Course Code: CEF 468

Course Title: Introduction to PHP and Database Management

Course Overview

CEF 468 - Introduction to PHP and Database Management is a comprehensive course designed to introduce students to the fundamental concepts of **PHP programming** and **relational database management systems (RDBMS)**, with a particular focus on **MySQL**. The course aims to equip students with the necessary tools and techniques to develop dynamic, data-driven web applications. By the end of the course, students will be proficient in working with **PHP** for backend development and **MySQL** for managing and manipulating databases.

The course is divided into practical lab exercises, starting from the basics of databases and SQL, progressing to more advanced topics like object-oriented programming (OOP), integrating PHP with MySQL, and building full-stack applications.

Course Objectives

By the end of the course, students will be able to:

1. **Understand Database Concepts**: Explain the basics of relational databases, tables, rows, and columns, and be able to design and interact with databases using **MySQL**.
2. **Develop Dynamic Web Applications**: Use **PHP** to create interactive websites that can handle user input, process forms, and interact with databases.
3. **Work with MySQL**: Perform CRUD (Create, Read, Update, Delete) operations, design efficient database schemas, and optimize SQL queries.
4. **Understand Object-Oriented Programming**: Implement OOP concepts like classes, objects, inheritance, and polymorphism within PHP.
5. **Build Real-World Web Applications**: Integrate PHP with MySQL to build fully functional web applications, including features such as user authentication and CRUD functionality.
6. **Ensure Web Application Security**: Understand and apply best practices in securing web applications against common threats like SQL injection, XSS, and CSRF.

Course Format

The course will be **lab-based** with hands-on exercises that reinforce theoretical concepts:

- **Lectures**: Introduction to key concepts and live demonstrations.
- **Lab Exercises**: Guided practice sessions where students will write PHP scripts and interact with MySQL databases.
- **Projects**: Students will develop real-world applications based on the concepts learned in the lab exercises.
- **Assessments**: Regular quizzes, assignments, and a final project to demonstrate understanding and application of PHP and MySQL.

Lab 1: Understanding Databases and Developing Dynamic Web Applications

Objective

By the end of this lab, students will:

1. Understand the basic concepts of relational databases, tables, rows, columns, and how they relate to each other.
2. Be able to design and interact with databases using **MySQL**.
3. Develop dynamic web applications using **PHP** that can handle user input, process forms, and interact with databases.
4. Implement basic **CRUD (Create, Read, Update, Delete)** operations using **PHP** and **MySQL**.

Exercise 1: Theory Questions on Database Concepts

Objective:

Understand and explain the basics of relational databases, tables, rows, columns, and the importance of database design.

Questions:

1. **What is a database?**
    - Define a **database** and explain its primary purpose in managing large amounts of data efficiently.
2. **What are the key differences between relational databases and flat files?**
    - Discuss how data is organized and accessed in relational databases versus flat files, and explain the advantages of using a relational database.
3. **Explain the concepts of tables, rows, and columns in a relational database.**
    - Define **tables**, **rows**, and **columns**, and explain their roles in organizing data within a relational database.
4. **What is database normalization? Why is it important?**
    - Define **normalization** and explain the reasons for normalizing databases. What problems can occur if a database is not normalized?

5. **What is the difference between 1NF, 2NF, and 3NF in database normalization?**
    - Briefly describe **1st**, **2nd**, and **3rd Normal Forms (1NF, 2NF, 3NF)** and why they are used in database design.
6. **What is atomicity in a database?**
    - Explain **atomicity** and its role in ensuring that database transactions are fully completed or not performed at all.
7. **What is a primary key, and why is it important?**
    - Define a **primary key** and explain how it ensures data integrity in a relational database.
8. **What is a foreign key, and how does it link tables?**
    - Define a **foreign key** and explain how it is used to establish a relationship between two tables in a relational database.
9. **What is the role of indexes in database management systems?**
    - Explain how **indexes** help improve query performance and make data retrieval faster in relational databases.
10. **What is the difference between atomic and non-atomic operations in database systems?**

- Discuss the significance of atomic operations in ensuring data consistency in multi-step transactions.

Exercise 2: Installing MySQL

Objective:

Learn how to install MySQL on Windows, Linux, or macOS systems.

Instructions:

Copy and paste the installation links and steps for each operating system to install MySQL.

1. Install MySQL on Windows:

1. Go to the official MySQL Downloads page https://dev.mysql.com/downloads/installer/.
2. Download the **MySQL Installer** for Windows.

3. Run the installer and select **Server Only** installation.
4. Follow the setup prompts, setting the default configuration options (root password, server settings).
5. After installation, open the **MySQL Command Line Client** and log in with the root password to start interacting with MySQL.

2. Install MySQL on Linux (Ubuntu):

1. Open the terminal and run the following commands:

   sudo apt update

   sudo apt install mysql-server

2. After installation, run:

   sudo service mysql start

3. Secure MySQL by running:

   sudo mysql_secure_installation

4. Log in to MySQL:

   sudo mysql -u root -p

3. Install MySQL on macOS:

1. Download **MySQL Community Server** from the MySQL Downloads page.
2. Install the package by following the installation prompts.
3. Once installed, start MySQL with:

   sudo /usr/local/mysql/support-files/mysql.server start

4. Log in to MySQL:

   mysql -u root -p

Exercise:

1. After installing MySQL, open the MySQL client and create a database named TestDB using the following SQL command:

   CREATE DATABASE TestDB;

2. Verify the creation of the database:

SHOW DATABASES;

3. Create a table called Books with the following structure:

```
CREATE TABLE Books (
  book_id INT AUTO_INCREMENT PRIMARY KEY,
  title VARCHAR(255),
  author VARCHAR(255),
  publication_year INT,
  genre VARCHAR(100),
  price FLOAT
);
```

4. Insert a few records into the Books table. For example:

```
INSERT INTO Books (title, author, publication_year, genre, price)
VALUES ('The Great Gatsby', 'F. Scott Fitzgerald', 1925, 'Fiction', 10.99),
    ('To Kill a Mockingbird', 'Harper Lee', 1960, 'Fiction', 7.99);
```

Exercise 3: Installing PHP

Objective:

Learn how to install PHP and run PHP scripts on your local machine.

Instructions:

Copy and paste the installation links and steps for each operating system to install PHP.

1. Install PHP on Windows:

1. Download **PHP** from the PHP for Windows download page https://windows.php.net/download/.
2. Extract the downloaded file to a folder, such as C:\php.
3. Add C:\php to the **Path** environment variable:
    - Right-click **This PC** -> **Properties** -> **Advanced system settings** -> **Environment Variables**.
    - Add C:\php to the **Path** variable.

4. Open Command Prompt and check if PHP is installed:
5. php -v

2. Install PHP on Linux (Ubuntu):

1. Open the terminal and run:

   sudo apt update

   sudo apt install php libapache2-mod-php
2. Restart Apache server:

   sudo service apache2 restart

3. Install PHP on macOS:

1. Install **Homebrew** from the Homebrew website https://brew.sh/.
2. Install PHP:

   brew install php

Exercise for Students:

1. After installing PHP, check the PHP version by running:

   php -v
2. Create a simple PHP script hello.php that displays "Hello, World!" and run it:

   <?php

   echo "Hello, World!";

   ?>
3. Run the script from the command line:
4. php hello.php

Exercise 4: Basic PHP CRUD Application

Objective:

Develop a basic **CRUD (Create, Read, Update, Delete)** application using **PHP** and **MySQL**.

Steps:

1. **Create a Database:**
    - Create a database called LibraryDB in MySQL:

```
CREATE DATABASE LibraryDB;

USE LibraryDB;
```

2. **Create a Table:**
    - Create a Books table:

```
CREATE TABLE Books (
  book_id INT AUTO_INCREMENT PRIMARY KEY,
  title VARCHAR(255),
  author VARCHAR(255),
  publication_year INT,
  genre VARCHAR(100),
  price FLOAT
);
```

3. **Create the PHP Scripts:**
    - **Create**: Create a PHP form (create_book.php) that allows users to add books to the database.

```
<form method="POST" action="create_book.php">
  Title: <input type="text" name="title" required><br>
  Author: <input type="text" name="author" required><br>
  Year: <input type="number" name="year" required><br>
  Genre: <input type="text" name="genre" required><br>
  Price: <input type="text" name="price" required><br>
  <input type="submit" value="Add Book">
</form>
```

    - **Read**: Display all books in the Books table (read_books.php).

- **Update**: Allow updating book information (update_book.php).
- **Delete**: Create a delete button to remove books from the database (delete_book.php).

4. **Test the Application:**
   - Open each PHP script in a browser and test adding, viewing, updating, and deleting books from the database.

Lab 2: Developing Dynamic Web Applications Using PHP - Part 1

Objective

By the end of **Lab 2**, students will:

1. Learn how to create dynamic web applications using **PHP** that handle user input via forms and interact with **MySQL databases**.
2. Develop applications that validate and display data from the database.
3. Understand how to normalize a database to **1NF** (First Normal Form).

Exercise 1: PHP Form to Add and View Data in MySQL

Objective:

Create a simple PHP application that accepts user input via an HTML form, inserts the data into a MySQL database, and displays it.

Steps:

1. **Database Setup:**
    - Create a new database called **WebAppDB**.
    - Create a table called Users with the following columns: id (auto-increment), name (VARCHAR), email (VARCHAR), and age (INT).
    - Ensure the table is in **1NF** (First Normal Form) by ensuring no repeating groups and each column contains atomic values.
2. **Create the PHP Form (File: user_form.php):**
    - Design a form that asks for name, email, and age from the user.
    - The form will use the **POST** method to send the data to a PHP script called process_form.php.
3. **Process the Form Data (File: process_form.php):**
    - This script will process the data sent from the form.
    - Validate the inputs to ensure they are not empty and properly formatted (e.g., check if the email is valid).
    - Insert the data into the Users table in WebAppDB.
4. **Display Data from the Database (File: view_users.php):**

- Create a page that retrieves and displays all user data from the Users table in a table format.
- Use a **SELECT** SQL query to fetch all records and **echo** the results in an HTML table.

5. **Test the Application:**
   - Open user_form.php in a browser and submit new user data.
   - Use view_users.php to verify that the data has been correctly inserted into the database.

Questions:

1. What is the importance of using **1NF (First Normal Form)** while designing a database?
2. Why should data be sanitized and validated before inserting it into the database?
3. How does displaying data from a MySQL database help in the management of information in a web application?

Exercise 2: PHP and MySQL with Data Validation and Display

Objective:

Enhance the PHP application to include data validation, display database records with proper formatting, and ensure the system adheres to **2NF** (Second Normal Form).

Steps:

1. **Database Setup:**
   - Create a database named **LibrarySystemDB**.
   - Create two tables: Books (for book information) and Authors (for author details).
   - Ensure the database is in **2NF** by removing partial dependencies. The Books table will store the author_id as a **foreign key** referencing the Authors table.
2. **Create the PHP Form (File: add_book.php):**
   - This form will accept book_title, author_id (a dropdown), genre, and price from the user.
   - Populate the dropdown menu for author_id with data from the Authors table.

- The form will submit data to process_book.php.

3. **Process the Form Data (File: process_book.php):**
    - The script will validate the input and ensure price is a valid number before inserting the book details into the Books table.
    - It will also ensure that data is entered correctly into the Books and Authors tables.
4. **Display Data from the Database (File: view_books.php):**
    - This page will fetch all books and display them along with the author's name by performing an **INNER JOIN** query between Books and Authors.
5. **Test the Application:**
    - Open add_book.php and submit new book information.
    - View the added books and their authors in view_books.php.

Questions:

1. What is the difference between **1NF** and **2NF** in database normalization?
2. Why is it necessary to use foreign keys in relational databases?
3. How can you ensure that the data entered into the form is valid before inserting it into the database?

Lab 3: Developing Dynamic Web Applications Using PHP - Part 2

Objective

By the end of **Lab 3**, students will:

1. Learn how to implement **database normalization** to **3NF** (Third Normal Form).
2. Build a **PHP CRUD** (Create, Read, Update, Delete) application that can interact with a MySQL database.

Exercise 1: Database Normalization to 3NF

Objective:

Normalize a database to **3NF** (Third Normal Form) and understand the relationship between tables in a normalized database.

Steps:

1. **Database Setup:**
    - Create a database called **EmployeeDB**.
    - Create an initial unnormalized table EmployeeInfo with the columns: emp_id, emp_name, emp_salary, emp_dept, dept_location.
    - Normalize the table by:
        - Splitting it into two tables: Employee (containing employee details) and Department (containing department details).
        - Establish a relationship between the Employee table and Department table using a **foreign key** (emp_dept_id).
2. **Create the PHP Form (File: add_employee.php):**
    - This form will accept emp_name, emp_salary, and emp_dept (dropdown, populated with department names from Department table).
    - Submit the form data to process_employee.php.
3. **Process the Form Data (File: process_employee.php):**
    - Insert the employee data into the Employee table, using the emp_dept_id as a foreign key to the Department table.
    - Validate the input to ensure all fields are properly filled.

4. **Display Data from the Database (File: view_employees.php):**
    - This page will display employee data, including the department name, by performing an **INNER JOIN** between the Employee and Department tables.
5. **Test the Application:**
    - Add new employee information using add_employee.php.
    - View the employee details with their department information in view_employees.php.

Questions:

1. How does **3NF** differ from **2NF**, and why is it important to apply **3NF** in database design?
2. What problems might arise if a database is not normalized beyond **1NF**?
3. What are the advantages of splitting tables in relational database design?

Exercise 2: Basic PHP CRUD Application

Objective:

Develop a simple **PHP CRUD** application that performs basic operations on a MySQL database.

Steps:

1. **Database Setup:**
    - Create a database called **StudentDB** and a table Students with columns: student_id, name, email, and phone_number.
2. **Create the PHP Form for Adding a New Student (File: add_student.php):**
    - This form will accept name, email, and phone_number as inputs.
    - The form submits data to insert_student.php.
3. **Process the Form Data and Insert into Database (File: insert_student.php):**
    - This script will insert the new student record into the Students table.
4. **Display Students (File: view_students.php):**
    - Create a page to display all student records in a table format.

5. **Update Student Data (File: edit_student.php):**
    - Allow users to edit student information by retrieving the student's data using student_id and updating the record.
6. **Delete Student Data (File: delete_student.php):**
    - Allow users to delete a student record using student_id.
7. **Test the Application:**
    - Test the **Create**, **Read**, **Update**, and **Delete** (CRUD) functionalities.

Questions:

1. What is the purpose of a **foreign key** in a relational database, and why is it important for maintaining referential integrity?
2. Describe the **CRUD** operations and give one example of each in the context of PHP and MySQL.
3. How would you modify a CRUD application to prevent SQL injection?

Lab 4: Understanding Object-Oriented Programming (OOP) in PHP

Objective

By the end of **Lab 4**, students will:

1. Understand the basics of **Object-Oriented Programming (OOP)** in PHP.
2. Learn how to implement **classes** and **objects**, **inheritance**, and **polymorphism** in PHP.
3. Build a small PHP application using these OOP principles to model a real-world scenario (a **Library System**).

Exercise 1: Introduction to Classes and Objects

Objective:

Learn how to define **classes** and create **objects** in PHP.

Steps:

1. **Create a Class (File: Book.php):**
    - Define a Book class with the following properties:
        - title (string)
        - author (string)
        - publication_year (int)
        - genre (string)
        - price (float)
    - Include a constructor to initialize these properties when a new object is created.
2. **Create an Object from the Class (File: create_book.php):**
    - In this file, create an object of the Book class.
    - Use the object to set the values for the properties and display the values using a simple method (e.g., displayBookInfo()).

Instructions:

- Define the Book class and its constructor in Book.php.

- In create_book.php, instantiate the Book class, set the values for a specific book, and call the displayBookInfo() method to show the book's details.

Questions:

1. What is a class in PHP, and how does it differ from an object?
2. What is the role of the constructor in a class?
3. How do you instantiate an object from a class?

Exercise 2: Implementing Inheritance in PHP

Objective:

Learn how to implement **inheritance** in PHP to create a class hierarchy.

Steps:

1. **Create a Parent Class (File: Product.php):**
    - Define a Product class with the following properties:
        - product_name (string)
        - product_price (float)
    - Include a method displayProduct() to display the name and price of the product.
2. **Create a Child Class (File: Book.php):**
    - Inherit from the Product class to create a Book class.
    - Add additional properties specific to books, like author, publication_year, and genre.
    - Override the displayProduct() method to also display book-specific information, such as author and publication_year.
3. **Test Inheritance (File: test_inheritance.php):**
    - In test_inheritance.php, create objects of both the Product and Book classes and call their respective displayProduct() methods.

Instructions:

- Create the Product class in Product.php.

- Create the Book class in Book.php, inheriting from Product.php and overriding the displayProduct() method.
- Instantiate and test both classes in test_inheritance.php.

Questions:

1. What is inheritance in OOP, and how does it promote code reuse?
2. How do you override methods in a child class?
3. Can a child class call a method from the parent class? If so, how?

Exercise 3: Implementing Polymorphism in PHP

Objective:

Learn how to implement **polymorphism** in PHP.

Steps:

1. **Create an Interface (File: Discountable.php):**
    - Define an interface Discountable with a method getDiscount().
    - This interface will enforce that any class implementing it must have a getDiscount() method.
2. **Implement the Interface in Classes (File: Book.php):**
    - Modify the Book class to implement the Discountable interface.
    - Implement the getDiscount() method to calculate and return the discount for a book.
3. **Test Polymorphism (File: test_polymorphism.php):**
    - In test_polymorphism.php, create instances of Book and other product classes (like Electronics) and call their getDiscount() methods.

Instructions:

- Define the Discountable interface in Discountable.php.
- Implement the interface in Book.php and any other class, such as Electronics, with their own getDiscount() methods.
- Demonstrate polymorphism in test_polymorphism.php by invoking the getDiscount() method on multiple objects of different classes.

Questions:

1. What is polymorphism, and why is it important in OOP?
2. What is the difference between **method overloading** and **method overriding**?
3. How does PHP support polymorphism via interfaces and method overriding?

Exercise 4: Building a Library System Using OOP

Objective:

Combine all the learned OOP concepts (classes, objects, inheritance, and polymorphism) into a complete project.

Steps:

1. **Database Setup:**
    - Create a database called **LibraryDB** with tables like Books (with fields: book_id, title, author, price, genre, year) and Members (with fields: member_id, name, email, membership_date).
    - Create another table BookLoans to track which member borrows which book.
2. **Create the Classes:**
    - Create a Book class (File: Book.php) with properties: title, author, price, and genre, and methods like borrowBook() and returnBook().
    - Create a Member class (File: Member.php) with properties: name, email, and membership_date, and methods to view borrowed books.
3. **Inheritance and Polymorphism:**
    - Create a Loanable interface (File: Loanable.php) that contains the borrowBook() and returnBook() methods.
    - Implement the interface in Book.php.
    - Create an Ebook subclass (File: Ebook.php) that extends Book and adds functionality specific to eBooks (e.g., download()). This class should implement the getDiscount() method from Discountable interface for discounted pricing.
4. **Connecting PHP with MySQL:**

- Use mysqli or PDO in PHP to interact with the LibraryDB database.
- Create a script to display books available in the library and allow members to borrow and return books.

5. **Testing the Application:**
    - Create a test page (File: library_test.php) that allows members to borrow and return books, and shows which books are currently borrowed.

Instructions:

- Create the Book and Member classes, implement inheritance and polymorphism, and interact with the LibraryDB database to manage book loans.
- Display borrowed books and allow users to borrow and return books from the LibraryDB.

Project File Names:

- Book.php — Define the Book class and its methods.
- Member.php — Define the Member class and its methods.
- Loanable.php — Define the Loanable interface.
- Discountable.php — Define the Discountable interface.
- Ebook.php — Define the Ebook class, a subclass of Book.
- library_test.php — Create a testing page for the Library System.

Lab 5: Building Real-World Web Applications with PHP and MySQL

Objective

By the end of **Lab 5**, students will:

1. Integrate **PHP** with **MySQL** to build a fully functional web application.
2. Implement **user authentication** and restrict access to pages.
3. Implement **Google OAuth authentication** and ensure users can access only authorized pages.
4. Use **CRUD functionality** to manage records in a database.

Exercise 1: User Authentication with MySQL

Objective:

Ensure that only authenticated users can access the application. Users will need to log in to access the library system.

Steps:

1. **Database Setup:**
    - **Ensure the users table is set up** in the LibraryDB database as in **Lab 4**. The users table should include columns: id, username, email, and password.
    - If needed, you can also create a new database LibraryDB and the users table from scratch.
2. **Create the Registration Page (File: register.php):**
    - This page will allow users to register with a username, email, and password.
    - Use **password hashing** (via password_hash()) to store the password securely.
    - Insert the user's registration details into the users table.
3. **Create the Login Page (File: login.php):**
    - This page will allow users to log in with their username and password.
    - Use password_verify() to compare the entered password with the stored hashed password.

- On successful login, create a session for the user (store user info in $_SESSION).
- Redirect authenticated users to the **home page**.

4. **Logout Page (File: logout.php):**
    - Destroy the user session to log the user out.
    - Redirect the user back to the login page after logging out.
5. **Authentication Check (File: auth_check.php):**
    - Include this file on every page that needs to be restricted to logged-in users.
    - The auth_check.php file will verify if a user is logged in. If not, it will redirect the user to login.php.

Questions:

1. How do you store user passwords securely in a database? Why is password hashing important?
2. Why is **session management** important in PHP authentication systems?
3. How would you modify the login system to allow multiple users to log in at the same time without overwriting each other's session?

Exercise 2: Integrating Google OAuth Authentication

Objective:

Allow users to authenticate with **Google OAuth** in addition to the regular login system. This will allow users to log in with their Google account.

Steps:

1. **Set Up Google OAuth:**
    - Go to the Google Developer Console, create a new project, and enable **Google+ API**.
    - Generate **OAuth 2.0 credentials** (Client ID and Client Secret).
    - Download the credentials file and include it in your project securely.
2. **Install Google API Client (File: composer.json):**

- Use **Composer** to install the Google API client by running the following command in your project directory:
- composer require google/apiclient:^2.0

3. **Google Login Page (File: google_login.php):**
   - Create a button that allows users to log in with their Google account.
   - Use the **Google API Client** to authenticate users via OAuth.
   - Once authenticated, retrieve user information such as name, email, etc.
4. **Store Google User Data (File: google_auth.php):**
   - Save the authenticated Google user's details (such as name, email) into the users table if they are new. If they are a returning user, simply log them in using Google OAuth.
5. **Redirect to Home (File: home.php):**
   - Once the user is logged in (via either the standard login system or Google OAuth), redirect them to the **home page** where they can see library details and their account information.
6. **Display User Profile (File: profile.php):**
   - Display the user's profile, including their Google profile details (if authenticated via Google).

Questions:

1. How does **OAuth** enhance the security and usability of your web application?
2. How do you manage **multiple authentication methods** (Google and traditional login) in a single application?
3. What are the security risks of using **OAuth** for user authentication, and how can you mitigate them?

Exercise 3: Restricting Access to Pages Behind Authentication

Objective:

Ensure that only authenticated users can access pages like the library catalog, book borrowing, and profile. All pages should be protected by the authentication system.

Steps:

1. **Authentication Check (File: auth_check.php):**
    - Create a function auth_check() that checks if the user is logged in by verifying the $_SESSION variable.
    - If the user is not logged in, redirect them to login.php.
2. **Home Page (File: home.php):**
    - This page should be publicly accessible and provide an option to either log in (if the user is not logged in) or go to the library if they are authenticated.
    - Display a message welcoming the logged-in user (either via username/password login or Google OAuth).
3. **Library Page (File: library.php):**
    - This page should only be accessible by authenticated users.
    - Display all available books in the library and allow users to borrow or return books.
4. **Access Control (File: access_control.php):**
    - Ensure that all pages except home.php, login.php, and register.php are protected by the auth_check.php function.
5. **Test the Application:**
    - Test the login system by logging in via traditional credentials and Google OAuth.
    - Test page restrictions by ensuring that only authenticated users can access protected pages.

Questions:

1. What are the benefits of using **session-based authentication** in web applications?
2. How can you prevent **unauthorized access** to pages and protect sensitive data in a PHP web application?
3. How can you handle **session expiration** and ensure that users are logged out after a certain period of inactivity?

Exercise 4: PHP CRUD Application for Book Management (basically lab 4)

Objective:

Develop a **PHP CRUD** (Create, Read, Update, Delete) application for managing books in the library system. Ensure that only authenticated users can perform these actions.

Steps:

1. **Database Setup (File: db_setup.php):**
    - Ensure the Books table exists in LibraryDB with fields like book_id, title, author, price, genre, and year.
2. **Create Book (File: add_book.php):**
    - Create a form that allows users to add a new book to the library.
    - Ensure that only authenticated users can add books.
3. **Read Books (File: view_books.php):**
    - Display all books from the Books table.
    - Only authenticated users should be able to view books.
4. **Update Book (File: edit_book.php):**
    - Allow authenticated users to edit the details of a book.
    - Pre-populate the form with existing book details.
5. **Delete Book (File: delete_book.php):**
    - Allow authenticated users to delete books from the Books table.
6. **Test the CRUD Operations:**
    - Test adding, viewing, updating, and deleting books to ensure all CRUD functionalities are working.

Questions:

1. Why is it important to validate user input before performing **CRUD operations** on a database?
2. How do you ensure that **CRUD operations** are only accessible to authenticated users?
3. What security measures should be taken when allowing users to **edit** or **delete** data in a database?

Project File Names:

- auth_db.php — Set up the users table in MySQL for authentication.
- register.php — User registration page.
- login.php — User login page.
- logout.php — User logout script.
- google_login.php — Google OAuth login page.
- google_auth.php — Handle Google authentication.
- home.php — Home page accessible to all users.
- library.php — Library page with CRUD functionality.
- profile.php — User profile page.
- add_book.php — Form to add new books.
- view_books.php — Display all books.
- edit_book.php — Form to edit existing books.
- delete_book.php — Delete book script.
- auth_check.php — Authentication check function.
- db_setup.php — Database setup and configuration.

Lab 6: Ensure Web Application Security

Objective

By the end of **Lab 6**, students will:

1. Understand common web application security threats like **SQL injection**, **Cross-Site Scripting (XSS)**, and **Cross-Site Request Forgery (CSRF)**.
2. Learn how to **prevent SQL injection** and **XSS** by sanitizing and validating user input.
3. Implement **CSRF protection** in the application.
4. Host the PHP web application on a computer and access it from another computer on the same network.

Exercise 1: Preventing SQL Injection

Objective:

Learn how to protect your application from **SQL injection** attacks by using **prepared statements** in PHP.

Steps:

1. **Review Vulnerable Code (File: add_book.php):**
    - In **Lab 5**, the application allows users to add books to the library using a form. Review the code where the form data is inserted into the Books table.
    - Identify the areas where **SQL injection** could potentially occur if user inputs are not sanitized properly.
2. **Implement Prepared Statements (File: add_book.php):**
    - Modify the existing SQL queries in **add_book.php** to use **prepared statements** with **bound parameters** instead of directly inserting user input into the query.
    - Use mysqli or PDO to create prepared statements:
        - **Example using mysqli:**
        - $stmt = $conn->prepare("INSERT INTO Books (title, author, price, genre, year) VALUES (?, ?, ?, ?, ?)");
        - $stmt->bind_param("sssdi", $title, $author, $price, $genre, $year);

        - $stmt->execute();

3. **Test SQL Injection Protection:**
    - Test the add_book.php form by trying to enter malicious SQL code in the input fields (e.g., '; DROP TABLE Books; --). Ensure that the application prevents SQL injection and does not execute harmful SQL commands.

Questions:

1. Why is **SQL injection** a threat to web applications, and how do prepared statements prevent it?
2. How do **bound parameters** in prepared statements ensure that user input is handled securely?
3. What are other methods, besides prepared statements, to prevent SQL injection?

Exercise 2: Preventing Cross-Site Scripting (XSS)

Objective:

Learn how to protect your application from **Cross-Site Scripting (XSS)** attacks by **escaping output**.

Steps:

1. **Review Vulnerable Code (File: view_books.php):**
    - In **Lab 5**, the view_books.php page displays user-submitted data (such as book title, author, etc.) directly on the page.
    - Review this page and identify where **XSS vulnerabilities** could exist if user input is not properly sanitized before displaying it.
2. **Escape Output (File: view_books.php):**
    - Use **PHP's htmlspecialchars() function** to escape user input before displaying it on the page.
        - Example:
        - echo htmlspecialchars($row['title'], ENT_QUOTES, 'UTF-8');
3. **Test XSS Protection:**

- Test the application by submitting data containing malicious JavaScript (e.g., <script>alert('XSS');</script>).
- Ensure that the input is displayed as plain text and does not execute as JavaScript.

Questions:

1. What is **XSS** and why is it a security risk for web applications?
2. How does **escaping output** prevent XSS attacks?
3. What other techniques can be used to mitigate XSS vulnerabilities besides htmlspecialchars()?

Exercise 3: Implementing CSRF Protection

Objective:

Learn how to protect your web application from **Cross-Site Request Forgery (CSRF)** attacks by using **CSRF tokens**.

Steps:

1. **Review Vulnerable Code (File: add_book.php):**
   - Review the add_book.php form where users submit data to be inserted into the database. This form currently does not have any protection against CSRF attacks.
   - If a malicious website makes a POST request to this form on behalf of a logged-in user, a **CSRF attack** could occur.
2. **Generate CSRF Token (File: csrf_token.php):**
   - Create a script to generate a **CSRF token** and store it in the user's session.
     - Example:
     - if (empty($_SESSION['csrf_token'])) {
     - $_SESSION['csrf_token'] = bin2hex(random_bytes(32));
     - }
3. **Add CSRF Token to Forms (File: add_book.php):**

- Add a hidden field in the form to include the CSRF token:
- <input type="hidden" name="csrf_token" value="<?php echo $_SESSION['csrf_token']; ?>">

4. **Verify CSRF Token on Form Submission (File: process_book.php):**
    - In the script that processes the form submission (process_book.php), verify that the CSRF token in the form matches the one stored in the session:
    - if ($_POST['csrf_token'] !== $_SESSION['csrf_token']) {
    - die("CSRF token validation failed.");
    - }
5. **Test CSRF Protection:**
    - Test the application by trying to submit the form from another page on a different website (i.e., simulate a CSRF attack). The application should reject the request because the CSRF token will not match.

Questions:

1. What is **CSRF** and why is it a security concern for web applications?
2. How does **using CSRF tokens** protect against CSRF attacks?
3. What is the role of the **session** in CSRF protection?

Exercise 4: Hosting the Application on a Local Network

Objective:

Learn how to host the PHP application on one computer and access it from another computer on the same network.

Steps:

1. **Enable Apache and MySQL:**
    - Ensure that **XAMPP/WAMP/LAMP** is installed on the host computer.
    - Start the **Apache** and **MySQL** servers from the XAMPP/WAMP/LAMP control panel.
2. **Find the IP Address of the Host Computer:**
    - On the host computer, find its **local IP address**:

        - On Windows, open Command Prompt and type: ipconfig
        - On Linux/macOS, open Terminal and type: ifconfig
    - Note the **IPv4 Address** (e.g., 192.168.x.x).
3. **Access the Application from Another Computer:**
    - On another computer connected to the same network, open a web browser.
    - In the address bar, type the host computer's **local IP address** followed by the application's folder name (e.g., http://192.168.x.x/library).
    - The application should load, and you can interact with it as if it were hosted locally.
4. **Testing the Setup:**
    - Test the authentication system, Google login, and CRUD functionalities from the second computer to ensure that everything is working as expected.

Questions:

1. What is the difference between **localhost** and **IP address** in terms of hosting web applications?
2. How can you ensure that the application is accessible from other computers on the same network?
3. What are the security risks of making your application accessible on the network, and how can you mitigate them?

Final Project File Names:

- auth_db.php — Set up and configure the users table in MySQL for authentication.
- register.php — User registration page with password hashing.
- login.php — User login page with password verification.
- logout.php — User logout script.
- google_login.php — Google OAuth login page.
- google_auth.php — Handle Google authentication.
- home.php — Home page with login/logout functionality.

- library.php — CRUD functionality to manage books (only accessible to authenticated users).
- add_book.php — Form to add new books (protected by CSRF tokens).
- view_books.php — Display all books from the database.
- edit_book.php — Form to edit book details.
- delete_book.php — Script to delete books from the database.
- csrf_token.php — Script to generate CSRF tokens.
- auth_check.php — Authentication check function for page access control.
- db_setup.php — Database setup and configuration.
- test_security.php — Test page for SQL injection, XSS, and CSRF vulnerabilities.

Instructions for the Final Projects:

1. **Collaboration**: Students will work in groups of no more than **3,** each group is to select one project. Each group must contribute equally and demonstrate their understanding of the course content.
2. **Technologies**: Students must use **PHP**, **MySQL**, **HTML/CSS**, and **JavaScript**. They are also required to use **Google OAuth** for authentication in at least one of the projects.
3. **Security**: Ensure that all applications are **secure** by following best practices, including protection against **SQL injection**, **Cross-Site Scripting (XSS)**, **Cross-Site Request Forgery (CSRF)**, and **session management**.
4. **OOP**: The projects should include an **object-oriented approach (OOP)** for managing business logic. Use **classes**, **objects**, **inheritance**, and **polymorphism** where applicable.
5. **Deliverables**: At the end of the course, students will present their projects with a live demo and submit the following:
    - **Source code** (with comments).
    - **Documentation**: A README file that explains the project, technologies used, instructions for setup, and how to run it locally.
    - **Security report**: A report detailing how security threats like **SQL injection**, **XSS**, and **CSRF** were mitigated.

Project 1: Online Bookstore Management System

Summary:

An **Online Bookstore** where users can browse and purchase books, and admins can manage the bookstore's inventory.

Key Features:

- **User Authentication**: Users can register and log in (with Google OAuth as an optional authentication method).
- **Admin Panel**: Admins can perform CRUD operations on books (Add, Edit, Delete).

- **Shopping Cart**: Users can add books to their cart, checkout, and view their order history.
- **Security Measures**: Prevent **SQL injection**, **XSS**, and **CSRF** using appropriate validation, prepared statements, and secure password storage.

Complexity:

This project will involve **user authentication**, **shopping cart functionality**, **admin dashboard**, and **secure payment processing** (simulated).

Project 2: Movie Ticket Booking System

Summary:

A **Movie Ticket Booking** platform where users can search for movies, view showtimes, and book tickets, while admins manage the movie listings and showtimes.

Key Features:

- **User Authentication**: Secure login and registration, with the option to authenticate using **Google OAuth**.
- **Admin Panel**: Admins can add/edit movie listings, manage showtimes, and view ticket bookings.
- **Booking System**: Users can book tickets based on showtimes, check available seats, and make payments (simulated).
- **Security Measures**: Prevent **SQL injection**, **XSS**, and **CSRF**.

Complexity:

This project will require the use of **CRUD operations** for managing movies and tickets, along with user authentication and a **searchable movie database**.

Project 3: Online Learning Management System (LMS)

Summary:

An **Online Learning Management System (LMS)** where students can enroll in courses, take quizzes, and track their progress, while instructors manage the courses.

Key Features:

- **User Authentication**: Separate roles for students and instructors, authenticated with **PHP/MySQL** or **Google OAuth**.
- **Course Management**: Instructors can add, edit, and delete courses, as well as upload learning materials.
- **Quizzes and Assignments**: Students can take quizzes and assignments related to their enrolled courses.
- **Progress Tracker**: Students can track their learning progress and view completed courses.
- **Security Measures**: Prevent **SQL injection**, **XSS**, and **CSRF**.

Complexity:

The project will involve **role-based authentication**, **course management**, **quizzes**, and **progress tracking**. Security is crucial to ensure the integrity of quiz submissions and student data.

Project 4: Social Media Platform

Summary:

A **Social Media Platform** where users can post updates, like posts, and comment on others' posts, while admins manage user activity.

Key Features:

- **User Authentication**: Users can register, log in, and update profiles, with **Google OAuth** as an option.
- **Post and Comment System**: Users can create posts, like posts, and comment on posts.
- **Admin Panel**: Admins can manage users and posts, and monitor reports.
- **Security Measures**: Protect against **SQL injection**, **XSS**, **CSRF**, and ensure secure authentication.

Complexity:

This project will involve **user authentication**, **CRUD functionality** for posts and comments, and role-based access for admins. The security of user data and content is key.

Project 5: Job Portal System

Summary:

A **Job Portal System** where employers can post job openings and candidates can apply for jobs. The system includes an admin panel for managing job posts and applications.

Key Features:

- **User Authentication**: Separate roles for employers and job seekers, with **Google OAuth** as an optional authentication method.
- **Job Management**: Employers can post job openings, while job seekers can search and apply for jobs.
- **Admin Panel**: Admins can manage job posts, monitor applications, and review user activity.
- **Security Measures**: **SQL injection** prevention, **XSS** protection, and **CSRF** mitigation.

Complexity:

This project involves **role-based authentication**, **job posting** and **application management**, and **admin control** over job postings and applications.

Project 6: E-commerce Platform with Multi-User Roles

Summary:

A fully functional **E-commerce Platform** where customers can browse products, add products to a shopping cart, and make purchases, while admins can manage the product inventory, track orders, and manage user accounts.

Key Features:

- **User Authentication**: Users can register and log in, with **Google OAuth** as an optional login method.
- **Product Management**: Admins can add, edit, and delete products, and manage categories.
- **Shopping Cart**: Customers can add items to the cart and proceed to checkout.

- **Order Management**: Customers can track their orders and view order history.
- **Security Measures**: Use **prepared statements** to prevent **SQL injection**, sanitize user input to avoid **XSS**, and implement **CSRF** tokens for form submissions.

Complexity:

This project will require implementation of **user roles** (admin vs customer), **order tracking**, **shopping cart functionality**, and **secure payment processing** (simulated). It will also involve **advanced security features** to protect both user and transaction data.

Summary of Final Projects:

1. **Online Bookstore Management System**: E-commerce platform for books with shopping cart and admin panel.
2. **Movie Ticket Booking System**: Movie booking platform with CRUD for movies and showtimes.
3. **Online Learning Management System (LMS)**: Learning platform with courses, quizzes, and progress tracking.
4. **Social Media Platform**: Social network where users can post, comment, and like posts.
5. **Job Portal System**: Job search and application platform with separate roles for job seekers and employers.
6. **E-commerce Platform with Multi-User Roles**: Full-fledged online store with customer and admin roles.

Evaluation Criteria:

- **Functionality**: The project should work as expected with all required features and user flows.
- **Code Quality**: Use of **OOP** principles, clean code, proper error handling, and good practices in coding.
- **Security**: All projects must address key security concerns such as **SQL injection**, **XSS**, **CSRF**, and **session management**.

- **User Experience**: User-friendly interface and navigation, along with responsive design.
- **Documentation**: Comprehensive project documentation, including setup instructions, explanation of key features, and security measures.