.

4 Lolcat of teh Day! (10:00–30:00)

- If you haven't already, be sure to buy yourself a Happy Cat Plush Doll from the LOLMart. David is not ashamed to admit that he already has!
- So how do we go about implementing the lolcat of teh day? Let's take a look at the code below:

```
<!DOCTYPE html PUBLIC
     "-//W3C//DTD XHTML 1.0 Transitional//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Lolcat of teh Day</title>
  </head>
  <body>
    <div align="center" style="padding: 20px;">
      <h1>Lolcat of teh Day</h1>
      <?

          $xml = new SimpleXMLElement(file_get_contents("http://
           feedproxy.google.com/ICanHasCheezburger?format=xml"));
          $item = $xml->channel->item[0];
          preg_match("/^.* - (.*)</", $item->description, $matches);
          $alt = htmlspecialchars($matches[1], ENT_QUOTES);
          $link = $item->link;
          foreach ($item->children("http://search.yahoo.com/mrss/")
            as $content)
          {
```

```
            $attributes = $content->attributes();
            $src = $attributes["url"];
        }
        print("<a href='{$link}'><img alt='{$alt}'
         border='0' src='{$src}' /></a>");

    ?>
   </div>
  </body>
</html>
```

First we have some XHTML that simply spits out a template for displaying the image. Second, we're grabbing the RSS feed in XML format just as last week we were grabbing our own XML file using the `file_get_contents()` function. What about the next line of code?

```
$item = $xml->channel->item[0];
```

Here, we're taking a bit of a leap of faith in assuming that the lolcat will always be the first item under the `channel` tag. Obviously, this is not extensible since if the site changes the format of their RSS feed, our code will break. Moving on:

```
preg_match("/^.* - (.*)</", $item->description, $matches);
$alt = htmlspecialchars($matches[1], ENT_QUOTES);
```

In the first line, we're doing some regular expression matching. We're saying that the item description that we're interested in is going to start with some amount of text (hence the first `.*`) which will be followed by a hyphen and then another amount of text (this time, the `(.*)`) followed by a left-angle bracket. What's the significance of the parentheses? Basically, our regular expression will be matching for the above content but it will also be saving, in the array `$matches` we've specified, everything that's between the two parentheses. For our purposes, this is the caption of the lolcat, which is precisely what we want to capture. After we've gotten the caption, we're calling the `htmlspecialchars()` function, which will automatically escape certain characters like ampersands so that it doesn't invalidate our XHTML when we include it in the `alt` attribute of the `img` tag. The `ENT_QUOTES` specifies that we're worried about quotation marks, in particular. Note that the 1 index of the `$matches` array is due to the fact that all matches to our regular expression are by default stored in the 0 index of the array whereas the caption that we're interested in will be stored at the 1 index.

- What's the deal with the quoted URL that appears in parentheses next to the `children` element we're referencing in the `foreach` loop? Suffice it

to say that there's an aspect of XML called *namespaces* that we haven't mentioned yet. Don't worry too much about this, just know that in this context we want to access the Yahoo namespace, which specifies a certain format for writing the children.

- Once we've duplicated all the attributes, we're ready to spit out the image and its link. Hence the call to the `print()` function. And that's it!

- A few points about the regular expression. First, we're ending the expression with a left-angle bracket. Technically, the lolcat folks have abused XML by using the HTML escape character for the left-angle bracket (`<`). What they should've done is to use the `<![CDATA[...]]` tag. Also, David's regular expression could stand to be a little more precise. Instead of the first `.*`, it might've been better to put `[^\-]*`, which would match all characters except a hyphen. Can you see why?

5 A Little Bit More with XML and CSV (30:00–65:00)

- Databases are all about persistence of data.[4] Project 1 invokes a database of sorts in the form of an XML file. There's no read/write access and no local storage, however. In Project 2, we'll ask you to empower users to register themselves as well as make purchases which will be "remembered" by the server from visit to visit.

- One of the ways we'll be storing persistent data is a SQL (Structured Query Language) database. This is generally pretty easy to get up and running, but there are some caveats. You might need root access on the server, for example, which isn't always possible if the server is shared and you might also incur unwanted overhead in accessing the database. Because of these caveats, it's important to realize that you *do* have alternatives.

- One alternative is the CSV file format. For Project 2, we'll find that Yahoo Finance allows us to download realtime data in CSV file format. Variables such as stock symbol and price are simply separated by commas, making the data very easy to parse.

- Let's say we want to store persistently the contents of a user's shopping cart. Amazon does this, for example, even if your visits to its site are separated by months or longer. How do they do this? They can't use the `$_SESSION` variable, of course, because eventually that data goes away. The cookie will certainly have *some* expiration date which might even be as soon as the browser is closed. Instead, then, you might use a CSV file to store the data on the server's disk. The password files on Unix servers implement data storage in a similar fashion using colons to separate usernames and passwords.

[4] LDO.

- What else could we use? XML has the advantage of being self-evident. That is, unlike CSV, the structure of the data is stored along with the data. Thus, you can infer from it what the fields are. This metadata is what XPath leverages in order to get quick access to the data contained within an XML file.
- Thus if you wanted to implement a database in XML, you could certainly do that. That's what we're doing effectively for Project 1. In fact, the `SimpleXMLElement` has a method called `asXML` which allows you to output your data as an XML string quite easily.
- Realize that there's no particular advantage to storing menus and other data which remains the same across multiple users' pages because sessions are not actually stored in any kind of virtual memory. Rather, they are written to and read from disk in a temporary directory.
- Let's take a look at a real-life example of using XML databases which is very similar to Project 1. Not too long ago, David and his fellow colleagus were subjected to the horror of e-mailing their lunch orders from Rebecca's Cafe to an administrator who was in charge of placing all the orders. David thought, "This is a job for XML!"[5]
- One night at 10 PM, David went through the painstaking process of modeling the menu using XML. The biggest problem, as it turned out, was that users can create their own sandwiches. Thus, David, had to create a `fillings` tag and a `breads` tag and a `cheeses` tag, etc, to encompass all the different sandwich-building options. This was no easy task.
- One of the design decisions you'll notice that David made if you take a look at `menu.xml` is to write the description for the pre-made sandwiches in `CDATA` tags. Why might he have done this? As it turns out, there was one simple corner case in which the sandwich description bolded the words "choice of topping," so David decided to preserve this emphasis and to account for the possibility that *any* sandwich description might have `<strong>` tags.
- Take a look at the code for `lunch.php`, which implements only the Specialty Sandwiches section of the menu for instructive purposes:

```
<?
    $xml = new SimpleXMLElement(file_get_contents("menu.xml"));
?>

<!DOCTYPE html PUBLIC
     "-//W3C//DTD XHTML 1.0 Transitional//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

[5] Yes, that's exactly what David thought. Trust me.

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Lunch</title>
  </head>
  <body>
    <form action="sqlite.php" method="post">
      <b>Name:</b> <input name="name" type="text" />
          <input type="submit" value="Submit Order" />
      <br /><br />
      <table border="0">
        <? foreach ($xml->xpath("/menu/category[@name='Specialty
         Sandwiches']/item") as $item): ?>
          <tr>
            <td valign="top"><input id="<?= $item["name"] ?>"
             name="item" type="radio" value="<?= $item["name"] ?>" />
            </td>
            <td>
              <label for="<?= $item["name"] ?>">
                <b><?= $item["name"] ?></b>
                <br />
                <?= $item ?>
              </label>
            </td>
          </tr>
        <? endforeach ?>
      </table>
    </form>
  </body>
</html>
```

There isn't much to it—and it's a little ugly—but it gets the job done. Notice that we're accessing only the Specialty Sandwich items using XPath. For each of those items, we're adding a row in our table, which will lay things out nicely on the page. One little trick to notice—by using the `label` tag, we can allow users to click anywhere in the text next to the radio button and still select that radio button. A neat feature.

- It's worth noting that this page isn't valid XHTML. Tsk tsk. Can you figure out why?
- So what page is this data being submitted to? You can see that the `action` attribute of the form points to `csv.php`. Let's take a look at it:

```
<?

    // ensure complete form was submitted
```

```
    if (!isset($_POST["name"]) || !isset($_POST["item"]))
    {
        header("Location: http://www.cs75.net/lectures/4/src/lunch/lunch.php");
        exit;
    }

    // open CSV file for appending
    $handle = fopen("orders.csv", "a");

    // acquire exclusive lock
    flock($handle, LOCK_EX);

    // add order to CSV file
    $order = array($_POST["name"], $_POST["item"]);
    fputcsv($handle, $order);
    fclose($handle);

?>

<!DOCTYPE html PUBLIC
     "-//W3C//DTD XHTML 1.0 Transitional//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Lunch</title>
  </head>
  <body>
    One <?= $_POST["item"] ?> for <?= $_POST["name"] ?>, coming right up!
  </body>
</html>
```

Ultimately, the only thing we're outputting is the XHTML at the bottom, beginning with the DTD. Hopefully, though, the data we're outputting has also been stored persistently somewhere. As you can see, we're opening and writing to a file called `orders.csv` which accomplishes exactly that.

- The first bit of code is some error-checking. If no order has been submitted, we redirect the user back to to `lunch.php`.
- How do we write to this file? First, we make a call to `fopen()` and pass it the option `"a"` in order to use *append* mode so that the file is not clobbered each time we write to it. If the file does not exist, `fopen()` will attempt to create it.
- Notice we're also using file locking, which solves the problem posed by race conditions. If two users submit an order nearly simultaneously, it's

unclear whose data will be written first or whether both orders will be written at all. Basically, one user will have exclusive privileges for writing to `orders.csv` until he finishes at which time the other user will have exclusive write privileges.

- The beauty of PHP is apparent in the call to function `fputcsv()`. As you can see, we don't have to worry about formatting the CSV output since the function auto-magically does this for us. The reverse process is accomplished by `fgetcsv()`. The function `fgetcsv()` takes an optional argument to specify the delimiter, so if you wanted to write a Unix passwords file, you could specify a colon rather than a comma.
- Finally, we free up resources by calling `fclose()` to release the file handle.
- So what if we want to use XML instead of CSV? Take a look at `xml.php` to find out how we might do this:

```
<?

    // ensure complete form was submitted
    if (!isset($_POST["name"]) || !isset($_POST["item"]))
    {
        header("Location: http://www.cs75.net/lectures/4/src/lunch/lunch.php");
        exit;
    }

    // open XML file for reading + writing
    $handle = fopen("orders.xml", "r+");

    // acquire exclusive lock
    flock($handle, LOCK_EX);

    // read contents of XML file
    $contents = fread($handle, filesize("orders.xml"));

    // build DOM out of contents
    $xml = new SimpleXMLElement($contents);

    // add order
    $order = $xml->addChild("order");
    $order->addChild("name", $_POST["name"]);
    $order->addChild("item", $_POST["item"]);

    // overwrite original XML file
    rewind($handle);
    fwrite($handle, $xml->asXML());
    fclose($handle);
```

```
?>

<!DOCTYPE html PUBLIC
     "-//W3C//DTD XHTML 1.0 Transitional//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Lunch</title>
  </head>
  <body>
    One <?= $_POST["item"] ?> for <?= $_POST["name"] ?>, coming right up!
  </body>
</html>
```

Can you spot the differences between this and `csv.php`? First, we're opening a file in `"r+"` mode, which is for both reading and writing. As before, we're using a file lock to prevent race conditions. Next, we're reading in all of the contents of the XML file we just opened using a call to `fread()` and ultimately creating a `SimpleXMLElement` object from that data.

- A quick, but important sidebar: why are we using `fread()` instead of `file_get_contents()`? As it turns out, the latter is *too* efficient for our purposes. It opens the file, reads in the data, and then closes the file, which poses a problem when we're trying to update it. What happens, for example, if the file has changed state between the time we've closed it after reading and the time we open it for writing? Bad news. For our purposes, we're better off opening the file, reading in the data and then updating the file *before* we close the file. This prevents any sort of race condition.
- The code for adding orders is pretty straightforward. We simply call the `addChild` method and specify the name of the child we're adding and optionally, that child's value.
- Why the call to `rewind()`? Well, if we're completely rewriting the entire file after adding the children we wanted, we're going to have to reset the file pointer to the beginning of the file. This is what `rewind()` does.
- Check for yourself that `orders.xml` is updated when you submit an order.

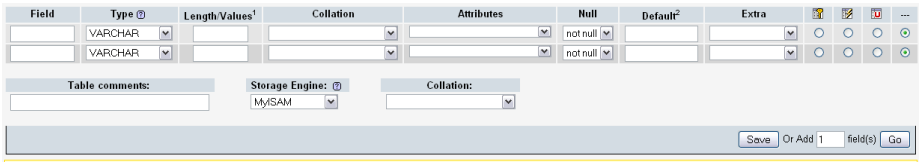
6 Finally, SQL (65:00–105:00)

- As we've discussed, there are problems with implementing databases in XML and CSV. One of the major ones is this necessity for file locking. Although XML and CSV might work perfectly fine for a small set of users

who will only rarely collide with each other in writing persistent data to a file, this doesn't scale very well. What happens when you have hundreds of thousands of users who all need to write persistent data? Are you going to make them each wait for all other users to finish writing to the file? Not likely.

- What you'll want is some system which implements row-level locking, not file locking. This is precisely what database engines such as MySQL achieve. Using MySQL (and other database engines like SQL Server, Oracle, and PostgreSQL), multiple users can write simultaneously to a database without colliding with each other and without clobbering each other's data.
- The world of relational databases essentially boils down to tables. If you've ever used Microsoft Excel or a spreadsheet at all, you have all the understanding you need to begin working with relational databases.
- Those of you who downloaded and installed XAMPP already have an instance of MySQL up and running on your own machine. By default, SQL databases use port 3306 so that they are accessible on servers which also operate as web servers and mail servers.
- At the end of the day, all data in SQL databases is stored in local files. The advantage of SQL databases over XML and CSV databases, however, is that when the SQL database engine is first started, most or all of the data is loaded into virtual memory. In this way, data is much more readily accessible.
- One of the ways we'll be interacting with SQL databases in this course is via phpMyAdmin, a free, open-source GUI. Conveniently, phpMyAdmin, which is packaged with XAMPP, allows you to create, delete, modify, and view tables without having to navigate the more arcane command-line interface.
- Besides the convenient GUI which phpMyAdmin provides, how do we interact with our MySQL database? Using a whole, new programming language, of course! Luckily, the learning curve for SQL is not very steep at all. If it helps, consider the following analogy: XPath is to XML as SQL is to a relational database.
- The definitive source for MySQL help is the online manual.
- DirectAdmin allows you to create MySQL databases and associate users with them via the online interface. Once you've done that, you can login to phpMyAdmin as the user you created here. At that point, all of the databases which that user has access to will be visible.
- Using phpMyAdmin, we can create tables on the database. If we wanted to implement a table for orders, for example, we might create a table

named `orders` and specify that it will contain 2 fields. We'll then be brought to a screen that looks like the following:



For the names of the two fields, we'll go with `name` and `item` as you might expect. Under type, we're going to choose `VARCHAR`. What does this mean exactly? Well, SQL has a number of different data types. Generally, we'll want to choose one that best fits the data that we're storing so as to optimize storage and lookup. In this case, the name and item fields are both strings of indeterminate length. What we can do with SQL is specify a *maximum* length for the field. The `VARCHAR` data type will write only the minimum number of bytes needed to store that string, up to the maximum, and no more. Thus, we use the bare minimum of disk space necessary. For the `name` field, we'll specify 255, which is a convention.

- What might be the advantage of using the `CHAR` data type as opposed to `VARCHAR`? As you might've guessed, `VARCHAR` optimizes for space but `CHAR` optimizes for lookup time. If we know exactly how long each string is when it's stored, we don't have to search for the ends of strings when we're looking up some piece of data.

- The collation option you needn't worry too much about. Basically, think of it as the character set. By default, this will actually be Swedish for MySQL (after the developers), but it will include all of the characters you're familiar with. When in doubt, leave collation blank.

- As for attributes, you can infer from its possible values that it doesn't have much relevance to strings. If we were using integer data types, however, the `UNSIGNED` attribute becomes important. Again, we'll leave this blank.

- Next is the null characteristic. Here we can specify that a field can never be written as `NULL`. This is useful as a second layer of error-checking to prevent empty names from being written to your database, for example.

- Default allows for a default value, as you might have guessed. Extra has only one option, `auto_increment`, which we'll talk about next week. This allows for automatic unique identifier generation. Finally, we have some extra constraints, including indices and primary keys which we can specify for fields.

- The final option worth noting is the storage engine. Analogous to the world of file systems, wherein you have multiple options, e.g. FAT32 and

NTFS, the world of database engines also has multiple options, e.g. MyISAM and InnoDB. Whereas the default MyISAM is arguably the fastest, it doesn't implement *transactions*, which we'll find useful in the weeks to come. MyISAM is not very efficient for large datasets since it requires you to lock an entire table while writing to it.

- Why not simply specify an egregiously large value for `VARCHAR` length? Without going into more specifics, just trust us that there are performance implications for this and it's not advisable.

- The beauty of phpMyAdmin is that when you interact with your MySQL server in any way, it will show you the SQL query that is executed. You would do well to study this SQL code each time you execute a query so that you can learn SQL along the way!

- Know that phpMyAdmin takes the precaution of adding backtick quotes ` before and after every field and table name. This allows you to use protected words like `primary` as field and table names so long as you properly escape them. However, adding these backtick quotes is not explicitly necessary, so you may find David writing SQL queries without them.

- So what's in our table? If we click the Browse tab, we see that there's nothing there. The Insert table provides us with a convenient way of quickly adding rows to our table. Try adding "David" under `name` and "Chicken Salad" under `item`. When we click Browse now, we can see the row we added alongside several options, including edit and delete.

- Before we delve into the finer details of how David implemented the actual database for Rebecca's Cafe (which he actually did using SQLite), play around with the production version of the ordering system. You'll notice first the much-improved layout and color scheme as well as several features such as auto-complete which David implemented using JavaScript.

- As we mentioned, David implemented the back end of his ordering system using SQLite. What he decided that MySQL was too heavy-handed for such a small-scale use, but that SQL itself provided a very convenient way of accessing the data. The SQL query he eventually used was as follows:

  ```
  SELECT DISTINCT name, item, requests FROM orders ORDER BY date ASC
  ```

  As you can see, SQL is quite easy to read. This query simply says, "give me all the unique names as well as the items and requests from the `orders` table, arranged by ascending date."

- When you execute a SQL query in PHP, you'll get an associative array returned to you. Each of the elements of this array corresponds to one of the rows in your table. Thus, we can use the `foreach()` construct.

- Know that David is using the PDO library which enables him to use the try-catch syntax that might be familiar to you from Java, C++, C#, JavaScript, etc.
- SQL queries by convention have protected words like `SELECT` and `FROM` in all-caps in order to help visually parse them. However, SQL queries are case-insensitive.[6] More specific queries can be accomplished using the `WHERE` clause.
- Let's take a look at `sqlite.php`:

```
<?

    // ensure complete form was submitted
    if (!isset($_POST["name"]) || !isset($_POST["item"]))
    {
        header("Location: http://www.cs75.net/lectures/4/src/lunch/lunch.php");
        exit;
    }

    try
    {
        // open database
        $dbh = new PDO("sqlite:orders.db");
        $dbh->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

        // prepare fields
        $name = $dbh->quote($_POST["name"]);
        $item = $dbh->quote($_POST["item"]);

        // insert order
        $dbh->exec("INSERT INTO orders (name, item) VALUES($name, $item)");
    }
    catch (PDOException $e)
    {
        die($e->getMessage());
    }

?>

<!DOCTYPE html PUBLIC
     "-//W3C//DTD XHTML 1.0 Transitional//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
```

[6] Assuming that the `lowercase_table_names` variable is set to 1 or 2.

```
  <head>
    <title>Lunch</title>
  </head>
  <body>
    One <?= $_POST["item"] ?> for <?= $_POST["name"] ?>, coming right up!
  </body>
</html>
```

The first few lines of code allow us to use the PDO library. They're basically copied and pasted from the manual for PDO.

- What does the `quote()` method do? In the PDO library, this actually prevents something called a SQL injection attack, which you can read more about here. Err, sorry, here.
- If we access the SQLite command line, we can create a table just as we did using the phpMyAdmin interface for MySQL. We might execute the following command, for example:

  ```
  CREATE TABLE orders (name VARCHAR(255) NOT NULL, item VARCHAR(255) NOT NULL);
  ```

  We can type `.dump` to see this command printed out once we've executed it.
- Next time, we'll take a look at indexes and constraints. These allow us to find data more quickly because the database engine can optimally sort the data if it knows how we're going to be searching for it.