69/B Monwara Plaza (Level-5)
East Panthapath
Dhaka-1205
Cell: 01197 11 82 77

Lecture Sheet on "PHP+MySQL"
Day-15: (PHP)

Web Application Security **Time: 2** hours

This course provides the learner with information about implementing pagination for a enormous length of data retrieved from database or other source.

Objectives	Topics
❖ Prevent forms from being submitted multiple times ❖ Validate form data ❖ Define patterns and both match and replace them ❖ Manage database encryption and decryption techniques	❖ Preventing multiple submissions ❖ Validating the right form ❖ Handling HTML, Validating data ❖ JavaScript form validation ❖ Defining a pattern with regular expressions ❖ Matching regular expression patterns ❖ Database security and encryption

Preventing multiple submissions:

You may have more than one form in a page and many submit buttons, than all the fileds used in that page will be posted and inserted into your database unless you properly define an effective script to prevent those unwanted multiple submissions. Moreover you could restrict the users from multiple submissions for a stipulated period by using a code snippet by recognizing the users IP. Now, the question may arise, 'how can we do that?'. Well, we can do it in several ways using PHP or JavaScript. The preferable ways are given below:

1. Checking the isset() value of submit button
2. Checking session key in cookies during each submissions
3. Checking the username whether it is already exists in database or not

```
<?php
function prevent_multi_submit($type = "post", $excl = "validator") {
   $string = "";
   foreach ($_POST as $key => $val) {
      // this test is new in version 1.01 to exclude a single variable
      if ($key != $excl) {          $string .= $val;       }
   }
   if (isset($_SESSION['last'])) {
      if ($_SESSION['last'] === md5($string)) {          return false;
      } else {          $_SESSION['last'] = md5($string);          return true;       }
   } else {       $_SESSION['last'] = md5($string);       return true;    }
}
/* example of use:
if (isset($_POST)) {
   if ($_POST['field'] != "" && strlen < 25) { // place here the form validation and other controls
      if (prevent_multi_submit()) { // use the function before you call the database
         mysql_query("INSERT INTO tabel..."); // or send a mail like...
         mail($mailto, $sub, $body);
      } else {          echo "The form is already processed";       }
   } else {       // your error about invalid fiels    }
} */?>
```

Validating the right form:

- All web applications must validate user input data
  - Ensure the data is syntactically correct, and meets input constraints
- Ideally want one PHP page to create form, validate input, and handle correct input
  - This is a little tricky

All-in-one Input Handling Page

- Overall logic:
  - If first time displaying page Then
    - Create form with initial default values
  - Else have just received user input Then
    - Validate input
    - If input not valid Then
      - Redisplay page, with just-received user input data as default values (so as not to lose the user's input)
    - Else
      - Handle the input (typically by writing it to a database)
      - Proceed to next screen in the web application

How to determine if first time

- Can check if the $_POST[] array is empty
  - Will be empty first time through
  - if (empty($_POST)) { create initial form }
  - if (!empty($_POST)) { validate input }

All-in-one structure

- Functions for input validation
- Function(s) that creates form
  - Default values set from contents of $_POST[]
- HTML preamble
- Create initial form
- Validate input
- Recreate form
- Code to handle valid input
- HTML end

Libraries

- There are many libraries that exist for handling input, and performing validation
- These libraries can also create client-side Javascript for client-side value checking
- PEAR HTML_quickform is one example

JavaScript Form Validation

JavaScript can be used to validate input data in HTML forms before sending off the content to a server. Form data that typically are checked by a JavaScript could be:

- has the user left required fields empty?
- has the user entered a valid e-mail address?
- has the user entered a valid date?
- has the user entered text in a numeric field?

The entire script, with the HTML form could look something like this:

```
<html><head><script type="text/javascript">
function validate_required(field,alerttxt){
        with (field){
        if (value==null||value=="")  { alert(alerttxt);return false;}
        else {return true}
        }
```

```
}
function validate_form(thisform){
        with (thisform){
if (validate_required(email,"Email must be filled out!")==false)   {email.focus();return false;}
}
}
</script></head>
<body>
<form action="submitpage.htm"
onsubmit="return validate_form(this)"
method="post">
Email: <input type="text" name="email" size="30">
<input type="submit" value="Submit">
</form>
</body></html>
```

E-mail Validation

The function below checks if the content has the general syntax of an email. This means that the input data must contain at least an @ sign and a dot (.). Also, the @ must not be the first character of the email address, and the last dot must at least be one character after the @ sign:

The entire script, with the HTML form could look something like this:

```
<html><head>
<script type="text/javascript">
function validate_email(field,alerttxt){
        with (field){
apos=value.indexOf("@");
dotpos=value.lastIndexOf(".");
if (apos<1||dotpos-apos<2) {alert(alerttxt);return false;}  else {return true;}  }  }
function validate_form(thisform){
        with (thisform){
if (validate_email(email,"Not a valid e-mail address!")==false)  {email.focus();return false;} }}
</script>
</head><body>
<form action="submitpage.htm" onsubmit="return validate_form(this);" method="post">
Email: <input type="text" name="email" size="30">
<input type="submit" value="Submit">
</form></body></html>
```

PHP Provides Three Sets of Regular Expression Functions

PHP is an open source language for producing dynamic web pages, similar to ASP. PHP has three sets of functions that allow you to work with regular expressions. Each set has its advantages and disadvantages.

The ereg Function Set

The ereg functions require you to specify the regular expression as a string, as you would expect. ereg('regex', "subject") checks if regex matches subject. You should use single quotes when passing a regular expression as a literal string. Several special characters like the dollar and backslash are also special characters in double-quoted PHP strings, but not in single-quoted PHP strings.
int **ereg** (string pattern, string subject [, array groups]) returns the length of the match if the regular expression pattern matches the subject string or part of the subject string, or zero otherwise. Since

zero evaluates to False and non-zero evaluates to True, you can use ereg in an if statement to test for a match. If you specify the third parameter, ereg will store the substring matched by the part of the regular expression between the first pair of round brackets in $groups[1]. $groups[2] will contain the second pair, and so on. Note that grouping-only round brackets are not supported by ereg. ereg is case sensitive. eregi is the case insensitive equivalent.

string **ereg_replace** (string pattern, string replacement, string subject) replaces all matches of the regex patten in the subject string with the replacement string. You can use backreferences in the replacement string. \\0 is the entire regex match, \\1 is the first backreference, \\2 the second, etc. The highest possible backreference is \\9. ereg_replace is case sensitive. eregi_replace is the case insensitive equivalent.

array **split** (string pattern, string subject [, int limit]) splits the subject string into an array of strings using the regular expression pattern. The array will contain the substrings between the regular expression matches. The text actually matched is discarded. If you specify a limit, the resulting array will contain at most that many substrings. The subject string will be split at most limit-1 times, and the last item in the array will contain the unsplit remainder of the subject string. split is case sensitive. spliti is the case insensitive equivalent.

THE MB_EREG FUNCTION SET

The mb_ereg functions work exactly the same as the ereg functions, with one key difference: while ereg treats the regex and subject string as a series of 8-bit characters, mb_ereg can work with multi-byte characters from various code pages. E.g. encoded with Windows code page 936 (Simplified Chinese), the word 中国 ("China") consists of four bytes: D6D0B9FA. Using the ereg function with the regular expression . on this string would yield the first byte D6 as the result. The dot matched exactly one byte, as the ereg functions are byte-oriented. Using the mb_ereg function after calling mb_regex_encoding("CP936") would yield the bytes D6D0 or the first character 中 as the result.

To make sure your regular expression uses the correct code page, call mb_regex_encoding() to set the code page. If you don't, the code page returned by or set by mb_internal_encoding() is used instead.

If your PHP script uses UTF-8, you can use the preg functions with the /u modifier to match multi-byte UTF-8 characters instead of individual bytes. The preg functions do not support any other code pages.

THE PREG FUNCTION SET

All of the preg functions require you to specify the regular expression as a string using Perl syntax. In Perl, /regex/ defines a regular expression. In PHP, this becomes preg_match('/regex/', $subject). Forward slashes in the regular expression have to be escaped with a backslash. So http://www\.jgsoft\.com/ becomes '/http:\/\/www\.jgsoft\.com\//'. Just like Perl, the preg functions allow any non-alphanumeric character as regex delimiters. The URL regex would be more readable as '%http://www\.jgsoft\.com/%' using percentage signs as the regex delimiters.

Unlike programming languages like C# or Java, PHP does not require all backslashes in strings to be escaped. If you want to include a backslash as a literal character in a PHP string, you only need to escape it if it is followed by another character that needs to be escaped. In single quoted-strings, only the single quote and the backslash itself need to be escaped. That is why in the above regex, I didn't have to double the backslashes in front of the literal dots. The regex \\ to match a single backslash would become '/\\\\/' as a PHP preg string. Unless you want to use variable interpolation in your regular expression, you should always use single-quoted strings for regular expressions in

PHP, to avoid messy duplication of backslashes.

To specify regex matching options such as case insensitivity are specified in the same way as in Perl. '/regex/i' applies the regex case insensitively. '/regex/s' makes the dot match all characters. '/regex/m' makes the start and end of line anchors match at embedded newlines in the subject string. '/regex/x' turns on free-spacing mode. You can specify multiple letters to turn on several options. '/regex/misx' turns on all four options.

A special option is the /u which turns on the Unicode matching mode, instead of the default 8-bit matching mode. You should specify /u for regular expressions that use \x{FFFF}, \X or \p{L} to match Unicode characters, graphemes, properties or scripts. PHP will interpret '/regex/u' as a UTF-8 string rather than as an ASCII string.

Like the ereg function, bool **preg_match** (string pattern, string subject [, array groups]) returns TRUE if the regular expression pattern matches the subject string or part of the subject string. If you specify the third parameter, preg will store the substring matched by the first capturing group in $groups[1]. $groups[2] will contain the second pair, and so on. If the regex pattern uses named capture, you can access the groups by name with $groups['name']. $groups[0] will hold the overall match.

int **preg_match_all** (string pattern, string subject, array matches, int flags) fills the array "matches" with all the matches of the regular expression pattern in the subject string. If you specify PREG_SET_ORDER as the flag, then $matches[0] is an array containing the match and backreferences of the first match, just like the $groups array filled by preg_match. $matches[1] holds the results for the second match, and so on. If you specify PREG_PATTERN_ORDER, then $matches[0] is an array with full subsequent regex matches, $matches[1] an array with the first backreference of all matches, $matches[2] an array with the second backreference of each match, etc.

array **preg_grep** (string pattern, array subjects) returns an array that contains all the strings in the array "subjects" that can be matched by the regular expression pattern.

mixed **preg_replace** (mixed pattern, mixed replacement, mixed subject [, int limit]) returns a string with all matches of the regex pattern in the subject string replaced with the replacement string. At most limit replacements are made. One key difference is that all parameters, except limit, can be arrays instead of strings. In that case, preg_replace does its job multiple times, iterating over the elements in the arrays simultaneously. You can also use strings for some parameters, and arrays for others. Then the function will iterate over the arrays, and use the same strings for each iteration. Using an array of the pattern and replacement, allows you to perform a sequence of search and replace operations on a single subject string. Using an array for the subject string, allows you to perform the same search and replace operation on many subject strings.

preg_replace_callback (mixed pattern, callback replacement, mixed subject [, int limit]) works just like preg_replace, except that the second parameter takes a callback instead of a string or an array of strings. The callback function will be called for each match. The callback should accept a single parameter. This parameter will be an array of strings, with element 0 holding the overall regex match, and the other elements the text matched by capturing groups. This is the same array you'd get from preg_match. The callback function should return the text that the match should be replaced with. Return an empty string to delete the match. Return $groups[0] to skip this match.

Callbacks allow you to do powerful search-and-replace operations that you cannot do with regular expressions alone. E.g. if you search for the regex (\d+)\+(\d+), you can replace 2+3 with 5 using the callback:

```
function regexadd($groups) {    return $groups[1] + $groups[2]; }
```

69/B Monwara Plaza (Level-5)
East Panthapath
Dhaka-1205
Cell: 01197 11 82 77

array **preg_split** (string pattern, string subject [, int limit]) works just like split, except that it uses the Perl syntax for the regex pattern.

Validating E-Mail with PHP without REGEX

```
<?php
function email_is_valid($email) {
  if (substr_count($email, '@') != 1)       return false;
  if ($email{0} == '@')      return false;
  if (substr_count($email, '.') < 1)       return false;
  if (strpos($email, '..') !== false)      return false;
  $length = strlen($email);
  for ($i = 0; $i < $length; $i++) {
    $c = $email{$i};
    if ($c >= 'A' && $c <= 'Z')          continue;
    if ($c >= 'a' && $c <= 'z')          continue;
    if ($c >= '0' && $c <= '9')          continue;
    if ($c == '@' || $c == '.' || $c == '_' || $c == '-')          continue;
    return false;
  }
  $TLD = array ('COM',   'NET',  'ORG',   'MIL', 'EDU',   'GOV', 'BIZ',   'NAME',
       'MOBI',  'INFO', 'AERO',  'JOBS', 'MUSEUM');
  $tld = strtoupper(substr($email, strrpos($email, '.') + 1));
  if (strlen($tld) != 2 && !in_array($tld, $TLD))      return false;
  return true;
}
?>
```

69/B Monwara Plaza (Level-5)
East Panthapath
Dhaka-1205
Cell: 01197 11 82 77

Regular expressions

If you try to create a checker function with string manipulation routines such as strlen, strpos,… then it will result in a quite complicated if-else condition structure.
An other solution is using regular expressions. In this case you should define a pattern which should fit on the relevant string. For example: pattern is '^Test' and the subject is 'Test string'. This will pass as the pattern is valid for all strings which begins with the substring 'Test'.

The pattern for the email validation is a little bit more complicated.

^[_a-z0-9-]+(\.[_a-z0-9-]+)*@[a-z0-9-]+(\.[a-z0-9-]+)*(\.[a-z]{2,3})

Hoop's it is really quite complex. I try to explain it. Let's divide the pattern into smaller parts. The first part is until the '@' character.

^[_a-z0-9-]+(\.[_a-z0-9-]+)*

Explanation:
-- ^ means that we start the check from the first character of the string.
-- [_a-z0-9-]+: There must be at least 1 character between a and z or between 0-9 or '_' .
-- (\.[_a-z0-9-]+)* : The first character group will be followed by 0 or more character groups which always begins with a '.'

Now try to interpret the second part: **@[a-z0-9-]+(\.[a-z0-9-]+)*(\.[a-z]{2,3})$**

Explanation:
-- The '@' means that after the first part this character is mandatory exactly once.
-- [_a-z0-9-]+: As before
-- (\.[_a-z0-9-]+)*: As before
-- (\.[a-z]{2,3})$: This means that at the end of the email string there must be a 2 or 3 character long substring. And before it a '.' is mandatory.

Implement in PHP
PHP has some built in function to support regular expressions. Now we just use one of them the eregi.
Usage : bool eregi (string pattern, string string [, array regs]);

A new function was created to test the email string whether it is valid or not. Later you can just call this function. It returns true if the email is valid and false otherwise.

Let's see the code:

```
<?php
  // This function tests whether the email address is valid
  function isValidEmail($email){
    $pattern = "^[_a-z0-9-]+(\.[_a-z0-9-]+)*@[a-z0-9-]+(\.[a-z0-9-]+)*(\.[a-z]{2,3})$";

    if (eregi($pattern, $email)){       return true;     }
    else {       return false;     }
  }
?>
```

It' quite easy, isn't it?
Now we have the validation code. Let's make a small test form to see it in action:

```
<?php
  // This function tests whether the email address is valid
```

```
  function isValidEmail($email){
    $pattern = "^[_a-z0-9-]+(\.[_a-z0-9-]+)*@[a-z0-9-]+(\.[a-z0-9-]+)*(\.[a-z]{2,3})$";
    if (eregi($pattern, $email)){          return true;      }
    else {         return false;      }
  }
?>
<head>  <title>Email validation form</title></head>
<body>  <form action="<?php echo $_SERVER['PHP_SELF']; ?>" method="post" name="emailForm">
    <table>
     <tr><td>Email:<input name="email"></td><tr>
     <?php
      if (isset($_POST['submitemail']))        {
        if (isValidEmail($_POST['email'])){  echo "<tr><td>The email: ".$_POST['email']." is valid!</td></tr>";
 }
        else{              echo "<tr><td>The email: ".$_POST['email']." is invalid!</td></tr>";           }
      }
     ?>
     <tr><td><input type="submit" name="submitemail" value="Validate email"></td></tr>
    </table>  </form></body>
```

The patterns used in pattern matching are regular expressions such as those supplied in the Version 8 regex routines. (In fact, the routines are derived (distantly) from Henry Spencer's freely redistributable reimplementation of the V8 routines.).
In particular the following metacharacters have their standard *egrep*-ish meanings:

\ Quote the next metacharacter
^ Match the beginning of the line
. Match any character (except newline)
$ Match the end of the line (or before newline at the end)
| Alternation
() Grouping
[] Character class

The following standard quantifiers are recognized:

* Match 0 or more times
\+ Match 1 or more times
? Match 1 or 0 times
{n} Match exactly n times
{n,} Match at least n times
{n,m} Match at least n but not more than m times