LAB 15

Error Handling and Validation

What You Will Learn

- How to pre-validate in JavaScript
- How to validate user input in PHP
- Creation and usage of regular expressions
- How to use exceptions

Approximate Time

The exercises in this lab should take approximately 50 minutes to complete.

Fundamentals of Web Development, 2nd Ed

Randy Connolly and Ricardo Hoar

Textbook by Pearson
http://www.funwebdev.com

Date Last Revised: Mar 3, 2017

Error Management

Preparing Directories

1. If you haven't done so already, create a folder in your personal drive for all the labs for this book.
2. From the main `labs` folder (either downloaded from the textbook's web site using the code provided with the textbook or in a common location provided by your instructor), copy the folder titled `lab15` to your course folder created in step one.

This lab walks you through both error and exception handling. You will learn about both client and server side error handling practices.

Exercise 15.1 — Turn on Reporting

1. Open, examine, and test lab15-exercise01.php.
2. Add the following line of code at the top of the file:

```
error_reporting(E_ALL);
```

When you reload the page you may or may not see errors output to the screen. This will be controlled in the next example. Later you can change this value to control what gets output and recorded.

Exercise 15.2 — Display Errors

1. Continue working on your file from Exercise 1.
2. To control how errors are output to the browser, add the following to your file:

```
ini_set('display_errors',1);
```

You should now see output of the errors and warning contained in the original file provided for you. Notice that our script never reaches the final `echo`, since a fatal error stops execution. It should output Notice, Warning and Fatal Error messages.

3. You can change the constant passed to `error_reporting` function from `E_ALL` to `E_ERROR`, `E_WARNING`, `E_NOTICE` or 0. See how the errors displayed change with each value.

Exercise 15.3 - Tail Your Logs

1. Working from the file you created from the last two exercises, we will now examine where errors go when they are not output to the user.
2. Normally the location for your error log files is set up for all the scripts on your domain in php.ini and can be determined by running `phpinfo()`. You should familiarize yourself with that location since often you cannot override the place to output error and access files. In our case files are placed in /var/log/apache2/ and are named error_log and

access_log.

3 To force our script to output errors to our working directory add the following to our exercise file.

```
ini_set('error_log', 'demo_errors');
```

You will then need to create the demo_errors file and change permissions so that the apache server can write to it. *(chmod 777 works in a pinch, but should not be used in production).*

Refresh the browser, and the errors will be output to your file, which can be output as using the **cat** command from the command prompt as follows:

```
cat demo_errors
```

And produces output similar to:

```
[02-Mar-2017 13:57:42] PHP Notice:  Undefined variable: someString in
/var/www/LABS/lab15/done/lab15-exercise01.php on line 9
[02-Mar-2017 13:57:42] PHP Warning:  file_get_contents(asdsasdsa.txt):
failed to open stream: No such file or directory in
/var/www/LABS/lab15/done/lab15-exercise01.php on line 11
[02-Mar-2017 13:57:42] PHP Fatal error:  Call to undefined function
nonexistant_function() in /var/www/LABS/lab15/done/lab15-exercise01.php on
line 13
```

4 As time goes on any such log file will get very large, and rather than examine the entire thing, you will want to examine just the latest errors. To do this use the **tail** command instead of the **cat** command as follows

```
tail –n 100 demo_errors
```

The value after **–n** tells the command how many lines to print out. This command provides a mechanism for programmers to get feedback that does not get seen by end users in the browser (so long as **display_errors** is turned off). If we have errors in a production server they will be logged for later analysis.

ERROR HANDLING

Exercise 15.4 — Custom Error Handlers

1 Open, examine, and test lab15-exercise04.php. Notice that it contains a form and when posted a simple message displaying the contents of $_POST is output.

2 For this exercise we will create a custom error handler function that will be called whenever a page reaches a fatal error. Paste the code below into lab15-exercise04.php.

```
function my_error_handler($errno, $errstr, $errfile, $errline) {
   // put together a detailed exception message
   $msg = "<p>Custom Handling [$errno] ";
```

```
    $msg .= $errstr. " occurred on line ";
    $msg .= "<strong>" . $errline . "</strong>";
    $msg .= " in the file: ";
    $msg .= "<strong>" . $errfile . "</strong> </p>";

    // if exception serious then stop execution and tell maintenance fib
    if ($exception->getCode() !== E_NOTICE) {
        die("Sorry the system is down for maintenance. Please try again
soon");
    }
}

set_error_handler('my_error_handler');
```

Now generate an error of any type. For example divide by 0.

$x = 100/0;

REGULAR EXPRESSIONS

Exercise 15.5 — GETTING STARTED WITH REGEX

1. Open, examine, and test lab15-exercise05.php.
2. We will write a simple regular expression to validate email address is valid. Write a function named validateEmail as follows:

```
function validateEmail($email) {
    $pattern = '/^[\-0-9a-zA-Z\.\+_]+@[\-0-9a-zA-Z\.\+_]+\.[a-zA-
Z\.]{2,5}$/';
    if ( preg_match($pattern, $email) ) {
      return true;
    }
    return false;
}
```

3. Have a look at the regular expression being used. Note that this regex does not comply with the email standard (RFC5322), but comes close enough for many applications.

   ^[\-0-9a-zA-Z\.\+_]+@[\-0-9a-zA-Z\.\+_]+\.[a-zA-Z\.]{2,5}$

   Describe, in words what the above regular expression matches. *Hint: it begins with:*

   - The string starts
   - Then match any of the characters -,0-9, a-z, A-Z, ., + or _ (one or more times)
   - Then match the @ character ….
   - ….

4 Test this regular expression with valid and invalid emails. When there is an error, your script should highlight the error in the form. To accomplish this write the following function.

```
function echoCssError($name) {
   if ($name=='email' &&  !validateEmail($_POST['email'])) {
      echo "has-error";
   }
}
```

5 Now modify the HTML form to call this function for each element, so it outputs a bootstrap style to indicate error. In addition the previously posted values re-populate the form.

```
<form name='mainForm' id='mainForm' method='post'>
  <div class="form-group <?php echoCssError('name'); ?>">
    <legend>Required Information:</legend>
      <label for="name">Name</label>
      <input name="name" id="name" type="text" class="form-control"
             value="<?php echo $_POST['name']; ?>"/>
  </div>
  <div class='form-group <?php echoCssError('email'); ?>'>
      <label for="email">Email</label>
      <input name="email" id="email" type="text" class="form-control"
             value="<?php echo $_POST['email']; ?>" />
  </div>
  <div class='form-group  <?php echoCssError('phone'); ?>'>
      <label for="phone">Phone</label>
      <input name="phone" id="phone" type="text" class="form-control"
             value="<?php echo $_POST['phone']; ?>"/>
  </div>
  <div class="form-group">
        <legend>Optional Information:</legend>
      <label for="how">How did you hear about us?</label>
      <input name="how" id="how" type="text" class="form-control"
         value="<?php echo $_POST['how']; ?>"/>
  </div>
  <input type="submit" value="Submit Form" class="form-control" />
</form>
```

Now, when a bad email is entered, your form should highlight the field red as shown in Figure 12.2

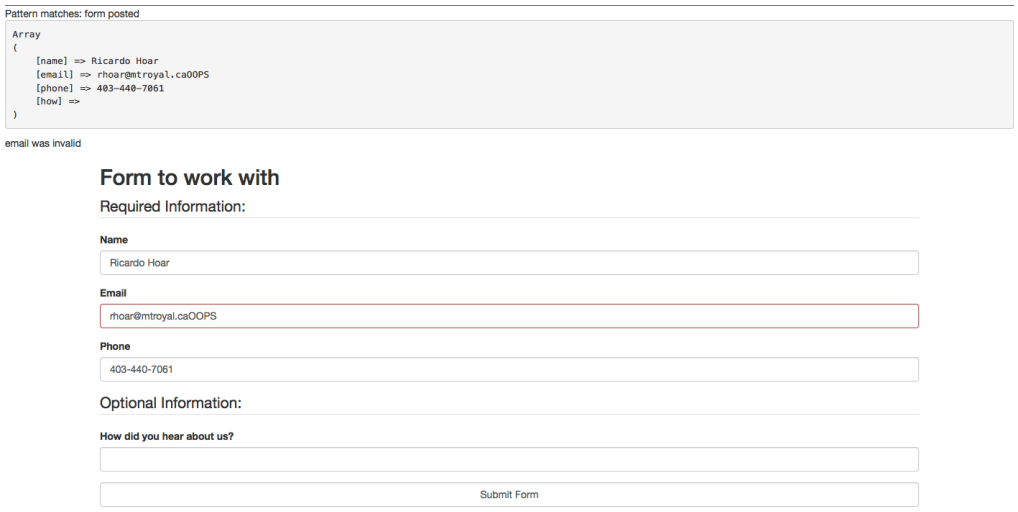

Figure 12.1 – Exercise 15.5 complete

Exercise 15.6 - Advanced Regular Expressions

1 This Exercise will build on the previous one, by adding validation for the phone number and name. We will begin by creating stubs for functions we know we will have to write as follows:

```
function validateName($name) {
   return false;
}
function validatePhone($phone) {
   return false;
}
```

Modify the `echoCssError` function as follows to use our new methods:

```
function echoCssError($name){
   if ($_SERVER['REQUEST_METHOD']=="GET")
      return;  //prevents from highlighting when first loaded.

   if ($name=='email' && !validateEmail($_POST['email'])) {
      echo "has-error";
   }
   if ($name=='name' && !validateName($_POST['name'])) {
      echo "has-error";
   }
   if ($name=='phone' && !validatePhone($_POST['phone'])) {
      echo "has-error";
   }
}
```

Now when you post the page, the name and phone number fields should always be highlighted in error.

2 Now write a regular expression for the name, Consider that the length should be very long and space, dashes, periods, apostrophes and other characters will be required.

The following function uses just such a regex to validate the name.

```
function validateName($name) {
   $pattern = "/^[A-Za-z\.'\s]+$/";
   if ( preg_match($pattern, $name) ) {
     return true;
   }
   return false;
}
```

3 Now write a regular expression to handle the format for north American phone numbers with multiple ways of delineating various aspects. Using the regular expression developed in the chapter you get the function:

```
function validatePhone($phone) {
   $pattern =  "/^\(?\s*\d{3}\s*[\)-\.]?\s*[2-9]\d{2}\s*[-\.]\s*\d{4}$/";
   if ( preg_match($pattern, $phone) ) {
     return true;
   }
   return false;
}
```

5 As a final aspect of error handling, set up a placeholder to help the user see what kind of data you want. For email this will look like:

```
<input name="email" id="email" type="text" class="form-control"
placeholder='example@example.com' value="<?php echo $_POST['email']; ?>" />
```

Set up placeholders for the phone and name as well, and test your form with both valid and invalid input.