Introduction to PHP

Abdulla Ebrahim Subah

October 22, 2024

Introduciton to Server-side Development

Static Sites

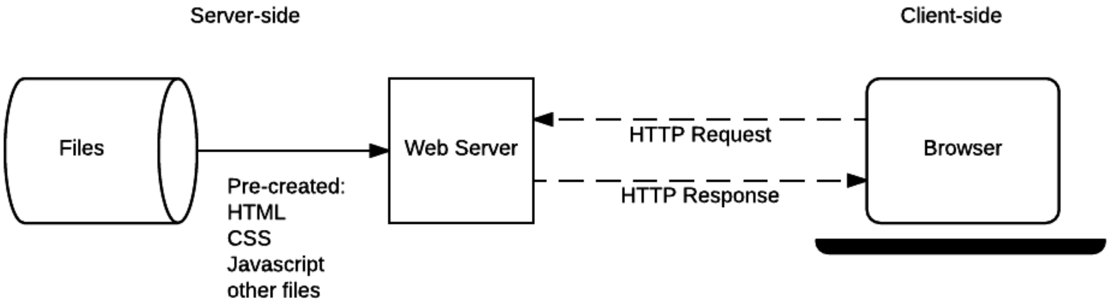

Figure 1: Basic Static App Server

Image Source: MDN Web Docs

Dynamic Sites

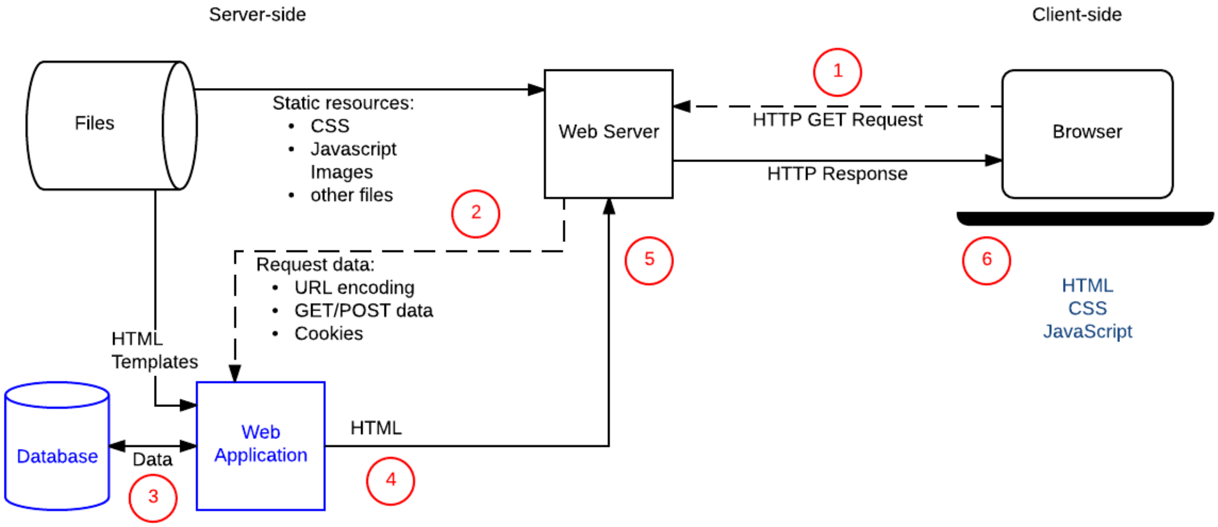

Image Source: MDN Web Docs)

Server-Side Languages

Server-side languages are used to create dynamic web pages. They run on the server and generate HTML that is sent to the client's browser. Examples include:

- **PHP**: Widely used for web development and can be embedded into HTML.
- **Python**: Known for its readability, often used with frameworks like Django and Flask.
- **Ruby**: Popular for its simplicity and productivity, commonly used with Ruby on Rails.
- **JavaScript (Node.js)**: Allows JavaScript to be used for server-side scripting, enabling full-stack development with a single language.

Server-Side Databases

Server-side databases are used to store and manage data for web applications. They run on the server and can be accessed by server-side languages to perform CRUD (Create, Read, Update, Delete) operations. Examples include:

- **MySQL**: An open-source relational database management system.
- **PostgreSQL**: A powerful, open-source object-relational database system.
- **SQLite**: A self-contained, serverless, and zero-configuration database engine.
- **Microsoft SQL Server**: A relational database management system developed by Microsoft.
- **Oracle Database**: A multi-model database management system produced by Oracle Corporation.

NoSQL Databases

NoSQL databases are designed to handle large volumes of data and are optimized for specific data models. Examples include:

- **MongoDB**: A document-oriented NoSQL database known for its flexibility and scalability.
- **Cassandra**: A wide-column store NoSQL database designed for high availability and scalability.
- **Redis**: An in-memory key-value store known for its speed and performance.
- **CouchDB**: A document-oriented NoSQL database that uses JSON to store data.
- **Neo4j**: A graph database that uses graph structures for queries with nodes, edges, and properties.

Web Servers

Web servers are software applications that serve web pages to users upon request. They handle HTTP requests from clients (browsers) and deliver the requested resources. Examples include:

- **Apache HTTP Server**: A widely-used open-source web server known for its flexibility and power.
- **Nginx**: A high-performance web server and reverse proxy server known for its speed and scalability.
- **Microsoft Internet Information Services (IIS)**: A web server created by Microsoft for use with Windows Server.

Selected Stack for This Course

In this course, we will be using the following stack for our server-side development:

- **Apache HTTP Server**: A widely-used open-source web server known for its flexibility and power. It will handle HTTP requests and serve web pages to users.
- **MySQL/MariaDB**: Both are open-source relational database management systems. MySQL is known for its reliability and performance, while MariaDB is a fork of MySQL with additional features and improvements.
- **PHP**: A popular server-side scripting language that is widely used for web development. It can be embedded into HTML and is known for its ease of use and flexibility.

This stack is commonly referred to as the AMP stack (Apache, MySQL/MariaDB, PHP) and is a powerful combination for building dynamic web applications.

Introduction to PHP

Wikipedia:

> **PHP** is a general-purpose scripting language geared towards web development. It was originally created by Danish-Canadian programmer Rasmus Lerdorf in 1993 and released in 1995. The

> PHP reference implementation is now produced by the PHP Group. PHP was originally an abbreviation of **Personal Home Page**, but it now stands for the recursive acronym **PHP: Hypertext Preprocessor**.

PHP Language Manual: https://www.php.net/manual/en/langref.php

PHP Variables

```php
<?php
$var = 'Bob';
$Var = 'Joe';

// outputs "Bob, Joe"
echo "$var, $Var";

// invalid; starts with a number
$4site = 'not yet';

// valid; starts with an underscore
$_4site = 'not yet';

// valid; 'ä' is (Extended) ASCII 228.
$täyte = 'mansikka';

// Assign the value 'Bob' to $foo
$foo = 'Bob';

// Reference $foo via $bar.
$bar = &$foo;

// Alter $bar...
$bar = "My name is $bar";
echo $bar;

// $foo is altered too.
echo $foo;
?>
```

PHP Data Types

PHP supports various data types, which can be categorized as follows:

- **null**: Represents a variable with no value.
- **bool**: Represents a boolean value, either `true` or `false`.
- **int**: Represents an integer value.
- **float**: Represents a floating-point number (also known as double).
- **string**: Represents a sequence of characters.
- **array**: Represents a collection of values, which can be indexed by integers or strings.
- **object**: Represents an instance of a class.
- **callable**: Represents a function that can be called.
- **resource**: Represents a reference to an external resource, such as a file or a database connection.

```php
<?php
// Unset AND unreferenced (no use context) variable;
// outputs NULL
var_dump($unset_var);

// Boolean usage; outputs 'false'
// (See ternary operators for more on this syntax)
echo $unset_bool ? "true\n" : "false\n";

// String usage; outputs 'string(3) "abc"'
$unset_str .= 'abc';
var_dump($unset_str);

// Integer usage; outputs 'int(25)'
$unset_int += 25; // 0 + 25 => 25
var_dump($unset_int);
// Float usage; outputs 'float(1.25)'
$unset_float += 1.25;
var_dump($unset_float);
// Array usage; outputs array(1) {  [3]=>  string(3) "def" }
// array() + array(3 => "def") => array(3 => "def")
$unset_arr[3] = "def";
var_dump($unset_arr);
// Object usage; creates new stdClass object
// (see http://www.php.net/manual/en/reserved.classes.php)
// Outputs: object(stdClass)#1 (1) {  ["foo"]=>  string(3) "bar" }
$unset_obj->foo = 'bar';
var_dump($unset_obj);
?>
```

Predefined PHP Variables

PHP provides a range of predefined variables that are accessible in all scopes. These variables are known as superglobals and include:

- **$GLOBALS**: References all variables available in the global scope.
- **$_SERVER**: Contains information about the server and execution environment.
- **$_GET**: Contains HTTP GET variables.
- **$_POST**: Contains HTTP POST variables.
- **$_FILES**: Contains HTTP File Upload variables.
- **$_REQUEST**: Contains HTTP Request variables.
- **$_SESSION**: Contains session variables.
- **$_ENV**: Contains environment variables.
- **$_COOKIE**: Contains HTTP Cookies.
- **$php_errormsg**: Contains the previous error message.
- **$http_response_header**: Contains HTTP response headers.
- **$argc**: Contains the number of arguments passed to the script.
- **$argv**: Contains an array of arguments passed to the script.

if else

```
<?php
if ($a > $b) {
    echo "a is bigger than b";
} elseif ($a == $b) {
    echo "a is equal to b";
} else {
    echo "a is smaller than b";
}
?>
```

for loops

```php
<?php
/*
 * This is an array with some data we want to modify
 * when running through the for loop.
 */
$people = array(
    array('name' => 'Kalle', 'salt' => 856412),
    array('name' => 'Pierre', 'salt' => 215863)
);

for($i = 0; $i < count($people); ++$i) {
    $people[$i]['salt'] = mt_rand(000000, 999999);
}

$arr = array(1, 2, 3, 4);
foreach ($arr as &$value) {
    $value = $value * 2;
}
// $arr is now array(2, 4, 6, 8)
unset($value); // break the reference with the last element
?>
```

PHP Arrays

Arrays in PHP are versatile data structures that can hold multiple values under a single variable name.

Types of Arrays in PHP

PHP supports three types of arrays:

1. **Indexed Arrays**: Arrays with numeric keys
2. **Associative Arrays**: Arrays with named keys
3. **Multidimensional Arrays**: Arrays containing one or more arrays

Indexed Arrays

```php
<?php
$fruits = array("Apple", "Banana", "Cherry");
```

```php
// or
$fruits = ["Apple", "Banana", "Cherry"];

echo $fruits[0]; // Outputs: Apple
echo count($fruits); // Outputs: 3
?>
```

Associative Arrays

```php
<?php
$age = array("Ali"=>35, "Omar"=>37, "Ahmed"=>43);
// or
$age = ["Ali"=>35, "Omar"=>37, "Ahmed"=>43];

echo $age['Ali']; // Outputs: 35
?>
```

Multidimensional Arrays

```php
<?php
$cars = array (
  array("Volvo",22,18),
  array("BMW",15,13),
  array("Toyota",5,2)
);

echo $cars[0][0]; // Outputs: Volvo
?>
```

Array Functions

PHP provides numerous built-in functions to work with arrays:

- `array_push()`: Adds one or more elements to the end of an array
- `array_pop()`: Removes the last element from an array
- `array_shift()`: Removes the first element from an array
- `array_unshift()`: Adds one or more elements to the beginning of an array
- `sort()`: Sorts an array in ascending order
- `rsort()`: Sorts an array in descending order

- `array_merge()`: Merges one or more arrays

Example: Array Functions

```php
<?php
$fruits = ["Apple", "Banana"];
array_push($fruits, "Cherry");
print_r($fruits);
// Output: Array ( [0] => Apple [1] => Banana [2] => Cherry )

$last = array_pop($fruits);
echo $last; // Outputs: Cherry

sort($fruits);
print_r($fruits);
// Output: Array ( [0] => Apple [1] => Banana )
?>
```

PHP Regular Expressions

> ***Note:*** Check tutorial 3 for examples.

Basic Syntax:

In PHP, regular expressions are typically enclosed in delimiters. The most common delimiters are forward slashes '/' or curly braces '{}' or pipes '|':

```php
$pattern = '/pattern/';
$pattern = '{pattern}';
$pattern = '|pattern|';
```

Matching Functions:

PHP provides several functions for working with regex:

- preg_match(): Checks if a pattern matches a string
- preg_match_all(): Finds all occurrences of a pattern in a string
- preg_replace(): Replaces matches with a specified string
- preg_split(): Splits a string by a regular expression

Note: In this course only `preg_match()` is required. And you are expected to use it for input validation.

Simple Pattern Matching:

```php
$text = "Hello, World!";
$pattern = '/Hello/';
if (preg_match($pattern, $text)) {
   echo "Match found!";
}
```

Character Classes:

- `[abc]`: Matches any single character in the set
- `[^abc]`: Matches any single character not in the set
- `[a-z]`: Matches any single character in the range

Metacharacters:

- `.` (dot): Matches any single character except newline
- `\d`: Matches any digit (0-9)
- `\w`: Matches any word character (a-z, A-Z, 0-9, _)
- `\s`: Matches any whitespace character

Quantifiers:

- `*`: Matches 0 or more occurrences
- `+`: Matches 1 or more occurrences
- `?`: Matches 0 or 1 occurrence
- `{n}`: Matches exactly n occurrences
- `{n,}`: Matches n or more occurrences
- `{n,m}`: Matches between n and m occurrences

Anchors:

- `^`: Matches the start of a string
- `$`: Matches the end of a string

Modifiers:

Add modifiers after the closing delimiter: - `i`: Case-insensitive matching - `m`: Multi-line mode - `s`: Dot matches newline

```php
$pattern = '/pattern/i'; // Case-insensitive
```

Capturing Groups:

Use parentheses () to create capturing groups:

```php
$text = "Omar Ali";
$pattern = '/(\w+)\s(\w+)/';
preg_match($pattern, $text, $matches);
print_r($matches);
```

Object-Oriented PHP

Classes

```php
<?php

class MyClass {
    public const MY_CONSTANT = 'constant value';

    public $publicProperty;
    protected $protectedProperty;
    private $privateProperty;

    public function __construct($publicValue, $protectedValue, $privateValue) {
        $this->publicProperty = $publicValue;
        $this->protectedProperty = $protectedValue;
        $this->privateProperty = $privateValue;
    }

    public function myMethod() {
        return $this->publicProperty;
    }
}
```

```
$instance = new MyClass('public value', 'protected value', 'private value');
echo $instance->publicProperty; // Outputs: public value
echo $instance->myMethod(); // Outputs: public value
echo MyClass::MY_CONSTANT; // Outputs: constant value
?>
```

Inheritance

```
class BaseClass {
    public $baseProperty;

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

    public function baseMethod() {
        return $this->baseProperty;
    }
}

class DerivedClass extends BaseClass {
    public $derivedProperty;

    public function __construct($baseValue, $derivedValue) {
        parent::__construct($baseValue);
        $this->derivedProperty = $derivedValue;
    }

    public function derivedMethod() {
        return $this->derivedProperty;
    }
}
?>
```

Abstract Classes

```
abstract class AbstractClass {
    abstract protected function abstractMethod();

    public function concreteMethod() {
        return "This is a concrete method.";
    }
}

class ConcreteClass extends AbstractClass {
    protected function abstractMethod() {
        return "This is an implementation of the abstract method.";
    }
}
?>
```

Interfaces

```
interface MyInterface {
    public function interfaceMethod();
}

class ImplementingClass implements MyInterface {
    public function interfaceMethod() {
        return "Implemented method";
    }
}
?>
```

Creating MySQL Databases

Using the Terminal

To create a MySQL database from the terminal, follow these steps:

1. **Open your terminal** and log in to the MySQL server: `sh    mysql -u root -p` Enter your MySQL root password when prompted.

2. **Create a new database**: `sql     CREATE DATABASE my_database;` Replace `my_database` with your desired database name.

3. **Verify the database creation**: `sql     SHOW DATABASES;` You should see `my_database` listed among the databases.

Using phpMyAdmin

To create a MySQL database using phpMyAdmin, follow these steps:

1. **Log in to phpMyAdmin** using your web browser.

2. **Click on the "Databases" tab** at the top of the page.

3. **Enter the name of the new database** in the "Create database" field.

4. **Select the collation** (optional) and click the "Create" button.

Your new database should now appear in the list of databases.

PHP PDO

PHP PDO (PHP Data Objects) is a lightweight, consistent interface for accessing databases in PHP. It's an abstraction layer that provides a uniform method of interaction with multiple databases. Some features of PDO:

1. Database agnostic: PDO works with different database systems (MySQL, PostgreSQL, SQLite, etc.) using the same code.

2. Prepared statements: It supports prepared statements, which help prevent SQL injection attacks.

3. Error handling: PDO uses exceptions for error handling, making it easier to catch and manage database-related errors.

4. Object-oriented: It provides an object-oriented interface for database operations.

5. Security: PDO offers better security features compared to older MySQL extensions.

6. Consistent API: It provides a consistent naming convention for all database functions.

PDO is generally considered a more modern and secure way to interact with databases in PHP compared to older methods like the mysql_ functions.

Basic MySQL Database Connection

```php
<?php
$pdo = new PDO("mysql:host=localhost;port=3307;dbname=testdb", 'my_user', 'my_password');

$result = $pdo->query("SELECT email FROM user");
foreach ($result as $row) {
    echo $row['email'] . '<br>';
}
?>
```

DSN

A DSN, or Data Source Name, is a string that contains the information required to connect to a specific database. In the context of PDO (PHP Data Objects), the DSN is used to specify the database driver to use and provide the necessary details for establishing a connection. Here's a breakdown of what a DSN typically includes:

1. Driver prefix: Indicates which PDO driver to use (e.g., mysql, pgsql, sqlite).
2. Host: The server where the database is located.
3. Port: The port number for the database server (if different from the default).
4. Database name: The name of the specific database to connect to.
5. Additional parameters: Can include charset, SSL settings, etc.

The format of a DSN can vary slightly depending on the database driver being used. Here are a few examples:

1. MySQL:

   ```
   mysql:host=localhost;dbname=mydatabase;charset=utf8mb4
   ```

2. PostgreSQL:

   ```
   pgsql:host=localhost;port=5432;dbname=mydatabase
   ```

3. SQLite:

   ```
   sqlite:/path/to/database.sqlite
   ```

When creating a PDO connection, the DSN is typically used as the first parameter in the PDO constructor:

```php
$pdo = new PDO($dsn, $username, $password);
```

PDO Prepared Statements

Example 1

This code prepares and executes a SQL query to select an email from the user table where the email matches the provided email address. This query can be used to check if an email already exists in a database:

```
$sql = "SELECT email FROM user WHERE email = ?";
$statement = $db->prepare($sql);
$statement->execute([$email]);
$user = $statement->fetch();
```

SQL Query Preparation:

```
$sql = "SELECT email FROM user WHERE email = ?";
```

- This line defines an SQL query to select the `email` field from the `user` table where the `email` matches a specific value.
- The `?` is a placeholder for a parameter that will be bound later, which helps prevent SQL injection attacks.

Preparing the Statement:

```
$statement = $db->prepare($sql);
```

- Here, the `$db->prepare($sql)` method prepares the SQL query for execution. `$db` is assumed to be a PDO (PHP Data Objects) instance, which provides a secure way to interact with the database.

Executing the Statement:

```
$statement->execute([$email]);
```

- The `execute` method runs the prepared statement.
- The array `[$email]` provides the value for the placeholder `?` in the SQL query. This binds the `$email` variable to the placeholder, ensuring safe and correct execution.

Fetching the Result:

```
$user = $statement->fetch();
```

- The `fetch` method retrieves the next row from the result set.
- In this case, it fetches the row where the `email` matches the provided `$email` value.
- If a matching email is found, `$user` will contain the result. If no match is found, `$user` will be `false`.

Example 2

This PHP code securely inserts a new user's email and hashed password into the `user` table in a database using prepared statements.

```
$statement = $db->prepare("INSERT INTO user (email, password) VALUES (?, ?)");
$statement->execute([$email, $hashed_password]);
```

Prepare an SQL Statement:

```
$statement = $db->prepare("INSERT INTO user (email, password) VALUES (?, ?)");
```

- This line prepares an SQL `INSERT` statement to add a new row to the `user` table with two columns: `email` and `password`.
- The `?` placeholders are used for parameterized query execution, which helps prevent SQL injection.

Execute the Prepared Statement:

```
$statement->execute([$email, $hashed_password]);
```

- This line executes the prepared statement with the provided parameters.
- `$email` and `$hashed_password` are the actual values that will be inserted into the `email` and `password` columns of the `user` table, respectively.

Example 3

This code prepares and executes a SQL query to fetch a user record from the user table based on the provided email. It then retrieves the result as an associative array. You can use this code to check if the user name and password are correct:

```
$statement = $db->prepare("SELECT * FROM user WHERE email = :email");
$statement->bindParam(':email', $email);
$statement->execute();
$user = $statement->fetch(PDO::FETCH_ASSOC);
```

Prepare an SQL Statement:

```
$statement = $db->prepare("SELECT * FROM user WHERE email = :email");
```

- `$db`: This variable represents a PDO (PHP Data Objects) instance, which is used to interact with the database.
- `prepare()`: This method prepares an SQL statement for execution. The SQL query here is `SELECT * FROM user WHERE email = :email`.

- `:email`: This is a named placeholder used in the SQL query. It will be replaced with the actual email value later.

Binding Paramters

```
$statement->bindParam(':email', $email);
```

- `bindParam()`: This method binds a PHP variable to a named placeholder in the SQL statement. Here, the placeholder `:email` is bound to the PHP variable `$email`.
- `$email`: This variable should contain the email address provided by the user attempting to log in.

Execute the Prepared Statement:

```
$statement->execute();
```

- `execute()`: This method executes the prepared statement. At this point, the SQL query is sent to the database with the bound email value.

Fetching the Result:

```
$user = $statement->fetch(PDO::FETCH_ASSOC);
```

- `fetch()`: This method retrieves the next row from the result set. The `PDO::FETCH_ASSOC` parameter specifies that the row should be returned as an associative array, where the column names are the keys.
- `$user`: This variable will hold the fetched user data if a user with the provided email exists in the database. If no such user exists, `$user` will be `false`.

Uploading Files

The following is an example of uploading a file to a server by using HTML and PHP.

Note: This code is insecure, in production you need to validate the request content before you use the file.

```
<?php
if ($_SERVER["REQUEST_METHOD"] == "POST") {
    $target_dir = "uploads/";
    $target_file = $target_dir . basename($_FILES["fileToUpload"]["name"]);

    if (move_uploaded_file($_FILES["fileToUpload"]["tmp_name"], $target_file)) {
        echo "The file ". basename( $_FILES["fileToUpload"]["name"]). " has been uploaded.";
    } else {
        echo "Sorry, there was an error uploading your file.";
    }
}
?>

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Image Upload Example</title>
</head>
<body>
    <h2>Upload an Image</h2>
    <form action="<?php echo $_SERVER['PHP_SELF']; ?>" method="post" enctype="multipart/form-data">
        Select image to upload:
        <input type="file" name="fileToUpload" id="fileToUpload">
        <input type="submit" value="Upload Image" name="submit">
    </form>
</body>
</html>
```

Explination:

1. `$target_dir = "uploads/";`
    - This line sets the directory where uploaded files will be stored.
    - "uploads/" is a relative path, meaning it's a subdirectory in the same directory as the PHP script.
    - The trailing slash (/) is important to ensure the proper formation of the full path.
2. `$target_file = $target_dir . basename($_FILES["fileToUpload"]["name"]);`
    - This line constructs the full path for the uploaded file.
    - `$target_dir` is the directory we defined earlier ("uploads/").
    - `$_FILES["fileToUpload"]["name"]` is the original name of the uploaded file.
    - `basename()` is a PHP function that extracts the filename from a path. It's used here as a simple security measure to prevent directory traversal attacks.
    - The `.` operator concatenates the directory path with the filename.
3. `if (move_uploaded_file($_FILES["fileToUpload"]["tmp_name"], $target_file)) {`
    - This line attempts to move the uploaded file from its temporary location to the target location.
    - `move_uploaded_file()` is a PHP function specifically designed for handling uploaded files securely.

   - `$_FILES["fileToUpload"]["tmp_name"]` is the temporary filename of the uploaded file on the server.
   - `$target_file` is the destination path we constructed in step 2.
   - The function returns true if the move was successful, false otherwise.
   - The `if` statement allows us to perform different actions based on whether the move was successful or not.

4. `action="<?php echo $_SERVER['PHP_SELF'] ?>`:
   - The action attribute specifies where to send the form-data when the form is submitted.
   - $_SERVER[‘PHP_SELF’] is a PHP superglobal variable that returns the filename of the currently executing script.
   - Using this means the form will submit to the same page it’s on.
5. `enctype="multipart/form-data"`:
   - This attribute is crucial for file uploads.
   - It specifies how the form-data should be encoded when submitted to the server.
   - “multipart/form-data” is required when the form includes file upload controls.

Some additional notes: - “fileToUpload” in `$_FILES["fileToUpload"]` corresponds to the `name` attribute of the file input field in the HTML form. - This code doesn’t include any validation or security checks, which would be crucial in a real-world scenario. - The `move_uploaded_file()` function automatically handles some security concerns, such as ensuring that the file is indeed an uploaded file.

$_FILES

The `$_FILES` array in PHP contains information about uploaded files. It’s a superglobal array that’s automatically created when a file is uploaded through an HTML form. Here’s a typical structure of the `$_FILES` array for a single file upload:

```php
$_FILES['input_name'] = array(
    'name'      => 'filename.jpg',
    'type'      => 'image/jpeg',
    'size'      => 12345,
    'tmp_name'  => '/tmp/php/php1h4j1o',
    'error'     => 0
);
```

1. ‘name’: The original name of the file on the client machine.

2. ‘type’: The MIME type of the file, if the browser provided this information. An example would be “image/jpeg” for JPEG images, “application/pdf” for PDF files, etc.

3. ‘size’: The size of the uploaded file in bytes.
4. ‘tmp_name’: The temporary filename of the file in which the uploaded file was stored on the server.
5. ‘error’: The error code associated with this file upload. 0 means no error.

For multiple file uploads, the structure would be slightly different:

```
$_FILES['input_name'] = array(
    'name'      => array('filename1.jpg', 'filename2.png'),
    'type'      => array('image/jpeg', 'image/png'),
    'size'      => array(12345, 67890),
    'tmp_name'  => array('/tmp/php/php1h4j1o', '/tmp/php/php6hst32'),
    'error'     => array(0, 0)
);
```

The error codes you might encounter are:

- 0: `UPLOAD_ERR_OK` (No error)
- 1: `UPLOAD_ERR_INI_SIZE` (Exceeds upload_max_filesize in php.ini)
- 2: `UPLOAD_ERR_FORM_SIZE` (Exceeds MAX_FILE_SIZE in HTML form)
- 3: `UPLOAD_ERR_PARTIAL` (File was only partially uploaded)
- 4: `UPLOAD_ERR_NO_FILE` (No file was uploaded)
- 6: `UPLOAD_ERR_NO_TMP_DIR` (Missing a temporary folder)
- 7: `UPLOAD_ERR_CANT_WRITE` (Failed to write file to disk)
- 8: `UPLOAD_ERR_EXTENSION` (A PHP extension stopped the file upload)

When working with file uploads, it’s important to always check the ‘error’ value before processing the file to ensure the upload was successful.

Upload Validation

To validate an updated file, you can do the following:

1. Validate file types:

```
$allowed_types = ['image/jpeg', 'image/png', 'image/gif'];
$file_type = $_FILES['fileToUpload']['type'];

if (!in_array($file_type, $allowed_types)) {
    exit("Error: Only JPG, PNG, and GIF files are allowed.");
}
```

Explanation: This code defines an array of allowed MIME types. It then checks if the uploaded file's type is in this array. If not, it stops execution with an error message.

2. Check file size:

```
$max_size = 5 * 1024 * 1024; // 5 MB in bytes
if ($_FILES['fileToUpload']['size'] > $max_size) {
    exit("Error: File size cannot exceed 5 MB.");
}
```

Explanation: This sets a maximum file size (5 MB in this case) and checks if the uploaded file exceeds this limit. If it does, it stops execution with an error message.

3. Generate unique filenames:

```
$file_name = basename($_FILES["fileToUpload"]["name"]);
$file_ext = pathinfo($file_name, PATHINFO_EXTENSION);
$unique_name = uniqid() . '.' . $file_ext;
$target_file = $target_dir . $unique_name;
```

Explanation: This generates a unique filename using `uniqid()` function and appends the original file extension. This ensures that each upload gets a unique name, preventing overwrites.

4. Implement user authentication and authorization:

```
session_start();
if (!isset($_SESSION['user_id']) || $_SESSION['user_role'] !== 'admin') {
    exit("Error: You don't have permission to upload files.");
}
```

Explanation: This assumes you have a login system that sets session variables. It checks if the user is logged in and has the correct role before allowing the upload.

5. Sanitize user inputs:

```
 // Sanitize and get file information
$file_name = htmlspecialchars(basename($_FILES["fileToUpload"]["name"]));
```

Explanation: This uses PHP's built-in function to sanitize the filename, removing or encoding potentially harmful characters.

6. Set appropriate file permissions:

```
if (move_uploaded_file($_FILES["fileToUpload"]["tmp_name"], $target_file)) {
    chmod($target_file, 0644);
    echo "File uploaded successfully.";
}
```

Explanation: After moving the uploaded file, this sets its permissions to 0644 (owner can read and write, others can only read). Adjust as needed for your security requirements.

7. Use a more secure upload directory outside the web root:

```
$target_dir = "/var/www/uploads/";
```

Explanation: This sets the upload directory to a location outside the web root. Users can't directly access files in this directory through a URL, adding an extra layer of security.

Putting it all to gether:

```
<?php
// File upload handling
if ($_SERVER["REQUEST_METHOD"] == "POST") {
    // Define constants
    $target_dir = "/var/www/uploads/";
    $allowed_types = ['image/jpeg', 'image/png', 'image/gif'];
    $max_size = 5 * 1024 * 1024; // 5 MB

    // Validate file existence
    if (!isset($_FILES['fileToUpload']) || $_FILES['fileToUpload']['error'] === UPLOAD_ERR_NO_FILE) {
        exit("Error: No file was uploaded.");
    }

    // Validate file type
    $file_type = $_FILES['fileToUpload']['type'];
    if (!in_array($file_type, $allowed_types)) {
        exit("Error: Only JPG, PNG, and GIF files are allowed.");
    }

    // Check file size
    if ($_FILES['fileToUpload']['size'] > $max_size) {
        exit("Error: File size cannot exceed 5 MB.");
    }

    // Sanitize filename
    $original_filename = htmlspecialchars(basename($_FILES["fileToUpload"]["name"]), ENT_QUOTES, 'UTF-8
    $file_ext = strtolower(pathinfo($original_filename, PATHINFO_EXTENSION));

    // Generate a unique name and check if it already exists
    do {
        $unique_name = uniqid() . '.' . $file_ext;
        $target_file = $target_dir . $unique_name;
    } while (file_exists($target_file));

    // Attempt to upload file
    if (move_uploaded_file($_FILES["fileToUpload"]["tmp_name"], $target_file)) {
        // Set appropriate permissions
        chmod($target_file, 0644);

        // Log the upload (you should implement proper logging)
        error_log("File uploaded: $unique_name (Original: $original_filename)");

        echo "File uploaded successfully.";
    } else {
        echo "Sorry, there was an error uploading your file.";
    }
} else {
    echo "No file upload attempt detected.";
}
?>
```