Contents

Module 2 – Advanced PHP Exercises: OOP Concepts

Inheritance, Polymorphism, and Abstraction

Theory Exercise:

- **Inheritance** allows a class to use methods and properties of another class. A subclass (child) inherits all methods and properties from the parent class but can also add new ones or override existing ones.
- **Polymorphism** allows methods to be used interchangeably, meaning that a child class can override a method from the parent class while maintaining the same interface.
- **Abstraction** hides complex implementation details and exposes only the essential features of an object.

Practical Exercise:

- **Encapsulation**: Create a class with private and public properties and methods.

```
class Person {
    private $name;
    private $age;

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

    public function setAge($age) {
        $this->age = $age;
    }

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

    public function getAge() {
        return $this->age;
    }
}

$person = new Person();
$person->setName("John");
$person->setAge(30);
echo $person->getName() . " is " . $person->getAge() . " years old.";
```

OOPs Concepts

Class Structure

Theory Exercise:

- A **class** in PHP is a blueprint for creating objects, which define properties (variables) and methods (functions). A class can have:
    - **Properties**: Variables that store the data.
    - **Methods**: Functions that perform actions using the properties.

Practical Exercise:

- Create a `Car` class.

```
class Car {
    public $make;
    public $model;
    public $year;

    public function displayDetails() {
        echo "Car: $this->make $this->model, Year: $this->year";
    }
}

$car = new Car();
$car->make = "Toyota";
$car->model = "Corolla";
$car->year = 2020;
$car->displayDetails();  // Output: Car: Toyota Corolla, Year: 2020
```

Object

Theory Exercise:

- An **object** in OOP is an instance of a class. Objects are created by instantiating a class using the `new` keyword.

Practical Exercise:

- Instantiate multiple `Car` objects and access their properties and methods.

```
$car1 = new Car();
$car1->make = "Honda";
$car1->model = "Civic";
$car1->year = 2021;

$car2 = new Car();
$car2->make = "Ford";
$car2->model = "Focus";
$car2->year = 2022;
```

```
$car1->displayDetails();  // Output: Car: Honda Civic, Year: 2021
$car2->displayDetails();  // Output: Car: Ford Focus, Year: 2022
```

Extends (Inheritance)

Theory Exercise:

- **Inheritance** allows a class to inherit properties and methods from another class. In PHP, inheritance is done using the `extends` keyword.

Practical Exercise:

- Create a `Vehicle` class and extend it with a `Car` class.

```
class Vehicle {
    public $make;
    public $model;

    public function displayDetails() {
        echo "Vehicle: $this->make $this->model";
    }
}

class Car extends Vehicle {
    public $year;

    public function displayCarDetails() {
        echo "Car: $this->make $this->model, Year: $this->year";
    }
}

$car = new Car();
$car->make = "Toyota";
$car->model = "Camry";
$car->year = 2021;
$car->displayCarDetails();  // Output: Car: Toyota Camry, Year: 2021
```

Overloading

Theory Exercise:

- **Method Overloading** is a feature in PHP that allows you to define multiple methods with the same name but with different arguments. PHP doesn't support method overloading natively, but it provides `__call()` to achieve a similar effect.

Practical Exercise:

- Create a class that demonstrates method overloading.

```
class Calculator {
    public function __call($name, $arguments) {
        if ($name == 'add') {
            return array_sum($arguments);
        }
    }
}

$calc = new Calculator();
echo $calc->add(2, 3);  // Output: 5
echo $calc->add(1, 2, 3, 4);  // Output: 10
```

Abstraction & Interface

Theory Exercise:

- **Abstraction** is the concept of hiding complex details and showing only necessary functionality. **Interfaces** define a contract that classes must follow by implementing the methods specified in the interface.

Practical Exercise:

- Define an interface `VehicleInterface` and implement it in multiple classes.

```
interface VehicleInterface {
    public function start();
    public function stop();
}

class Car implements VehicleInterface {
    public function start() {
        echo "Car started";
    }

    public function stop() {
        echo "Car stopped";
    }
}

$car = new Car();
$car->start();  // Output: Car started
```

```
$car->stop();   // Output: Car stopped
```

Constructor

Theory Exercise:

- A **constructor** in PHP is a special method used to initialize an object's properties when it is created. It is called automatically when an object is instantiated.

Practical Exercise:

- Create a class with a constructor that initializes properties.

```
class Car {
    public $make;
    public $model;

    public function __construct($make, $model) {
        $this->make = $make;
        $this->model = $model;
    }

    public function displayDetails() {
        echo "Car: $this->make $this->model";
    }
}

$car = new Car("Toyota", "Corolla");
$car->displayDetails();  // Output: Car: Toyota Corolla
```

Destructor

Theory Exercise:

- A **destructor** is a special method that is called when an object is destroyed, typically for cleanup tasks.

Practical Exercise:

- Write a class that implements a destructor.

```
class Person {
    public function __construct() {
        echo "Object created\n";
    }

    public function __destruct() {
        echo "Object destroyed\n";
    }
}

$person = new Person();
```

```
unset($person);  // Output: Object created
                 // Output: Object destroyed
```

Magic Methods

Theory Exercise:

- **Magic methods** in PHP are special methods with names that begin with double underscores, such as `__construct`, `__call`, `__get`, `__set`, etc.

Practical Exercise:

- Create a class that uses magic methods.

```
class Magic {
    private $data = [];

    public function __get($name) {
        return isset($this->data[$name]) ? $this->data[$name] : null;
    }

    public function __set($name, $value) {
        $this->data[$name] = $value;
    }
}

$obj = new Magic();
$obj->name = "John";
echo $obj->name;  // Output: John
```

Scope Resolution (Static Properties and Methods)

Theory Exercise:

- The **scope resolution operator (`::`)** is used to access static properties and methods, or to access constants within a class.

Practical Exercise:

Create a class with static properties and methods.

```
class MyClass {
    public static $count = 0;

    public static function incrementCount() {
        self::$count++;
    }
}

MyClass::incrementCount();
echo MyClass::$count;  // Output: 1
```

Traits

Theory Exercise:

- **Traits** are a way to include reusable code in classes. A trait can contain methods that can be included in multiple classes.

Practical Exercise:

Create and use traits.

```
trait Logger {
    public function log($message) {
        echo "Log: $message\n";
    }
}

class User {
    use Logger;
}

$user = new User();
$user->log("User created");  // Output: Log: User created
```

Visibility (Access Modifiers)

Theory Exercise:

- **Visibility** refers to the accessibility of properties and methods in a class. PHP supports three types:
    - `public`: Accessible anywhere.
    - `protected`: Accessible within the class and its subclasses.
    - `private`: Accessible only within the class.

Practical Exercise:

- Show visibility with properties and methods.

```
class MyClass {
    public $publicVar = "I am public";
    protected $protectedVar = "I am protected";
    private $privateVar = "I am private";

    public function showVars() {
        echo $this->publicVar . "\n";
        echo $this->protectedVar . "\n";
        echo $this->privateVar . "\n";
    }
}
```

```
$obj = new MyClass();
$obj->showVars();  // Output: I am public I am protected I am private
```

Type Hinting

Theory Exercise:

- **Type hinting** allows you to define the expected data types for function/method parameters.

Practical Exercise:

- Demonstrate type hinting.

```
class Math {
    public function add(int $a, int $b): int {
        return $a + $b;
    }
}

$math = new Math();
echo $math->add(2, 3);  // Output: 5
```

Final Keyword

Theory Exercise:

- The **final** keyword prevents a class from being extended or a method from being overridden.

Practical Exercise:

- Attempt to extend a final class.

```
final class MyClass {
    public function sayHello() {
        echo "Hello";
    }
}

class NewClass extends MyClass {  // Error: Cannot extend final class
}
```

Email Security Function

Theory Exercise:

- **Email security** involves sanitizing and validating email input to prevent attacks such as header injection and SQL injection.

Practical Exercise:

- Write a function to sanitize and validate an email address.

```
function validateEmail($email) {
    return filter_var($email, FILTER_VALIDATE_EMAIL) ? true : false;
}

$email = "test@example.com";
if (validateEmail($email)) {
    echo "Valid email address!";
} else {
    echo "Invalid email address.";
}
```

File Handling

Theory Exercise:

- File handling in PHP includes functions to open, read, write, and close files.

Practical Exercise:

- Create a script to read from a file.

```
$file = fopen("example.txt", "r");
while (($line = fgets($file)) !== false) {
    echo $line;
}
fclose($file);
```

Connection with MySQL Database

Theory Exercise:

- To connect PHP to a MySQL database, you can use either `mysqli` or `PDO`. Both methods provide ways to connect and interact with databases securely.

Practical Exercise:

- Connect to a MySQL database using `mysqli`.

```
$conn = new mysqli("localhost", "username", "password", "database");

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

echo "Connected successfully";
$conn->close();
```

THEORY EXERCISE:

Definition of SQL Injection:

SQL Injection is a code injection technique that exploits a vulnerability in an application's software by inserting or "injecting" malicious SQL code into a query. It allows attackers to manipulate the application's SQL query execution to access or alter data in the database in unauthorized ways.

Implications on Security:

- **Unauthorized Data Access:** Attackers can read sensitive data such as user credentials, personal information, and more.
- **Data Modification:** Attackers can update or delete records in the database, leading to data corruption or loss.
- **Authentication Bypass:** Attackers can bypass authentication mechanisms to impersonate other users or administrators.
- **Denial of Service (DoS):** Attackers can cause the database to crash by issuing complex or harmful queries.
- **Exploiting Privileges:** Attackers can escalate their privileges to execute arbitrary system commands.

PRACTICAL EXERCISE:

Vulnerable SQL Query:

```
$username = $_POST['username'];
$password = $_POST['password'];
$query = "SELECT * FROM users WHERE username = '$username' AND password =
'$password'";
$result = mysqli_query($conn, $query);
```

- **Exploit Example:**
  - If the attacker inputs `' OR '1'='1` for both the username and password, the SQL query becomes:
  - `SELECT * FROM users WHERE username = '' OR '1'='1' AND password = '' OR '1'='1'`
  - This query would return all users in the database, bypassing the login process.

Preventing SQL Injection with Prepared Statements:

```
$stmt = $conn->prepare("SELECT * FROM users WHERE username = ? AND
password = ?");
$stmt->bind_param("ss", $username, $password);
$stmt->execute();
$result = $stmt->get_result();
```

- Here, the query is prepared, and the user input is safely bound using `bind_param()` to prevent SQL injection.

Exception Handling with Try-Catch for Database Connection and Queries

PRACTICAL EXERCISE:

```
try {
    $conn = new mysqli($servername, $username, $password, $dbname);

    if ($conn->connect_error) {
        throw new Exception("Connection failed: " . $conn->connect_error);
    }

    // Example query execution
    $sql = "SELECT * FROM users";
    $result = $conn->query($sql);

    if (!$result) {
        throw new Exception("Query failed: " . $conn->error);
    }
} catch (Exception $e) {
    echo "Error: " . $e->getMessage();
} finally {
    $conn->close();
}
```

- This handles database connection and query exceptions by using `try-catch` blocks and ensures the connection is closed in the `finally` block.

Server-Side Validation using Regular Expressions

PRACTICAL EXERCISE:

```
if ($_SERVER['REQUEST_METHOD'] == 'POST') {
    $email = $_POST['email'];
    $password = $_POST['password'];

    // Email validation
    if (!preg_match("/^[a-zA-Z0-9._%-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}$/",
$email)) {
        echo "Invalid email format";
    }

    // Password validation (minimum 8 characters, at least one number, one
uppercase, and one lowercase)
    if (!preg_match("/^(?=.*[A-Za-z])(?=.*\d)[A-Za-z\d]{8,}$/", $password)) {
        echo "Password must be at least 8 characters long and contain at
least one number.";
    }
}
```

Send Mail While Registration

PRACTICAL EXERCISE:

```
if ($_SERVER['REQUEST_METHOD'] == 'POST') {
    $email = $_POST['email'];
    // Registration logic here

    // Send confirmation email
    $to = $email;
    $subject = "Registration Confirmation";
    $message = "Thank you for registering!";
    $headers = "From: no-reply@example.com";

    if (mail($to, $subject, $message, $headers)) {
        echo "Confirmation email sent!";
    } else {
        echo "Failed to send confirmation email.";
    }
}
```

Sessions and Cookies

THEORY EXERCISE:

Sessions:

- Store user data on the server (e.g., user ID, preferences) and are typically used to track user activity across multiple pages.
- Session data is stored in temporary files on the server.
- Sessions expire after a certain period or when the user logs out.

Cookies:

- Small pieces of data stored on the user's browser (e.g., login info, preferences).
- Cookies have an expiration date and can persist across sessions.
- Cookies can be read and modified by both the server and client.

PRACTICAL EXERCISE:

```
// Start a session and store user data
session_start();
$_SESSION['username'] = 'john_doe';

// Retrieve session data on another page
echo "Welcome, " . $_SESSION['username'];

// Set and retrieve cookies
setcookie("user", "john_doe", time() + 3600, "/"); // Cookie expires in 1
hour
echo $_COOKIE['user'];
```

File Upload

THEORY EXERCISE:

- **File Upload Functionality in PHP:**
  - PHP provides the `$_FILES` superglobal to handle file uploads.
  - It is essential to validate file types, check for file size limits, and store the file securely.
- **Security Implications:**
  - Improper validation can allow attackers to upload malicious files (e.g., scripts).
  - Always validate the file type and size before uploading.
  - Store uploaded files outside the web root or ensure proper permissions are set.

PRACTICAL EXERCISE:

```
if ($_SERVER['REQUEST_METHOD'] == 'POST' && isset($_FILES['file'])) {
    $file = $_FILES['file'];
    $allowed_types = ['image/jpeg', 'image/png'];

    if (in_array($file['type'], $allowed_types)) {
        move_uploaded_file($file['tmp_name'], 'uploads/' . $file['name']);
        echo "File uploaded successfully!";
    } else {
        echo "Invalid file type!";
    }
}
```

PHP with MVC Architecture

PRACTICAL EXERCISE:

1. **Implementing a Basic CRUD Application with MVC Architecture:**
   - **Model:** Handles data-related logic (e.g., interacting with the database).
   - **View:** Displays data to the user.
   - **Controller:** Processes user input and updates the model and view accordingly.

   Example:

   - **Model:** `UserModel.php`
   - **View:** `userView.php`
   - **Controller:** `UserController.php`

Extra Practice for Grade A

1. Class Hierarchy for an E-Commerce System

```
class Product {
    private $name;
    private $price;

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

    public function getDetails() {
        return "Product: " . $this->name . ", Price: $" . $this->price;
    }
}

class Category {
    private $name;

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

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

class Order {
    private $products = [];
    private $totalPrice = 0;

    public function addProduct(Product $product) {
        $this->products[] = $product;
        $this->totalPrice += $product->getPrice();
    }

    public function getOrderDetails() {
        return "Order Total: $" . $this->totalPrice;
    }
}
```

---

2. Class for Book with Discount

```
class Book {
    private $title;
    private $author;
    private $price;

    public function __construct($title, $author, $price) {
        $this->title = $title;
        $this->author = $author;
        $this->price = $price;
    }

    public function applyDiscount($percentage) {
        $this->price -= ($this->price * $percentage / 100);
        return $this->price;
    }

    public function getDetails() {
        return "Book: $this->title by $this->author, Price: $" . $this-
>price;
    }
}
```

MVC Architecture

E-Commerce System 18. Extend the Simple MVC Application to Include a Model for Managing User Profiles, a View for Displaying User Details, and a Controller for Handling User Actions

Practical Exercise:

Model (UserProfileModel.php):

```
class UserProfileModel {
    private $db;

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

    public function getUserById($userId) {
        $stmt = $this->db->prepare("SELECT * FROM users WHERE id = ?");
        $stmt->bind_param("i", $userId);
        $stmt->execute();
        $result = $stmt->get_result();
        return $result->fetch_assoc();
    }

    public function updateUser($userId, $name, $email) {
        $stmt = $this->db->prepare("UPDATE users SET name = ?, email = ?
WHERE id = ?");
        $stmt->bind_param("ssi", $name, $email, $userId);
        return $stmt->execute();
    }
}
```

View (userProfileView.php):

```
echo "<h2>User Profile</h2>";
echo "<p>Name: " . $user['name'] . "</p>";
echo "<p>Email: " . $user['email'] . "</p>";
echo "<a href='editProfile.php?id=" . $user['id'] . "'>Edit Profile</a>";
```

Controller (UserProfileController.php):

```
class UserProfileController {
    private $model;

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

    public function viewProfile($userId) {
        $user = $this->model->getUserById($userId);
        include 'views/userProfileView.php';
    }

    public function updateProfile($userId, $name, $email) {
        if ($this->model->updateUser($userId, $name, $email)) {
            header("Location: profile.php?id=$userId");
        } else {
            echo "Error updating profile.";
        }
    }
}
```

Main Execution:

```
$controller = new UserProfileController(new UserProfileModel($db));
$controller->viewProfile($userId);
```

Implementation of all the OOPs Concepts

19. Develop a Project That Simulates a Library System with Classes for User, Book, and Transaction, Applying All OOP Principles

Practical Exercise:

User Class:

```
class User {
    private $userId;
    private $name;

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

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

Book Class:

```
class Book {
    private $bookId;
    private $title;
    private $author;

    public function __construct($bookId, $title, $author) {
        $this->bookId = $bookId;
        $this->title = $title;
        $this->author = $author;
    }

    public function getBookDetails() {
        return "Title: $this->title, Author: $this->author";
    }
}
```

Transaction Class:

```
class Transaction {
    private $user;
    private $book;
    private $transactionDate;

    public function __construct(User $user, Book $book) {
        $this->user = $user;
        $this->book = $book;
        $this->transactionDate = date("Y-m-d");
    }

    public function getTransactionDetails() {
        return "User: {$this->user->getName()}, Book: {$this->book-
>getBookDetails()}, Date: $this->transactionDate";
    }
}
```

Example Usage:

```
$user = new User(1, 'John Doe');
$book = new Book(101, 'PHP for Beginners', 'Jane Smith');
$transaction = new Transaction($user, $book);

echo $transaction->getTransactionDetails();
```

Connection with MySQL Database

20. Write a Class Database That Handles Database Connections and Queries. Use This Class in Another Script to Fetch User Data from a Users Table

Practical Exercise:

1. **Database Class:**

```
class Database {
    private $conn;

    public function __construct($servername, $username, $password,
$dbname) {
        $this->conn = new mysqli($servername, $username, $password,
$dbname);

        if ($this->conn->connect_error) {
            throw new Exception("Connection failed: " . $this->conn-
>connect_error);
        }
    }

    public function fetchUserData($userId) {
        $stmt = $this->conn->prepare("SELECT * FROM users WHERE id = ?");
        $stmt->bind_param("i", $userId);
        $stmt->execute();
        $result = $stmt->get_result();
        return $result->fetch_assoc();
    }

    public function close() {
        $this->conn->close();
    }
}
```

2. **Fetching User Data (Example Usage):**

```
try {
    $db = new Database("localhost", "root", "", "test_db");
    $user = $db->fetchUserData(1);
    print_r($user);
} catch (Exception $e) {
    echo "Error: " . $e->getMessage();
}
```

SQL Injection

21. Create a Vulnerable PHP Script That Demonstrates SQL Injection. Then, Rewrite It Using Prepared Statements to Prevent SQL Injection Attacks

Practical Exercise:

Vulnerable Script (SQL Injection Example):

```
$username = $_POST['username'];
$password = $_POST['password'];

$query = "SELECT * FROM users WHERE username = '$username' AND password =
'$password'";
$result = mysqli_query($conn, $query);
```

- **SQL Injection Attack:** Using `' OR '1'='1` for username and password fields will bypass authentication.

Prepared Statements to Prevent SQL Injection:

```
$stmt = $conn->prepare("SELECT * FROM users WHERE username = ? AND
password = ?");
$stmt->bind_param("ss", $username, $password);
$stmt->execute();
$result = $stmt->get_result();
```

Handing with Try-Catch for Database Connection and Queries

22. Implement a Complete Registration Process with a Database Connection That Uses Try-Catch Blocks to Handle Exceptions for All Operations

Practical Exercise:

```
try {
    $conn = new mysqli("localhost", "root", "", "test_db");

    if ($conn->connect_error) {
        throw new Exception("Connection failed: " . $conn->connect_error);
    }
    if ($_SERVER['REQUEST_METHOD'] == 'POST') {
        $username = $_POST['username'];
        $password = password_hash($_POST['password'], PASSWORD_DEFAULT);

        $stmt = $conn->prepare("INSERT INTO users (username, password) VALUES
(?, ?)");
        $stmt->bind_param("ss", $username, $password);
        if ($stmt->execute()) {
            echo "Registration successful!";
        } else {
            throw new Exception("Registration failed.");
        }
    }
} catch (Exception $e) {
    echo "Error: " . $e->getMessage();
} finally {
    $conn->close();
}
```

Server-Side Validation while Registration using Regular Expressions

23. Server-Side Validation While Registration Using Regular Expressions, Providing Feedback on Any Validation Errors

Practical Exercise:

```
if ($_SERVER['REQUEST_METHOD'] == 'POST') {
    $username = $_POST['username'];
    $password = $_POST['password'];

    // Username validation (letters and numbers only)
    if (!preg_match("/^[a-zA-Z0-9]*$/", $username)) {
        echo "Invalid username. Only letters and numbers are allowed.";
    }

    // Password validation (at least 8 characters, one uppercase, one number)
    if (!preg_match("/^(?=.*[A-Z])(?=.*\d).{8,}$/", $password)) {
        echo "Password must be at least 8 characters long, with at least one
uppercase letter and one number.";
    }
}
```

Send Email While Registration

24. Extend the Registration Process to Send a Confirmation Email to the User After Successful Registration and Validate the Email Format

Practical Exercise:

```
if ($_SERVER['REQUEST_METHOD'] == 'POST') {
    $email = $_POST['email'];

    // Email validation
    if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
        echo "Invalid email format.";
    } else {
        // Send confirmation email
        $to = $email;
        $subject = "Registration Confirmation";
        $message = "Thank you for registering!";
        $headers = "From: no-reply@example.com";

        if (mail($to, $subject, $message, $headers)) {
            echo "Confirmation email sent!";
        } else {
            echo "Failed to send confirmation email.";
        }
    }
}
```

Sessions and Cookies

25. Implement a Login System That Uses Sessions to Keep Track of User Authentication and Demonstrates Cookie Usage for "Remember Me" Functionality

Practical Exercise:

```
// Start session
session_start();

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

    // Check credentials from the database
    // Assuming username and password are valid:
    $_SESSION['username'] = $username;

    // Remember Me functionality
    if (isset($_POST['remember'])) {
        setcookie("username", $username, time() + (86400 * 30), "/"); // 30
days
    }

    echo "Login successful!";
}
```

Upload Files

26. Create a File Upload Feature That Allows Users to Upload Images. Ensure That the Uploaded Images Are Checked for File Type and Size for Security

Practical Exercise:

```
if ($_SERVER['REQUEST_METHOD'] == 'POST' && isset($_FILES['image'])) {
    $image = $_FILES['image'];
    // Check file type
    $allowedTypes = ['image/jpeg', 'image/png'];
    if (!in_array($image['type'], $allowedTypes)) {
        echo "Invalid file type. Only JPG and PNG are allowed.";
    }

    // Check file size (max 2MB)
    if ($image['size'] > 2 * 1024 * 1024) {
        echo "File size exceeds 2MB.";
    }

    // Move uploaded file
    move_uploaded_file($image['tmp_name'], 'uploads/' . $image['name']);
    echo "File uploaded successfully!";
}
```

PHP with MVC Architecture

27. Build a Small Blog Application Using the MVC Architecture, Where Users Can Create, Read, Update, and Delete Posts

Practical Exercise:

In this example, we'll build a small blog application with the following basic operations using MVC architecture:

- **Model:** Handles the database interaction.
- **View:** Displays the data to the user.
- **Controller:** Manages the interaction between the model and the view.

---

1. Database Structure:

Let's assume we have a database called `blog_db` and a table called `posts`:

```
CREATE TABLE posts (
    id INT AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(255) NOT NULL,
    content TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

---

2. Model (PostModel.php):

The model interacts with the database to fetch and manipulate post data.

```
class PostModel {
    private $db;

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

    public function getAllPosts() {
        $query = "SELECT * FROM posts";
        $result = $this->db->query($query);
        return $result->fetch_all(MYSQLI_ASSOC);
    }

    public function getPostById($id) {
        $query = "SELECT * FROM posts WHERE id = ?";
        $stmt = $this->db->prepare($query);
        $stmt->bind_param("i", $id);
        $stmt->execute();
        $result = $stmt->get_result();
        return $result->fetch_assoc();
    }

    public function createPost($title, $content) {
        $query = "INSERT INTO posts (title, content) VALUES (?, ?)";
        $stmt = $this->db->prepare($query);
        $stmt->bind_param("ss", $title, $content);
        return $stmt->execute();
    }

    public function updatePost($id, $title, $content) {
        $query = "UPDATE posts SET title = ?, content = ? WHERE id = ?";
        $stmt = $this->db->prepare($query);
        $stmt->bind_param("ssi", $title, $content, $id);
        return $stmt->execute();
    }

    public function deletePost($id) {
        $query = "DELETE FROM posts WHERE id = ?";
        $stmt = $this->db->prepare($query);
        $stmt->bind_param("i", $id);
        return $stmt->execute();
    }
}
```

3. Controller (PostController.php):

The controller handles user requests and interacts with the model to manipulate data.

```
class PostController {
    private $postModel;

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

    // Display all posts
    public function index() {
        $posts = $this->postModel->getAllPosts();
        include 'views/posts/index.php';
    }

    // Display single post
    public function view($id) {
        $post = $this->postModel->getPostById($id);
        include 'views/posts/view.php';
    }

    // Show the form for creating a new post
    public function create() {
        include 'views/posts/create.php';
    }

    // Store new post data
    public function store() {
        $title = $_POST['title'];
        $content = $_POST['content'];
        if ($this->postModel->createPost($title, $content)) {
            header("Location: index.php");
        }
    }

    // Show the form for editing a post
    public function edit($id) {
        $post = $this->postModel->getPostById($id);
        include 'views/posts/edit.php';
    }

    // Update post data
    public function update($id) {
        $title = $_POST['title'];
        $content = $_POST['content'];
        if ($this->postModel->updatePost($id, $title, $content)) {
            header("Location: index.php");
        }
    }

    // Delete a post
    public function delete($id) {
        if ($this->postModel->deletePost($id)) {
            header("Location: index.php");
        }
    }
}
```

4. Views:

- **views/posts/index.php**: Displays all posts.

```
<h1>All Posts</h1>
<a href="create.php">Create New Post</a>
<ul>
    <?php foreach ($posts as $post): ?>
        <li>
            <a href="view.php?id=<?php echo $post['id']; ?>"><?php echo
$post['title']; ?></a>
            <a href="edit.php?id=<?php echo $post['id']; ?>">Edit</a>
            <a href="delete.php?id=<?php echo $post['id']; ?>">Delete</a>
        </li>
    <?php endforeach; ?>
</ul>
```

- **views/posts/create.php**: Form to create a new post.

```
<h1>Create New Post</h1>
<form action="store.php" method="POST">
    <input type="text" name="title" placeholder="Title" required>
    <textarea name="content" placeholder="Content" required></textarea>
    <button type="submit">Submit</button>
</form>
```

- **views/posts/view.php**: Displays a single post.

```
<h1><?php echo $post['title']; ?></h1>
<p><?php echo $post['content']; ?></p>
<a href="index.php">Back to all posts</a>
```

5. Main Execution (index.php):

```
require_once 'PostModel.php';
require_once 'PostController.php';

// Database connection
$db = new mysqli('localhost', 'root', '', 'blog_db');
$postModel = new PostModel($db);
$postController = new PostController($postModel);

// Display posts
$postController->index();
```

Insert, Update, Delete MVC

28. Expand the Blog Application to Include a Feature for User Comments, Allowing Users to Insert, Update, and Delete Their Comments

Practical Exercise:

We'll now extend the blog application to allow users to comment on posts.

---

1. Database Structure for Comments:

Add a `comments` table to the database:

```
CREATE TABLE comments (
    id INT AUTO_INCREMENT PRIMARY KEY,
    post_id INT,
    content TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (post_id) REFERENCES posts(id)
);
```

---

2. Model (CommentModel.php):

This model will handle the comment-related operations.

```
class CommentModel {
    private $db;

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

    public function getCommentsByPostId($postId) {
        $query = "SELECT * FROM comments WHERE post_id = ?";
        $stmt = $this->db->prepare($query);
        $stmt->bind_param("i", $postId);
        $stmt->execute();
        $result = $stmt->get_result();
        return $result->fetch_all(MYSQLI_ASSOC);
    }

    public function addComment($postId, $content) {
        $query = "INSERT INTO comments (post_id, content) VALUES (?, ?)";
        $stmt = $this->db->prepare($query);
        $stmt->bind_param("is", $postId, $content);
        return $stmt->execute();
    }

    public function updateComment($id, $content) {
        $query = "UPDATE comments SET content = ? WHERE id = ?";
        $stmt = $this->db->prepare($query);
        $stmt->bind_param("si", $content, $id);
        return $stmt->execute();
    }

    public function deleteComment($id) {
        $query = "DELETE FROM comments WHERE id = ?";
        $stmt = $this->db->prepare($query);
        $stmt->bind_param("i", $id);
        return $stmt->execute();
    }
}
```

---

3. Controller (CommentController.php):

The comment controller will manage the comment functionality.

```
class CommentController {
    private $commentModel;

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

    public function store($postId) {
        $content = $_POST['content'];
        if ($this->commentModel->addComment($postId, $content)) {
            header("Location: view.php?id=$postId");
        }
    }

    public function update($id) {
        $content = $_POST['content'];
        if ($this->commentModel->updateComment($id, $content)) {
            header("Location: view.php?id=$id");
        }
    }

    public function delete($id) {
        if ($this->commentModel->deleteComment($id)) {
            header("Location: index.php");
        }
    }
}
```

---

4. Views:

- **views/comments/create.php**: Form to add a comment to a post.

```
<form action="addComment.php" method="POST">
    <textarea name="content" placeholder="Add your comment"
required></textarea>
    <button type="submit">Submit</button>
</form>
```

- **views/comments/show.php**: Display comments on a post.

```
<h2>Comments</h2>
<ul>
    <?php foreach ($comments as $comment): ?>
        <li>
            <p><?php echo $comment['content']; ?></p>
            <a href="editComment.php?id=<?php echo $comment['id'];
?>">Edit</a>
            <a href="deleteComment.php?id=<?php echo $comment['id'];
?>">Delete</a>
        </li>
    <?php endforeach; ?>
</ul>
```

---

5. Main Execution (view.php):

```
// Display post with comments
$post = $postModel->getPostById($postId);
$comments = $commentModel->getCommentsByPostId($postId);

include 'views/posts/view.php';
include 'views/comments/show.php';
```

Modules 3 – WebServices, API, Extensions

LAB EXERCISE

Note that these code snippets aim to guide you through implementing each functionality but may require additional setup (like API keys or service configurations) to work fully.

1. Payment Gateway Integration (Using Stripe)

```
// Stripe PHP SDK
require_once('vendor/autoload.php');

// Set your Stripe secret key
\Stripe\Stripe::setApiKey('sk_test_your_secret_key');

// Payment form processing
if ($_SERVER['REQUEST_METHOD'] == 'POST') {
    try {
        $token = $_POST['stripeToken'];
        $charge = \Stripe\Charge::create([
            'amount' => 5000, // Amount in cents
            'currency' => 'usd',
            'description' => 'Sample Product',
            'source' => $token,
        ]);
        echo 'Payment successful!';
    } catch (Exception $e) {
        echo 'Error: ' . $e->getMessage();
    }
}
```

---

2. Create API with Header (PHP)

```
// Simple API that accepts custom headers
if ($_SERVER['REQUEST_METHOD'] == 'GET') {
    $customHeader = $_SERVER['HTTP_X_CUSTOM_HEADER'] ?? 'Header not set';
    header('Content-Type: application/json');
    echo json_encode(['custom_header' => $customHeader]);
}
```

To send the header from client:

```
fetch('https://your-api.com/endpoint', {
    method: 'GET',
    headers: {
        'X-Custom-Header': 'CustomHeaderValue'
    }
})
.then(response => response.json())
.then(data => console.log(data));
```

---

3. API with Image Uploading (PHP)

```
// Handle image upload securely
if ($_SERVER['REQUEST_METHOD'] == 'POST' && isset($_FILES['image'])) {
    $file = $_FILES['image'];
    $allowedTypes = ['image/jpeg', 'image/png', 'image/gif'];

    if (in_array($file['type'], $allowedTypes) && $file['size'] <= 2000000) {
// 2MB
        $uploadDir = 'uploads/';
        $filePath = $uploadDir . basename($file['name']);
        if (move_uploaded_file($file['tmp_name'], $filePath)) {
            echo 'File uploaded successfully!';
        } else {
            echo 'File upload failed!';
        }
    } else {
        echo 'Invalid file type or size.';
    }
}
```

HTML Form:

```
<form action="" method="POST" enctype="multipart/form-data">
    <input type="file" name="image">
    <input type="submit" value="Upload">
</form>
```

---

4. SOAP and REST APIs (Simple REST API)

REST API (PHP):

```
// Example of a simple REST API (Product catalog)
$method = $_SERVER['REQUEST_METHOD'];
$product = ['id' => 1, 'name' => 'Product 1', 'price' => 100];

if ($method == 'GET') {
    header('Content-Type: application/json');
    echo json_encode($product);
}
```

Questions:

5. Product Catalog (Database Integration)

```
// Simple Product Catalog with MySQL
$pdo = new PDO('mysql:host=localhost;dbname=shop', 'username', 'password');

$query = 'SELECT * FROM products';
$stmt = $pdo->query($query);
$products = $stmt->fetchAll(PDO::FETCH_ASSOC);

foreach ($products as $product) {
    echo "<div>{$product['name']} - {$product['price']}</div>";
}
```

6. Shopping Cart (PHP Session)

```
// Start session to store cart
session_start();

if ($_SERVER['REQUEST_METHOD'] == 'POST') {
    $productId = $_POST['product_id'];
    if (!isset($_SESSION['cart'])) {
        $_SESSION['cart'] = [];
    }

    $_SESSION['cart'][] = $productId;
    echo "Product added to cart!";
}

if ($_SERVER['REQUEST_METHOD'] == 'GET' && isset($_SESSION['cart'])) {
    echo 'Cart contains: ' . implode(', ', $_SESSION['cart']);
}
```

7. Web Services (Simple RESTful Service)

```
// Simple Web Service that returns product data
$method = $_SERVER['REQUEST_METHOD'];
if ($method == 'GET') {
    $product = ['id' => 1, 'name' => 'Sample Product', 'price' => 20.00];
    header('Content-Type: application/json');
    echo json_encode($product);
}
```

8. Create Web Services for MVC Project (Extending an MVC Project)

In an existing MVC project, you might add the following:

```
// In ProductController.php
public function getProductData($id) {
    $product = $this->model->getProductById($id);
    echo json_encode($product);
}

// In model
public function getProductById($id) {
    $stmt = $this->db->prepare("SELECT * FROM products WHERE id = ?");
    $stmt->execute([$id]);
    return $stmt->fetch();
}
```

---

9. Integration of API in Project (Using OpenWeatherMap API)

```
// Example to get weather from OpenWeatherMap API
$apiKey = 'your_api_key';
$city = 'London';
$response =
file_get_contents("https://api.openweathermap.org/data/2.5/weather?q=$city&ap
pid=$apiKey");
$data = json_decode($response, true);

echo 'Weather in ' . $city . ': ' . $data['weather'][0]['description'];
```

---

10. Implement RESTful Principles (PHP)

```
// Example of a RESTful API for products (GET, POST, DELETE)
if ($_SERVER['REQUEST_METHOD'] == 'GET') {
    $productId = $_GET['id'];
    if ($productId) {
        // Fetch single product by ID
        $product = getProductById($productId); // Assume function is defined
        echo json_encode($product);
    } else {
        // Fetch all products
        echo json_encode(getAllProducts()); // Assume function is defined
    }
}

if ($_SERVER['REQUEST_METHOD'] == 'POST') {
    $data = json_decode(file_get_contents('php://input'), true);
    addProduct($data); // Assume function is defined
    echo 'Product added successfully!';
}
```

---

11. OpenWeatherMap API (Weather Dashboard)

```
// Example to display weather data
$apiKey = 'your_api_key';
$city = 'London';
$response =
file_get_contents("https://api.openweathermap.org/data/2.5/weather?q=$city&ap
pid=$apiKey");
$data = json_decode($response, true);

echo '<h1>Weather in ' . $city . '</h1>';
echo 'Temperature: ' . $data['main']['temp'] . '°C';
echo 'Condition: ' . $data['weather'][0]['description'];
```

---

12. Google Maps Geocoding API (Location Application)

```
// Example to get coordinates using Google Maps Geocoding API
$address = '1600+Amphitheatre+Parkway,+Mountain+View,+CA';
$apiKey = 'your_api_key';
$response =
file_get_contents("https://maps.googleapis.com/maps/api/geocode/json?address=
$address&key=$apiKey");
$data = json_decode($response, true);
```

```
$latitude = $data['results'][0]['geometry']['location']['lat'];
$longitude = $data['results'][0]['geometry']['location']['lng'];

echo "Coordinates: $latitude, $longitude";
```

These are basic code examples for each of the exercises in

Module 3: Web Services, API, Extensions.

They are meant to get you started, but you might need to adjust them depending on your setup, such as API keys or database configurations.

Let me know if you'd like further elaboration or additional examples!

Below are the code examples and explanations for each of the **Challenging Practical Exercises** you mentioned. Each exercise will cover how to implement the specified functionality, including the integration of multiple technologies like APIs, MVC architecture, and third-party services.

Challenging Practical Exercises

1. Payment Gateway Integration (PayPal and Stripe)

Practical:

- Integrate both PayPal and Stripe for payment processing.
- Implement a checkout process with error handling.
- Use webhooks to update order statuses based on payment results.

```
// PayPal Payment Integration
$paypalUrl = 'https://www.sandbox.paypal.com/cgi-bin/webscr';
$paypalEmail = 'your-paypal-email@example.com';

// Form submission to PayPal
echo '
    <form action="'.$paypalUrl.'" method="post">
        <input type="hidden" name="business" value="'.$paypalEmail.'">
        <input type="hidden" name="cmd" value="_xclick">
        <input type="hidden" name="item_name" value="Sample Product">
        <input type="hidden" name="amount" value="100.00">
        <input type="hidden" name="currency_code" value="USD">
        <input type="submit" value="Pay with PayPal">
    </form>
';

// Stripe Payment Integration (requires Stripe PHP SDK)
require 'vendor/autoload.php';
\Stripe\Stripe::setApiKey('sk_test_your_secret_key');

if ($_SERVER['REQUEST_METHOD'] == 'POST') {
    $token = $_POST['stripeToken'];
    $charge = \Stripe\Charge::create([
        'amount' => 10000,  // $100.00
        'currency' => 'usd',
        'description' => 'Sample Product',
        'source' => $token,
    ]);
    echo 'Payment successful!';
}
```

---

2. Create API with Header & API with Image Uploading

Practical:

- Create a RESTful API that handles user registration with image uploads and custom headers.

```
// Handle Image Upload and Authentication (using custom headers)
if ($_SERVER['REQUEST_METHOD'] == 'POST') {
    // Custom header authentication
    $authToken = $_SERVER['HTTP_AUTHORIZATION'] ?? '';
    if ($authToken != 'Bearer your-token') {
        echo json_encode(['error' => 'Unauthorized']);
        exit;
    }

    $image = $_FILES['avatar'];
    if ($image['type'] == 'image/jpeg' || $image['type'] == 'image/png') {
        // Move image to server
        move_uploaded_file($image['tmp_name'], 'uploads/'.$image['name']);
        echo json_encode(['message' => 'Image uploaded successfully']);
    } else {
        echo json_encode(['error' => 'Invalid image format']);
    }
}
```

---

3. Payment Gateway Implementation on MVC Project

Practical:

- Implement user authentication, store cart items in sessions, handle payment, and order confirmation using MVC.

```
// Controller (CheckoutController.php)
public function checkoutAction() {
    if ($_SERVER['REQUEST_METHOD'] == 'POST') {
        $paymentMethod = $_POST['payment_method'];
        $cartItems = $_SESSION['cart'];
        // Process payment with Stripe or PayPal based on the selected method
        // Confirm order and save details in the database
        echo "Order confirmed!";
    }
}

// Model (OrderModel.php)
public function saveOrder($cartItems, $paymentMethod) {
    // Insert order data into the database
    $stmt = $this->db->prepare("INSERT INTO orders (items, payment_method)
VALUES (?, ?)");
    $stmt->execute([json_encode($cartItems), $paymentMethod]);
}
```

---

4. SOAP and REST API Creation for CRUD Operations (Library System)

Practical:

- Create SOAP and REST APIs for CRUD operations.

```
// REST API for Books (BookController.php)
public function getBooks() {
    $books = $this->bookModel->getAllBooks();
    echo json_encode($books);
}

// SOAP API for Books (BookController.php)
$server = new SoapServer(null, array('uri' => 'http://localhost/'));
$server->addFunction('getAllBooks');
$server->handle();

function getAllBooks() {
    // Fetch books from the database
    return $books;
}
```

---

5. Product Catalog with Search & Filtering (AJAX)

Practical:

- Implement a dynamic product catalog with search and filtering.

```
// Product Search API (ProductController.php)
public function searchProducts() {
    $search = $_GET['search'] ?? '';
    $category = $_GET['category'] ?? '';
    $priceRange = $_GET['priceRange'] ?? '';

    // Fetch products from database with search and filter
    $query = "SELECT * FROM products WHERE name LIKE ? AND category LIKE ?
AND price BETWEEN ? AND ?";
    $stmt = $this->db->prepare($query);
    $stmt->execute(["%$search%", "%$category%", $priceRange[0],
$priceRange[1]]);
    $products = $stmt->fetchAll();
    echo json_encode($products);
}
```

AJAX for live search:

```
$('#searchBox').on('input', function() {
    let searchQuery = $(this).val();
    $.get('/searchProducts', { search: searchQuery }, function(data) {
        // Update product list
        $('#productList').html(data);
    });
});
```

---

6. Persistent Shopping Cart with User Accounts

Practical:

- Store cart items in the database and associate them with user accounts.

```
// Shopping Cart Model (CartModel.php)
public function saveCart($userId, $cartItems) {
    foreach ($cartItems as $item) {
        // Insert items into cart table
        $stmt = $this->db->prepare("INSERT INTO cart (user_id, product_id, 
quantity) VALUES (?, ?, ?)");
        $stmt->execute([$userId, $item['product_id'], $item['quantity']]);
    }
}

// Retrieve Cart Items (CartController.php)
public function getCart($userId) {
    $stmt = $this->db->prepare("SELECT * FROM cart WHERE user_id = ?");
    $stmt->execute([$userId]);
    return $stmt->fetchAll();
}
```

---

7. Web Services with Multiple APIs Integration

Practical:

- Integrate multiple APIs (Weather, Country Info, GitHub).

```
// Web service for integrating multiple APIs (WebServiceController.php)
public function getCombinedData() {
    $weather = 
file_get_contents('https://api.openweathermap.org/data/2.5/weather?q=London&a
ppid=your_api_key');
    $countryInfo = 
file_get_contents('https://restcountries.com/v3.1/name/UK');
    $github = 
file_get_contents('https://api.github.com/users/octocat/repos');

    $data = [
        'weather' => json_decode($weather),
        'country_info' => json_decode($countryInfo),
        'github_repos' => json_decode($github)
    ];

    echo json_encode($data);
}
```

---

8. Create Web Services for MVC Project (User Data API)

Practical:

- Secure API for user data with token-based authentication.

```
// UserController.php
public function getUserData($token) {
    if ($this->validateToken($token)) {
        // Fetch and return user data
        $user = $this->userModel->getUserByToken($token);
        echo json_encode($user);
    } else {
        echo json_encode(['error' => 'Unauthorized']);
    }
}

// Token validation
public function validateToken($token) {
    // Validate token logic
    return true;
}
```

---

9. Integration of API in Project (Weather Dashboard)

Practical:

- Combine OpenWeatherMap API with user input.

```
// Weather dashboard
if ($_SERVER['REQUEST_METHOD'] == 'POST') {
    $city = $_POST['city'];
    $weatherData =
file_get_contents("https://api.openweathermap.org/data/2.5/weather?q=$city&ap
pid=your_api_key");
    echo json_encode(json_decode($weatherData));
}
```

---

10. RESTful Principles Implementation (Inventory Management API)

Practical:

- Implement endpoints for adding, retrieving, updating, and deleting inventory items.

```
// InventoryController.php
public function addInventoryItem() {
    // Add new inventory item
}

public function getInventoryItems() {
    // Get all inventory items
}

public function updateInventoryItem($id) {
    // Update inventory item by ID
}

public function deleteInventoryItem($id) {
    // Delete inventory item by ID
}
```

---

11. OpenWeatherMap API (Weather Comparison)

Practical:

- Compare weather conditions across multiple cities.

```
// Fetch weather data for multiple cities
$cities = ['London', 'Paris', 'New York'];
$weatherData = [];
foreach ($cities as $city) {
    $response =
file_get_contents("https://api.openweathermap.org/data/2.5/weather?q=$city&ap
pid=your_api_key");
    $weatherData[$city] = json_decode($response, true);
}
echo json_encode($weatherData);
```

---

12. Google Maps Geocoding API (Directions Service)

Practical:

- Provide directions and distance between two addresses.

```
// Google Maps Geocoding API for directions
$start = 'New York, NY';
$end = 'Los Angeles, CA';
$response =
file_get_contents("https://maps.googleapis.com/maps/api/directions/json?origi
n=$start&destination=$end&key=your_api_key");
echo $response;
```

13. GitHub API (Profile Viewer)

Practical:

- Fetch user data and display repositories.

```
// Fetch user repositories from GitHub
$username = 'octocat';
$response =
file_get_contents("https://api.github.com/users/$username/repos");
$repos = json_decode($response);
echo json_encode($repos);
```

14. Twitter API (Sentiment Analysis Tool)

Practical:

- Fetch tweets and analyze sentiment.

```
// Fetch tweets with Twitter API and analyze sentiment (basic example)
$tweets = fetchTweets('keyword');
$sentiment = analyzeSentiment($tweets); // Implement basic sentiment analysis
```

15. REST Countries API (Travel App)

Practical:

- Fetch and compare country data.

```
// Fetch country data from REST Countries API
$country = file_get_contents('https://restcountries.com/v3.1/name/USA');
echo $country;
```

16. SendGrid Email API - Notification System

Practical:

- Send confirmation emails, newsletters, and track delivery status.
- Create an admin interface to manage email templates and view reports.

```
// Install SendGrid PHP SDK
// composer require sendgrid/sendgrid

use SendGrid;

$sendgridApiKey = 'your-sendgrid-api-key';
$from = new SendGrid\Mail\From("no-reply@example.com", "Your App");
```

```
$to = new SendGrid\Mail\To("user@example.com");
$subject = "Confirmation Email";
$content = new SendGrid\Mail\Content("text/plain", "Thank you for your
registration!");
$mail = new SendGrid\Mail\Mail($from, $to, $subject, $content);

$sendgrid = new SendGrid($sendgridApiKey);
$response = $sendgrid->send($mail);

echo "Email Status Code: " . $response->statusCode() . "\n";
echo "Response Body: " . $response->body() . "\n";
echo "Response Headers: " . $response->headers() . "\n";

// Track delivery status and handle failures
if ($response->statusCode() != 202) {
    echo "Error: Email sending failed\n";
    // Log error or handle retry mechanism
}
```

Admin Interface (Example for managing templates):

```
// Assuming we store templates in the database, this will allow the admin to
edit templates.
public function getTemplates() {
    $stmt = $this->db->query("SELECT * FROM email_templates");
    return $stmt->fetchAll();
}
```

---

17. Social Authentication (Google and Facebook)

Practical:

- Implement OAuth for Google and Facebook authentication.

```
// Google OAuth Integration (using Google Client Library)
require_once 'vendor/autoload.php';

$client = new Google_Client();
$client->setClientId('your-client-id');
$client->setClientSecret('your-client-secret');
$client->setRedirectUri('your-redirect-uri');
$client->addScope('email');

// Redirect to Google's OAuth page
if (!isset($_GET['code'])) {
    $authUrl = $client->createAuthUrl();
    header('Location: ' . $authUrl);
    exit;
} else {
    $token = $client->fetchAccessTokenWithAuthCode($_GET['code']);
    $client->setAccessToken($token);

    // Get user info
    $googleService = new Google_Service_Oauth2($client);
    $userInfo = $googleService->userinfo->get();
    // Store user data securely (use encryption and hashing)
    saveUserToDatabase($userInfo);
}

// Facebook OAuth Integration (using Facebook SDK)
require_once 'vendor/autoload.php';

$fb = new Facebook\Facebook([
    'app_id' => 'your-app-id',
    'app_secret' => 'your-app-secret',
    'default_graph_version' => 'v12.0',
]);

$helper = $fb->getRedirectLoginHelper();
$permissions = ['email']; // Optional permissions
$loginUrl = $helper->getLoginUrl('your-redirect-url', $permissions);

// Redirect to Facebook's OAuth page
echo '<a href="' . $loginUrl . '">Log in with Facebook</a>';
```

---

18. Email Sending APIs (Mailgun - Marketing Email System)

Practical:

- Allow users to subscribe/unsubscribe to newsletters.
- Use Mailgun to send bulk marketing emails and track open/click rates.

```
// Mailgun Integration (Install via Composer: composer require
mailgun/mailgun-php)
use Mailgun\Mailgun;

$mgClient = Mailgun::create('your-mailgun-api-key');
$domain = 'your-domain.com';

$data = [
    'from'    => 'no-reply@your-domain.com',
    'to'      => 'user@example.com',
    'subject' => 'Welcome to Our Newsletter',
    'text'    => 'Thank you for subscribing to our newsletter!'
];

// Send email
$response = $mgClient->messages()->send($domain, $data);

// Track open/click rates
// You would set up webhooks in your Mailgun account to handle events like
opens, clicks, and bounces.
```

Subscription Interface:

```
// User subscribe/unsubscribe management (stored in database)
public function manageSubscription($userId, $action) {
    if ($action == 'subscribe') {
        // Insert subscription into DB
    } elseif ($action == 'unsubscribe') {
        // Remove subscription from DB
    }
}
```

19. SMS Sending APIs (Twilio)

Practical:

- Send SMS notifications using Twilio for events like appointments.
- Track delivery status and handle errors.

```
// Twilio Integration (Install via Composer: composer require twilio/sdk)
use Twilio\Rest\Client;

$sid = 'your_twilio_sid';
$token = 'your_twilio_auth_token';
$twilio = new Client($sid, $token);

$twilio->messages->create(
    '+1234567890', // recipient phone number
    [
        'from' => '+1987654321', // Twilio number
        'body' => 'Your appointment is confirmed for tomorrow at 10:00 AM.'
    ]
);

// Track delivery status with Twilio Webhooks
// Implement a webhook handler to listen for status updates
```

20. Google Maps API (Location-Sharing Application)

Practical:

- Use Google Maps API to display a map and user locations.
- Implement check-in and share location functionality.

```
<!-- Google Maps Integration with JavaScript -->
<!DOCTYPE html>
<html>
<head>
    <title>Location Sharing</title>
    <script src="https://maps.googleapis.com/maps/api/js?key=your-api-
key&callback=initMap" async defer></script>
    <script>
        let map;

        function initMap() {
            const userLocation = { lat: 40.7128, lng: -74.0060 }; // Example
location

            map = new google.maps.Map(document.getElementById("map"), {
                center: userLocation,
                zoom: 10
            });

            // Display user location
            new google.maps.Marker({
                position: userLocation,
                map: map,
                title: "Your Location"
            });

            // Listen for user check-in (using Geolocation API)
            if (navigator.geolocation) {
                navigator.geolocation.getCurrentPosition(function(position) {
                    const userPos = {
                        lat: position.coords.latitude,
                        lng: position.coords.longitude
                    };

                    map.setCenter(userPos);
                    new google.maps.Marker({
                        position: userPos,
                        map: map,
                        title: "You are here"
                    });
                });
            }
        }
    </script>
</head>
<body onload="initMap()">
    <div id="map" style="height: 400px; width: 100%;"></div>
</body>
</html>
```

Functionality for Sharing Locations with Friends

```
// Allow users to share their location with friends
function shareLocationWithFriend(friendId) {
    navigator.geolocation.getCurrentPosition(function(position) {
        const locationData = {
            lat: position.coords.latitude,
            lng: position.coords.longitude,
            friendId: friendId
        };
        // Send location data to the server
        fetch('/share-location', {
            method: 'POST',
            body: JSON.stringify(locationData),
            headers: { 'Content-Type': 'application/json' }
        });
    });
}
```

Backend (PHP) for Storing and Retrieving Shared Locations:

```
// Store user location
public function storeLocation($userId, $latitude, $longitude) {
    $stmt = $this->db->prepare("INSERT INTO user_locations (user_id, lat,
lng) VALUES (?, ?, ?)");
    $stmt->execute([$userId, $latitude, $longitude]);
}

// Retrieve friend locations
public function getFriendLocations($friendIds) {
    $stmt = $this->db->prepare("SELECT * FROM user_locations WHERE user_id IN
(?)");
    $stmt->execute([implode(',', $friendIds)]);
    return $stmt->fetchAll();
}
```