COMP3311 18s1 Database Systems

Prac Exercise 08
PHP Scripts and Databases

Aims

This exercise aims to give you practice in:

- implementing PHP scripts to manipulate databases

You will find massive amounts of useful information on PHP in the online manual. If there are any PHP constructs used in the samples below that you don't understand, look them up in the manual. There is a index of functions that will be useful to find out what libraries are available and then find out what individual functions do. To find out about core language constructs, consult the language reference section of the online manual.

Background

PHP Scripts

PHP is normally used in scripts that are invoked from a Web server, and operate in a Web server environment, where they access their data via CGI parameters, cookies, and, of course, from a database. However, PHP works perfectly well as a scripting language like Perl, and we can write PHP scripts that run from the command-line.

PHP scripts have the following structure:

```
HTML ...
<?
PHP code ...
?>
more HTML ...
<?
more PHP code ...
?>
...
```

The PHP interpreter handles scripts as follows:

- any text outside `<? ...?>` is copied directly to output without any change
- any text inside `<? ...?>` is executed by the PHP engine; it only produces output via explicit `echo` or `print` statements

Note that if you forget to put in the `<?...?>` (which can also be written as `<?php...?>`), then the PHP engine will simply display your program script without executing it.

A special case is the code fragment

```
<?= Expr ?>
```

is equivalent to

```
<? echo Expr; ?>
```

This is especially useful for inserting computed values into an HTML output stream, e.g.

```
<table> <tr>
<td> x + y </td> <td> <?= $x + $y ?> </td>
</tr> </table>
```

In this exercise, however, we will not be concerned with "mixed" PHP scripts such as the above. We will be dealing with pure PHP scripts

```
<?
PHP code ...
?>
```

If a script such as the above was in a file called `a.php`, it could be executed via the Unix command:

```
$ php a.php
```

Note that there are several PHP engines installed on the CSE workstations:

```
/usr/bin/php
/usr/X11/bin/php
/srvr/cs3311psql/lib/php535/bin/php
```

Only the last of these supports the PostgreSQL libraries, and so you should make sure that you are using this one. The command:

```
$ which php
```

will tell you which one you'll get if you run `php` from the command line.

And, of course, you need to be logged in to Grieg and running your virtual server to make everything work correctly.

To get started, create a file called `eko.php` containing the code:

```
<?
for ($i = 1; $i < $argc; $i++)
    echo "$argv[$i] ";
echo "\n";
?>
```

Now execute the command:

```
$ php eko.php this is fun
this is fun
$ php eko.php

$ php eko.php  just  like  the  echo  command
just like the echo command
```

The script behaves like the Unix `echo` command and displays its command line arguments on a single line, separated from each other by a single space. Note that this version actually prints a trailing space at the end of the line as well. Note also that it skips `$argv[0]`, which contains the name of the script (change the lower bound of the loop to 0 in your script to see what happens).

The online PHP manual has information about control structures and command-line execution. These manual pages are long; read them selectively, not completely.

You might be wondering about the PHP `print` operation, and how it's different to the `echo` operation that we've been using. The only difference is that `print` takes a single argument, which it converts to a string representation, and then writes to the standard output stream, while `echo` takes a comma-separated list of arguments, converts each to a string, and then writes them to the standard output. The following shows examples of how to display a mixture of expressions and string constants using the two different approaches:

```
$a = 3;
echo "Value of \$a^2 is ", $a*$a, "\n";
print "Value of \$a^2 is " . $a*$a . "\n";
$asq = $a * $a;
echo "Value of \$a^2 is $asq\n";
print "Value of \$a^2 is $asq\n";
```

The reason that we use `echo` is because it's one less character to type :-)

An alternative version of the `echo` clone from above is available in the file

```
/home/cs3311/web/18s1/pracs/08/eko
```

Grab a copy of this file and make sure that it's directly executable via the command:

```
$ chmod +x eko
```

You should then be able to execute the `eko` script simply by typing its name (prefaced by `./` if you don't put "." in your PATH):

```
$ ./eko  Kind  of Obvious
Kind of Obvious
```

Put an empty line between the `#!` line and the `<?` line in the script and seeing what difference that makes to the output. It echoes the empty line before it prints the command line arguments. This is an example of a more general principle of how PHP scripts work; anything not enclosed in `<?...?>` will be written to standard output.

For command-line scripts, the above behaviour is relatively harmless. For web scripts, it creates a big problem; if you leave even a single empty line at the start of a script, this will be sent to the Web server before the HTTP header information, which will completely mess up the output of your script.

Mini-exercise: see if you can use PHP array library functions to turn the script into a "one-liner". Hint: `array_slice` and `join` (also called `implode`) will help.

[A solution is available, but try to find the answer yourself first]

To get a better idea of what's happening with the PHP command-line arguments (if you haven't worked it out from the above examples), take a look at the script:

```
/home/cs3311/web/18s1/pracs/08/args
```

Grab a copy of this, make it executable and run it on some interesting collections of command-line arguments, e.g.

```
./args a b c
./args "John Shepherd" has lots of 'fun!'
./args 10 "green bottles" hanging "on the wall"
./args random rubbish ^#*@*@^&!%^#!@
./args random rubbish '^#*@*@^&!%^#!@'
```

The `args` script makes use of the `print_r()` function, which is an extremely useful way of dumping the contents of a complex data structure for debugging purposes.

PHP Scripts and Databases

The next thing to do is to get databases involved. To do this, we'll be using a local PHP library that handles access to PostgreSQL databases. To use this library, you first need to make a copy of it into your working directory:

```
$ cp /home/cs3311/web/18s1/pracs/08/db.php .
```

You don't need to read the `db.php` file in detail at this stage (although you might learn a bit more PHP if you did). What you need to know is that `db.php` provides a number of useful functions for manipulating databases:

$db = dbConnect($conn)

Takes a connection string like "`dbname=myDatabase`" and makes a connection to that database; if successful it returns a database access handle for the named DB. This handle is used when invoking subsequent operations on the database. The function assumes that the user has direct DB access, i.e. could access the database via the command `psql DB`. Note that if the database is not accessible, the `dbConnect` function exits the script with an error message; if a value is returned at all, it is guaranteed to be a valid database handle.

$results = dbQuery($db,$query)

Executes the specified query on the specified database and returns a handle to the result set. The handle can be used (via `dbNext`) to iterate sequentially through the tuples in the result set. If the query is syntactically incorrect, a debugging trace will be written to the standard output.

$tuple = dbNext($result)

Takes a result set handle and returns the next tuple from the result set. The tuple is made available as an associative array where the array indexes correspond to the names of the columns in the result set. E.g.

```
$qry = "select p.unswid, count(e.course)".
       " from People p, Students s, CourseEnrolments e".
       " where p.id = s.id and s.id = e.student  group by p.unswid";
$res = dbQuery($db,$qry);
while ($tup = dbNext($res)) { ... }
results in tuples like ...
$tup: array("unswid"=>SID_value, "count"=>course_count)
```

dbNResults($result)

Returns the number of tuples available in a result set derived from an SQL `select` statement.

dbNChanges($result)

Returns the number of tuples affected by an SQL update statement (e.g. for `delete`, it would be the number of tuples removed from the table).

dbUpdate($db,$update)

Performs and SQL modification statement and returns how many rows were affected.

$tuple = dbOneTuple($db,$query)

Executes an SQL select statement that is assumed to return one tuple and returns the tuple as an array. E.g.

```
$t = dbOneTuple($db,"select name,age from Emp where id = 12345");
might return a result like
$t: array("name"=>"John Smith","age"=>22)
```

dbOneValue($db,$query)

Executes an SQL select statement that is assumed to return one tuple containing just one attribute, and returns the value of that attribute. E.g.

```
$n = dbOneValue($db,"select name from Emp where id = 12345");
might return a result like
$n: "John Smith"
```

dbNoMatches($db,$query)

Returns true if the result set from the supplied SQL query would contain no tuples.

mkSQL()

Constructs "safe" SQL query strings by taking a query template string and filling in `printf`-like slots in the template with values supplied in subsequent arguments. The function takes a variable number of arguments; the first is always a query templte string, with the following arguments corresponding exactly to the slots in the template. E.g.

```
$id = 3012345; $name = "O'Brien"; $age = 35;
$q1 = mkSQL("select * from R where id = %d",$id);
$q2 = mkSQL("select * from S where name = %s and age > %d",$name,$age);
```

would create the query strings:

```
$q1: "select * from R where id = 12345"
$q2: "select * from S where name = 'O''Brien' and age > 40"
```

Note that you don't need to put quotes around string values; the `%s` slot in the template ensures that this is done for you.

Such a function is useful when building queries from user-supplied data values; it makes "SQL injection attacks" more difficult by preventing users from constructing query strings that do additional operations other than the intended query.

An example of an SQL injection attack:

```
$qry = "select * from R where id = $id";
$res = dbQuery($db,$qry);
if ($res) $tup = dbNext($res);
```

Consider what query is executed for each of the following values of `$id`

```
$id = "12345;";
--> "select * from R where id = 12345"
OR
$id = "12345; delete from R";
--> "select * from R where id = 12345; delete from R"
```

Because the value is simply interpolated directly into the query string, the second example runs the query and then wipes out the table. The `mkSQL` function prevents this from happening by taking *only* the numeric part of interpolated value for `%d` data items. Using `mkSQL` would build the query

```
$qry: "select * from R where id = 12345"
```

for both cases of `$id`.

The Database

This exercise assumes that you have already installed the database for Assignment 2. This is quite simple to do, but you should remove your Assignment 1 database first, otherwise you're likely to exceed your disk quota for your `/srvr/YOU/` directory:

```
$ dropdb a1
$ createdb a2
$ psql a2 -f /home/cs3311/web/18s1/assignments/a2/a2.db
... lots of CREATE TABLE ... ALTER TABLE ... and hopefully no errors ...
```

If you're working on your home machine, download the compressed version to save bandwidth:

```
/home/cs3311/web/18s1/assignments/a2/a2.db.bz2
```

Exercises

1. As a simple example of using the `db.php` library, we have written a small script to take a student ID as a command line argument, check whether the student exists, extract their details from the `a2` database, and display the details on standard output. The script is available in the file

   ```
   /home/cs3311/web/18s1/pracs/08/sinfo
   ```

   Place a copy of this script in your working directory, making sure that the `db.php` script is also located there, and then test it out with some example arguments, e.g.

   ```
   $ ./sinfo
   Usage: sinfo SID
   $ ./sinfo abc
   Usage: sinfo SID
   $ ./sinfo 12345
   Invalid SID: 12345
   $ ./sinfo 3252201
   SID       : 3252201
   Name      : Mr Christopher Kretschmer
   Email     : z3252201@student.unsw.edu.au
   Gender    : Male
   Origin    : Australia
   ```

```
$ ./sinfo ....
```

You can find other student ids by taking a look at the `Students` and `People` tables using `psql` on the `a2` database.

If you get a message that the database is unavailable, check that you are (a) logged into grieg with your PostgreSQL server running, (b) you have your `a2` database available, (c) you are running the correct version of PHP. The following commands will probably give you enough clues to solve most problems:

```
$ hostname
$ which php
$ psql a2
```

Once the script is behaving as expected, take a look at it and make sure that you're clear about how it works. Note that the *only* documentation for the database library is what is written above and the source code itself. The above database library does not appear in the PHP online manual; however, you will find descriptions of the low-level PostgreSQL library (e.g., functions `pg_connect`, `pg_query`, `pg_fetch_array` in the PostgreSQL section of the on-line manual).

> If you want to do this lab using a PHP interpreter on your own machine, you'll need to change the
>
> ```
> #!/srvr/cs3311psql/lib/php525/bin/php
> ```
>
> on the first line of most of the scripts to suit your local setup.

2. The queries in Example #1 returned only a single value and a single tuple respectively. The next example of using the `db.php` library runs a query that returns zero or more tuples, and displays multiple results. It is a variation on the previous script that takes a family name as the first argument and produces a list of details of all students who have exactly that family name. The script is available in the file:

   ```
   /home/cs3311/web/18s1/pracs/08/sfind
   ```

   Place a copy of this script in your working directory, making sure that the `db.php` script is also located there, and then test it out with some example arguments, e.g.

   ```
   $ ./sfind
   Usage: sfind FamilyName
   $ ./sfind 123
   No student matching '123'
   $ ./sfind abc
   No student matching 'abc'
   $ ./sfind Ryan
   php sfind Ryan
   SID       : 3243162
   Name      : Mr Steve Ryan
   Email     : z3243162@student.unsw.edu.au
   Gender    : Male
   Origin    : Australia

   SID       : 3277321
   ```

```
Name      : Ms Leena Ryan
Email     : z3277321@student.unsw.edu.au
Gender    : Female
Origin    : United Kingdom

SID       : 3381207
Name      : Ms Heather Ryan
Email     : z3381207@student.unsw.edu.au
Gender    : Female
Origin    : United Kingdom

SID       : 3323803
Name      : Mr Shaun Ryan
Email     : z3323803@student.unsw.edu.au
Gender    : Male
Origin    : United States

$ ./sfind Wang
many tuples ...
```

Take a look at the `sfind` script, and compare it to the `sinfo` script. Make sure that you understand how both scripts work before proceeding with the rest of this prac Exercise.

The current script is a bit limiting as a search tool, since you need to know the exact spelling and case of the family name. Modify the script (you only need to change the query) so that it does case-insensitive matching, and will accept fragments of the family name. For example, "`sfind1 mita`" would match all of the students who have "mita" occurring anywhere in their family name, as in:

```
$ ./sfind1 mita
SID       : 3209985
Name      : Mr Piraveen Armitage
Email     : z3209985@student.unsw.edu.au
Gender    : Male
Origin    : Australia

SID       : 3403965
Name      : Mr Bhumik Mital
Email     : z3403965@student.unsw.edu.au
Gender    : Male
Origin    : India
```

[A solution is available, but try to find the answer yourself first]

3. Write a PHP script called `ts` that uses the `a2` database to write academic transcripts on the standard output. The script accepts a single command-line argument which is interpreted as a student ID, and writes its output in the following format:

```
$ ./ts
Usage: ts SID
$ ./ts 2222333
Invalid SID: 2222333
$ ./ts 3344687
Transcript for Bree Kordahi (3344687)
```

```
10s1 COMP1917 Computing 1                      61   UF    0
10s1 INFS1603 Business Databases               65   CR    6
10s1 MATH1081 Discrete Mathematics             51   PS    6
10s1 MATH1141 Higher Mathematics 1A            73   CR    6
10s2 ACCT1501 Accounting & Financial Mgt 1A    66   CR    6
10s2 COMP1917 Computing 1                      80   DN    6
10s2 MATH1231 Mathematics 1B                   57   PS    6
10s2 SENG1031 Software Eng Workshop 1          72   CR    6
11s1 COMP1927 Computing 2                      70   CR    6
11s1 COMP2111 System Modelling and Design      73   CR    6
11s1 ECON1101 Microeconomics 1                 58   PS    6
11s1 INFS2603 Business Systems Analysis        69   CR    6
11s1 SENG2010 Software Eng Workshop 2A         79   DN    3
11s2 COMP2911 Eng. Design in Computing         60   PS    6
11s2 COMP3711 Software Project Management      82   DN    6
11s2 MATH2859 Prob, Stats and Information      72   CR    3
11s2 PHYS1160 Introduction to Astronomy        71   CR    6
11s2 SENG2020 Software Eng Workshop 2B         81   DN    3
12s1 COMP2121 Microprocessors & Interfacing    54   PS    6
12s1 INFS2607 Networking and Infrastructure    56   PS    6
12s1 SENG3010 Software Eng Workshop 3A         91   HD    3
12s1 SENG3020 Software Eng Workshop 3B         91   HD    3
12s1 SENG4921 Professional Issues and Ethics   70   CR    6
12s2 COMP3171 Object-Oriented Programming      50   PS    6
12s2 COMP9321 Web Applications Engineering     81   DN    6
12s2 EDST4080 Special Education                87   HD    6

$ ./ts 3210516
Transcript for Susan Taualai (3210516)

08s1 EDST1101 Educational Psychology 1         63   PS    6
08s1 MGMT1001 Managing Organisations&People    57   PS    6
08s1 POLS1018 Politics, Power, Principle       75   DN    6
08s1 SPAN1001 Introductory Spanish 1A          43   FL    0
08s2 AUST1001 Australia                        51   PS    6
08s2 MGMT1002 Manag. Organisat. Behaviour      66   CR    6
08s2 POLS1002 Power & Democracy in Australia   57   PS    6
08s2 WOMS1001 Introduction to Feminism         58   PS    6
09s1 MGMT1101 Global Business Environment      65   CR    6
09s1 MGMT2718 Human Resource Management        60   PS    6
09s1 MGMT2725 Career Planning & Management     80   DN    6
09s1 POLS2037 International Law                64   PS    6
09s2 MGMT3701 Legal Aspects of Employment      67   CR    6
09s2 MGMT3728 Managing Pay and Performance     84   DN    6
09s2 POLS2020 Sex, Human Rights & Justice      84   DN    6
09s2 POLS2039 International Organisation       54   PS    6
10x1 ARTS2542 Gods, Heroines and Heroes         69   CR    6
10x1 GENS8200 Workplace Safety and Env Mgmt     81   DN    6
10s1 ARTS1270 The Big Picture: Intro to Hist    66   CR    6
10s1 GENL2031 Cyberspace Law                   85   HD    3
10s1 HUMS3002 Making Histories                 73   CR    6
10s1 POLS3043 US Hegemony & Intern.Law         67   CR    6
10s2 ARTS1190 Australian Legends               56   PS    6
10s2 GENC6002 Marketing and the Consumer       65   CR    3
10s2 MGMT2705 Industrial Relations             86   HD    6
```

```
10s2 POLS3052 Sovereignty, Order & the State    67   CR    6
11x1 ARTS2602 Islam in Asia                     68   CR    6
11x1 HUMS3005 East Asian Values                 65   CR    6
```

Most of the output fields are obvious. The last column, however, shows the number of UC *awarded* to the student for the corresponding course. If they fail, they get zero credit for it, otherwise they get the UC associated with the subject.

A skeleton for this script is available to get you started, and you can find how to get the transcript tuples in the solutions to the Exercises.

Note that you could solve this quite simply using the PLpgSQL `transcript()` function from lectures and small amount of PHP code to send the tuples to standard output. However, for this exercise we would prefer that you did more of the work in PHP, so our SQL template is relatively simple.

Some of the transcripts may look a bit strange. This is a quirk of our data generation code and the NSS data that we have access to. That's life.

[A solution is available, but try to work out the answer yourself first]