Unit 5: File Handling, Sessions, and Error Handling

Presented By:
Bipin Maharjan

Learning Objectives

- Learn about the importance of file handling in web development, including reading file content and storing user-generated content, logging application data etc.
- Understand the various file reading and writing modes and their respective use cases.
- Know the differences between file inclusion methods (include and require) and their respective use cases.
- Gain practical knowledge in managing sessions and cookies for maintaining user state and tracking user interactions.
- Acquire skills in implementing session-based authentication and authorization mechanisms to control access to web applications and resources.
- Learn to efficiently manage and gracefully handle unexpected errors or exceptional conditions in the application

Table of Contents

File Handling

Reading from and writing to files

- In PHP File System allows us to create file, read file line by line, read file character by character, write file, append file, delete file and close file.
- It is useful for logs, configurations, storing user data, or exporting reports.
- PHP supports following functions to handle (Create, Read, Write, Delete and Close) files.
  - fopen()
  - fread()
  - fwrite()
  - fclose()
  - unlink()

fopen() Function

- fopen() function is used to open file or URL.
- The fopen() function accepts two arguments: $filename and $mode.
- The $filename represents the file to be opended and $mode represents the file mode for example read-only, read-write, write-only etc.

Syntax:

fopen (string $filename, string $mode)

Supported modes are:

r - Read Only

r+ - Read & Write

w - Write Only

w+ - Read & Write

a - Write Only(Append).

fopen() Example

```
<?php

$filename = "D:\\madhu\\file.txt";

$fp = fopen($filename, "r");//open file in read mode

?>
```

fread() Function

- fread() function is used to read data of the file.
- It requires two arguments: file resource and file size.

Syntax:

fread (resource $fp, int $length)

$fp represents file pointer that is created by fopen() function.

$length represents length of file (bytes)to be read.

fread() Example

```
<?php
$filename = "D:\\madhu\\file.txt";
$fp = fopen($filename, "r");//open file in read mode
$content = fread($fp, filesize($filename));//read file
echo "$content";//printing data of file
fclose($fp);//close file
?>
```

Note: To read content of file, we need to open file with r , r+, w+ mode.

fwrite() function

- fwrite() function is used to write content (string) into file.
- fwrite() function will erase the previous data of the file and writes the new data.
- It requires two arguments: file resource and data(string).

```
fwrite(resource $fp, string $data)
```

- $fp represents file pointer that is created by fopen() function.
- $data represents text to be write.
- To write data into file, you need to use w,r+,w+ mode

fwrite() Example

```
<?php

$filename = "D:\\madhu\\file.txt";

$fp = fopen($filename, "w");//open file in write mode

fwrite($fp,'PHP is a Scripting');

fclose($fp);//close file

?>
```

Output: PHP is a Scripting

```
<?php

$filename = "D:\\madhu\\file.txt";

$fp = fopen($filename, “a");//open file in write mode

fwrite($fp,’language at server side’);

fclose($fp);//close file

?>
```

Output: PHP is a Scripting language at server side

fclose() Function

- fclose() function is used to close an open file.
- It requires one argument i.e. “File Pointer”.

Syntax

```
fclose ( resource $fp )
```

$fp represents file pointer that is created by fopen() function.

fclose() Example

```
<?php

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

echo fread($file, filesize("example.txt"));

fclose($file); // important!
```

unlink() Function

- In PHP, we can delete any file using unlink() function.
- The unlink() function accepts one argument only: file name.

Syntax

```
unlink ( string $filename )
```

$filename represents the name of the file to be deleted.

unlink() Example

```
<?php
$status = unlink('data.txt');
if($status) {
	echo "File deleted successfully";
} else {
	echo "Sorry!";
}
?>
```

Reading from and writing to files: Example

```
<?php
// 1. Create and write into a file
$file = fopen("students.txt", "w");
fwrite($file, "Welcome to PHP File Handling!\n");
fwrite($file, "We are learning fopen, fwrite, fread, fclose, and unlink.");
fclose($file);

// 2. Read from the file
$file = fopen("students.txt", "r");
echo "File content:<br>";
echo fread($file, filesize("students.txt"));
fclose($file);

// 3. Delete the file
unlink("students.txt");
?>
```

Importance of File Handling

- Permanent Data Storage
  - Stores data permanently on the server even after the program ends.
  - Example: Saving user feedback or form submissions in a text file.
- Content Management
  - Allows dynamic web pages by reading content (like articles, posts) from files.
  - Example: A CMS loading blog posts from text or JSON files.
- Activity Logging
  - Helps track user actions and system events through log files.
  - Example: Recording login attempts or page visits in a log file.
- File Upload Handling
  - Enables web applications to store uploaded files such as images or documents.
  - Example: Saving user profile pictures or uploaded resumes.
- Simpler Alternative to Database (for small apps)
  - Useful for small projects where a database isn’t needed.
  - Example: A simple guestbook saving entries to a text file.

Understanding File Types and Permissions

File types affect how information is stored in files and retrieved from them

File permissions determine the actions that a specific user can and cannot perform on a file

Working with File Permissions

Files and directories have three levels of permissions that control access.

- Owner (User) - person who owns the file
- Group - other users in the same group
- Other - everyone else

The three typical permissions for files and directories are:

- Read (r) - can read/view file contents
- Write (w) - can modify or delete file
- Execute (x) - can run the file (for programs/scripts)

Working with File Permissions

File permissions are calculated using a four-digit octal (base 8) value

- Octal values encode three bits per digit, which matches the three permission bits per level of access
- The first digit is always 0
- To assign more than one value to an access level, add the values of the permissions together

Working with File Permissions

Permissions	First Digit (Leftmost) always 0	Second Digit User (u)	Third Digit Group (g)	Fourth Digit (Rightmost) Other (o)
Read (r)	0	4	4	4
Write (w)	0	2	2	2
Execute (x)	0	1	1	1

Octal values for the *mode* parameter of the chmod() function

Working with File Permissions

- The chmod() function is used to change the permissions or modes of a file or directory
- The syntax for the chmod() function is

```
chmod ($filename, $mode)
```

- Where $filename is the name of the file to change and $mode is an integer specifying the permissions for the file

File Permission Example

-rw-r--r-- 1 user group 1234 Oct 3 notes.txt

Here,

- rw- → owner can read & write.
- r-- → group can only read.
- r-- → others can only read.

Changing File Permissions

PHP provides the chmod() function to change file permissions

```
chmod("example.txt", 0644); // Owner: read & write, Others: read only

chmod("uploads/", 0755);   // Owner: full access, others: read & execute
```

Common Permission Values

- 0644 → files: owner read/write, others read.
- 0755 → directories: owner full access, others can read/enter.
- 0600 → private file: only owner can read/write.

Checking Permissions

- The fileperms() function is used to read permissions associated with a file
  - The fileperms() function takes one argument and returns an integer bitmap of the permissions associated with the file
  - Permissions can be extracted using the arithmetic modulus operator with an octal value of 01000
- The dococt() function converts a decimal value to an octal value

Security Considerations in File Handling

- Do not allow user input directly in file paths.
  - Example: $_GET['file'] could be misused with ../../etc/passwd (path traversal attack).
  - Solution: sanitize input, whitelist allowed filenames, or use database IDs instead.
- Store uploaded files outside the web root.
  - Prevents users from directly accessing uploaded files via URL.
- Restrict file types (for uploads).
  - Check file extensions (.jpg, .txt) and MIME types.
  - Never allow .php or other executable files to be uploaded inside web root.
- Set safe permissions.
  - Files: 0644 (readable, but only writable by owner).
  - Folders: 0755 (owner full control, others read/enter only).
  - Never use 0777 (anyone can read/write/execute).

Security Considerations in File Handling

- Use file locking for concurrency.
  - Prevents race conditions when multiple users write to the same file (flock() or LOCK_EX).
- Check file existence before use.

```
if (file_exists("data.txt")) {
    $file = fopen("data.txt", "r");
}
```

- Hide errors from users.
  - Don’t show raw error messages like “Permission denied” — attackers could use that info.

File Inclusion

- File inclusion allows you to reuse code across multiple web pages instead of writing the same code repeatedly.
- It helps in organizing large PHP projects by separating code into multiple files (for example, header, footer, database connection, etc.).
- In PHP, file inclusion means importing the content of one PHP file into another before the server executes it.
- When PHP encounters an inclusion statement (like include or require), it reads and executes the code in the included file as if it were written inside the current file.

File Inclusion Example

```
<?php

include "header.php"; // imports header section

echo "<h2>Welcome to My Website</h2>";

include "footer.php"; // imports footer section

?>
```

Here, header.php and footer.php may contain HTML and PHP code that will be displayed on every page.

Why use File Inclusion

- **Code Reusability** – Common sections (like navigation bar, header, footer, or database connection) can be written once and reused.
- **Maintainability** – If you change one file (e.g., header.php), it automatically updates on all pages that include it.
- **Clean Project Structure** – Helps to keep code modular and organized.
- **Avoid Repetition** – Reduces code duplication and errors.

PHP File Inclusion Methods

PHP Provides four inclusion statements:

Function	Behavior
`include 'file.php';`	Includes and executes file; gives a **warning** if file not found but script continues.
`require 'file.php';`	Includes and executes file; gives a **fatal error** if file not found and script stops.
`include_once 'file.php';`	Same as include, but includes the file **only once** (prevents duplicate loading).
`require_once 'file.php';`	Same as require, but includes file **only once** (even if called multiple times).

Differences between include and require

Aspect	`include`	`require`
Error Type	Generates a *warning* if the file is missing	Generates a *fatal error* if the file is missing
Script Behavior	Script continues to execute	Script stops execution
Best Use Case	Optional files (like footer, sidebar)	Essential files (like database connection or configuration)
Example Output	Shows warning message and continues	Stops and displays fatal error

Example using include

```
<?php
echo "Before including file.<br>";
include "header.php"; // If header.php not found, script continues
echo "After including file.<br>";
?>
```

If header.php doesn’t exist, PHP shows a warning but continues execution.

Example using require

```
<?php
echo "Before requiring file.<br>";
require "config.php"; // If config.php not found, script stops
echo "This line will not be executed if file is missing.";
?>
```

If config.php is missing, PHP shows a fatal error and stops execution.

Example using include_once and require_once

```
<?php
include_once "functions.php"; // Included only once
include_once "functions.php"; // Ignored second time

require_once "dbconnect.php"; // Included once
require_once "dbconnect.php"; // Ignored second time
?>
```

Prevents reloading the same file multiple times, avoiding function redefinition or variable duplication errors.

Typical Real-world Use Cases

File Name	Purpose	Inclusion Type
`config.php`	Database connection settings	`require`
`header.php`	Common header or navigation section	`include`
`footer.php`	Common footer for all pages	`include`
`functions.php`	Helper functions used across pages	`require_once`
`dbconnect.php`	Database connection file	`require_once`

Sessions & Cookies

What is a Cookie?

- A cookie is often used to identify a user.
- A cookie is a small file that the server embeds on the user’s computer.
- Each time the same computer requests a page with a browser, it will send the cookie too.
- With PHP, you can both create and retrieve cookie values.
- Cookies are primarily used to store the user’s browsing history.

Create Cookies with PHP

- A cookie is created with the setcookie() function
- Syntax: **setcookie(name, value, expire, path, domain, security);**
- Parameters: The setcookie() function requires six arguments in general which are:
  - **Name:** It is used to set the name of the cookie
  - **Value:** It is used to set the value of the cookie
  - **Expire:** It is used to set the expiry timestamp of the cookie after which the cookie can’t be accessed.
  - **Path:** It is used to specify the path on the server for which the cookie will be available.
  - **Domain:** It is used to specify the domain for which the cookie is available.
  - **Security:** It is used to indicate that the cookie should be sent only if a secure HTTPS connection exists.

Creating a Cookie

```
<?php
// Create a cookie that lasts for 1 hour
setcookie("username", "John", time() + 3600, "/");
echo "Cookie has been set!";
?>
```

- The cookie username=John will exist for 1 hour (3600 seconds).
- It’s available to the entire website because of /.
- Cookies must be set before any HTML output — otherwise you’ll get a “headers already sent” error.

Accessing a Cookie

- Cookies are accessed through the $_COOKIE superglobal array.

```
<?php
if (isset($_COOKIE["username"])) {
    echo "Welcome back, " . $_COOKIE["username"];
} else {
    echo "Hello, new visitor!";
}
?>
```

- isset() checks if the cookie exists.
- $_COOKIE["username"] retrieves the stored value.

Modifying an Existing Cookie

- To modify a cookie, simply call setcookie() again with the same name but a new value.

```
<?php

setcookie("username", "Alice", time() + 3600, "/");

echo "Cookie value updated!";

?>
```

Deleting a Cookie

- To delete a cookie, set its expiration time in the past.

```
<?php
setcookie("username", "", time() - 3600, "/");
echo "Cookie deleted!";
?>
```

- This tells the browser that the cookie has expired, so it deletes it immediately.

Example: Theme Preference using Cookies

```
<?php

$theme = $_GET['theme'];

setcookie("theme_color", $theme, time() +
(86400 * 30), "/"); // 30 days

echo "Theme color saved as $theme!";

?>
```

```
<?php
if (isset($_COOKIE["theme_color"])) {
    $theme = $_COOKIE["theme_color"];
} else {
    $theme = "light"; // Default theme
}
?>
<!DOCTYPE html>
<html>
<head>
    <title>Cookie Example</title>
</head>
<body style="background-color: <?php echo $theme == 'dark' ?
'#333' : '#fff'; ?>; color: <?php echo $theme == 'dark' ? '#fff' : '#000';
?>;">
    <h2>Welcome!</h2>
    <p>Your current theme is: <?php echo ucfirst($theme); ?></p>
    <a href="set_theme.php?theme=light">Switch to Light Mode</a> |
    <a href="set_theme.php?theme=dark">Switch to Dark Mode</a>
</body>
</html>
```

Use Cases of Cookies

- Remember username
- Track page visits
- Personalization
- Analytics
- “Remember Me” login

What is a Session?

- When you work with an application, you open it, do some changes, and then you close it.
- The computer knows who you are. It knows when you start the application and when you end.
- But on the internet there is one problem: the web server does not know who you are or what you do, because the HTTP address doesn’t maintain state.

Contd…

- Session variables solve this problem by storing user information to be used across multiple pages. By default, session variables last until the user closes the browser.
- So, session variables hold information about one single user, and are available to all pages in one application.
- A session is a way to store information on the server (not in the user's browser) that can be used across multiple pages.
- Session technique is widely used in shopping websites where we need to store and pass cart information e.g. username, product code, product name, product price etc from one page to another.

Why do we need Sessions?

- HTTP is a stateless protocol — meaning each page request is independent; the server doesn’t automatically remember who you are between pages.
- So if a user logs in or adds something to their cart, the next page would normally “forget” that.
- Sessions solve this problem — they allow you to maintain user data temporarily on the server while the user is active.

How Sessions Work

- User visits your website → PHP creates a unique session ID (like a9e8f6c7b2).
- This ID is stored in a cookie in the user's browser (or passed in the URL).
- On every new page request, the browser sends back the session ID.
- PHP uses that ID to retrieve the user's data stored on the server.

So the data is on the server, not in the browser — unlike cookies.

Starting a PHP Session

- A session is started with the **session_start()** function.
- Session variables are set with the PHP global variable: **$_SESSION**

Note: session_start() must be written before any HTML output.

- <?php session_start(); ?>
  <html>
  <body></body>
  </html>

Storing data in a Session

- The correct way to store and retrieve session variables is to use the PHP $_SESSION variable.

```
<?php
session_start();
$_SESSION["username"] = "John";
$_SESSION["role"] = "student";
?>
```

- **$_SESSION** is an associative array that contains all session variables. It is used to set and get session variable values.
- The data stays available across pages until the session is destroyed or expired.

Accessing Session Data on Another Page

```
<?php

session_start();

echo "Welcome " . $_SESSION["username"] . "!<br>";

echo "Your role is: " . $_SESSION["role"];

?>
```

- This will display the same data stored earlier, even though it’s a new page request.

Modifying Session Data

```
<?php

session_start();

$_SESSION["username"] = "Alice"; // Updated username

echo "Username updated!";

?>
```

Removing Specific Session Data

- If you wish to delete some session data, you can use the unset() or the session_destroy() function.
- The unset function is used to free the specified session variable.

```
<?php

session_start();

unset($_SESSION["username"]); // Remove only one variable

?>
```

Destroying the entire Session

- You can also completely destroy the session by calling the session_destroy() function.

```
<?php
session_start();
session_unset();  // Remove all session variables
session_destroy(); // Destroy the session completely
echo "Session destroyed!";
?>
```

- After destroying the session, you'll need to start a new one if you want to set variables again.

What is Authentication?

- Authentication means verifying who the user is.
- It checks the identity of the user
- Example: When you log into Gmail with your username and password — that’s authentication.

What is Authorization?

- Authorization determines what the user is allowed to do after they’re authenticated.
- Example
  - A student can view their marks.
  - An admin can add or delete marks.
- Even though both are logged in, their permissions differ — that’s authorization.

Why use sessions for authentication?

Because HTTP is stateless, the server forgets the user after each request.

So, after login, we store user information (like username, role, or user_id) in a session.

This allows the server to:

- Remember who’s logged in.
- Control which pages they can access.
- Log them out easily when needed.

Workflow of Session-Based Authentication

Here’s how it works step by step:

1. User logs in → username & password sent to PHP script.
2. PHP verifies credentials (hardcoded or from database).
3. On success → PHP creates a session and stores user details.
4. On every protected page → PHP checks if session exists.
5. If not logged in → redirect to login page.
6. When user logs out → session is destroyed.

Source Code: https://github.com/BipinMhzn/internet_technology_2

Error Handling

Why Error Handling is Necessary?

- Normally, it displays a message indicating the cause of the error and may also terminate script execution when a PHP script encounters an error.
- Now, while this behavior is acceptable during the development phase, it cannot continue once a PHP application has been released to actual users.
- In “live” situations, it is unprofessional to display cryptic error messages (which are usually incomprehensible to non-technical users).
- It is more professional to intercept these errors and either resolve them (if resolution is possible), or notify the user with an easily-understood error message (if not).

What is Error Handling?

Error handling in PHP means detecting and managing problems in code during execution — so that the program doesn’t stop unexpectedly.

Instead of showing users scary error messages, PHP can:

- Log errors to a file.
- Show friendly error messages.
- Take recovery actions (like using backup data).

Error Types

There are 12 unique error types, which can be grouped into 3 main categories:

- Notice Error
- Warning Error
- Fatal Error

Notice Error

- These are trivial, non-critical errors that PHP encounters while executing a script.
- By default, such errors are not displayed to the user at all - we can change this default behaviour.
- Example: use of an undefined variable, defining a string without quotes, etc.

Notice Error: Example

Using an undefined variable or trying to access an array index that doesn’t exist.

```
<?php
echo "Example of Notice Error<br>";

$studentName = "John";
echo $studentAge;  // Undefined variable (Notice)
echo "<br>Script continues running normally.";
?>
```

Output:

Notice: Undefined variable: studentAge in /path/to/file.php on line 5

Example of Notice Error

Script continues running normally.

- PHP warns you that $studentAge doesn’t exist.
- But it doesn’t stop execution — the rest of the code runs fine.

Warning Error

- These are more serious errors than a Notice
- Example: attempting to include() a file which does not exist, database not available, missing function arguments, etc.
- By default, these errors are displayed to the user, but they do not result in script termination.

Warning Error: Example

Trying to include a missing file or opening a non-existent file.

```
<?php
echo "Example of Warning Error<br>";

include("non_existing_file.php"); // File does not exist (Warning)
echo "<br>Script continues running after the warning.";
?>
```

Output:

```
Warning: include(non_existing_file.php): Failed to open stream: No such file or directory in /path/to/file.php on line 4

Warning: include(): Failed opening 'non_existing_file.php' for inclusion (include_path='.') in /path/to/file.php on line 4

Example of Warning Error

Script continues running after the warning.
```

- PHP couldn’t find the file, so it gives a Warning.
- Script continues, but that file’s contents won’t be available.

Fatal Errors

- These are critical errors
- Example: instantiating an object of a non-existent class, or calling a non-existent function.
- These errors cause the immediate termination of the script, and PHP’s default behavior is to display them to the user when they take place.

Fatal Error: Example

Calling a function that doesn’t exist or using a class that’s undefined.

```
<?php
echo "Example of Fatal Error<br>";

nonExistingFunction(); // Function does not exist (Fatal Error)

echo "<br>This line will never be executed.";
?>
```

Output:

Example of Fatal Error

Fatal error: Uncaught Error: Call to undefined function nonExistingFunction() in /path/to/file.php on line 4

- PHP stops execution immediately when it reaches a fatal error.
- Anything after the error line will not run.

Traditional Error Handling Functions

Before PHP introduced exception handling, developers used functions like:

Function	Description
`error_reporting()`	Sets which PHP errors are reported
`set_error_handler()`	Defines a custom function to handle errors
`trigger_error()`	Manually trigger an error
`error_log()`	Send error messages to a log file

Traditional Error Handling Functions: Example

```
<?php
// Report all errors
error_reporting(E_ALL);

// Trigger a custom error
trigger_error("Custom warning: Something went wrong!", E_USER_WARNING);
?>
```

What is an Exception?

- An exception is an object that represents an error condition.
- When an error occurs, PHP throws an exception that can be caught and handled by the program.
- In PHP, a try–catch block is used to handle exceptions — which are special objects that represent runtime errors.

Syntax:

```
try {
    // Code that may cause an error
} catch (Exception $e) {
    // Code to handle the error
}
```

Basic Try-Catch Example

```
<?php
try {
    $num1 = 10;
    $num2 = 0;

    if ($num2 == 0) {
        throw new Exception("Division by zero not
allowed!");
    }

    $result = $num1 / $num2;
    echo "Result: $result";
}
catch (Exception $e) {
    echo "Error: " . $e->getMessage();
}
?>
```

Output:

Error: Division by zero not allowed!

Exception Object Methods

Method	Description
`$e->getMessage()`	Returns the error message
`$e->getCode()`	Returns the error code
`$e->getFile()`	Returns the file name where the error occurred
`$e->getLine()`	Returns the line number of the error
`$e->getTrace()`	Returns a backtrace array
`$e->getTraceAsString()`	Returns backtrace as a string

Exception Object Methods: Example

```
<?php
try {
    throw new Exception("File not found!", 404);
}
catch (Exception $e) {
    echo "Error Message: " . $e->getMessage() .
"<br>";
    echo "Error Code: " . $e->getCode() . "<br>";
    echo "Error File: " . $e->getFile() . "<br>";
    echo "Error Line: " . $e->getLine();
}
?>
```

Output:

Error Message: File not found!

Error Code: 404

Error File: /path/to/script.php

Error Line: 4

Custom Exception Classes

You can create your own exception classes to handle specific error types more clearly.

```
<?php
class FileNotFoundException extends Exception {}

try {
    $filename = "missing.txt";
    if (!file_exists($filename)) {
        throw new FileNotFoundException("File '$filename'
not found!");
    }
}
catch (FileNotFoundException $e) {
    echo "Custom Exception: " . $e->getMessage();
}
catch (Exception $e) {
    echo "General Exception: " . $e->getMessage();
}
?>
```

Output:

Custom Exception: File 'missing.txt' not found!

Multiple Catch Blocks

```
<?php
class FileException extends Exception {}
class InputException extends Exception {}

try {
    $filename = "data.txt";
    if (!file_exists($filename)) {
        throw new FileException("File not found!");
    }

    $age = -5;
    if ($age < 0) {
        throw new InputException("Invalid age entered!");
    }

}
catch (FileException $e) {
    echo "File Error: " . $e->getMessage();
}
catch (InputException $e) {
    echo "Input Error: " . $e->getMessage();
}
catch (Exception $e) {
    echo "General Error: " . $e->getMessage();
}
?>
```

Output:

File Error: File not found!

The finally Block

- finally is always executed, whether or not an exception occurs
- Often used for cleanup like closing files or database connections.

```
<?php
try {
    echo "Opening file...<br>";
    throw new Exception("File not found!");
} catch (Exception $e) {
    echo "Error: " . $e->getMessage() . "<br>";
} finally {
    echo "Closing file and cleaning up
resources.";
}
?>
```

Output:

Opening file...

Error: File not found!

Closing file and cleaning up resources.

Any Questions?