PHP/MySQL Tutorial: Effective Code Examples and Explanations

This document provides a comprehensive guide to PHP and MySQL, covering fundamental concepts, syntax, and practical applications with detailed code examples and explanations.

1. Introduction to PHP

PHP (Hypertext Preprocessor) is a widely-used open-source scripting language especially suited for web development. It is primarily used for server-side programming, meaning it runs on the web server rather than in the user's browser. This allows PHP to interact with databases, handle form data, manage sessions, and perform other server-side operations to generate dynamic web content.

1.1. What is PHP and its role in web development

PHP plays a crucial role in web development by enabling the creation of dynamic and interactive websites. When a user requests a PHP page, the web server processes the PHP code, generates HTML output, and then sends that HTML to the user's browser. This allows for personalized content, database interaction, and complex application logic that static HTML pages cannot provide.

Key roles of PHP in web development include:

- **Server-side Scripting:** PHP executes on the server, handling tasks like processing form data, generating dynamic page content, and managing user sessions.
- **Database Interaction:** PHP can connect to and manipulate various databases, most commonly MySQL, to store and retrieve data for web applications.
- **Dynamic Content Generation:** It allows developers to create web pages that change based on user input, time of day, or data from a database.
- **Session and Cookie Management:** PHP provides built-in functionalities to manage user sessions and cookies, enabling features like user logins and personalized experiences.
- **File System Operations:** PHP can interact with the server's file system, allowing for file uploads, downloads, and manipulation.

1.2. Basic PHP syntax and structure

PHP code is embedded within HTML documents using special PHP tags. The most common and recommended tag is `<?php ... ?>`. Any code within these tags is interpreted as PHP code by the server. PHP statements are typically terminated with a semicolon (`;`).

Example: Basic PHP Structure

PHP

```
<!DOCTYPE html>
<html>
<head>
    <title>My First PHP Page</title>
</head>
<body>

    <h1>Welcome to my website!</h1>

    <?php
        // This is a single-line comment in PHP
        /*
         * This is a multi-line comment
         * in PHP
         */
        echo "<p>Hello from PHP!</p>";
    ?>

    <p>This is regular HTML content.</p>

</body>
</html>
```

Explanation:

- The `<?php` and `?>` tags define the boundaries of the PHP code block. Everything outside these tags is treated as plain HTML.
- `//` is used for single-line comments.
- `/* ... */` is used for multi-line comments.
- The `echo` statement is a PHP construct used to output strings to the browser. In this case, it outputs an HTML paragraph.

1.3. Writing your first PHP script

To write your first PHP script, you need a text editor and a web server with PHP installed (e.g., Apache with PHP, Nginx with PHP-FPM, or a local development environment like XAMPP/WAMP/MAMP). Save your file with a `.php` extension (e.g., `hello.php`).

Example: hello.php

PHP

```
<?php
    echo "Hello, World!";
?>
```

To run this script, place it in your web server's document root (e.g., `htdocs` for Apache) and access it through your web browser (e.g., `http://localhost/hello.php`). The browser will display Hello, World!

1.4. PHP tags and output statements

PHP code is always enclosed within specific tags that the PHP parser recognizes. The most common and recommended tag is the standard `<?php ... ?>` tag. There are also shorthand tags like `<?=` for echoing variables directly, but their use is generally discouraged for better compatibility and readability.

Output Statements:

PHP provides several constructs for outputting data to the browser or other output streams. The most frequently used are `echo` and `print`.

- **`echo`:** This is a language construct (not a true function) used to output one or more strings. It is slightly faster than `print` and does not return a value.
- **`print`:** This is also a language construct, similar to `echo`, but it can only output a single string and always returns a value of 1 (which makes it slightly slower than `echo`).

Key Differences between `echo` and `print`:

Feature	`echo`	`print`
Type	Language construct	Language construct
Return Value	None	Always returns 1
Multiple Args	Yes	No
Speed	Slightly faster	Slightly slower

2. PHP Syntax and Variables

Understanding PHP syntax and how to work with variables is fundamental to writing any PHP application. PHP is a loosely typed language, meaning you don't have to declare the data type of a variable before using it; PHP automatically converts the variable to the correct data type depending on its value.

2.1. PHP syntax rules and conventions

PHP syntax is similar to C, Java, and Perl. Here are some basic rules and conventions:

- **PHP Tags:** PHP code must always be enclosed within `<?php` and `?>` tags.
- **Statements:** Each statement in PHP must end with a semicolon (`;`).
- **Case Sensitivity:** Variable names are case-sensitive (e.g., `$name` is different from `$Name`). Function names and keywords (like `if` , `else` , `echo`) are generally case-insensitive, but it's good practice to use them consistently (e.g., always `echo` not `ECHO`).
- **Whitespace:** PHP ignores whitespace (spaces, tabs, newlines) outside of string literals, allowing for flexible formatting.
- **Comments:** Use `//` for single-line comments and `/* ... */` for multi-line comments.

Example:

PHP

```
<?php
    $firstName = "John"; // Variable names are case-sensitive
    $lastName = "Doe";

    echo "Hello, " . $firstName . " " . $lastName . "!"; // Statement ends
with a semicolon

    // This is a single-line comment
    /*
     * This is a
     * multi-line comment
     */
?>
```

2.2. Variable declaration and naming

In PHP, variables are used to store information. All variables in PHP start with a dollar sign (`$`) followed by the variable name. PHP variable names must start with a letter or an underscore, followed by any number of letters, numbers, or underscores.

Rules for PHP variable names:

- Must start with a `$` sign.
- Must start with a letter (a-z, A-Z) or an underscore (`_`).
- Can only contain alpha-numeric characters and underscores (`A-z` , `0-9` , and `_`).

- Are case-sensitive (`$age` and `$Age` are two different variables).

Example:

```php
<?php
    $name = "Alice";       // Valid
    $_age = 30;            // Valid
    $city_name = "New York"; // Valid
    $zipCode123 = "10001"; // Valid

    // Invalid variable names (will cause errors or unexpected behavior):
    // $123invalid = ""; // Cannot start with a number
    // $my-variable = ""; // Cannot contain hyphens

    echo "Name: " . $name . "<br>";
    echo "Age: " . $_age . "<br>";
?>
```

2.3. Data type basics

PHP supports several data types to hold different kinds of values. PHP is a loosely typed language, meaning you don't have to explicitly declare the data type of a variable. The data type is determined by the value assigned to it.

Common PHP Data Types:

- **String:** A sequence of characters (e.g., "Hello world!").
- **Integer:** A non-decimal number (e.g., 10, -5).
- **Float (floating point number):** A number with a decimal point or in exponential form (e.g., 10.5, 2.1e3).
- **Boolean:** Represents two possible states: TRUE or FALSE.
- **Array:** Stores multiple values in a single variable.
- **Object:** An instance of a class, storing both data and information on how to process that data.
- **NULL:** A special data type that can only have one value: NULL. It represents a variable with no value.
- **Resource:** A special variable, holding a reference to an external resource (e.g., a database connection, a file handle).

Example:

PHP

```
<?php
    $name = "Bob";          // String
    $age = 25;              // Integer
    $price = 19.99;         // Float
    $isStudent = true;      // Boolean
    $colors = array("Red", "Green", "Blue"); // Array
    $noValue = NULL;        // NULL

    echo "Name: " . $name . " (Type: " . gettype($name) . ")<br>";
    echo "Age: " . $age . " (Type: " . gettype($age) . ")<br>";
    echo "Price: " . $price . " (Type: " . gettype($price) . ")<br>";
    echo "Is Student: " . ($isStudent ? "True" : "False") . " (Type: " .
gettype($isStudent) . ")<br>";
    echo "Colors: " . implode(", ", $colors) . " (Type: " . gettype($colors)
. ")<br>";
    echo "No Value: " . var_export($noValue, true) . " (Type: " .
gettype($noValue) . ")<br>";
?>
```

2.4. Variable scope

Variable scope refers to the context within which a variable is defined and can be accessed. In PHP, variables can have four main scopes:

- **Local Scope:** Variables declared inside a function are local to that function and can only be accessed within it.
- **Global Scope:** Variables declared outside any function have a global scope and can be accessed from anywhere in the script, *except* inside functions without explicitly declaring them as `global` or using the `$GLOBALS` superglobal array.
- **Static Scope:** A `static` variable retains its value even after the function has finished executing. It is initialized only once.
- **Function Parameters:** These are local to the function and receive values passed during the function call.

Example:

PHP

```
<?php
    $globalVar = "I am a global variable."; // Global scope

    function testScope() {
        $localVar = "I am a local variable."; // Local scope
        echo $localVar . "<br>";
```

```
        // echo $globalVar; // This would cause an error: Undefined variable
$globalVar

        global $globalVar; // Accessing global variable inside function
        echo $globalVar . "<br>";

        echo $GLOBALS['globalVar'] . "<br>"; // Another way to access global
variable
    }

    function staticExample() {
        static $staticVar = 0; // Static variable
        $staticVar++;
        echo "Static variable: " . $staticVar . "<br>";
    }

    testScope();
    echo $globalVar . "<br>";
    // echo $localVar; // This would cause an error: Undefined variable
$localVar

    staticExample(); // Output: Static variable: 1
    staticExample(); // Output: Static variable: 2 (value retained)
?>
```

2.5. Constants and magic constants

Constants:

A constant is an identifier (name) for a simple value. As the name suggests, that value cannot change during the execution of the script. Constants are defined using the `define()` function or the `const` keyword.

Rules for PHP constants:

- Constants are case-sensitive by default (though `define()` allows for case-insensitivity).
- Conventionally, constant names are always in uppercase.
- Constants do not need a `$` sign before them.
- Constants can be accessed from anywhere in the script, regardless of scope.

Example:

PHP

```
<?php
    define("SITE_NAME", "My Awesome Website");
    const MAX_USERS = 100;
```

```
    echo SITE_NAME . "<br>";
    echo MAX_USERS . "<br>";

    function displayConstants() {
        echo SITE_NAME . " (accessed from function)<br>";
    }
    displayConstants();
?>
```

Magic Constants:

PHP provides a large number of predefined constants, known as

magic constants, that change value depending on where they are used. These constants start and end with two underscores (`__`).

Common Magic Constants:

Constant	Description
`__LINE__`	The current line number of the file.
`__FILE__`	The full path and filename of the file.
`__DIR__`	The directory of the file.
`__FUNCTION__`	The function name.
`__CLASS__`	The class name.
`__METHOD__`	The class method name.
`__NAMESPACE__`	The name of the current namespace.

Example:

PHP

```
<?php
    echo "Line number: " . __LINE__ . "<br>";
    echo "File path: " . __FILE__ . "<br>";
    echo "Directory: " . __DIR__ . "<br>";

    function myFunction() {
        echo "Function name: " . __FUNCTION__ . "<br>";
    }
    myFunction();
```

```
    class MyClass {
        public function myMethod() {
            echo "Class name: " . __CLASS__ . "<br>";
            echo "Method name: " . __METHOD__ . "<br>
";
        }
    }
    $obj = new MyClass();
    $obj->myMethod();
?>
```

3. PHP Data Types

PHP supports various data types to store different kinds of information. Understanding these types is crucial for effective programming, as PHP is a loosely typed language that performs automatic type conversion (type juggling) when necessary.

3.1. Scalar types (string, integer, float, boolean)

Scalar types represent single values. PHP has four scalar data types:

- **String:** Used for text. Strings can be enclosed in single quotes (`'`) or double quotes (`"`). Double quotes allow for variable parsing and escape sequences.
- **Integer:** Used for whole numbers (non-decimal). Integers can be positive or negative.
- **Float (Floating Point Number / Double):** Used for numbers with a decimal point or in exponential form.
- **Boolean:** Represents two possible states: `TRUE` or `FALSE`. Booleans are often used in conditional statements.

3.2. Compound types (array, object)

Compound types can hold multiple values or more complex structures.

- **Array:** An array is a special variable that can hold more than one value at a time. PHP arrays are ordered maps, meaning they associate values to keys. Keys can be integers (indexed arrays) or strings (associative arrays).
- **Object:** An object is an instance of a class. A class is a blueprint for creating objects, providing initial values for state (member variables) and implementations of behavior (member functions or methods).

3.3. Special types (NULL, resource)

PHP has two special data types:

- **NULL:** The `NULL` data type has only one value: `NULL`. A variable is considered `NULL` if it has been assigned the constant `NULL`, has not been assigned a value yet, or has been unset.
- **Resource:** A resource is a special variable, holding a reference to an external resource (like a database connection, a file handle, or an image canvas). Resources are created and used by special functions.

3.4. Type juggling and type casting

PHP is a loosely typed language, which means it often performs automatic type conversion, known as **type juggling**, when an operation requires a different data type. This can sometimes lead to unexpected results.

Type Juggling Example:

PHP

```
<?php
    $str = "10";
    $int = 5;

    $sum = $str + $int; // PHP juggles $str to an integer
    echo "Sum: " . $sum . " (Type: " . gettype($sum) . ")<br>"; // Output:
Sum: 15 (Type: integer)

    $bool = true;
    $result = $bool + $int; // PHP juggles $bool to 1
    echo "Result: " . $result . " (Type: " . gettype($result) . ")<br>"; //
Output: Result: 6 (Type: integer)

    $str2 = "hello";
    $sum2 = $str2 + $int; // "hello" is juggled to 0
    echo "Sum2: " . $sum2 . " (Type: " . gettype($sum2) . ")<br>"; //
Output: Sum2: 5 (Type: integer) - A warning will also be issued.
?>
```

Type Casting:

Type casting allows you to explicitly convert a value from one data type to another. This is done by placing the desired type in parentheses before the variable.

Common Type Casts:

- `(int)` or `(integer)`: Casts to integer
- `(float)` or `(double)` or `(real)`: Casts to float

- `(string)` : Casts to string
- `(bool)` or `(boolean)` : Casts to boolean
- `(array)` : Casts to array
- `(object)` : Casts to object
- `(unset)` : Casts to NULL

Example:

PHP

```
<?php
    $value = "100.50";

    $int_value = (int)$value;
    echo "Integer value: " . $int_value . " (Type: " . gettype($int_value) .
")<br>"; // Output: Integer value: 100 (Type: integer)

    $float_value = (float)$value;
    echo "Float value: " . $float_value . " (Type: " . gettype($float_value)
. ")<br>"; // Output: Float value: 100.5 (Type: double)

    $number = 123;
    $string_number = (string)$number;
    echo "String number: " . $string_number . " (Type: " .
gettype($string_number) . ")<br>"; // Output: String number: 123 (Type:
string)

    $zero = 0;
    $bool_zero = (bool)$zero;
    echo "Boolean zero: " . ($bool_zero ? "True" : "False") . " (Type: " .
gettype($bool_zero) . ")<br>"; // Output: Boolean zero: False (Type: boolean)
?>
```

3.5. Type checking functions

PHP provides several built-in functions to check the data type of a variable. These functions are useful for validation and ensuring that your code operates on the expected data types.

Common Type Checking Functions:

Function	Description
`gettype()`	Returns the type of a variable as a string.
`is_string()`	Checks if a variable is a string.

`is_int()`	Checks if a variable is an integer.
`is_float()`	Checks if a variable is a float.
`is_bool()`	Checks if a variable is a boolean.
`is_array()`	Checks if a variable is an array.
`is_object()`	Checks if a variable is an object.
`is_null()`	Checks if a variable is NULL.
`is_resource()`	Checks if a variable is a resource.
`is_numeric()`	Checks if a variable is a number or a numeric string.

Example:

PHP

```
<?php
    $data1 = "Hello";
    $data2 = 123;
    $data3 = 45.67;
    $data4 = true;
    $data5 = [1, 2, 3];
    $data6 = NULL;

    echo "Type of data1: " . gettype($data1) . "<br>";
    echo "Is data1 a string? " . (is_string($data1) ? "Yes" : "No") . "<br>";

    echo "Type of data2: " . gettype($data2) . "<br>";
    echo "Is data2 an integer? " . (is_int($data2) ? "Yes" : "No") . "<br>";

    echo "Type of data3: " . gettype($data3) . "<br>";
    echo "Is data3 a float? " . (is_float($data3) ? "Yes" : "No") . "<br>";

    echo "Type of data4: " . gettype($data4) . "<br>";
    echo "Is data4 a boolean? " . (is_bool($data4) ? "Yes" : "No") . "<br>";

    echo "Type of data5: " . gettype($data5) . "<br>";
    echo "Is data5 an array? " . (is_array($data5) ? "Yes" : "No") . "<br>";

    echo "Type of data6: " . gettype($data6) . "<br>";
    echo "Is data6 NULL? " . (is_null($data6) ? "Yes" : "No") . "<br>";

    $numeric_string = "123.45";
```

```
    echo "Is '" . $numeric_string . "' numeric? " .
(is_numeric($numeric_string) ? "Yes" : "No") . "<br>";
?>
```

4. PHP Operators

Operators are symbols that tell the PHP interpreter to perform specific mathematical, relational, or logical operations and produce a final result. PHP supports a wide range of operators.

4.1. Arithmetic operators

Arithmetic operators are used to perform common mathematical operations.

Operator	Name	Description	Example	Result
`+`	Addition	Sum of `$a` and `$b`.	`$a + $b`	
`-`	Subtraction	Difference of `$a` and `$b`.	`$a - $b`	
`*`	Multiplication	Product of `$a` and `$b`.	`$a * $b`	
`/`	Division	Quotient of `$a` and `$b`. Returns a float.	`$a / $b`	
`%`	Modulus	Remainder of `$a` divided by `$b`.	`$a % $b`	
`**`	Exponentiation	`$a` raised to the power of `$b` (PHP 5.6+).	`$a ** $b`	

Example:

PHP

```
<?php
    $x = 10;
    $y = 3;

    echo "Addition: " . ($x + $y) . "<br>";       // Output: 13
    echo "Subtraction: " . ($x - $y) . "<br>";    // Output: 7
    echo "Multiplication: " . ($x * $y) . "<br>"; // Output: 30
    echo "Division: " . ($x / $y) . "<br>";       // Output: 3.333...
    echo "Modulus: " . ($x % $y) . "<br>";        // Output: 1 (remainder of
10 / 3)
    echo "Exponentiation: " . ($x ** $y) . "<br>"; // Output: 1000 (10 to
```

```
the power of 3)
?>
```

4.2. Assignment operators

Assignment operators are used to assign values to variables. The basic assignment operator is `=`. Compound assignment operators perform an operation and then assign the result.

Operator	Example	Same As	Description
`=`	`$a = $b`		Assigns the value of `$b` to `$a`.
`+=`	`$a += $b`	`$a = $a + $b`	Adds `$b` to `$a`, then assigns to `$a`.
`-=`	`$a -= $b`	`$a = $a - $b`	Subtracts `$b` from `$a`, then assigns to `$a`.
`*=`	`$a *= $b`	`$a = $a * $b`	Multiplies `$a` by `$b`, then assigns to `$a`.
`/=`	`$a /= $b`	`$a = $a / $b`	Divides `$a` by `$b`, then assigns to `$a`.
`%=`	`$a %= $b`	`$a = $a % $b`	Modulus of `$a` by `$b`, then assigns to `$a`.
`**=`	`$a **= $b`	`$a = $a ** $b`	Exponentiates `$a` by `$b`, then assigns to `$a`.
`.`	`$a .= $b`	`$a = $a . $b`	Concatenates `$b` to `$a`, then assigns to `$a`.

Example:

PHP

```
<?php
    $x = 10;

    $x += 5; // $x is now 15 (10 + 5)
    echo "x after += 5: " . $x . "<br>";

    $x -= 3; // $x is now 12 (15 - 3)
    echo "x after -= 3: " . $x . "<br>";

    $x *= 2; // $x is now 24 (12 * 2)
    echo "x after *= 2: " . $x . "<br>";

    $x /= 4; // $x is now 6 (24 / 4)
    echo "x after /= 4: " . $x . "<br>";

    $text = "Hello";
    $text .= " World!"; // $text is now "Hello World!"
```

```
    echo "Text after .=: " . $text . "<br>";
?>
```

4.3. Comparison operators

Comparison operators are used to compare two values and return a Boolean (`TRUE` or `FALSE`) result.

Operator	Name	Description	Example	Result
`==`	Equal	`TRUE` if `$a` is equal to `$b` (after type juggling).	`$a == $b`	
`===`	Identical	`TRUE` if `$a` is equal to `$b` , and they are of the same type.	`$a === $b`	
`!=`	Not equal	`TRUE` if `$a` is not equal to `$b` .	`$a != $b`	
`<>`	Not equal	`TRUE` if `$a` is not equal to `$b` .	`$a <> $b`	
`!==`	Not identical	`TRUE` if `$a` is not equal to `$b` , or they are not of the same type.	`$a !== $b`	
`>`	Greater than	`TRUE` if `$a` is greater than `$b` .	`$a > $b`	
`<`	Less than	`TRUE` if `$a` is less than `$b` .	`$a < $b`	
`>=`	Greater than or equal to	`TRUE` if `$a` is greater than or equal to `$b` .	`$a >= $b`	
`<=`	Less than or equal to	`TRUE` if `$a` is less than or equal to `$b` .	`$a <= $b`	
`<=>`	Spaceship	Returns -1 if `$a < $b` , 0 if `$a == $b` , or 1 if `$a > $b` (PHP 7+).	`$a <=> $b`	

Example:

PHP

```
<?php
    $num1 = 10;
    $num2 = "10";
    $num3 = 5;

    echo "10 == '10': " . ($num1 == $num2 ? "True" : "False") . "<br>";   //
True (type juggling)
    echo "10 === '10': " . ($num1 === $num2 ? "True" : "False") . "<br>"; //
False (different types)
    echo "10 != 5: " . ($num1 != $num3 ? "True" : "False") . "<br>";    //
True
    echo "10 > 5: " . ($num1 > $num3 ? "True" : "False") . "<br>";     //
True
    echo "5 <= 10: " . ($num3 <= $num1 ? "True" : "False") . "<br>";   //
True

    echo "10 <=> 5: " . ($num1 <=> $num3) . "<br>"; // Output: 1
    echo "5 <=> 10: " . ($num3 <=> $num1) . "<br>"; // Output: -1
    echo "10 <=> 10: " . ($num1 <=> $num1) . "<br>"; // Output: 0
?>
```

4.4. Logical operators

Logical operators are used to combine conditional statements.

Operator	Name	Description	Example	Result
`and`	And	`TRUE` if both `$a` and `$b` are `TRUE`.	`$a and $b`	
`or`	Or	`TRUE` if either `$a` or `$b` is `TRUE`.	`$a or $b`	
`xor`	Xor	`TRUE` if either `$a` or `$b` is `TRUE`, but not both.	`$a xor $b`	
`&&`	And	`TRUE` if both `$a` and `$b` are `TRUE`.	`$a && $b`	
`` ` ``		`` ` ``	Or	`TRUE` if either `$a` or `$b` is `TRUE`.	`` `$a ``		`$b ` ``

`!`	Not	`TRUE` if `$a` is not `TRUE`.	`!$a`	

Note on `and` / `or` vs. `&&` / `||`: The `and` and `or` operators have lower precedence than `&&` and `||`. This can lead to different results in complex expressions. It's generally recommended to use `&&` and `||` for logical operations.

Example:

PHP

```
<?php
    $age = 25;
    $isStudent = true;

    if ($age > 18 && $isStudent) {
        echo "Eligible for student discount.<br>";
    }

    $hasLicense = true;
    $hasCar = false;

    if ($hasLicense || $hasCar) {
        echo "Can drive or has a car.<br>";
    }

    $isAdmin = false;
    if (!$isAdmin) {
        echo "User is not an administrator.<br>";
    }
?>
```

4.5. Increment/Decrement operators

Increment and decrement operators are used to increase or decrease the value of a variable by one.

Operator	Name	Description	Example	Result
`++$a`	Pre-increment	Increments `$a` by one, then returns `$a`.	`$a = 5; echo ++$a;`	`6` (then `$a` is `6`)
`$a++`	Post-increment	Returns `$a`, then increments `$a` by one.	`$a = 5; echo $a++;`	`5` (then `$a` is `6`)

`--$a`	Pre-decrement	Decrements `$a` by one, then returns `$a`.	`$a = 5; echo --$a;`	`4` (then `$a` is `4`)
`$a--`	Post-decrement	Returns `$a`, then decrements `$a` by one.	`$a = 5; echo $a--;`	`5` (then `$a` is `4`)

Example:

PHP

```
<?php
    $counter = 5;

    echo "Pre-increment: " . (++$counter) . "<br>"; // Output: 6, $counter
is now 6
    echo "Post-increment: " . ($counter++) . "<br>"; // Output: 6, $counter
is now 7
    echo "Current counter: " . $counter . "<br>";   // Output: 7

    echo "Pre-decrement: " . (--$counter) . "<br>";  // Output: 6, $counter
is now 6
    echo "Post-decrement: " . ($counter--) . "<br>"; // Output: 6, $counter
is now 5
    echo "Current counter: " . $counter . "<br>";   // Output: 5
?>
```

5. PHP Control Structures

Control structures are fundamental to programming as they allow you to control the flow of execution in your script based on conditions or to repeat blocks of code. This section covers conditional statements and loops.

5.1. if, else, elseif statements

Conditional statements allow you to execute different blocks of code based on whether a specified condition is true or false.

- **`if` statement:** Executes a block of code if the specified condition is true.
- **`if...else` statement:** Executes one block of code if the condition is true, and another block if the condition is false.
- **`if...elseif...else` statement:** Allows you to specify multiple conditions. It executes different blocks of code for different conditions.

5.2. switch statements

The `switch` statement is used to perform different actions based on different conditions. It is an alternative to long `if...elseif...else` chains when you are comparing a single variable against multiple possible values.

Example:

PHP

```
<?php
    $dayOfWeek = "Monday";

    switch ($dayOfWeek) {
        case "Monday":
            echo "It's the start of the week.<br>";
            break;
        case "Friday":
            echo "It's almost the weekend!<br>";
            break;
        case "Sunday":
            echo "Enjoy your Sunday.<br>";
            break;
        default:
            echo "It's a regular weekday.<br>";
    }

    $num = 2;
    switch ($num) {
        case 1:
        case 2:
        case 3:
            echo "Number is 1, 2, or 3.<br>";
            break;
        default:
            echo "Number is something else.<br>";
    }
?>
```

Explanation:

- The `switch` statement evaluates the expression once.
- The value of the expression is compared with the values of each `case`.
- If a match is found, the code block associated with that `case` is executed.
- The `break` keyword is crucial; it terminates the `switch` statement. Without `break`, the execution would

fall through to the next `case`.

- The `default` keyword specifies the code to run if there is no match.

5.3. while, do-while loops

Loops are used to execute a block of code repeatedly as long as a specified condition is true.

- **`while` loop:** Executes a block of code as long as the specified condition is true. The condition is evaluated *before* each iteration.
- **`do...while` loop:** Executes a block of code once, and then repeats the loop as long as the specified condition is true. The condition is evaluated *after* each iteration, guaranteeing that the block of code is executed at least once.

5.4. for, foreach loops

- **`for` loop:** Used when you know how many times you want to execute a block of code. It consists of three parts: initialization, condition, and increment/decrement.
- **`foreach` loop:** Specifically designed to iterate over arrays and objects. It provides an easy way to loop through each item in a collection.

5.5. break and continue statements

- **`break` statement:** Used to terminate the execution of the current loop (or `switch` statement) immediately. Control passes to the statement immediately following the terminated loop.
- **`continue` statement:** Used to skip the rest of the current iteration of the loop and proceed to the next iteration. Control passes to the condition (for `while` and `do-while`) or the increment/decrement part (for `for` loop).

6. PHP Functions

Functions are blocks of code that perform a specific task and can be reused throughout your program. PHP has a vast library of built-in functions, and you can also define your own custom functions.

6.1. Function declaration and calling

To declare a function in PHP, you use the `function` keyword, followed by the function name, parentheses (which may contain parameters), and curly braces `{}` that enclose the function's code.

Syntax:

PHP

```
function functionName(parameter1, parameter2, ...) {
    // code to be executed
    return value; // Optional
}
```

To call a function, you simply use its name followed by parentheses, passing any required arguments.

Example:

PHP

```
<?php
    function sayHello() {
        echo "Hello, World!<br>";
    }

    // Calling the function
    sayHello();

    function greet($name) {
        echo "Hello, " . $name . "!<br>";
    }

    // Calling the function with an argument
    greet("Alice");
    greet("Bob");
?>
```

6.2. Parameters and return values

Functions can accept input values called **parameters** (or arguments) and can return a value using the `return` statement.

- **Parameters:** Variables listed inside the parentheses in the function definition. They act as placeholders for the values that will be passed into the function when it is called.
- **Return Values:** A function can return a single value (of any type) using the `return` statement. Once a `return` statement is executed, the function terminates, and the returned value is sent back to the caller.

Example:

PHP

```php
<?php
    function addNumbers($num1, $num2) {
        $sum = $num1 + $num2;
        return $sum;
    }

    $result = addNumbers(10, 5);
    echo "Sum: " . $result . "<br>"; // Output: Sum: 15

    function multiply($a, $b = 2) { // $b has a default value
        return $a * $b;
    }

    echo "Multiply (5 * 2): " . multiply(5) . "<br>"; // Uses default value
for $b
    echo "Multiply (5 * 3): " . multiply(5, 3) . "<br>"; // Overrides
default value

    function createFullName($firstName, $lastName) {
        return $firstName . " " . $lastName;
    }

    $fullName = createFullName("Jane", "Doe");
    echo "Full Name: " . $fullName . "<br>";
?>
```

6.3. Variable scope in functions

As discussed in Section 2.4, variables in PHP have different scopes. Inside a function, variables declared are local to that function. To access global variables within a function, you need to use the `global` keyword or the `$GLOBALS` superglobal array.

Example:

PHP

```php
<?php
    $globalMessage = "This is a global message.";

    function displayMessage() {
        // echo $globalMessage; // This would cause an error (undefined
variable)

        global $globalMessage; // Accessing the global variable
        echo $globalMessage . "<br>";

        $localMessage = "This is a local message.";
```

```
        echo $localMessage . "<br>";
    }

    displayMessage();
    // echo $localMessage; // This would cause an error (undefined variable)

    echo $globalMessage . "<br>";

    // Using $GLOBALS superglobal
    function displayGlobalUsingGlobals() {
        echo $GLOBALS["globalMessage"] . " (using $GLOBALS)<br>";
    }
    displayGlobalUsingGlobals();
?>
```

6.4. Anonymous functions

Anonymous functions (also known as closures) are functions that have no specified name. They can be assigned to variables and passed as arguments to other functions. They are particularly useful as callback functions.

Example:

PHP

```
<?php
    $greet = function($name) {
        echo "Hello, " . $name . " from an anonymous function!<br>";
    };

    $greet("Charlie");

    // Anonymous function as a callback
    $numbers = [1, 2, 3, 4, 5];
    $squaredNumbers = array_map(function($number) {
        return $number * $number;
    }, $numbers);

    echo "Squared numbers: " . implode(", ", $squaredNumbers) . "<br>";

    // Anonymous function using 'use' to inherit variables from parent scope
    $factor = 10;
    $multiplyByFactor = function($num) use ($factor) {
        return $num * $factor;
    };
```

```
    echo "5 multiplied by factor: " . $multiplyByFactor(5) . "<br>";
?>
```

6.5. Built-in functions

PHP comes with a vast library of built-in functions that perform a wide range of tasks, from string manipulation and mathematical calculations to file handling and database interaction. Using these functions saves development time and ensures robust code.

Examples of common built-in functions:

- **String Functions:**
  - `strlen()` : Returns the length of a string.
  - `str_replace()` : Replaces all occurrences of a substring with another substring.
  - `strtoupper()` : Converts a string to uppercase.
  - `strtolower()` : Converts a string to lowercase.
- **Array Functions:**
  - `count()` : Returns the number of elements in an array.
  - `array_push()` : Adds one or more elements to the end of an array.
  - `array_pop()` : Removes the last element from an array.
  - `in_array()` : Checks if a value exists in an array.
- **Mathematical Functions:**
  - `rand()` : Generates a random integer.
  - `sqrt()` : Returns the square root of a number.
  - `round()` : Rounds a floating-point number.
- **Date and Time Functions:**
  - `date()` : Formats a local date/time.
  - `time()` : Returns the current Unix timestamp.

Example:

PHP

```
<?php
    // String functions
    $text = "Hello, PHP World!";
    echo "Length of text: " . strlen($text) . "<br>";
    echo "Replaced text: " . str_replace("PHP", "Awesome", $text) . "<br>";
    echo "Uppercase: " . strtoupper($text) . "<br>";
```

```
    // Array functions
    $numbers = [10, 20, 30];
    echo "Number of elements: " . count($numbers) . "<br>";
    array_push($numbers, 40);
    echo "Array after push: " . implode(", ", $numbers) . "<br>";
    echo "Does 20 exist in array? " . (in_array(20, $numbers) ? "Yes" :
"No") . "<br>";

    // Mathematical functions
    echo "Random number (1-100): " . rand(1, 100) . "<br>";
    echo "Square root of 64: " . sqrt(64) . "<br>";

    // Date and Time functions
    echo "Current date: " . date("Y-m-d H:i:s") . "<br>";
?>
```

7. PHP Arrays

Arrays are powerful data structures that allow you to store multiple values in a single variable. PHP arrays are highly flexible and can serve as indexed arrays, associative arrays, or even multidimensional arrays.

7.1. Indexed arrays

Indexed arrays are arrays where each element has a numeric index, starting from 0 by default. You can create them by simply assigning values or by using the `array()` constructor or square brackets `[]`.

Example:

PHP

```
<?php
    // Method 1: Assigning values directly (index starts from 0)
    $fruits[0] = "Apple";
    $fruits[1] = "Banana";
    $fruits[2] = "Cherry";

    echo "Fruit at index 0: " . $fruits[0] . "<br>";

    // Method 2: Using array() constructor
    $cars = array("Volvo", "BMW", "Toyota");
    echo "Car at index 1: " . $cars[1] . "<br>";

    // Method 3: Using short array syntax (PHP 5.4+)
```

```
    $animals = ["Dog", "Cat", "Bird"];
    echo "Animal at index 2: " . $animals[2] . "<br>";

    // Adding elements to an indexed array (automatically assigns next
available index)
    $animals[] = "Fish";
    echo "New animal: " . $animals[3] . "<br>";

    // Looping through an indexed array
    echo "<p>All fruits:</p>";
    for ($i = 0; $i < count($fruits); $i++) {
        echo $fruits[$i] . "<br>";
    }
?>
```

7.2. Associative arrays

Associative arrays use named keys that you assign to them, instead of numeric indices. This allows you to use meaningful names to reference array elements.

Example:

PHP

```
<?php
    // Method 1: Using array() constructor with key-value pairs
    $student = array(
        "name" => "Alice",
        "age" => 20,
        "major" => "Computer Science"
    );

    echo "Student Name: " . $student["name"] . "<br>";
    echo "Student Age: " . $student["age"] . "<br>";

    // Method 2: Using short array syntax (PHP 5.4+)
    $person = [
        "firstName" => "Bob",
        "lastName" => "Smith",
        "email" => "bob.smith@example.com"
    ];

    echo "Person Email: " . $person["email"] . "<br>";

    // Adding/modifying elements
    $person["phone"] = "123-456-7890";
    echo "Person Phone: " . $person["phone"] . "<br>";
```

```
    // Looping through an associative array
    echo "<p>Student Details:</p>";
    foreach ($student as $key => $value) {
        echo $key . ": " . $value . "<br>";
    }
?>
```

7.3. Multidimensional arrays

A multidimensional array is an array containing one or more arrays. This allows you to store data in a table-like structure (rows and columns) or even more complex hierarchies.

Example:

PHP

```
<?php
    // A 2-dimensional array (array of arrays)
    $studentsGrades = [
        "John" => [
            "Math" => 90,
            "Science" => 85,
            "English" => 92
        ],
        "Jane" => [
            "Math" => 95,
            "Science" => 88,
            "English" => 90
        ],
        "Mike" => [
            "Math" => 78,
            "Science" => 80,
            "English" => 82
        ]
    ];

    echo "John's Math grade: " . $studentsGrades["John"]["Math"] . "<br>";
    echo "Jane's English grade: " . $studentsGrades["Jane"]["English"] . "
<br>";

    // Looping through a 2D array
    echo "<p>All Students' Grades:</p>";
    foreach ($studentsGrades as $studentName => $subjects) {
        echo "<h3>" . $studentName . "</h3>";
        foreach ($subjects as $subjectName => $grade) {
            echo $subjectName . ": " . $grade . "<br>";
        }
    }
```

```
    // A 3-dimensional array (e.g., storing city temperatures for different
days)
    $cityTemperatures = [
        "New York" => [
            "Monday" => ["morning" => 15, "afternoon" => 20],
            "Tuesday" => ["morning" => 12, "afternoon" => 18]
        ],
        "London" => [
            "Monday" => ["morning" => 10, "afternoon" => 14],
            "Tuesday" => ["morning" => 8, "afternoon" => 12]
        ]
    ];

    echo "New York Monday Afternoon Temperature: " . $cityTemperatures["New
York"]["Monday"]["afternoon"] . "°C<br>";
?>
```

7.4. Array functions

PHP provides a rich set of built-in functions for manipulating arrays. These functions allow you to perform various operations like sorting, searching, merging, filtering, and more.

Common Array Functions:

Function	Description
`count()`	Returns the number of elements in an array.
`sort()`	Sorts an indexed array in ascending order.
`rsort()`	Sorts an indexed array in descending order.
`asort()`	Sorts an associative array by value in ascending order.
`ksort()`	Sorts an associative array by key in ascending order.
`array_push()`	Adds one or more elements to the end of an array.
`array_pop()`	Removes the last element from an array.
`array_merge()`	Merges one or more arrays.
`in_array()`	Checks if a value exists in an array.
`array_keys()`	Returns all the keys or a subset of the keys of an array.
`array_values()`	Returns all the values of an array.

`array_unique()`	Removes duplicate values from an array.

Example:

PHP

```
<?php
    $numbers = [4, 2, 8, 1, 5];
    echo "Original numbers: " . implode(", ", $numbers) . "<br>";

    sort($numbers);
    echo "Sorted numbers (asc): " . implode(", ", $numbers) . "<br>";

    $ages = ["Peter" => 35, "Ben" => 37, "Joe" => 43];
    asort($ages); // Sort by value
    echo "Sorted ages (by value): ";
    foreach ($ages as $name => $age) {
        echo $name . ": " . $age . " ";
    }
    echo "<br>";

    array_push($numbers, 9, 7);
    echo "Numbers after push: " . implode(", ", $numbers) . "<br>";

    $lastElement = array_pop($numbers);
    echo "Popped element: " . $lastElement . ", Array now: " . implode(", ",
$numbers) . "<br>";

    $moreNumbers = [10, 1, 20];
    $mergedArray = array_merge($numbers, $moreNumbers);
    echo "Merged array: " . implode(", ", $mergedArray) . "<br>";

    echo "Is 8 in merged array? " . (in_array(8, $mergedArray) ? "Yes" :
"No") . "<br>";

    $uniqueArray = array_unique([1, 2, 2, 3, 1, 4]);
    echo "Unique array: " . implode(", ", $uniqueArray) . "<br>";
?>
```

7.5. Array iteration

Iterating through arrays is a common task in PHP. The `foreach` loop is the most common and convenient way to iterate over array elements.

Example: Iterating Indexed Arrays

```
PHP
<?php
    $fruits = ["Apple", "Banana", "Orange"];

    echo "<p>Iterating Indexed Array with foreach:</p>";
    foreach ($fruits as $fruit) {
        echo $fruit . "<br>";
    }

    echo "<p>Iterating Indexed Array with for loop (using index):</p>";
    for ($i = 0; $i < count($fruits); $i++) {
        echo $fruits[$i] . "<br>";
    }
?>
```

Example: Iterating Associative Arrays

```
PHP

<?php
    $student = [
        "name" => "David",
        "age" => 21,
        "course" => "Physics"
    ];

    echo "<p>Iterating Associative Array with foreach:</p>";
    foreach ($student as $key => $value) {
        echo ucfirst($key) . ": " . $value . "<br>"; // ucfirst()
capitalizes the first letter
    }
?>
```

8. PHP Strings

Strings are fundamental data types in PHP, used to represent sequences of characters. PHP provides extensive functionality for creating, manipulating, and formatting strings.

8.1. String creation and manipulation

Strings can be created using single quotes (`'`) or double quotes (`"`). The choice between them often depends on whether you need variable parsing or escape sequences.

- **Single Quoted Strings:** Literal interpretation. Variables and most escape sequences are not parsed.

- **Double Quoted Strings:** Variables are parsed, and most escape sequences (like `\n` for newline, `\t` for tab) are interpreted.

Example: String Creation

PHP

```
<?php
    $name = "Alice";

    $singleQuoteString = 'Hello, $name!'; // Output: Hello, $name!
    $doubleQuoteString = "Hello, $name!"; // Output: Hello, Alice!

    echo $singleQuoteString . "<br>";
    echo $doubleQuoteString . "<br>";

    $multilineString = "This is a
    multi-line string.";
    echo nl2br($multilineString) . "<br>"; // nl2br converts newlines to
<br> for browser display
?>
```

String Manipulation:

PHP offers numerous functions for manipulating strings, such as getting length, finding substrings, replacing parts, and changing case.

Example: String Manipulation

PHP

```
<?php
    $text = "  Hello PHP World!  ";

    echo "Original: '" . $text . "'<br>";
    echo "Length: " . strlen($text) . "<br>";
    echo "Trimmed: '" . trim($text) . "'<br>"; // Removes whitespace from
beginning and end
    echo "Uppercase: " . strtoupper($text) . "<br>";
    echo "Lowercase: " . strtolower($text) . "<br>";

    $substring = substr($text, 5, 3); // Start at index 5, get 3 characters
    echo "Substring (PHP): '" . $substring . "'<br>";

    $replacedText = str_replace("PHP", "Awesome", $text);
    echo "Replaced: '" . $replacedText . "'<br>";

    $position = strpos($text, "PHP");
```

```
    echo "Position of 'PHP': " . ($position !== false ? $position : "Not
found") . "<br>";
?>
```

8.2. String functions

PHP has a comprehensive set of built-in string functions. Here are some commonly used ones:

Function	Description
`strlen()`	Returns the length of a string.
`str_word_count()`	Counts the number of words in a string.
`strrev()`	Reverses a string.
`strpos()`	Finds the position of the first occurrence of a substring in a string.
`str_replace()`	Replaces all occurrences of a substring with another substring.
`substr()`	Returns a part of a string.
`trim()`	Removes whitespace or other characters from both sides of a string.
`ltrim()`	Removes whitespace or other characters from the left side of a string.
`rtrim()`	Removes whitespace or other characters from the right side of a string.
`strtoupper()`	Converts a string to uppercase.
`strtolower()`	Converts a string to lowercase.
`ucfirst()`	Converts the first character of a string to uppercase.
`lcfirst()`	Converts the first character of a string to lowercase.
`ucwords()`	Converts the first character of each word in a string to uppercase.
`implode()`	Joins array elements with a string.
`explode()`	Splits a string by a string.

Example:

```
PHP
```

```php
<?php
    $sentence = "PHP is a powerful scripting language.";

    echo "Original: " . $sentence . "<br>";
    echo "Word count: " . str_word_count($sentence) . "<br>";
    echo "Reversed: " . strrev($sentence) . "<br>";

    $email = "  test@example.com  ";
    echo "Trimmed email: '" . trim($email) . "'<br>";

    $csvData = "apple,banana,cherry";
    $fruitsArray = explode(",", $csvData);
    echo "Exploded array: " . implode(" | ", $fruitsArray) . "<br>";

    $formattedName = "john doe";
    echo "Formatted Name: " . ucwords($formattedName) . "<br>";
?>
```

8.3. String concatenation

String concatenation is the process of joining two or more strings together to form a single string. In PHP, the dot (`.`) operator is used for string concatenation.

Example:

PHP

```php
<?php
    $firstName = "Alice";
    $lastName = "Smith";

    $fullName = $firstName . " " . $lastName; // Concatenating strings with
a space
    echo "Full Name: " . $fullName . "<br>";

    $greeting = "Hello";
    $greeting .= " World!"; // Using the concatenation assignment operator
(.=)
    echo $greeting . "<br>";

    $age = 30;
    $info = "Name: " . $fullName . ", Age: " . $age . "."; // Concatenating
strings and numbers
    echo $info . "<br>";
?>
```

8.4. String formatting

String formatting involves presenting strings in a specific layout or style. PHP provides functions like `sprintf()` and `printf()` for formatted output, similar to C-style formatting.

- **`sprintf()` :** Returns a formatted string.
- **`printf()` :** Prints a formatted string.

Format Specifiers (common ones):

- `%s` : String
- `%d` : Signed decimal integer
- `%f` : Floating-point number
- `%%` : A literal percent sign

Example:

PHP

```
<?php
    $name = "Bob";
    $amount = 123.456;
    $quantity = 5;

    $formattedString = sprintf("Hello, %s! Your total is $%.2f for %d
items.", $name, $amount, $quantity);
    echo $formattedString . "<br>";

    printf("The temperature is %.1f degrees Celsius.<br>", 25.78);

    $product = "Laptop";
    $price = 999.99;
    $stock = 10;

    printf("Product: %-10s | Price: $%8.2f | Stock: %d<br>", $product,
$price, $stock);
    // %-10s: left-align string in a field of 10 characters
    // %8.2f: float with 2 decimal places, right-aligned in a field of 8
characters
?>
```

8.5. Regular expressions

Regular expressions (regex) are powerful patterns used for searching, matching, and manipulating strings based on complex rules. PHP uses PCRE (Perl Compatible Regular Expressions) functions, which are prefixed with `preg_` .

Common `preg_` functions:

- `preg_match()` : Performs a regular expression match.
- `preg_match_all()` : Performs a global regular expression match.
- `preg_replace()` : Performs a regular expression search and replace.
- `preg_split()` : Splits a string by a regular expression.

Basic Regex Syntax Elements:

- `/pattern/modifiers` : Delimiters (usually `/`), pattern, and optional modifiers.
- `.` : Any character (except newline).
- `*` : Zero or more occurrences of the preceding character/group.
- `+` : One or more occurrences.
- `?` : Zero or one occurrence.
- `[abc]` : Any character from the set a, b, or c.
- `[^abc]` : Any character NOT from the set.
- `[0-9]` : Any digit.
- `[a-z]` : Any lowercase letter.
- `\d` : Any digit (equivalent to `[0-9]`).
- `\w` : Any word character (alphanumeric + underscore).
- `\s` : Any whitespace character.
- `^` : Start of the string.
- `$` : End of the string.

Example:

PHP

```
<?php
    $text = "The quick brown fox jumps over the lazy dog.";

    // Check if string contains "fox"
    if (preg_match("/fox/", $text)) {
        echo "'fox' found in the text.<br>";
    }

    // Replace all occurrences of "the" (case-insensitive)
    $newText = preg_replace("/the/i", "a", $text);
    echo "Replaced text: " . $newText . "<br>";
```

```
    // Extract all numbers from a string
    $data = "Item1: 123, Item2: 45, Item3: 6789";
    preg_match_all("/\d+/", $data, $matches);
    echo "Numbers found: " . implode(", ", $matches[0]) . "<br>";

    // Validate email format (simple example)
    $email = "test@example.com";
    if (preg_match("/^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/",
$email)) {
        echo "Valid email format.<br>";
    } else {
        echo "Invalid email format.<br>";
    }
?>
```

9. PHP Forms and User Input

Handling HTML forms and processing user input is a core aspect of web development. PHP provides robust ways to collect, validate, and utilize data submitted through forms.

9.1. HTML forms

HTML forms are used to collect user input. They consist of various input elements (text fields, checkboxes, radio buttons, submit buttons, etc.) enclosed within `<form>` tags. The `action` attribute specifies where the form data should be sent, and the `method` attribute (`GET` or `POST`) defines how the data is sent.

Example: Basic HTML Form (`index.html` or `index.php`)

HTML

```
<!DOCTYPE html>
<html>
<head>
    <title>User Registration</title>
</head>
<body>

    <h2>Register Here</h2>
    <form action="process.php" method="post">
        <label for="name">Name:</label><br>
        <input type="text" id="name" name="username" required><br><br>

        <label for="email">Email:</label><br>
        <input type="email" id="email" name="email" required><br><br>
```

```
        <input type="submit" value="Submit">
    </form>

</body>
</html>
```

9.2. Form submission methods

HTML forms can submit data using two primary HTTP methods: `GET` and `POST`.

- **`GET` Method:**
  - Appends form data to the URL as query strings (e.g., `process.php?username=John&email=john@example.com`).
  - Data is visible in the URL, making it unsuitable for sensitive information.
  - Limited by URL length (typically around 2048 characters).
  - Suitable for non-sensitive data, search queries, or when users might want to bookmark the URL.
- **`POST` Method:**
  - Sends form data in the HTTP request body.
  - Data is not visible in the URL, making it more secure for sensitive information (like passwords).
  - No practical limits on data size.
  - Suitable for submitting large amounts of data, sensitive data, or when the submission results in changes on the server (e.g., creating a new record).

Example: `process.php` (to handle `POST` data)

PHP

```
<?php
    if ($_SERVER["REQUEST_METHOD"] == "POST") {
        $username = $_POST["username"];
        $email = $_POST["email"];

        echo "<h2>Registration Successful!</h2>";
        echo "<p>Welcome, " . htmlspecialchars($username) . "!</p>";
        echo "<p>Your email: " . htmlspecialchars($email) . "</p>";
    } else {
        echo "<p>Form not submitted via POST method.</p>";
    }
?>
```

9.3. Handling form data

PHP provides superglobal arrays to access form data: `$_GET`, `$_POST`, and `$_REQUEST`.

- **`$_GET`:** An associative array of variables passed to the current script via the URL parameters.
- **`$_POST`:** An associative array of variables passed to the current script via the HTTP POST method.
- **`$_REQUEST`:** An associative array that by default contains the contents of `$_GET`, `$_POST`, and `$_COOKIE`.

It's crucial to always check if form fields are set and not empty before using them, typically using `isset()` and `empty()`.

Example: Handling Form Data (`handle_data.php`)

PHP

```
<?php
    if (isset($_POST["submit"])) { // Assuming a submit button named "submit"
        $name = $_POST["username"] ?? ''; // Null coalescing operator (PHP
7+)
        $email = $_POST["email"] ?? '';

        if (!empty($name) && !empty($email)) {
            echo "<h2>Received Data:</h2>";
            echo "Name: " . htmlspecialchars($name) . "<br>";
            echo "Email: " . htmlspecialchars($email) . "<br>";
        } else {
            echo "<p>Please fill in all required fields.</p>";
        }
    } else {
        echo "<p>Form not submitted.</p>";
    }
?>
```

HTML Form (`form.html` or `form.php`)

HTML

```
<!DOCTYPE html>
<html>
<head>
    <title>Data Input</title>
</head>
<body>
```

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

</body>
</html>
```

9.4. Input validation

Input validation is the process of ensuring that user-supplied data meets specific criteria before it is processed or stored. This is critical for security, data integrity, and preventing errors. Never trust user input!

Key aspects of input validation:

- **Required Fields:** Check if mandatory fields are filled.
- **Data Type Validation:** Ensure data is of the expected type (e.g., number, email, URL).
- **Format Validation:** Check if data adheres to a specific format (e.g., email address pattern, phone number format).
- **Range/Length Validation:** Ensure numeric values are within a certain range, or strings are within a certain length.
- **Sanitization:** Remove or escape potentially harmful characters (e.g., HTML tags, SQL injection attempts).

PHP provides functions like `filter_var()` for validation and sanitization, and `preg_match()` for custom pattern matching.

Example: Input Validation (`validate_input.php`)

PHP

```
<?php
    $nameErr = $emailErr = $websiteErr = "";
    $name = $email = $website = "";

    if ($_SERVER["REQUEST_METHOD"] == "POST") {
        // Validate Name
        if (empty($_POST["name"])) {
            $nameErr = "Name is required";
        } else {
            $name = test_input($_POST["name"]);
            // Check if name contains only letters and whitespace
            if (!preg_match("/^[a-zA-Z ]*$/", $name)) {
```

```
                $nameErr = "Only letters and white space allowed";
            }
        }

        // Validate Email
        if (empty($_POST["email"])) {
            $emailErr = "Email is required";
        } else {
            $email = test_input($_POST["email"]);
            // Check if email address is well-formed
            if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
                $emailErr = "Invalid email format";
            }
        }

        // Validate Website (optional field)
        if (!empty($_POST["website"])) {
            $website = test_input($_POST["website"]);
            // Check if URL address syntax is valid
            if (!preg_match("/\b(?:(?:https?|ftp):\/\/|www\.)[-a-z0-9+&@#\/%?
=~_|!:,.;]*[-a-z0-9+&@#\/%=~_|]/i", $website)) {
                $websiteErr = "Invalid URL";
            }
        }
    }

    function test_input($data) {
        $data = trim($data);
        $data = stripslashes($data);
        $data = htmlspecialchars($data);
        return $data;
    }
?>

<!DOCTYPE html>
<html>
<head>
    <title>Form Validation</title>
    <style>
        .error {color: #FF0000;}
    </style>
</head>
<body>

    <h2>PHP Form Validation Example</h2>
    <p><span class="error">* required field</span></p>
    <form method="post" action="<?php echo
htmlspecialchars($_SERVER["PHP_SELF"]);?>">
```

```
        Name: <input type="text" name="name" value="<?php echo $name;?>">
        <span class="error">* <?php echo $nameErr;?></span>
        <br><br>
        E-mail: <input type="text" name="email" value="<?php echo $email;?>">
        <span class="error">* <?php echo $emailErr;?></span>
        <br><br>
        Website: <input type="text" name="website" value="<?php echo
$website;?>">
        <span class="error"><?php echo $websiteErr;?></span>
        <br><br>
        <input type="submit" name="submit" value="Submit">
    </form>

    <?php
        if ($_SERVER["REQUEST_METHOD"] == "POST" && empty($nameErr) &&
empty($emailErr) && empty($websiteErr)) {
            echo "<h2>Your Input:</h2>";
            echo "Name: " . $name . "<br>";
            echo "Email: " . $email . "<br>";
            echo "Website: " . $website . "<br>";
        }
    ?>

</body>
</html>
```

9.5. Security considerations

Form handling is a common entry point for malicious attacks. It's paramount to implement robust security measures to protect your application and user data. Never trust user input, and always validate and sanitize it.

Common Security Threats and Mitigations:

- **SQL Injection:** Occurs when an attacker inserts malicious SQL code into input fields, which is then executed by the database. **Mitigation:** Use prepared statements with parameterized queries (see Section 22.2) instead of concatenating user input directly into SQL queries.
- **Cross-Site Scripting (XSS):** Involves injecting malicious client-side scripts (e.g., JavaScript) into web pages viewed by other users. **Mitigation:** Always sanitize output using `htmlspecialchars()` or `strip_tags()` when displaying user-supplied data to prevent it from being interpreted as HTML or JavaScript.
- **Cross-Site Request Forgery (CSRF):** Tricks a user's browser into making an unwanted request to a web application where they are authenticated. **Mitigation:** Implement

CSRF tokens (unique, unpredictable tokens) in your forms. The server generates a token, embeds it in the form, and verifies it upon submission.

- **File Upload Vulnerabilities:** Malicious files (e.g., PHP scripts) can be uploaded and executed on the server. **Mitigation:**
  - Validate file type (MIME type, not just extension).
  - Limit file size.
  - Store uploaded files outside the web root or with restricted permissions.
  - Rename uploaded files to prevent execution.
  - Scan uploaded files for malware.
- **Session Hijacking/Fixation:** Attackers steal or fixate a user's session ID to impersonate them. **Mitigation:**
  - Use `session_regenerate_id()` after login to prevent session fixation.
  - Set `HttpOnly` flag on session cookies to prevent JavaScript access.
  - Use `Secure` flag for HTTPS-only cookies.
  - Implement session timeouts.

Example: Basic XSS Prevention with `htmlspecialchars()`

PHP

```
<?php
    $userInput = "<script>alert('XSS Attack!');</script><h1>Malicious
Title</h1>";

    // Without htmlspecialchars() - vulnerable to XSS
    echo "<h2>Unsafe Output:</h2>";
    echo $userInput;

    // With htmlspecialchars() - safe output
    echo "<h2>Safe Output:</h2>";
    echo htmlspecialchars($userInput);
?>
```

10. PHP Form Validation

Form validation is a critical step in handling user input, ensuring data quality and application security. It involves checking if the submitted data meets predefined rules and formats.

10.1. Client-side vs server-side validation

Form validation can occur on the client-side (in the user's browser) or on the server-side (on the web server).

- **Client-side Validation:**
  - **Where:** Performed in the user's web browser using HTML5 attributes (e.g., `required`, `type="email"`, `pattern`) or JavaScript.
  - **Pros:** Provides immediate feedback to the user, improves user experience, and reduces server load by catching simple errors early.
  - **Cons:** Can be bypassed by disabling JavaScript or manipulating browser requests. **Cannot be relied upon for security.**
- **Server-side Validation:**
  - **Where:** Performed on the web server using PHP (or other server-side languages).
  - **Pros:** Essential for security and data integrity. Cannot be bypassed by malicious users. Ensures data is valid before processing or storing.
  - **Cons:** Requires a round trip to the server, which can be slower and impact user experience if errors are frequent.

Best Practice: Always use both client-side and server-side validation. Client-side validation enhances user experience, while server-side validation provides the necessary security and data integrity.

10.2. Required fields

Ensuring that mandatory fields are filled out is a basic but crucial part of form validation. In PHP, you typically check if a field is empty using the `empty()` function or by checking its length after trimming whitespace.

Example:

PHP

```
<?php
    $name = $email = "";
    $nameErr = $emailErr = "";

    if ($_SERVER["REQUEST_METHOD"] == "POST") {
        if (empty($_POST["name"])) {
            $nameErr = "Name is required";
        } else {
            $name = htmlspecialchars(trim($_POST["name"]));
        }
```

```
        if (empty($_POST["email"])) {
            $emailErr = "Email is required";
        } else {
            $email = htmlspecialchars(trim($_POST["email"]));
        }

        if (empty($nameErr) && empty($emailErr)) {
            echo "<p>All required fields are filled!</p>";
            echo "Name: " . $name . "<br>";
            echo "Email: " . $email . "<br>";
        } else {
            echo "<p>Please correct the errors below.</p>";
        }
    }
?>

<!DOCTYPE html>
<html>
<head>
    <title>Required Fields Validation</title>
    <style>.error {color: red;}</style>
</head>
<body>
    <form method="post" action="<?php echo
htmlspecialchars($_SERVER["PHP_SELF"]);?>">
        Name: <input type="text" name="name" value="<?php echo $name;?>">
        <span class="error">* <?php echo $nameErr;?></span><br><br>

        Email: <input type="text" name="email" value="<?php echo $email;?>">
        <span class="error">* <?php echo $emailErr;?></span><br><br>

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

10.3. Data type validation

Data type validation ensures that the submitted data conforms to the expected type (e.g., integer, float, email, URL). PHP's `filter_var()` function is highly recommended for this purpose as it provides various filters for common data types.

Example:

PHP

```php
<?php
    $age = $email = $url = "";
    $ageErr = $emailErr = $urlErr = "";

    if ($_SERVER["REQUEST_METHOD"] == "POST") {
        // Validate Age (must be an integer)
        if (empty($_POST["age"])) {
            $ageErr = "Age is required";
        } else {
            $age = htmlspecialchars(trim($_POST["age"]));
            if (!filter_var($age, FILTER_VALIDATE_INT)) {
                $ageErr = "Invalid age format (must be an integer)";
            } else if ($age < 0 || $age > 120) {
                $ageErr = "Age must be between 0 and 120";
            }
        }

        // Validate Email
        if (empty($_POST["email"])) {
            $emailErr = "Email is required";
        } else {
            $email = htmlspecialchars(trim($_POST["email"]));
            if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
                $emailErr = "Invalid email format";
            }
        }

        // Validate URL
        if (!empty($_POST["url"])) {
            $url = htmlspecialchars(trim($_POST["url"]));
            if (!filter_var($url, FILTER_VALIDATE_URL)) {
                $urlErr = "Invalid URL format";
            }
        }

        if (empty($ageErr) && empty($emailErr) && empty($urlErr)) {
            echo "<p>All data types are valid!</p>";
            echo "Age: " . $age . "<br>";
            echo "Email: " . $email . "<br>";
            echo "URL: " . $url . "<br>";
        } else {
            echo "<p>Please correct the data type errors.</p>";
        }
    }
?>

<!DOCTYPE html>
```

```
<html>
<head>
    <title>Data Type Validation</title>
    <style>.error {color: red;}</style>
</head>
<body>
    <form method="post" action="<?php echo
htmlspecialchars($_SERVER["PHP_SELF"]);?>">
        Age: <input type="text" name="age" value="<?php echo $age;?>">
        <span class="error">* <?php echo $ageErr;?></span><br><br>

        Email: <input type="text" name="email" value="<?php echo $email;?>">
        <span class="error">* <?php echo $emailErr;?></span><br><br>

        Website: <input type="text" name="url" value="<?php echo $url;?>">
        <span class="error"><?php echo $urlErr;?></span><br><br>

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

10.4. Custom validation rules

While `filter_var()` covers many common validation scenarios, you often need to implement custom validation rules for specific business logic or complex data formats. Regular expressions (`preg_match()`) are invaluable for this.

Example: Custom Validation (e.g., password strength, specific ID format)

PHP

```
<?php
    $password = $studentId = "";
    $passwordErr = $studentIdErr = "";

    if ($_SERVER["REQUEST_METHOD"] == "POST") {
        // Custom Password Validation: At least 8 characters, one uppercase,
one lowercase, one number, one special character
        if (empty($_POST["password"])) {
            $passwordErr = "Password is required";
        } else {
            $password = htmlspecialchars(trim($_POST["password"]));
            $pattern = "/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[^\da-zA-Z]).
{8,}$/";
            if (!preg_match($pattern, $password)) {
                $passwordErr = "Password must be at least 8 characters long
```

```
and contain at least one uppercase letter, one lowercase letter, one number,
and one special character.";
            }
        }

        // Custom Student ID Validation: Format like 'S1234567' (S followed
by 7 digits)
        if (empty($_POST["student_id"])) {
            $studentIdErr = "Student ID is required";
        } else {
            $studentId = htmlspecialchars(trim($_POST["student_id"]));
            if (!preg_match("/^S\d{7}$/", $studentId)) {
                $studentIdErr = "Invalid Student ID format (e.g., S1234567)";
            }
        }

        if (empty($passwordErr) && empty($studentIdErr)) {
            echo "<p>Custom validation passed!</p>";
            echo "Password (hashed for display): " .
password_hash($password, PASSWORD_DEFAULT) . "<br>";
            echo "Student ID: " . $studentId . "<br>";
        } else {
            echo "<p>Please correct the custom validation errors.</p>";
        }
    }
?>

<!DOCTYPE html>
<html>
<head>
    <title>Custom Validation Rules</title>
    <style>.error {color: red;}</style>
</head>
<body>
    <form method="post" action="<?php echo
htmlspecialchars($_SERVER["PHP_SELF"]);?>">
        Password: <input type="password" name="password" value="<?php echo
$password;?>">
        <span class="error">* <?php echo $passwordErr;?></span><br><br>

        Student ID: <input type="text" name="student_id" value="<?php echo
$studentId;?>">
        <span class="error">* <?php echo $studentIdErr;?></span><br><br>

        <input type="submit" value="Submit">
    </form>
```

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

10.5. Error handling

Effective error handling in form validation involves providing clear, user-friendly feedback when validation fails. This typically means displaying error messages next to the problematic input fields and retaining valid user input so they don't have to re-enter everything.

Key principles:

- **Collect Errors:** Store all validation errors in an array or separate variables.
- **Display Errors:** Iterate through the errors and display them prominently, usually near the input field they relate to.
- **Retain Input:** Populate form fields with the data the user previously entered (if valid) so they only need to correct errors.

Example (integrated into previous examples):

The examples in Sections 10.2, 10.3, and 10.4 already demonstrate basic error handling by:

1. Initializing error variables (e.g., `$nameErr`, `$emailErr`) to empty strings.
2. Assigning error messages to these variables if validation fails.
3. Displaying the error messages next to the input fields using `<span>` tags with a distinct class (e.g., `error`).
4. Populating the `value` attribute of input fields with `<?php echo $name;?>` (or similar) to retain user input.

This approach ensures a good user experience by guiding them to correct their input efficiently.

11. PHP File Handling

PHP provides robust capabilities for interacting with the file system, allowing you to create, read, write, and manage files on the server. This is essential for tasks like logging, content management, and data storage.

11.1. Opening and closing files

Before you can perform any operations on a file, you must first open it using the `fopen()` function. After you're done with the file, it's crucial to close it using `fclose()` to free up system resources.

- **`fopen(filename, mode)` :** Opens a file or URL.
  - `filename` : The path to the file.
  - `mode` : Specifies the type of access you require to the stream.
    - `"r"` : Read only. Pointer at the beginning. (Default)
    - `"w"` : Write only. Creates new file or truncates existing. Pointer at the beginning.
    - `"a"` : Append only. Creates new file or appends to existing. Pointer at the end.
    - `"x"` : Create and write only. Returns `FALSE` if file already exists. Pointer at the beginning.
    - `"r+"` : Read/Write. Pointer at the beginning.
    - `"w+"` : Read/Write. Creates new file or truncates existing. Pointer at the beginning.
    - `"a+"` : Read/Write. Creates new file or appends to existing. Pointer at the end.
- **`fclose(file_handle)` :** Closes an open file pointer.

Example:

PHP

```
<?php
    $filename = "my_log.txt";

    // Open file for writing (creates if not exists, truncates if exists)
    $fileHandle = fopen($filename, "w");

    if ($fileHandle) {
        echo "File '" . $filename . "' opened successfully for writing.<br>";
        // Perform write operations here
        fwrite($fileHandle, "Log entry: " . date("Y-m-d H:i:s") . " -
Application started.\n");
        fclose($fileHandle);
        echo "File closed.<br>";
    } else {
        echo "Error: Could not open file '" . $filename . "' for writing.
<br>";
    }

    // Open file for reading
    $fileHandleRead = fopen($filename, "r");
    if ($fileHandleRead) {
        echo "File '" . $filename . "' opened successfully for reading.<br>";
        // Perform read operations here
        $content = fread($fileHandleRead, filesize($filename));
```

```
        echo "Content: " . htmlspecialchars($content) . "<br>";
        fclose($fileHandleRead);
        echo "File closed.<br>";
    } else {
        echo "Error: Could not open file '" . $filename . "' for reading.
<br>";
    }
?>
```

11.2. Reading from files

PHP offers several functions to read content from files, depending on whether you want to read the entire file, line by line, or character by character.

- **`fread(file_handle, length)`:** Reads up to `length` bytes from the file pointer.
- **`fgets(file_handle)`:** Reads a single line from the file pointer.
- **`file_get_contents(filename)`:** Reads entire file into a string. This is often the simplest way to read a whole file.
- **`file(filename)`:** Reads entire file into an array, with each element representing a line.

Example:

PHP

```
<?php
    $filename = "data.txt";

    // Create a dummy file for reading
    file_put_contents($filename, "Line 1\nLine 2\nLine 3\n");

    echo "<p>Reading entire file with file_get_contents():</p>";
    $content = file_get_contents($filename);
    echo nl2br(htmlspecialchars($content)) . "<br>";

    echo "<p>Reading file line by line with fgets():</p>";
    $fileHandle = fopen($filename, "r");
    if ($fileHandle) {
        while (!feof($fileHandle)) { // Loop until end of file
            echo htmlspecialchars(fgets($fileHandle)) . "<br>";
        }
        fclose($fileHandle);
    }

    echo "<p>Reading file into an array with file():</p>";
    $lines = file($filename);
    foreach ($lines as $lineNumber => $lineContent) {
```

```
        echo "Line " . ($lineNumber + 1) . ": " .
htmlspecialchars($lineContent) . "<br>";
    }
?>
```

11.3. Writing to files

To write data to a file, you typically use `fwrite()` after opening the file in a write (`"w"`), append (`"a"`), or read/write (`"w+"` , `"a+"`) mode. For simple writes, `file_put_contents()` is a convenient shortcut.

- **`fwrite(file_handle, string)` :** Writes `string` to the file pointer.
- **`file_put_contents(filename, data, flags)` :** Writes `data` to `filename` . If `filename` does not exist, it is created. If it exists, it is overwritten unless `FILE_APPEND` flag is used.

Example:

PHP

```
<?php
    $logFile = "app_log.txt";

    // Write (overwrite) content to file
    file_put_contents($logFile, "First log entry.\n");
    echo "'" . $logFile . "' created/overwritten with 'First log entry.'.
<br>";

    // Append content to file
    file_put_contents($logFile, "Second log entry (appended).\n",
FILE_APPEND);
    echo "'Second log entry' appended to '" . $logFile . "'.<br>";

    // Using fopen and fwrite for more control
    $fileHandle = fopen("notes.txt", "w");
    if ($fileHandle) {
        fwrite($fileHandle, "My important note.\n");
        fwrite($fileHandle, "Another line of notes.\n");
        fclose($fileHandle);
        echo "'notes.txt' created and written to.<br>";
    } else {
        echo "Error: Could not open 'notes.txt' for writing.<br>";
    }
?>
```

11.4. File permissions

File permissions determine who can read, write, or execute files and directories on the server. Correct permissions are crucial for security and proper application functioning. Permissions are typically represented by a three-digit octal number (e.g., 755, 644).

- **`chmod(filename, mode)` :** Changes file mode (permissions).
  - `mode` : An octal number representing permissions.
    - First digit: Owner permissions
    - Second digit: Group permissions
    - Third digit: Others (public) permissions
  - **Common Permissions:**
    - `7 (rwx)` : Read, Write, Execute
    - `6 (rw-)` : Read, Write
    - `5 (r-x)` : Read, Execute
    - `4 (r--)` : Read only
  - **Example Combinations:**
    - `0755` : Owner can read, write, execute; Group and Others can read and execute (common for directories).
    - `0644` : Owner can read, write; Group and Others can read (common for files).
    - `0600` : Owner can read, write; No access for Group and Others (very restrictive).

Example:

PHP

```
<?php
    $fileToProtect = "secret_data.txt";
    file_put_contents($fileToProtect, "This is sensitive information.");

    echo "Original permissions of '" . $fileToProtect . "': " .
substr(sprintf('%o', fileperms($fileToProtect)), -4) . "<br>";

    // Set permissions to 0600 (owner read/write, no one else access)
    if (chmod($fileToProtect, 0600)) {
        echo "Permissions changed to 0600 for '" . $fileToProtect . "'.<br>";
    } else {
        echo "Failed to change permissions for '" . $fileToProtect . "'.
<br>";
    }

    echo "New permissions: " . substr(sprintf('%o',
fileperms($fileToProtect)), -4) . "<br>";
```

```
    // Attempt to read by others (will fail if web server user is not owner)
    // This part depends on your server's user configuration
    // $content = file_get_contents($fileToProtect);
    // echo "Content (if readable): " . htmlspecialchars($content) . "<br>";
?>
```

11.5. File uploads

PHP makes it relatively easy to handle file uploads from HTML forms. However, it's crucial to implement robust security checks to prevent malicious file uploads.

HTML Form for File Upload (`upload_form.html` or `upload_form.php`):

HTML

```
<!DOCTYPE html>
<html>
<head>
    <title>File Upload</title>
</head>
<body>

    <h2>Upload an Image</h2>
    <form action="upload.php" method="post" enctype="multipart/form-data">
        Select image to upload:
        <input type="file" name="fileToUpload" id="fileToUpload"><br><br>
        <input type="submit" value="Upload Image" name="submit">
    </form>

</body>
</html>
```

PHP Script for Handling Upload (`upload.php`):

PHP

```
<?php
    $target_dir = "uploads/"; // Directory where uploaded files will be
stored
    $target_file = $target_dir . basename($_FILES["fileToUpload"]["name"]);
    $uploadOk = 1;
    $imageFileType = strtolower(pathinfo($target_file,PATHINFO_EXTENSION));

    // Check if image file is a actual image or fake image
    if(isset($_POST["submit"])) {
        $check = getimagesize($_FILES["fileToUpload"]["tmp_name"]);
```

```
        if($check !== false) {
            echo "File is an image - " . $check["mime"] . ".<br>";
            $uploadOk = 1;
        } else {
            echo "File is not an image.<br>";
            $uploadOk = 0;
        }
    }

    // Check if file already exists
    if (file_exists($target_file)) {
        echo "Sorry, file already exists.<br>";
        $uploadOk = 0;
    }

    // Check file size (e.g., max 500KB)
    if ($_FILES["fileToUpload"]["size"] > 500000) {
        echo "Sorry, your file is too large.<br>";
        $uploadOk = 0;
    }

    // Allow certain file formats
    if($imageFileType != "jpg" && $imageFileType != "png" && $imageFileType
!= "jpeg"
    && $imageFileType != "gif" ) {
        echo "Sorry, only JPG, JPEG, PNG & GIF files are allowed.<br>";
        $uploadOk = 0;
    }

    // Check if $uploadOk is set to 0 by an error
    if ($uploadOk == 0) {
        echo "Sorry, your file was not uploaded.<br>";
    // if everything is ok, try to upload file
    } else {
        if (move_uploaded_file($_FILES["fileToUpload"]["tmp_name"],
$target_file)) {
            echo "The file ". htmlspecialchars( basename(
$_FILES["fileToUpload"]["name"])) . " has been uploaded.<br>";
        } else {
            echo "Sorry, there was an error uploading your file.<br>";
        }
    }
?>
```

Explanation:

- `enctype="multipart/form-data"` in the HTML form is essential for file uploads.

- `$_FILES` superglobal array holds information about the uploaded file (name, type, size, temporary name, error code).
- `move_uploaded_file()` moves the uploaded file from its temporary location to your specified destination. This is the only safe way to handle uploaded files.
- **Security:** The example includes basic checks for file type, size, and existence. In a real-world application, you would need more rigorous validation (e.g., checking MIME types, scanning for malware, storing files outside the web root, renaming files to prevent execution).

12. PHP Cookies and Sessions

Cookies and sessions are mechanisms used to store data about a user across multiple page requests. They are crucial for maintaining user state, implementing login systems, and personalizing user experiences.

12.1. Setting and retrieving cookies

Cookies are small pieces of data stored on the client's (user's) browser. They are sent with every request to the server and can be used to remember user preferences, login status, or tracking information.

- **`setcookie(name, value, expire, path, domain, secure, httponly)`:** Sends a cookie to the user's web browser.
  - `name`: The name of the cookie.
  - `value`: The value of the cookie.
  - `expire`: The time the cookie expires (Unix timestamp). `time() + seconds`.
  - `path`: The path on the server where the cookie will be available.
  - `domain`: The domain that the cookie is available to.
  - `secure`: `TRUE` if the cookie should only be sent over secure HTTPS connections.
  - `httponly`: `TRUE` if the cookie should only be accessible through the HTTP protocol (prevents JavaScript access, mitigating XSS).
- **`$_COOKIE` superglobal:** An associative array containing all cookies sent by the browser to the server.

Example:

PHP

```
<?php
    // Set a cookie named "username" that expires in 1 hour
```

```
    $cookie_name = "username";
    $cookie_value = "JohnDoe";
    setcookie($cookie_name, $cookie_value, time() + (3600), "/"); // 3600 =
1 hour

    // Set a cookie with secure and httponly flags (recommended for
sensitive cookies)
    setcookie("session_id", "abc123def456", time() + (86400 * 30), "/",
".example.com", true, true);

    echo "<p>Cookies set.</p>";

    // Retrieve cookies
    if(isset($_COOKIE[$cookie_name])) {
        echo "Cookie '" . $cookie_name . "' is set!<br>";
        echo "Value is: " . $_COOKIE[$cookie_name] . "<br>";
    } else {
        echo "Cookie '" . $cookie_name . "' is not set.<br>";
    }

    // Check for other cookies
    if(isset($_COOKIE["session_id"])) {
        echo "Session ID cookie is set.<br>";
    }

    // To delete a cookie, set its expiration time to a past value
    // setcookie("username", "", time() - 3600);
    // echo "Cookie 'username' is deleted.<br>";
?>
```

Important Note: `setcookie()` must be called before any actual output is sent to the browser (e.g., before any HTML, `echo`, or `print` statements). Otherwise, you will get a "Headers already sent" error.

12.2. Session management

Sessions provide a way to store information (in variables) to be used across multiple pages. Unlike cookies, session data is stored on the server, and only a unique session ID (usually stored in a cookie) is sent to the client. This makes sessions more secure for sensitive data.

Key Session Functions:

- **`session_start()`:** Starts a new session or resumes an existing one. This function must be called at the very beginning of every PHP page that uses sessions.
- **`$_SESSION` superglobal:** An associative array used to store and access session variables.

- **`session_unset()` :** Frees all session variables.
- **`session_destroy()` :** Destroys all data registered to a session.

Example: Login System using Sessions (`login.php`)

PHP

```
<?php
    session_start(); // Start the session

    $message = "";

    if ($_SERVER["REQUEST_METHOD"] == "POST") {
        $username = $_POST["username"] ?? "";
        $password = $_POST["password"] ?? "";

        // Simple hardcoded check for demonstration. In real app, use
database.
        if ($username === "admin" && $password === "password123") {
            $_SESSION["loggedin"] = true;
            $_SESSION["username"] = $username;
            $_SESSION["start_time"] = time();
            $_SESSION["expire_time"] = time() + (30 * 60); // Session
expires in 30 minutes

            header("Location: dashboard.php"); // Redirect to dashboard
            exit();
        } else {
            $message = "Invalid username or password.";
        }
    }
?>

<!DOCTYPE html>
<html>
<head>
    <title>Login</title>
</head>
<body>
    <h2>Login</h2>
    <form method="post" action="<?php echo
htmlspecialchars($_SERVER["PHP_SELF"]);?>">
        Username: <input type="text" name="username"><br>
        Password: <input type="password" name="password"><br>
        <input type="submit" value="Login">
    </form>
    <p style="color:red;"><?php echo $message;?></p>
```

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

Example: Dashboard Page (`dashboard.php`)

PHP

```
<?php
    session_start();

    // Check if user is logged in
    if (!isset($_SESSION["loggedin"]) || $_SESSION["loggedin"] !== true) {
        header("Location: login.php");
        exit();
    }

    // Check for session timeout
    if (isset($_SESSION["expire_time"]) && time() >
$_SESSION["expire_time"]) {
        session_unset();
        session_destroy();
        header("Location: login.php?timeout=true");
        exit();
    }

    $username = $_SESSION["username"];
?>

<!DOCTYPE html>
<html>
<head>
    <title>Dashboard</title>
</head>
<body>
    <h2>Welcome, <?php echo htmlspecialchars($username); ?>!</h2>
    <p>This is your dashboard. You are logged in.</p>
    <p><a href="logout.php">Logout</a></p>
</body>
</html>
```

Example: Logout Page (`logout.php`)

PHP

```
<?php
    session_start();
```

```
    // Unset all session variables
    session_unset();

    // Destroy the session
    session_destroy();

    header("Location: login.php?loggedout=true");
    exit();
?>
```

12.3. Session security

While sessions are generally more secure than cookies for storing sensitive data, they are not immune to attacks. Proper session management and security practices are essential to protect user data and prevent unauthorized access.

Common Session Attacks and Mitigations:

- **Session Hijacking:** An attacker steals a valid session ID and uses it to impersonate the legitimate user. **Mitigation:**
  - **Regenerate Session ID:** Use `session_regenerate_id(true)` after successful login and any privilege escalation. This generates a new session ID and deletes the old one, preventing session fixation.
  - **Use `HttpOnly` and `Secure` flags for session cookies:**
    - `HttpOnly`: Prevents client-side scripts (JavaScript) from accessing the session cookie, mitigating XSS attacks that try to steal session IDs.
    - `Secure`: Ensures the session cookie is only sent over HTTPS connections, protecting it from eavesdropping.
    - These can be set in `php.ini` (`session.cookie_httponly = 1`, `session.cookie_secure = 1`) or via `session_set_cookie_params()`.
  - **Session Timeout:** Implement a reasonable session timeout to automatically log out inactive users. This limits the window of opportunity for attackers.
  - **Check User Agent/IP Address:** While not foolproof, checking if the user agent or IP address changes during a session can indicate a hijacking attempt. However, this can also lead to false positives (e.g., mobile users switching networks).
- **Session Fixation:** An attacker tricks a user into using a session ID known to the attacker. When the user logs in with this fixed ID, the attacker gains access to their authenticated session. **Mitigation:** Always regenerate the session ID upon login (`session_regenerate_id(true)`).

Example: Enhanced Session Security

PHP

```
<?php
    // Set session cookie parameters for better security
    session_set_cookie_params([
        'lifetime' => 0, // Session cookie lasts until browser closes
        'path' => '/',
        'domain' => $_SERVER["HTTP_HOST"], // Restrict to current domain
        'secure' => true, // Only send over HTTPS
        'httponly' => true, // Prevent JavaScript access
        'samesite' => 'Lax' // Protect against CSRF (PHP 7.3+)
    ]);

    session_start();

    // Regenerate session ID on login (or any privilege change)
    if (!isset($_SESSION["loggedin"]) && isset($_POST["username"])) {
        // This is where you'd typically verify credentials
        // For demonstration, assume login is successful
        if ($_POST["username"] === "secure_user" && $_POST["password"] ===
"secure_pass") {
            session_regenerate_id(true); // Crucial for session fixation
prevention
            $_SESSION["loggedin"] = true;
            $_SESSION["username"] = "secure_user";
            $_SESSION["last_activity"] = time(); // Track last activity for
timeout
            echo "Login successful. Session ID regenerated.<br>";
        } else {
            echo "Invalid credentials.<br>";
        }
    }

    // Implement session timeout (e.g., 30 minutes of inactivity)
    $session_timeout = 30 * 60; // 30 minutes
    if (isset($_SESSION["last_activity"]) && (time() -
$_SESSION["last_activity"] > $session_timeout)) {
        session_unset();
        session_destroy();
        echo "Your session has expired due to inactivity. Please log in
again.<br>";
        // Redirect to login page
        // header("Location: login.php?timeout=true");
        // exit();
    }
    $_SESSION["last_activity"] = time(); // Update last activity time on
each request
```

```
    if (isset($_SESSION["loggedin"]) && $_SESSION["loggedin"] === true) {
        echo "Welcome, " . htmlspecialchars($_SESSION["username"]) . "! You
are logged in.<br>";
    } else {
        echo "Please log in.<br>";
    }

    // Example of logout
    if (isset($_GET["logout"])) {
        session_unset();
        session_destroy();
        echo "You have been logged out.<br>";
    }
?>

<form method="post">
    Username: <input type="text" name="username" value="secure_user"><br>
    Password: <input type="password" name="password" value="secure_pass"><br>
    <input type="submit" value="Login">
</form>
<a href="?logout=true">Logout</a>
```

13. PHP and MySQL Interaction

Connecting PHP to a MySQL database is fundamental for building dynamic web applications that store and retrieve data. PHP provides extensions like MySQLi and PDO for database interaction.

13.1. Introduction to MySQL

MySQL is an open-source relational database management system (RDBMS) that uses Structured Query Language (SQL) to manage and manipulate data. It is a popular choice for web applications due to its speed, reliability, and ease of use, often paired with PHP (forming part of the LAMP/WAMP/MAMP stack).

Key Concepts in Relational Databases:

- **Database:** A structured collection of data.
- **Table:** A collection of related data organized in rows and columns.
- **Row (Record/Tuple):** A single entry in a table, representing a set of related data.
- **Column (Field/Attribute):** A specific category of data within a table (e.g., `name`, `email`).
- **Primary Key:** A column (or set of columns) that uniquely identifies each row in a table.

- **Foreign Key:** A column (or set of columns) in one table that refers to the primary key in another table, establishing a relationship between them.
- **SQL (Structured Query Language):** The standard language for managing and manipulating relational databases (e.g., `SELECT`, `INSERT`, `UPDATE`, `DELETE`).

13.2. Connecting to MySQL (MySQLi and PDO)

PHP offers two main extensions for connecting to and interacting with MySQL databases:

- **MySQLi (MySQL Improved Extension):** Specifically designed for MySQL databases. It offers both a procedural and an object-oriented interface.
- **PDO (PHP Data Objects):** A database abstraction layer that provides a consistent interface for connecting to various database types (MySQL, PostgreSQL, SQLite, etc.). It is generally recommended for new projects due to its flexibility and better support for prepared statements.

13.2.1. MySQLi (Object-Oriented)

Example: Connecting to MySQL using MySQLi (Object-Oriented)

PHP

```
<?php
    $servername = "localhost";
    $username = "root";
    $password = ""; // Your MySQL password
    $dbname = "mydatabase"; // The database you want to connect to

    // Create connection
    $conn = new mysqli($servername, $username, $password, $dbname);

    // Check connection
    if ($conn->connect_error) {
        die("Connection failed: " . $conn->connect_error);
    }
    echo "Connected successfully using MySQLi (Object-Oriented)<br>";

    // Close connection
    $conn->close();
?>
```

13.2.2. MySQLi (Procedural)

Example: Connecting to MySQL using MySQLi (Procedural)

PHP

```php
<?php
    $servername = "localhost";
    $username = "root";
    $password = "";
    $dbname = "mydatabase";

    // Create connection
    $conn = mysqli_connect($servername, $username, $password, $dbname);

    // Check connection
    if (!$conn) {
        die("Connection failed: " . mysqli_connect_error());
    }
    echo "Connected successfully using MySQLi (Procedural)<br>";

    // Close connection
    mysqli_close($conn);
?>
```

13.2.3. PDO (PHP Data Objects)

Example: Connecting to MySQL using PDO

PHP

```php
<?php
    $servername = "localhost";
    $username = "root";
    $password = "";
    $dbname = "mydatabase";

    try {
        $conn = new PDO("mysql:host=$servername;dbname=$dbname", $username,
$password);
        // Set the PDO error mode to exception
        $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
        echo "Connected successfully using PDO<br>";
    } catch(PDOException $e) {
        die("Connection failed: " . $e->getMessage());
    }

    // Close connection (PDO connection is closed when the script ends or
when the object is unset)
    $conn = null;
?>
```

13.3. Creating a database and tables

Before you can store data, you need to create a database and then tables within that database. This is typically done once during the setup phase of your application.

Example: Creating a Database and Table (`create_db_table.php`)

PHP

```
<?php
    $servername = "localhost";
    $username = "root";
    $password = "";
    $dbname = "my_new_database"; // Name of the new database

    // Create connection
    $conn = new mysqli($servername, $username, $password);

    // Check connection
    if ($conn->connect_error) {
        die("Connection failed: " . $conn->connect_error);
    }

    // Create database
    $sql_create_db = "CREATE DATABASE IF NOT EXISTS $dbname";
    if ($conn->query($sql_create_db) === TRUE) {
        echo "Database ".$dbname." created successfully<br>";
    } else {
        echo "Error creating database: " . $conn->error . "<br>";
    }

    // Select the database
    $conn->select_db($dbname);

    // SQL to create table
    $sql_create_table = "CREATE TABLE IF NOT EXISTS Users (
        id INT(6) UNSIGNED AUTO_INCREMENT PRIMARY KEY,
        firstname VARCHAR(30) NOT NULL,
        lastname VARCHAR(30) NOT NULL,
        email VARCHAR(50),
        reg_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP
    )";

    if ($conn->query($sql_create_table) === TRUE) {
        echo "Table Users created successfully<br>";
    } else {
        echo "Error creating table: " . $conn->error . "<br>";
```

```
    }

    $conn->close();
?>
```

13.4. Inserting data

Inserting data into a MySQL table is done using the `INSERT INTO` SQL statement. It's crucial to use prepared statements to prevent SQL injection vulnerabilities.

13.4.1. MySQLi Prepared Statements (Object-Oriented)

Example: Inserting Data using MySQLi Prepared Statements

PHP

```
<?php
    $servername = "localhost";
    $username = "root";
    $password = "";
    $dbname = "my_new_database"; // Use the database created above

    // Create connection
    $conn = new mysqli($servername, $username, $password, $dbname);

    // Check connection
    if ($conn->connect_error) {
        die("Connection failed: " . $conn->connect_error);
    }

    // Prepare and bind
    $stmt = $conn->prepare("INSERT INTO Users (firstname, lastname, email)
VALUES (?, ?, ?)");
    $stmt->bind_param("sss", $firstname, $lastname, $email);

    // Set parameters and execute
    $firstname = "John";
    $lastname = "Doe";
    $email = "john@example.com";
    $stmt->execute();

    $firstname = "Mary";
    $lastname = "Moe";
    $email = "mary@example.com";
    $stmt->execute();

    echo "New records created successfully<br>";
```

```
    $stmt->close();
    $conn->close();
?>
```

13.4.2. PDO Prepared Statements

Example: Inserting Data using PDO Prepared Statements

PHP

```
<?php
    $servername = "localhost";
    $username = "root";
    $password = "";
    $dbname = "my_new_database";

    try {
        $conn = new PDO("mysql:host=$servername;dbname=$dbname", $username,
$password);
        $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

        // Prepare SQL and bind parameters
        $stmt = $conn->prepare("INSERT INTO Users (firstname, lastname,
email) VALUES (:firstname, :lastname, :email)");
        $stmt->bindParam(":firstname", $firstname);
        $stmt->bindParam(":lastname", $lastname);
        $stmt->bindParam(":email", $email);

        // Insert a row
        $firstname = "Jane";
        $lastname = "Doe";
        $email = "jane@example.com";
        $stmt->execute();

        // Insert another row
        $firstname = "Mike";
        $lastname = "Ross";
        $email = "mike@example.com";
        $stmt->execute();

        echo "New records created successfully<br>";
    } catch(PDOException $e) {
        echo "Error: " . $e->getMessage();
    }
```

```
    $conn = null;
?>
```

13.5. Retrieving data

Retrieving data from a MySQL table is done using the `SELECT` SQL statement. You can fetch results row by row or all at once.

13.5.1. MySQLi (Object-Oriented)

Example: Retrieving Data using MySQLi (Object-Oriented)

PHP

```
<?php
    $servername = "localhost";
    $username = "root";
    $password = "";
    $dbname = "my_new_database";

    // Create connection
    $conn = new mysqli($servername, $username, $password, $dbname);

    // Check connection
    if ($conn->connect_error) {
        die("Connection failed: " . $conn->connect_error);
    }

    $sql = "SELECT id, firstname, lastname, email FROM Users";
    $result = $conn->query($sql);

    if ($result->num_rows > 0) {
        // Output data of each row
        while($row = $result->fetch_assoc()) {
            echo "id: " . $row["id"]. " - Name: " . $row["firstname"]. " " .
$row["lastname"]. " - Email: " . $row["email"]. "<br>";
        }
    } else {
        echo "0 results";
    }

    $conn->close();
?>
```

13.5.2. PDO

Example: Retrieving Data using PDO

PHP

```
<?php
    $servername = "localhost";
    $username = "root";
    $password = "";
    $dbname = "my_new_database";

    try {
        $conn = new PDO("mysql:host=$servername;dbname=$dbname", $username,
$password);
        $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

        $stmt = $conn->prepare("SELECT id, firstname, lastname, email FROM
Users");
        $stmt->execute();

        // Set the resulting array to associative
        $result = $stmt->setFetchMode(PDO::FETCH_ASSOC);

        foreach($stmt->fetchAll() as $k=>$v) {
            echo "id: " . $v["id"]. " - Name: " . $v["firstname"]. " " .
$v["lastname"]. " - Email: " . $v["email"]. "<br>";
        }
    } catch(PDOException $e) {
        echo "Error: " . $e->getMessage();
    }
    $conn = null;
?>
```

13.6. Updating data

Updating existing data in a MySQL table is done using the `UPDATE` SQL statement. Again, prepared statements are essential for security.

13.6.1. MySQLi Prepared Statements (Object-Oriented)

Example: Updating Data using MySQLi Prepared Statements

PHP

```
<?php
    $servername = "localhost";
    $username = "root";
    $password = "";
```

```
    $dbname = "my_new_database";

    // Create connection
    $conn = new mysqli($servername, $username, $password, $dbname);

    // Check connection
    if ($conn->connect_error) {
        die("Connection failed: " . $conn->connect_error);
    }

    // Prepare statement
    $stmt = $conn->prepare("UPDATE Users SET email = ? WHERE id = ?");
    $stmt->bind_param("si", $email, $id);

    // Set parameters and execute
    $email = "john_new@example.com";
    $id = 1;
    $stmt->execute();

    echo "Record updated successfully<br>";

    $stmt->close();
    $conn->close();
?>
```

13.6.2. PDO Prepared Statements

Example: Updating Data using PDO Prepared Statements

PHP

```
<?php
    $servername = "localhost";
    $username = "root";
    $password = "";
    $dbname = "my_new_database";

    try {
        $conn = new PDO("mysql:host=$servername;dbname=$dbname", $username,
$password);
        $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

        // Prepare statement
        $stmt = $conn->prepare("UPDATE Users SET email = :email WHERE id =
:id");
        $stmt->bindParam(":email", $email);
        $stmt->bindParam(":id", $id);
```

```
        // Set parameters and execute
        $email = "jane_new@example.com";
        $id = 2;
        $stmt->execute();

        echo "Record updated successfully<br>";
    } catch(PDOException $e) {
        echo "Error: " . $e->getMessage();
    }
    $conn = null;
?>
```

13.7. Deleting data

Deleting data from a MySQL table is done using the `DELETE FROM` SQL statement. Always use prepared statements with `WHERE` clauses to avoid accidentally deleting all records.

13.7.1. MySQLi Prepared Statements (Object-Oriented)

Example: Deleting Data using MySQLi Prepared Statements

PHP

```
<?php
    $servername = "localhost";
    $username = "root";
    $password = "";
    $dbname = "my_new_database";

    // Create connection
    $conn = new mysqli($servername, $username, $password, $dbname);

    // Check connection
    if ($conn->connect_error) {
        die("Connection failed: " . $conn->connect_error);
    }

    // Prepare statement
    $stmt = $conn->prepare("DELETE FROM Users WHERE id = ?");
    $stmt->bind_param("i", $id);

    // Set parameters and execute
    $id = 3;
    $stmt->execute();

    echo "Record deleted successfully<br>";
```

```
    $stmt->close();
    $conn->close();
?>
```

13.7.2. PDO Prepared Statements

Example: Deleting Data using PDO Prepared Statements

PHP

```
<?php
    $servername = "localhost";
    $username = "root";
    $password = "";
    $dbname = "my_new_database";

    try {
        $conn = new PDO("mysql:host=$servername;dbname=$dbname", $username,
$password);
        $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

        // Prepare statement
        $stmt = $conn->prepare("DELETE FROM Users WHERE id = :id");
        $stmt->bindParam(":id", $id);

        // Set parameters and execute
        $id = 4;
        $stmt->execute();

        echo "Record deleted successfully<br>";
    } catch(PDOException $e) {
        echo "Error: " . $e->getMessage();
    }
    $conn = null;
?>
```

13.8. Prepared statements and SQL injection prevention

Prepared statements are a crucial feature for securing database interactions, especially against SQL injection attacks. They work by separating the SQL query structure from the actual data. This ensures that user-supplied data is treated as data, not as executable SQL code.

How Prepared Statements Work:

1. **Prepare:** The SQL query template is sent to the database server. Placeholders (e.g., `?` for MySQLi, `:name` for PDO) are used for values that will be supplied later.
2. **Bind:** The actual values are bound to the placeholders. The database driver automatically escapes these values, preventing them from being interpreted as SQL commands.
3. **Execute:** The prepared statement is executed with the bound values.

Benefits:

- **Security:** Prevents SQL injection by ensuring data is always treated as data.
- **Performance:** The database server can parse and optimize the query template once, and then reuse it multiple times with different data, leading to better performance for repetitive queries.

Example (reiterating from previous sections for emphasis):

PHP

```
<?php
    $servername = "localhost";
    $username = "root";
    $password = "";
    $dbname = "my_new_database";

    try {
        $conn = new PDO("mysql:host=$servername;dbname=$dbname", $username,
$password);
        $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

        // SQL query with named placeholders
        $stmt = $conn->prepare("INSERT INTO Users (firstname, lastname,
email) VALUES (:fname, :lname, :email)");

        // Bind parameters
        $stmt->bindParam(":fname", $fname);
        $stmt->bindParam(":lname", $lname);
        $stmt->bindParam(":email", $email);

        // Execute with different data
        $fname = "Alice";
        $lname = "Wonder";
        $email = "alice@example.com";
        $stmt->execute();

        $fname = "Bob";
        $lname = "Builder";
```

```
        $email = "bob@example.com";
        $stmt->execute();

        echo "New records inserted securely using PDO prepared statements.
<br>";

    } catch(PDOException $e) {
        echo "Error: " . $e->getMessage();
    }
    $conn = null;
?>
```

14. Error Handling and Debugging

Effective error handling and debugging are crucial skills for any PHP developer. They help in identifying, understanding, and resolving issues in your code.

14.1. PHP error reporting levels

PHP has several error reporting levels that control which types of errors are displayed. These can be configured in `php.ini` or dynamically within your script using `error_reporting()` and `ini_set()`.

Common Error Reporting Constants:

- `E_ERROR`: Fatal run-time errors. Execution of the script is halted.
- `E_WARNING`: Run-time warnings. Non-fatal errors. Script execution is not halted.
- `E_PARSE`: Compile-time parse errors. Generated by the parser.
- `E_NOTICE`: Run-time notices. Indicate something that might be an error, but could also happen in normal course of running a script (e.g., accessing an undefined variable).
- `E_DEPRECATED`: Run-time notices. Enable this to be warned about code that will not work in future versions of PHP.
- `E_ALL`: All errors and warnings, except `E_STRICT` in PHP < 5.4, and `E_DEPRECATED` in PHP < 5.3.

Recommended Settings:

- **Development Environment:** Set `error_reporting(E_ALL)` and `ini_set('display_errors', 1)` to see all errors and notices immediately. This helps in catching bugs early.
- **Production Environment:** Set `error_reporting(E_ALL & ~E_NOTICE & ~E_DEPRECATED)` and `ini_set('display_errors', 0)`. Instead, log errors to a file (`ini_set('log_errors', 1)`, `ini_set('error_log', '/path/to/php_errors.log')`). Never display errors directly to users in production, as it can reveal sensitive information.

Example:

```
PHP

<?php
    // In development: show all errors
    ini_set('display_errors', 1);
    ini_set('display_startup_errors', 1);
    error_reporting(E_ALL);

    echo "<p>Error reporting set for development.</p>";

    // Example of a Notice (accessing undefined variable)
    echo $undefined_variable; // This will generate an E_NOTICE

    // Example of a Warning (file not found)
    $file = fopen("non_existent_file.txt", "r"); // This will generate an
E_WARNING

    // Example of a Fatal Error (calling undefined function)
    // undefined_function(); // Uncommenting this would halt script
execution with E_ERROR

    echo "<p>Script continues after notice and warning.</p>";

    // In production: hide errors from display, log to file
    // ini_set('display_errors', 0);
    // ini_set('log_errors', 1);
    // ini_set('error_log', '/var/log/php/php_errors.log'); // Make sure
this directory is writable by web server
    // error_reporting(E_ALL & ~E_NOTICE & ~E_DEPRECATED);
?>
```

14.2. Custom error handlers

You can define your own custom error handler function using `set_error_handler()`. This allows you to gracefully handle errors, log them, or display custom messages instead of PHP's default error output.

Example:

```
PHP

<?php
    // Custom error handler function
    function myErrorHandler($errno, $errstr, $errfile, $errline) {
        // $errno: The level of the error raised
```

```
        // $errstr: The error message
        // $errfile: The filename that the error was raised in
        // $errline: The line number the error was raised at

        echo "<div style=\"border: 1px solid red; padding: 10px; margin:
10px 0;\">";
        echo "<b>Custom Error:</b> [{$errno}] {$errstr}<br>";
        echo "Error on line {$errline} in {$errfile}<br>";
        echo "</div>";

        // Don't execute PHP's internal error handler
        return true;
    }

    // Set custom error handler
    set_error_handler("myErrorHandler");

    // Trigger some errors
    echo $nonExistentVar; // E_NOTICE

    $result = 10 / 0; // E_WARNING (Division by zero)

    // Restore default error handler (optional)
    // restore_error_handler();

    echo "<p>Script continues after custom error handling.</p>";
?>
```

14.3. Logging errors

Logging errors to a file is essential for production environments, as it allows you to track and debug issues without exposing sensitive information to users. You can configure error logging in `php.ini` or use `error_log()`.

`php.ini` settings for logging:

Plain Text

```
display_errors = Off
log_errors = On
error_log = /var/log/php/php_errors.log ; Ensure this path is writable by
the web server
```

Example: Logging errors with `error_log()`

PHP

```
<?php
    // Ensure display_errors is Off in production
    ini_set('display_errors', 0);
    ini_set('log_errors', 1);
    ini_set('error_log', __DIR__ . '/application_errors.log'); // Log to a
file in the current directory

    echo "<p>Error logging configured. Check application_errors.log for
messages.</p>";

    // Log a custom error message
    error_log("User 'admin' attempted to access restricted area.");

    // Trigger a warning that will be logged
    $file = fopen("another_non_existent_file.txt", "r");

    echo "<p>Script finished.</p>";
?>
```

14.4. Debugging techniques (var_dump, print_r, Xdebug)

Debugging is the process of finding and fixing errors in your code. PHP offers several tools and techniques for debugging.

- `var_dump()` **:** Displays structured information about one or more expressions, including its type and value. Arrays and objects are explored recursively with values indented to show structure.
- `print_r()` **:** Prints human-readable information about a variable. If given an array or object, it will print a structured representation. Less detailed than `var_dump()` .
- **Xdebug:** A powerful PHP extension that provides debugging and profiling capabilities. It allows you to set breakpoints, step through code, inspect variables, and analyze performance. Xdebug requires configuration in `php.ini` and an IDE (like VS Code with PHP Debug extension) to connect to it.
    - **Step Debugging:** Execute code line by line.
    - **Breakpoints:** Pause execution at specific lines.
    - **Variable Inspection:** View the values of variables at any point.
    - **Stack Traces:** See the call stack leading to the current execution point.
    - **Profiling:** Analyze script performance.

zend_extension=xdebug.so
xdebug.mode=debug
xdebug.start_with_request=yes

xdebug.client_host=127.0.0.1
xdebug.client_port=9003
```

Plain Text

```
After configuring, restart your web server (e.g., `sudo service apache2
restart` or `sudo service nginx restart`). Then, configure your IDE to
listen for Xdebug connections.
```

15. PHP Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data and code. PHP has robust support for OOP, allowing for more modular, reusable, and maintainable code.

15.1. Classes and objects

- **Class:** A blueprint or a template for creating objects. It defines properties (variables) and methods (functions) that objects of that class will have.
- **Object:** An instance of a class. When a class is defined, no memory is allocated until an object is created from it.

Example:

PHP

```
<?php
    class Dog {
        // Properties (attributes)
        public $name;
        public $breed;
        public $age;

        // Constructor method (called when a new object is created)
        public function __construct($name, $breed, $age) {
            $this->name = $name;
            $this->breed = $breed;
            $this->age = $age;
            echo "A new dog named " . $this->name . " has been created.<br>";
        }

        // Methods (behaviors)
        public function bark() {
            return $this->name . " says Woof! Woof!<br>";
```
```

```
        }

        public function getInfo() {
            return $this->name . " is a " . $this->age . "-year-old " .
$this->breed . ".<br>";
        }

        // Destructor method (called when the object is destroyed or script
ends)
        public function __destruct() {
            echo $this->name . " is no longer in memory.<br>";
        }
    }

    // Creating objects (instantiating the class)
    $dog1 = new Dog("Buddy", "Golden Retriever", 3);
    $dog2 = new Dog("Lucy", "Labrador", 5);

    // Accessing properties
    echo $dog1->name . "<br>";

    // Calling methods
    echo $dog1->bark();
    echo $dog2->getInfo();

    // Unsetting an object (explicitly calls destructor)
    unset($dog1);
    echo "Buddy object unset.<br>";

    echo "End of script.<br>";
?>
```

15.2. Properties and methods

- **Properties:** Variables defined within a class. They represent the attributes or state of an object.
- **Methods:** Functions defined within a class. They represent the behaviors or actions that an object can perform.

Access Modifiers:

Properties and methods can have access modifiers that control their visibility:

- `public` **:** Accessible from anywhere (inside the class, inherited classes, and outside the class).
- `protected` **:** Accessible only within the class itself and by inherited (child) classes.

- **`private`:** Accessible only within the class that defines it. Not accessible from inherited classes or outside.

Example:

PHP

```
<?php
    class BankAccount {
        public $accountNumber;
        private $balance; // Private property
        protected $accountHolderName;

        public function __construct($accNum, $holderName, $initialBalance) {
            $this->accountNumber = $accNum;
            $this->accountHolderName = $holderName;
            $this->balance = $initialBalance; // Set initial balance
        }

        public function deposit($amount) {
            if ($amount > 0) {
                $this->balance += $amount;
                echo "Deposited $" . $amount . ". New balance: $" . $this-
>balance . "<br>";
            } else {
                echo "Deposit amount must be positive.<br>";
            }
        }

        public function withdraw($amount) {
            if ($amount > 0 && $this->balance >= $amount) {
                $this->balance -= $amount;
                echo "Withdrew $" . $amount . ". New balance: $" . $this-
>balance . "<br>";
            } else {
                echo "Invalid withdrawal amount or insufficient balance.
<br>";
            }
        }

        public function getBalance() {
            return $this->balance; // Public method to access private
property
        }

        protected function getAccountHolder() {
            return $this->accountHolderName;
        }
```

```
    }
\n    $account = new BankAccount("12345", "Jane Doe", 1000);
    echo "Account Number: " . $account->accountNumber . "<br>";
    // echo $account->balance; // Error: Cannot access private property
    // echo $account->accountHolderName; // Error: Cannot access protected
property directly

    $account->deposit(500);
    $account->withdraw(200);
    echo "Current Balance: $ " . $account->getBalance() . "<br>";
?>
```

15.3. Constructors and destructors

- **Constructor (`__construct()`):** A special method that is automatically called when a new object of a class is created. It's typically used to initialize object properties.
- **Destructor (`__destruct()`):** A special method that is automatically called when an object is destroyed or when the script finishes execution. It's often used for cleanup tasks, like closing database connections or releasing resources.

Example: (See `Dog` class example in Section 15.1 for practical usage).

15.4. Inheritance

Inheritance is a mechanism that allows a new class (child or derived class) to inherit properties and methods from an existing class (parent or base class). This promotes code reusability and establishes an

is-a relationship.

Syntax: `class ChildClass extends ParentClass { ... }`

Example:

PHP

```
<?php
    class Animal {
        protected $name;

        public function __construct($name) {
            $this->name = $name;
        }

        public function eat() {
            return $this->name . " is eating.<br>";
        }
```

```
        public function getName() {
            return $this->name;
        }
    }

    class Cat extends Animal {
        public function meow() {
            return $this->name . " says Meow!<br>";
        }

        // Override the eat method
        public function eat() {
            return $this->name . " is eating fish.<br>";
        }
    }

    class Dog extends Animal {
        public function bark() {
            return $this->name . " says Woof!<br>";
        }
    }

    $cat = new Cat("Whiskers");
    echo $cat->eat(); // Output: Whiskers is eating fish.
    echo $cat->meow();
    echo $cat->getName();

    $dog = new Dog("Buddy");
    echo $dog->eat(); // Output: Buddy is eating.
    echo $dog->bark();
?>
```

15.5. Abstraction

Abstraction is the concept of hiding the complex implementation details and showing only the essential features of an object. In PHP, abstract classes and abstract methods are used to achieve abstraction.

- **Abstract Class:** A class that cannot be instantiated on its own and must be inherited by other classes. It can contain both abstract and non-abstract methods.
- **Abstract Method:** A method declared in an abstract class that has no implementation. Child classes must implement all abstract methods declared in their parent abstract class.

Syntax: `abstract class ClassName { abstract public function methodName(); }`

Example:

PHP

```
<?php
    abstract class Vehicle {
        protected $brand;

        public function __construct($brand) {
            $this->brand = $brand;
        }

        abstract public function drive(); // Abstract method (no body)

        public function getBrand() {
            return $this->brand;
        }

        public function stop() {
            return $this->brand . " vehicle stopped.<br>";
        }
    }

    class Car extends Vehicle {
        public function drive() {
            return $this->brand . " car is driving on the road.<br>";
        }
    }

    class Bicycle extends Vehicle {
        public function drive() {
            return $this->brand . " bicycle is being pedaled.<br>";
        }
    }

    // $vehicle = new Vehicle("Generic"); // Error: Cannot instantiate
abstract class

    $myCar = new Car("Toyota");
    echo $myCar->drive();
    echo $myCar->stop();

    $myBicycle = new Bicycle("Giant");
    echo $myBicycle->drive();
?>
```

15.6. Interfaces

An interface defines a contract: it specifies a set of methods that a class must implement. Interfaces contain only method declarations (no method bodies) and constants. A class can implement multiple interfaces.

Syntax: `interface InterfaceName { public function methodName(); }`

Example:

PHP

```
<?php
    interface Logger {
        public function logMessage($message);
        public function logError($error);
    }

    class FileLogger implements Logger {
        private $logFile;

        public function __construct($filename) {
            $this->logFile = $filename;
            file_put_contents($this->logFile, "", FILE_APPEND); // Ensure
file exists
        }

        public function logMessage($message) {
            file_put_contents($this->logFile, "[MESSAGE] " . $message .
"\n", FILE_APPEND);
            echo "Logged message to file.<br>";
        }

        public function logError($error) {
            file_put_contents($this->logFile, "[ERROR] " . $error . "\n",
FILE_APPEND);
            echo "Logged error to file.<br>";
        }
    }

    class DatabaseLogger implements Logger {
        public function logMessage($message) {
            // Logic to log message to a database
            echo "Logged message to database.<br>";
        }

        public function logError($error) {
            // Logic to log error to a database
            echo "Logged error to database.<br>";
```

```
        }
    }

    $fileLogger = new FileLogger("app.log");
    $fileLogger->logMessage("User logged in.");
    $fileLogger->logError("Database connection failed.");

    $dbLogger = new DatabaseLogger();
    $dbLogger->logMessage("New user registered.");
?>
```

15.7. Traits

Traits are a mechanism for code reuse in single inheritance languages like PHP. A trait is a group of methods that you intend to include in multiple classes. Traits reduce the limitations of single inheritance by allowing a class to use multiple traits.

Syntax: `trait TraitName { public function methodName() { ... } }`

Example:

PHP

```
<?php
    trait Greetable {
        public function greet($name) {
            return "Hello, " . $name . "!<br>";
        }
    }

    trait Loggable {
        public function log($message) {
            return "[LOG] " . $message . "\n";
        }
    }

    class User {
        use Greetable, Loggable;

        public $username;

        public function __construct($username) {
            $this->username = $username;
        }

        public function welcome() {
            return $this->greet($this->username);
        }
```

```
    }

    $user = new User("Alice");
    echo $user->welcome();
    echo $user->log("User Alice accessed the system.");
?>
```

15.8. Polymorphism

Polymorphism means "many forms." In OOP, it refers to the ability of objects of different classes to respond to the same method call in their own specific ways. This is often achieved through inheritance and interfaces.

Example:

PHP

```
<?php
    interface Shape {
        public function calculateArea();
    }

    class Circle implements Shape {
        private $radius;

        public function __construct($radius) {
            $this->radius = $radius;
        }

        public function calculateArea() {
            return M_PI * $this->radius * $this->radius;
        }
    }

    class Rectangle implements Shape {
        private $width;
        private $height;

        public function __construct($width, $height) {
            $this->width = $width;
            $this->height = $height;
        }

        public function calculateArea() {
            return $this->width * $this->height;
        }
    }
```

```
    function printArea(Shape $shape) {
        echo "Area: " . $shape->calculateArea() . "<br>";
    }

    $circle = new Circle(5);
    $rectangle = new Rectangle(4, 6);

    printArea($circle); // Calls Circle's calculateArea()
    printArea($rectangle); // Calls Rectangle's calculateArea()
?>
```

16. PHP Advanced Topics

This section delves into more advanced PHP concepts that are crucial for building robust, scalable, and maintainable applications.

16.1. Namespaces

Namespaces are a way of encapsulating items. They allow you to organize your code, prevent naming collisions between classes, functions, and constants, and improve code readability. Namespaces are particularly useful when working with large applications or third-party libraries.

Syntax: `namespace MyProject\SubModule;`

Example:

PHP

```
<?php
    // file: MyProject/Database/Connection.php
    namespace MyProject\Database;

    class Connection {
        public function connect() {
            return "Connecting to database from
MyProject\Database\Connection.<br>";
        }
    }

    // file: MyProject/Utils/Helper.php
    namespace MyProject\Utils;

    class Helper {
        public function formatText($text) {
            return "Formatted: " . strtoupper($text) . " from
```

```
MyProject\Utils\Helper.<br>";
        }
    }

    // file: index.php
    // To use classes from other namespaces, you can use the `use` keyword
    use MyProject\Database\Connection;
    use MyProject\Utils\Helper;

    $dbConnection = new Connection();
    echo $dbConnection->connect();

    $textHelper = new Helper();
    echo $textHelper->formatText("hello world");

    // You can also use fully qualified names
    $anotherConnection = new \MyProject\Database\Connection();
    echo $anotherConnection->connect();
?>
```

16.2. Autoloading classes

Autoloading is a mechanism that allows PHP to automatically load class files only when they are needed, rather than manually including them with `require` or `include` statements. This improves performance and simplifies code management.

`spl_autoload_register()` : The recommended way to implement autoloading. It registers a given function as an `__autoload()` implementation.

Example: PSR-4 Autoloading (common standard)

Let's assume a project structure like this:

Plain Text

```
project/
├── index.php
└── src/
    ├── MyNamespace/
    │   ├── Database/
    │   │   └── Connection.php
    │   └── Models/
    │       └── User.php
```

`src/MyNamespace/Database/Connection.php` :

PHP

```
<?php
namespace MyNamespace\Database;

class Connection {
    public function connect() {
        return "Database connection established.<br>";
    }
}
?>
```

src/MyNamespace/Models/User.php :

```
PHP

<?php
namespace MyNamespace\Models;

class User {
    public function getName() {
        return "John Doe from User Model.<br>";
    }
}
?>
```

index.php :

```
PHP

<?php
    // Autoloader function (simple example, for real projects use Composer's
autoloader)
    spl_autoload_register(function ($className) {
        // Convert namespace to directory structure
        $className = str_replace("\\", DIRECTORY_SEPARATOR, $className);
        $file = __DIR__ . DIRECTORY_SEPARATOR . "src" . DIRECTORY_SEPARATOR
. $className . ".php";

        if (file_exists($file)) {
            require_once $file;
        }
    });

    use MyNamespace\Database\Connection;
    use MyNamespace\Models\User;

    $db = new Connection();
```

```
    echo $db->connect();

    $user = new User();
    echo $user->getName();
?>
```

Note: For real-world applications, **Composer** is the standard tool for dependency management and autoloading in PHP. It generates an optimized autoloader based on PSR-4 standards, making manual autoloading unnecessary for most projects.

16.3. Composer and dependency management

Composer is a dependency manager for PHP. It allows you to declare the libraries your project depends on and it will manage (install/update) them for you. It also provides a convenient autoloader for all your project's classes and its dependencies.

Key Concepts:

- **`composer.json`:** A file where you declare your project's dependencies and other metadata.
- **`composer.lock`:** A file that records the exact versions of all installed dependencies, ensuring consistent installations across environments.
- **`vendor/` directory:** Where Composer installs all the project's dependencies.
- **`composer install`:** Installs all dependencies defined in `composer.json` (and `composer.lock` if it exists).
- **`composer update`:** Updates dependencies to their latest allowed versions and updates `composer.lock`.

Example `composer.json`:

JSON

```
{
    "name": "my-vendor/my-project",
    "description": "A simple PHP project.",
    "type": "project",
    "license": "MIT",
    "autoload": {
        "psr-4": {
            "App\\": "src/"
        }
    },
    "require": {
        "php": ">=7.4",
        "monolog/monolog": "^2.0"
```

```
    },
    "require-dev": {
        "phpunit/phpunit": "^9.0"
    }
}
```

Steps to use Composer:

1. **Install Composer:** Download and install Composer from getcomposer.org.
2. **Create `composer.json`:** Create this file in your project root.
3. **Install Dependencies:** Run `composer install` in your terminal within the project root. This will create the `vendor/` directory and `composer.lock` file.
4. **Include Autoloader:** In your main PHP script, include Composer's autoloader:

'/vendor/autoload.php';

Plain Text

```
    // Now you can use classes from your dependencies and your own
namespaced classes
    use Monolog\Logger;
    use Monolog\Handler\StreamHandler;

    // Create a logger instance
    $log = new Logger("my_app");
    $log->pushHandler(new StreamHandler("app.log", Logger::WARNING));

    // Add some records
    $log->warning("Foo");
    $log->error("Bar");

    echo "Logs written to app.log.<br>";

    // Example of using your own namespaced class (assuming
src/App/MyClass.php exists)
    // use App\MyClass;
    // $myObject = new MyClass();
?>
```

16.4. Exception handling

Exception handling is a structured way to deal with runtime errors in an application. It allows you to separate error-handling code from the normal program logic, making your code cleaner and more robust.

Keywords:

- `try` **:** A block of code that might throw an exception.
- `catch` **:** A block of code that handles the exception if one is thrown in the `try` block.
- `finally` **:** (PHP 5.5+) A block of code that will always be executed, regardless of whether an exception was thrown or caught.
- `throw` **:** Used to explicitly throw an exception.

Example:

PHP

```
<?php
    function divide($numerator, $denominator) {
        if ($denominator === 0) {
            throw new Exception("Division by zero is not allowed.");
        }
        return $numerator / $denominator;
    }

    try {
        echo "Result 1: " . divide(10, 2) . "<br>";
        echo "Result 2: " . divide(5, 0) . "<br>"; // This will throw an
exception
        echo "This line will not be executed.<br>";
    } catch (Exception $e) {
        echo "Caught exception: " . $e->getMessage() . "<br>";
    } finally {
        echo "Finally block executed.<br>";
    }

    echo "<p>Program continues after exception handling.</p>";

    // Custom Exception Class
    class CustomException extends Exception {
        public function __construct($message, $code = 0, Throwable $previous
= null) {
            parent::__construct($message, $code, $previous);
        }

        public function customErrorMessage() {
            return "Custom Error: " . $this->getMessage() . " on line " .
$this->getLine() . " in file " . $this->getFile() . ".<br>";
        }
    }

    function processData($data) {
```

```
        if (!is_numeric($data)) {
            throw new CustomException("Invalid data: Expected a number.");
        }
        return $data * 2;
    }

    try {
        echo "Processed data: " . processData(100) . "<br>";
        echo "Processed data: " . processData("abc") . "<br>";
    } catch (CustomException $e) {
        echo $e->customErrorMessage();
    } catch (Exception $e) { // Catch any other general exceptions
        echo "General Exception: " . $e->getMessage() . "<br>";
    }
?>
```

16.5. Date and time functions

PHP provides a comprehensive set of functions for working with dates and times, allowing you to format, parse, and manipulate date/time values.

- **`date(format, timestamp)`:** Formats a local date/time.
- **`time()`:** Returns the current Unix timestamp (number of seconds since January 1, 1970).
- **`strtotime(time_string)`:** Parses an English textual datetime description into a Unix timestamp.
- **`DateTime` class:** (Recommended for complex operations) Provides an object-oriented way to handle dates and times.

Example:

PHP

```
<?php
    // Current date and time
    echo "Current Date and Time: " . date("Y-m-d H:i:s") . "<br>";
    echo "Current Unix Timestamp: " . time() . "<br>";

    // Formatting dates
    echo "Full Date: " . date("l, F j, Y") . "<br>"; // e.g., Monday, August
18, 2025
    echo "Short Date: " . date("m/d/y") . "<br>"; // e.g., 08/18/25
    echo "Time: " . date("h:i:s A") . "<br>"; // e.g., 03:30:00 PM

    // Using strtotime()
    $nextWeek = strtotime("+1 week");
    echo "Next week: " . date("Y-m-d", $nextWeek) . "<br>";
```

```
    $nextMonth = strtotime("next month");
    echo "Next month: " . date("Y-m-d", $nextMonth) . "<br>";

    // Using DateTime Object (recommended)
    $date = new DateTime(); // Current date and time
    echo "DateTime Object: " . $date->format("Y-m-d H:i:s") . "<br>";

    $date->modify("+1 day");
    echo "Tomorrow: " . $date->format("Y-m-d") . "<br>";

    $specificDate = new DateTime("2025-12-25 10:00:00");
    echo "Specific Date: " . $specificDate->format("Y-m-d H:i:s") . "<br>";

    $interval = $date->diff($specificDate);
    echo "Difference: " . $interval->days . " days, " . $interval->h . "
hours.<br>";
?>
```

17. PHP Security Best Practices

Security is paramount in web development. PHP applications are often targets for various attacks. Implementing security best practices is crucial to protect your application and user data.

17.1. Input sanitization and validation

As discussed in Section 9.4 and 10, input sanitization and validation are the first lines of defense against many attacks. Always treat all user input as potentially malicious.

- **Validation:** Ensure data conforms to expected types, formats, and ranges. Use `filter_var()` for common types (email, URL, int, float) and regular expressions (`preg_match()`) for custom patterns.
- **Sanitization:** Clean or escape data to remove or neutralize potentially harmful characters. Use `htmlspecialchars()` for outputting user-supplied data to HTML, and `strip_tags()` to remove HTML/PHP tags.

Example (reiterating for emphasis):

PHP

```
<?php
    $unsafe_input = "<script>alert(\'XSS\');</script>User Name";
    $email_input = "test@example.com";
    $number_input = "123a";
```

```
    // Sanitization for HTML output
    $safe_html_output = htmlspecialchars($unsafe_input, ENT_QUOTES, 'UTF-8');
    echo "Safe HTML Output: " . $safe_html_output . "<br>";

    // Validation for email
    if (filter_var($email_input, FILTER_VALIDATE_EMAIL)) {
        echo "Valid Email: " . $email_input . "<br>";
    } else {
        echo "Invalid Email: " . $email_input . "<br>";
    }

    // Validation for integer
    if (filter_var($number_input, FILTER_VALIDATE_INT)) {
        echo "Valid Integer: " . $number_input . "<br>";
    } else {
        echo "Invalid Integer: " . $number_input . "<br>"; // This will
output Invalid Integer
    }

    // Sanitization for database insertion (not directly shown here, but
implies prepared statements)
    // $sanitized_for_db = $conn->real_escape_string($unsafe_input); // For
MySQLi (deprecated for prepared statements)
    // Use prepared statements instead for database interaction!
?>
```

17.2. SQL injection prevention (prepared statements)

This is critical and has been covered in detail in Section 13.8. Always use prepared statements with parameterized queries (MySQLi or PDO) for all database interactions involving user input. Never concatenate user input directly into SQL queries.

17.3. Cross-Site Scripting (XSS) prevention

Also covered in Section 9.5. XSS attacks occur when malicious scripts are injected into web pages. The primary defense is to escape all user-supplied data before outputting it to the browser.

- **`htmlspecialchars()`:** Converts special characters to HTML entities. This is the most common and effective way to prevent XSS when displaying user-generated content.
- **`strip_tags()`:** Removes HTML and PHP tags from a string. Useful if you want to allow plain text only.

Example:

PHP

```php
<?php
    $comment = "Hello, <b>world</b>! <script>alert(\'You are hacked!\');
</script>";

    echo "<h3>Original Comment:</h3>";
    echo $comment; // Vulnerable to XSS

    echo "<h3>Safe Comment (htmlspecialchars):</h3>";
    echo htmlspecialchars($comment, ENT_QUOTES, 'UTF-8'); // Prevents XSS

    echo "<h3>Plain Text Comment (strip_tags):</h3>";
    echo strip_tags($comment); // Removes all HTML/script tags
?>
```

17.4. Cross-Site Request Forgery (CSRF) prevention

CSRF attacks trick authenticated users into submitting malicious requests. The primary defense is to use CSRF tokens.

How CSRF Tokens Work:

1. When a form is loaded, the server generates a unique, unpredictable token.
2. This token is embedded in a hidden field within the form and also stored in the user's session.
3. When the form is submitted, the server compares the token from the form with the token in the session.
4. If they don't match, the request is rejected.

Example:

PHP

```php
<?php
    session_start();

    // Generate a CSRF token if one doesn't exist in the session
    if (empty($_SESSION["csrf_token"])) {
        $_SESSION["csrf_token"] = bin2hex(random_bytes(32));
    }

    $message = "";

    if ($_SERVER["REQUEST_METHOD"] == "POST") {
        // Check if the CSRF token is valid
        if (!isset($_POST["csrf_token"]) || $_POST["csrf_token"] !==
```

```
$_SESSION["csrf_token"]) {
            die("CSRF token validation failed. Request blocked.");
        }

        // If token is valid, process the form data
        $action = htmlspecialchars($_POST["action"] ?? "");
        $message = "Action '" . $action . "' performed successfully!";

        // Regenerate token after successful submission to prevent replay
attacks
        $_SESSION["csrf_token"] = bin2hex(random_bytes(32));
    }
?>

<!DOCTYPE html>
<html>
<head>
    <title>CSRF Protection Example</title>
</head>
<body>
    <h2>Perform an Action</h2>
    <form method="post" action="<?php echo
htmlspecialchars($_SERVER["PHP_SELF"]);?>">
        <input type="hidden" name="csrf_token" value="<?php echo
$_SESSION["csrf_token"];?>">
        <label for="action">Action:</label>
        <input type="text" id="action" name="action" value="Transfer Money">
        <input type="submit" value="Submit">
    </form>
    <p style="color:green;"><?php echo $message;?></p>
</body>
</html>
```

17.5. Password hashing

Never store plain text passwords in your database. Always hash them using a strong, one-way hashing algorithm. PHP provides built-in functions for secure password hashing.

- **`password_hash(password, algorithm)` :** Creates a new password hash using a strong one-way hashing algorithm. `PASSWORD_DEFAULT` is the recommended algorithm (currently bcrypt).
- **`password_verify(password, hash)` :** Verifies that a password matches a hash.

Example:

PHP

```php
<?php
    $plainPassword = "mySecretPassword123!";

    // Hash the password
    $hashedPassword = password_hash($plainPassword, PASSWORD_DEFAULT);
    echo "Plain Password: " . $plainPassword . "<br>";
    echo "Hashed Password: " . $hashedPassword . "<br>";

    // Simulate login attempt
    $inputPassword = "mySecretPassword123!"; // User enters this

    if (password_verify($inputPassword, $hashedPassword)) {
        echo "Password verified successfully! User logged in.<br>";
    } else {
        echo "Invalid password. Access denied.<br>";
    }

    // Test with a wrong password
    $wrongPassword = "wrongPassword";
    if (password_verify($wrongPassword, $hashedPassword)) {
        echo "This should not happen: Wrong password verified.<br>";
    } else {
        echo "Correctly rejected wrong password.<br>";
    }
?>
```

17.6. Secure file uploads

This topic was introduced in Section 11.5. To reiterate, handling file uploads securely is critical. Beyond basic checks, consider:

- **Strict MIME Type Validation:** Don't rely solely on file extensions. Use `finfo_open()` or `getimagesize()` to verify the actual MIME type of the uploaded file.
- **Random File Naming:** Rename uploaded files to a unique, random name to prevent path traversal and execution of malicious files.
- **Store Outside Web Root:** Store uploaded files in a directory that is not directly accessible via the web server. If they must be served, use a script to serve them after authorization.
- **Limit File Size:** Prevent denial-of-service attacks.
- **Scan for Malware:** Integrate with antivirus software if possible.

Example (revisiting `upload.php` with more security considerations):

PHP

```php
<?php
    $target_dir = "../uploads/"; // Store outside web root for better
security
    $uploadOk = 1;
    $message = "";

    if(isset($_POST["submit"])) {
        $originalFileName = basename($_FILES["fileToUpload"]["name"]);
        $imageFileType = strtolower(pathinfo($originalFileName,
PATHINFO_EXTENSION));

        // Generate a unique file name to prevent overwrites and direct
execution
        $newFileName = uniqid() . "." . $imageFileType;
        $target_file = $target_dir . $newFileName;

        // Check if file is an actual image (more robust check)
        $finfo = finfo_open(FILEINFO_MIME_TYPE);
        $mimeType = finfo_file($finfo, $_FILES["fileToUpload"]["tmp_name"]);
        finfo_close($finfo);

        $allowedMimeTypes = ["image/jpeg", "image/png", "image/gif"];

        if (!in_array($mimeType, $allowedMimeTypes)) {
            $message .= "Sorry, only JPG, JPEG, PNG & GIF files are allowed
(MIME type check).<br>";
            $uploadOk = 0;
        }

        // Check file size (max 2MB)
        if ($_FILES["fileToUpload"]["size"] > 2000000) {
            $message .= "Sorry, your file is too large (max 2MB).<br>";
            $uploadOk = 0;
        }

        // Check if $uploadOk is set to 0 by an error
        if ($uploadOk == 0) {
            echo "Sorry, your file was not uploaded. " . $message;
        } else {
            // Create uploads directory if it doesn't exist and set
permissions
            if (!is_dir($target_dir)) {
                mkdir($target_dir, 0755, true);
            }

            if (move_uploaded_file($_FILES["fileToUpload"]["tmp_name"],
$target_file)) {
```

```
                echo "The file ". htmlspecialchars($originalFileName) . "
has been uploaded as " . htmlspecialchars($newFileName) . ".<br>";
                echo "You can access it (if served by another script) at: "
. htmlspecialchars($target_file) . "<br>";
            } else {
                echo "Sorry, there was an error uploading your file.<br>";
            }
        }
    }
?>
```

18. PHP and Web Services (APIs)

PHP is frequently used to interact with and build web services (APIs), allowing applications to communicate with each other over the internet. This involves making HTTP requests to external APIs or creating your own API endpoints.

18.1. Consuming RESTful APIs (cURL, file_get_contents)

To consume (make requests to) external RESTful APIs, PHP offers several ways:

- **`file_get_contents()`:** A simple way to make GET requests to URLs. Suitable for basic API calls where you just need to fetch data.
- **cURL (Client URL Library):** A powerful library for making various types of HTTP requests (GET, POST, PUT, DELETE), handling headers, cookies, authentication, and more. Recommended for complex API interactions.

18.1.1. Using `file_get_contents()`

Example: Fetching data from a public API (e.g., JSONPlaceholder)

PHP

```
<?php
    $apiUrl = "https://jsonplaceholder.typicode.com/posts/1";

    // Use file_get_contents to fetch data
    $jsonData = @file_get_contents($apiUrl); // @ suppresses warnings/errors

    if ($jsonData === FALSE) {
        echo "Error fetching data from API.<br>";
    } else {
        $data = json_decode($jsonData, true); // Decode JSON into an
associative array
```

```
        if ($data) {
            echo "<h2>Post Details:</h2>";
            echo "Title: " . htmlspecialchars($data["title"]) . "<br>";
            echo "Body: " . htmlspecialchars($data["body"]) . "<br>";
        } else {
            echo "Failed to decode JSON data.<br>";
        }
    }
?>
```

18.1.2. Using cURL

Example: Making a GET request with cURL

PHP

```
<?php
    $apiUrl = "https://jsonplaceholder.typicode.com/users/1";

    // Initialize cURL session
    $ch = curl_init();

    // Set cURL options
    curl_setopt($ch, CURLOPT_URL, $apiUrl);
    curl_setopt($ch, CURLOPT_RETURNTRANSFER, true); // Return the response
as a string
    curl_setopt($ch, CURLOPT_HEADER, false); // Don't include header in the
output

    // Execute cURL request
    $response = curl_exec($ch);

    // Check for cURL errors
    if (curl_errno($ch)) {
        echo "cURL Error: " . curl_error($ch) . "<br>";
    } else {
        $data = json_decode($response, true);
        if ($data) {
            echo "<h2>User Details:</h2>";
            echo "Name: " . htmlspecialchars($data["name"]) . "<br>";
            echo "Email: " . htmlspecialchars($data["email"]) . "<br>";
            echo "Phone: " . htmlspecialchars($data["phone"]) . "<br>";
        } else {
            echo "Failed to decode JSON data.<br>";
        }
    }
```

```
    // Close cURL session
    curl_close($ch);
?>
```

Example: Making a POST request with cURL

PHP

```
<?php
    $apiUrl = "https://jsonplaceholder.typicode.com/posts";
    $postData = [
        "title" => "foo",
        "body" => "bar",
        "userId" => 1
    ];

    $ch = curl_init();

    curl_setopt($ch, CURLOPT_URL, $apiUrl);
    curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
    curl_setopt($ch, CURLOPT_POST, true); // Set as POST request
    curl_setopt($ch, CURLOPT_POSTFIELDS, json_encode($postData)); // Send
data as JSON
    curl_setopt($ch, CURLOPT_HTTPHEADER, [
        "Content-Type: application/json",
        "Content-Length: " . strlen(json_encode($postData))
    ]);

    $response = curl_exec($ch);

    if (curl_errno($ch)) {
        echo "cURL Error: " . curl_error($ch) . "<br>";
    } else {
        $data = json_decode($response, true);
        if ($data) {
            echo "<h2>New Post Created:</h2>";
            echo "ID: " . htmlspecialchars($data["id"]) . "<br>";
            echo "Title: " . htmlspecialchars($data["title"]) . "<br>";
        } else {
            echo "Failed to decode JSON data.<br>";
        }
    }

    curl_close($ch);
?>
```

18.2. Creating simple RESTful APIs

PHP can be used to create your own RESTful API endpoints that other applications can consume. A simple API typically involves:

1. Receiving HTTP requests (GET, POST, PUT, DELETE).
2. Processing the request (e.g., interacting with a database).
3. Returning a response, usually in JSON format.

Example: Simple PHP REST API (`api.php`)

This example demonstrates a very basic API for managing

a list of users (stored in an array for simplicity).

PHP

```
<?php
    // Set response header to JSON
    header("Content-Type: application/json");

    // Simple in-memory data store (in a real app, this would be a database)
    $users = [
        1 => ["name" => "Alice", "email" => "alice@example.com"],
        2 => ["name" => "Bob", "email" => "bob@example.com"],
        3 => ["name" => "Charlie", "email" => "charlie@example.com"]
    ];

    $method = $_SERVER["REQUEST_METHOD"];
    $request_uri = explode("/", trim($_SERVER["REQUEST_URI"], "/"));
    $resource = $request_uri[1] ?? null; // Assuming URL is like
/api.php/users/1
    $id = $request_uri[2] ?? null;

    if ($resource !== "users") {
        http_response_code(404);
        echo json_encode(["error" => "Resource not found"]);
        exit();
    }

    switch ($method) {
        case "GET":
            if ($id !== null) {
                // Get a specific user
                if (isset($users[$id])) {
                    echo json_encode($users[$id]);
                } else {
                    http_response_code(404);
```

```
                    echo json_encode(["error" => "User not found"]);
                }
            } else {
                // Get all users
                echo json_encode($users);
            }
            break;

        case "POST":
            // Create a new user
            $data = json_decode(file_get_contents("php://input"), true);
            if (isset($data["name"]) && isset($data["email"])) {
                $newId = max(array_keys($users)) + 1;
                $users[$newId] = ["name" => $data["name"], "email" =>
$data["email"]];
                http_response_code(201); // Created
                echo json_encode(["message" => "User created", "id" =>
$newId]);
            } else {
                http_response_code(400); // Bad Request
                echo json_encode(["error" => "Invalid input"]);
            }
            break;

        case "PUT":
            // Update an existing user
            if ($id !== null && isset($users[$id])) {
                $data = json_decode(file_get_contents("php://input"), true);
                if (isset($data["name"]) && isset($data["email"])) {
                    $users[$id] = ["name" => $data["name"], "email" =>
$data["email"]];
                    echo json_encode(["message" => "User updated"]);
                } else {
                    http_response_code(400);
                    echo json_encode(["error" => "Invalid input"]);
                }
            } else {
                http_response_code(404);
                echo json_encode(["error" => "User not found"]);
            }
            break;

        case "DELETE":
            // Delete a user
            if ($id !== null && isset($users[$id])) {
                unset($users[$id]);
                echo json_encode(["message" => "User deleted"]);
            } else {
```

```
                http_response_code(404);
                echo json_encode(["error" => "User not found"]);
            }
            break;

        default:
            http_response_code(405); // Method Not Allowed
            echo json_encode(["error" => "Method not allowed"]);
            break;
    }
?>
```

Note: This is a very basic example. Real-world APIs would use a routing library (like FastRoute) or a micro-framework (like Slim or Lumen) to handle requests more elegantly, and would connect to a database for data persistence.

18.3. Working with JSON

JSON (JavaScript Object Notation) is a lightweight data-interchange format that is easy for humans to read and write and easy for machines to parse and generate. It is the de facto standard for data exchange in modern web APIs.

PHP provides built-in functions for working with JSON:

- **`json_encode(value, flags)` :** Returns the JSON representation of a PHP value (array or object).
- **`json_decode(json_string, associative, depth, flags)` :** Decodes a JSON string into a PHP variable. Setting the second argument `associative` to `true` returns an associative array; otherwise, it returns an object.

Example:

PHP

```
<?php
    // PHP array to JSON
    $user = [
        "id" => 101,
        "name" => "Jane Doe",
        "email" => "jane.doe@example.com",
        "is_active" => true,
        "courses" => ["Math", "Science"]
    ];

    $jsonString = json_encode($user, JSON_PRETTY_PRINT);
    echo "<pre>" . $jsonString . "</pre>";
```

```
    // JSON string to PHP object
    $jsonInput =
    {
        "product_id": "P123",
        "product_name": "Laptop",
        "price": 999.99,
        "tags": ["electronics", "computer"]
    }
    ;

    $productObject = json_decode($jsonInput);
    echo "Product Name (from object): " . $productObject->product_name . "
<br>";
    echo "First Tag (from object): " . $productObject->tags[0] . "<br>";

    // JSON string to PHP associative array
    $productArray = json_decode($jsonInput, true);
    echo "Product Name (from array): " . $productArray["product_name"] . "
<br>";
    echo "First Tag (from array): " . $productArray["tags"][0] . "<br>";

    // Handling JSON errors (PHP 7.3+)
    $invalidJson = "{key: value}"; // Invalid JSON (keys must be in double
quotes)
    json_decode($invalidJson);

    if (json_last_error() !== JSON_ERROR_NONE) {
        echo "JSON Decode Error: " . json_last_error_msg() . "<br>";
    }
?>
```

18.4. Working with XML

XML (eXtensible Markup Language) is another data-interchange format, though it is more verbose than JSON and less commonly used for modern APIs. PHP provides several extensions for working with XML, with `SimpleXML` being one of the easiest.

- **`SimpleXMLElement` :** Represents an XML document as an object, allowing you to access elements and attributes as if they were object properties and array indices.

Example:

PHP

```
<?php
    // XML string
    $xmlString =
    <
```

```
    ?xml version=\"1.0\" encoding=\"UTF-8\"?>
    <bookstore>
        <book category=\"cooking\">
            <title lang=\"en\">Everyday Italian</title>
            <author>Giada De Laurentiis</author>
            <year>2005</year>
            <price>30.00</price>
        </book>
        <book category=\"children\">
            <title lang=\"en\">Harry Potter</title>
            <author>J. K. Rowling</author>
            <year>2005</year>
            <price>29.99</price>
        </book>
    </bookstore>
    ;

    // Load XML string into a SimpleXMLElement object
    $xml = new SimpleXMLElement($xmlString);

    // Accessing elements
    echo "<h2>Bookstore Contents:</h2>";
    foreach ($xml->book as $book) {
        echo "Category: " . $book["category"] . "<br>"; // Accessing
attribute
        echo "Title: " . $book->title . "<br>";
        echo "Author: " . $book->author . "<br>";
        echo "Year: " . $book->year . "<br>";
        echo "Price: " . $book->price . "<br>";
        echo "---<br>";
    }

    // Creating XML from scratch
    $newXml = new SimpleXMLElement("<?xml version=\"1.0\"?><movies>
</movies>");
    $movie1 = $newXml->addChild("movie");
    $movie1->addChild("title", "Inception");
    $movie1->addChild("year", "2010");

    $movie2 = $newXml->addChild("movie");
    $movie2->addChild("title", "The Matrix");
    $movie2->addChild("year", "1999");

    echo "<h2>Generated XML:</h2>";
    echo "<pre>" . htmlspecialchars($newXml->asXML()) . "</pre>";
?>
```

19. PHP and AJAX

AJAX (Asynchronous JavaScript and XML) is a technique for creating fast and dynamic web pages. It allows web pages to be updated asynchronously by exchanging small amounts of data with the server behind the scenes, without having to reload the entire page.

19.1. Introduction to AJAX

AJAX is not a single technology, but a combination of several:

- **HTML/CSS:** For presentation.
- **JavaScript and the DOM:** For dynamic display and interaction.
- **`XMLHttpRequest` or `Fetch API`:** For asynchronous communication with the server.
- **XML or JSON:** As the data format for exchange (JSON is more common).

How AJAX Works:

1. An event occurs on the web page (e.g., a button click).
2. JavaScript creates an `XMLHttpRequest` or `Fetch` request to the server.
3. The server processes the request (e.g., queries a database, performs a calculation).
4. The server sends a response back to the web page.
5. JavaScript receives the response and updates the page content (using the DOM) without reloading the page.

19.2. Making AJAX requests with JavaScript

Modern JavaScript provides the `Fetch API`, which is a simpler and more powerful way to make asynchronous requests compared to the older `XMLHttpRequest` object.

Example: `index.html` (or `index.php`) with JavaScript

HTML

```
<!DOCTYPE html>
<html>
<head>
    <title>PHP and AJAX Example</title>
</head>
<body>

    <h2>Get User Information</h2>
    <input type="text" id="userId" placeholder="Enter User ID (e.g., 1)">
    <button onclick="getUser()">Get User</button>
```

```
    <div id="userInfo"></div>

    <script>
        function getUser() {
            const userId = document.getElementById("userId").value;
            const userInfoDiv = document.getElementById("userInfo");

            if (!userId) {
                userInfoDiv.innerHTML = "Please enter a User ID.";
                return;
            }

            // Using Fetch API to make a GET request
            fetch(`get_user.php?id=${userId}`)
                .then(response => {
                    if (!response.ok) {
                        throw new Error("Network response was not ok");
                    }
                    return response.json(); // Parse JSON response
                })
                .then(data => {
                    if (data.error) {
                        userInfoDiv.innerHTML = `<p
style="color:red;">${data.error}</p>`;
                    } else {
                        userInfoDiv.innerHTML = `
                            <h3>User Details:</h3>
                            <p><strong>Name:</strong> ${data.name}</p>
                            <p><strong>Email:</strong> ${data.email}</p>
                        `;
                    }
                })
                .catch(error => {
                    console.error("Fetch error:", error);
                    userInfoDiv.innerHTML = "<p style="color:red;">Error
fetching user data.</p>";
                });
        }
    </script>

</body>
</html>
```

19.3. Handling AJAX requests in PHP

The PHP script that handles the AJAX request should perform the required server-side logic and then `echo` a response, usually in JSON format. It should not output any HTML unless

that is the intended response.

Example: `get_user.php` (the PHP backend for the AJAX request)

PHP

```
<?php
    // Set response header to JSON
    header("Content-Type: application/json");

    // Simple in-memory data store (in a real app, this would be a database)
    $users = [
        1 => ["name" => "Alice Smith", "email" => "alice.smith@example.com"],
        2 => ["name" => "Bob Johnson", "email" => "bob.johnson@example.com"],
        3 => ["name" => "Charlie Brown", "email" =>
"charlie.brown@example.com"]
    ];

    // Get the user ID from the GET request
    $userId = $_GET["id"] ?? null;

    if ($userId === null) {
        http_response_code(400); // Bad Request
        echo json_encode(["error" => "User ID not provided"]);
        exit();
    }

    // Validate that the ID is a number
    if (!is_numeric($userId)) {
        http_response_code(400);
        echo json_encode(["error" => "Invalid User ID format"]);
        exit();
    }

    // Find the user
    if (isset($users[$userId])) {
        // User found, return their data
        echo json_encode($users[$userId]);
    } else {
        // User not found
        http_response_code(404); // Not Found
        echo json_encode(["error" => "User not found"]);
    }
?>
```

19.4. Live search example

A common use case for AJAX is creating a live search feature, where search results are displayed as the user types, without submitting a form.

Example: `live_search.html` (or `live_search.php`)

HTML

```
<!DOCTYPE html>
<html>
<head>
    <title>Live Search with PHP and AJAX</title>
</head>
<body>

    <h2>Search for a Name</h2>
    <input type="text" id="searchQuery" onkeyup="showHint(this.value)"
placeholder="Start typing a name...">
    <p>Suggestions: <span id="txtHint"></span></p>

    <script>
        function showHint(str) {
            if (str.length == 0) {
                document.getElementById("txtHint").innerHTML = "";
                return;
            }

            fetch(`search.php?q=${str}`)
                .then(response => response.text()) // Get response as plain
text
                .then(data => {
                    document.getElementById("txtHint").innerHTML = data;
                })
                .catch(error => {
                    console.error("Fetch error:", error);
                    document.getElementById("txtHint").innerHTML = "Error
fetching suggestions.";
                });
        }
    </script>

</body>
</html>
```

Example: `search.php` (the PHP backend for the live search)

PHP

```php
<?php
    // Array of names
    $names = ["Anna", "Brittany", "Cinderella", "Diana", "Eva", "Fiona",
"Gunda", "Hege", "Inga", "Johanna", "Kitty", "Linda", "Nina", "Ophelia",
"Petunia", "Amanda", "Raquel", "Cindy", "Doris", "Eve", "Evita", "Sunniva",
"Tove", "Unni", "Violet", "Liza", "Elizabeth", "Ellen", "Wenche", "Vicky"];

    // Get the query parameter from the URL
    $q = $_REQUEST["q"];

    $hint = "";

    // Lookup all hints from array if $q is not empty
    if ($q !== "") {
        $q = strtolower($q);
        $len = strlen($q);
        foreach($names as $name) {
            if (stristr($q, substr($name, 0, $len))) {
                if ($hint === "") {
                    $hint = $name;
                } else {
                    $hint .= ", $name";
                }
            }
        }
    }

    // Output "no suggestion" if no hint was found or output correct values
    echo $hint === "" ? "no suggestion" : $hint;
?>
```

20. PHP Frameworks

PHP frameworks provide a structured and standardized way to build web applications. They offer pre-built components, libraries, and tools that streamline development, improve code organization, and enforce best practices.

20.1. Introduction to PHP frameworks

Why use a PHP framework?

- **Speed and Efficiency:** Frameworks provide ready-to-use components for common tasks (routing, database access, authentication), saving development time.
- **Structure and Organization:** They enforce a specific structure (like MVC - Model-View-Controller), leading to cleaner, more maintainable code.

- **Security:** Frameworks often have built-in security features to protect against common vulnerabilities like SQL injection, XSS, and CSRF.
- **Best Practices:** They encourage modern development practices and design patterns.
- **Community and Support:** Popular frameworks have large communities, extensive documentation, and active support.

MVC (Model-View-Controller) Architecture:

Many PHP frameworks are based on the MVC architectural pattern, which separates the application logic into three interconnected components:

- **Model:** Represents the data and business logic of the application. It interacts with the database.
- **View:** The user interface. It displays the data provided by the model.
- **Controller:** Handles user requests, interacts with the model to retrieve data, and then passes that data to the view for rendering.

20.2. Popular PHP frameworks (Laravel, Symfony, CodeIgniter)

There are many PHP frameworks available, each with its own strengths and weaknesses. Here are some of the most popular ones:

- **Laravel:**
  - **Description:** Currently the most popular PHP framework, known for its elegant syntax, extensive features, and developer-friendly tools.
  - **Key Features:** Eloquent ORM (Object-Relational Mapper), Blade templating engine, Artisan command-line tool, built-in authentication and authorization, robust routing, and a large ecosystem of packages (e.g., Laravel Nova, Telescope).
  - **Best for:** A wide range of applications, from small projects to large enterprise systems. Excellent for developers who value speed, elegance, and a rich feature set.
- **Symfony:**
  - **Description:** A set of reusable PHP components and a high-performance PHP framework for web applications. It is known for its stability, flexibility, and adherence to standards.
  - **Key Features:** A large set of decoupled and reusable components, Doctrine ORM, Twig templating engine, robust dependency injection container, and strong community support. Laravel itself is built on top of many Symfony components.
  - **Best for:** Large, complex, and long-term enterprise projects where flexibility and maintainability are paramount.

- **CodeIgniter:**
  - **Description:** A lightweight and fast framework with a small footprint. It is known for its simplicity, excellent performance, and clear documentation.
  - **Key Features:** Simple and easy to learn, exceptional performance, requires minimal configuration, and provides simple solutions without imposing strict coding rules.
  - **Best for:** Small to medium-sized applications where performance is a key concern and developers prefer a more minimalist approach.

20.3. Getting started with a framework (e.g., Laravel)

Getting started with a modern framework like Laravel typically involves using Composer to create a new project.

Prerequisites:

- PHP installed
- Composer installed

Steps to create a new Laravel project:

1. **Install the Laravel Installer:**
2. **Create a new project:**
3. **Navigate to the project directory:**
4. **Start the development server:**

Example: Creating a simple route and view in Laravel

1. **Define a route in `routes/web.php`:**
2. **Create a new view file in `resources/views/greeting.blade.php`:**

Now, if you navigate to `http://127.0.0.1:8000/greeting` in your browser, you will see the greeting message with the name "Alice". This demonstrates the basic routing and view system in Laravel.

21. PHP and MySQL Project: Simple Blog

This section provides a step-by-step guide to building a simple blog application using PHP and MySQL. This project will integrate many of the concepts covered in this tutorial, including database interaction, form handling, and security.

21.1. Project setup and database design

Project Structure:

Plain Text

```
/simple_blog/
├── config.php        # Database configuration
├── index.php         # Main page, lists all posts
├── post.php          # Displays a single post
├── new_post.php      # Form to create a new post
├── edit_post.php     # Form to edit an existing post
├── delete_post.php   # Script to delete a post
└── style.css         # Basic CSS for styling
```

Database Design:

We will create a database named `simple_blog_db` with a single table named `posts`.

`posts` table structure:

- `id` (INT, PRIMARY KEY, AUTO_INCREMENT)
- `title` (VARCHAR(255), NOT NULL)
- `body` (TEXT, NOT NULL)
- `created_at` (TIMESTAMP, DEFAULT CURRENT_TIMESTAMP)

SQL to create the database and table:

SQL

```
CREATE DATABASE IF NOT EXISTS simple_blog_db;
USE simple_blog_db;

CREATE TABLE IF NOT EXISTS posts (
    id INT AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(255) NOT NULL,
    body TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

21.2. Creating the database connection (`config.php`)

Create a configuration file to store database credentials and establish a connection. This makes it easy to manage connection details in one place.

`config.php`:

PHP

```
<?php
    define("DB_SERVER", "localhost");
```

```
    define("DB_USERNAME", "root");
    define("DB_PASSWORD", ""); // Your MySQL password
    define("DB_NAME", "simple_blog_db");

    // Create connection using MySQLi (Object-Oriented)
    $conn = new mysqli(DB_SERVER, DB_USERNAME, DB_PASSWORD, DB_NAME);

    // Check connection
    if ($conn->connect_error) {
        die("Connection failed: " . $conn->connect_error);
    }
?>
```

21.3. Displaying posts (`index.php`)

This is the main page of the blog. It will fetch all posts from the database and display them.

`index.php` :

PHP

```
<?php
    require_once "config.php";

    // Fetch all posts
    $sql = "SELECT id, title, body, created_at FROM posts ORDER BY
created_at DESC";
    $result = $conn->query($sql);
?>

<!DOCTYPE html>
<html>
<head>
    <title>Simple Blog</title>
    <link rel="stylesheet" type="text/css" href="style.css">
</head>
<body>
    <div class="container">
        <h1>My Simple Blog</h1>
        <a href="new_post.php" class="btn">+ New Post</a>

        <?php if ($result->num_rows > 0): ?>
            <?php while($row = $result->fetch_assoc()): ?>
                <div class="post">
                    <h2><a href="post.php?id=<?php echo $row["id"]; ?>"><?
php echo htmlspecialchars($row["title"]); ?></a></h2>
                    <p><?php echo
nl2br(htmlspecialchars(substr($row["body"], 0, 150))); ?>...</p>
```

```
                    <small>Posted on <?php echo date("F j, Y",
strtotime($row["created_at"])); ?></small>
                    <br>
                    <a href="edit_post.php?id=<?php echo $row["id"]; ?>"
class="btn-edit">Edit</a>
                    <a href="delete_post.php?id=<?php echo $row["id"]; ?>"
class="btn-delete" onclick="return confirm(\"Are you sure you want to delete
this post?\");">Delete</a>
                </div>
            <?php endwhile; ?>
        <?php else: ?>
            <p>No posts found.</p>
        <?php endif; ?>
    </div>
</body>
</html>

<?php
    $conn->close();
?>
```

21.4. Creating new posts (`new_post.php`)

This file contains the form for creating a new blog post and the PHP logic to handle the form submission.

`new_post.php` :

PHP

```
<?php
    require_once "config.php";

    $title = $body = "";
    $title_err = $body_err = "";

    if ($_SERVER["REQUEST_METHOD"] == "POST") {
        // Validate title
        if (empty(trim($_POST["title"]))) {
            $title_err = "Please enter a title.";
        } else {
            $title = trim($_POST["title"]);
        }

        // Validate body
        if (empty(trim($_POST["body"]))) {
            $body_err = "Please enter the post content.";
        } else {
```

```
            $body = trim($_POST["body"]);
        }

        // Check for errors before inserting into database
        if (empty($title_err) && empty($body_err)) {
            $sql = "INSERT INTO posts (title, body) VALUES (?, ?)";

            if ($stmt = $conn->prepare($sql)) {
                $stmt->bind_param("ss", $param_title, $param_body);

                $param_title = $title;
                $param_body = $body;

                if ($stmt->execute()) {
                    header("location: index.php");
                    exit();
                } else {
                    echo "Something went wrong. Please try again later.";
                }
                $stmt->close();
            }
        }
        $conn->close();
    }
?>

<!DOCTYPE html>
<html>
<head>
    <title>New Post</title>
    <link rel="stylesheet" type="text/css" href="style.css">
</head>
<body>
    <div class="container">
        <h2>Create New Post</h2>
        <form action="<?php echo htmlspecialchars($_SERVER["PHP_SELF"]); ?>"
method="post">
            <div class="form-group">
                <label>Title</label>
                <input type="text" name="title" value="<?php echo $title; ?
>">
                <span class="error"><?php echo $title_err; ?></span>
            </div>
            <div class="form-group">
                <label>Body</label>
                <textarea name="body"><?php echo $body; ?></textarea>
                <span class="error"><?php echo $body_err; ?></span>
            </div>
```

```
            <div class="form-group">
                <input type="submit" class="btn" value="Submit">
                <a href="index.php" class="btn-cancel">Cancel</a>
            </div>
        </form>
    </div>
</body>
</html>
```

21.5. Editing and deleting posts (`edit_post.php` , `delete_post.php`)

`edit_post.php` :

This file will fetch the existing post data, display it in a form, and handle the update logic.

PHP

```
<?php
    require_once "config.php";

    $title = $body = "";
    $title_err = $body_err = "";
    $id = $_GET["id"];

    // Processing form data when form is submitted
    if ($_SERVER["REQUEST_METHOD"] == "POST") {
        $id = $_POST["id"];

        // Validate title
        if (empty(trim($_POST["title"]))) {
            $title_err = "Please enter a title.";
        } else {
            $title = trim($_POST["title"]);
        }

        // Validate body
        if (empty(trim($_POST["body"]))) {
            $body_err = "Please enter the post content.";
        } else {
            $body = trim($_POST["body"]);
        }

        // Check for errors before updating database
        if (empty($title_err) && empty($body_err)) {
            $sql = "UPDATE posts SET title = ?, body = ? WHERE id = ?";

            if ($stmt = $conn->prepare($sql)) {
                $stmt->bind_param("ssi", $param_title, $param_body,
```

```
$param_id);

                $param_title = $title;
                $param_body = $body;
                $param_id = $id;

                if ($stmt->execute()) {
                    header("location: index.php");
                    exit();
                } else {
                    echo "Something went wrong. Please try again later.";
                }
                $stmt->close();
            }
        }
    } else {
        // Get existing data for the form
        if (isset($_GET["id"]) && !empty(trim($_GET["id"]))) {
            $id = trim($_GET["id"]);
            $sql = "SELECT * FROM posts WHERE id = ?";
            if ($stmt = $conn->prepare($sql)) {
                $stmt->bind_param("i", $param_id);
                $param_id = $id;

                if ($stmt->execute()) {
                    $result = $stmt->get_result();
                    if ($result->num_rows == 1) {
                        $row = $result->fetch_assoc();
                        $title = $row["title"];
                        $body = $row["body"];
                    } else {
                        echo "No record found.";
                        exit();
                    }
                } else {
                    echo "Oops! Something went wrong.";
                    exit();
                }
                $stmt->close();
            }
        } else {
            echo "Invalid request.";
            exit();
        }
    }
    $conn->close();
?>
```

```
<!DOCTYPE html>
<html>
<head>
    <title>Edit Post</title>
    <link rel="stylesheet" type="text/css" href="style.css">
</head>
<body>
    <div class="container">
        <h2>Edit Post</h2>
        <form action="<?php echo
htmlspecialchars(basename($_SERVER["REQUEST_URI"])); ?>" method="post">
            <input type="hidden" name="id" value="<?php echo $id; ?>"/>
            <div class="form-group">
                <label>Title</label>
                <input type="text" name="title" value="<?php echo $title; ?
>">
                <span class="error"><?php echo $title_err; ?></span>
            </div>
            <div class="form-group">
                <label>Body</label>
                <textarea name="body"><?php echo $body; ?></textarea>
                <span class="error"><?php echo $body_err; ?></span>
            </div>
            <div class="form-group">
                <input type="submit" class="btn" value="Update">
                <a href="index.php" class="btn-cancel">Cancel</a>
            </div>
        </form>
    </div>
</body>
</html>
```

delete_post.php :

This script will handle the deletion of a post.

PHP

```
<?php
    require_once "config.php";

    if (isset($_GET["id"]) && !empty(trim($_GET["id"]))) {
        $id = trim($_GET["id"]);

        $sql = "DELETE FROM posts WHERE id = ?";

        if ($stmt = $conn->prepare($sql)) {
            $stmt->bind_param("i", $param_id);
```

```
            $param_id = $id;

            if ($stmt->execute()) {
                header("location: index.php");
                exit();
            } else {
                echo "Oops! Something went wrong. Please try again later.";
            }
            $stmt->close();
        }
    } else {
        echo "Invalid request.";
    }
    $conn->close();
?>
```

21.6. Styling the blog (`style.css`)

Create a basic stylesheet to make the blog look presentable.

`style.css`:

CSS

```
body {
    font-family: Arial, sans-serif;
    background-color: #f4f4f4;
    color: #333;
    line-height: 1.6;
}

.container {
    width: 80%;
    margin: auto;
    overflow: hidden;
    padding: 20px;
}

.post {
    background: #fff;
    padding: 20px;
    margin-bottom: 20px;
    border: 1px solid #ddd;
}

.post h2 a {
    text-decoration: none;
    color: #333;
```

```
}

.btn {
    display: inline-block;
    background: #337ab7;
    color: #fff;
    padding: 8px 15px;
    text-decoration: none;
    border: none;
    cursor: pointer;
}

.btn-edit {
    background: #f0ad4e;
    color: #fff;
    padding: 5px 10px;
    text-decoration: none;
}

.btn-delete {
    background: #d9534f;
    color: #fff;
    padding: 5px 10px;
    text-decoration: none;
}

.btn-cancel {
    background: #777;
    color: #fff;
    padding: 8px 15px;
    text-decoration: none;
}

.form-group {
    margin-bottom: 15px;
}

.form-group label {
    display: block;
    margin-bottom: 5px;
}

.form-group input[type="text"],
.form-group textarea {
    width: 100%;
    padding: 8px;
    box-sizing: border-box;
}
```

```
.error {
    color: red;
    font-size: 0.9em;
}
```

This completes the simple blog project, demonstrating key PHP and MySQL concepts in a practical application.

22. Advanced MySQL Topics

This section covers more advanced MySQL topics that are important for building scalable, efficient, and robust database-driven applications.

22.1. Joins (INNER, LEFT, RIGHT, FULL)

Joins are used to combine rows from two or more tables based on a related column between them. They are fundamental to querying relational databases.

Setup for Join Examples:

Let's assume we have two tables: `customers` and `orders`.

SQL

```
CREATE TABLE customers (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL
);

CREATE TABLE orders (
    id INT AUTO_INCREMENT PRIMARY KEY,
    customer_id INT,
    product VARCHAR(255) NOT NULL,
    amount DECIMAL(10, 2),
    FOREIGN KEY (customer_id) REFERENCES customers(id)
);

INSERT INTO customers (name) VALUES (
    ("Alice"),
    ("Bob"),
    ("Charlie"),
    ("Diana")
);

INSERT INTO orders (customer_id, product, amount) VALUES
    (1, "Laptop", 1200.00),
```

```
    (1, "Mouse", 25.00),
    (2, "Keyboard", 75.00),
    (NULL, "Monitor", 300.00); -- An order with no customer
```

- **`INNER JOIN`:** Returns records that have matching values in both tables. It effectively finds the intersection of the two tables.
- **`LEFT JOIN` (or `LEFT OUTER JOIN`):** Returns all records from the left table (`customers`), and the matched records from the right table (`orders`). The result is `NULL` from the right side if there is no match.
- **`RIGHT JOIN` (or `RIGHT OUTER JOIN`):** Returns all records from the right table (`orders`), and the matched records from the left table (`customers`). The result is `NULL` from the left side if there is no match.
- **`FULL OUTER JOIN`:** Returns all records when there is a match in either the left or right table. MySQL does not directly support `FULL OUTER JOIN`, but it can be emulated using a `UNION` of `LEFT JOIN` and `RIGHT JOIN`.

22.2. Transactions

Transactions are a sequence of operations performed as a single logical unit of work. They are used to ensure data integrity. A transaction has four key properties, known as ACID:

- **Atomicity:** All operations within a transaction are completed successfully, or none of them are. If one part fails, the entire transaction is rolled back.
- **Consistency:** The database remains in a consistent state before and after the transaction.
- **Isolation:** Transactions are isolated from each other. Concurrent transactions do not interfere with each other.
- **Durability:** Once a transaction is committed, its changes are permanent, even in the event of a system failure.

Transaction Control Statements:

- **`START TRANSACTION` (or `BEGIN`):** Starts a new transaction.
- **`COMMIT`:** Saves all changes made in the transaction.
- **`ROLLBACK`:** Undoes all changes made in the transaction.

Example: Bank Transfer (using PDO)

PHP

```
<?php
    require_once "config.php"; // Assuming config.php sets up a PDO
```

```
connection as $pdo

    try {
        // Start a transaction
        $pdo->beginTransaction();

        // 1. Debit from Account A
        $stmt1 = $pdo->prepare("UPDATE accounts SET balance = balance - 100
WHERE id = 1");
        $stmt1->execute();

        // 2. Credit to Account B
        $stmt2 = $pdo->prepare("UPDATE accounts SET balance = balance + 100
WHERE id = 2");
        $stmt2->execute();

        // If both queries succeed, commit the transaction
        $pdo->commit();
        echo "Transaction successful: Money transferred.<br>";

    } catch (Exception $e) {
        // If any query fails, roll back the transaction
        $pdo->rollBack();
        echo "Transaction failed: " . $e->getMessage() . "<br>";
    }
?>
```

22.3. Indexes

Indexes are special lookup tables that the database search engine can use to speed up data retrieval. An index is a pointer to data in a table. Creating an index on a column allows the database to find the data without scanning the entire table.

When to use indexes:

- On columns that are frequently used in `WHERE` clauses.
- On columns used in `JOIN` conditions.
- On columns used in `ORDER BY` and `GROUP BY` clauses.

When to avoid indexes:

- On tables with a small number of rows.
- On columns with a high number of `NULL` values.
- On columns that are frequently updated (as indexes also need to be updated, which can slow down write operations).

Example:

```
SQL

-- Create an index on the `email` column of the `Users` table
CREATE INDEX idx_email ON Users (email);

-- Create a composite index on multiple columns
CREATE INDEX idx_name ON Users (lastname, firstname);

-- Show existing indexes on a table
SHOW INDEX FROM Users;

-- Drop an index
DROP INDEX idx_email ON Users;
```

22.4. Stored procedures

A stored procedure is a set of SQL statements that are stored in the database. It can be called by applications, other stored procedures, or triggers. Stored procedures can accept input parameters and return output parameters.

Benefits of Stored Procedures:

- **Performance:** Stored procedures are pre-compiled and stored in the database, which can improve performance.
- **Reduced Network Traffic:** Instead of sending multiple SQL statements over the network, an application can just call the stored procedure.
- **Security:** You can grant permissions to execute a stored procedure without granting permissions to the underlying tables.
- **Reusability and Maintainability:** Business logic can be encapsulated in the database, making it reusable and easier to maintain.

Example:

```
SQL

-- Create a stored procedure to get a user by ID
DELIMITER //

CREATE PROCEDURE GetUserById(IN userId INT)
BEGIN
    SELECT id, firstname, lastname, email FROM Users WHERE id = userId;
END //
```

```
DELIMITER ;

-- Call the stored procedure
CALL GetUserById(1);
```

Calling a Stored Procedure from PHP (PDO):

PHP

```
<?php
    require_once "config.php"; // PDO connection as $pdo

    try {
        $stmt = $pdo->prepare("CALL GetUserById(?)");
        $userId = 1;
        $stmt->bindParam(1, $userId, PDO::PARAM_INT);
        $stmt->execute();

        $user = $stmt->fetch(PDO::FETCH_ASSOC);

        if ($user) {
            echo "User found: " . htmlspecialchars($user["firstname"]) . " "
. htmlspecialchars($user["lastname"]) . "<br>";
        } else {
            echo "User not found.<br>";
        }

    } catch (PDOException $e) {
        echo "Error: " . $e->getMessage();
    }
?>
```

22.5. Triggers

A trigger is a special type of stored procedure that automatically runs when a specific event (e.g., `INSERT`, `UPDATE`, `DELETE`) occurs on a table. Triggers are often used to maintain data integrity, log changes, or enforce complex business rules.

Example: Audit Log Trigger

Let's create an `audit_log` table to track changes to the `Users` table.

SQL

```
CREATE TABLE audit_log (
    id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT,
```

```
    action VARCHAR(50),
    change_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Create a trigger that runs after a new user is inserted
DELIMITER //

CREATE TRIGGER after_user_insert
AFTER INSERT ON Users
FOR EACH ROW
BEGIN
    INSERT INTO audit_log (user_id, action) VALUES (NEW.id, "New user
created");
END //

DELIMITER ;

-- Create a trigger that runs after a user is updated
DELIMITER //

CREATE TRIGGER after_user_update
AFTER UPDATE ON Users
FOR EACH ROW
BEGIN
    INSERT INTO audit_log (user_id, action) VALUES (NEW.id, "User
information updated");
END //

DELIMITER ;
```

Now, whenever you `INSERT` a new row into the `Users` table or `UPDATE` an existing one, a corresponding entry will be automatically created in the `audit_log` table.

23. Final Project: Building a Complete Web Application

This final project aims to bring together all the concepts learned throughout this tutorial to build a more complete web application. We will create a simple e-commerce product listing site with user registration and login.

23.1. Project planning and database schema

Features:

- User registration and login system.
- Product listing page (visible to all).
- Product detail page.

- Authenticated users can add products to a shopping cart.
- Shopping cart page.

Database Schema (`ecommerce_db`):

- **`users` table:**
  - `id` (INT, PK, AI)
  - `username` (VARCHAR(50), UNIQUE, NOT NULL)
  - `email` (VARCHAR(100), UNIQUE, NOT NULL)
  - `password` (VARCHAR(255), NOT NULL)
  - `created_at` (TIMESTAMP)
- **`products` table:**
  - `id` (INT, PK, AI)
  - `name` (VARCHAR(255), NOT NULL)
  - `description` (TEXT)
  - `price` (DECIMAL(10, 2), NOT NULL)
  - `image_url` (VARCHAR(255))
- **`cart` table:**
  - `id` (INT, PK, AI)
  - `user_id` (INT, FK to `users.id`)
  - `product_id` (INT, FK to `products.id`)
  - `quantity` (INT, NOT NULL)
  - `added_at` (TIMESTAMP)

23.2. User authentication (registration and login)

This involves creating forms for registration and login, and PHP scripts to handle them securely.

- **Registration (`register.php`):**
  - Form to collect username, email, and password.
  - Validate input (check for empty fields, valid email, password strength).
  - Check if username or email already exists in the database.
  - Hash the password using `password_hash()` .
  - Insert the new user into the `users` table.

- **Login (`login.php`):**
  - Form to collect username and password.
  - Fetch user from the database based on username.
  - If user exists, verify the password using `password_verify()` .
  - If password is correct, start a session, store user ID and username, and regenerate the session ID (`session_regenerate_id(true)`).
  - Redirect to the main page or user dashboard.
- **Logout (`logout.php`):**
  - Destroy the session and redirect to the login page.

23.3. Product listing and detail pages

- **Product Listing (`index.php`):**
  - Fetch all products from the `products` table.
  - Display them in a grid or list format, showing image, name, and price.
  - Each product should link to its detail page.
- **Product Detail (`product.php`):**
  - Get product ID from the URL (`?id=...`).
  - Fetch the specific product from the database.
  - Display all product details (image, name, description, price).
  - If the user is logged in, show an "Add to Cart" button.

23.4. Shopping cart functionality

- **Add to Cart (`add_to_cart.php`):**
  - This script will be the target of the "Add to Cart" form.
  - Requires the user to be logged in (check session).
  - Get `product_id` and `quantity` from the form submission.
  - Check if the product already exists in the user's cart.
    - If yes, update the quantity.
    - If no, insert a new row into the `cart` table.
  - Redirect back to the product page or to the cart page with a success message.
- **Shopping Cart Page (`cart.php`):**
  - Requires the user to be logged in.

- Fetch all items in the user's cart by joining the `cart` and `products` tables.
- Display the cart items in a table (product name, price, quantity, subtotal).
- Provide options to update quantity or remove items from the cart.
- Display the total price of all items in the cart.

23.5. Putting it all together (code structure and final thoughts)

Code Structure:

- Use a consistent structure (e.g., separate files for config, templates, and logic).
- Use a `header.php` and `footer.php` to include common HTML head/foot content and avoid code duplication.
- Organize your code into functions or classes for better readability and reusability.
- Implement all security best practices discussed (prepared statements, output escaping, CSRF protection, session security).

This final project serves as a comprehensive review and practical application of the skills learned. By building it, you will gain a solid understanding of how to create dynamic, database-driven web applications with PHP and MySQL.