Practical Session 1 – Working with Endpoints and Documentation

Practical Learning Outcomes

1. Create a basic API endpoint using PHP.
2. Test the endpoint using Postman.
3. Write simple endpoint documentation for the created API.

Practical Activities

Activity 1: Creating a Simple API Endpoint in PHP

1. **Step 1: Set Up the Environment**
   - Ensure that **XAMPP** is running with Apache.
   - Create a new file in your local server directory (e.g., `htdocs`) named `api_example.php`.
2. **Step 2: Define an Endpoint**
   - Add the following PHP code to handle GET requests:

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

if ($_SERVER['REQUEST_METHOD'] === 'GET' && isset($_GET['endpoint']) &&
$_GET['endpoint'] === 'user') {
    $userId = $_GET['id'] ?? null;

    if ($userId) {
        echo json_encode([
            "id" => $userId,
            "name" => "John Doe",
            "email" => "johndoe@example.com"
        ]);
    } else {
        echo json_encode([
            "error" => "User ID is required."
        ]);
    }
} else {
    echo json_encode([
        "error" => "Invalid endpoint or method."
    ]);
}
?>
```

3. **Step 3: Test the Endpoint in the Browser**
   - Open your browser and navigate to: `http://localhost/api_example.php?endpoint=user&id=123`
   - Verify that the JSON response is displayed.

Activity 2: Testing the Endpoint Using Postman

1. **Step 1: Open Postman**
   - If Postman is not installed, download and install it first.
2. **Step 2: Make a GET Request**
   - In Postman, create a new request and set it to `GET`.
   - Enter the URL: `http://localhost/api_example.php?endpoint=user&id=123`.
3. **Step 3: Send the Request**
   - Click **Send** and observe the response in the output panel.
4. **Task:**
   - Modify the URL in Postman to exclude the `id` parameter and check the error response.

Activity 3: Documenting the Endpoint

1. **Step 1: Create a Basic Documentation Template**
   - Open any text editor (or use Notepad, Word, etc.) and create the following documentation:

~~~
# User Retrieval Endpoint Documentation

## Endpoint
`/user`

## HTTP Method
`GET`

## Parameters
- `id` (Required): The ID of the user to retrieve.

## Response
- **200 OK** (When ID is provided):
  ```json
  {
      "id": "123",
      "name": "John Doe",
      "email": "johndoe@example.com"
  }
~~~

   - **400 Bad Request** (When ID is missing):

```
{
    "error": "User ID is required."
}
```

Example Request

```
GET http://localhost/api_example.php?endpoint=user&id=123
```

Example Response

```
{
    "id": "123",
    "name": "John Doe",
    "email": "johndoe@example.com"
}
```

2. **Step 2: Save and Share the Documentation**

- Save the documentation as a `.txt` or `.md` file (e.g., `user_endpoint_doc.md`).
- Share it with your classmates for feedback.

Activity 4: Expanding the API

1. **Step 1: Add a New Endpoint**
   - Extend the `api_example.php` file to include a `/product` endpoint:

```
if ($_SERVER['REQUEST_METHOD'] === 'GET' && isset($_GET['endpoint']) &&
$_GET['endpoint'] === 'product') {
    $productId = $_GET['id'] ?? null;

    if ($productId) {
        echo json_encode([
            "id" => $productId,
            "name" => "Sample Product",
            "price" => 19.99
        ]);
    } else {
        echo json_encode([
            "error" => "Product ID is required."
        ]);
    }
}
```

2. **Step 2: Test the New Endpoint**
   - Use Postman to test the `/product` endpoint.
   - Example URL:
     `http://localhost/api_example.php?endpoint=product&id=456`
3. **Step 3: Update Documentation**
   - Add details about the new `/product` endpoint to your existing documentation.

Tasks for Students:

- Implement a POST, PUT, PATCH and DELETE endpoints that accepts an ID as input and returns a success message after simulating each operation.
- Example response for delete operation:

```
{
   "status": "success",
   "message": "Record with ID 5 deleted successfully."
}
```

Practical Session 2 – Implementing Sessions and Cookies in PHP

Practical Learning Outcomes

1. Use PHP to create and manage sessions.
2. Use PHP to create, read, and delete cookies.
3. Demonstrate state management by overcoming statelessness with sessions and cookies.

Practical Activities

Activity 1: Implementing Sessions in PHP

1. **Step 1: Start a Session**
   - Create a file named `session_example.php` and add the following code:

```
<?php
// Start a new session
session_start();

// Check if a session variable is already set
if (!isset($_SESSION['username'])) {
    $_SESSION['username'] = "JohnDoe";
    echo "Session started and username set to: JohnDoe";
} else {
    echo "Welcome back, " . $_SESSION['username'];
}
?>
```

2. **Step 2: Test the Session**
   - Open the file in your browser (e.g., `http://localhost/session_example.php`).
   - Refresh the page and observe how the session persists.
3. **Step 3: Destroy the Session**
   - Add the following code to another file (e.g., `session_destroy.php`):

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

// Destroy session
session_destroy();
echo "Session destroyed. Reload to create a new session.";
?>
```

   - Open `session_destroy.php` and test how the session resets.

Activity 2: Using Cookies in PHP

1. **Step 1: Set a Cookie**
   - Create a file named `cookie_example.php` and add the following code:

```
<?php
// Set a cookie named "user" that expires in 1 hour
setcookie("user", "JaneDoe", time() + 3600, "/");

echo "Cookie has been set. Reload to check its value.";
?>
```

2. **Step 2: Access the Cookie**
   - Add the following code to the same file:

```
if (isset($_COOKIE['user'])) {
    echo "Hello, " . $_COOKIE['user'];
} else {
    echo "Cookie not found.";
}
```

3. **Step 3: Delete the Cookie**
   - Add the following code to delete the cookie:

```
// Delete the cookie by setting its expiration time in the past
setcookie("user", "", time() - 3600, "/");

echo "Cookie has been deleted.";
```

Activity 3: Overcoming Statelessness with Sessions and Cookies

1. **Scenario**:
   Build a simple login system to demonstrate how sessions and cookies help maintain user state.
2. **Step 1: Create a Login Form**
   - Create a file named `login.php`:

```
<form method="POST" action="login_process.php">
    <label for="username">Username:</label>
    <input type="text" name="username" required>
    <label for="remember">Remember Me:</label>
    <input type="checkbox" name="remember">
    <button type="submit">Login</button>
</form>
```

3. **Step 2: Process the Login**
   - Create a file named `login_process.php`:

```
<?php
session_start();

// Capture the username
$username = $_POST['username'];

// Store username in session
$_SESSION['username'] = $username;

// Set a cookie if "Remember Me" is checked
if (isset($_POST['remember'])) {
    setcookie("username", $username, time() + 3600, "/");
}

echo "Welcome, " . $username . "! Go to your <a
href='dashboard.php'>dashboard</a>.";
?>
```

4. **Step 3: Create a Dashboard**
   - Create a file named `dashboard.php`:

```
<?php
session_start();

if (isset($_SESSION['username'])) {
    echo "Hello, " . $_SESSION['username'] . ". Welcome to your
dashboard.";
} elseif (isset($_COOKIE['username'])) {
    echo "Hello, " . $_COOKIE['username'] . ". Welcome to your dashboard.";
} else {
    echo "You are not logged in. <a href='login.php'>Login</a>";
}
?>
```

5. **Step 4: Logout and Clear Session/Cookie**
   - Create a file named `logout.php`:

```
<?php
session_start();

// Clear session
session_destroy();

// Clear cookie
setcookie("username", "", time() - 3600, "/");

echo "You have been logged out. <a href='login.php'>Login again</a>";
?>
```

Tasks for Students: Build a Login System using Sessions and Cookies:

- Create a simple login form that accepts a username and password
- On successful login, store the user's data in a session and set a cookie for "remember me" functionality
- Implement a "logout" button to destroy the session and delete the cookie.

Practical Session 3 – Implementing Endpoint Security, Authentication, and Authorization in PHP

Practical Learning Outcomes

1. Implement secure endpoint handling in PHP.
2. Mitigate vulnerabilities like SQL injection, XSS, and CSRF.
3. Use JWT for authentication.
4. Implement role-based authorization with roles and permissions.
5. Use middleware for request validation and logging.

Practical Activities

Activity 1: Securing Against SQL Injection

1. **Step 1: Create a Database Connection**
   - Create a file named `db.php`:

```
<?php
$host = "localhost";
```

```
$username = "root";
$password = "";
$database = "test_db";

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

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

2. **Step 2: Use Prepared Statements to Prevent SQL Injection**
   - Create a file named `secure_query.php`:

```
<?php
include 'db.php';

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

// Secure query
$stmt = $conn->prepare("SELECT * FROM users WHERE username = ? AND password
= ?");
$stmt->bind_param("ss", $username, $password);

$stmt->execute();
$result = $stmt->get_result();

if ($result->num_rows > 0) {
    echo "Login successful!";
} else {
    echo "Invalid credentials.";
}
?>
```

Activity 2: Securing Against XSS

1. **Step 1: Create a Form to Accept Input**
   - Create a file named `xss_form.php`:

```
<form method="POST" action="xss_handler.php">
    <label for="comment">Comment:</label>
    <textarea name="comment" id="comment"></textarea>
    <button type="submit">Submit</button>
</form>
```

2. **Step 2: Escape Input to Prevent XSS**
   - Create a file named `xss_handler.php`:

```
<?php
$comment = htmlspecialchars($_POST['comment'], ENT_QUOTES, 'UTF-8');

echo "Your comment: " . $comment;
?>
```

Activity 3: Securing Against CSRF

1. **Step 1: Generate a CSRF Token**
   - Create a file named `csrf_form.php`:

```
<?php
session_start();

// Generate CSRF token
if (empty($_SESSION['csrf_token'])) {
    $_SESSION['csrf_token'] = bin2hex(random_bytes(32));
}
?>
<form method="POST" action="csrf_handler.php">
    <input type="hidden" name="csrf_token" value="<?php echo
$_SESSION['csrf_token']; ?>">
    <label for="data">Data:</label>
    <input type="text" name="data" id="data">
    <button type="submit">Submit</button>
</form>
```

2. **Step 2: Validate the CSRF Token**
   - Create a file named `csrf_handler.php`:

```
<?php
session_start();

if ($_POST['csrf_token'] === $_SESSION['csrf_token']) {
    echo "Valid request!";
} else {
    echo "CSRF detected!";
}
?>
```

Activity 4: Using JWT for Authentication

1. **Step 1: Install JWT Library**
   - Use Composer to install the Firebase JWT library (if Composer is not available, download the library manually):

```
composer require firebase/php-jwt
```

2. **Step 2: Generate a JWT**
   - Create a file named `jwt_generate.php`:

```
<?php
require 'vendor/autoload.php';

use Firebase\JWT\JWT;
use Firebase\JWT\Key;

$key = "your_secret_key";
$payload = [
    "user_id" => 123,
    "role" => "admin",
    "iat" => time(),
    "exp" => time() + 3600
];

$jwt = JWT::encode($payload, $key, 'HS256');
echo "JWT: " . $jwt;
?>
```

3. **Step 3: Validate a JWT**
   - Create a file named `jwt_validate.php`:

```php
<?php
require 'vendor/autoload.php';

use Firebase\JWT\JWT;
use Firebase\JWT\Key;

$key = "your_secret_key";

$jwt = $_POST['jwt']; // Received from client
try {
    $decoded = JWT::decode($jwt, new Key($key, 'HS256'));
    echo "User ID: " . $decoded->user_id;
} catch (Exception $e) {
    echo "Invalid token: " . $e->getMessage();
}
?>
```

Activity 5: Role-Based Authorization

1. **Step 1: Implement Authorization Logic**
   - Add the following logic in a file named `authorization.php`:

```php
<?php
session_start();

$_SESSION['role'] = "user"; // Example role

function authorize($required_role) {
    if ($_SESSION['role'] !== $required_role) {
        die("Access Denied!");
    }
}

// Check if user has admin access
authorize("admin");
echo "Welcome, Admin!";
?>
```

Activity 6: Implementing Middleware

1. **Step 1: Create Middleware for Request Validation**
   - Add the following middleware function in a file named `middleware.php`:

```php
<?php
function validate_request() {
    if (empty($_SERVER['HTTP_AUTHORIZATION'])) {
        die("Unauthorized request");
    }
}
?>
```

2. **Step 2: Use Middleware**
   - Include the middleware in your endpoint:

```php
<?php
```

```
include 'middleware.php';

validate_request();

echo "Request is valid!";
?>
```

Tasks for Students

1. **Implement password Hashing:**
    - Create a script that stores hashed passwords in the database using `password_hash()`. Validate passwords during login using `password_verify()`.
2. **Implement CSRF Protection:**
    - Generate and validate CSRF tokens for forms to prevent Cross-Site Request Forgery.
3. **API Security:**
    - Secure API endpoints using HTTPS and restrict access to specific IP ranges.

Practical Session 4 – Database Connection and Query Execution

Practical Learning Outcomes

1. Establish a database connection using MySQLi and PDO.
2. Execute basic queries to insert, retrieve, update, and delete data.
3. Retrieve data into a result set and traverse it using loops.

Practical Activities

Activity 1: Establishing a Database Connection

Step 1: Create a Database

1. Open **phpMyAdmin** or any database management tool.
2. Create a database named `test_db`.
3. Create a table named `users` with the following structure:
    - `id` (Primary Key, Auto Increment, Integer)
    - `name` (VARCHAR(100))
    - `email` (VARCHAR(100))

Step 2: Connect to the Database Using MySQLi

1. Create a file named `mysqli_connect.php`:

```
<?php
$host = "localhost";
$username = "root";
$password = "";
$database = "test_db";

// Create connection
$conn = mysqli_connect($host, $username, $password, $database);

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

2. Open the file in the browser to test the connection.

Step 3: Connect to the Database Using PDO

1. Create a file named `pdo_connect.php`:

```
<?php
$dsn = "mysql:host=localhost;dbname=test_db";
$username = "root";
$password = "";

try {
    $conn = new PDO($dsn, $username, $password);
    $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    echo "Connected successfully using PDO!";
} catch (PDOException $e) {
    echo "Connection failed: " . $e->getMessage();
}
?>
```

2. Open the file in the browser to test the connection.

Activity 2: Executing Basic Queries

Step 1: Insert Data into the `users` Table

1. **Using MySQLi**:
   Create a file named `mysqli_insert.php`:

```
<?php
include 'mysqli_connect.php';

$sql = "INSERT INTO users (name, email) VALUES ('John Doe', 'john@example.com')";

if (mysqli_query($conn, $sql)) {
    echo "New record created successfully!";
} else {
    echo "Error: " . $sql . "<br>" . mysqli_error($conn);
}
```

```
mysqli_close($conn);
?>
```

2. **Using PDO**:
   Create a file named `pdo_insert.php`:

```
<?php
include 'pdo_connect.php';

try {
    $sql = "INSERT INTO users (name, email) VALUES ('Jane Doe',
'jane@example.com')";
    $conn->exec($sql);
    echo "New record created successfully!";
} catch (PDOException $e) {
    echo "Error: " . $e->getMessage();
}
?>
```

Step 2: Retrieve Data from the `users` Table

1. **Using MySQLi**:
   Create a file named `mysqli_select.php`:

```
<?php
include 'mysqli_connect.php';

$sql = "SELECT * FROM users";
$result = mysqli_query($conn, $sql);

if (mysqli_num_rows($result) > 0) {
    while ($row = mysqli_fetch_assoc($result)) {
        echo "ID: " . $row["id"] . " - Name: " . $row["name"] . " - Email: " .
$row["email"] . "<br>";
    }
} else {
    echo "No results found.";
}

mysqli_close($conn);
?>
```

2. **Using PDO**:
   Create a file named `pdo_select.php`:

```
<?php
include 'pdo_connect.php';

try {
    $stmt = $conn->query("SELECT * FROM users");
    foreach ($stmt as $row) {
        echo "ID: " . $row["id"] . " - Name: " . $row["name"] . " - Email: " .
$row["email"] . "<br>";
    }
} catch (PDOException $e) {
    echo "Error: " . $e->getMessage();
}
?>
```

Activity 3: Traversing the Result Set

Step 1: Use a Loop to Display Data

1. Modify the `mysqli_select.php` file to include better formatting:

   ```
   while ($row = mysqli_fetch_assoc($result)) {
       echo "<p><strong>ID:</strong> " . $row["id"] . " | <strong>Name:</strong> " .
   $row["name"] . " | <strong>Email:</strong> " . $row["email"] . "</p>";
   }
   ```

2. Similarly, modify the `pdo_select.php` file to use loops for better formatting:

   ```
   foreach ($stmt as $row) {
       echo "<p><strong>ID:</strong> " . $row["id"] . " | <strong>Name:</strong> " .
   $row["name"] . " | <strong>Email:</strong> " . $row["email"] . "</p>";
   }
   ```

Activity 4: Error Handling in Database Queries

1. Add error handling for MySQLi queries:

   ```
   $result = mysqli_query($conn, $sql);

   if (!$result) {
       die("Query failed: " . mysqli_error($conn));
   }
   ```

2. Add error handling for PDO queries:

   ```
   try {
       $stmt = $conn->query($sql);
   } catch (PDOException $e) {
       echo "Error: " . $e->getMessage();
   }
   ```

Tasks for Students

1. **Create a Dynamic CRUD Application:**
   - Build a small application that allows users to create, read, update, and delete records from the `users` table using forms and PHP scripts.
2. **Handle Errors Gracefully:**
   - Modify your scripts to handle and log database connection or query errors for debugging purposes.
3. **Extend students tasks in practical session 1**
   - Implement database integration (refer back to student's tasks in practical session 1) for actual resource persistence. The code will need to interact with a database to handle CRUD (Create, Read, Update, and Delete) operations for users and products.

Practical Session 5 – Using cURL in PHP

Practical Learning Outcomes

1. Write PHP scripts to make GET and POST requests using cURL.
2. Fetch data from external APIs.
3. Send data to an external server using POST requests.
4. Handle errors during cURL execution.

Practical Activities

Activity 1: Making a GET Request Using cURL

Steps:

1. **Create a file named `curl_get.php`:**

```
<?php
// Initialize a cURL session
$ch = curl_init();

// Set cURL options
curl_setopt($ch, CURLOPT_URL, "https://jsonplaceholder.typicode.com/posts/1"); //
API endpoint
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true); // Return the response as a
string

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

// Check for errors
if (curl_errno($ch)) {
    echo "Error: " . curl_error($ch);
} else {
    echo "<pre>";
    print_r(json_decode($response, true)); // Decode and display JSON response
    echo "</pre>";
}

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

2. **Run the Script in a Browser**:
   Open the file in your browser to see the response data from the endpoint.

Activity 2: Making a POST Request Using cURL

Steps:

1. **Create a file named `curl_post.php`:**

```php
<?php
// Initialize a cURL session
$ch = curl_init();

// Data to send in the POST request
$data = array(
    "title" => "Test Post",
    "body" => "This is a test message.",
    "userId" => 1
);
$jsonData = json_encode($data);

// Set cURL options
curl_setopt($ch, CURLOPT_URL, "https://jsonplaceholder.typicode.com/posts");
curl_setopt($ch, CURLOPT_POST, true);
curl_setopt($ch, CURLOPT_POSTFIELDS, $jsonData);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
curl_setopt($ch, CURLOPT_HTTPHEADER, array(
    "Content-Type: application/json",
    "Content-Length: " . strlen($jsonData)
));

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

// Check for errors
if (curl_errno($ch)) {
    echo "Error: " . curl_error($ch);
} else {
    echo "Response from Server:<br>";
    echo "<pre>";
    print_r(json_decode($response, true)); // Decode and display JSON response
    echo "</pre>";
}

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

2. **Run the Script in a Browser**:
   Open the file in your browser to see the response from the server confirming the POST request was successful.

Activity 3: Error handling in cURL

Steps:

1. **Create a file named `curl_error_handling.php`:**

```php
<?php
// Initialize a cURL session
$ch = curl_init();

// Set an invalid URL
curl_setopt($ch, CURLOPT_URL, "https://invalid-api-url.com/resource");
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);

// Execute the request
$response = curl_exec($ch);
```

```
// Check for errors
if (curl_errno($ch)) {
    echo "Error: " . curl_error($ch);
} else {
    echo "Response: " . $response;
}

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

2. **Run the Script in a Browser**:
   Observe how errors are displayed when the URL or connection fails.

Activity 4: Combine GET and POST Requests

Steps:

1. **Create a file named `curl_get_post_combined.php`:**

```
<?php
// Part 1: GET Request
$chGet = curl_init();
curl_setopt($chGet, CURLOPT_URL, "https://jsonplaceholder.typicode.com/posts/1");
curl_setopt($chGet, CURLOPT_RETURNTRANSFER, true);
$getResponse = curl_exec($chGet);
curl_close($chGet);

echo "GET Response:<br>";
echo "<pre>";
print_r(json_decode($getResponse, true));
echo "</pre>";

// Part 2: POST Request
$chPost = curl_init();
$postData = array("title" => "New Title", "body" => "New Body Content", "userId"
=> 1);
curl_setopt($chPost, CURLOPT_URL, "https://jsonplaceholder.typicode.com/posts");
curl_setopt($chPost, CURLOPT_POST, true);
curl_setopt($chPost, CURLOPT_POSTFIELDS, json_encode($postData));
curl_setopt($chPost, CURLOPT_RETURNTRANSFER, true);
curl_setopt($chPost, CURLOPT_HTTPHEADER, array("Content-Type:
application/json"));
$postResponse = curl_exec($chPost);
curl_close($chPost);

echo "POST Response:<br>";
echo "<pre>";
print_r(json_decode($postResponse, true));
echo "</pre>";
?>
```

2. **Run the Script in a Browser**:
   Observe the results of both GET and POST requests.

Tasks for Students

1. **Consume a REST API:**

- Use cURL to build a small PHP application that interacts with REST API, performing CRUD operations (e.g., GET, POST, PUT, DELETE).

2. **Test API Endpoints using Postman and cURL:**
    - Compare the responses of an API endpoint using Postman and cURL, and ensure they match.
3. **Extend students tasks in practical session 1**
    - Use **curl** to test the API (refer back to student's tasks in practical session 1).

Practical Session 6 – Git and Remote Repositories

Practical Learning Outcomes

By the end of this session, students will be able to:

1. Install and set up Git on their local system.
2. Initialize a local repository and perform basic Git operations.
3. Push a local repository to GitHub and sync changes.
4. Clone a remote repository and collaborate using Git.

Practical Activities

Activity 1: Install and Configure Git

Steps:

1. **Download Git**:
    - Visit Git Downloads and download the Git installer for your operating system.
2. **Install Git**:
    - Run the downloaded installer and follow the installation wizard.
    - Ensure the option to add Git to your system PATH is selected.
3. **Verify Git Installation**:
    - Open a terminal or command prompt and type:

    ```
    git --version
    ```

    - You should see the installed Git version.
4. **Configure Git**:
    - Set your username and email for Git:

    ```
    git config --global user.name "Your Name"
    git config --global user.email "your.email@example.com"
    ```

- Verify the configuration:

  ```
  git config --list
  ```

Activity 2: Initialize a Local Repository

Steps:

1. **Create a New Directory**:
   - Open your terminal or command prompt and run:

     ```
     mkdir my-first-repo
     cd my-first-repo
     ```

2. **Initialize Git**:
   - Run the following command to turn the directory into a Git repository:

     ```
     git init
     ```

3. **Create a File**:
   - Create a new file in the directory (e.g., `index.html`) and add some content:

     ```
     <h1>My First Git Repository</h1>
     ```

4. **Add the File to Staging**:
   - Run:

     ```
     git add index.html
     ```

5. **Commit the File**:
   - Save the changes to the repository:

     ```
     git commit -m "Initial commit: Add index.html"
     ```

6. **Check Repository Status**:
   - Run:

     ```
     git status
     ```

Activity 3: Create a Remote Repository on GitHub

Steps:

1. **Create a GitHub Account**:
   - Go to GitHub and create a free account if you don't already have one.
2. **Create a New Repository**:
   - Log in to GitHub, click on the **New Repository** button, and fill in the required details.
   - Do not initialize the repository with a README or .gitignore file.
3. **Connect Local and Remote Repositories**:
   - Copy the remote repository URL from GitHub.
   - Link the local repository to the remote repository:

```
git remote add origin <repository_url>
```

4. **Push Changes to GitHub**:
   - Upload your local commits to GitHub:

     ```
     git push -u origin main
     ```

5. **Verify on GitHub**:
   - Visit your GitHub repository URL to confirm the files were uploaded.

Activity 4: Clone a Remote Repository

Steps:

1. **Share the Repository URL**:
   - Provide students with the repository URL created in the previous activity.
2. **Clone the Repository**:
   - Run:

     ```
     git clone <repository_url>
     ```

   - This creates a copy of the remote repository on your local machine.
3. **Make Changes and Push**:
   - Open the cloned repository folder, modify a file, and commit the changes:

     ```
     git add <file_name>
     git commit -m "Update file content"
     git push
     ```

4. **Pull Changes from Remote**:
   - If changes are made to the remote repository by others, fetch and merge them:

     ```
     git pull
     ```

Activity 5: Collaborate Using Git

Steps:

1. **Fork a Repository**:
   - Demonstrate how to fork a repository from another GitHub account.
2. **Submit a Pull Request**:
   - Make changes in the forked repository and submit a pull request to the original repository.
3. **Resolve Merge Conflicts**:
   - Simulate a merge conflict by editing the same file from different branches and guide students through resolving it.

Tasks for Students

1. **Tagging and Releases**
   - Tag a specific commit for a release version:

```
git tag -a v1.0 -m "version 1.0 release"
git push origin v1.0
```

- **Deliverable:** Show the tag in the remote repository.

2. **Interactive Rebase**
   - Use git rebase to clean up commit history before pushing to the remote repository.

Practical Session 7 – Hands-On with Laravel (PHP Framework)

Practical Learning Outcomes

By the end of this session, students will be able to:

1. Install Laravel on their local machine.
2. Set up a basic Laravel project.
3. Understand the directory structure of a Laravel project.
4. Create a basic route and view in Laravel.

Practical Activities

Activity 1: Install Laravel

Steps:

1. **Ensure Development Environment is Ready**:
   - Verify that XAMPP is installed and running (Apache and MySQL services must be active).
   - Install **Composer**, the dependency manager for PHP:
     - Download Composer from getcomposer.org.
     - Run the installer and ensure it's added to the system PATH.
   - Verify Composer installation:

     ```
     composer --version
     ```
2. **Install Laravel via Composer**:
   - Open the terminal and navigate to the `htdocs` folder in your XAMPP directory:

     ```
     cd /path/to/htdocs
     ```
   - Install a new Laravel project using the Composer command:

     ```
     composer create-project laravel/laravel my-first-laravel-app
     ```
3. **Start the Laravel Development Server**:
   - Navigate to the project folder:

     ```
     cd my-first-laravel-app
     ```

- Start the development server:

  ```
  php artisan serve
  ```

- Open the browser and go to http://127.0.0.1:8000 to see the default Laravel welcome page.

Activity 2: Explore the Laravel Directory Structure

Steps:

1. **Explain Key Directories**:
   - Open the `my-first-laravel-app` folder in a code editor like VS Code.
   - Discuss the purpose of these folders:
     - `routes/`: Contains route files like `web.php` (used for defining web routes).
     - `resources/views/`: Contains Blade template files (used for views).
     - `app/Http/Controllers/`: Contains controller files (used to handle logic).
     - `public/`: Contains assets like images, CSS, and JavaScript.
2. **Locate Key Files**:
   - Explain the `composer.json` file for managing dependencies.
   - Discuss the `.env` file for environment variables, like database credentials.

Activity 3: Create a Basic Route and View

Steps:

1. **Define a Route**:
   - Open the `routes/web.php` file and add a new route:

     ```
     Route::get('/hello', function () {
         return view('hello');
     });
     ```

2. **Create a View**:
   - Inside the `resources/views/` folder, create a new file called `hello.blade.php`.
   - Add the following content:

     ```
     <!DOCTYPE html>
     <html>
     <head>
         <title>Hello Laravel</title>
     </head>
     <body>
         <h1>Welcome to Laravel Framework!</h1>
         <p>This is your first Laravel view.</p>
     </body>
     </html>
     ```

3. **Test the Route**:
   - Open your browser and go to http://127.0.0.1:8000/hello.
   - The "Hello Laravel" page should appear.

Activity 4: Add Dynamic Data to the View

Steps:

1. **Pass Data from the Route**:
   - Modify the route in `web.php`:

     ```
     Route::get('/hello', function () {
         return view('hello', ['name' => 'Student']);
     });
     ```

2. **Modify the View**:
   - Update `hello.blade.php` to display the dynamic data:

     ```
     <h1>Hello, {{ $name }}!</h1>
     ```

3. **Test the Dynamic View**:
   - Refresh the browser at http://127.0.0.1:8000/hello.
   - The page should now display "Hello, Student!".

Activity 5: Integrate a Controller

Steps:

1. **Create a Controller**:
   - Generate a new controller using the Artisan CLI:

     ```
     php artisan make:controller HelloController
     ```

   - A new file `HelloController.php` will be created in the `app/Http/Controllers/` directory.
2. **Add a Method to the Controller**:
   - Open `HelloController.php` and add the following method:

     ```
     public function greet() {
         return view('hello', ['name' => 'Laravel Enthusiast']);
     }
     ```

3. **Update the Route**:
   - Modify the route in `web.php` to use the controller:

     ```
     Route::get('/hello', [App\Http\Controllers\HelloController::class,
     'greet']);
     ```

4. **Test the Controller**:
   - Refresh the browser at http://127.0.0.1:8000/hello.
   - The page should now display "Hello, Laravel Enthusiast!"

Tasks for Students

1. **Middleware for Authentication**
   - Add middleware to restrict access to routes for authenticated users only.

    - **Deliverable:** Show how access is restricted when not logged in.

2. **Pagination for Task List**
    - Use Laravel’s pagination feature to display tasks in chunks.
    - **Deliverable:** Show the paginated task list.

Practical Session 8 – Implementing Testing in PHP

Practical Learning Outcomes

By the end of this session, students will be able to:

1. Set up a testing framework in PHP.
2. Write and run basic unit tests for PHP functions.
3. Perform basic integration testing for backend functionality.

Practical Activities

Activity 1: Setting up a Testing Framework

Steps:

1. **Install PHPUnit**:
    - PHPUnit is a popular testing framework for PHP.
    - Install PHPUnit via Composer by running this command in your project directory:

      ```
      composer require --dev phpunit/phpunit
      ```

    - Verify the installation by running:

      ```
      vendor/bin/phpunit --version
      ```

2. **Set Up the PHPUnit Configuration File**:
    - Create a `phpunit.xml` file in the root directory of your project:

      ```
      <?xml version="1.0" encoding="UTF-8"?>
      <phpunit bootstrap="vendor/autoload.php">
          <testsuites>
              <testsuite name="Application Test Suite">
                  <directory>tests</directory>
              </testsuite>
          </testsuites>
      </phpunit>
      ```

3. **Create the Test Directory**:
    - Create a `tests` directory in your project’s root folder:

```
mkdir tests
```

Activity 2: Writing Unit Tests

Steps:

1. **Create a PHP File to Test**:
   - Create a file named `Math.php` in the `src` directory:

   ```
   <?php

   namespace App;

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

       public function multiply($a, $b) {
           return $a * $b;
       }
   }
   ```

2. **Create a Test File**:
   - Create a file named `MathTest.php` in the `tests` directory:

   ```
   <?php

   use PHPUnit\Framework\TestCase;
   use App\Math;

   class MathTest extends TestCase {
       public function testAdd() {
           $math = new Math();
           $result = $math->add(2, 3);
           $this->assertEquals(5, $result);
       }

       public function testMultiply() {
           $math = new Math();
           $result = $math->multiply(2, 3);
           $this->assertEquals(6, $result);
       }
   }
   ```

3. **Run the Test**:
   - Run PHPUnit in the terminal:

   ```
   vendor/bin/phpunit
   ```

   - Observe the output confirming that both tests passed.

Activity 3: Writing an Integration Test

Steps:

1. **Create a Simple Backend API Endpoint**:
   - Add the following route to your `web.php` file:

   ```
   Route::get('/api/sum', function (Request $request) {
       $a = $request->query('a', 0);
       $b = $request->query('b', 0);
       return response()->json(['sum' => $a + $b]);
   });
   ```

2. **Write the Integration Test**:
   - Create a file named `ApiTest.php` in the `tests` directory:

   ```
   <?php

   use PHPUnit\Framework\TestCase;
   use Illuminate\Foundation\Testing\RefreshDatabase;
   use Illuminate\Support\Facades\Http;

   class ApiTest extends TestCase {
       public function testSumEndpoint() {
           $response = Http::get('http://127.0.0.1:8000/api/sum?a=3&b=7');
           $this->assertEquals(200, $response->status());
           $this->assertEquals(10, $response->json()['sum']);
       }
   }
   ```

3. **Run the Integration Test**:
   - Start your Laravel server:

   ```
   php artisan serve
   ```

   - Run PHPUnit again:

   ```
   vendor/bin/phpunit
   ```

   - Confirm that the integration test passed successfully.

Activity 4: Debugging Failing Tests

Steps:

1. **Introduce a Bug**:
   - Modify the `add` method in `Math.php` to deliberately return incorrect results:

   ```
   public function add($a, $b) {
       return $a - $b; // Incorrect logic
   }
   ```

2. **Run the Tests Again**:
   - Execute PHPUnit:

   ```
   vendor/bin/phpunit
   ```

   - Observe the failed test and the error message.
3. **Fix the Bug**:

- Correct the `add` method:

  ```
  public function add($a, $b) {
      return $a + $b;
  }
  ```

4. **Re-run the Tests**:
   - Verify that all tests pass after fixing the bug.

Tasks for Students

1. **Test Database Interactions with Mocking**
   - Mock database queries using a library like Mockery or PHPUnit's mocking tools.
   - **Deliverable:** Simulate database query results in a test.

2. **Pagination for Task List**
   - Integrate PHPUnit tests into a CI/CD pipeline using GitHub Actions or a similar tool.
   - **Deliverable:** Provide a GitHub Actions YAML file for running tests automatically.