CHAPTER 4

User-Facing Software

After slugging through the preliminary information necessary to understand developing PHP in a nonweb context, you're now getting to the nitty-gritty of how to start communicating with your users without the rendering engine of a web browser.

Command-Line Interface Basics

Although graphical interfaces seem to garner the most attention these days, there are still plenty of uses for text-based interfaces, particularly in environments with technically adept users. When creating a text-based program to run on the command line, there are three primary considerations over and above the PHP you are already accustomed to.

- Getting keyboard input
- Outputting text (and graphics) to the screen
- Program flow control

Rather than learn about each one in isolation, you will instead look at a simple program that contains elements of each. Read through the following code and comments. The program is a screen-saver type of routine that fills the shell with color via a wiggling snake-like cursor.

```
<?php

# First we will define some named constants.
# These are shell escape codes, used for formatting
# Defining them as named constants helps to make our code more readable.

define("ESC", "\033");
define("CLEAR", ESC."[2J");
define("HOME", ESC."[0;0f");

# We will output some instructions to the user. Note that we use
# fwrite rather than echo. The aim is to write our output back to the
# shell where the user will see it. fwrite(STDOUT... writes to the
```

R. Aley, *PHP CLI*, DOI 10.1007/978-1-4842-2238-6_4

```
# php://stdout stream. Echo (and print) write to the php://output
# stream. Usually these are both the same thing, but they don't have to
# be. Additionally php://output is subject to the Output control &
# buffering functions (http://www.php.net/manual/en/book.outcontrol.php)
# which may or may not be desirable.

fwrite(STDOUT, "Press Enter To Begin, And Enter Again To End");

# Now we wait for the user to press enter. By default, STDIN is
# a blocking stream, which means that when we try to read from it,
# our script will stop and wait some input. Keyboard input to the shell
# is passed to our script (via fread) when the user presses Enter.

fread(STDIN,1);

# We want the program to run until the user presses enter again. This
# means that we want to periodically check for input with fread, but not
# to pause/block the program if there isn't any input. So we set STDIN to
# be non-blocking.

stream_set_blocking(STDIN, 0);

# In preparation for our output, we want to clear the terminal and draw a
# pretty frame around it. To do this we need to know how big the terminal
# window currently is. There is no in-built way to do this, so we call an
# external shell command called tput, which gives information about the
# current terminal.

$rows = intval(`tput lines`);
$cols = intval(`tput cols`);

# We now write two special escape codes to the terminal, the first
# of which (\033[2J) clears the screen, the second of which (\033[0;0f)
# puts the cursor at the top left of the screen. We've already defined
# these as the constants CLEAR and HOME at the start of the script

fwrite(STDOUT, CLEAR.HOME);

# Now we want to draw a frame around our window. The simplest way to draw
# "graphics" (or "semigraphics") in the terminal is to use box drawing
# characters that are included with most fixed-width fonts used in
# terminals.

# Draw the vertical frames by moving the cursor step-by-step down each
# side. The cursor is moved with the escape code generated by
# ESC."[$rowcount;1f"
```

```
for ($rowcount = 2; $rowcount < $rows; $rowcount++) {
  fwrite(STDOUT, ESC."[$rowcount;1f"."║"); # e.g. \033[7;1f║ for line 7
  fwrite(STDOUT, ESC."[$rowcount;${cols}f"."║");
}

# Now do the same for the horizontal frames.

for ($colcount = 2; $colcount < $cols; $colcount++) {
  fwrite(STDOUT, ESC."[1;${colcount}f"."═");
  fwrite(STDOUT, ESC."[$rows;${colcount}f"."═");
}

# And finally fill in the corners.

fwrite(STDOUT, ESC."[1;1f"."╔");
fwrite(STDOUT, ESC."[1;${cols}f"."╗");
fwrite(STDOUT, ESC."[$rows;1f"."╚");
fwrite(STDOUT, ESC."[$rows;${cols}f"."╝");

# You can see the range of box drawing characters available at
# http://en.wikipedia.org/wiki/Box-drawing_character
# They are just "text" like any other character, so you can easily copy
# and paste them into most editors.

# $p is an array [x,y] that holds the position of our cursor. We will
# initialise it to be the centre of the screen.

$p = ["x"=>intval($cols/2), "y"=>intval($rows/2)];

# Now for our first element of flow control. We need to keep the program
# running until the user provides input. The simplest way to do this is to
# use a never-ending loop using while(1). "1" always evaluates to true, so
# the while loop will never end. When we (or the user) are ready to end
# the program, we can use the "break" construct to step out of the loop
# and continue the remaining script after the end of the loop.

while (1) {

# Each time we go through the loop, we want to check if the user has
# pressed enter while we were in the last loop. Remember that STDIN is
# no longer blocking, so if there is no input the program continues
# immediately. If there is input we use break to leave the while loop.

  if (fread(STDIN,1)) { break; };

# We will step the position of the cursor, stored in $p, by a random
# amount in both the x and y axis. This makes our snake crawl!
```

```
  $p['x'] = $p['x'] + rand(-1,1);
  $p['y'] = $p['y'] + rand(-1,1);

# We check that our snake won't step onto or over the frame, to keep
# it in its box!

  if ($p['x'] > ($cols-1)) { $p['x'] = ($cols-1);};
  if ($p['y'] > ($rows-1)) { $p['y'] = ($rows-1);};
  if ($p['x'] < 2) { $p['x'] = 2;};
  if ($p['y'] < 2) { $p['y'] = 2;};

# We want a pretty trail, so we need to pick random colours for the
# foreground and background colour of our snake, that change at
# each step. Colours in the terminal are set with yet more escape
# codes, from a limited palette, specified by integers.

  $fg_color = rand(30,37);
  $bg_color = rand(40,47);

# Once chosen, we set the colours by outputting the escape codes. This
# doesn't immediately print anything, it just sets the colour of
# whatever else follows.

  fwrite(STDOUT, ESC."[${fg_color}m"); # \033[$32m sets green foreground
  fwrite(STDOUT, ESC."[${bg_color}m"); # \033[$42m sets green background

# Finally we output a segment of snake (another box drawing character)
# at the new location. It will appear with the colours we just set, at
# the location stored in $p

  fwrite(STDOUT, ESC."[${p['y']};${p['x']}f"."╬");

# Before we let the while loop start again, we need to do one more
# very important thing. We need to give your processor a rest.
# If we just continued our loop straight away, you would find your
# processor being hammered, just for our relatively simple program.
# Our snake would also consume the screen at super-speed!
# usleep pauses execution of the program, so others can use the
# processor or the processor can "rest". Every little bit helps the
# responsiveness of your machine, so even if you need your program
# to loop as fast as possible, consider even a small usleep if you can

  usleep(1000);
};

# If this line of code has been reached, it means that we have 'break'd
# from the while loop.
# To be a good citizen of the terminal, we need to clean up the screen
# before we exit. Otherwise, the cursor will remain on which-ever line
# our snake left it, and the background/foreground colours will be
# the last ones chosen for our snake segment.

# The following escape code tells the terminal to use its default colours.

fwrite(STDOUT, ESC."[0m");

# We then clear the screen and put the cursor at the top-left, as we
# did earlier.

fwrite(STDOUT, CLEAR.HOME);
```

This program should demonstrate the three basics listed earlier.

- *Getting keyboard input*: You can read from `STDIN` in the same way you would any stream.
- *Outputting text (and graphics) to the screen*: You can output to `STDOUT` (or use `echo`/`print`), control the appearance and cursor with escape characters, and use block drawing characters to make "semigraphics."
- *Program flow control*: A `while(1)` loop is useful for keeping a program running, with `break` to continue flow outside the loop. It's important to use `usleep` or `sleep` to stop your process from hogging a processor.

Advanced Command-Line Input

The previous section showed how to use `fread()` to read keyboard input. This is suitable for simple programs, but if you are looking to create a more complex interface to allow users to issue commands, then you may want to look at the readline extension, which you can use to implement a shell-like editable command-line program. Unfortunately, for Windows users, the readline library works only under Linux and Unix, and there is nothing comparable for the Windows platform.

The following example script shows how to implement a simple bespoke command-line type interface with the readline library:

```
<?php

# Create arrays to hold our command history and list of valid commands.

$history = array();
$validCommands = array();

# Define some valid commands.
```

```
$validCommands[] = 'kill';
$validCommands[] = 'destroy';
$validCommands[] = 'obliterate';
$validCommands[] = 'history';
$validCommands[] = 'byebye';

# We want to enable tab-completion of commands, which allows the user to
# start typing a command and then press tab to have it completed, as
# happens in Bash shells and the like. We need to provide a function (via
# readline_completion_function) that will provide an array of possible
# functions names. This can be based on the $partial characters the user
# has typed or the point in the program we are at, or any other
# factors we want. In our case, we'll simply provide an array of ALL of
# the valid commands we have.

function tab_complete ($partial) {
  global $validCommands;
  return $validCommands;
};

readline_completion_function('tab_complete');

# We now enter our main program loop. Note that we don't include a usleep,
# as readline pauses our program execution while it waits for input from
# the user.

while (1) {

# We call readline with a string that forms the command prompt. In our
# case we'll put the date & time in there to show that we can change
# it each time it's called. Whatever the user enters is returned. This
# one simple line implements most of the readline magic. At this stage
# the user can take advantage of tab-completion, history (use up/down
# cursor keys) and so on.

  $line = readline(date('H:i:s')." Enter command > ");

# We need to manually add commands to the history. This is used for
# the command history that the user accesses with the up/down cursor
# keys. We could choose to ignore commands (mis-typed ones or
# intermediate input, for example) if we want.

  readline_add_history($line);

# If we want to programmatically retrieve the history, we can use a
# function called readline_list_history(). However, this is only
# available if PHP has been compiled using libreadline. In most cases,
# modern distributions compile it using the compatible libedit library
# for licensing and other reasons. So we will keep a parallel copy of
# the history in an array for programatic access.

$history[] = $line;

# Now we decide what to do with the users input. In real life, we may
# want to trim(), strtolower() and otherwise filter the input first.

  switch ($line) {

      case "kill":
          echo "You don't want to do that.\n";
          break;

      case "destroy":
          echo "That really isn't a good idea.\n";
          break;

      case "obliterate":
          echo "Well, if we really must.\n";
          break;

      case "history":

# We will use the parallel copy of the command history that we
# created earlier to display the command history.

          $counter = 0;

          foreach($history as $command) {
            $counter++;
            echo("$counter: $command\n");
          };

          break;

      case "byebye":

# If it's time to leave, we want to break from both the switch
# statement and the while loop, so we break with a level of 2.

          break 2;

      default :

# Always remember to give feedback in the case of user error.

      echo("Sorry, command ".$line." was not recognised.\n");
  }

};

# If we reached here, the user typed byebye.

echo("Bye bye, come again soon!\n");
```

You may have noticed that I chose to use byebye as the command to quit the program. This was not just a whimsical choice on my part but to illustrate the need to think about discoverability. If you were presented with this program, without seeing the previous source code, and asked to close it, it's likely you would try quit, exit, end, and so on, before resorting to a good old Ctrl-C. In a GUI interface, you would have no such problems when faced with a button that said "Bye Bye!" With text-based input, it is best to stick to common and memorable formats for commands, provide visual guidance and clues where possible, and aid in discoverability with good documentation, a help command, and user training.

Further Reading

- Readline extension in the PHP manual
    - http://www.php.net/manual/en/intro.readline.php

Using STDIN, STOUT, and STDERR

The PHP CLI SAPI automatically opens the standard streams for you when your script starts, so there is no need to issue commands like fopen('php://stdin', 'r'). You can treat these streams just like any other PHP stream and start using them straightaway. You saw some examples earlier, but here are a few more to illustrate the options available:

```
<?php

# Get one line of input from STDIN

echo ('Please Type Something In : >');

$line1 = fgets(STDIN);

echo ('**** Line 1 : '.$line1." ****\n\n");

# Get one line of input, without the newline character

echo ('Please Type Something Else In : >');

$line2 = trim(fgets(STDIN));
```

```
echo ('**** Line 2 : '.$line2." ****\n\n");

# Write an array out to STDOUT in CSV format.
# First, create an array of arrays...

$records[] = array('User', 'Full Name', 'Gender');
$records[] = array('Rob', 'Rob Aley', 'M');
$records[] = array('Ada', 'Augusta Ada King, Countess of Lovelace', 'F');
$records[] = array('Grete', 'Grete Hermann', 'F');

echo ("The following is your Data in CSV format :\n\n");

# ...then convert each array to CSV on the fly as we write it out

foreach ($records as $record) {

  fputcsv(STDOUT, $record);

};

echo ("\n\nEnd of your CSV data\n");

# Pause until the user enters something starting with a number

echo ('Please type one or more numbers : >');

while (! fscanf(STDIN, "%d\n", $your_number) ) {

  echo ("No numbers found :>");

};

echo ("Your number was $your_number\n\n");

# Send the text of a web page to STDOUT

echo ("Press enter for some interwebs :\n\n");

fread(STDIN, 1); # fread blocks until enter pressed

fwrite(STDOUT, strip_tags( file_get_contents('http://www.cam.ac.uk') ) );

# Send an error message to STDERR. You can just fwrite(STDERR,...
# if you want, or you can use the error_log function, which uses the
# defined error handling routine. By default for the CLI SAPI this is
# printing to STDERR.

error_log('System ran out of beer. ABORT. ABORT.', 4);
```

The error logged on the last line will usually appear in your shell along with the other output because that is where most shells put STDERR by default. If you want to check that it did come via STDERR rather than STDOUT, the following bash command will highlight any STDERR output (denoted by the 2>) in red. It uses escape codes to color the error (31 sets the color to red, 07 reverses it, and then 0 clears it).

```
~$ php script.php 2> >(while read errors; do echo -e "\e[07;31m$errors\e
[0m" >&2;  done)
```

In short, you can use the standard streams in any number of ways, often treating them as standard file pointers or streams.

CLI Helper Libraries

There are some prewritten libraries/components available that can take some of the effort out of creating interactive console software. I've listed three common ones in this section. As with most libraries of this type, they are quite "opinionated" in how your program should be structured, so do look thoroughly through the documentation of each before choosing which best suits your project. The Symfony console (part of the Symfony Framework project) is well-tested and stable; as with most Symfony components, it is well-documented and supported. If you are familiar with the Symfony Framework, then the code style should be familiar, but it is equally at home with other frameworks (it is used for the Laravel Framework Artisan Console tool). The Webmozart Console toolkit is a refactored version of the Symfony Console component so has similar features but a different coding style. It is also currently still in Beta. Finally, the Hoa Console is perhaps the most distinct of the three, with a coding style that focuses on real-world tasks and is arguably easier to get to grips with as a developer.

Toolbox	**Symfony Console**
The Console component of the Symfony PHP Framework project	
Main website	https://github.com/symfony/Console

Toolbox	**Webmozart Console**
A (Beta) Console component refactored from the Symfony Console	
Main website	https://github.com/webmozart/console

Toolbox	**The Hoa\Console Library**
A Console library aimed at industrial and research use	
Main website	https://github.com/hoaproject/Console