Fasthosts Customer Support

An Introduction to PHP Scripting

This guide will introduce some simple yet powerful features of PHP, a popular scripting language, and help you take your first steps towards building a strong web presence.

Contents

Introduction

If you have been following this series of guides from the beginning, you will already be familiar with HTML. HTML is not a scripting language, it's a mark-up language; it simply instructs your web browser what content to display on the page. You can add text, images, and even the elements required to create a form. However, HTML alone is only able to display these elements, if you want to add interactivity to your site, you will need to use a scripting language.

A scripting language allows a web developer to write a series of commands that are executed by the web server that hosts the website. The output from that code is then sent to your web browser. Because all of the code is processed before the page is loaded in the end-user's web browser, these languages are often referred to as “server-side”.

PHP is a popular and extremely flexible scripting language. Because it is free and open source, it is used by many 3rd party applications such as Wordpress, Joomla, and Drupal. Even our very own support sites and control panel use PHP to allow you to search for knowledge base articles, register domain names, add web hosting and services, update your billing details, and so on.

Getting Started

Using a text editor

When writing with a scripting language you will need a text editor. While you can use Notepad, which is supplied with Windows, we highly recommend using an advanced text editor which includes features designed for developers. One of the most important features you should look for in your text editor is colour coding which makes your code much clearer to read, and helps vastly when searching for errors.

If you already have a preferred text editor please use this, otherwise there are many text editors available for free under a General Public License (GNU). Two commonly used text editors are Notepad++ and Crimson Editor. This guide will assume that you have downloaded NotePad++ from http://notepad-plus-plus.org/download.

Creating your page

Your PHP pages should be in text format, and should use the extension **.php**. The name you give to your home page may vary depending on your web hosting provider, but a common name for your default web page is **index.php**.

Using PHP on your page

PHP code must always appear between the opening tag **<?php** and the closing tag **?>**.

```
<?php

?>
```

Comments

When writing any code, particularly PHP code, it is very important to get into the habit of commenting your code. This will help you better understand how your code works if you have to modify it at a later date. There are two ways to add comments in PHP.

- **Single line comments**: These start with two slashes (**//**). The comment only applies to the rest of the line; any code on the following line will be run as normal.

- **Multiple line comments**: These start a slash and then an asterisk (**/***). Unlike the single line comment, everything after the /* will be treated as a comment, even on the following lines of code. You must close this comment block with an asterisk and then a slash (***/**).

```
<?php

//This is a single line comment

/* This is a multiple
line comment. The comment
must be closed with */

?>
```

Displaying text

Displaying text on your page with PHP is important when developing web applications. To do this you can use either of the following commands:

- **echo**: For example echo “Hello World!”;
- **print**: For example print “Hello World”;

```
<?php
      echo “Hello World!";                  // Using echo
      print “Hello World!";                 // Using print
?>
```

The text you want to display should be written within double or single quotation marks. The quotation marks tell PHP that everything inside is a string. This will be explained in the chapter Data types.

Important: Each line of code you write in PHP must end with a semicolon. This tells PHP that the current line of code is finished, and it can move on to the next command. There are exceptions to this rule. For example, a semicolon is not required on the <?php opening tag or after comments. There are other exceptions, which we will cover later in this guide.

Integrating PHP and HTML

PHP code can be placed anywhere on your page, and you can combine HTML and PHP on the same page. When your visitor browses to your website the PHP code will run first, and the results are then sent to your web browser alongside your HTML code.

Copy the following code and save it to a file called **hello.php**. Upload the file to your FTP server with your chosen FTP client and browse to it in your web browser. For example, if your web address is *http://www.yourdomain.com* visit *http://www.yourdomain.com/hello.php*.

```
<!DOCTYPE html>
<html>
      <head>
            <title><?php echo "Hello World!"; ?></title>
      </head>
      <body>
            <?php
                  // Display a heading
```

```
                    echo "<h1>Hello World!</h1>";
              ?>
        </body>
</html>
```

When you view **hello.php** in a web browser you should see a page that displays the text “Hello World!” in a large font.

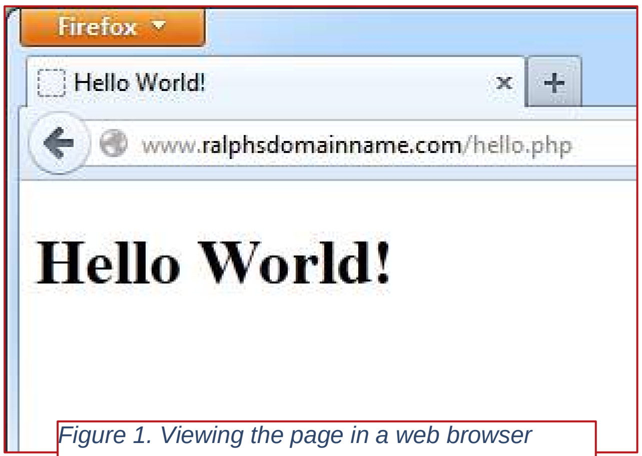

Figure 1. Viewing the page in a web browser

As you can see, you can insert HTML tags into your output. In this case we have printed the string “<h1>Hello World</h1>”. Your web browser sees this as HTML code, and displays the text as a heading.

Quick tip: To improve the readability of your code it is a good idea to indent it in code blocks. As a general rule indent by one tab (or four spaces) for each new block of code, to help you identify that code.
In the example above we have indented the <head></head> and <body></body> tags to help you identify the code associated with those blocks. We have also further indented the <?php   ?> code within the body.

If your code worked, well done – you are now a web developer, and you’ve just built your first web application!

If you saw a blank page or received an error, it is possible that there is an error somewhere in your code. It is worth checking that…

- You have entered a correct opening PHP tag (**<?php**), and the corresponding closing (**?>**) PHP tag.
- The text you want to display with the **echo** command is enclosed in one opening and one closing single or double quotation mark.
- There is a semicolon (**;**) at the end of each line of PHP code, before the closing PHP tag.

If you are still receiving an error message, see **Error! Reference source not found.** on page **Error! Bookmark not defined.**.

Variables

Variables allow you to hold certain information, which you can then manipulate within your code. They work on the same principle as algebra, which allows you to substitute numbers within formulae with letters.

You can simply define a variable in PHP, which can hold any value. Your code can then refer to that variable to retrieve or modify its value. This allows you to write code that can perform different tasks depending on the value of one or more variables.

Variables names always begin with a dollar sign ($). You can give your variables any name you like, but you can only use letters, numbers (though not the first character), or underscores (_).

These are examples of valid variable names	These are examples of bad variable names
• `$a`	• `$5`
• `$b`	• `$my variable`
• `$My_Variable`	• `$1variable`
• `$_variable`	• `$variable#one`
• `$variable1`	• `variable`

Please also remember that the name of a variable is case sensitive. This means you can define a variable called *$MyVariable*, and one called *$myvariable*, and they can both hold different values.

To reduce the risk of confusion when reading your code, it's a good idea to decide on a standard naming convention before you start your project. For example, most PHP developers will only use lower case characters when naming variables in PHP, with underscores to represent spaces (such as *$my_variable*).

Example: Creating a variable

```
<?php
      $x = 1;            // Create a variable called “$x” with a value of 1
      echo $x;           // Display the value of the variable, “1”
      echo ”<br />”      // Print a line break tag

      $x = $x +1;       // Add 1 to the value of $x
      echo $x;          // This should display “2”
```

This example code defines a new variable called "*$x*” with a numeric value of “1”. The code then uses the echo command to print the value of *$x*, followed by another echo command to print the “<br />” tag. This instructs your web browser to insert a line break.

The value for the *$x* variable does not need quotation marks around it, because we are not using text. Instead we are defining a number that we can then manipulate.

Finally we add 1 to the value of *$x*, and again print the value, “2”.

Wherever possible try to give your variables helpful names that will help you instantly recognise which data that variable will hold. This will help you no end when writing, reviewing, or debugging your code.

Arithmetic operators

The following operators allow you to set or modify the value of a variable.

Operator	Description	Example	Value of $z
=	Set the value of a the variable.	$z = 5;	5
.	Concatenation – join two strings together.	$z = "Hello" . " " . "World";	"Hello World"
+	Addition – add the second value to the first.	$x = 5; $y = 10; $z = $x + $y;	15
-	Subtraction – subtract the second value from the first.	$x = 20; $y = 10; $z = $x - $y	10
*****	Multiply – multiply the two values together.	$x = 10; $y = 5; $z = $x * $y;	50
/	Divide – divide the first value by the second.	$x = 50; $y = 5; $z = $x / $y;	10
%	Modulus – the remainder of the first variable divided by the second.	$x = 10; $y = 8; $z = $x % $y;	2

The following example shows the use of the concatenation character to join several strings together to form a word.

```
<?php
        $x = ”H” . “e” . “l” . “l” . “o”;      // Create the variable
        echo ”<p>” . $x . ”</p>”;             // Prints <p>Hello</p>, end
        user will see words in HTML paragraph.
?>
```

This example shows the use of several of the arithmetic operators with an integer.

```
<?php
        $x = 1;       // Create a variable called “$x” with a value of 1
        $x = $x + 9;  // Add 9 to the value of $x to make 10
        $x = $x - 5;  // Subtract 5 to make 5
        $x = $x * 10; // Multiply by 10 to make 50
        $x = $x / 2;  // Divide by 2 to make 25
        echo ”<p>” . $x . ”<p>”; //Display the final number
?>
```

Data types

PHP has several data types which you can use when defining a variable. Here are the most common variable types:

Data Type	Description	Example
String	Text. When defining a string variable you must place the value within single or double quotation marks.	$x = “Hello World”; $y = ‘Hello World’;
Boolean	True or False.	$x = true; $y = false;
Integer	A whole number, not a fraction.	$x = 10;
Float	Floating point number, one that is too large or small to represent as an integer.	$x = 1.23456780; $y = 0.00001; $z = 7E-10;
Null	Represents a variable that has no value.	$x = null;
Arrays	Allows you to create one variable that holds multiple values. You can then access a specific value, or loop through the entire array and process each value in turn.	$x = array(“Value 1”, “Value 2”, “Value 3”);

In some programming languages you must specify the type when you define the variable, but with PHP this is not necessary. Instead PHP will look at the value you have supplied for the variable and set the appropriate data type for you.

Arrays

Arrays allow you to assign multiple values to one variable. Each value in the array has a key, which you can use to retrieve or edit that value. There are three different types of array, numerical, associative, and multidimensional.

Numerical arrays

To define an array, you can use the following syntax.

```
<?php
      // Define an array containing working days
      $var = array("Monday","Tuesday","Wednesday","Thursday",”Friday”);
?>
```

The above example will create a variable, $*var*, to represent an array containing 5 string items. Each item can be referred to by its key, which PHP will create automatically for you. The first item in an array always starts at 0, so the key for “Monday” is 0, the key for “Tuesday” is 1, the key for “Wednesday” is 3, and so on.

Because the keys are numbers, this type of array is known as a numeric array.

To retrieve or edit data in an array, just reference the variable name and then the key within square brackets.

```
<?php
        // Define an array containing working days
        $var = array("Monday","Tuesday","Wednesday","Thursday",”Friday”);

        // Define an array containing working days
        echo $var[2]; // Don’t forget the first item is 0, so the third is 2
?>
```

The above example will print “Wednesday” to the page. You can easily edit existing values by referring to it by its key. If an item with the supplied key doesn’t exist, it is created allowing you to add extra items to the array.

```
<?php
        // Define an array containing working days
        $var = array("Monday","Tuesday","Wednesday","Thursday",”Friday”);

        // Change the first item
        $var[0] = “Beginning of the week”;

        // Add another item
        $var[5] = “Saturday”;
?>
```

Associative arrays

Array keys do not need to be numeric; you can define your own textual keys if you wish, using the combination “key”=>”value”. These are called associative arrays.

```
<?php
      // Define an array containing working days
      $var = array(”mon”=>"Monday", ”tue”=>"Tuesday", ”wed”=>"Wednesday",
      ”thu”=>"Thursday", ”fri”=>”Friday”);

      // Display the value for the key “wed”
      echo $var[”wed”]; // Outputs “Wednesday”
?>
```

The above example will print “Wednesday” to the page. Alternatively, you can declare an empty array and then add each item separately.

```
<?php
      // Define an empty array
      $var = array();

      // Add each day separately
      $var["mon"] = "Monday";
      $var["tue"] = "Tuesday";
      $var["wed"] = "Wednesday";
      $var["thu"] = "Thursday";
      $var["fri"] = "Friday";
?>
```

Multidimensional arrays

These sound more complicated than they are! A multidimensional array is simply an array that contains further arrays. For example:

```php
<?php
      // Define an empty array that will store a five-course menu
      $var = array();

      // Add each course separately
      $menu["appetiser"] = array("Bread","Olives",“Breadsticks”);
      $menu["starter"] = array("Soup","Prawn Cocktail",“Chicken Satay”);
      $menu["main"] = array("Pizza","Cheeseburger",“Lasagne”,”Veg Curry”);
      $menu[“desert"] = array("Cheesecake",”Ice Cream",“Profiteroles”);
      $menu["drinks"] = array("Tea",”Coffee”,“Juice”,”Water",”Wine”);
?>
```

Because you have arrays within arrays, to access a value you will need the key from both, in the format *array[key1][key2]*;

```php
<?php
      // Access the third entry in the main courses (remember that
      numerical arrays start at 0)
      $choice = $menu[“main”][2]; //Lasagne
?>
```

Find the number of entries in an array

You may need to know how many values are contained within your array. This is achieved with PHP's *count()* function:

```
<?php
      // Define an array
      $arr = (“Monday”, “Tuesday”, “Wednesday”, “Thursday”, “Friday”);

      // Use the count() function to display the number of entries in our
      $arr array
      $items = count($arr);
      echo “These are” . $items . “items in this array”;
?>
```

Conditional statements

Conditional statements in PHP are fundamental to building a web application. They let you execute particular code depending on certain conditions.

For example, let's say you have created a web form for your users to contact you. You will need to process the information your visitor has entered, and send yourself the relevant email. However, you need to first ensure that you only run the necessary code when your visitor submits the form, and the data they entered has been checked to ensure it is valid.

Comparison operators

A conditional statement will compare two values using an operator. If the condition is true, the code supplied is executed. You can use the following operators to compare the values.

Operator	Description	Example
==	Equals – return true if the two values are equal.	1 == 1 returns true 1 == “1” returns true 1 == 2 return false
===	Identical – return true if the two values are equal and are of the same data type.	1 === 1 returns true 1 === “1” returns false 1 === 2 returns false
!=	Not equal – returns true if the two values are not equal.	1 != 1 returns false 1 != 2 returns true 1 != “1” returns false
<>	Not equal – returns true if the two values are not equal.	1 <> 1 returns false 1 <> 2 returns true 1 <> “1” returns false
!==	Not identical – returns true if the two values are not equal, or are not of the same data type.	1 !== 2 returns true 1 !== 1 returns false 1 !== “1” returns true
>	Greater than – returns true if the first number is greater than the second.	5 > 4 returns true 5 > 5 returns false 5 > 6 returns false
>=	Greater than or equal – returns true if the first number is greater than or equal to the second.	5 >= 4 returns true 5 >= 5 returns true 5 >= 6 returns false
<	Less than – returns true if the first number is less than the second.	5 < 4 returns false 5 < 5 returns false 5 < 6 returns true
<=	Less than or equal – returns true if the first number is less than or equal to the second.	5 <= 4 returns false 5 <= 5 returns true 5 <= 6 returns true

if()

The most common conditional statement is the *if()* statement. At its most basic it looks like this:

```
<?php
        if(condition){
                // Run code within the { and } brackets if condition is met
        }
?>
```

The *if()* condition always appears in normal brackets. Following the condition are the { and } brackets. The code you would like to execute if the condition is met should be placed within these brackets.

Note: Conditional statements, such as if(), do not require a semicolon to close the line. All code within the { and } group brackets, however, should be written with semicolons as usual.

For example:

```
<?php
      $x = 10;       // Create a variable called $x
      if($x == 10){  // If $x variable is equal to 10 run the code within
      the { and } brackets
            echo $x; // Print the value of $x
      }
?>
```

In this example we have used the “==” operator to compare the value of the variable *$x* with the number 10. If they are equal, the example then prints the value to the page.

Important: When checking if two values are equal, make sure you use ==. The single = sign is an arithmetic operator (see page **Error! Bookmark not defined.**), rather than a comparison operator, and will change the first value to equal the second.
For example, *if($x = 10)* will set the value of *$x* to 10, rather than checking to see if *$x* is equal to 10, and will always return true.

if()… else

There might be an occasion where you wish to run one block of code if a condition is met, and another block of code if that condition is not met. For these cases the *else* statement can be introduced to follow an *if()* statement.

```
<?php
      $x = 15; // Create a variable called $x
      if($x == 10){
            // $x is equal to ten
            echo "Condition met”;
      } else {
            // $x is not equal to ten, run this code instead
            echo "Condition was not met”;
      }
?>
```

As you can see from the example above, the value of *$x*, which is 15, does not meet the criteria specified in the *if()* statement. Therefore, when this code is run it will display “Condition was not met”.

if()… elseif()

You can check several conditions at once using the *elseif()* statement, which must always follow an *if()* statement.

```
<?php
      $x =  15; // Create a variable called $x
      if($x == 10){
            // $x is equal to 10
            echo “Value is 10";
      } elseif($x == 15){
            // $x is equal to 15
            echo “Value is 15";
      }
?>
```

In this example, the *if()* statement checks the value of the *$x* variable to see if it is equal to 10. It isn’t, so the code moves on the the *elseif()* statement which checks to see if the value is equal to 15. It is, and therefore the second group of code is executed, printing “Value is 15” to the page.

You can continue the *if()* statement with as many *elseif()* statements as you wish.

```
<?php
      $x =  15; // Create a variable called $x
      if($x == 10){
            // $x is equal to 10
            echo “Value is 10";
      } elseif($x == 15){
            // $x is equal to 15
            echo “Value is 15";
      } elseif($x == 20){
            // $x is equal to 20
            echo “Value is 20";
      } else {
            // $x has another value
            echo “Value is" . $x;
      }
?>
```

Nested if() statements

You can nest *if()* statements if you wish. This means you can have one or more *if()* statements within another conditional code group.

```
<?php
      $x =  15; // Create a variable called $x
      if($x > 10){
            // $x is greater than 10
            if($x == 15){
                  // $x is 15
                  echo “Value is 15";
            } else {
                  // $x has another value but it’s greater than 10
                  echo “Value is not 15 but is greater than 10";
            }
      } else {
            // $x is less than or equal to 10
            echo “Value is less than or equal to 10";
      }
?>
```

Loops

A loop is a block of code that executes several times until its task has been completed, often with one or more variables that change each time. There are four different types of loop within PHP.

while()

While loops let you loop through a block of code while a certain condition is true.

```
<?php
        // Create a variable containing our start number
        $num = 1;

        // Loop while the value of $num is less than or equal to 10
        while($num <= 10) {
                echo $num “<br />";

                // Increase our number by one
                $num = $num +1;
        }
?>
```

In this example we create a variable called *$num* with a value of 1. We then run the code within the *while()* block as long as the value of *$num* is equal or less than 10. The code inside the block prints the current value on the page, and then increases it by one.

As soon as the value of *$num* reaches 11, the *while()* loop will stop running because the supplied condition is no longer true.

Important: When using a loop, make sure you don't get stuck in a situation where your condition is always true. In our example above, if we didn't increase the value of *$num* by 1 on each iteration our loop would continue indefinitely. While your web hosting server will force your script to stop running after a certain time period, no further code will execute and your visitor will have a bad user experience with your application.

do… while()

The *do… while()* loop is very similar to the *while()* loop. The only difference is that *while()* will only execute if the condition is true, but *do… while()*will always run the code once, and then repeat if the supplied condition is true.

```
<?php
      // Create a variable containing our start number
      $num = 1;

      // Loop while the value of $num is less than or equal to 10
      do {
            echo $num "<br />";

            // Increase our number by one
            $num = $num +1;
      } while($num <= 10);
?>
```

for()

The *for()* loop will run a set number of times. When creating the *for()* loop you need to enter the following in brackets and separated by semi-colons (;)…

- A variable and the starting value. This must be a number.
- The condition that must be true for the code to run.
- The increment by which the variable will change with each iteration of the loop.

```
<?php
      // Loop 5 times
      for ($num = 1; $num <= 10; $num = $num +2) {
            // Display the current number
            echo $num . "<br />";
      }
?>
```

In this example our *for()* loop creates a variable called *$num*, with a starting value of 1. The code within the *for()* loop will only run if *$num* is equal or less than 10, and on each iteration of the loop the value of $num will increase by 2.

foreach()

The *foreach()* loop allows you to cycle through all the values in an array, in the order in which they appear.

```php
<?php
	// Create an array
	$food = array("Pizza", "Fish and chips", "Sausages", "Lasagne");

	// Loop through each item in the array
	foreach($food as $item){
		Echo $item . "<br />";
	}
?>
```

In the *foreach()* statement you supply name of the variable containing the array first, then you specify the variable that will be used to represent the current item in the loop. This must always follow the "as" keyword.

In our example, the value of *$item* will change on each iteration of the array, starting with "Pizza" on the first iteration, then "Fish and chips" in the second, and so on until it reaches the end of the array.

Functions

A function is a portion of code that can be called independently and performs a specific task. Often a function is written when code that performs the same job needs to be run repetitively or accessed from different blocks of code, or even pages.

PHP has hundreds of built in functions that you can use, or you can create your own.

Most functions accept parameters: information required by the function to do its job. Most functions also return a value, such as a Boolean value (true or false) to tell you whether the function ran successfully.

Defining a function

Use the following syntax to create a function…

```
<?php
      function function_name($parameter1, $parameter2, $parameter3) {
            // Create an array
      }
?>
```

Create a custom function to validate an email address

For this example, let's create a function that asks for an email address, checks that it looks valid, and then returns true if it's valid or false if it is not.

Writing a simple function to do this is surprisingly easy, thanks to PHP's built in functions.

In your text editor create a new file called **validate.php** and copy the following code to it.

```
<?php
      // Function to validate an email address
      function validate_email() {

      }
?>
```

We have called our function "validate_email", but you can give it any name you like.

Quick tip: It's good practice to choose a naming convention before you begin coding. As a general rule, function and variable names in PHP use lower case characters with underscores to separate words.

```
<?php
      // Call our function
      validate_email();

      // Define function to validate an email address
      Function validate_email(){

      }
?>
```

The opening and closing brackets after the function name are important; they tell PHP that you are referring to a function. If you try to define or run a function without brackets, an error will occur.

Parameters

When you define a function, you can also define parameters that can be passed into the function. Function parameters are just like variables that are only available to the code within the function.

In the case of our function, we will need to know the email address that we are validating, so we will create a parameter called $*email*.

```
<?php
        // Call our function
        validate_email(“ralph@ralphsdomainname.com”);

        // Funtion to validate an email address
        function validate_email($email){
                echo $email; // Print the value of our $email parameter on the
                page
        }
?>
```

The parameter is defined when we create our *validate_email()* function, placed within the brackets. You can define as many parameters as you would like, separated by commas.

When calling the function we then specify a value for the parameter. When the code within our function is executed, the variable *$email* will have the string value “*ralph@ralphsdomainname.com*”.

If you copy the above code to your **validate.php** script and run it, you should see the email address printed on the page.

Passing variables

You can also pass variables to functions. For example, we could define a variable containing our email address and pass that to the *validate_email()* function.

```
<?php
      // Call our function
      $address = ralph@ralphsdomainname.com;
      validate_email($address);

      // Funtion to validate an email address
      function validate_email($email){
            echo $email; // Print the value of our $email parameter on the
            page
      }
?>
```

Optional parameters

You can also specify that a parameter is optional, i.e. you can supply it when calling the function if necessary but you don’t need to. To identify a variable as optional simply give the variable a default value when you define the function.

```
<?php
	// Call our function
	display_message();

	// Funtion to display a message
	function display_message($message = ”No message supplied, displaying default.”){
		echo $message; // Print the value of our $message parameter on the page
	}
?>

```

In this example we have defined a function to display a message on the screen. When calling the function we can supply the *$message* parameter. However, if we do not, the default will be used resulting on the message “No message supplied, displaying default” being printed to the page.

When defining your function you should always define optional parameters after any compulsory parameters. Take this example of a function where an optional parameter is supplied before a mandatory one.

```
<?php
      function test_function($a = ”Optional”, $b){

      }
?>
```

In *test_function()*, *$b* is a required parameter but *$a* is not. However, because *$b* follows *$a* in the definition, you must always supply *$a*. If you only supplied one parameter when calling this function, it would take the place of *$a*, not *$b*, and therefore generate an error.

Returning a value from a function

Functions have the ability to return a value back to the code that called them. This is achieved with the *return* statement.

In our example our code will validate the email address. If the email address appears to be valid our function will return true, otherwise it will return false. The first step in validating the email address, is to check that something has been supplied to our *$email* parameter.

```
<?php
      // Call our function
      $valid = validate_email(”ralph@ralphsdomainname.com”);

      // Function to validate an email address
      function validate_email($email){
            if($email == ””) {
                  // No email address returned
                  Return false;
            } else {
                  // Email address supplied
                  return true;
            }
      }
?>
```

Our function now contains an *if()* statement that checks the value of the parameter *$email*. If email is a blank string (“”), i.e. nothing has been supplied, the function returns false, otherwise the function returns true.

When we call the function, we now assign the function to a variable called *$valid*. The return value from the function gets assigned to *$valid*, so the example above will return true because we are supplying an email address.

If we ran the same function with the following function call, we would receive a false value from our *validate_email()* function.

```
<?php
      // Call our function
      $valid = validate_email(“”);
?>
```

Completing our example function

Our function is not yet complete. At the moment it simply checks to ensure an email address has been provided, but doesn't check to see if the email address looks valid.

Luckily there is a function within PHP that we can call to check whether our email address is valid or not. It is called *filter_var()* and it essentially takes a variable, and filters it to ensure it is valid.

```
<?php
        // Call our function
        $valid =
        filter_var(ralph@ralphsdomainname.com,FILTER_VALIDATE_EMAIL);
?>
```

This function asks for a string value, and the second parameter supplied states that we are validating an email address. This built in PHP function returns true if the email address is in the correct format, and false if not.

We need to add this built in PHP function to our custom function. Because the *filter_var()* function returns a value, we can use it in our *if()… else* statement directly. We will introduce an *elseif()* statement.

```
<?php
        // Call our function
        $valid = validate_email("ralph@ralphsdomainname.com");

        // Function to validate an email address
        function validate_email($email) {
                if($email == ““){
                        // No email address returned
                        return false;
                } elseif(filter_var($email, FILTER_VALIDATE_EMAIL) == false){
                        // The email address is not in the correct format
                        return false;
                } else {
                        // Email address is valid
                        return true;
                }
        }
?>
```

Now our function checks to ensure a value for *$email* is supplied, and returns false if not. If *$email* is supplied our code moves to the next *if()… elseif()* statement, which runs PHP's built in *filter_var()* function using our *$email* variable. If *filter_var()* returns false the second condition is met, and we return false. If *filter_var()* returns true, our code moves to the else statement and returns true.

At the moment our example script doesn't output anything so we can't see if it's working. Let's add an if() statement when we call our function to display different text on the page, depending on whether the email address is valid or not.

```
<?php
        // Call our function
        $valid = validate_email(“ralph@ralphsdomainname.com”);
        if($valid == true){
                echo “Email address is valid”;
        } else {
                echo “Email address is not valid”;
```

```
	}

	// Function to validate an email address
	function validate_email($email) {
		if($email == ““){
			// No email address returned
			return false;
		} elseif(filter_var($email, FILTER_VALIDATE_EMAIL) == false){
			// The email address is not in the correct format
			return false;
		} else {
			// Email address is valid
			return true;
		}
	}
?>
```

Copy this code to your **validate.php** file, and upload it. Your script should display the text “Email address is valid”. Try playing around with the value supplied for email address, and see what results you get. For example, you could try calling our *validate_email()* function with the following values…

```
<?php
	// Call our function
	$valid = validate_email(“ralph@ralphsdomainname.com”); // Valid
	$valid = validate_email(“ralphsdomainname.com”); // Not Valid
	$valid = validate_email(“ralphsdomainnamecom”); // Not Valid
	$valid = validate_email(“ralph @ ralphsdomainnamecom”); // Not Valid
	$valid = validate_email(“@ralphsdomainname.com”); // Not Valid
?>
```

Creating a web form

HTML allows you to draw elements on a page, including forms which allow your visitors to submit data through your website. Pages such as a contact form, or login form, are useless without a scripting language, such as PHP, to process the information entered by your users.

Let's create a contact form to ask your visitor to enter their email address and a message. We will validate the data and eventually we will write some code to send the message via an email to an address you choose. However, for now let's just take a look at the basics.

We are going to create a form that will ask for the following information:

- The visitors email address, so we can reply to email submitted through the form.
- A subject.
- The message.

To request this information we will need four input controls:

- A text input control, which we will call "email".
- A text input control, which we will call "subject".
- A text area control, which we will call "message".
- A submit button, which the user will click in order to send the message.

In your text editor, create a new page called **contact.html**. Copy the following code into it.

```
<!DOCTYPE html>
<html>
      <head>
            <title>Contact us</title>
      </head>
      <body>
            <form action="contact.php" method="post">
                  <label for="email">Your email address</label><br />
                  <input type="text" name="email" id="email" /><br />

                  <label for="subject">Message subject</label><br />
                  <input type="text" name="subject" id="subject" /><br />

                  <label for="message">Message subject</label><br />
                  <textarea name="message" id="message"></textarea><br />

                  <input type="Submit" value="Send Message" />
            </form>
      </body>
</html>
```

This code on line 7 defines the form. The *action* attribute instructs your web browser to load the **contact.php** page when the form is submitted. We will create this page shortly.

```
7     <form action="contact.php" method="post">
…     …
18    </form>
```

The method attribute tells your web browser how to submit the form. There are two options you can set. For the moment enter “post” as the value. We’ll talk a little more about the method attribute shortly.

We then create the text field that will ask the user to enter their email address. The input control needs a *type* attribute which we have set to “text” to specify that this appears as a text field. We have given it the *name* “email”, which we will use to refer to this control in our PHP code later. We have also supplied an *id*, which for simplicity we have also called “email”.

The *id* attribute is used by the label text. When we create our label we add the *for* attribute, in which we enter the *id* of the control we want the label to point towards.

```
8     <label for="email">Your email address</label><br />
9     <input type="text" name="email" id="email" /><br />
```

We’ve also created another text field called “subject” which will ask the user to enter a subject, and a text area control called “message”. The text area control is very similar to the text input control, but it allows the user to enter text that spans multiple lines.

```
<label for="subject">Message subject</label><br />
<input type="text" name="subject" id="subject" /><br />

<label for="message">Message</label><br />
<textarea name="message" id="message"></textarea><br />
```

Finally we have added another input control, but we have set the *type* attribute to “submit”. This appears in your text browser as a button, and when clicked, will submit the form. The *value* attribute sets the text that will be displayed on the button.

```
<input type="Submit" value="Send Message" />
```

Save the **contact.html** page and upload it to your web server, then visit it in your web browser.

You will see this page. As you can see, it isn’t pretty!

You can add the form to any page on your existing website, and you can use CSS code to style it to fit in with your web presence.

If you click the **Send Message** button you will receive an error, such as “No input file specified”.

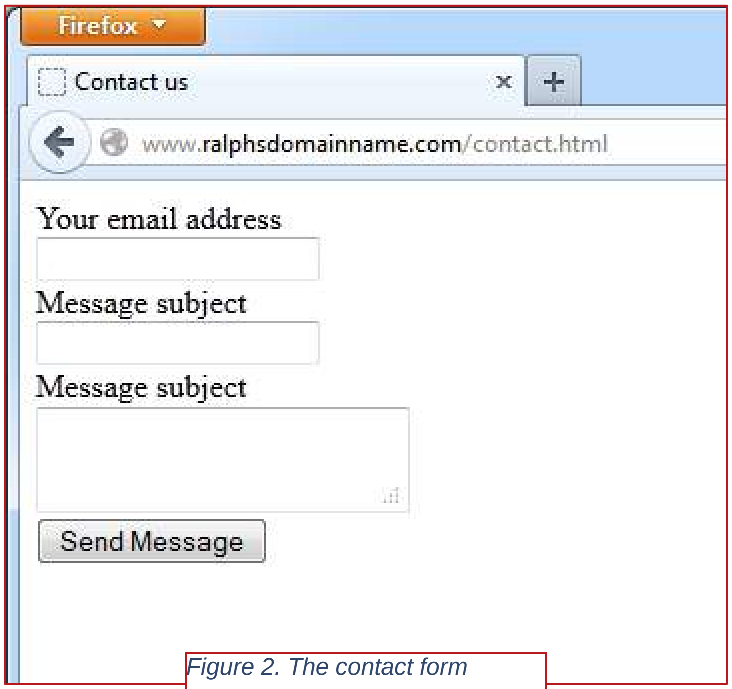

Figure 2. The contact form

This error is displayed because we have told this form to load **contact.php** when submitted, but we haven’t created this file yet.

Using PHP to get user input from a form

HTML is not a scripting language, we can create the form but we can’t do anything with it using pure HTML. In order to process the information entered by the user, we’ll need to use PHP.

PHP provides a very easy way to retrieve the information entered into a form – it assigns the values entered to a variable, called *$_POST*. The values are added as an array, which means each field’s value is assigned to the same *$_POST* variable. To access them, you would use the following syntax *$_POST*[“*field_name*”]. For example:

```php
<?php
      // Retrieve the value of the email field
      $email = $_POST[“email”];
      $subject = $_POST[“subject”];
      $message = $_POST[“message”];

      // $_POST is case sensitive, this will not work
      $fail = $_post[“email”];
?>
```

POST or GET

When you created your form, you specified that the method to use is “post”. You can choose to use one of two available methods, POST or GET.

```
<form action="contact.php" method="post">
…
</form>

<form action="contact.php" method="get">
…
</form>
```

The difference between the two is fairly minor. When submitting a form using GET, which is the default form method, the contents of the form are submitted as part of the URL. As an example, our form would submit to the following URL when using GET:

- *http://www.yourdomain.com/contact.php?email=email@address.com&subject=Subject&message=Message*

There are security implications involved in submitting a form with GET, particularly if you are creating a form that requests personal details or a password. There are also restrictions in the length of a URL that will render very large GET forms unusable. However, there are also advantages. For example, you might want to be able to create a link that submits your form with certain values entered.

If in doubt as to which method to use, you should choose the POST method. With a POST form the entered information is sent with the page request itself, so there is no limitation to the amount of data that can be sent.

Note: When collecting data from a form that was submitted using the GET method, the values are added to a variable called *$_GET* instead of *$_POST*. For example, *$_GET*[“email”] would retrieved the value of an input control named “email”.

Process the form data

Let’s process the information submitted by your contact form. Create a new file in your text editor called **contact.php**. This file will be called when your web form is submitted.

The first thing we need to do is check to make sure the form has been submitted. We can do this easily by establishing if the *$_POST* variable has been created for one of our controls. PHP has a built in function called is *isset()* which we can use to test if a variable has been set.

```
<?php
      // Check to see if our form has been submitted
      if(isset($_POST["email"]) == false){
      // $_POST variable for our "email" control doesn’t exist. The form
      has not been submitted
            header("Location: contact.html");
      } else {
            // Form has been submitted
      }
?>
```

If there is no variable called *$_POST*["email"], our script uses another built in PHP function, *header()*, to send an HTTP header to the web browser. The header we send tells the web browser to redirect the user to the **contact.html** page, containing the form.

You can test to see if this works or not by uploading both your contact.html and **contact.php** pages. If you browse directly to your **contact.php** page, it should direct you automatically back to **contact.html**. If you submit the form, you should see a blank page because we haven't added any code yet to process the data. We'll add that now.

```
<?php
      // Check to see if our form has been submitted
      if(isset($_POST["email"]) == false){
      // $_POST variable for our "email" control doesn't exist. The form
      has not been submitted
            header("Location: contact.html");
      } else {
            // Form has been submitted
            if(validate_email($_POST["email"]) == false) {
                  // Email address is invalid
                  echo "The email address is invalid";
            } elseif($_POST["subject"] == ""){
                  // No subject entered
                  echo  "No subject entered";
            } elseif($_POST["message"] == ""){
                  // No message entered
                  echo  "No message entered";
            } else {
                  // Validation passed
                  echo "Validation passed!";
20            }
21        }
22    ?>
```

We’ve added an *if()* statement to validate our email address, and check to ensure the “subject” and “message” fields hold a value.

However, this code won’t work at the moment. If you submit the form you will receive the following error message…

Fatal error: Call to undefined function validate_email()

We haven’t added our custom *validate_email()* function yet! You are calling this function in the first *if()* statement on line 8, but PHP can’t find this function in your code. Let’s add it underneath our existing code…

```
1     <?php
2         // Check to see if our form has been submitted
3         if(isset($_POST["email"]) == false){
4         // $_POST variable for our "email" control doesn’t exist. The form
          has not been submitted
5             header("Location: contact.html");
6         } else {
7             // Form has been submitted
8             if(validate_email($_POST["email"]) == false) {
9                 // Email address is invalid
10                echo ”The email address is invalid";
11            } elseif($_POST[“subject”] == ""){
12                // No subject entered
13                echo  “No subject entered”;
14            } elseif($_POST[“message”] == ""){
```

```
                  // No message entered
                  echo  “No message entered”;
            } else {
                  // Validation passed
                  echo “Validation passed!”;
            }
      }

      // Function to validate an email address
      function validate_email($email) {
            if($email == “”){
                  // No email address returned
                  return false;
            } elseif(filter_var($email, FILTER_VALIDATE_EMAIL) == false) {
                  // The email address is not in the correct format
                  return false;
            } else {
                  // Email address is valid
                  return true;
            }
      }
?>
```

Upload your **contact.php** file to the web server and try out your web form. If you enter a valid email address and any subject and message, you should see the message “Validation passed!”.

Securing your forms

There are several security implications when you create a web form. Often these implications get overlooked, but it is very important that you consider the following when writing the code that will process your web form.

Never assume the data retrieved from the form is valid.

Unfortunately there is always a chance that a malicious user will attempt to exploit your web form. You should always validate any data from an external source, such as a GET or POST request, before you process it.

Make sure that the data is in a format you are expecting. For example, if you are asking for a numeric value, there are a number of PHP functions to validate a number. For information see **Error! Reference source not found.** on page **Error! Bookmark not defined.**.

If you are expecting text, make sure you only accept characters that you would like to allow. If you are validating an address, for example, you probably only need to accept alphanumeric characters, spaces, and possibly dashes. You don't need special characters such as < > ; and * which can be used within malicious code. A useful built in function to strip all characters from a string, except those which you allow, is *preg_replace()*.

preg_replace()

The *preg_replace()* function uses a regular expression to perform a search and replace on a given string. A regular expression allows you to match characters based on a given pattern.

The following example will string all characters that are not alphanumeric (letters and numbers).

```
<?php
	// Define our string, and then use preg_replace to strip non-alphanumeric characters
	$string = "Strip $ all ^ non-alphanumeric % characters.";
	$string = preg_replace("/[^a-zA-Z0-9]+/", "",$string);
?>
```

Never print data directly onto the page

Unsecure web forms are often used by malicious users to hack a website. A common method is known as cross site scripting, in which JavaScript code is inserted into the form input controls. If your code then prints the contents of the form to the page, that JavaScript code could be executed.

In cross site scripting the JavaScript code can be used to read cookies and steal data, such as personal information about your visitors or even their passwords if you use a login

You should never print data directly from an external source onto the page. Instead there are a number of PHP built in functions that you can use to sanitise and convert the data into a safe format to display on a page.

htmlentities()

This function converts a given string into a harmless string containing HTML entities. For example, the tag “<script>” which tells your web browser that the text that follows is JavaScript, would become “&lt;script&gt;”. Because the function has converted the < and > brackets into html entities, your web browser will not treat it as an HTML tag and will not execute any JavaScript code. The string when displayed on the page, however, will appear as normal.

For example, try copying this code into a PHP file, uploading it to your server, and then visiting the page in your web browser. You should see a page that contains the string "<script>alert('This is a JavaScript alert');</script>".

```
<?php
      // Use htmlentities() to make a string safe
      $x= "<script>alert('This is a JavaScript alert');</script>";
      echo htmlentities($x);
?>
```

Now try removing the *htmlentities()* function, and printing the *$x* variable directly to the page. You will notice that, instead of printing the string on the page, a JavaScript alert box pops up.

Hopefully this demonstrates how important this function can be. If you accept data from a web form, and print it directly to the page without *htmlentities()*, it would be easy for a malicious user to submit their own JavaScript code in your web form. If they can do this, they can perform a number of activities, such as hijacking user’s cookie or session data to steal personal information, or infecting your website with a virus.

strip_tags()

This function does exactly what the name suggests; it strips any HTML tags from a given string.

```
<?php
      // Use strip_tags() to make a string safe
      $x= "<script>alert('This is a JavaScript alert');</script>";
      echo strip_tags($x);
?>
```

This example will remove the “<script>” and closing “</script>” tags from the string. Without these tags, the JavaScript code will not run.

Never send unprocessed data to a database

We haven’t covered connecting to a database yet, but another common method of exploiting a vulnerable script is known as SQL injection. This vulnerability can only exist where a web script uses the data entered into a web form to query a database for information. For example, a search field, login form, or registration form.

If the data entered into the form is not sanitised before it is sent to the database, a malicious user could inject their own SQL code.

This could allow them, in theory, to modify, add, or even delete the data in your database.

Including files

Another extremely useful feature of PHP is the ability to include additional files into your code. Let’s say you have written a function that you need to run on more than one page of your website. Rather than copying the code for each page, you can instead create another PHP file and place your function into that. You can then call that file within each page that needs to access your function.

For example, on our website we might want to ask users to enter their email address to sign up for a newsletter. When they sign up we will need to validate their email address to make sure it’s valid. We’ve already written a function to do this, which is used by our contact form. There is no need to write that function again, we could just move that function into a separate file that both the contact form and the newsletter sign up form could access.

Not only are we streamlining our code by removing unnecessary duplication, we are making life much easier for ourselves if we need to find and fix any bugs within our function, or add extra functionality.

Using your text editor, create a new file called **include.php**. Copy the *validate_email()* function we wrote earlier to it, and upload it to the same folder as your **contact.html** and **contact.php** files.

```
<?php
      // Function to validate an email address
      function validate_email($email) {
            if($email == "") {
                  // No email address returned
                  return false;
            } elseif(filter_var($email, FILTER_VALIDATE_EMAIL) == false) {
                  // The email address is not in the correct format
                  return false;
            } else {
                  // Email address is valid
                  return true;
            }
      }
?>
```

PHP will not allow two functions to be defined with the same name, so open up your **contact.php** file in your web browser and delete the *validate_email()* function code.

Now let’s include our new file. Add the following line to the top of your **contact.php** file.

```
  // Include our additional file
  include "include.php”;
```

Your **contact.php** file should now look like this…

```
  <?php
        // Include our additional file
        include "include.php";

        // Check to see if our form has been submitted
        if(isset($_POST["email"]) == false) {
              // $_POST variable for our "email" control doesn’t exist. The
              form has not been submitted
              header(’Location: contact.html’)
        } else {
              // Form has been submitted
              if(validate_email($_POST["email”]) == false) {
                    // Email address is invalid
                    echo "The email address is invalid";
              } elseif($_POST["subject”] == ""){
                    // No subject entered
                    echo "No subject entered”;
              } elseif($_POST["message”] == ""){
                    // No message entered
                    echo "No message entered”;
              } else {
                    // validation passed
                    echo "Validation passed”;
              }
        }
  ?>
```

Save the file, upload it to your web server, and try it out. If you get any error messages check to ensure that the file you are calling in your *include* call is spelt correctly, and that the **contact.php** and **include.php** files have been uploaded to the same folder on your web server.

Important: If you are hosting your website on a Linux web server, remember that file names are case sensitive. This means if you called your file “include.php”, calling, for example, “Includes.php” or “INCLUDES.PHP” will cause an error.
File names on Windows web servers are not case sensitive, but it’s good practice to use the correct case when referring to files.

Error handling

It is important that you trap any errors that occur in your code, to allow you to track down and resolve the cause. Bugs often creep in to web applications, particularly larger ones, but if you have a robust error handling process, trapping and fixing them shouldn’t be difficult.

PHP allows you to create your own error handling function, which will be called whenever an error occurs. You could use this function to send a report of the error by email, and/or to log the error to a text file, as well as displaying the error on the screen.

To define a custom error handler, use the PHP function *set_error_handler()*. Copy the following code at the top of your **includes.php** file, before your existing *validate_email()* function.

```
<?php
    // Set your own error handler
    set_error_handler("error_handler");
```

```

        // Create the function that will handle the error
        function error_handler($err_number,$err_text,$err_file,$err_line){
              // Perform your error handling here
        }
?>
```

In this example we set the error handler to a custom function that we've called "error_handler".

The custom error handling function can accept four parameters, which we've called *$err_number*, *$err_text*, *$err_file*, and *$err_line*.

- **$err_number** – Each error in PHP has a unique code. The first parameter will contain this code for the error that's just occurred.
- **$err_text** – The second parameter contains the text of the error message.
- **$err_file** – The third parameter contains the name of the file in which the error occurred. This is extremely useful for tracking the code that caused the error.
- **$err_line –** The fourth parameter tells you on which line of your code the error was encountered. Again, this is essential for tracking down the cause of the error.

Quick tip: These parameters are always passed to your custom error handler in this order. However, you can give the parameters any name you would like.

Let’s modify our error handler to display a detailed error message.

```php
<?php
      // Set your own error handler
      set_error_handler("error_handler");

      // Create the function that will handle the error
      function error_handler($err_number,$err_text,$err_file,$err_line){
            // Display this error
            echo "Oops, an error occurred: " . $err_text . "<br />";
            echo "Error code: " . $err_number . "<br />";
            echo "In file: " . $err_file . "<br />";
            echo "On line: " . $err_line . “<br />”;
      }
?>
```

Upload your modified **includes.php** file to your web server.

Testing your error handler

If you want to test your error, handler you can use PHP's *trigger_error()* function. This will trigger a user defined error.

```
<?php
      // Trigger an error
      trigger_error("This is an error message!");
?>
```

Sending an email

Let's create a function to send an email. The built in PHP function *mail()* can be used to send the mail, but before we do we need to apply a little validation.

Our function will have three parameters, *$from*, *$subject*, and *$message*. When calling the function we will supply the email address we are sending the mail from, the subject of the email, and the message itself. We will validate the supplied email address, check that a subject and message have been supplied, and send the mail.

Open the **include.php** file in your text editor, and add the following code below the *validate_email()* function, but before the closing PHP ?> tag.

```
// Function to send an email
function send_email($from, $subject, $message){
      if(validate_email($from) == false) {
            // Sender email address is invalid
            return false;
```

```
        } elseif($subject == "") {
              // No subject supplied
              return false;
        } elseif($message == "") {
              // No message supplied
              Return false;
        } else {
              // Send email
        }
  }
```

As you can see we are defining a function called "send_email" with our three parameters. The *if()* statement validates each parameter in turn and returns false if any of those parameters are invalid or not supplied.

Now we'll create the code to prepare and send the email if the validation passes successfully.

The *mail()* function needs to know what time zone to use when sending the email, so we'll set this first. While your server may have the time zone set by default, it's best to set this manually to be safe especially if you are hosting your website on a shared web server.

We will use the PHP built in function *date_default_timezone_set()* to set the time zone to "Europe/London".

```
        // Set timezone
        date_default_timezone_set ("Europe/London");
```

Next we need to set the email address from which the email will be sent. This must be a valid email address.

Important: If your website is hosted with Fasthosts, either the email address you are sending to or the email address you are sending from must be an existing mailbox on the Fasthosts email platform.

If you are not hosting your website with Fasthosts, your hosting provider may have a similar requirement.

```
        // The email address to send to (don't forget to change this to your own email address)
        $to = "ralph@ralphsdomainname.com";
```

We now need to prepare our email address to send from the email address supplied to the function, which will eventually be the email address your visitor provides when filling out the contact form. We set the email address in the email headers, so we'll create another variable called *$headers* for these.

```
        // Prepare the email headers
        $headers = "From: " . $from . "\r\n";
        $headers = $headers . "Reply-To: " . $from . "\r\n";
```

The "\r\n" characters specify a line break should be used. Because email headers are plain text, and not HTML, the "<br />" tag will not create a line break here.

With our headers prepared, we can send the email. Before we do this, however, we need to use another PHP function called *ini_set()* to set the sender email address configuration setting for your website. This is necessary if sending mail from Fasthosts web servers.

Following the *ini_set()* function is the *mail()* function. We supply the following variables as parameters.

- ***$to*** – The email address we are sending to, which we have just set.
- ***$subject*** – The subject of the message, passed to the function as a parameter.
- ***$message*** – The message body, again passed to the function as a parameter.
- ***$headers*** – The headers we compiled earlier.
- **"-f" . *$from*** – Fasthosts web hosting require you to again supply the senders email address, along with the "-f" flag. If you are not hosting your site with Fasthosts, you may not need this additional parameter.

```
        // Send the email
        ini_set("sendmail_from", $from);
        return mail($to, $subject, $message, $headers,“-f “ . $from);
```

The *mail()* function returns true if the email was accepted for delivery, and false if not. Our function will return the value that the *mail()* function supplies.

That's all there is to it! Your complete *send_email()* function should look like this…

```
// Function to send an email
function send_email($from, $subject, $message){
        if(validate_email($from) == false) {
                // Sender email address is invalid
                return false;
        } elseif($subject == “”){
                // No subject supplied
                return false;
        } elseif($message == “”){
                // No message supplied
                return false;
        } else {
                // Set timezone
                date_default_timezone_set(“Europe/London”);

                // The email address to send to (don’t forget to change this
                to your own email address)
                $to = “ralph@ralphsdomainname.com”;

                // Prepare the email headers
                $headers = "From: " . $from . "\r\n";
                $headers = $headers . "Reply-To: " . $from . "\r\n";

                // Send the email
                ini_set(“sendmail_from”,$from);
                return mail($to, $subject, $message, $headers, "-f" .$from);
        }
}
```

Important: Don’t forget to substitute the “ralph@ralphsdomainname.com” email address with your own!

Using your new function to send an email

Let’s add a call to our new *send_email()* function to our **contact.php** page, so when a visitor submits your contact form you receive an email.

Because our *send_email()* function returns true if successful, and false if not, we are going to call the function inside another *if()* conditional statement. This will allow us to display an error if we can’t send the email and a success message if we can.

This is the code we will add after the validation. It calls our *send_email()* function with the values retrieved from the form input controls called “email”, “subject”, and “message”.

```
// Validation passed
if(send_email($_POST["email"], $_POST["subject"], $_POST["message"])) {
      // Message sent
      echo "Thankyou, your email has been sent";
} else {
      //Error sending email
      echo ” An error occurred whilst sending the email, please try again
      later”;
}
```

Open the **contact.php** file in your text editor, and add the code above to it as shown below.

```
<?php
      // Include our additional file
      include “include.php”;

      // Check to see if our form has been submitted
      if(isset($_POST["email"]) == false) {
            // $_POST variable for our "email" control doesn’t exist. The
            form has not been submitted
            header('Location: contact.html');
      } else {
            // Form has been submitted
            if(validate_email($_POST["email"]) == false) {
                  // Email address is invalid
                  echo "The email address is invalid";
            } elseif($_POST["subject"] == "") {
                  // No subject entered
                  echo "No subject entered";
            } elseif($_POST["message"] == "") {
                  // No message entered
                  echo "No message entered";
            } else {
                  // Validation passed
                  if(send_email($_POST["email"], $_POST["subject"],
                  $_POST["message"])) {
                        // Message sent
                        echo "Thankyou, your email has been sent";
                  } else {
                        // Error sending email
                        echo "An error occurred whilst sending the email,
                        please try again later";
                  }
            }
      }
?>
```

Upload the **contact.php** file to your web server, along with the **include.php** file, and test it. Did you receive the email? If you didn't make sure both the email addresses you are sending to and from exist, and that one of them is hosted with your web hosting provider.

Sending an error report

Now that our *send_email()* function is complete, we can modify our custom error handler to send us a report by email should any error occur. This demonstrates the beauty of functions – rather than having to write additional code to send the email, we can simply call our existing code and put it to a slightly different use.

We'll add the following code to send the error report…

```
// Send an error report
$report = "Error: " . $err_text . "\r\n";
$report = $report . "Error code: " . $err_number . "\r\n";

$report = $report . "In file: " . $err_file . "\r\n";
$report = $report . "On line: " . $err_line;
$success = send_mail("error@ralphsdomainname.com", "Error report",
$report);
if($success == true) {
      echo "This error has been reported to the website administrator.<br
      />";
}
```

This code compiles the report to send, using the *$report* variable. We then call our *send_email()* function, specifying that we are sending the email from “error@ralphsdomainname.com”, with a subject of “Error report”.

Important: Don’t forget to substitute the “error@ralphsdomainname.com” email address with your own!

As a courtesy we inform the user that an error report has been sent to the website administrator if the email is sent successfully.

Let’s put this code into our error handling function. Open the **include.php** file in your text editor and add the above code to your *error_handler()* function.

```
<?php
        // Set your own error handler
        set_error_handler("error_handler");

        // Create the function that will handle the error
        function error_handler($err_number, $err_text, $err_file,$err_line){
                // Display this error
                echo "Oops, an error occurred: " . $err_text . "<br />";
                echo "Error code: " . $err_number . "<br />";
                echo "In file: " . $err_file . "<br />";
                echo "On line: " . $err_line . "<br />";

                // Send an error report
                $report = "Error: " . $err_text . "\r\n";
                $report = $report . "Error code: " . $err_number . "\r\n";
                $report = $report . "In file: " . $err_file . "\r\n";
                $report = $report . "On line: " . $err_line;
                $success = send_mail("error@ralphsdomainname.com","Error report",$report);
                if($success == true) {
                    echo "This error has been reported to the website administrator.<br />";
                }
        }
?>
```

Testing the error handler

Let’s test the error handler to make sure you receive the error report. Add the following code to your **contact.php** file.

Make sure you trigger the error after you have included your **include.php** file, because that’s where you define your custom error handler.

```
<?php
    // Include our additional file
    include "include.php";

    // Trigger an error
    trigger_error("This is an error message");

```

Save both your **contact.php** and **include.php** files, upload them to your web server, and test your form. Don’t forget to remove the *trigger_error()* function call once you’ve finished testing your error handler.

Connecting to a database

One of the most powerful features of PHP is the ability to connect to a database. You can use a database to significantly extend the capabilities of your website. For example, a database makes it rather simple to add or modify website content through a content management system, without the need to modify your site’s source files. You could also implement search features, an account sign up and login system, or a shopping basket.

The database of choice for PHP users is MySQL which, just like PHP, is free and open source software. It is also extremely powerful, and as a result is one of the most popular database solutions available today.

An introduction to SQL

SQL is an acronym for Structured Query Language. It is a programming language specifically designed for querying and managing the data held within a relational database.

With PHP, you simply connect to the MySQL database and send an SQL query to the database server. The MySQL server will then process the query, and send the requested data back for PHP to handle.

SQL queries can be very basic. One of the most common types of query is a SELECT query, which simply returns data from the database. For example, the following query will display all records found in a table called “Products”.

```
SELECT * FROM Products;
```

The asterisk (*) is a shorthand method of telling MySQL to select all fields in the table. If you only needed to retrieve the value of two fields called “ID” and “Name” you could use.

```
SELECT ID, Name FROM Products;
```

Preparing your database

In order to complete this chapter you will need a MySQL database with at least one table. If you do not already have a MySQL database, you can add one in your Fasthosts control panel. If you host your website with another hosting provider, they probably also offer MySQL.

You will need the following details for your database:

- **The database host** – This is the address you use to connect to the database. If the database is hosted on the same server as your website this will be “localhost”, otherwise it will probably be an IP address.
- **The database name** – This is the name you chose when you created the database.
- **The database user** – The username you use to connect to the database.
- **The database password** – The password for the above user.

The script that we will write will retrieve a list of products from a table called “Products” in your database. If this table does not exist, you can either create it, with the following fields, or just substitute the table and field names for your own in the code.

Our “Products” table contains the following fields.

Field name	Field type	Description
ID	INTEGER	The unique ID of each record. This is the primary key for this table.
Name	VARCHAR(20)	String of up to 20 characters containing the name of the product.
Description	VARCHAR(100)	String of up to 100 characters containing a short description of the product.

Connecting to the database

To connect to the database we are going to use the MySQL Improved (MySQLi) extension. This extension is supplied with PHP, and should be enabled for the majority of web hosting providers.

This extension is the recommended method of connecting to a database because it has built upon an earlier MySQL extension to provide enhanced security and stability

Note: The MySQLi extension has been supplied with PHP from version 5.0.0. However, in some installations the extension may not be enabled by default. If you are configuring your own development environment, please see the PHP article on installing the MySQLi extension libraries.

Preparing your PHP code

Let's start writing a script that will connect to the database. Create a new file in your text editor called **database.php**.

First, we will define the database connection details. We do this using a feature of PHP called constants. Constants are similar to variables, except that their value cannot be changed during runtime.

They are defined in your code like this.

```php
<?php
	define("CONSTANT_NAME", "Value");
	echo CONSTANT_NAME;
?>
```

When defining the constant, you can use any name you wish. However, developers usually use upper case names for constants to help them easily identify them in their code.

Note that when defining a constant it is named within a string, but when referring to it from then on no quotation marks are required.

Let’s define four constants that will hold our database connection details.

```php
<?php
define("DB_HOST", "213.171.200.57"); // The host address of the database.
define("DB_NAME", "ralphsdatabase"); // The name of the database.
define("DB_USER", "username"); // The username to connect with.
define("DB_USER", "password"); // The password for the above user.
?>
```

Quick tip: For security it’s a good idea to define these connection details in a separate include file that is located in a private folder that cannot be directly browsed to in a web browser.

Next, we’ll prepare the SQL query that we will be using. It’s good practice to only open the connection to the database for the minimum time necessary to run the query and return the results, so we’ll prepare the query before we open the connection.

All our query will do is return the ID, name, and a description of each product in our “Products” table. The SQL query will look like this.

```sql
SELECT ID, Name, Description FROM Products ORDER BY Name ASC;
```

This query will select the fields “ID”, “Name”, and “Description” from the table called “Products”. It will also order the results in ascending order by the “Name” field. This will result in all products being listed in alphabetical order.

```
<?php
define(" DB_HOST", "213.171.200.57"); // The host address of the database.
define("DB_NAME", "ralphsdatabase"); // The name of the database.
define("DB_USER", "username"); // The username to connect with.
define("DB_USER", "password"); // The password for the above user.

// Prepare our query
$query = "SELECT ID, Name, Description FROM Products ORDER BY Name ASC;";
?>
```

Creating the connection

Now that we have prepared our database and query details, let’s attempt to make the connection. To do this we use the line…

```
// Connect to the database
$mysqli = new mysqli(DB_HOST, DB_USER, DB_PASSWORD, DB_NAME);
```

This code creates a new variable called “mysqli” (you can use any name you like), which contains the database connection. When we initialise the connection we are creating a new object based on the MySQLi extension.

Essentially, an object is similar to an array in that it can contain different values. However, an object can also contain one or more code functions to perform specific tasks. In this case the MySQLi object that we have created and assigned to the *$mysqli* variable contains several functions that we can use to query the database and retrieve the results.

When creating the database object we specify the database host, user, password, and database name. These parameters must be placed in this order.

At the moment we have no idea if the connection to the database has been successful. So let’s find out.

```
// See if the connection was successful
if(mysqli_connect_errno()) {
      // Error connecting to database
      echo "Unable to connect to the database: " . $mysqli-
      >connect_error();
} else {
      // Database connection successful
      echo "Connection successful";

      // Close database connection
      $mysqli->close();
}
```

If there was an error connecting to the database, the *mysqli_connect_errno()* function will return that error. Our code above checks to see if there was an error, and displays the error if there was. If there was no error it prints “Connection successful” to the page.

After this the command *$mysqli->close()* closes the connection to the database.

Important: You must always close database connections as soon as you have finished with them. If you don't, your database server will start to suffer from performance issues as more and more connections open.

In this code we run two functions in the MySQLi object, *connect_error()* and *close()*. Notice how to run these functions you must specify the name of the variable containing the object, followed by a dash and greater than symbol (->), then the function name.

The full code in your **database.php** file should look similar to this…

```php
<?php
    define("DB_HOST","213.171.200.57");//The host address of the database.
    define("DB_NAME", "ralphsdatabase");//The name of the database.
    define("DB_USER", "username");//The username to connect with.
    define("DB_PASSWORD", "password");//The password for the above user.

    //Prepare our query
    $query = "SELECT ID, Name, Description FROM Products ORDER BY Name ASC;";

    //Connect to the database
    $mysqli = new mysqli(DB_HOST, DB_USER, DB_PASSWORD, DB_NAME);

    //See if the connection was successful
    if(mysqli_connect_errno()) {
        //Error connecting to database
```

```
			echo "Unable to connect to the database: " . $mysqli-
			>connect_error();
		} else {
			//Database connection successful
			echo "Connection successful";

			//Close database connection
			$mysqli->close();
		}
	?>
```

Save the **database.php** file and upload it to your web server, it’s time to test it.

Connection Errors

If it worked and you're now looking at a page with the words "Connection successful" then well done! If it didn't work, then there is a good chance you're looking at one of these errors (or one very similar).

Warning: mysqli::mysqli() [mysqli.mysqli]: (28000/1045): Access denied for user '<database user>'@'<database host>' (using password: YES)

Cause	Solution
This is a simple "Access denied" error. The username and/or the password you are using are not correct.	Check that your username and password are spelt correctly. Remember they are case sensitive. If your database is hosted with Fasthosts you can change the password in your control panel.

Warning: mysqli::mysqli() [mysqli.mysqli]: (HY000/2003): Can't connect to MySQL server on '<database host>' (10061)

Cause	Solution
The database host you supplied is incorrect or the database server is unavailable.	Check that the database host is correct. If your database is hosted with Fasthosts this will be an IP address, which is available in your control panel.

Running a query and retrieving results

All we've done so far is successfully connect to the database; we have yet to query and data from it.

To query the data we are going to create a prepared statement. Prepared statements allow optimised performance when querying the database, and minimise the risk of SQL injection where a malicious user is able to insert their own code.

In our current example this isn't a risk, because we are not using data from an external source (such as a form) to create our SQL statement. However, it's good practice to get into the habit of using prepared statements.

Once the connection to the database is established, we need to prepare our statement with the following code.

```
// Create the statement
$stmt = $mysqli->stmt_init();
if(!$stmt->prepare($query)) {
      // Unable to prepare statement, probably an SQL error.
      echo "Unable to prepare the statement: " . $stmt->error;
} else {
      //Statement created
      echo "Statement created";

      //Close statement
      $stmt->close();
}
```

The first line of code uses the MySQLi objects *stmt_init()* function to initiate the prepared statement engine. This creates another object, which we've assigned to a variable called *$stmt*. This object will allow us to prepare and run the statement, and retrieve the results.

The following line then prepares the statement using the query we defined earlier, in the *$query* variable. If there is a problem preparing the query, we display the error message.

If the query is prepared successfully you must remember to close the statement when you have finished using it, just like you must close the database connection.

Executing the statement

At this stage we still haven’t queried the database, but we are now ready to do this, with the following code.

```
//Execute query and store result
$stmt = $mysqli->stmt_init();
if(!$stmt->prepare($query)) {
```

These lines of code are relatively easy to understand. The first, *$stmt->execute()*, simply executes our SQL query.

The second, *$stmt->store_result()*, stores the results from the database which will allow our following code to access those results. Let’s check to see if there were any records returned.

```
//Check if there were any results
if($stmt->num_rows == 0) {
      //No records found
      echo "No records";
} else {
      //There are records
      echo "There are " , $stmt->num_rows . " records<br />";
}
```

The property “num_rows” of our statement contains the number of records that were retrieved from the database. If this is 0, our code displays a message to inform the user that there were no records; otherwise it displays the number of records found.

Binding the results to variables

If records were returned from our query, we are going to need to write some code to retrieve these values. As this is a demonstration, our code is simply going to print each returned record on a new line.

With prepared statements we need to create a new variable for each field that we have asked for in our query. Let’s take another look at the SQL we are using:

```
SELECT ID, Name, Description FROM Products ORDER BY Name ASC;
```

We are asking for the values of the ID, Name, and Description fields. We must bind, or link, each of these three fields to its own variable.

It doesn’t matter what you call these variables. However, to make it easy to understand which variable contains data from which field, we’ll call them *$id*, *$name*, and *$description*.

```
//Bind the results from the database to variables
$stmt->bind_result($id, $name, $description);
```

The *bind_result()* function connects the fields in our query with the variables. It is important that your variables appear in the same order when passed to the *bind_result()* function as they do in your SQL statement.

Looping through the results

Now that our results are bound to our variables, it’s nice and easy to loop through all the results. To do this we’ll use a *while()* loop.

```
// Loop through each record
while($stmt->fetch()) {
      echo $name . " (ID: " . $id . ") - " . $description . "<br />";
}
```

The prepared statement’s *fetch()* function will retrieve the values from the first record returned, and then each time it is called it will return the next record until it reaches the last. When there are no results left, the function will return false.

Our *while()* loop will loop until *fetch()* returns false, i.e. it will loop once for each record returned from our results.

Each time the *fetch()* function is called it will assign the data for each field in the current record to the corresponding variable that we bound earlier. Hence, for each iteration of the loop our *$id*, *$name*, and *$description* variables will contain the relevant details from the current record, which we can print on the page.

The complete code

Here is the code in all its glory! If you haven’t already, copy this to your **database.php** file and upload it to your web hosting server. Don’t forget to update the database connection details with your own.

```php
<?php
    define("DB_HOST","213.171.200.57");//The host address of the database.
    define("DB_NAME", "ralphsdatabase");//The name of the database.
    define("DB_USER", "username");//The username to connect with.
    define("DB_PASSWORD", "password");//The password for the above user.

    //Prepare our query
    $query = "SELECT ID, Name, Description FROM Products ORDER BY Name ASC;";

    //Connect to the database
    $mysqli = new mysqli(DB_HOST, DB_USER, DB_PASSWORD, DB_NAME);

    //See if the connection was successful
    if(mysqli_connect_errno()) {
        // Error connecting to database
        echo "Unable to connect to the database: " . $mysqli->connect_error();
    } else {
        // Create the statement
        $stmt = $mysqli->stmt_init();
        if(!$stmt->prepare($query)) {
            // Unable to prepare statement, probably an SQL error.
            echo "Unable to prepare the statement: " . $stmt->error;
        } else {
            // Execute query and store result
            $stmt->execute();
            $stmt->store_result();

            // Check if there were any results
            if($stmt->num_rows == 0) {
                // No records found
                echo "No records";
            } else {
                // There are records
                echo "There are " . $stmt->num_rows . " records<br />";

                // Bind the results from the database to variables
                $stmt->bind_result($id, $name, $description);

                // Loop through each record
                while($stmt->fetch()) {
                    echo $name . " (ID: " . $id . ") - " . $description . "<br />";
                }
            }

            // Close statement
            $stmt->close();
        }

        // Close database connection
        $mysqli->close();
    }
?>
```

Congratulations, you have now connected to a database and retrieved a series of data! Of course, this is a very simple demonstration. The power of MySQL is far greater than we can cover in this guide.

If you would like to learn more about setting up and querying MySQL databases, as well as building a powerful web application using a database, then take a look at our next guide in this series An Introduction to MySQL Databases.

Appendix A: Common errors

Here is a list of some of the common errors that you might encounter when you start programming with PHP, along with some possible solutions.

No input file specified.

Cause	Solution
This error message is displayed when you browse to a PHP file that doesn't exist.	Check that the URL you are browsing to exists on your web server. If you are hosting your website on a Linux web server, remember that file names are case sensitive, so a file called "index.php" would not be found if you browsed to "Index.php".

Parse error: syntax error, unexpected <statement>, expecting ',' or ';'

Cause	Solution
This error message is most often caused by a missing semi-colon at the end of the line.	Check the line before the line number given in the error message to make sure there is a semi-colon (;) at the end of the text.

Parse error: syntax error, unexpected $end

Cause	Solution
This error message is often caused by an unclosed *if()* condition, loop, or string.	Check that each *if()* conditional statement and loop (*while()*, *do… while()*, *for()*, *foreach()*) has the correct number of opening and closing { } brackets. Check also that every string you define has an opening and closing single or double quotation mark.

Missing argument <n> for <function>()

Cause	Solution
You have defined a function, <function>, which requires one or more parameters to be supplied when the function is called. However, your code is missing one or more parameters (<n>).	The error message will tell you on which line your function is called. Check the code on that line to make sure you are supplying the correct number of parameters.

Parse error: syntax error, unexpected '='

Cause	Solution
You are defining a variable but have not used the dollar ($) sign when naming it. For example "*x = 5;*" instead of "*$x = 5;*".	The error will give you the line number on which the error occurs. Check that line in your code to make sure you are defining the variable correctly.

Notice: Use of undefined constant <name> - assumed '<name>'

Cause	Solution
You are most likely referring to a variable incorrectly. For example, you have defined a variable called *$x* in your code, but are referring to it as "x".	The error will give you the line number on which the error occurs. Make sure you are referring to your variable correctly.

Fatal error: Call to undefined function <function>()

Cause	Solution
You are calling a function, <function>(), which does not exist.	Check the spelling of the function you are calling. If the spelling is correct, make sure the file in which the function exists has been included in your project.

Appendix B: Glossary of terms

Term	Description
Array	An array allows you to store several pieces of data in one variable. Each piece of data has a numerical key which can be used to refer to that data.
Associative array	An associative array allows you to store several pieces of data in one variable. Each piece of data has its own key which can be used to refer to that data. In an associative array each key has a unique name, rather than a number.
Boolean	A Boolean is a data type that holds the value "true" or "false".
Camel case	Camel case is where names with multiple words are written as one word without spaces, with capital letters to separate the words. This is common when naming functions, for example *myFunctionName()*.
Client side	Something that is described as client side is executed on the user's machine. JavaScript, for example, is a client side language because it is run by the web browser after the page has downloaded.
Concatenation Operator	The concatenation operator is a full stop (.) and allows two values to be joined together.
Conditional statement	A conditional statement allows different code to be executed depending on whether a condition is met. An example is the *if()* statement.

Constant	A constant is a given name that refers to a value. It is similar to a variable, but the value of a constant cannot be changed at run time.
Cookie	A cookie is a text file that is stored in the client's web browser. They store data from a website that the site can retrieve at a later date.
Error Handler	An error handler is a function that is designed to trap and process any errors that occur in the code.
Expression	An expression is essentially anything that has a value, for example variables or constants. A function that returns a value can also be classed as an expression.
Float	A float, or floating point number, is a number that can be a whole number or a decimal. This data type is usually used for large decimal numbers.
Function	A function is a collection of code that runs independently to any other code. Functions can be called from any block of code, and are often used to run repetitive tasks to avoid code duplication.
Integer	An integer is a whole number.
Loop	A loop allows a block of code to run repeatedly, either a set number of times or while a given condition is true.
Multidimensional array	A multidimensional array is an array that holds one or more arrays within it.
Null	A data type that represents a variable with no value.

Operator	An operator is a symbol used to assign a value to a variable, or compare two or more variables in a conditional statement.
Prepared Statement	When connecting to a database, a prepared statement allows you to prepare an SQL statement before you execute it. Prepared statements allow you to run similar queries with greater efficiency.
Query	In a database a query is a piece of SQL code that is executed by the database in order to modify data or return a list of records.
Server side	Something that is described as server side is executed on the web hosting server, before any results are sent to the client. PHP is a server side language.
SQL	Structured Query Language (SQL) is a language used to create, retrieve, and organise data in a database.
String	A string is a data type that holds text.
Variable	A variable is a named element that holds a piece of data. The data held within a variable can change during runtime.

Appendix C: Useful PHP functions

There are hundreds of built in PHP functions to make your life much easier. Here are just a few essential functions along with links to the official PHP documentation.

Variable functions

Function Name	Description
filter_var()	Filter a variable to see if it appears valid.
is_array()	Check to see if a given variable is an array.
is_bool()	Check to see if a given variable is a Boolean (true or false) data type.
is_int()	Check to see if a given variable is an integer (whole number) data type.
is_null()	Check to see if a given variable is null, i.e. has no value.
is_numeric()	Check to see if a given variable is a number. Unlike is_int(), the number doesn't necessarily have to be an integer.
is_string()	Check to see if a given variable is a string data type.
isset()	Check to see if a given variable has been initialized.
unset()	Destroy a variable.

String functions

Function Name	Description
addslashes()	Returns a string with backslashes before characters that need to be escaped for database queries (e.g. single/double quotes and backslash characters).
echo()	Print text to the page. This function is the same as print().
html_entity_decode()	The opposite of htmlentities(), converts a string containing HTML entities to normal characters.
htmlentities()	Convert a string to HTML entities. This is a useful security measure, and should be used whenever printing data from an external source to the page.
ltrim()	Trim whitespace (e.g. spaces, tabs, line breaks) from the left side of a string.

print()	Print text to the page. This function is the same as echo().
rtrim()	Trim whitespace (e.g. spaces, tabs, line breaks) from the right side of a string.
strip_tags()	Remove any HTML tags from a given string variable.
strlen()	Return the number of characters in a string.
strtolower()	Convert the supplied text to lower case characters.
strtoupper()	Convert the supplied text to upper case characters.
trim()	Trim whitespace (eg spaces, tabs, line breaks) from both ends of a string.

Array functions

Function Name	Description
array()	Create an array.
array_pop()	Remove the last entry from an array.
count()	Count the number of items in an array.
in_array()	Check to see if the given value exists in an array.
sort()	Sort the elements in an array from lowest to highest.

Error functions

Function Name	Description
set_error_handler()	Set the error handler to a custom function. This gives you much more control over how your application deals with errors.
trigger_error()	Triggers a custom error.

Appendix D: The Source code

For your reference, the full source code for the contact form built during this guide is available below. The project consists of three files, **contact.html**, **contact.php**, and **include.php**.

contact.html

```
<!DOCTYPE html>
<html>
	<head>
		<title>Contact us</title>
	</head>
	<body>
		<form action="contact.php" method="post">
			<label for="email">Your email address</label><br />
			<input type="text" name="email" id="email" /><br />

			<label for="subject">Message subject</label><br />
			<input type="text" name="subject" id="subject" /><br />

			<label for="message" >Message</label><br />
			<input type="“textarea"name="message"id="message"/><br />

			<input type="Submit" value="Send Message" />
		</form>
	</body>
</html>
```

contact.php

```
<?php
	// Include our additional file
	include "include.php";

	// Check to see if our form has been submitted
	if(isset($_POST["email"]) == false) {
		// $_POST variable for our "email" control doesn’t exist. The
		form has not been submitted
		header('Location: contact.html');
	} else {
		// Form has been submitted
		if(validate_email($_POST["email"]) == false) {
			// Email address is invalid
			echo "The email address is invalid";
		} elseif($_POST["subject"] == "") {
			// No subject entered
			echo "No subject entered";
		} elseif($_POST["message"] == "") {
			// No message entered
			echo "No message entered";
		} else {
			// Validation passed
			if(send_email($_POST["email"], $_POST["subject"],
			$_POST["message"])) {
				// Message sent
				echo "Thankyou, your email has been sent";
			} else {
				// Error sending email
```

```
                        echo "An error occurred whilst sending the email,
                        please try again later";
                    }
                }
            }
?>
```

include.php

```
<?php
    // Set your own error handler
    set_error_handler("error_handler");

    // Create the function that will handle the error
    function error_handler($err_number,$err_text,$err_file,$err_line) {
        // Display this error
        echo "Oops, an error occurred: " . $err_text . "<br />";
        echo "Error code: " . $err_number . "<br />";
        echo "In file: " . $err_file . "<br />";
        echo "On line: " . $err_line . "<br />";

        // Send an error report
        $report = "Error: " . $err_text . "\r\n";
        $report = $report . "Error code: " . $err_number . "\r\n";
        $report = $report . "In file: " . $err_file . "\r\n";
        $report = $report . "On line: " . $err_line;
        $success = send_mail("error@ralphsdomainname.com", "Error
        report", $report);
        if($success == true) {
            echo "This error has been reported to the website
            administrator.<br />";
            }
    }

    // Function to validate an email address
    function validate_email($email) {
            if($email == "") {
                    // No email address returned
                    return false;
            } elseif(filter_var($email, FILTER_VALIDATE_EMAIL) == false) {
                    // The email address is not in the correct format
                    return false;
            } else {
                    // Email address is valid
                    return true;
            }
    }

    // Function to send an email
    function send_email($from, $subject, $message) {
            if(validate_email($from) == false) {
                    // Sender email address is invalid
                    return false;
            } elseif($subject == "") {
                    // No subject supplied
                    return false;
            } elseif($message == "") {
                    // No message supplied
                    return false;
            } else {
                    // Set timezone
                    date_default_timezone_set("Europe/London");

                    // The email address to send to
                    $to = "ralph@ralphsdomainname.com";

                    // Prepare the email headers
                    $headers = "From: " . $from . "\r\n";
                    $headers = $headers . "Reply-To: " . $from . "\r\n";

                    // Send the email
                    ini_set("sendmail_from", $from);
                    return mail($to,$subject,$message,$headers,"-f" .$from);
              }
        }
?>
```

Important: Don’t forget to substitute the “ralph@ralphsdomainname.com” and “error@ralphsdomainname.com” email addresses with your own!