CMSC 23500 — Introduction to Database Systems
(Originally created by Borja Sotomayor)

May 11, 2010

In your next homework, you will write a database-driven web application using PHP. In preparation for that assignment, this document will provide an introduction to running PHP scripts on CS machines. PHP is a scripting language with a syntax similar to C and Perl so, for the purposes of this session, you should be able to understand the code examples without knowing the full PHP syntax.

1 Hello, world!

Let's start off with a simple "Hello, world!" script. Create a file called `hello.php` with the following contents:

```
<?php
  $msg = "Hello, world!\n";
  print $msg;
?>
```

You can run this script in a CS machine like this:

```
$ php-cgi hello.php
```

You should see the following output:

```
X-Powered-By: PHP/4.4.4-8
Content-type: text/html

Hello, world!
```

If you've already used scripting languages (such as Perl, Python, Ruby, etc.) a couple things above will probably look odd. First of all, notice how the actual code is delimited by `<?php` and `?>`. This will make sense soon, and is due to PHP being mostly used to program web applications. Next, notice how PHP prepends a couple lines to the actual output of the script. These lines are *HTTP headers*. You are not required to know what they are, but they are necessary to use PHP for web application programming. Also note that, if you install PHP on your personal machine, the command you will have to use is `php`, not `php-cgi` (this is an idiosyncrasy of how PHP is configured in the CS machines).

Since PHP is mainly used for web programming, we don't want to run PHP scripts from the command-line, although this can be useful for small tests (specially if you are getting acquainted with the language). Instead, we want to use PHP to dynamically generate web pages accessed through the HTTP protocol. In other words, we want to be able to type in an address such as `http://www.cs.uchicago.edu/~foo/hello.php` and have the web server return the result of running `hello.php`. To do this, you must first activate your CGI service.

2 Activating and testing CGI service for your account

Go to the following URL: `https://tools.cs.uchicago.edu/activate_cgi_service/`. Type in your CS username and password, select "CMSC CGI" as the Instructional Use, and specify "CMSC 23500" as the Course Number. After clicking "I agree", your CGI service will be automagically activated.

Activating your CGI service creates the following directory:

```
/stage/cgi-cmsc/username/public_html/
```

Any files you place in that directory will be accessible through the following URL:

```
http://cgi-cmsc.cs.uchicago.edu/~username/
```

More importantly, any PHP files you place there will be run through the PHP interpreter before they are served to a web client. Copy your `hello.php` file to your CGI directory, and access the following URL from a web browser:

```
http://cgi-cmsc.cs.uchicago.edu/~username/hello.php
```

The web server will not return the contents of the file verbatim but, rather, will run the PHP script first and then return its output.

3 Mixing HTML and PHP

Although illuminating as a PHP example, the `hello.php` is far from being a "web page". More specifically, our script should generate valid HTML. One possibility would be to do the following:

```
<?php
  print "<html>\n";
  print "<head>\n";
  print "    <title>Simple PHP example</title>\n";
  print "</head>\n";
  print "<body>\n";
  $msg = "Hello, world!";
  print "<p>$msg</p>\n";
  print "</body>\n";
  print "</html>\n";
?>
```

The above is ugly and unpleasant. This is where the PHP delimiters come into play: the PHP interpreter will only run code between the `<?php` and `?>`, and will just output everything else verbatim. So, we could rewrite the above in the following (more readable) form:

```
<?php
  $msg = "Hello, world!";
?>
<html>
<head>
  <title>Simple PHP example</title>
</head>
<body>
<p><?php print $msg ?></p>
</body>
</html>
```

Create the above as `hello2.php` in your CGI directory, and observe the result.

4 Accessing a SQLite database from PHP

We can also use PHP to dynamically generate a web page based on the contents of a database. You will be seeing all the details on this in this week's lectures so, for now, let's make sure that your PHPs can correctly access a SQLite database. The following PHP script will generate a page with the list of all the physicians in the hospital database:

```
<html>
<head>
<title>Hospital physicians</title>
</head>
<body>

<?php
try
{
  $dbconn = new PDO('sqlite:hospital.db');
?>

  <h1>Hospital physicians</h2>

  <ul>
<?php
  foreach($dbconn->query('SELECT * FROM Physician') as $row)
  {
          print "<li>" . $row["Name"] . "</li>";
  }
?>
  </ul>

<?php
}
catch(PDOException $e)
{
  print "Error connecting to database!";
}
?>
</body>
</html>
```

Save the above as `hospital.php` in your CGI directory. For it to run correctly, you have to copy the hospital database to your CGI directory too. Browse to `http://cgi-cmsc.cs.uchicago.edu/~username/hospital.php` and observe the result. What happens if you rename the `hospital.db` file to `hospital2.db`? Can you see why? Notice how PHP and HTML can be intertwined in such a way that HTML code is placed between the curly brace delimiters of the `try-catch` statement.

You can practice by trying the following exercises:

1. Modify the script so it will also print the physician's position.
2. Modify the script so, for each physician, it will also print the names of the patients that have that physician as a PCP. Note: If you use a JOIN, you're taking the complicated road.
3. Create a PHP called `appts.php` that prints out all appointments, including the physician's name, patient's name, and nurse's name. If you are familiar with HTML tables, use them.

5 Caveats

As many of you have already discovered, the software environment in the CS machines is... peculiar. The `cgi-cmsc` server is no exception, and you should take some caveats into account while writing your

code.

First of all, running INSERTs from your PHPs requires a bit of extra effort. PHPs with a .php extension are run by Apache (using the PHP module) using the special `www-data` UNIX user, which means that the database has to be writeable by that user for INSERTs to be successful. There are two workarounds for this:

1. `chmod` the database to be world-writeable and put it in a world-writeable directory.
2. Change the extension of your PHP to .cgi and add the following at the top:

```
#!/usr/bin/php-cgi
```

   You will also need to `chmod` the .cgi file to be `744`. This means that your PHP will be run as a CGI (`http://en.wikipedia.org/wiki/Common_Gateway_Interface`) with your UNIX account, meaning that it will be enough for your database to be writeable by your UNIX user.

Workaround 1 is definitely A Bad Thing^{TM} which you should never do in practice. Workaround 2 is not ideal, but it is the least evil of the two.

Next, take into account that PDO code will (by default) fail "silently". To make PDO as cranky and curmudgeonly as possible (and complain of every possible problem), add the following line after you create the db connection:

```
$dbconn->setAttribute( PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION );
```

That way, any error will raise an exception. If you catch the exception, you can print db-specific error information using the errorCode() and errorInfo() PDO methods. More details at `http://us2.php.net/manual/en/pdo.error-handling.php`.

You can also print a stack trace by using the getTrace() or getTraceAsTring() methods of the Exception object. More details at `http://us2.php.net/exceptions`.

Of course, you can also just not use try/catch statement while debugging, and let your PHP b0rk as soon as it encounters an error (the default behaviour is to print a stack trace when this happens).