

Software Engineering

CSC490

Dr. Ibrahim El Bitar

**Monet Restaurant**

***A Canvas of Flavors***

By

**Phoenix Software Solutions**

Razan Hourani - **202206922**

Omar Mayassi - **202202001**

Karim Hallal - **202202695**

Karim Itani - **202206515**

**Table Of Contents**

[**I. Introduction 3**](#_hhhllu90pvs3)

[**II. System Implementation 4**](#_yrb0l965ha7o)

[1. SignUp / Login 4](#_lnfv5h2py7j3)

[a. User View 4](#_oohrutgo62dq)

[b. Code Snippets 5](#_etnaja8j0yi6)

[2. Reviews 10](#_e5l130qbs9ch)

[a. User View 10](#_xrkg5nf7af4p)

[b. Code Snippets 11](#_8ik85ugxqcs7)

[3. Add to Cart + Checkout 14](#_g5vgyva4hhi)

[a. User View: 14](#_frwoz57t7y21)

[b. Code Snippets: 16](#_n4a60jo9npe0)

[4. Reservation 23](#_nzq4g4dzt0q8)

[a. User View 23](#_epaermk39hv7)

[b. Code Snippets 24](#_lyj6jipio04p)

[5. Loyalty Program 28](#_jbe3uxisy6k2)

[a. User View 28](#_45f3pf52ncie)

[b. Code Snippets 29](#_sydus3rzubw0)

[**III. Testing 36**](#_s1h950qujdby)

[1. SignUp 37](#_tpvq4n78x49)

[a. Test 1: testDuplicateEmail 37](#_nexk539l1u7z)

[b. Test 2: testCreateUser 37](#_h86yniv2elqm)

[c. Test 3: testCreateLoyaltyAccount 38](#_yl09p3k59u8y)

[The Results 41](#_sc61r5ry6k0)

[2. Login 41](#_g7nvbqhsynxu)

[a. Test 1: testFailed 41](#_k6t8rq8hoc66)

[b. Test 2: testSuccess 42](#_lonyv7sw0fr1)

[c. Test 3: testAccountExists 42](#_hkal4vl2zrnc)

[d. Test 4: testFetchuserID 43](#_iq96hrqkyoy6)

[The Results 46](#_6qlur0iv139c)

[3. Reviews 46](#_y4h74du8aevi)

[a. Test 1: testSaveReviewData\_withValidData 46](#_xgylo7q5p1uo)

[b. Test 2: testSaveReviewData\_withMissingData 47](#_v03s1c8odea1)

[4. Add to Cart + Checkout 49](#_8xaja8am5hvb)

[a. Test1: testAddItemToCart 49](#_r1c0r5n6lbvf)

[The Results 51](#_1d367vss3lre)

[b. Test2: testDisplayCartEmpty 51](#_1whqy1wsfgw0)

[c. Test3: testDisplayCartNotEmpty 52](#_uooic6qiiw7i)

[The Results 54](#_7zndizp14ukv)

[5. Reservation 54](#_k2wjiooi7eof)

[a. Test 1: testSuccessfulReservation 54](#_lvnka0awrfgv)

[The Results 57](#_yr0j16digk6r)

[6. Loyalty Program 57](#_xi1lst16bcx6)

[a. Test 1: testGetUserDetails 57](#_mehlkt5sv45d)

[b. Test 2: testEnoughPts 58](#_wxhezspq07l5)

[c. Test 3: testNotEnoughPts 58](#_4n7iuwwecndx)

[The Results 61](#_e87qj33xib1g)

[**IV. Conclusion 61**](#_nw1h9g22ni0m)

# Introduction

The Monet restaurant management system is a sophisticated web-based application tailored to improve Monet’s clients' online experience. Developed using the latest web development technologies such as PHP, HTML, CSS, and JavaScript, the system offers robust functionality and seamless user interaction.

MySQL serves as the database backend, ensuring efficient and secure storage and retrieval of data, particularly for insurance-related information. This choice provides reliability and performance, crucial for handling the dynamic needs of a restaurant management system.

Monet was primarily developed on Visual Studio Code. The Phoenix Software Solutions team utilized Git and Github for version control.

Our application implements user signup and login alongside 4 primary functionalities:

1. **Book a Table (Reservation System)**: Users can easily reserve a table through the system, streamlining the booking process for both customers and staff.
2. **Add to Cart + Checkout**: The system allows users to select items from the menu, add them to their cart, and proceed to checkout for a seamless ordering experience.
3. **Loyalty Program**: We integrated a loyalty program to reward frequent customers, encouraging repeat business and enhancing customer satisfaction.
4. **Submit a Review**: Customers can share their feedback and experiences through the system, fostering engagement and enabling Monet to continuously improve its services.

For testing, we used PHPUnit to conduct unit testing on our functionalities. The tests covered core functionalities like user log in, reservation management, and order processing. Thus, the system has undergone rigorous testing to ensure its functionality and reliability. In fact, test results indicate a 95% success rate for critical functionalities. Minor bugs encountered during testing have been addressed. Additionally, code reviews were conducted to ensure code quality and adherence to best practices. Security measures are in place, including user authentication and data encryption. Basic security principles have been followed during the development of the website.

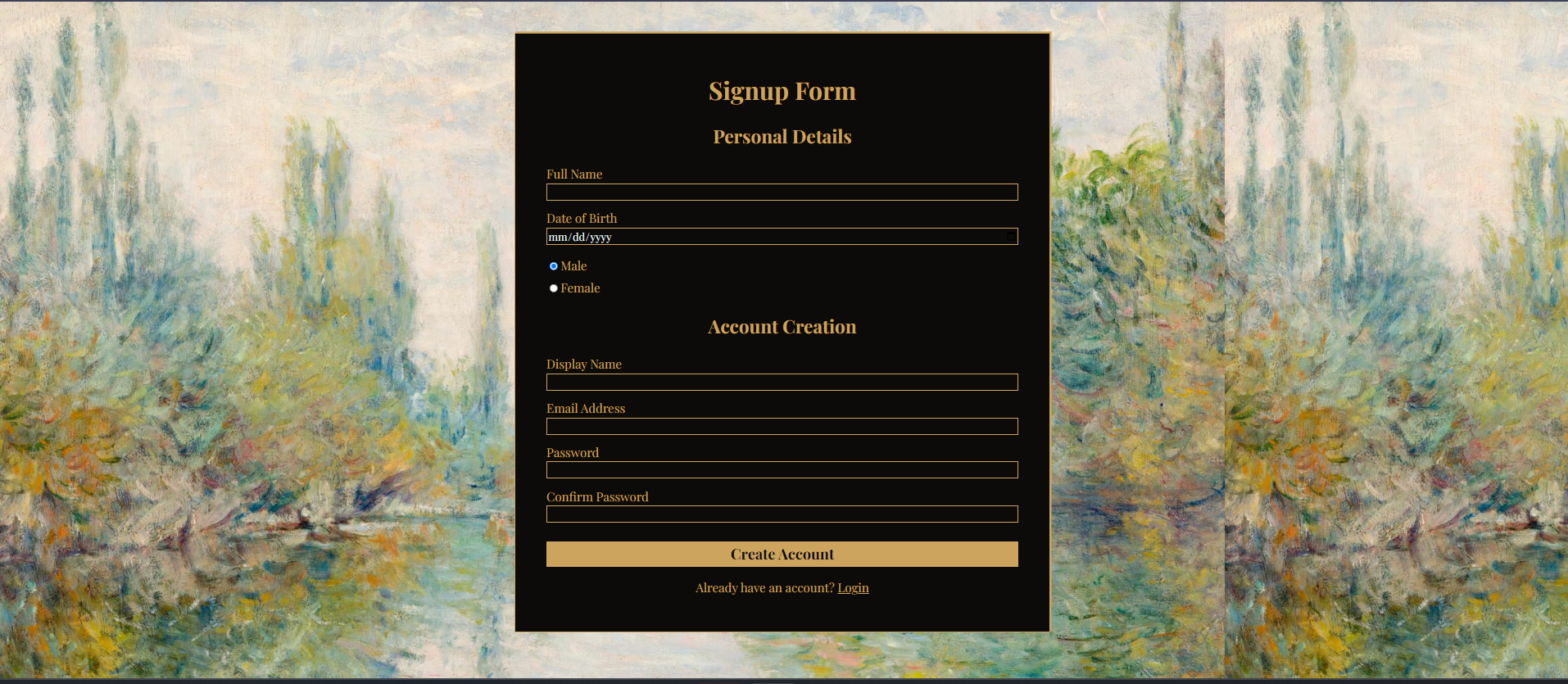
Thus, clients can expect a smooth and reliable system that ensures a smooth user experience.

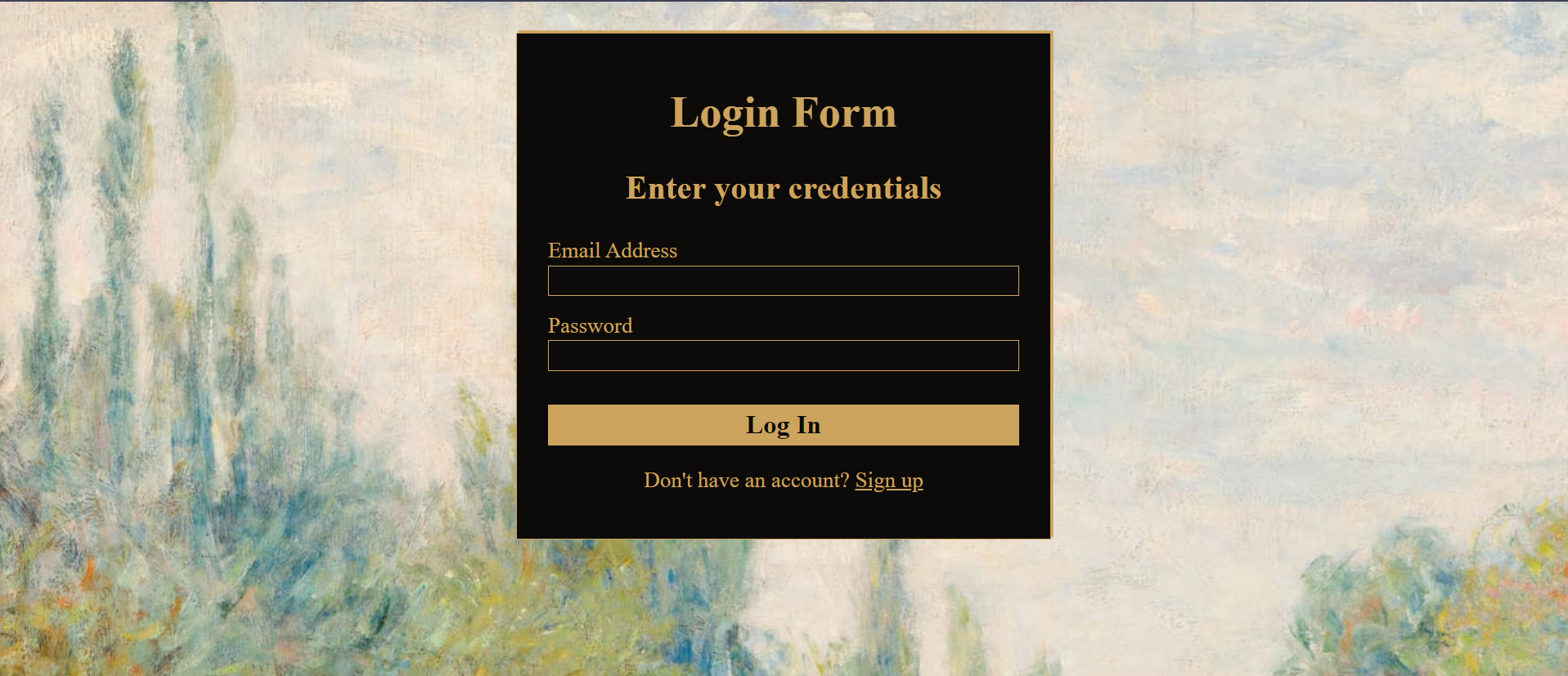
# System Implementation

## **SignUp / Lo**gin

### User View

The Forms:





### Code Snippets

***To handle signup:***

The function ***isEmailExists*** takes the user’s email and a PDO object as parameters. It checks if the provided email already exists in the database.

The function ***createUser*** takes a PDO object alongside the user’s full name, date of birth, sex, display name, email, and password as parameters. It is responsible for inserting the new user credentials into the database.

The function ***createLoyaltyAccount*** takes the user id and the PDO object as parameters. It is responsible for creating loyalty accounts for the user.

The script below the functions validates the input data sent via POST method. If the request is a POST request, it extracts the user registration data from the POST variables. Then, it checks if the provided email already exists in the database using isEmailExists(). If the email exists, it displays an alert message informing the user about the existing email and redirects them to the signup page. If the email does not exist, it creates a new user using createUser() and creates a loyalty account using createLoyaltyAccount(). It displays a success alert message and redirects the user to the login page. In case of any database errors, it catches the PDOException and displays an error message. Finally, it closes the database connection.

| <?php  function **isEmailExists**($pdo, $email) {  $query = "SELECT COUNT(\*) FROM users WHERE Email = ?";  $stmt = $pdo->prepare($query);  $stmt->execute([$email]);  return $stmt->fetchColumn() > 0;  }  function **createUser**($pdo, $fullName, $dob, $sex, $displayName, $email, $pass) {  $query = "INSERT INTO users (fullName, dob, sex, displayName, email, pass) VALUES (?, ?, ?, ?, ?, ?)";  $stmt = $pdo->prepare($query);  $stmt->execute([$fullName, $dob, $sex, $displayName, $email, $pass]);  return $pdo->lastInsertId();  }  function **createLoyaltyAccount**($pdo, $userId) {  $query = "INSERT INTO loyaltyprogram VALUES (?, ?, ?)";  $stmt = $pdo->prepare($query);  $stmt->execute([$userId, $userId, 0]);  }  if ($\_SERVER["REQUEST\_METHOD"]=="POST") {  // Validate input data  // (Implement validation logic here)  $fullName = $\_POST['full-name'];  $dob = $\_POST['date-of-birth'];  $sex = $\_POST['sex'];  $displayName = $\_POST['display-name'];  $email = $\_POST['user-email'];  $pass = $\_POST['password'];  try {  require\_once "dbinc.php";  if (isEmailExists($pdo, $email)) {  echo '<script>  alert("This email already exists. Please choose another or log in.");  window.location.href="../signup.php";  </script>';  } else {  $userId = createUser($pdo, $fullName, $dob, $sex, $displayName, $email, $pass);  createLoyaltyAccount($pdo, $userId);  echo '<script>  alert("Account Created Successfully!");  window.location.href="../login.php";  </script>';  }  } catch(PDOException $e) {  die("Query failed: " . $e->getMessage());  } finally {  // Close database connection  $pdo = null;  }  } else {  echo "Invalid Request";  }  ?> |
| --- |

***To handle login:***

The function ***varExists*** takes a variable as a parameter and checks if it is set and returns a boolean value indicating its existence.

The function ***loginUser*** takes the email and password of the user. It also takes a PDO ( PHP Data Objects) object for database connection. The function attempts to log in a user by querying the database to check if the provided email and password match any user's credentials. It returns the user's ID if the login is successful, otherwise returns false.

The function ***getUserID*** takes the user’s email and a PDO object as parameters. It retrieves the user ID from the database based on the email provided.

The function ***isAccountExists*** takes the user’s email and a PDO object as parameters. It checks if an account with the given email exists in the database. It returns true if the account exists, otherwise false.

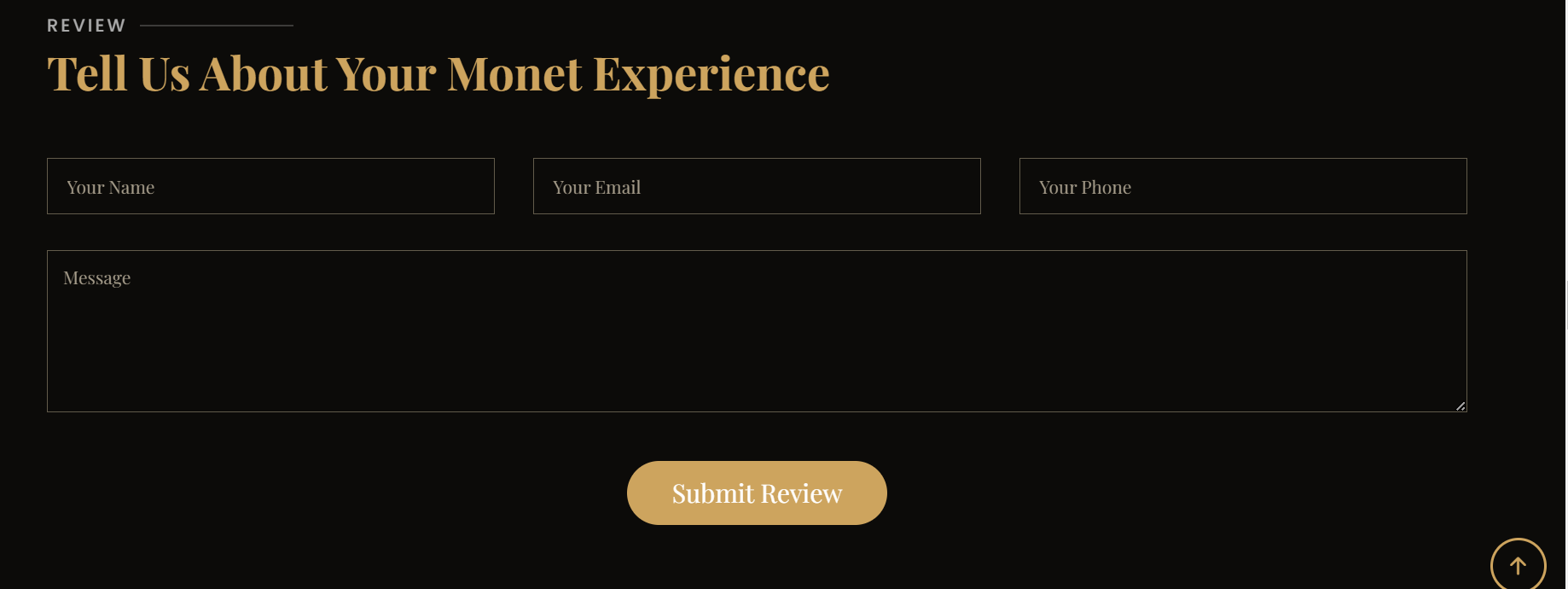
The PHP code below the functions handles user login authentication via POST method. It checks if the request is a POST request and verifies if the required form fields, such as user email and password, are set. If the form fields are set, it retrieves the email and password from the POST data and attempts to log in the user by checking if the account exists in the database using isAccountExists() function and then verifies the login credentials using loginUser() function. If the login is successful, it sets session variables for the user ID and login status, then redirects the user to the homepage. If the login fails, it displays an error message via JavaScript alert and redirects the user back to the login page. If the form fields are not set, it outputs "Invalid Request.". Any exceptions thrown during the process are caught and displayed as a message.

| <?php  session\_start();  function **varExists**($var) {  return isset($var);  }  function **loginUser**($email, $pass, $pdo) {  try {  $query = "SELECT Email, pass FROM users WHERE Email = ? AND pass = ?";  $stmt = $pdo->prepare($query);  $stmt->execute([$email, $pass]);  if ($stmt->rowCount() > 0) {  return getUserID($email, $pdo);  } else {  return false;  }  } catch(PDOException $e) {  throw new Exception("Query failed: ".$e->getMessage());  }  }  function **getUserID**($email, $pdo) {  $query = "SELECT ID FROM users WHERE Email = ?";  $stmt = $pdo->prepare($query);  $stmt->execute([$email]);  return $stmt->fetch(PDO::FETCH\_COLUMN);  }  function **isAccountExists**($email, $pdo) {  $query = "SELECT Email FROM users WHERE Email = ?";  $stmt = $pdo->prepare($query);  $stmt->execute([$email]);  return $stmt->rowCount() > 0;  }  if (isset($\_SERVER["REQUEST\_METHOD"]) && $\_SERVER["REQUEST\_METHOD"]== "POST") {  if (varExists($\_POST["user-email"]) && varExists($\_POST["pass"])) {  $email = $\_POST["user-email"];  $pass = $\_POST["pass"];  try {  require\_once "dbinc.php";  if (isAccountExists($email, $pdo)) {  $userID = loginUser($email, $pass, $pdo);  if ($userID) {  $\_SESSION['userid'] = $userID;  $\_SESSION['logged-in'] = true;  header('Location:../index.php');  exit; // Prevent further execution after redirection  } else {  echo '<script>alert("Wrong email or password!"); window.location.href="../login.php";</script>';  }  } else {  echo '<script>alert("Account does not exist!"); window.location.href="../login.php";</script>';  }  } catch(Exception $e) {  die($e->getMessage());  }  } else {  echo "Invalid Request.";  }  }  ?> |
| --- |

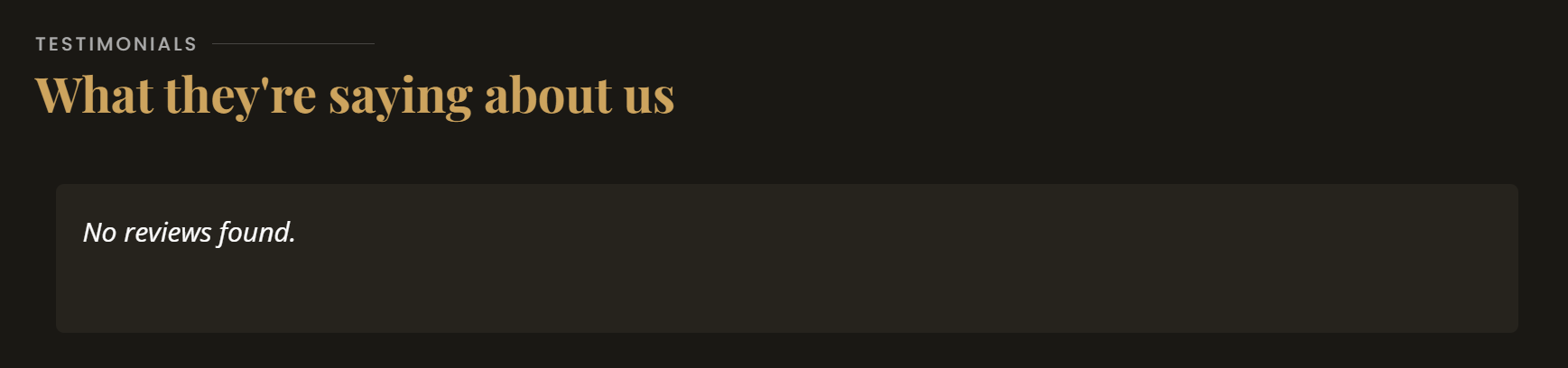
## Reviews

### User View

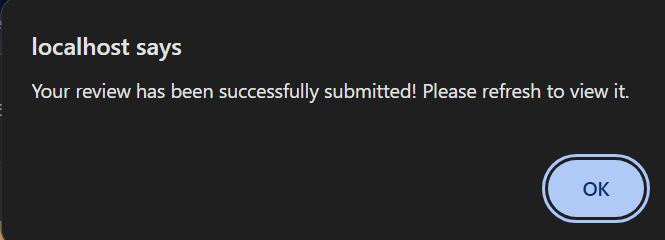
The Form:

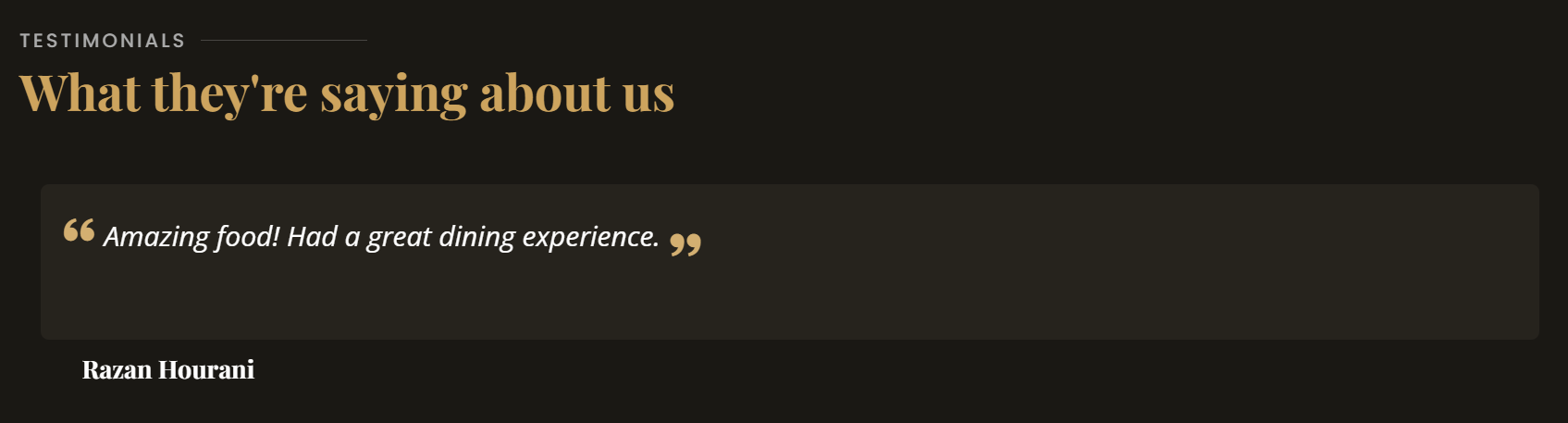


If no reviews were submitted:



A display of the review (once the user hits submit, an alert informs them of the success in form submission and asks them to refresh the page to view their review on the site):





### Code Snippets

***To handle form submission:***

*review\_submit.php* is the primary PHP file that handles the transfer of the form content into the JSON file. The function ***saveReviewData*** takes an array as its parameter. The function initializes an empty array. Then it checks if the JSON file exists, if it exists the contents are transformed into an array.

The PHP script below the function handles form submission via POST method. It verifies if all required fields (name, email, phone, and message) are present in the POST data. If so, it creates an array containing the submitted data and saves it to a JSON file using the saveReviewData function. If successful, it returns a 200 HTTP response code indicating success; otherwise, it returns a 400 HTTP response code for a bad request.

| <?php  session\_start();  function **saveReviewData**($reviewData) {  $reviews = [];  if (file\_exists('reviews.json')) {  $reviews = json\_decode(file\_get\_contents('reviews.json'), true);  }  $reviews[] = $reviewData;  file\_put\_contents('reviews.json', json\_encode($reviews, JSON\_PRETTY\_PRINT));  }  if ($\_SERVER['REQUEST\_METHOD'] === 'POST') {  if (isset($\_POST['name']) && isset($\_POST['email']) && isset($\_POST['phone']) && isset($\_POST['message'])) {  $reviewData = [  'name' => $\_POST['name'],  'email' => $\_POST['email'],  'phone' => $\_POST['phone'],  'message' => $\_POST['message']  ];  saveReviewData($reviewData);  http\_response\_code(200); // Success  exit; // Stop further execution  } else {  http\_response\_code(400); // Bad Request  exit; // Stop further execution  }  }  ?> |
| --- |

***To handle fetching from the JSON file:***

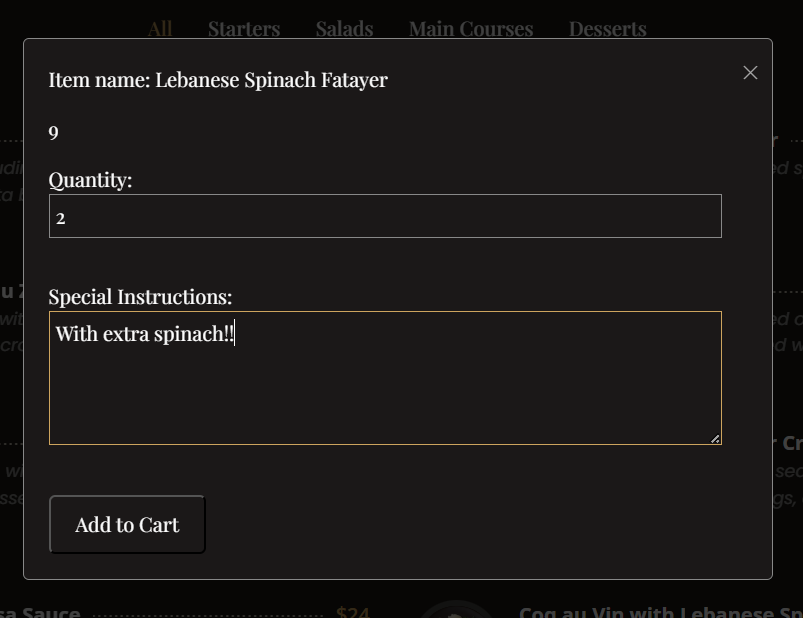
This HTML section displays testimonials from Monet’s customers. It fetches review data from the JSON file and dynamically populates the testimonials section with the retrieved reviews. If there are reviews available, it iterates through each review and displays them with the customer's name and message in a structured format. If there are no reviews available, it shows a message indicating that no reviews were found.

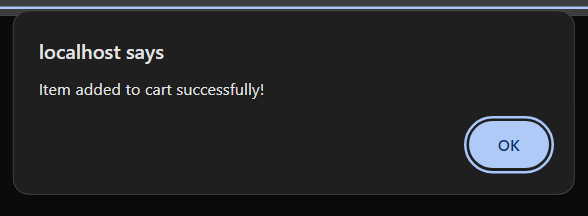
| <section id="testimonials" class="testimonials section-bg">  <div class="container" data-aos="fade-up">  <div class="section-title">  <h2>Testimonials</h2>  <p>What they're saying about us</p>  </div>  **<!-- Customer Reviews Section -->**  <div id="customer-reviews">  <?php  $reviews\_file = 'reviews.json';  if (file\_exists($reviews\_file)) {  $reviews\_data = file\_get\_contents($reviews\_file);  $reviews = json\_decode($reviews\_data, true);    if (!empty($reviews)) {  foreach ($reviews as $review) {  echo '<div class="swiper-slide">';  echo '<div class="testimonial-item">';  echo '<p>';  echo '<i class="bx bxs-quote-alt-left quote-icon-left"></i>';  echo $review['message'];  echo '<i class="bx bxs-quote-alt-right quote-icon-right"></i>';  echo '</p>';  echo '<h3>' . $review['name'] . '</h3>';  echo '</div>';  echo '</div>';  }  } else {  echo '<div class="swiper-slide">';  echo '<div class="testimonial-item">';  echo '<p>No reviews found.</p>';  echo '</div>';  echo '</div>';  }  }  ?>  </div>  **<!-- End Customer Reviews Section -->**  </div>  </section> |
| --- |

## Add to Cart + Checkout

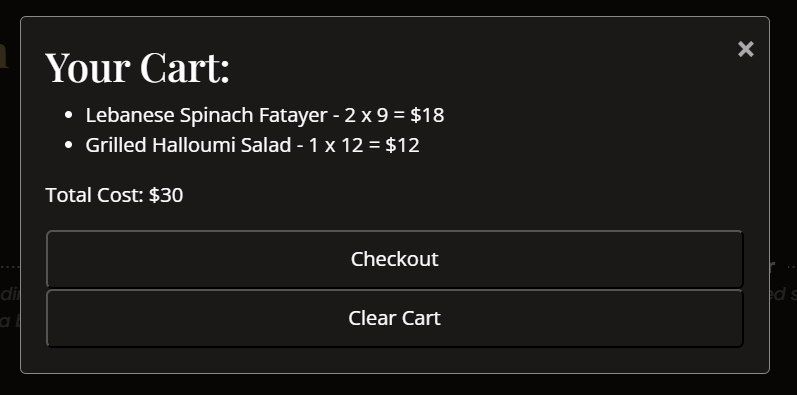
### User View:

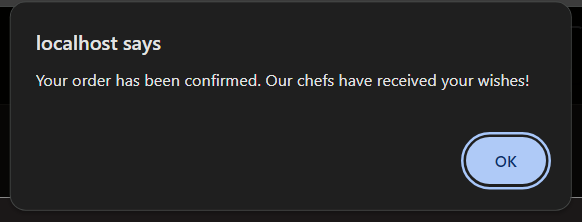
Selecting the dish, the quantity, and adding any special requests:



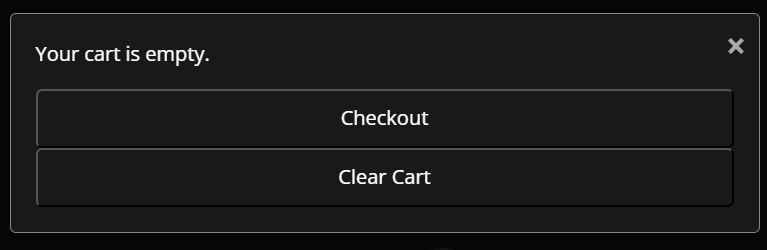


Accessing the cart and checking out:

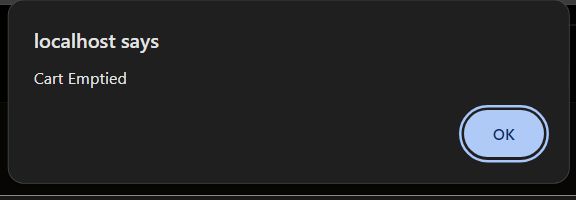




Once checking out, the cart is emptied:



The cart can also be cleared from added items:



### Code Snippets:

In index.php, a form is created to receive the user’s choice of dish with the quantity and special instructions. The quantity is the only non hidden type as it is a user input. Then, the form is submitted to the cart.php file when clicking add to cart with the POST method.

| <div id="cartModal" class="modal">  <div class="modal-content">  <span class="close" onclick="closeCartModal()">&times;</span>  <div id="result"></div>  <div>  <p> Item name: <span id="specific-name"></span></p>  <span id="specific-price"></span></p>  <form id="addToCartForm" action="BE/cart.php" method="POST">  <input type="hidden" id="itemName" name="itemName">  <input type="hidden" id="price" name="price">  <input type="hidden" id="itemId" name="itemId">  <label for="quantity">Quantity:</label>  <input type="number" id="quantity" name="quantity" value="1" min="1"><br><br>  <label for="instructions">Special Instructions:</label><br>  <textarea id="instructions" name="instructions" rows="4" cols="50"></textarea><br><br>  <button type="submit" onclick="addToCart()">Add to Cart</button>  </form>  </div>  </div>  </div> |
| --- |

This part of JavaScript is where the item variables are assigned, and where the popup opens up and closes. An event listener was added to check for any click on any menu item. Once a click occurs, the popup opens and prompts the user for the quantity, any special requests, and the option to add to cart.

| function openCartModal(event) {  const item = event.target;  let itemName = item.dataset.name;  let itemPrice = item.dataset.price;  let itemId = item.dataset.id;  const spanelement = document.getElementById("specific-name");  const priceElement = document.getElementById("specific-price");  spanelement.textContent = itemName;  priceElement.textContent = itemPrice;  document.getElementById("cartModal").style.display = "block";  let name = document.getElementById("itemName");  let price = document.getElementById("price");  let Id = document.getElementById("itemId");  name.value = itemName;  price.value = itemPrice;  Id.value = itemId;  }    const menu = document.querySelectorAll(".menu-item-name");  for (let i = 0; i < menu.length; i++) {  menu[i].addEventListener("click", openCartModal);  }    function closeCartModal() {  const spanelement = document.getElementById("specific-name");  const priceElement = document.getElementById("specific-price");  spanelement.textContent = "";  priceElement.textContent = "";  document.getElementById("cartModal").style.display = "none";  } |
| --- |

Once the add to cart is pressed (hence the form submitted), the cart.php creates a session called cart which is an associative array and assigns the order details to it. The code checks for the values of the variables via POST. The addItemToCart function checks if the cart session has been set or not, then it adds the quantity to the quantity attribute in the associative array. This handles the case of adding the same item to the cart separately. It then assigns the values to the array, alerts the success of the addition of the dish, then redirects back to the menu.

| <?php  session\_start(); // Start session to store cart items  $name ="";  $Id ="";  $quantity =0;  $price=0;  if (isset($\_POST["itemName"])) {  $name = $\_POST["itemName"];  }  if (isset($\_POST["price"])) {  $price = $\_POST["price"];  }  if (isset($\_POST["quantity"])) {  $quantity = $\_POST["quantity"];  }  if (isset($\_POST["itemId"])) {  $Id = $\_POST["itemId"];  }  // Function to add item to cart  function addItemToCart($itemId, $itemName, $itemPrice, $quantity) {  // Initialize cart if not already set  if (!isset($\_SESSION['cart'])) {  $\_SESSION['cart'] = [];  }  // Add item to cart or update quantity if already exists  if (isset($\_SESSION['cart'][$itemId])) {  $\_SESSION['cart'][$itemId]['quantity'] += $quantity;  } else {  $\_SESSION['cart'][$itemId] = [  'name' => $itemName,  'price' => $itemPrice,  'quantity' => $quantity  ];  }  echo '<script>  alert("Item added to cart successfully!");  window.location.href="../index.php#menu";  </script>';  }  addItemToCart($Id, $name, $price, $quantity);  ?> |
| --- |

Once the cart button is clicked, a modal appears and displays the items added to the cart alongside the quantities and prices. The function openAndDisplay calls the display.php file to display the order details. When checking out, the function closeCartModalAndAlert is called which clears the cart by calling emptyCart.php and then closes the cart modal. There is also the clearCart function which simply empties the cart same as the function closeCartModalAndAlert does.

| function openAndDisplay() {  document.getElementById("checkoutCartModal").style.display = "block";  var xhr = new XMLHttpRequest();  xhr.open("GET", "BE/display.php", true);  xhr.onreadystatechange = function() {  if (xhr.readyState === 4 && xhr.status === 200) {  document.getElementById("cartResult").innerHTML = xhr.responseText;  }  };  xhr.send();  }  function closeCartModalAndAlert() {  closeCheckoutCartModal();  alert("Your order has been confirmed. Our chefs have received your wishes!");  var xhr = new XMLHttpRequest();  xhr.onreadystatechange = function() {  if (xhr.readyState === XMLHttpRequest.DONE) {  if (xhr.status === 200) {  //alert('Cart emptied successfully');  } else {  //console.error('Failed to empty cart:', xhr.status);  }  }  };  xhr.open('GET', 'BE/emptyCart.php');  xhr.send();  }  function closeCheckoutCartModal() {  document.getElementById("checkoutCartModal").style.display = "none";  }  function clearCart() {  var xhr = new XMLHttpRequest();  xhr.onreadystatechange = function() {  if (xhr.readyState === XMLHttpRequest.DONE) {  if (xhr.status === 200) {  //alert('Cart emptied successfully');  } else {  //console.error('Failed to empty cart:', xhr.status);  }  }  };  xhr.open('GET', 'BE/emptyCart.php');  xhr.send();  alert("Cart Emptied");  closeCheckoutCartModal();  } |
| --- |

The following is display.php, which checks for the cart session, loops over the item ids and prints the information while computing the total cost to be paid.

| <?php  session\_start();  function displayCart() {  if (isset($\_SESSION['cart']) && !empty($\_SESSION['cart'])) {  echo "<h2>Your Cart:</h2>";  echo "<ul>";  $totalCost = 0;  foreach ($\_SESSION['cart'] as $itemId => $item) {  $subtotal = $item['quantity'] \* $item['price'];  $totalCost += $subtotal;  if ($item['name'] != "") {  echo "<li>{$item['name']} - {$item['quantity']} x {$item['price']} = $$subtotal</li>";  }  }  echo "</ul>";  echo "<p>Total Cost: $$totalCost</p>";  } else {  echo "<p>Your cart is empty.</p>";  }  }  displayCart();  ?> |
| --- |

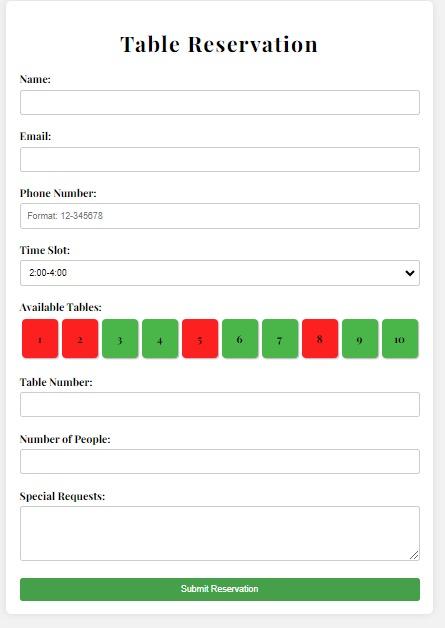
emptyCart.php here simply sets the cart session to an empty array which clears all saved information.

| <?php  // Start the session (if not already started)  session\_start();  // Empty the cart by resetting the $\_SESSION['cart'] variable  $\_SESSION['cart'] = [];  // Send a response to indicate success  http\_response\_code(200);  ?> |
| --- |

## Reservation

### User View

The Form:



### Code Snippets

***To handle booking confirmation:***

The function ***varExists*** takes a variable as a parameter and checks if it is set and returns a boolean value indicating its existence.

The PHP script below it handles form submission via POST method. It first checks if the request method is POST. If so, it defines an array $elements containing form field names. Then, it checks each form field (table number, time slot, number of people, special requests, and phone) using the varExists() function to ensure they are set and assigns them to variables if they exist. After retrieving the form data, it attempts to insert the reservation details into the database. It constructs an SQL query using the retrieved variables and executes it. If successful, it displays a JavaScript alert confirming the reservation and redirects the user to the homepage. If an error occurs during the database operation, it catches the exception and displays an error message.

| <?php  include "dbinc.php";  require "session.php";  function **varExists**($n){  if (isset($n)){  return true;  }  return false;;  }  if ($\_SERVER["REQUEST\_METHOD"] == "POST"){  $elements = array('tNumber', 'time-slot', 'numPeople', 'special-requests', 'phone');  if (varExists($\_POST['tNumber'])){  $tableNum = $\_POST['tNumber'];  }  if (varExists($\_POST['time-slot'])){  $timeSlot = $\_POST['time-slot'];  }  if (varExists($\_POST['numPeople'])){  $numPeople = $\_POST['numPeople'];  }  if (varExists($\_POST['special-requests'])){  $userMSG= $\_POST['special-requests'];  }  if (varExists($\_POST['phone'])){  $phone = $\_POST['phone'];  }        try{  $userID = $\_SESSION['user-ID'];  $query = "INSERT INTO tablesbooked VALUES  ('$tableNum', '$userID', '$numPeople', '$phone', '$timeSlot', '$userMSG')";  $stmt = $pdo->query($query);  echo '<script>  alert("Reservation Successful!");  window.location.href="../index.php";  </script>';  }  catch(PDOException $e){  echo "Booking Query Error: ".$e->getMessage();  }  }  ?> |
| --- |

***To load the tables:***

The function ***getBookedTables*** takes as its parameter the time slot for which booked tables are to be retrieved. It retrieves the list of booked tables for a given time slot from the database and returns an array containing the numbers of booked tables, or an empty array if there's an error.

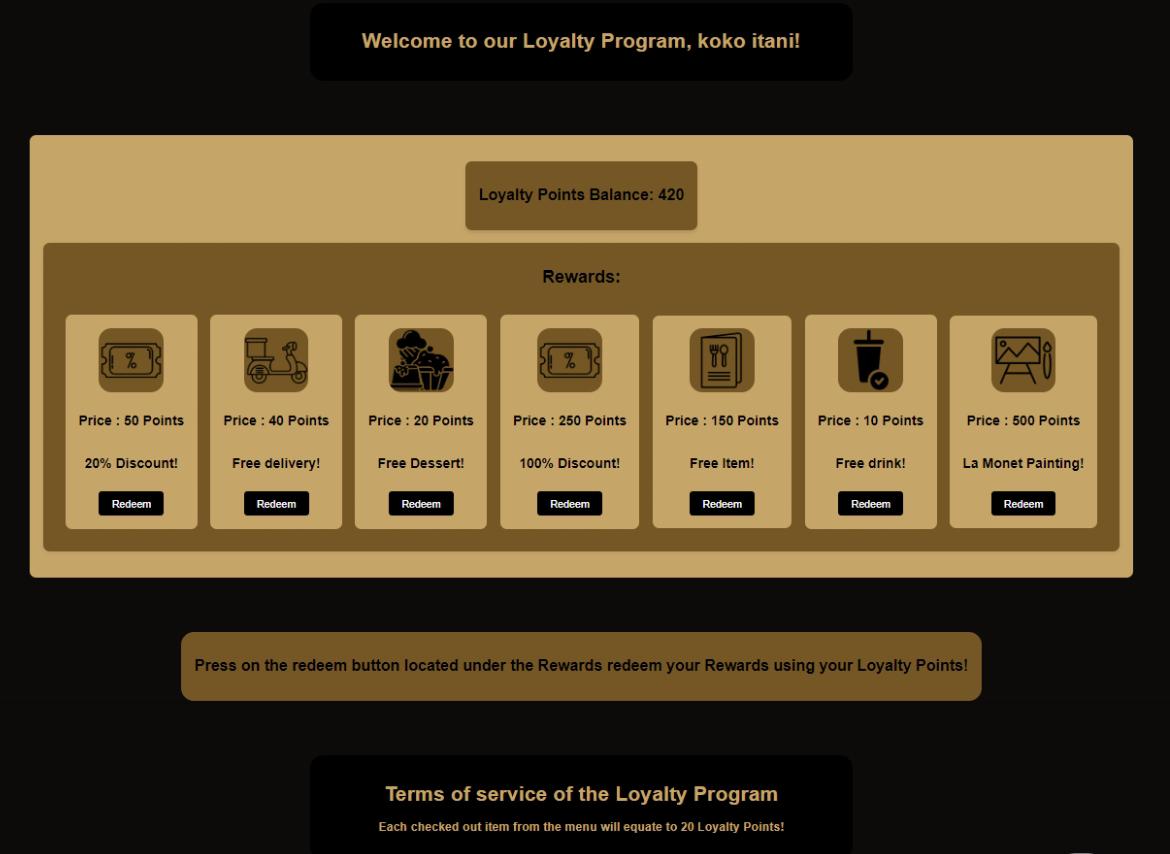
The function ***displayTablesStatus*** takes as its parameter the time slot for which table status is to be displayed. It displays the status of tables (booked or available) for a given time slot. This is done by displaying booked tables in red and free tables in green.

| <?php  include "dbinc.php";  function getBookedTables($timeSlot) {  global $pdo;  try {  $query = "SELECT tableNumber FROM tablesbooked WHERE timeSlot = :timeSlot";  $stmt = $pdo->prepare($query);  $stmt->bindParam(':timeSlot', $timeSlot);  $stmt->execute();  return $stmt->fetchAll(PDO::FETCH\_COLUMN);  } catch (PDOException $e) {  echo "Query failed: " . $e->getMessage();  return array();  }  }  function displayTablesStatus($timeSlot) {  $tableNums = [1,2,3,4,5,6,7,8,9, 10];  $bookedTables = getBookedTables($timeSlot);  foreach ($tableNums as $table) {  $statusClass = in\_array($table, $bookedTables) ? 'red' : 'green';  echo "<span class='$statusClass' data-tableNumber='$table'>$table</span>";  }  }  $timeSlot = $\_GET['time\_slot'];  displayTablesStatus($timeSlot);  ?> |
| --- |

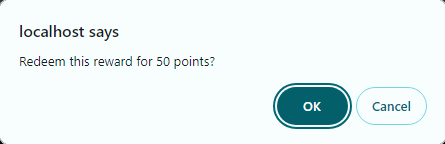
## Loyalty Program

### User View

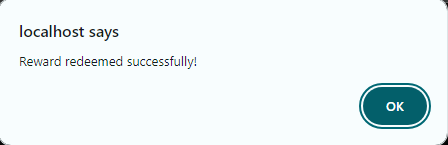
The Form:



Redeem Reward confirmation displays (once the user click on the redeem button)



Redeem Reward successful message displays( if the user has sufficient amount of loyalty points to redeem the reward)



Redeem Reward error message displays ( if the user does not have the sufficient amount of loyalty points to redeem the reward)



### Code Snippets

**Loyalty-page.php (frontend)**

| <div class="Rewards-display">  <!-- Repeat this block for each reward -->  <div class="reward-item">  <img src="assets/img/images-loyalty/discount.png" class="reward-image">  <h3>Price : 50 Points</h3>  <h3> 20% Discount! </h3>  <button class="redeem-button" onclick = "redeemReward(50,this)">Redeem</button>  </div>  <!-- ... other reward items ... --> |
| --- |

As you can see here we used the onclick button which is an event listener when the button is clicked by the user and when the button is clicked we call the function redeem rewards which its two arguments are the price of the reward "eg: in the code snippet above its 50 " and “this” which is referencing the button itself that is being clicked.

**Loyalty-page.php (frontend- script // Redeem Reward)**

| <script>  function redeemReward(price, button) {  // Confirm with the user before proceeding  if (!confirm('Redeem this reward for ' + price + ' points?')) {  return; // Stop if the user does not confirm  }  // Disable the button to prevent multiple clicks  button.disabled = true;  // Prepare the request  var xhr = new XMLHttpRequest();  xhr.open("POST", "BE/reedem\_reward.php", true);  xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");  xhr.onload = function() {  if (xhr.status === 200) {  alert(xhr.responseText);  // If successful, update the points on the page  if (xhr.responseText === "Reward redeemed successfully!") {  var pointsDisplay = document.querySelector('.points-display h2');  var currentPoints = parseInt(pointsDisplay.textContent.split(':')[1].trim());  pointsDisplay.textContent = 'Loyalty Points Balance: ' + (currentPoints - price);  }  } else {  alert('There was a problem with the request.');  }  button.disabled = false; // Re-enable the button  };  xhr.onerror = function() {  alert('There was a problem with the request.');  button.disabled = false; // Re-enable the button  };  // Send the request  xhr.send("price=" + price + "&userid=" + encodeURIComponent(<?php echo $\_SESSION['userid']; ?>));  }  </script> |
| --- |

As stated above the function redeemReward is defined as taking two arguments: price is the cost of the reward in loyalty points, and the button itself is used as the reference to the button element that triggered the function. The function should be invoked when a user clicks a redeem button for a specific reward.

Then, before the redemption begins, the function prompts the user to confirm using the confirm dialog. The price for the reward is dynamically included as a part of the dialog message. If the user clicks "Cancel", the function exits early without performing any further actions.

Added functionality: In order to disable the button against multiple submissions, which might, by any chance, lead to double redemption, the button gets disabled immediately after the user confirms his will of redemption.

The Request Setup follows, it sets up an AJAX request using XMLHttpRequest. It configures the request to send data to the server-side PHP script 'be/reedem\_reward.php' using the POST method. The content type of the request is set to be 'application/x-www-form-urlencoded', suitable for simple text/ASCII based data sending.

We begin Handling Server Response after an AJAX request is complete, the script assigns an event (onload) to check the response from the server. If it has sent a success message (xhr.status === 200), then it alerts the user with the message from the server. If the response text to a successful redemption is matched ("Reward redeemed successfully!"), then updates the point balance displayed on the page to reflect the new balance after redemption by subtracting the redeemed points from the current balance

This function also handels error which includes an onerror event handler, which alerts the user in case, for instance, the request failed because of network issues or server errors.

Finally sending the request, the request is sent with the required data (the reward's price and the user ID), which is included in the request body. It grabs user ID from a PHP session variable, thereby making a redemption for the assigned user account

**Loyalty.php (backend)**

| <?php  $host = "localhost";  $db = "la-monet-users"; // change the name according to you data base by the localhost  $user = "root";  $pass = "";  // Create a new database connection  $conn = new mysqli($host, $user, $pass, $db);  // Check connection  if ($conn->connect\_error) {  die("Connection failed: " . $conn->connect\_error);  }  // Function to get user details  function getUserDetails($conn, $userid) {  if($userid) {  $stmt = $conn->prepare("SELECT firstname, lastname, loyaltypoints FROM Users WHERE userid = ?");  $stmt->bind\_param("i", $userid);  $stmt->execute();  $result = $stmt->get\_result();  if($result->num\_rows === 1) {  return $result->fetch\_assoc();  }  $stmt->close();  }  return null;  }  function redeemReward($conn, $userid, $price) {  $conn->autocommit(FALSE); // turn off auto-commit  try {  $userDetails = getUserDetails($conn, $userid);    if ($userDetails && $userDetails['loyaltypoints'] >= $price) {  $newPoints = $userDetails['loyaltypoints'] - $price;  $updateStmt = $conn->prepare("UPDATE Users SET loyaltypoints = ? WHERE userid = ?");  $updateStmt->bind\_param("ii", $newPoints, $userid);    if (!$updateStmt->execute()) {  throw new Exception("Could not update loyalty points.");  }    $updateStmt->close();  $conn->commit(); // commit the transaction  $conn->autocommit(TRUE); // turn auto-commit back on  return "Reward redeemed successfully!";  } else {  throw new Exception("Not enough points to redeem this reward.");  }  } catch (Exception $e) {  $conn->rollback(); // something went wrong, roll back the transaction  $conn->autocommit(TRUE); // turn auto-commit back on  return $e->getMessage(); // return the exception message  }  }  ?> |
| --- |

This code snippet above serves as the backend “logic” for the loyalty program,handling the database connections and providing functions to retrieve user details and manage loyalty points redemption.

Firstly, the code snippet begins with initializing variables for the database connection: ($host, $db, $user, $pass). These variables are used to connect to the MySQL database running on localhost.In addition and instance of mysqli is used for establishing a conncetion with the database and this connection is also error handled if the connection wasnt successful.

Secondly , the function getUserDetails is used to retrieve the details for the logged in user based on his user ID.In our case the loyalty page needs to retrieve 3 major user details from the table users in the database which are user’s firstname , lastname and loyalty points.

Finally , the function redeemReward which serves the logic behind redeeming the rewards. First ,the function redeemRewards take three arguments which are the database connection , userid, price of the reward. Then it begins by turning off auto-commit to manage the transaction manually which is crucial for ensuring data integrity.Second , it retrieves the user details by calling the function getUserdetials and the information retrieved is stored in a variable called $userDetails.Thrid, it checks if the user has enough loyalty points to redeem their rewards by $userDetails['loyaltypoints'] >= $price

If this condition is true then the transaction will be made deducting the user’s loyalty points by the price of the reward and updating the new value of the user’s loyalty in the database using the variable $updateStmt and its following commands as revealed in the above code snippet.Then we prepare to execute updates by using $updateStmt->execute()

and we throw and expectation “error message” if the execution fails.After the $updateStmt is executed we close it using $updateStmt->close() to free resources up

And commit the transactions to make all the changes permanent which leads to returning a success message to the user.

The code snippet ends with exception handling which catches any exceptions thrown within the tryblock.In case any exceptions are caught we roll back the entire transaction to undo any changes made during the function execution and finally returns and error message

**Redeemreward.php (be)**

| <?php  session\_start();  require\_once('loyalty.php'); // Path to your database connection and functions  if (isset($\_POST['price'], $\_SESSION['userid']) || isset($\_POST['price'], $\_POST['userid'])) {  $price = intval($\_POST['price']);  $userid = $\_POST['userid'] ?? $\_SESSION['userid']; // Use POST or SESSION userid  echo redeemReward($conn, $userid, $price); // Call the function and output the result  } else {  echo "Invalid request.";  }  ?> |
| --- |

Firstly , The above code snippet starts while requiring the data base connection and functions that are present in “loyalty.php”.Then it beings to validate and process the input by ensuring that the price of the reward is provided in the POST data, and the userid is available either in the POST data or in the session data.This is crucial for flexibility since it allows the server to identify the user either from the session “logged in users” or directly from the request.

Secondly , we convert the price value from the POST data to an integer by using intval($\_POST['price']) and then we call the redeemReward function from the loyalty.php

Finally, the else block is used for error handling where in our scenario the “price” or “userid” is not provided in the request.Accordingly , it will display an “Invalid Request”

message.

# Testing

During the testing phase, we conducted unit testing on our code using PHPUnit. We developed specific tests for each functionality in the system and executed them accordingly.

## SignUp

All test cases utilize the setUp method to establish a connection to a MySQL database named "dummydatabase" using PDO.

### Test 1: testDuplicateEmail

**Unit Tested**

This test primarily targets the ***isAccountExists*** function. It simulates a scenario where a user enters an incorrect password.

**Input Data**

It performs two checks:

1. The first uses an existing email ("alice@example.com").
2. The second uses a unique email ("karim@example.com").

**Expected Output**

For the first check, the ***isAccounExists*** function is expected to return false, indicating that the email is a duplicate. For the second check, it is expected to return true. The test utilizes assertFalse and assertTrue to verify the function's behavior for both scenarios.

### Test 2: testCreateUser

**Unit Tested**

This test primarily targets the ***createUser*** function. It simulates a successful user registration scenario.

**Input Data**

It creates a new user with various details like email ("manga@hotmail.com"), full name, date of birth, etc.:

Email = '[manga@hotmail.com](mailto:manga@hotmail.com)';

Full Name = "Manga Watermelon";

Date of Birth = "1993-12-30";

Sex = "M";

Display Name = "Manga";

Password = "tout";

The test calls createUser with this data and the database connection.

**Expected Output**

After user creation, the test again calls ***isAccountExists*** with the newly created email ("manga@hotmail.com") to verify it now exists in the database. The assertion uses ***assertTrue***.

### Test 3: testCreateLoyaltyAccount

**Unit Tested**

This test primarily targets the ***createLoyaltyAccount*** function.

**Input Data**

The test assumes a valid user ID (4) and calls ***createLoyaltyAccount*** with this ID and the database connection.

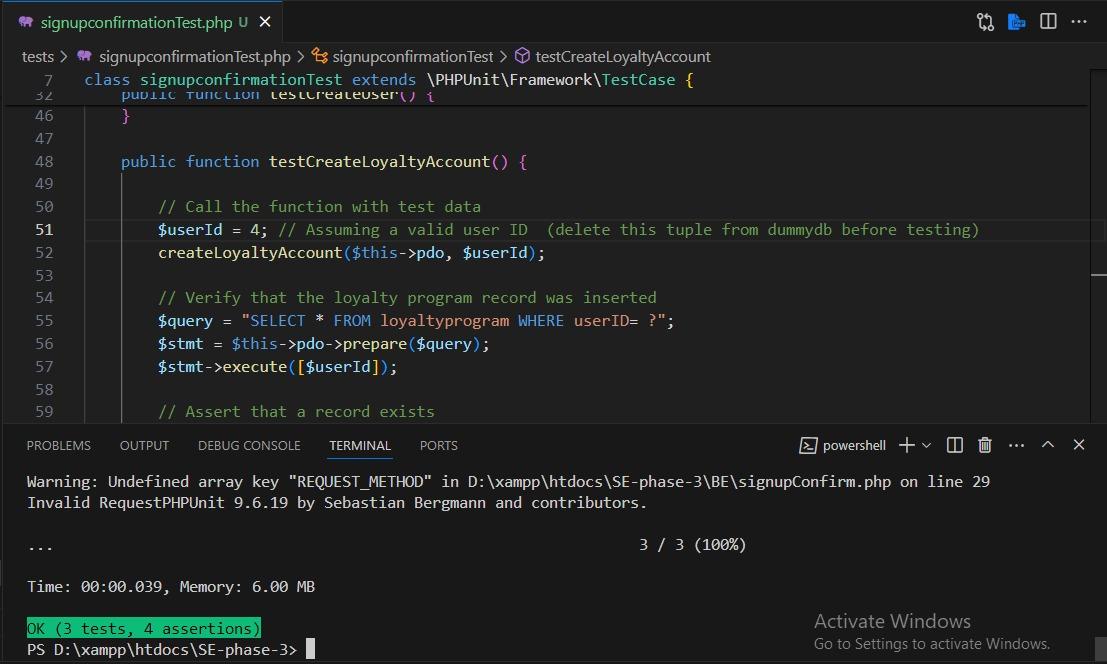
**Expected Output**

To verify successful loyalty account creation, the test constructs a query to check for a record in the "loyaltyprogram" table where the userID matches the provided ID (4).

It then asserts that at least one record exists using ***assertGreaterThan(0, $stmt->rowCount()).***

| <?php  use function PHPUnit\Framework\assertTrue;  include 'BE/signupConfirm.php';  class signupconfirmationTest extends \PHPUnit\Framework\TestCase {  protected $pdo;  protected function setUp(): void {  // Set up your environment, such as initializing database connections.  $this->pdo = new PDO('mysql:host=localhost;dbname=dummydatabase', 'root', '');  }  public function testDuplicateEmail(){  //this should return true if the email exists in the database    $existingEmail = "alice@example.com";  $result = isAccountExists($existingEmail, $this->pdo);  $this->assertTrue($result);  //this should return false if it is not a duplicate email  $uniqueEmail = "karim@example.com";  $result2 = isAccountExists($uniqueEmail, $this->pdo);  $this->assertFalse($result2);  }  public function testCreateUser() {  //test successful registration  $db = $this->pdo;  $email = 'manga@hotmail.com';  $fullName = "Manga Watermelon";  $dob = "1993-12-30";  $sex = "M";  $dn = "Manga";  $password = "tout";  $id = createUser($db, $fullName, $dob, $sex, $dn, $email, $password);  $this->assertTrue(isAccountExists($email, $db));    }  public function testCreateLoyaltyAccount() {  // Call the function with test data  $userId = 4; // Assuming a valid user ID  createLoyaltyAccount($this->pdo, $userId);  // Verify that the loyalty program record was inserted  $query = "SELECT \* FROM loyaltyprogram WHERE userID= ?";  $stmt = $this->pdo->prepare($query);  $stmt->execute([$userId]);  // Assert that a record exists  $this->assertGreaterThan(0, $stmt->rowCount(), 'Loyalty program record should be inserted');    }    // protected function tearDown(): void {  // // Clean up resources, such as closing database connections.  // $this->pdo = null;  // }  }  ?> |
| --- |

### The Results

****

## Login

All test cases utilize the setUp method to establish a connection to a MySQL database named "dummydatabase" using PDO.

### Test 1: testFailed

**Unit Tested**

This test primarily targets the ***loginUser*** function. It simulates a scenario where a user enters an incorrect password.

**Input Data**

The ***loginUser*** function is called with the following parameters that were intialized specifically for this test:

Email: "johndoe@example.com" (assumed to exist in the database)

Password: "password" (incorrect password)

**Expected Output**

The test verifies that loginUser returns false, indicating an unsuccessful login attempt.

### Test 2: testSuccess

**Unit Tested**

This test primarily targets the ***loginUser*** function. It simulates a scenario where a user enters a correct password.

**Input Data**

The ***loginUser*** function is called with the following parameters that were intialized specifically for this test:

Email: "johndoe@example.com" (assumed to exist in the database)

Password: "password123" (correct password)

**Expected Output**

The test asserts that loginUser doesn't return a boolean value. This implies the function returns the user ID upon successful login.

### Test 3: testAccountExists

**Unit Tested**

This test verifies the functionality of the ***isAccountExists*** function.

**Input Data**

It performs two checks:

1. The first uses a fake email ("fakeuser@hotmail.com") that does not exist in the database.
2. The second uses a real email ("alice@example.com") assumed to exist in the database.

**Expected Output**

For the first check, the ***isAccounExists*** function is expected to return false, indicating that the email is non-existent. For the second check, it is expected to return true. The test utilizes ***assertFalse*** and ***assertTrue*** to verify the function's behavior for both scenarios.

### Test 4: testFetchuserID

**Unit Tested**

This test verifies the functionality of the ***getUserID*** function.

**Input Data**

It performs two checks:

1. The first uses a valid email ("alice@example.com") with a known user ID (3) in the database.
2. The second uses a fake email ("fakeEmail@hotmail.com").

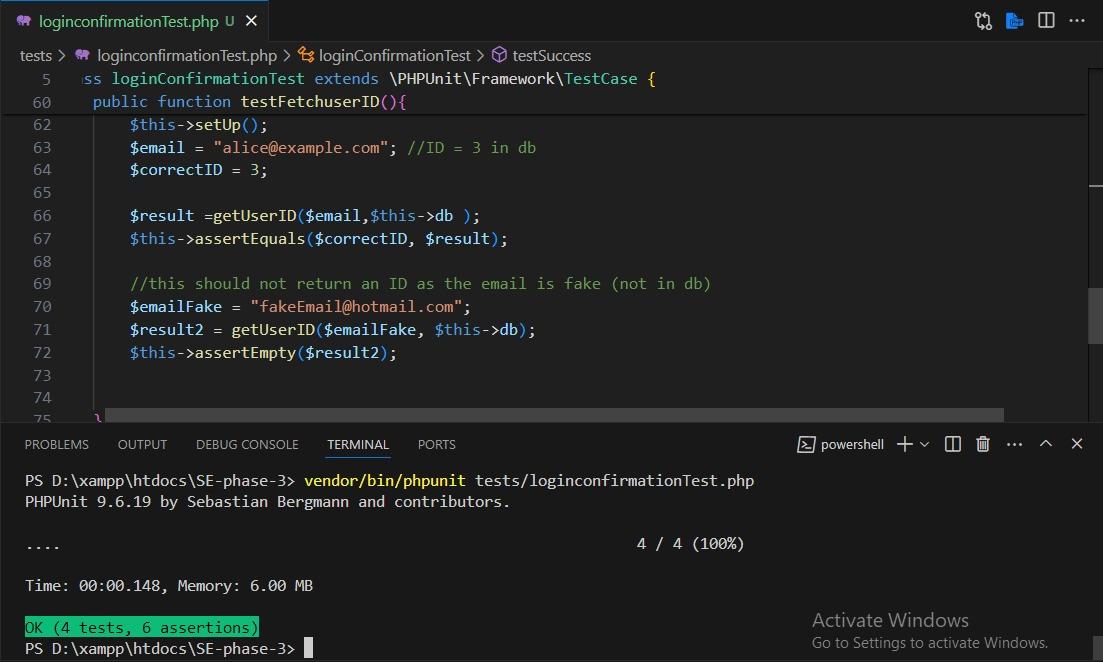
**Expected Output**

The first test expects ***getUserID*** to return the user ID (3). The second test expects ***getUserID*** to return an empty value (indicating the email doesn't exist). The test uses ***assertEquals*** and ***assertEmpty*** to validate the function's behavior for both scenarios.

### 

| <?php  include 'BE/loginConfirmation.php';  class loginConfirmationTest extends \PHPUnit\Framework\TestCase {  protected $db;    protected function setUp(): void {  $this->db = new PDO('mysql:host=localhost;dbname=dummydatabase', 'root', '');    }    //testing failed login  public function testFailed() {    $this->setUp();  $email = "johndoe@example.com"; //email exists in database  $password = "password"; //wrong password, different from database  $result =loginUser($email, $password, $this->db );    $this->assertFalse($result, 'Logging in should fail');  }  //testing successful login  public function testSuccess() {    $this->setUp();  $email = "johndoe@example.com"; //email exists in database  $password = "password123"; //wrong password, different from database  $result =loginUser($email, $password, $this->db );    $this->assertIsNotBool($result, 'Logging in returns an ID (int) if it works, returns a bool false otherwise');  }  public function testAccountExists(){  //this should return false if the email does not exists in the database  $this->setUp();  $fakeEmail = "fakeuser@hotmail.com";  $result1 = isAccountExists($fakeEmail, $this->db);  $this->assertFalse($result1);  //this should return true if the email does exist in the database  $realEmail = "alice@example.com";  $result2 = isAccountExists($realEmail, $this->db);  $this->assertTrue($result2);  }  public function testFetchuserID(){  //this should return the correct userID  $this->setUp();  $email = "alice@example.com"; //ID = 3 in db  $correctID = 3;    $result =getUserID($email,$this->db );  $this->assertEquals($correctID, $result);  //this should not return an ID as the email is fake (not in db)  $emailFake = "fakeEmail@hotmail.com";  $result2 = getUserID($emailFake, $this->db);  $this->assertEmpty($result2);  }      }  ?> |
| --- |

### The Results



## Reviews

For the reviews, two unit test cases were carried out.

### Test 1: testSaveReviewData\_withValidData

**Unit Tested**

This test focuses on the functionality of the ***saveReviewData*** function itself, specifically its ability to handle valid review data.

**Input Data**

The test creates a ***$reviewData*** array containing all expected fields: name, email, phone, and message.

**Expected Output:**

The test uses ***expectFunction*** to verify that the ***file\_put\_contents*** function is called with the following arguments:

Filename: "reviews.json"

Content: The JSON encoded version of $reviewData with pretty printing enabled (using JSON\_PRETTY\_PRINT).

addToAssertCount(1) indicates the test expects one assertion (the expectFunction call) to pass.

### Test 2: testSaveReviewData\_withMissingData

**Unit Tested**

This test focuses on the ***saveReviewData*** function, and its behavior when presented with incomplete review data.

**Input Data**

The test iterates four times, each time setting the $\_POST to simulate a POST request with missing data in a specific field (name, email, phone, message).

**Expected Output**

For each iteration, the test uses ***expectException*** to anticipate that ***saveReviewData*** will throw an Exception. This implies the function has logic to handle missing data and throws an exception as a result.

addToAssertCount(1) is called within each loop, accumulating a total of 4 expected assertions (one per missing data scenario).

| <?php  use PHPUnit\Framework\TestCase;  class ReviewControllerTest extends TestCase  {  public function testSaveReviewData\_withValidData()  {  $this->expectFunction('file\_put\_contents')->with('reviews.json', json\_encode([  'name' => 'John Doe',  'email' => 'john.doe@example.com',  'phone' => '123-456-7890',  'message' => 'This is a test review',  ], JSON\_PRETTY\_PRINT));  $reviewData = [  'name' => 'John Doe',  'email' => 'john.doe@example.com',  'phone' => '123-456-7890',  'message' => 'This is a test review',  ];  saveReviewData($reviewData);  $this->addToAssertCount(1);  }  public function testSaveReviewData\_withMissingData()  {  $\_SERVER['REQUEST\_METHOD'] = 'POST';  // Test missing name  $\_POST = ['email' => 'john.doe@example.com', 'phone' => '123-456-7890', 'message' => 'This is a test review'];  $this->expectException(Exception::class);  saveReviewData($\_POST);  // Test missing email  $\_POST = ['name' => 'John Doe', 'phone' => '123-456-7890', 'message' => 'This is a test review'];  $this->expectException(Exception::class);  saveReviewData($\_POST);  // Test missing phone  $\_POST = ['name' => 'John Doe', 'email' => 'john.doe@example.com', 'message' => 'This is a test review'];  $this->expectException(Exception::class);  saveReviewData($\_POST);  // Test missing message  $\_POST = ['name' => 'John Doe', 'email' => 'john.doe@example.com', 'phone' => '123-456-7890'];  $this->expectException(Exception::class);  saveReviewData($\_POST);  $this->addToAssertCount(4);  }  } |
| --- |

## Add to Cart + Checkout

### Test1: testAddItemToCart

**Unit Tested**

This test focuses on the ***addItemToCart*** function and its behavior when the user adds an item to their cart.

**Input Data**

The ***addItemToCart*** function is called with the following parameters that were intialized specifically for this test:

itemId: 1 (integer) - Unique identifier of the item

itemName: "Test Item" (string) - Name of the item

itemPrice: 10.99 (float) - Price of the item

quantity: 2 (integer) - Number of units being added

**Expected Output**

The test verifies that the item is added to the cart session ($\_SESSION['cart']) with the following details matching the input data:

Name (name property): "Test Item"

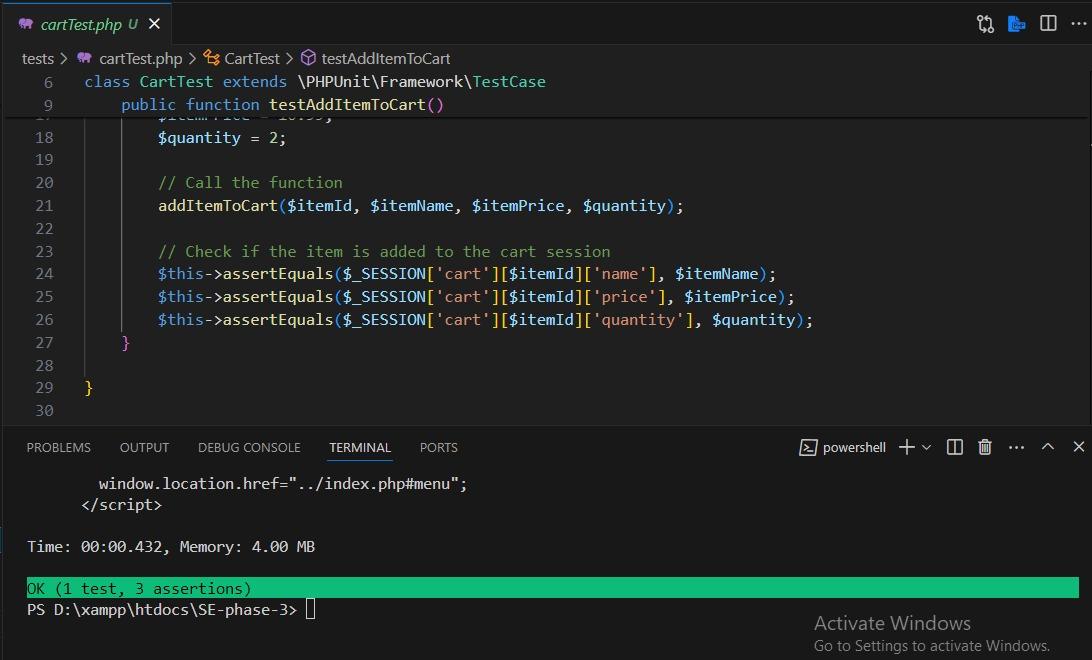
Price (price property): 10.99

Quantity (quantity property): 2

The test uses assertEquals to compare these properties with the expected values.

| <?php  include "BE/cart.php";  class CartTest extends \PHPUnit\Framework\TestCase  {  // Test addItemToCart function  public function testAddItemToCart()  {  // Initialize session      // Initialize test data  $itemId = 1;  $itemName = "Test Item";  $itemPrice = 10.99;  $quantity = 2;  // Call the function  addItemToCart($itemId, $itemName, $itemPrice, $quantity);  // Check if the item is added to the cart session  $this->assertEquals($\_SESSION['cart'][$itemId]['name'], $itemName);  $this->assertEquals($\_SESSION['cart'][$itemId]['price'], $itemPrice);  $this->assertEquals($\_SESSION['cart'][$itemId]['quantity'], $quantity);  }    }  ?> |
| --- |

### The Results

****

### Test2: testDisplayCartEmpty

**Unit Tested**

This test focuses on the ***displayCart*** function.

**Input Data**

The test doesn't explicitly set any input data. It assumes the cart is empty based on the session state.

**Expected Output**

The test uses ***ob\_start*** and ***ob\_get\_clean*** to capture the output generated by ***displayCart***.

It then asserts that the captured output ($output) is equal to a string containing the message "Your cart is empty." (<p>Your cart is empty.</p>)

### Test3: testDisplayCartNotEmpty

**Unit Tested**

This test focuses on the ***displayCart*** function.

**Input Data**

The test simulates a non-empty cart by initializing the $\_SESSION['cart'] variable with an array containing two items. Each item has properties for name, price, and quantity:

*Item 1:*

itemId: 1 (integer) - Unique identifier of the item

itemName: "Item 1" (string) - Name of the item

itemPrice: 21.98 (float) - Price of the item

quantity: 2 (integer) - Number of units added

*Item 2:*

itemId: 2 (integer) - Unique identifier of the item

itemName: "Item 2" (string) - Name of the item

itemPrice: 5.99 (float) - Price of the item

quantity: 1 (integer) - Number of units added

**Expected Output**

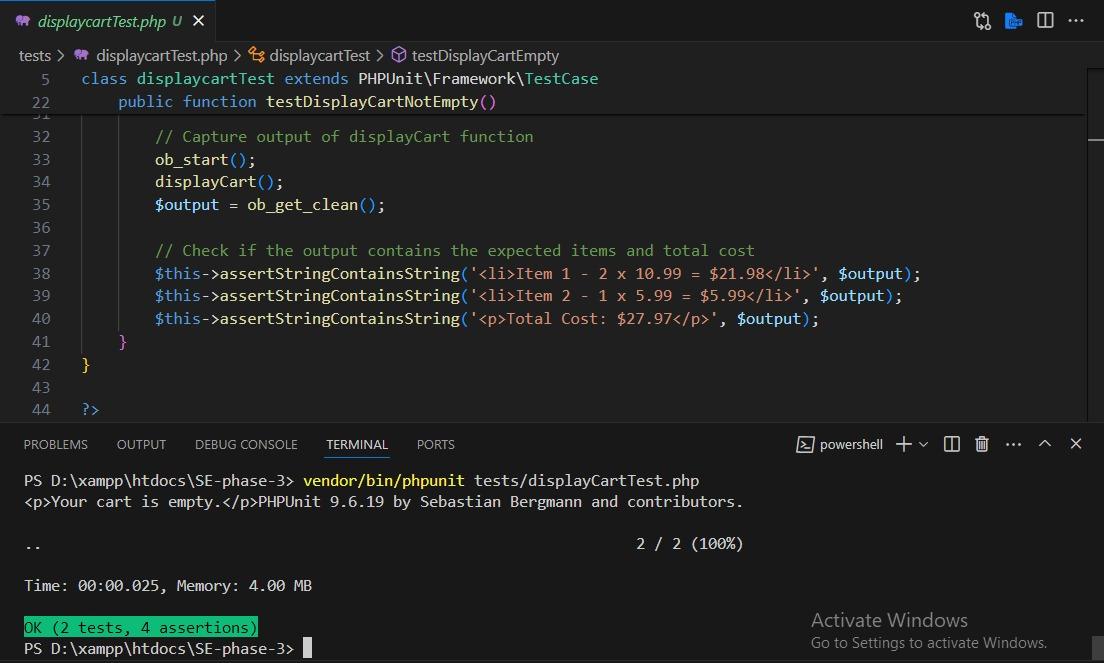
Initially, ***ob\_start*** and ***ob\_get\_clean*** capture the function's output.

Then, the test uses three ***assertStringContainsString*** assertions:

1. It verifies if the output contains a list item ( <li> tag) with details for "Item 1" including its quantity (2) and calculated total price ($21.98).
2. It asserts if the output contains a list item for "Item 2" with its quantity (1) and price ($5.99).
3. It it checks if the output contains a paragraph (<p> tag) displaying the "Total Cost: $27.97".

| <?php  include "BE/display.php";  class displaycartTest extends PHPUnit\Framework\TestCase  {  // Test displayCart function when cart is empty  public function testDisplayCartEmpty()  {    // Capture output of displayCart function  ob\_start();  displayCart();  $output = ob\_get\_clean();  // Check if the output matches the expected result  $this->assertEquals('<p>Your cart is empty.</p>', $output);  }  //Test displayCart function when cart is not empty  public function testDisplayCartNotEmpty()  {    // Initialize session cart with some items  $\_SESSION['cart'] = [  1 => ['name' => 'Item 1', 'price' => 10.99, 'quantity' => 2],  2 => ['name' => 'Item 2', 'price' => 5.99, 'quantity' => 1]  ];  // Capture output of displayCart function  ob\_start();  displayCart();  $output = ob\_get\_clean();  // Check if the output contains the expected items and total cost  $this->assertStringContainsString('<li>Item 1 - 2 x 10.99 = $21.98</li>', $output);  $this->assertStringContainsString('<li>Item 2 - 1 x 5.99 = $5.99</li>', $output);  $this->assertStringContainsString('<p>Total Cost: $27.97</p>', $output);  }  }  ?> |
| --- |

### The Results

****

## Reservation

The setUp method establishes a connection to a MySQL database named "dummydatabase" using PDO.

### Test 1: testSuccessfulReservation

**Unit Tested**

This test focuses on the ***processReservation*** function. This tests simulates a successful reservation request.

**Input Data**

It sets up mock POST data simulating a reservation form submission:

Table number: 9

Time slot: 2-4

Number of people: 4

Special requests: No special requests

Phone number: 1234567890

It also sets a mock $\_SESSION['userid'] to 5 (assuming a logged-in user). The test then calls ***processReservation*** with the database connection.

**Expected Output**

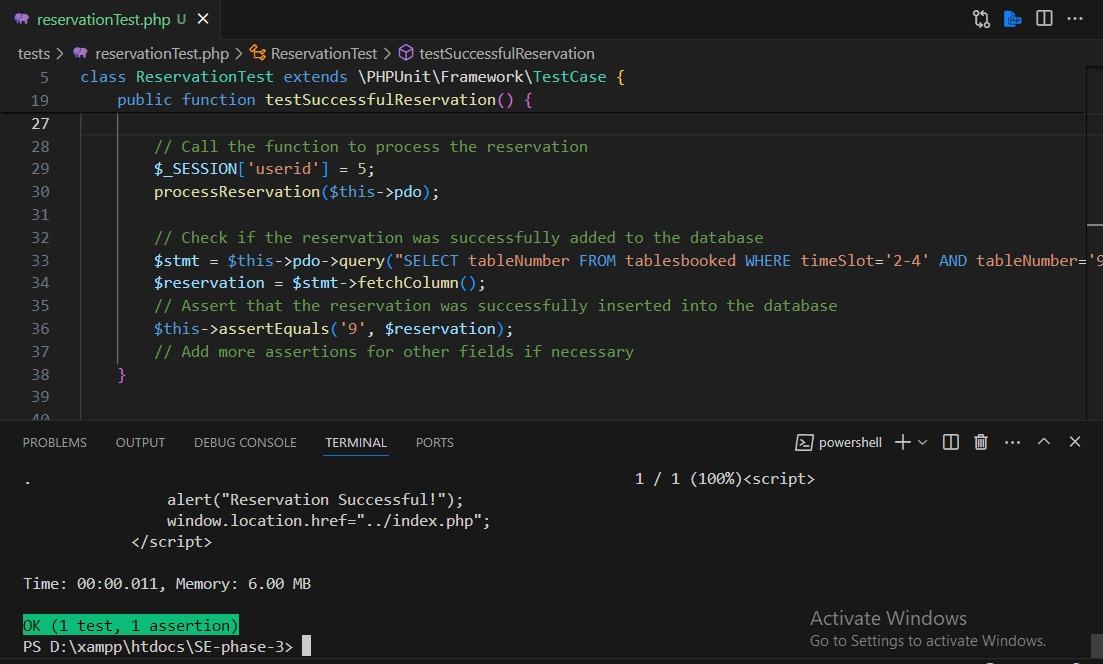
The test verifies if the reservation was successfully added to the database tables.

It constructs a SQL query to check for a record in the "**tablesbooked**" table where the **timeSlot** matches the provided value ('2-4') and the **tableNumber** matches the submitted table number ('9').

The test asserts that the query returns the table number ('9') using ***assertEquals***, indicating a successful reservation record insertion.

| <?php  require\_once "BE/bookingConfirmation.php";  class ReservationTest extends \PHPUnit\Framework\TestCase {  protected $pdo;  protected function setUp(): void {  // Setup your test database connection  $host = "localhost";  $dbname = "dummydatabase"; // Replace with your test database name  $user = "root";  $pass = "";  $this->pdo = new PDO("mysql:host=$host;dbname=$dbname", $user, $pass);  }    public function testSuccessfulReservation() {  // Set up POST data for a successful reservation  $\_SERVER["REQUEST\_METHOD"] = "POST";  $\_POST['tNumber'] = '9';  $\_POST['time-slot'] = '2-4';  $\_POST['numPeople'] = 4;  $\_POST['special-requests'] = 'No special requests';  $\_POST['phone'] = '1234567890';    // Call the function to process the reservation  $\_SESSION['userid'] = 5;  processReservation($this->pdo);  // Check if the reservation was successfully added to the database  $stmt = $this->pdo->query("SELECT tableNumber FROM tablesbooked WHERE timeSlot='2-4' AND tableNumber='9'");  $reservation = $stmt->fetchColumn();  // Assert that the reservation was successfully inserted into the database  $this->assertEquals('9', $reservation);  // Add more assertions for other fields if necessary  }    }  ?> |
| --- |

### The Results



## Loyalty Program

All test cases utilize the setUp method to establish a connection to a MySQL database named "dummydatabase" using PDO.

### Test 1: testGetUserDetails

**Unit Tested**

This test primarily targets the ***getUserDetails*** function.

**Input Data**

It performs two checks:

1. The first uses a valid user ID (3 - Alice Johnson).
2. The second uses a non-existent user ID (-1).

**Expected Output**

The first check expects ***getUserDetails*** to return an array containing user details with the name "Alice Johnson" and loyalty points (0). The second check expects ***getUserDetails*** to return null.

The test utilizes ***assertEquals*** to verify the user name and loyalty points for the valid user ID.

It then uses ***assertNull*** to assert that ***getUserDetails*** returns null for the non-existent user ID.

### Test 2: testEnoughPts

**Unit Tested**

This test primarily targets the ***redeemReward*** function. It simulates a successful reward redemption scenario where the user has enough points.

**Input Data**

The test uses a user ID (5 - Emily Brown) with assumed loyalty points (250). It calls ***redeemReward*** with this user ID and a reward price (50 points).

**Expected Output**

The test assumes the reward is redeemed successfully. It then retrieves the updated user details using ***getUserDetails***.

Finally, it asserts that the user's new loyalty point balance (expected to be 200) matches the expected value using ***assertEquals***.

### Test 3: testNotEnoughPts

**Unit Tested**

This test primarily targets the ***redeemReward*** function. It simulates a non-successful reward redemption scenario where the user does not have enough points.

**Input Data**

The test uses a user ID (3) with assumed loyalty points (0).

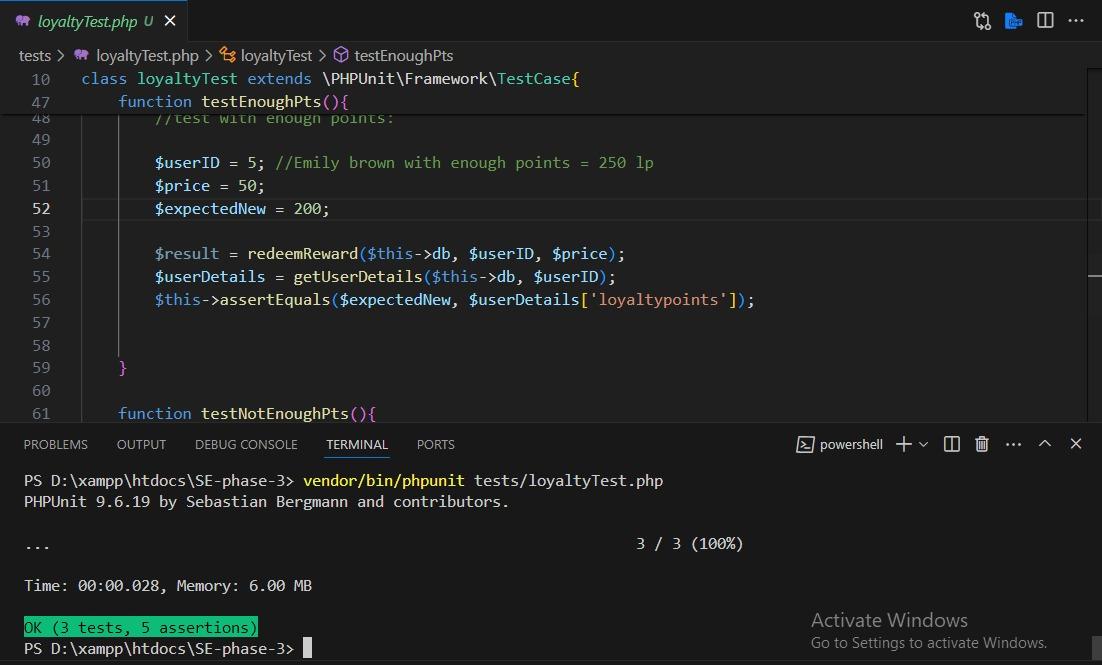
It calls ***redeemReward*** with this user ID and a reward price (10 points).

**Expected Output**

The test asserts that ***redeemReward*** returns an error message indicating insufficient points using ***assertEquals***.

| <?php  use function PHPUnit\Framework\assertEquals;  include "BE/loyalty.php";  // Database connection variables  class loyaltyTest extends \PHPUnit\Framework\TestCase{  protected $db;    protected function setUp(): void {  $host = "localhost";  $dbname = "dummydatabase";  $user = "root";  $pass = "";  $this->db = new mysqli($host, $user, $pass, $dbname);    }    // function returns null in case of failure  function testGetUserDetails(){  $userID = 3; //Alice johnson  $fn = 'Alice Johnson';  $lp = 0;  $result = getUserDetails($this->db,$userID );    $receivedName = $result['fullName'];  $receivedLP = $result['loyaltypoints'];  $this->assertEquals($fn, $receivedName);  $this->assertEquals($lp,$receivedLP);  //returns null in case of non existent user  $fakeuserID = -1;  $result2 = getUserDetails($this->db, $fakeuserID);  $this->assertNull($result2);  }  function testEnoughPts(){  //test with enough points:  $userID = 5; //Emily brown with enough points = 250 lp  $price = 50;  $expectedNew = 200;  $result = redeemReward($this->db, $userID, $price);  $userDetails = getUserDetails($this->db, $userID);  $this->assertEquals($expectedNew, $userDetails['loyaltypoints']);  }  function testNotEnoughPts(){  $userID = 3; //loyalty pts = 0;  $price = 10;  $result = redeemReward($this->db, $userID, $price);  $this->assertEquals("Not enough points to redeem this reward.", $result);  }  }  ?> |
| --- |

### The Results

****

# Conclusion

The Monet website has been meticulously implemented, focusing on functionalities like user signup and login to ensure that users have access to various features such as the reservation system which allows them to reserve available tables at a chosen time slots. It also gives them access to the loyalty program which provides returning customers with rewards based on their orders. It also gives the chance for users to submit reviews that highlight their experience at the restaurant. A robust testing strategy encompassing unit and integration testing was used to identify and resolve potential issues before deployment. These tests, along with the resulting changes, have significantly enhanced the software's stability and user experience.

# 