

---

# Forms, Cookies and Sessions

A WebApp Tutorial

Adrian Giurca

Chair of Internet Technology, Institute for Informatics



November 6, 2006

Revision History

Sept. 20, 2005

Sept. 12, 2006

Nov. 06, 2006

Revision 1

Revision 2

Revision 3

## Table of Contents

1. HTML Forms (GET and POST) .....	1
2. Validation .....	3
2.1. Interactive models are difficult to implement in the web environment .....	4
2.2. Post-validation models are practical in web database applications .....	4
2.3. Server-Side Validation with PHP .....	4
2.3.1. Mandatory Data .....	4
2.3.2. Validating Strings .....	5
2.3.3. Validating numbers .....	8
2.3.4. Validating credit cards .....	9
2.3.5. Validating Dates and Times .....	11
2.4. Client-Side Validation .....	12
3. Cookies .....	12
3.1. Setting Cookies .....	12
3.2. Retrieving and Altering Cookies .....	13
3.3. A Simple Application: Personalization of web pages .....	14
4. Sessions .....	16
4.1. Session Management .....	16
4.2. PHP Session Management .....	17
4.2.1. Starting a Session .....	17
4.2.2. Using Session Variables .....	17
4.2.3. Ending a Session .....	18
4.2.4. Designing Session-Based Applications .....	19
4.2.5. Reasons to Use Sessions .....	20
4.2.6. Reasons to Avoid Sessions .....	21
Bibliography .....	21

## 1. HTML Forms (GET and POST)

One of the most powerful features of PHP is the way it handles HTML forms. The basic concept that is important to understand is that any form element will automatically be available to your PHP scripts.

When a form is submitted to a PHP script, the information from that form is automatically made available to the script. There are many ways to access this information:

### Example 1. A Simple HTML form

```
<form action="action.php" method="post">
  Name: <input type="text" name="username" /><br/>
  Email: <input type="text" name="email" /><br/>
  <input type="submit" name="submit" value="Submit me!"/>
</form>
```

There is nothing special about this form. It is a straight HTML form with no special tags of any kind. When the user fills in this form and hits the submit button, the `action.php` page is called. In this file you would write something like this:

### Example 2. Printing data from our form

```
<?php
$name = $_POST['username'];
$age = $_POST['age'];
print"<p> Hi $name. You are $age years old.</p>";
?>
```

A sample output of this script may be:

```
Hi Joe. You are 22 years old.
```

Depending on your particular setup and personal preferences, there are many ways to access data from your HTML forms.

### Example 3. Accessing data from a simple POST HTML form

```
<?php
if (array_key_exists('my_name', $_POST)) {
    print '<h1>'. "Hello, ". $_POST['my_name']. '</h1>';
} else {
    $action=$_SERVER['PHP_SELF'];
    print<<HTML
    <form action=$action method="post">
      User <input name="my_name" type="text"><br/>
      <input type="submit" value="Hello">
    </form>
HTML;
}
?>
```

Using a GET form is similar except you'll use the appropriate GET predefined variable instead. GET also applies to the `QUERY_STRING` (the information after the `?` in a URL). So, for example, `http://www.example.com/test.php?id=3` contains GET data which is accessible with `$_GET['id']`.

## Example 4. More complex form variables

```
<?php
if (isset($_POST['action']) && $_POST['action'] == 'submitted') {
    echo '<pre>';
    print_r($_POST);
    echo '<a href="'. $_SERVER['PHP_SELF'] .' ">Please try again</a>';
    echo '</pre>';
} else {
?>
<form action="<?php echo $_SERVER['PHP_SELF']; ?>" method="post">
    Name: <input type="text" name="personal[name]" /><br />
    Email: <input type="text" name="personal[email]" /><br />
    Beer: <br />
    <select multiple name="beer[]">
        <option value="warthog">Warthog</option>
        <option value="guinness">Guinness</option>
        <option value="stuttgarter">Stuttgarter Schwabenbräu</option>
    </select><br />
    <input type="hidden" name="action" value="submitted" />
    <input type="submit" name="submit" value="submit me!" />
</form>
<?php
}
?>
```

### Note

When submitting a form, it is possible to use an image instead of the standard submit button with a tag like:

```
<input type="image" src="image.gif" name="sub" />
```

When the user clicks somewhere on the image, the accompanying form will be transmitted to the server with two additional variables, `sub_x` and `sub_y`. These contain the coordinates of the user click within the image.

## 2. Validation

Validation is actually two processes: *finding errors* and *presenting error messages*. Finding errors can be interactive, where data is checked as it's entered, or post-validation, where the data is checked after entry. Presenting errors can be field-by-field—where a new error message is presented to the user for each error found—or it can be batched, where all errors are presented as a single message. There are other dimensions to validation and error processing, such as the degree of error that is tolerated and the experience level of the user. However, considering only the basic processes, the choice of when to error-check and when to notify the user, leads to four common approaches:

- *Interactive validation with field-by-field errors.* The data in each field is validated when the user exits or changes the field. If there is an error, the user is alerted to that error and may be required to fix the error before proceeding.
- *Interactive validation with batched errors.* The data in all fields is validated when the user leaves one field. If there are one or more errors, the user is alerted to these, and can't proceed beyond the current page without fixing all errors.
- *Post-validation with field-by-field errors.* The user first enters all data with no validation. The data is then checked and errors are reported for each field, one by one. The user fixes each error in turn and resubmits the data for revalidation.
- *Post-validation with batched errors.* The user first enters all data with no validation. The data is then checked, and all errors in the data are reported in one message to the user. The user then fixes all errors and resubmits the data for revalidation.

## 2.1. Interactive models are difficult to implement in the web environment

Server-side scripts are impractical for this task, because an HTTP request and response is required to validate each field that's entered. This is usually unacceptable, because the user is required to submit the data after entering each field. The result is that response times are likely to be slow and the server load high.

Client-side scripts can implement an interactive model. However, validation on the client side should not be the only method of validation because the user can passively or actively bypass the client-side processes.

## 2.2. Post-validation models are practical in web database applications

Both client- and server-side scripts can validate all form data during the submission process. In many applications, reasonably comprehensive validation is performed on the client side when the user clicks the form submit button.

Client-side validation reduces server and network load, because the user's browser ensures the data is valid prior to the HTTP request. Client-side validation is also usually faster for the user. If client-side validation succeeds, data is submitted to the server and the same (or often more comprehensive) validation is performed. Duplicating client validation on the server is essential because of the unreliability of client-side scripts and lack of control over the client environment.

The post-validation model can be combined with either field-by-field or batch error reporting. For server-side validation, the batch model is preferable to a field-by-field implementation, as the latter approach has more overhead and is usually slower because each form error requires an additional HTTP request and response.

For client-side post-validation, either error-reporting model can be used. The advantage of the field-by-field model is that it leads the user through the process of correcting the data and the cursor can be directed to the field containing the error, making error correction easier. The disadvantage is that several errors require several error messages. The advantage of the batch approach is that all errors are presented in one message but the disadvantage is that the cursor can't easily be directed to the field requiring correction and it's sometimes unclear to the user how to correct the data.

### Note

Server-side validation is essential to secure a web database and to ensure that system and DBMS constraints are met.

Client-side validation may be implemented in addition to server-side validation, but all client-side functionality should be duplicated at the server side. Never trust the user or the client browser.

The choice of which reporting model to use depends on the size and complexity of the form and on the system requirements.

## 2.3. Server-Side Validation with PHP

In this section, we introduce validation on the server using PHP. We show you how to validate numbers including currencies and credit cards, strings including email addresses and Zip Codes, and dates and times. We also show you how to check for mandatory fields, field lengths, and data types.

### 2.3.1. Mandatory Data

For example, to test if the user's surname has been entered, the following approach is used:

```
/// Validate the Surname
if (empty($surname)){
    formerror($template, "The surname field cannot be blank.", $errors);
}
```

The `formerror()` function outputs the error message as a batch error using a template. For simplicity and compactness in the remainder of our examples, we omit the `formerror()` function from code fragments and simply output the error messages using `print`.

## 2.3.2. Validating Strings

In this section, we discuss nonnumeric validation. We begin with the basics of validating strings, and then discuss the specifics of email addresses, URLs, and Zip or post codes.

### 2.3.2.1. Basic techniques

The most of the data entered by users will be strings and require validation: checking that strings contain legal characters, are of the correct length, or have the correct format is the most common validation task. Strings are popular for two reasons: first, all data from a form that is stored in the superglobals `$_GET` and `$_POST` is of the type string; and, second, some nonstring data such as a date of birth or a phone number is likely to be stored as a string in a database table because it may contain brackets, dashes, and slashes.

#### Example 5. Check if a string meets a minimum or maximum length requirement

```
if (strlen($password) < 4 || strlen($password) > 8){
    print("Password must contain between 4 and 8 characters");
}
```

Common tests for legal characters include checking if strings are uppercase, lowercase, alphabetic, or are drawn from a defined character set (such as, for example, alphabetic strings that may include hyphens or apostrophes). In PHP, the `is_string()` function can be used to check if a variable is a string type. However, this is of limited use in validation because a string can contain any character including (or even exclusively) digits or special characters. It's more useful to test what characters are in the string or detect characters that shouldn't be there.

#### Example 6. A simpler alphabetic test (English Language)

```
if (!eregi("^[a-z]*$", $string)){
    print("String must contain only alphabetic characters.");
}
```

#### Example 7. The firstname and surname contain only alphabetic characters, hyphens, and apostrophes; white space, numbers, and other special characters aren't allowed.

```
if (!eregi("^[a-z'-]*$", $firstName)){
    print("The first name can contain only alphabetic ".
        "characters or - or '");
}
```

#### Example 8. The customer's middle initial is exactly one alphabetic character

```
if (!empty($initial) && !eregi("^[a-z]$", $initial)){
    print("The initial field must be empty or one character in length.");
}
```

### 2.3.2.2. Validating Zip and postcodes

Zip or postcodes are numeric in most countries but are typically stored as strings because spaces, letters, and special characters are sometimes allowed. We might validate Zip Codes using a simple regular expression:

#### Example 9. Validating ZIP Codes

```
if (!ereg("^[0-9]{4,5}$", $zipcode)){
    print("The zipcode must be 4 or 5 digits in length.");
}
```

**Example 10. Validate many popular Zip and postcodes**

```
function checkcountry($country, $zipcode){

switch ($country){
  case "Austria":
  case "Australia":
  case "Belgium":
  case "Denmark":
  case "Norway":
  case "Portugal":
  case "Switzerland":
  if (!ereg("^[0-9]{4}$", $zipcode)){
    print("The postcode/zipcode must be 4 digits in length");
    return false;
  }
  break;

  case "Finland":
  case "France":
  case "Germany":
  case "Italy":
  case "Spain":
  case "USA":
  if (!ereg("^[0-9]{5}$", $zipcode)){
    print("The postcode/zipcode must be 5 digits in length");
    return false;
  }
  break;

  case "Greece":
  if (!ereg("^[0-9]{3}[ ][0-9]{2}$", $zipcode)){
    print("The postcode must have 3 digits, a space, and then 2 digits");
    return false;
  }
  break;

  case "Netherlands":
  if (!ereg("^[0-9]{4}[ ][A-Z]{2}$", $zipcode)){
    print("The postcode must have 4 digits, a space, and then 2 letters");
    return false;
  }
  break;

  case "United Kingdom":
  if(!ereg("^[([A-Z][0-9]{1,2})|([A-Z]{2}[0-9]{1,2})|".
    "([A-Z]{2}[0-9][A-Z])|([A-Z][0-9][A-Z])|".
    "([A-Z]{3})) [0-9][A-Z]{2}$", $zipcode)){
    print "The postcode must begin with a string of the format
      A9, A99, AA9, AA99, AA9A, A9A, or AAA,
      and then be followed by a space and a string
      of the form 9AA.
      A is any letter and 9 is any number.";
    return false;
  }
  break;
  default:
    // No validation
}
return true;
}
```

**2.3.2.3. Validating email addresses**

Email addresses are another common string that requires field organization checking. There is a standard maintained by the Internet Engineering Task Force (IETF) called RFC-2822 that defines what a valid email address can be, and it's much more complex than might be expected. For example, an address such as the following is valid:

```
" <test> "@example.com
```

We use a regular expression and network functions to validate an email address.

### Example 11. A function to validate an email address

```
function checkEmail($email){
    // Check syntax
    $validEmailExpr = "[0-9a-z~!#$%&_-]([.]?[0-9a-z~!#$%&_-])*".
        "@[0-9a-z~!#$%&_-]([.]?[0-9a-z~!#$%&_-])*$";

    // Validate the email
    if (empty($email)){
        print "The email field cannot be blank";
        return false;
    }
    elseif (!eregi($validEmailExpr, $email)){
        print "The email must be in the name@domain format.";
        return false;
    }
    elseif (strlen($email) > 30){
        print "The email address can be no longer than 30 characters.";
        return false;
    }
    elseif (function_exists("getmxrr") && function_exists("gethostbyname")){

        // Extract the domain of the email address
        $maildomain = substr(strstr($email, '@'), 1);
        if (!(getmxrr($maildomain, $temp) ||
            gethostbyname($maildomain) != $maildomain)){
            print "The domain does not exist.";
            return false;
        }
    }
    return true;
}
```

#### 2.3.2.4. Validating URLs

Home pages, links, and other URLs are sometimes entered by users. In PHP, validating these is straightforward because we use the library function `parse_url()`.

The `parse_url()` function takes one parameter, a URL string, and returns an associative array that contains the components of the URL.

### Example 12. Using `parse_url()`

```
$bits =
    parse_url("http://www.example.com/test.php?status=F#message");
foreach($bits as $var => $val){
    echo "{$var} is {$val}\n";
}
```

produces the output:

```
scheme is http
host is www.example.com
path is /test.php
query is status=F
fragme is message
```

**Example 13. Validating with `parse_url()`**

```
$bits = parse_url($url);

if ($bits["scheme"] != "http"){
    print "URL must begin with http://.";
}
elseif (empty($bits["host"])){
    print "URL must include a host name.";
}
elseif (function_exists('checkdnsrr') && !checkdnsrr($bits["host"], 'A')){
    print "Host does not exist.";
}
```

**2.3.3. Validating numbers**

The two most common checks for numbers are whether they are in fact numeric and whether they're within a required range.

In PHP, the `is_numeric()` function can be used to check if a variable contains only digits or if it matches one of the legal number formats.

**Example 14. Check if a salary is numeric**

```
if (!is_numeric($salary)){
    print "Salary must be numeric";
}
```

**Note**

The `is_numeric()` function doesn't always behave in the way you expect. Leading and trailing spaces, carriage returns, commas, and spaces after minus signs can result in a false return value. Leading and trailing spaces can be removed with the `trim()` function, while allowing specialized formats may instead require the use of a regular expression.

**Example 15. Valid number**

Suppose that a whole-dollar salary is provided from a form through the `POST` method and is stored as `$_POST["salary"]`. To check if it's a valid number, use the following steps:

```
if (!is_numeric($salary)){
    print "Salary must be numeric";
}else{

    // remove spaces and convert to an integer
    $salary = intval($_POST["salary"]);
}
```

**Example 16. Check that an age is in a sensible range**

```
if ($age < 5 || $age > 105){
    print "Age must be in the range 5 to 105";
}
```

**Example 17. Check if a currency amount is in whole dollars and between four and six digits in length**

```
if (!ereg("[0-9]{4,6}", $salary)){
    print "Salary must be in whole dollars";
}
```

### Example 18. Validate a phone number using a regular expression

```
// Phone is optional, but if it is entered it must have
// correct format

$validPhoneExpr = "^[0-9]{2,3}[ ]*[0-9]{4}[ ]*[0-9]{4}$";
if (!empty($phone) && !ereg($validPhoneExpr, $phone)){
    print "The phone number must be 8 digits in length, " .
        "with an optional 2 or 3 digit area code";
}
```

#### 2.3.4. Validating credit cards

There are two steps to validating a credit card that's entered for payment of goods or services: first, we need to check the credit card number and its expiration date are valid; and, second, we need to verify that the payment will be honored by the bank or other credit card provider. If the user's entering their credit card as part of the account creation process, the second step isn't usually needed until they make a payment. In this section, we show you how to validate a credit card number. Expiration dates can be validated using the date checking functions discussed later in this section.

## Example 19. Validating Credit Card Number

```
function checkCard($cc, $ccType){
    if (!ereg("[0-9 ]*$", $cc)){
        print "Card number must contain only digits and spaces.";
        return (false);
    }

    // Remove spaces
    $cc = ereg_replace('[ ]', '', $cc);

    // Check first four digits
    $firstFour = intval(substr($cc, 0, 4));
    $type = "";
    $length = 0;

    if ($firstFour >= 8000 && $firstFour <= 8999){

        // Try: 8000 0000 0000 1001
        $type = "SurchargeCard";
        $length = 16;
    }elseif ($firstFour >= 9100 && $firstFour <= 9599){

        // Try: 9100 0000 0001 7
        $type = "AustralianExpress";
        $length = 13;
    }

    if (empty($type) || strcmp($type, $ccType) != 0){
        print "Please check your card details.";
        return (false);
    }

    if (strlen($cc) != $length){
        print "Card number must contain {$length} digits.";
        return (false);
    }

    $check = 0;

    // Add up every 2nd digit, beginning at the right end
    for($x=$length-1;$x>=0;$x-=2){
        $check += intval(substr($cc, $x, 1));
    }

    // Add up every 2nd digit doubled, beginning at the right end - 1.
    // Subtract 9 where doubled value is greater than 10
    for($x=$length-2;$x>=0;$x-=2){
        $double = intval(substr($cc, $x, 1)) * 2;
        if ($double >= 10){
            $check += $double - 9;
        }else{
            $check += $double;
        }
    }

    // Is $check not a multiple of 10?
    if ($check % 10 != 0){
        print "Credit card invalid. Please check number.";
        return (false);
    }
    return (true);
}
```

## 2.3.5. Validating Dates and Times

Dates of birth, expiry dates, order dates, and other dates are commonly entered by users. Most dates require specialized checks to see if the date is valid and if it's in a required date range. Times are less complicated, but specialized checks are still useful.

### 2.3.5.1. Dates

Dates can be given in several different formats and using many different calendars. We only discuss the Gregorian calendar here.

In the U.S., months are listed before days, but the majority of the rest of the world uses the opposite approach. Years can be provided as two or four digits, although we recommend avoiding two digit years for the obvious confusion caused when 99 comes before 00. This leads to four formats: DDMYY, DDMYYYY, MMDDYY, and MMDDYYYY, where *y* is a year digit, *m* is month digit, and *d* is a day digit.

In all date formats, a forward slash, a hyphen, or (rarely) a colon can be used to separate the groups, leading to twelve formats in total. For sorting, a thirteenth (convenient) format is YYYYMMDD without the separators. Dates can also be specified using month names, leading to strings such as 01-Aug-1966 and 01 August 1966.

### Example 20. Date-of-birth validation

Let's suppose the user is required to provide a date of birth in the format common to most of the world, DD/MM/YYYY. We then need to validate this date of birth to check that it has been entered and to check its format, its validity, and whether it's within a range. The range of valid dates in the example begins with the user being alive—for simplicity, we assume alive users are born after 1902—and ends with the user being at least 18 years of age.

```
function checkDOB($birth_date){
    if (empty($birthDate)){
        print "The date of birth field cannot be blank.";
        return false;
    }

    // Check the format and explode into $parts
    elseif (!ereg("^[0-9]{2}/([0-9]{2})/([0-9]{4})$", $birthDate, $parts)){
        print "The date of birth is not a valid date in the
        format DD/MM/YYYY";
        return false;
    }elseif (!checkdate($parts[2],$parts[1],$parts[3])){
        print "The date of birth is invalid. Please check that the month is
        between 1 and 12, and the day is valid for that month.";
        return false;
    }elseif (intval($parts[3]) < 1902 ||
        intval($parts[3]) > intval(date("Y"))){
        print "You must be alive to use this service.";
        return false;
    }else{
        $dob = mktime(0, 0, 0, $parts[2], $parts[1], $parts[3]);

        // Check whether the user is 18 years old.
        if ((float)$dob > (float)strtotime("-18years")){
            print "You must be 18+ years of age to use this service";
            return false;
        }
    }
    return true;
}
```

### Note

The `mktime()` function works for years between 1901 and 2038 on Unix systems, and only from 1970 to 2038 for variants of Microsoft Windows.

### 2.3.5.2. Times

Times are easier to work with than dates, but they also come in several valid formats. These include the 24-hour clock format `9999`, the 12-hour clock formats `99:99am` or `99:99pm` (or with a period instead of a colon), and formats that include seconds and hundredths of seconds. In each format, different ranges of values are allowed.

#### Example 21. Validating times

Consider an example where a user is required to enter a date in the 12-hour format using a colon as the separator. With this format, `12:42pm` and `1:01am` are valid times. You can validate this format using the following regular expression:

```
if (!eregi("^(1[0-2]|0[1-9]):([0-5][0-9])(am|pm)$", $time)){
    print "Time must be a valid 12-hour clock time in the format
        HH:MMam or HH:MMpm. ";
}
```

## 2.4. Client-Side Validation

Usually, JavaScript scripting language is the solution to client-side method for validation.

## 3. Cookies

Cookies are pieces of text that are sent to a user's Web browser. Cookies can help you create shopping carts, user communities, and personalized sites.

Take the shopping example. Suppose you assign an identification variable to a user so that you can track what he does when he visits your site. First, the user logs in, and you send a cookie with variables designed to say, *"This is Joe, and Joe is allowed to be here."* While Joe is surfing around your site, you can say, *"Hello, Joe!"* on each and every page. If Joe clicks through your catalog and chooses 14 different items to buy, you can keep track of these items and display them all in a bunch when Joe goes to the checkout area.

### 3.1. Setting Cookies

The `setcookie()` function, used to set one cookie at a time, expects six arguments:

1. *Name*. Holds the name of the variable that will be kept in the global `$_COOKIE` and will be accessible in subsequent scripts.
2. *Value*. The value of the variable passed in the name parameter.
3. *Expiration*. Sets a specific time at which the cookie value will no longer be accessible. Cookies without a specific expiration time will expire when the Web browser closes.
4. *Path*. Determines for which directories the cookie is valid. If a single slash is in the path parameter, the cookie is valid for all files and directories on the Web server. If a specific directory is named, this cookie is valid only for pages within that directory.
5. *Domain*. Cookies are valid only for the host and domain that set them. If no domain is specified, the default value is the host name of the server that generated the cookie. The domain parameter must have at least two periods in the string in order to be valid.
6. *Security*. If the security parameter is 1, the cookie will only be transmitted via HTTPS, which is to say, over a secure Web server.

## Example 22. Setting a Test Cookie

```
<?php
$cookieName = 'testCookie';
$cookieValue = 'test string!';
$cookieExpire = time()+3600; // in one hour
$cookieDomain = "127.0.0.1";
setcookie($cookieName, $cookieValue, $cookieExpire, "/examples" , $cookieDomain, 0);
print<<<HTMLBLOCK
<html xmlns="http://www.w3.org/1999/xhtml">
<head><title>Simple cookie set</title></head>
<body><p>I set a cookie</p></body></html>
HTMLBLOCK;
?>
```

## 3.2. Retrieving and Altering Cookies

When a Web browser accepts a cookie, you can't extract its value until the next HTTP request is made. In other words, if you set a cookie called name with a value of Julie on page 1, you can't extract that value until the user reaches page 2 (or page 5 or page 28—just some other page that isn't the page on which the cookie is initially set).

PHP allows you to access the cookie simply by using it as a variable! This means if you have a cookie called 'testCookie', putting `<?php $testCookie; ?>` any where in a PHP page will retrieve the variable. If you want to assign `$testCookie` a value other than the cookie value using a form, you must specifically do so as follows.

```
<?php
// a cookie named "testCookie" has the value of "test string!"
echo $testCookie; //will print "test string!"
// This page has recieved a POST value with the name of testCookie and a value of "another test !"
$testCookie= $_POST['testCookie'];
echo $testCookie; //will now print "another test !"
?>
```

## Example 23. Read specific cookies

In order to read the value of a specific named cookie we just insert the name as the key of the `$_COOKIE` array.

```
<?php
//The following will print out the value of the cookie named 'testCookie'
echo $_COOKIE['testCookie'];
//so will this!!
echo $testCookie;
?>
```

## Example 24. Retrieving All the Cookies

In order to retrieve a full list of the cookies, all we have to do is iterate through the `$_COOKIE` array.

```
<?php
echo "<p>There are ".count($_COOKIE)." cookies stored for this domain.</p>";
for ($i=0;$i < count($_COOKIE);$i++){
    echo "Cookie name: ".key($_COOKIE)." Cookie value: ".current($_COOKIE);
    next($_COOKIE);
}
?>
```

## Example 25. Altering Cookies

To alter a cookie we simply use the `setcookie()` function to pass new parameters to a cookie of the same name.

```
//change the value of fname cookie to Francis,and persist it for a year!
//old values : - setcookie("testCookie","test string!",time()+3600,"/examples",localhost",1)
setcookie("testCookie","another value",time()+(3600*24*365),"/examples",localhost",1)
```

## Example 26. Deleting Cookies

To delete a cookie simply set the expires date to a date in the past!

```
// delete this cookie
//old values : - setcookie("testCookie","test string!",time()+3600,"/examples",localhost,1)
//set to a date an hour ago
setcookie("testCookie","test string!",time()-3600,"/examples",localhost,1)
```

By default, a cookie can be accessed by the document that created the cookie, and by documents residing in the same folder, or in any descendant folders. If the path and domain values have been set then the cookie will be read by documents who meet these criteria.

Some cookies limitations:

- Browsers are not required to retain more than a total of 300 cookies
- Browsers are required to retain no more than 20 cookies for a single domain
- Cookies cannot contain more than 4K of Data
- Clients can also switch off Cookies on their browsers.

## 3.3. A Simple Application: Personalization of web pages

This lesson teaches you nothing new, it just builds a simple web application that shows you how to use PHP to personalize a clients pages, and how to persist data across pages by using cookies. This application gets no points for style, but the coding principles can easily be ported to larger projects.

The Application consists of three pages.

1. A *Welcome Page* where old customers get a personalized welcome (`/examples/cookies/welcome.php`).

```
<?php
// The lines below makes sure that we run a fresh un-cached page each time we run this page.
header("Expires: Mon, 26 Jul 1997 05:00:00 GMT"); // Date in the past
header("Last-Modified: " . gmdate("D, d M Y H:i:s") . " GMT"); // always modified
header("Cache-Control: no-store, no-cache, must-revalidate"); // HTTP/1.1
header("Cache-Control: post-check=0, pre-check=0", false);
header("Pragma: no-cache"); // HTTP/1.0

if($_COOKIE["myCookie"]){
    // If the cookie named "myCookie" exists, we read the cookie values into variables.
    // This is not strictly necessary because calling, say $lastname will get the cookie value anyway.
    $firstname=$_COOKIE["firstname"];
    $lastname=$_COOKIE["lastname"];
    $color=$_COOKIE["color"];
    $bgcolor=$_COOKIE["bgcolor"];
print<<<HTML
<body style="color:$color; bgcolor: $bgcolor">
<h2>Welcome $firstname $lastname</h2>
<p>
    If you are not $firstname $lastname, or to change the
    color scheme, please <a href="register.php">click here</a>
</p>
</body>
HTML;
}
else{
    // If the cookie "myCookie" does NOT exist, we redirect the user
    // to the registration page, and exit this page.
    header("Location: http://localhost/examples/cookies/register.php");
    exit;
}
?>
```

2. A *Registration Page*, where new customers can register and store their color preferences, or where old customers can change their preferences (/examples/cookies/register.php).

```
<?php
header("Expires: Mon, 26 Jul 1997 05:00:00 GMT"); // Date in the past
header("Last-Modified: " . gmdate("D, d M Y H:i:s") . " GMT"); // always modified
header("Cache-Control: no-store, no-cache, must-revalidate"); // HTTP/1.1
header("Cache-Control: post-check=0, pre-check=0", false);
header("Pragma: no-cache"); // HTTP/1.0
?>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Register</title>
</head>
<body>
<h2>Welcome</h2>
<p>This appears first time you visit us and we would like you to register</p>
<form name="fml" action="setcookie.php" method="post">
<input type="text" name="firstname" value=""/>
<input type="text" name="lastname" value=""/>
<p>Pick a color scheme</p>
<div style="width:5cm;color:#ffff00;background-color:#000066;">
Choose a Color Scheme
<input type="radio" name="rb1"
onclick="fml.bgcolor.value='#000066';fml.color.value='#ffff00'" />
</div>
<div style="width:5cm;color:#ff0000;background-color:#00ff00;">
Choose a Color Scheme
<input type="radio" name="rb1"
onclick="fml.bgcolor.value='#00ff00';fml.color.value='#ff0000'" />
</div>
<input type="hidden" name="color" />
<input type="hidden" name="bgcolor" />
<br />
<input type="submit" />
</form>
</body>
</html>
```

3. A *Set-Cookie Page*. The engine that sets the cookies (/examples/cookies/setcookie.php).

```
<?php
header("Expires: Mon, 26 Jul 1997 05:00:00 GMT"); // Date in the past
header("Last-Modified: " . gmdate("D, d M Y H:i:s") . " GMT"); // always modified
header("Cache-Control: no-store, no-cache, must-revalidate"); // HTTP/1.1
header("Cache-Control: post-check=0, pre-check=0", false);
header("Pragma: no-cache"); // HTTP/1.0

//variables will be in global domain, so reset them to the post variables setcookie
$firstname=$_POST["firstname"];
$lastname=$_POST["lastname"];
$color=$_POST["color"];
$bgcolor=$_POST["bgcolor"];

setcookie("lastname",$lastname,time()+3600);
setcookie("firstname",$firstname,time()+3600);
setcookie("color",$color,time()+3600);
setcookie("bgcolor",$bgcolor,time()+3600);

setcookie("myCookie","true",time()+3600);
?>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Set cookie</title>
</head>
<body>
<p>Thank you for registering, now please go to our <a href="welcome.php">welcome pages</a></p>
```

```
</body>  
</html>
```

When we are dealing with cookies it is important that we deal with non-cached pages. Therefore one of the first things we do is use the `header()` function to send information to the server that makes sure a fresh version of the page is downloaded each time.

## 4. Sessions

A fundamental characteristic of the Web is the stateless interaction between browsers and web servers. The stateless nature of HTTP allows users to browse the Web by following hypertext links and visiting pages in any order. HTTP also allows applications to distribute or even replicate content across multiple servers to balance the load generated by a high number of requests.

This stateless nature suits applications that allow users to browse or search collections of documents. However, applications that require complex user interaction can't be implemented as a series of unrelated, stateless web pages. An often-cited example is a shopping cart in which items are added to the cart while searching or browsing an on-line store. The state of the shopping cart (the selected items) needs to be stored somewhere to be displayed when the user visits the order page.

### 4.1. Session Management

A *session* manages the interaction between a web browser and a web server. For example, a session allows an application to track the items in a shopping cart, the status of a customer account application process, whether or not a user is logged in, or the finalizing of an order. Sessions are essential to most web database applications.

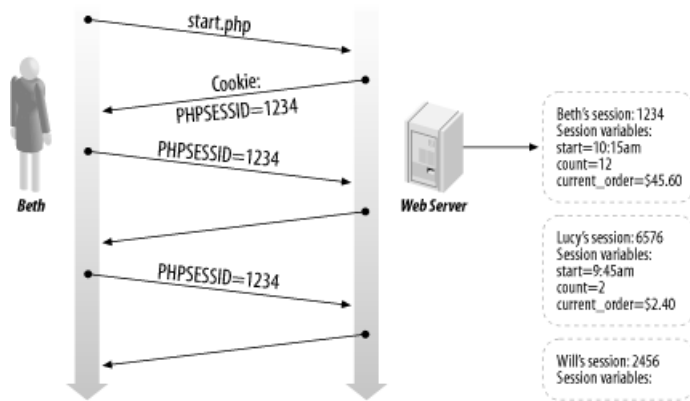
A session has two components: *session variables* and a *session identifier (ID)*. The session variables are the state information that's related to a user's interaction with an application. For example, the session variables might store that the user's shopping cart contains five items, what those items are, their price, what items the user has viewed, and that the user is logged into the application. The session variables are stored at the web server or database server, and are located using the session ID.

When a session is started, the user's browser is given a session ID. This ID is then included with subsequent requests to the server. When a browser makes a request, the server uses the session ID to locate the corresponding session variables, and the variables are read or written as required. In practice, session variables are typically stored at the web server in a file (the PHP default).

The figure below shows how the session variables for Beth's session are identified and stored in the web server environment; the session ID distinguishes between Beth's session and other users of the system.

Using sessions, all of the variables that represent the state of an application don't need to be transmitted over the Web. The session ID is transmitted between the browser and server with each HTTP request and response, but the session data itself is stored at the server. The session ID is therefore like the ticket given at a cloakroom. The ticket is much easier to carry around and ensures that you get back your own hat and coat. Storing variables at the server also helps prevent accidental or intentional tampering with state information.

The session ID is usually transmitted as a cookie. A cookie is a named piece of text that is stored in a web browser, and is sent with HTTP requests, like data sent with the `GET` or `POST` methods. You can find out more about cookies from the interesting Cookie Central web site at <http://www.cookiecentral.com/faq/> or more formally in RFC 2109 at <http://ietf.org/rfc/rfc2109.txt?number=2109>.

**Figure 1. Session ID**

There are three characteristics of session management over the Web:

1. *Information or state must be stored.* Information that must be maintained across multiple HTTP requests is stored in session variables.
2. Each HTTP request must carry an *identifier* that allows the server to process the request with the correct session variables.
3. Sessions need to have a *timeout*. Otherwise, if a user leaves the web site, there is no way the server can tell when the session should end.

## 4.2. PHP Session Management

The three important features of session management—identifying sessions, storing session variables, and cleaning up old sessions—are mostly taken care of by the PHP session management library.

### 4.2.1. Starting a Session

The `session_start()` function is used to create a new session. A session is unique to the interaction between a browser and a web database application. If you use your browser to access several sites at once, you'll have several unrelated sessions. Similarly, if several users access your application each has their own session. However, if you access an application using two browsers (or two browser windows) at the same time, in most cases the browsers will share the same session; this can lead to unpredictable behavior—that's the reason why many web sites warn against it. The first time a user requests a script that calls `session_start()`, PHP generates a new session ID and creates an empty file to store session variables. PHP also sends a cookie back to the browser that contains the session ID. However, because the cookie is sent as part of the HTTP headers in the response to the browser, you need to call `session_start()` before any other output is generated, just as with other functions that set HTTP header fields.

### 4.2.2. Using Session Variables

Once a script has called `session_start()`, PHP provides access to session variables through the superglobal associative array `$_SESSION`. When an existing session is found, PHP automatically reads the session variables from the session file into the array. PHP also automatically writes changes to the array back to the session file once the script ends. However, be careful: if your script doesn't call `session_start()`, the `$_SESSION` array behaves like any other variable and any values are lost when the script ends.

**Example 27. Using sessions** (`/examples/sessions/counter.php`)

```
<?
session_start();
$count++;
print "You have visited this page $count times during this session";
session_register("counter");
?>
```

**4.2.2.1. Unsetting session variables**

To unset a session variable, you use the `unset()` function. To remove all the session variables, you can unset the whole `$_SESSION` array or re-assign a new array.

**Example 28. Unsetting session variables**

```
// To remove the "count" session variable only
unset($_SESSION["count"]);

// To remove all the session variables without destroying the session
$_SESSION = array();
```

**4.2.2.2. Session variable types**

Session variables can be of any type supported by PHP. However, if objects are saved as session variables, you should include class definitions for those objects in all scripts that call `session_start()`, regardless of whether the scripts use the objects or not. This is needed so that PHP can correctly read and write objects from the session store (the file that stores the session variables). To do this, you might store your class definition in an require file such as `myClass.inc` as shown in the following example:

**Example 29. Storing objects in session variables**

```
require "myClass.inc";
// Find the session
start_session();

// later on in the script, store an object as a session variable
$_SESSION["some_object"] = new myClass();
```

**4.2.3. Ending a Session**

At some point in an application, sessions should be destroyed. For example, when a user logs out of an application, a call to the `session_destroy()` function should be made to clean-up the session variables and remove the session file.

**Note**

A call to `session_destroy()` removes the session file from the system but it doesn't remove the session cookie from the browser.

**Example 30. Ending a session**

```
<?php
session_start( );

// ...

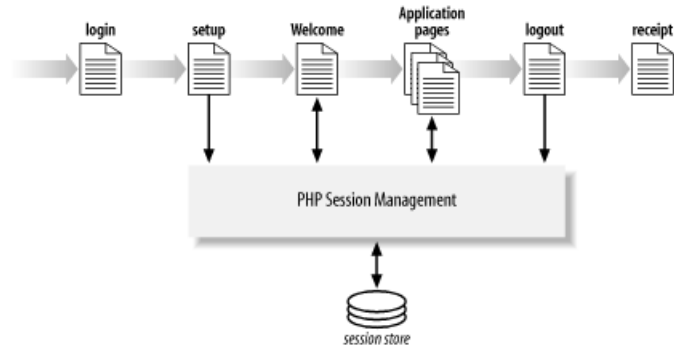
session_destroy( );
header("Location: logout.html");
?>
```

## 4.2.4. Designing Session-Based Applications

### 4.2.4.1. Session to track authenticated users

Applications that require a user to log in—such as online banking—often use sessions to manage the user interaction. The figure below shows a typical flow of pages in such an application, and interaction with PHP session management.

**Figure 2. Typical session-based application**



The login page collects the user credentials using an HTML form and passes these using the `POST` method to the setup script. The setup script is responsible for creating the new session with the first call to `session_start()` and for setting up the initial session variables. After the session is started, the setup script sends a `Location` header field, instructing the browser to relocate to the welcome page. Relocating to the welcome page prevents the setup script being re-run during the session. The following fragment shows the sequence of code in the setup script:

```
// Process the POST variables
$username = $_POST["username"];

// Start the session
session_start( );

// Setup the session variables
$_SESSION["name"] = $username;
$_SESSION["counter"] = 0;

// Relocate to the welcome page
header("Location: welcome.php");
```

The welcome page, and the other application pages, begin by calling `session_start()` to set up the `$_SESSION` superglobal array with the session variables. Each page of the application then interacts with the `$_SESSION` variables as required. For example:

```
// Find the session
session_start( );

// Welcome the user to the application
print "Hi {$_SESSION["name"]}, Welcome to my Application";

// ...

// Update session variables
$_SESSION["counter"]++;

// ...
```

Finally, the session is destroyed when the user requests the logout page. As with all the other pages that interact with the session, the logout script must begin by calling `session_start()`. As discussed previously, a logout page should also redirect to a receipt page to avoid the reload problem. Here's an example logout script:

```
<?
// Find the session
```

```
session_start( );

//...

// Destroy the session
session_destroy( );

// Redirect to a receipt page.
header("Location: logout.html");
?>
```

#### 4.2.4.2. Sessions to track anonymous users

Consider an application that tracks the pages a user has visited. Multiple pages share information using session variables, however the user can enter the web site from any page. Each page in the application can potentially start the session, so each page should be written to test for the existence of the session variable.

The following fragment of code shows how a session variable `$_SESSION["visitedPages"]` is tested and set up prior to being used:

```
// Start or find the session
session_start( );

// Test for the required session variables and add
// them to the session store if they don't exist
if (!isset($_SESSION["visited_pages"])){
    $_SESSION["visitedPages"] = array();
}

// Now use the session
// Add the name of this page to the end of the array
$_SESSION["visitedPages"][ ] = $_SERVER["PHP_SELF"];
```

If the variable isn't in the session store, it's initialized as a new array; if it is in the session store, it's restored automatically by the PHP session handler. After this first step, the name of the current page is added to the end of the array; the superglobal `$_SERVER` contains many elements that describe the server environment, including `PHP_SELF` which is the name of the current PHP script. This same fragment would be added to each script in the application.

The result of using this code is that, for every page the user visits, a new element is added to the array that contains the name of the page. If a user visits a page twice, a second element for that page is added. You could determine how many pages the user has visited by inspecting the size of the array with `count()`, or you could print out the list of pages as in the following fragment:

```
// Print out all of the pages the user has visited
foreach ($_SESSION["visitedPages"] as $page){
    print "Thanks for visiting the {$page} page<br/>";
}
```

For applications that record sensitive information in session variables, you should offer a page that destroys the session. However, for a non-sensitive application, setting a short session timeout and letting the PHP session management garbage collection remove the session is adequate.

#### 4.2.5. Reasons to Use Sessions

- *Performance.* In a stateless environment, an application may need to repeat a computationally expensive or slow operation. An example might be a financial calculation that requires many SQL statements and calls to mathematics libraries before displaying the results on several web pages. An application that uses a session variable to remember the result exposes the user, and the server, to the cost of the calculation only once.
- *Sequence of interaction.* Often a web database application needs to present a series of screens in a controlled order. One style of application (known as a wizard) guides a user through what would otherwise be a complex task using a sequence of screens. Wizards are sometimes used for complex configurations, such as some software installations, and often alter the flow of screens based on user input.

- *Intermediate results.* Many web database applications validate data before creating or updating a row in the database, preventing erroneous data from being saved. Sessions can keep the intermediate data, so that incomplete data can be corrected when errors are detected.
- *Personalization.* Sessions can be used to personalize a web site by tracking a user's preferences. For example, a user might specify a background color, layout preferences, or their interests. This information is then saved in the session store, and can be accessed by all scripts to personalize the application.

#### 4.2.6. Reasons to Avoid Sessions

The reasons to avoid sessions focus mainly on the stateless nature of HTTP. HTTP provides many features that enhance the performance and robustness of web browsing, and these are often limited by the requirements of a stateful application.

- *Need for centralized session store.* In an application that uses sessions, each HTTP request needs to be processed in the context of the session variables to which that request belongs. The state information recorded as the result of one request needs to be available to subsequent requests. Most applications that implement sessions store session variables at the web server. Once a session is created, all subsequent requests must be processed on the web server that holds the session variables. This requirement prevents such applications from using HTTP to distribute requests across multiple servers and therefore can't easily scale horizontally to handle large numbers of requests.
- *Performance.* When a server that offers session management processes a request, identifying and accessing session variables introduces unavoidable overhead. The session overhead results in longer processing times for requests, which affects the performance and capacity of a site. While sessions can improve application performance (for example, a session can keep the result of an expensive operation) the gains may be limited and outweighed by the extra processing required.
- *Timeouts.* Sessions can also cause synchronization problems. Because HTTP is stateless, there is no way of knowing when a user has really finished with an application.
- *Bookmark restrictions.* Because HTTP is stateless, browsers allow users to save URLs as a list of bookmarks or favorite sites. The user can return to a web site at a later date by simply selecting a bookmarked URL. Web sites that provide weather forecasts, stock prices, and even search results from a web search engine are examples of the sites a user might want to bookmark. Bookmarking can fail when sessions are used in the script that's bookmarked. For example, if a user bookmarks a session-based stock price page and comes back in a week, the session that stored the company details is unlikely to still exist, and the script fails to display the desired company's stock price.
- *Security.* Sessions can provide a way for an intruder to break into a system. Sessions can be open to hijacking; an intruder can take over after a legitimate user has logged into an application.

## Bibliography

PHP web site, <http://www.php.net>

PEAR web site, <http://www.pear.php.net>

Tim Converse, Joyce Park, Clark Morgan, PHP5 and MySQL Bible, Wiley Publishing, Inc., 2004.

David Sklar, Learning PHP 5, O'Reilly 2004.

David Lane, Hugh E. Williams, Web Database Application with PHP and MySQL, 2nd Edition, O'Reilly 2004.