Web Development with PHP
Practical - 6
Session Management, File handling and Error handling

PHP Sessions

Sessions are an essential component of web development as they allow developers to handle user-specific data and maintain the state of an application across multiple page requests. When a user interacts with a website, their actions and data need to be stored and accessible throughout their browsing session. PHP offers robust built-in features for managing sessions efficiently. Sessions in PHP are a vital aspect of web development, playing a crucial role in storing and managing user-related data. These sessions are created uniquely for each user when they interact with a website, allowing the server to track their actions effectively. This functionality is especially important for user authentication, as sessions enable the secure storage of login credentials. Consequently, users can seamlessly navigate through different pages while remaining authenticated throughout their browsing session.

Sessions serve various purposes in web applications. Some key uses of sessions include:

- User Authentication: Session in PHP plays a vital role in authenticating users. When a user logs in, their credentials are securely stored in a session variable. This session variable is then utilized to verify the user's authentication on subsequent requests, ensuring their identity and access rights.
- User Tracking: Sessions play a crucial role in tracking user activity on a website. They provide developers with valuable insights into user behavior by capturing essential data such as the pages visited, actions performed, and session duration. This information allows developers to better understand how users engage with their website and make informed decisions based on these observations.
- Security: By storing sensitive user information, such as login credentials or access tokens, in session variables, developers can ensure that this data is not exposed directly in the URL or HTML form fields. This helps prevent unauthorized access to sensitive data and protects against common attacks like session hijacking or session fixation.
- Data Persistence: Session in PHP provides a way to persist data across multiple page requests. Without sessions, every time a user interacts with a website, the server would have no memory of previous interactions. With sessions, data can be stored and accessed throughout the user's browsing session, allowing for a more dynamic and personalized experience.
- Language Localization: Sessions are a powerful tool in PHP applications for incorporating language localization. By utilizing a session variable to store the user's preferred language, websites can effortlessly present content in the language that is most suitable for each individual user. This feature proves to be exceptionally beneficial for websites catering to a global audience or offering multilingual support.
- Shopping Carts: Session in PHP is a valuable tool that enhances the shopping cart experience, offering users a seamless and customized journey. By utilizing sessions, individuals can effortlessly manage and monitor the products they have added to their carts, granting them the freedom to navigate the website without fear of losing their carefully chosen items.
- Personalization: Sessions allow for personalized user experiences. By using sessions, websites can store valuable user preferences and settings, enabling them to deliver tailor-made content and customize the user interface according to individual needs.

- Error Handling: Sessions can help in error handling and debugging. Developers can store error messages or diagnostic information in session variables, making it easier to track and investigate issues. This can be especially helpful in situations where errors occur across multiple page requests, as the session data persists and can be accessed for troubleshooting purposes.
- Integration with External APIs: Sessions can be utilized to integrate PHP applications with external APIs. For example, an e-commerce website may store the API access token in a session variable, allowing it to make authenticated requests to a payment gateway or shipping service on behalf of the user.

When a session in PHP is started, the following steps take place:

- Session ID Generation:

  PHP generates a unique 32 digit hexadecimal session ID for each user. This identifier serves as a crucial component in identifying and managing the associated session data. For example, the session ID might appear as 1234ac64a423rf2d123e123d648ce123, representing a randomly generated and secure identifier for a specific session.

- Session ID Storage:

  This unique identifier is securely stored in a cookie named PHPSESSID on the user's browser. By sending this cookie back and forth between the browser and the server with each request, our system can seamlessly identify and associate the corresponding session, allowing for a seamless and personalized user experience.

- Session Data Storage:

  The session data is typically stored on the server-side. In PHP, the default practice involves using a temporary directory on the server to save the session files. However, there is flexibility to customize this and opt for alternative storage mechanisms like databases. Each session file is named using a hexadecimal session ID, with the prefix sess_. For instance, if we consider the previously mentioned session ID as 1234ac64a423rf2d123e123d648ce123, it would be stored in a file named sess_1234ac64a423rf2d123e123d648ce123.

- Session Data Retrieval:

  When a user sends a subsequent request, PHP retrieves the session ID from the cookie transmitted by the user's browser. Subsequently, it retrieves the corresponding session data from the server and provides it to the PHP script. This mechanism ensures the seamless accessibility and utilization of the user's session information within the PHP application.

Example:

```
<?php
     session_start();

     $_SESSION['username'] = ABC';
     $_SESSION['email'] = 'abc@gmail.com';
?>

<html>
<body>
    <p>Username: <?php echo $_SESSION['username']; ?></p>
    <p>Email: <?php echo $_SESSION['email']; ?></p>
</body>
</html>

//-----------------------------------------------------
//Counter
<?php
     session_start();

     if(isset($_SESSION['counter'])){
         $_SESSION['counter']++;
     } else {
         $_SESSION['counter'] = 1;
     }

     echo "You have visited this page ".$_SESSION['counter']."
     time(s).";
?>
```

Sessions Without Cookies

PHP commonly uses cookies as the default method for tracking and identifying sessions. However, there are situations where users may choose to limit or disable cookies in their web browsers, which poses challenges in session management. In such cases, an alternative approach can be utilized to effectively handle sessions without relying on cookies.

One potential solution involves utilizing the SID constant. This constant is automatically initialized at the start of a session. When users consent to the use of cookies, the SID value remains an empty string. Conversely, if users decline the use of cookies, the SID is presented in the format session_name=session_id. This format can be employed to successfully register and store variables, even when cookies are not available.

```php
<?php
    session_start();

    if (empty($_COOKIE['PHPSESSID'])) {
        $sid = session_name() . '=' . session_id();
    } else {
        $sid = '';
    }

    // Use the $sid value to embed the session identifier
    in forms or URLs
    echo '<a href="page.php?' . $sid . '">Link</a>';

    //Alternate is can be hidden fields
?>
```

Security Considerations

Session Hijacking
Session hijacking occurs when an attacker gains unauthorized access to a user's session. We can mitigate session hijacking by enabling secure session handling (setting session.cookie_secure to true) and regularly regenerating session IDs using session_regenerate_id().

Session Fixation
Session fixation involves an attacker forcing a user to use a predetermined session ID, enabling them to impersonate the user. We can prevent session fixation by generating new session IDs upon user authentication or privilege level changes and destroying the old session after regeneration.

Securing Session Data

- Use appropriate data sanitization and validation techniques to prevent injection attacks or malformed session data.
- Encrypt sensitive session data using PHP's encryption functions or secure encryption libraries.
- Avoid storing sensitive information in sessions unless necessary. Instead, consider storing minimal user data and validating it against the server's database when required.

Predefined Constants

- SID (string)
  Constant contains either the session name and session ID in the form of "name=ID" or empty string if the session ID was set in an appropriate session cookie. This is the same id as the one returned by session_id().
- PHP_SESSION_DISABLED (int)
  Return value of session_status() if sessions are disabled.
- PHP_SESSION_NONE (int)
  Return value of session_status() if sessions are enabled, but no session exists.
- PHP_SESSION_ACTIVE (int)
  Return value of session_status() if sessions are enabled, and a session exists.

Session Functions

- session_abort — Discard session array changes and finish session
- session_cache_expire — Get and/or set current cache expire
- session_cache_limiter — Get and/or set the current cache limiter
- session_commit — Alias of session_write_close
- session_create_id — Create new session id
- session_decode — Decodes session data from a session encoded string
- session_destroy — Destroys all data registered to a session
- session_encode — Encodes the current session data as a session encoded string
- session_gc — Perform session data garbage collection
- session_get_cookie_params — Get the session cookie parameters
- session_id — Get and/or set the current session id
- session_module_name — Get and/or set the current session module
- session_name — Get and/or set the current session name
- session_regenerate_id — Update the current session id with a newly generated one
- session_register_shutdown — Session shutdown function
- session_reset — Re-initialize session array with original values
- session_save_path — Get and/or set the current session save path
- session_set_cookie_params — Set the session cookie parameters
- session_set_save_handler — Sets user-level session storage functions
- session_start — Start new or resume existing session
- session_status — Returns the current session status
- session_unset — Free all session variables
- session_write_close — Write session data and end session

File Handling in PHP

File handling in PHP refers to the various functions and methods available in the language that enable developers to read, write, manipulate, and manage files and directories on a server or local machine.

PHP provides several built-in functions like open (), fwrite(), fread(), fclose(), and others to manipulate files in different modes like read, write, append, binary, etc.

PHP also provides functions to manage directories, such as opendir() to open a directory, readdir() to read the contents of a directory and closedir() to close a directory. These functions enable developers to create, move, rename, delete, and manage files and directories.

File handling is a crucial aspect of programming that involves creating, reading, writing, and deleting files. In PHP, file handling plays a vital role in web development, especially when dealing with tasks such as data storage, retrieval, and manipulation. Understanding file handling in PHP requires knowledge of the functions and techniques available to effectively perform these tasks.

File handling starts with creating a file, reading its content, writing into a file to appending data into an existing file and finally closing the file.

- Create a File: fopen()
- Open a File: fopen()
- Read a File: fread()
- Write to a File: fwrite()
- Append to a File: fwrite()
- Close a File: fclose()
- Delete a File: unlink()

By changing one or more arguments, the same function can be used to perform multiple operations on file.

fopen() Function

The fopen() function in PHP is used to open a file or URL and returns a file pointer resource that can be used to read, write, or manipulate the file. When we use the function fopen() to open a file which doesn't exist then that file gets created. Technically, file is not opened, fopen() binds the resource(file) to a stream, which can then be used to read from the file or write data to the file.

```
resource fopen ( string $filename , string $mode [, bool
$use_include_path = false [, resource $context ]] )
```

- ➔ $filename is the name of the file or URL to open
- ➔ $mode is the mode in which to open the file. This can be one of the following:
  - ◆ 'r': read only
  - ◆ 'w': write only (truncates the file to zero length or creates a new file)
  - ◆ 'a': append-only (opens the file for writing at the end of the file)
  - ◆ 'x': exclusive write (creates a new file and opens it for writing only if it doesn't

already exist)
    - 'b': binary mode (used in conjunction with the above modes to indicate that the file should be opened in binary mode)
    - 't': text mode (used in conjunction with the above modes to indicate that the file should be opened in text mode)
- ➔ $use_include_path is a boolean parameter that indicates whether to search for the file in the include path (if set)
- ➔ $context is an optional parameter that allows you to specify a context for the file stream (e.g. HTTP headers, SSL settings, etc.)

Example:

```
$file = fopen('example.txt', 'r');

if ($file) {
    while (($line = fgets($file)) !== false) {
        echo $line;
    }
    fclose($file);
}

//--------------------------------------------

$myfile = 'file.txt';
//opens the file.txt file or implicitly creates the file
$handle = fopen($myfile, 'w') or die('Cannot open file: '.$myfile);
```

Mode	Description
r	Open file in **read-only** mode. It places the file pointer at the beginning of the file.
r+	Open file in **read-write** mode. It places the file pointer at the beginning of the file.
w	Open file in **write-only** mode. It places the file pointer to the beginning of the file and truncates the file to zero length. If a file is not found, it creates a new file.
w+	Open file in **read-write** mode. It places the file pointer to the beginning of the file and truncates the file to zero length. If a file is not found, it creates a new file.
a	Open file in **write-only** mode. It places the file pointer to the end of the file. If a file is not found, it creates a new file.
a+	Open file in **read-write** mode. It places the file pointer to the end of the file. If a file is not found, it creates a new file.

x	Creates and opens file in **write-only** mode. It places the file pointer at the beginning of the file. If a file is found, fopen() function returns FALSE.
x+	It is the same as x but it creates and opens files in **read-write** mode.
c	Open file in **write-only** mode. If the file does not exist, it is created. If it exists, it is neither truncated (as opposed to 'w'), nor the call to this function fails (as is the case with 'x'). The file pointer is positioned on the beginning of the file
c+	It is the same as c but it opens the file in **read-write** mode.

fclose() function

It is good practice to close a file resource after using it. In PHP, the fclose() function is used to close a file resources. It takes the file name as argument or the variable holding the file resource pointer as argument.

```
$myfile = 'file.txt';
//opens the file.txt file or implicitly creates the file
$handle = fopen($myfile, 'w') or die('Cannot open file:
'.$myfile);

// closing a file
fclose($handle);
```

fread() Function

The PHP fread() function is used to read a specified number of bytes from a file. It is a file-handling function in PHP that allows you to read data from a file.

```
fread($file_handle, $length)
```

- $file_handle: Required. Specifies the file pointer or file handle to read from.
- $length: Required. Specifies the number of bytes to read from the file.
- Other read functions
    - fgets()
      string fgets (resource $handle [, int $length])
    - fgetc()
      string fgetc (resource $handle)

fwrite() Function

The PHP fwrite() function is used to write data to a file. It takes three parameters: a file pointer, a string

to be written, and an optional parameter that specifies the maximum number of bytes to write.

```
fwrite ( resource $handle , string $string [, int $length ] ) : int|false
```

- resource $handle: A file pointer, which is a reference to the opened file. It can be obtained using functions such as fopen(), fsockopen(), and tmpfile().
- string $string: The string to be written to the file.
- int $length: An optional parameter that specifies the maximum number of bytes to be written. If this parameter is not specified, all the data in the string will be written to the file.
- The fwrite() function returns the number of bytes written to the file, or false on error.

unlink()

In PHP, the unlink() function is used to delete a file from the file system. It accepts a single argument, which is the path to the file that needs to be deleted.

```
bool unlink ( string $filename [, resource $context ] )
```

End-Of-File - feof()

The feof() function checks if the "end-of-file" (EOF) has been reached. The feof() function is useful for looping through data of unknown length.

File System Functions

Function	Description
basename()	Returns the filename component of a path
chgrp()	Changes the file group
chmod()	Changes the file mode
chown()	Changes the file owner
clearstatcache()	Clears the file status cache
copy()	Copies a file
delete()	- unlink()
dirname()	Returns the directory name component of a path
disk_free_space()	Returns the free space of a filesystem or disk
disk_total_space()	Returns the total size of a filesystem or disk
diskfreespace()	Alias of disk_free_space()
fclose()	Closes an open file
feof()	Checks if the "end-of-file" (EOF) has been reached for an open file
fflush()	Flushes buffered output to an open file
fgetc()	Returns a single character from an open file
fgetcsv()	Returns a line from an open CSV file
fgets()	Returns a line from an open file
fgetss()	Deprecated from PHP 7.3. Returns a line from an open file - stripped from HTML and PHP tags
file()	Reads a file into an array
file_exists()	Checks whether or not a file or directory exists
file_get_contents()	Reads a file into a string
file_put_contents()	Writes data to a file

fileatime()	Returns the last access time of a file
filectime()	Returns the last change time of a file
filegroup()	Returns the group ID of a file
fileinode()	Returns the inode number of a file
filemtime()	Returns the last modification time of a file
fileowner()	Returns the user ID (owner) of a file
fileperms()	Returns the file's permissions
filesize()	Returns the file size
filetype()	Returns the file type
flock()	Locks or releases a file
fnmatch()	Matches a filename or string against a specified pattern
fopen()	Opens a file or URL
fpassthru()	Reads from the current position in a file - until EOF, and writes the result to the output buffer
fputcsv()	Formats a line as CSV and writes it to an open file
fputs()	Alias of fwrite()
fread()	Reads from an open file (binary-safe)
fscanf()	Parses input from an open file according to a specified format
fseek()	Seeks in an open file
fstat()	Returns information about an open file
ftell()	Returns the current position in an open file
ftruncate()	Truncates an open file to a specified length
fwrite()	Writes to an open file (binary-safe)
glob()	Returns an array of filenames / directories matching a specified pattern
is_dir()	Checks whether a file is a directory

is_executable()	Checks whether a file is executable
is_file()	Checks whether a file is a regular file
is_link()	Checks whether a file is a link
is_readable()	Checks whether a file is readable
is_uploaded_file()	Checks whether a file was uploaded via HTTP POST
is_writable()	Checks whether a file is writable
is_writeable()	Alias of is_writable()
lchgrp()	Changes the group ownership of a symbolic link
lchown()	Changes the user ownership of a symbolic link
link()	Creates a hard link
linkinfo()	Returns information about a hard link
lstat()	Returns information about a file or symbolic link
mkdir()	Creates a directory
move_uploaded_file()	Moves an uploaded file to a new location
parse_ini_file()	Parses a configuration file
parse_ini_string()	Parses a configuration string
pathinfo()	Returns information about a file path
pclose()	Closes a pipe opened by popen()
popen()	Opens a pipe
readfile()	Reads a file and writes it to the output buffer
readlink()	Returns the target of a symbolic link
realpath()	Returns the absolute pathname
realpath_cache_get()	Returns realpath cache entries
realpath_cache_size()	Returns realpath cache size

rename()	Renames a file or directory
rewind()	Rewinds a file pointer
rmdir()	Removes an empty directory
set_file_buffer()	Alias of stream_set_write_buffer(). Sets the buffer size for write operations on the given file
stat()	Returns information about a file
symlink()	Creates a symbolic link
tempnam()	Creates a unique temporary file
tmpfile()	Creates a unique temporary file
touch()	Sets access and modification time of a file
umask()	Changes file permissions for files
unlink()	Deletes a file

File Upload

PHP allows you to upload single and multiple files through a few lines of code only. PHP file upload features allow you to upload binary and text files both. Moreover, you can have full control over the file to be uploaded through PHP authentication and file operation functions.

A PHP script can be used with a HTML form to allow users to upload files to the server. Initially files are uploaded into a temporary directory and then relocated to a target destination by a PHP script.

Information in the phpinfo.php page describes the temporary directory that is used for file uploads as upload_tmp_dir and the maximum permitted size of files that can be uploaded is stated as upload_max_filesize. These parameters are set into PHP configuration file php.ini

The process of uploading a file follows these steps −

- The user opens the page containing a HTML form featuring text files, a browse button and a submit button.
- The user clicks the browse button and selects a file to upload from the local PC.
- The full path to the selected file appears in the text field then the user clicks the submit button.
- The selected file is sent to the temporary directory on the server.
- The PHP script that was specified as the form handler in the form's action attribute checks that the file has arrived and then copies the file into an intended directory.
- The PHP script confirms the success to the user.

$_FILES

The PHP global $_FILES contains all the information of the file. By the help of $_FILES global, we can get file name, file type, file size, temp file name and errors associated with file.

- $_FILES['filename']['name'] - returns file name.
- $_FILES['filename']['type'] - returns MIME type of the file.
- $_FILES['filename']['size'] - returns size of the file (in bytes).
- $_FILES['filename']['tmp_name'] - returns temporary file name of the file which was stored on the server.
- $_FILES['filename']['error'] - returns error code associated with this file.

move_uploaded_file() function

The move_uploaded_file() function moves the uploaded file to a new location. The move_uploaded_file() function checks internally if the file is uploaded through the POST request. It moves the file if it is uploaded through the POST request.

Types of Errors in PHP

Like any programming language, errors can occur while writing and executing PHP code. Understanding these errors is essential for PHP developers as it can help them debug and fix issues quickly.

In PHP, errors are divided into three types: syntax errors, runtime errors, and logical errors.

Syntax errors occur when the code violates the rules of the PHP syntax, while runtime errors occur during the execution of the code. Logical fallacies, on the other hand, are not related to syntax or runtime, but occur when the code does not behave as expected.

- Syntax errors:
  Syntax errors are caused by mistakes in the code syntax. These errors occur when the PHP parser encounters an unexpected or incorrect statement or expression. For example, a missing semicolon or a misspelled function name can cause a syntax error.
- Runtime errors:
  Runtime errors occur during the execution of the code. These errors are usually caused by factors such as incorrect function arguments, incorrect variable types, or memory-related issues. Examples of runtime errors include division by zero, undefined function calls, or array out-of-bounds errors.
- Logical errors:
  Logical errors occur when the code does not behave as expected, even though it is syntactically and semantically correct. These errors are usually caused by incorrect or incomplete logic in the code.

Syntax Error or Parse Error

A syntax error, also known as a parse error, is an error in the PHP code that violates the language's syntax rules. Syntax errors can prevent the PHP code from executing, as they are detected by the PHP parser during the compilation stage.

```
<?php
  $number = 5
  echo "The number is: $number";
?>
```

Fatal Error

In PHP, a fatal error is a type of error that causes the script to terminate immediately. It is usually caused by a serious coding mistake, such as an undefined function call, or a syntax error that prevents the script from running.

```
<?php
// Trying to call an undefined function
my_function(); // This will cause a fatal error
?>
```

Fatal errors can be challenging to debug, as they do not provide any information about the error's cause or location. Therefore, it is essential to write code that is free of syntax errors and uses defined functions and variables correctly to avoid fatal errors.

Warning Errors

In PHP, warning errors are a type of runtime error that occurs when the code is executed, but a problem is encountered that does not cause the code to stop executing. Warning errors are less severe than fatal errors, but they should not be ignored as they can indicate potential issues that may cause problems later on. A common example of a warning error is using an undefined variable. For instance, consider the following code:

```
<?php
     echo $variable;
?>
```

To handle warning errors in PHP, developers can use error reporting functions such as error_reporting() and ini_set() to control the error level and output.

Notice Error

A Notice error is a type of runtime error in PHP that occurs when the code attempts to use a variable that has not been defined or assigned a value. This can happen when a variable is used before it is initialized or misspelled.

```
<?php
     $variable;
     echo $variable;
?>
```

PHP Error Constants

PHP defines several error constants that can be used to identify and handle different types of errors. These constants are defined in the E_prefix format, followed by a descriptive name. Here are some of the most commonly used error constants in PHP:

- E_ERROR: This constant indicates a fatal error that causes the script to terminate immediately. These errors are caused by serious issues such as undefined functions, maximum memory allocation reached, or syntax errors.
- E_WARNING: This constant indicates a non-fatal error that does not cause the script to terminate. Warnings are caused by issues such as using an undefined variable, accessing a file that does not exist, or exceeding the maximum execution time.
- E_NOTICE: This constant indicates a runtime notice error that occurs when a variable is used before it is initialized or when an undefined index is used in an array. Notices are less severe than warnings and do not cause the script to terminate.
- E_PARSE: This constant indicates a parsing error that occurs when the code violates the syntax rules. This error is usually caused by a missing semicolon, unmatched braces, or undefined constants.

PHP Error Reporting

PHP error reporting is a crucial aspect of developing and maintaining PHP applications. It allows developers to identify and resolve issues by providing detailed information about errors, warnings, and notices that occur during script execution. By enabling error reporting, developers gain insights into runtime errors, deprecated functions, undefined variables, and other potential issues. PHP offers various error reporting levels, such as displaying errors on the screen, logging them to a file, or suppressing them altogether.

As a server-side scripting language, PHP is widely used for creating dynamic websites and web applications. However, during the development and deployment stages, errors can occur, ranging from syntax mistakes to logical flaws, which can disrupt the proper execution of PHP code.

Error reporting in PHP involves the identification, reporting, and handling of these errors to maintain the reliability and efficiency of the application. PHP provides a comprehensive error reporting system that offers developers valuable insights into the nature and location of errors. By enabling error reporting, developers can receive detailed error messages, warnings, and notices, which assist in debugging and troubleshooting the code.

The error reporting system in PHP allows developers to set different levels of error reporting, ranging from displaying all errors to suppressing them entirely. This flexibility enables developers to customize the error reporting process according to the requirements of their projects. By properly configuring error reporting, developers can promptly identify issues, eliminate bugs, and enhance the overall quality of the PHP code.

Furthermore, error reporting in PHP plays a vital role in ensuring the security of web applications. It helps developers identify vulnerabilities and potential security risks by providing detailed error messages. By analyzing these messages, developers can address security flaws, such as information disclosure, which may occur due to improper error handling. PHP error reporting is a critical aspect of web development that aids in identifying and rectifying errors, enhancing the stability, performance, and security of PHP-based applications. With its comprehensive error reporting system, PHP empowers developers to efficiently debug their code, thereby delivering robust and reliable web solutions.

Parameters of PHP Error_reporting

The PHP error_reporting function allows developers to control the level and types of errors, warnings, and notices that are displayed or logged during the execution of PHP scripts. By specifying different parameters, developers can fine-tune the error-reporting behavior according to their requirements.

- Error Levels:
    - E_ALL:
      Reports all types of errors, warnings, and notices.
    - E_ERROR:
      Reports fatal errors that halt script execution.
    - E_WARNING:
      Reports non-fatal runtime warnings.
    - E_NOTICE:
      Reports notices about potential issues or suggestions for improvement.

    - E_DEPRECATED:
      Reports usage of deprecated features that may be removed in future PHP versions.

- Error Reporting Modes:
    - error_reporting(0):
      Disables error reporting altogether.
    - error_reporting(E_ALL):
      Enables reporting of all errors, warnings, and notices.
    - error_reporting(E_ALL & ~E_NOTICE):
      Reports all errors except notices.

- Error Logging:
    - error_reporting(E_ALL);:
      Sends error messages to the server's error log file.
    - error_reporting(E_ALL); ini_set('display_errors', 1);:
      Displays errors on the screen during script execution.

- Error Display:
    - display_errors:
      Determines whether errors are displayed on the screen or not. Set to On or Off.
    - html_errors:
      Specifies whether error messages are formatted as HTML or plain text.

- Custom Error Handling:
    - set_error_handler():
      Allows developers to define a custom error handler function to handle errors instead of the default behavior.

The PHP error_reporting function not only allows developers to set the level of error reporting but also provides return values that indicate the current error reporting level. These return values can be useful for checking and modifying error-reporting settings within PHP scripts. possible return values of the error_reporting function are:

- Integer Values:0:
  Indicates that error reporting is disabled. No errors, warnings, or notices will be displayed or logged.

- E_ERROR, E_WARNING, E_PARSE, E_NOTICE, E_CORE_ERROR, E_CORE_WARNING, E_COMPILE_ERROR, E_COMPILE_WARNING, E_USER_ERROR, E_USER_WARNING, E_USER_NOTICE, E_RECOVERABLE_ERROR, E_DEPRECATED, E_USER_DEPRECATED:
  These integer values correspond to specific error levels defined in PHP. They indicate the active error reporting level.
    - A combination of error level constants can be used as a bitmask value to represent multiple error levels. For example, E_ALL represents all error levels, and E_ALL & ~E_NOTICE represents all error levels except notices.

Checking Current Error Reporting Level:
Developers can use the return value of error_reporting to check the current error reporting level and perform conditional actions based on that information. For example, they can modify error handling behavior or adjust error display settings based on the active error reporting level.

Modifying Error Reporting Settings:
The return value can be stored in a variable and used to restore error reporting settings to their original state after modifying them temporarily. This ensures that the script maintains the desired error-reporting configuration throughout its execution.

Example of PHP Error_reporting

```
//Specify Different Error Level Reporting
error_reporting(E_ALL);
ini_set('display_errors', 1);

// Example code that generates different types of errors
echo $undefinedVariable; // Notice: Undefined variable

$number = "not a number";
echo $number + 10; // Warning: A non-numeric value encountered

require 'nonexistent_file.php';
require(): Failed opening required 'nonexistent_file.php'

// Disable error reporting for notices
error_reporting(E_ALL & ~E_NOTICE);

// Example code that generates a notice
echo $undefinedVariable;

// Disable error reporting altogether
error_reporting(0);

// Example code that generates an error
echo $undefinedVariable;
```

In this example, first enable error reporting for all types of errors using error_reporting(E_ALL). Additionally, set ini_set('display_errors', 1) to display the errors on the screen during script execution.

Then demonstrate code that generates different types of errors: a notice (undefined variable), a warning (non-numeric value encountered), and a fatal error (failed opening a nonexistent file). Run the above code in your editor for a better and clear explanation.

Example:

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

// Example code that generates different types of errors
echo $undefinedVariable; // Notice: Undefined variable

$number = "not a number";
echo $number + 10; // Warning: A non-numeric value encountered

require 'nonexistent_file.php';
require(): Failed opening required 'nonexistent_file.php'

// Disable error reporting for notices
error_reporting(E_ALL & ~E_NOTICE);
echo $undefinedVariable;

// Disable error reporting altogether
error_reporting(0);
echo $undefinedVariable;
```

- Enable error reporting for all types of errors using error_reporting(E_ALL).
- Set ini_set('display_errors', 1) to display the errors on the screen during script execution.
- Then demonstrate code that generates different types of errors: a notice (undefined variable), a warning (non-numeric value encountered), and a fatal error (failed opening a nonexistent file).

Important Points of Error_reporting()

- Error Reporting Levels:
  The error_reporting() function is used to set the level of error reporting in PHP. It allows you to specify which types of errors should be reported.

- Error Reporting Constants:
  PHP provides predefined constants that can be used with error_reporting() to specify the desired error reporting level. Some commonly used constants include E_ALL (report all types of errors), E_ERROR (report only fatal errors), and E_WARNING (report only warnings).

- Error Reporting Configuration:
  The error_reporting() function can be called at runtime to change the error reporting level dynamically. This is useful when you want to enable or disable error reporting for specific parts of your code.

- Displaying Errors:
  By default, PHP displays errors to the browser when they occur. However, you can control the display of errors using the display_errors directive in the PHP configuration file or by calling the ini_set() function.

- Logging Errors:
  In addition to displaying errors, you can also configure PHP to log errors to a specified file using the error_log directive in the PHP configuration file or by calling the ini_set() function.

- Error Reporting in Production:
  In a production environment, it is generally recommended to disable error reporting and log errors instead. This helps to prevent sensitive information from being displayed to the users and provides a more controlled way of handling errors.

- Debugging and Development:
  During the development and debugging phase, it is often beneficial to enable full error reporting to catch and fix any issues in the code.

Ways to handle errors

- Handling error in code using conditional statements or die() function
- Using Custom Error Handlers
- PHP error reporting

<u>die() function</u>

- The die() function in PHP is used to display any message and exit out of the current script at the same time.
- Hence once an error condition is detected, die() function can be used to exit the code execution.
- An error can occur when we try to divide a number with zero, or try to open a file which doesn't exist etc.
- Below example snippet shows that we will get the message Unable to open file noerror.txt on the screen, if there is no file with name noerror.txt at the given location.

```
<?php

    $fileName = "noerror.txt";
    // opening the file
    fopen($fileName, "r")
    or die("Unable to open file $fileName");

?>
```

- The die() function is an alias of exit() function which is also used to exit from code execution.
- This type of error handling is useful when we know situations which may lead to errors beforehand. For example, the below code will lead to error:

```
<?php

    function division($numerator, $denominator) {
        // perform division operation
        echo $numerator / $denominator;
    }

    // calling the function
    division(7, 0);

?>

` //Warning: Division by zero...
```

- In such a situation, where we know certain conditions can lead to error, we can use conditional statements to handle the corner case which will lead to error.

```
<?php

    function division($numerator, $denominator) {
        // use if statement to check for zero
        if($denominator != 0) {
            echo $numerator / $denominator;
        }
        else {
            echo "Division by zero(0) not allowed!";
        }
    }

    // calling the function
    division(7, 0);

?>
//Division by zero(0) not allowed!
```

- As you can see in the code above the error condition is handled. Now our code will not display any error on screen, rather it will display a message and exit gracefully.

Custom Error Handler

- We can use our custom method to display any message or execute any code when an error occurs.
- We can define our custom error handling function, which is straightforward to create.
- This function requires a minimum of two parameters and supports a maximum of five parameters.

```
custom_error_function(error_no, error_message, error_file, error_line, error_context)
```

Parameter	Description
error_no	It is required and must be a value number. It sets the level of error reporting for a user-defined error.
error_message	It is required and prints the user-defined error message.
error_file	It is optional. It specifies the file where an error has occurred.
error_line	It is optional. It prints the line where an error has occurred.
error_context	It is optional. When an error occurs, it prints an array containing all the variables with values.

- We have to do is set our method as the default error handler for PHP using the function set_error_handler()
- The PHP set_error_handler() function is a built-in error handler function that sets the error handler function defined by the user.

```
set_error_handler(errorhandler, E_ALL | E_STRICT)
```

Parameter	Description
errorhandler	It sets a user error handling function name.
E_ALL|E_STRICT	It specifies at which error report level the error defined by the user will be shown. The default value is E_ALL.

Constant	Description
E_ERROR	Run time fatal errors. Stop the execution of the script.
E_WARNING	Run time non-fatal errors. Do not stop the script execution.
E_PARSE	The parser generates the compile-time parsing error.
E_NOTICE	The script found something that could be an error.
E_CORE_ERROR	The fatal errors at the initial start-up of PHP
E_CORE_WARNING	Run time non-fatal errors occur at the initial start-up of PHP.
E_USER_ERROR	User-generated fatal error.E_ERROR generated by the programmer by using trigger_error()
E_USER_WARNING	User-generated non-fatal warning.E_WARNING generated by the programmer by using trigger_error()
E_USER_NOTICE	User-generated run-time notice.E_NOTICE generated by the programmer by using trigger_error()
E_STRICT	User-generated run-time notice
E_RECOVERABLE_ERROR	Catch fatal errors. It is similar to E_ERROR but caught by the user-defined handle.
E_ALL	It captures all the warnings and errors, except E_STRICT

```
<?php
     // custom function to handle error
    function sample_error_handler($error_no, $error_msg) {
        echo "Oops! Something unexpected happened...";
        echo "Possible reason: " . $error_msg;
        echo "We are working on it.";
    }
    // set the above function s default error handler
    set_error_handler("sample_error_handler");

    $result = 7 / 0;
?>


//Oops! Something unexpected happened...
//Possible reason: Division by zero
```

```
//We are working on it.


//-------------------------------------------

<?php
     /* User-defined error handling function */

     function customErrorsHandler($errorNo, $errorMessage,
     $errorFile, $errorLine) {
       echo "Error Message: [$errorNo] $errorMessage<br>";
       echo "Error on line $errorLine in $errorFile";
     }

     /* Set Error Handler */
     set_error_handler("customErrorsHandler");

     echo $hello;
?>

//Error Message: [8] Undefined variable: hello
//Error on line 12 in D:\xampp\htdocs\test\example.php
```

- We can define our own function like we did above with arguments $error_no and $error_msg and can handle them as we want.
- We can show custom messages, log errors in the database, send email to the developer reporting the error etc.

PHP Error Reporting

- PHP provides a default error reporting mechanism which can be utilized to display messages on screen when an error occurs.
- We can use the PHP function error_reporting() to set which errors to show and which errors to hide.

```
<?php
    // error reporting function
    error_reporting(reporting_level);
?>
```

- The value for the variable *reporting_level* defines which error to show and which errors to hide, if we do not provide any value for it, the error reporting will be set to default.

Reporting Level	Description
E_WARNING	Only displays warning messages and doesn't stop the execution of the script.
E_NOTICE	Displays notice that occur during normal code execution.
E_USER_ERROR	Display user generated errors i.e the custom error handlers.
E_USER_WARNING	Display user generated warning messages.
E_USER_NOTICE	Display user generated notices.
E_RECOVERABLE_ERROR	Displays non-fatal errors.
E_ALL	Display all errors and warnings.

Exception Handling

- We discussed error handling. Now the question arises, what is the difference between, Exception and an Error.
- With PHP 5 a new object oriented way of handling errors was introduced, which is called Exception.
- Exception handling is used to handle errors and redirect the course of code execution when it occurs, unlike errors where the code execution stops with an error message displayed on the screen.

Keyword	Description
try	In the "try" block, codes are written in which there may be an exception. If there is no exception, this code execution will continue. Otherwise, it "throws" an exception.
throw	If an exception is triggered, the control is transferred from the try block to the catch block using the keyword "throw". Each "throw" must have a "catch".
catch	The catch block catches the exception and creates an object which will hold exception information.

try and catch block

- The code which can lead to exception or error is enclosed within the try block, if no exception occurs the code is executed normally while in case of exception, the code execution exits the try block and enters the catch block.

```
<?php
     try {
         //code goes here that could lead to an exception
     }
     catch (Exception $e) {
         //exception handling code goes here
     }
?>
```

throw Exception

- We can manually trigger an exception if required using the throw keyword.
- Exception is a PHP class which is the parent class for all exception classes in PHP.
- To throw an exception we have to create an object of the exception class and then use the throw keyword to trigger that exception.

```
<?php
    // function declaration
    function triggerException() {
        // using throw keyword
        throw new Exception("Manually triggering exception...");
    }
?>
```

- If we call the above function triggerException() then it will throw an exception.

```
<?php
     try {
         // calling the function
         triggerException();
     }
     catch (Exception $e) {
         echo "Oops! Some unexpected error occurred ...";
     }
?>
//Oops! Some unexpected error occurred ...
```

- Inside the catch block, we can also get the message from the exception object using the getMessage() method.

Custom Exception Class

- We can create a custom exception class by extending the Exception class provided by PHP.

```
<?php
    // custom exception class
    class SampleException extends Exception {
        // define constructor
        function __construct() {
            // we can even take arguments if we want
        }
        // we can define class methods if required
    }
?>
```

- Custom exception classes are useful when you have requirements for custom error handling, for example logging errors in databases etc.

Handling Multiple Exceptions

- If a piece of code can throw different types of exceptions and based on the type of exception we have to perform some action, in such a situation we can have multiple catch blocks:

```
<?php

    try {
        // calling the function
        triggerException();
    }
    catch (SampleException $e) {
        // do something here...
    }
    catch (Exception $e) {
        echo "Oops! Some unexpected error occurred...";
    }

?>
```

Points to remember when handling multiple exceptions using multiple catch blocks:

- catch block handling child class of Exception class must be placed above the catch block handling the Exception class. Or in other words, Exception class handling catch blocks should be kept at last.
- catch block handling the Exception class can handle other exceptions as well, as all the exception classes are child classes of the Exception class.

Example:

```
<?php
     try{
       if(!file_exists("example.csv")) {/* Check if the file
     exists */
         throw new Exception("error: The required file is
     missing.");
       }
       $fp = fopen("example.csv", "r"); /* Open the file. */
     }
     catch (Exception $err) {
       echo "Caught an exception: ",  $err->getMessage();
       /* This data can be logged here that the file is not
     being opened for some reason. */
     }

     echo "The program execution continues."
?>
//Caught an exception: error: The required file is missing.
```

Following are some of the functions that can be used:

- getMessage() − Prints exception message.
- getCode() −Prints exception code.
- getFile() − Prints source filename.
- getLine() − Prints source line.
- getTrace() − Prints n-array of the backtrace().
- getTraceAsString() − Prints the formatted string of trace.

Different Error levels

Sr. No	Constant & Description	Value
1	**E_ERROR** Fatal run-time errors. Execution of the script is halted	1
2	**E_WARNING** Non-fatal run-time errors. Execution of the script is not halted	2
3	**E_PARSE** Compile-time parse errors. Parse errors should only be generated by the parser.	4
4	**E_NOTICE** Run-time notices. The script found something that might be an error, but could also happen when running a script normally	8
5	**E_CORE_ERROR** Fatal errors that occur during PHP's initial start-up.	16
6	**E_USER_ERROR** Fatal user-generated error. This is like an E_ERROR set by the programmer using the PHP function trigger_error()	256
7	**E_USER_WARNING** Non-fatal user-generated warning. This is like an E_WARNING set by the programmer using the PHP function trigger_error()	512
8	**E_USER_NOTICE** User-generated notice. This is like an E_NOTICE set by the programmer using the PHP function trigger_error()	1024
9	**E_STRICT** Run-time notices. Enable to have PHP suggest changes to your code which will ensure the best interoperability and forward compatibility of your code.	2048
10	**E_RECOVERABLE_ERROR** Catchable fatal error. This is like an E_ERROR but can be caught by a user defined handle (see also set_error_handler())	4096

11	**E_ALL** All errors and warnings, except level E_STRICT (E_STRICT will be part of E_ALL as of PHP 6.0)	8191

All the above error level can be set using following PHP built-in library function
int error_reporting ([int $level]);

e.g

```
<?php
     //Turn off all error reporting
     error_reporting(0);
     // Report simple running errors
     error_reporting(E_ERROR | E_WARNING | E_PARSE);
     /*Reporting E_NOTICE can be good too (to report uninitialized
     variables or  catch variable name misspellings ...) */
     error_reporting(E_ERROR | E_WARNING | E_PARSE | E_NOTICE);
?>
```

Trigger an Error

In a script where users can input data it is useful to trigger errors when an illegal input occurs. This is done by the trigger_error() function

Possible error types:

- E_USER_ERROR - Fatal user-generated run-time error. Errors that can not be recovered from. Execution of the script is halted
- E_USER_WARNING - Non-fatal user-generated run-time warning. Execution of the script is not halted
- E_USER_NOTICE - Default. User-generated run-time notice. The script found something that might be an error, but could also happen when running a script normally.

```
<?php
     $x = 'test';

     if(is_numeric($var)) {
     echo $var;
     } else {

     trigger_error('var must be numeric',E_USER_ERROR);
     // trigger_error('var must be numeric',E_USER_NOTICE); }
     echo "end script";
     }
?>
//Generate the fatal error
```

Exercise:

1. Create a php page which takes the student's name and marks of three subjects from the user. Manage all information via session and display all information on the next page.
2. Create php code which do following task
   Signup.php : take username and password from user and select hobbies(from checkbox) & store information as session variable. Store password in encrypted format using md5 encryption.
   Viewhobbie.php : if session variable username not set display "login required" message else display the hobbies of respective user in a well designed page.
   Singout.php : clear the session variables.
3. Develop a HTML form which takes the file name and string form the user. Create a php script which creates or opens a file (if it already exists) and appends the string into the given file when the write button is clicked. On click on read button it displays the content of the file into the browser.
4. Create a php application which has a login page, a product page, cart display page, purchase message page and logout page. Users can login into the system using username and password which are validated from an already stored file on the server. Once a user is logged in successfully the user can select a product to buy and quantity to buy. User can review the selected product with invoice details in cart page and click on buy to purchase and can signout from application
5. Develop a PHP form to get students marks of 5 subjects and generate results. Take the following inputs: Exam No, Course (Drop Down), Student photo and Semester. The photo should not be greater than 25 kb in size and must be in JPEG format. Check necessary validation and provide the option to upload the photograph on click on upload button. Display the rest of the detail on the browser on click on submit button.
6. Create a php script to demonstrate different types of errors.(Notice, warning, error)
7. Create PHP script for a custom error handling function in which two parameters are passed. If the first parameter is less than the second parameter then it should display a warning. If any of the parameters is not a number then it should generate fatal error
8. Create a php script which triggers an error/warning if the number entered by the user is negative with an appropriate message.