FEU INSTITUTE OF TECHNOLOGY

Brew Haven Coffee Shop:Complete E-Commerce Web Application

IT0049

(Web System and Technologies)

November 17, 2025

Submitted by:

Joshua Gabriel P. Ureta

BSITWMA

Submitted to:

Sir. Mar Eli Sagsagat

Brief Description of the Project:

This project involved the development of a transactional e-commerce website for "Brew Haven," a coffee shop, built using the CodeIgniter 4 PHP framework, which solves key problems such as user authentication, session management, real-time cart updates, and secure order processing with email confirmation. It streamlines the customer experience from registration to order fulfillment while providing administrators with robust tools for user and product management, including image handling, inventory tracking, and sales reporting. By implementing a scalable, secure, and user-friendly web system, the project aligns with **SDG 9: Industry, Innovation, and Infrastructure**, as it enhances digital infrastructure, supports small business innovation through modern web technologies, and promotes inclusive and sustainable industrialization by enabling efficient online service delivery.

Project's Output and Code Explanation

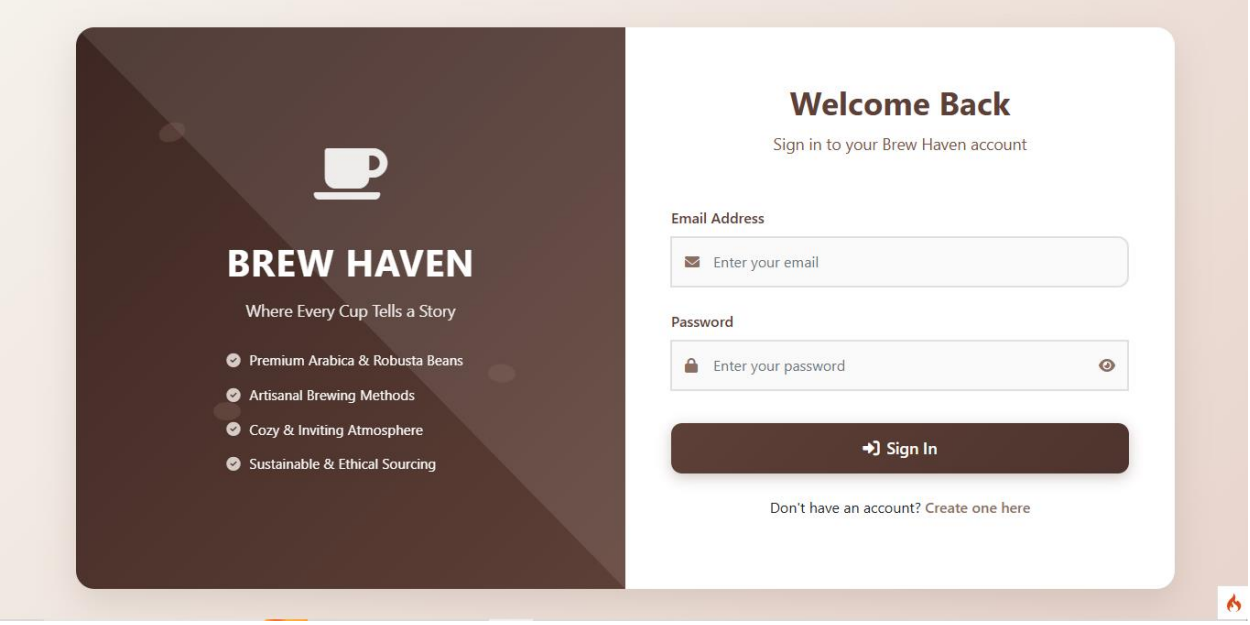

Figure 1.1 Login

Description:

The login form provides an essential functionality for user authentication, allowing individuals to access their Brew Haven accounts by entering their email address and password. The form shows a placeholder of text in input fields, a prominent sign-in button, and a registration option for new users to create an intuitive and accessible experience.

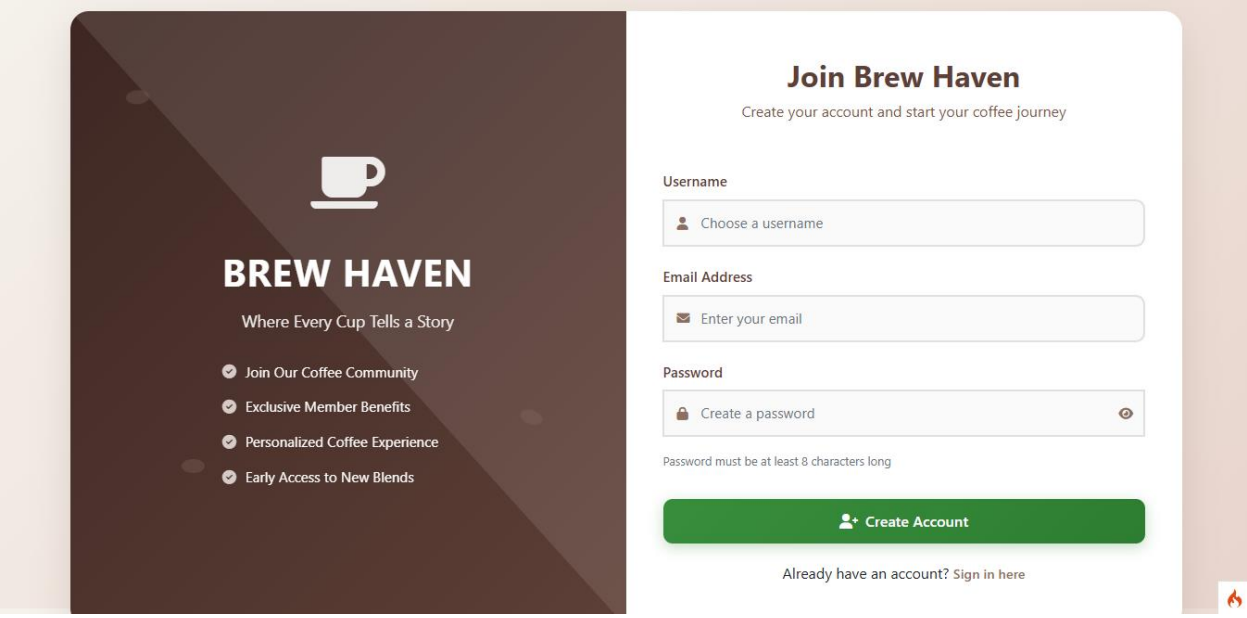

Figure 1.2 Register

Description:

The registration form enables new users to create an account for Brew Haven by submitting their email address and a chosen password. It efficiently provides an option for existing users to sign in instead, ensuring a straightforward and accessible account creation process.

Codes and Explanation:

Auth.php (Controller)

Form Validation Block:

```
$validation = \Config\Services::validation();
$validation->setRules([
    'email' => 'required|valid_email',
    'password' => 'required|min_length[1]'
]);
if (!$validation->withRequest($this->request)->run()) {
    $session->setFlashdata('errors', $validation->getErrors());
    return redirect()->to('/login');
}
```

What happens: CodeIgniter's validation service checks if email and password meet the rules. If validation fails, it stores errors in session flashdata and redirects back.

Session Creation Block:

```
$sessionData = [
    'id' => $user['id'],
    'username' => $user['username'],
    'isLoggedIn' => true
];
$session->set($sessionData);
```

What happens: After successful login, user data is stored in the session to maintain authentication state across pages.

Session Check Block:

```
if (session()->get('isLoggedIn')) {
    return redirect()->to('/dashboard');
}
```

What happens: Checks if user is already logged in by looking at session data, redirecting authenticated users away from login/register pages.

login.php (View)

```
<?php if (session()->getFlashdata('errors')): ?>
    <div class="alert alert-danger">
        <?php foreach (session()->getFlashdata('errors') as $error): ?>
            <li><?= $error ?></li>
        <?php endforeach; ?>
    </div>
<?php endif; ?>
```

What happens: Retrieves validation errors from session flashdata and displays them as list items in a red alert box.

Form Input Preservation:

```
<input value="<?= old('email', session()->getFlashdata('old')['email'] ?? '') ?>">
```

What happens: Pre-fills the email field with the user's previous input from session flashdata, so they don't have to retype everything.

register.php

Client-side Validation Block:

```
document.getElementById('registerForm').addEventListener('submit', function(e) {
    if (username.value.length < 3) {
        username.classList.add('is-invalid');
        isValid = false;
    }
    if (!isValid) e.preventDefault();
});
```

What happens: JavaScript provides immediate feedback by adding red borders to invalid fields before form submission, enhancing user experience.

Customer Side

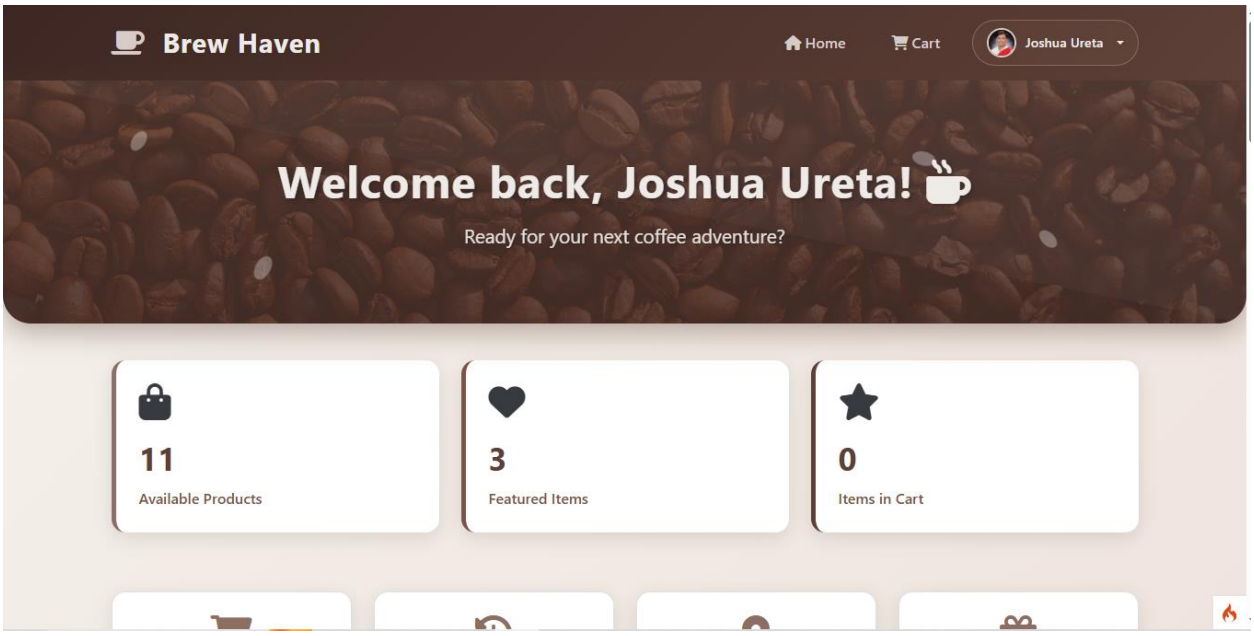

Figure 2.1 Landing page

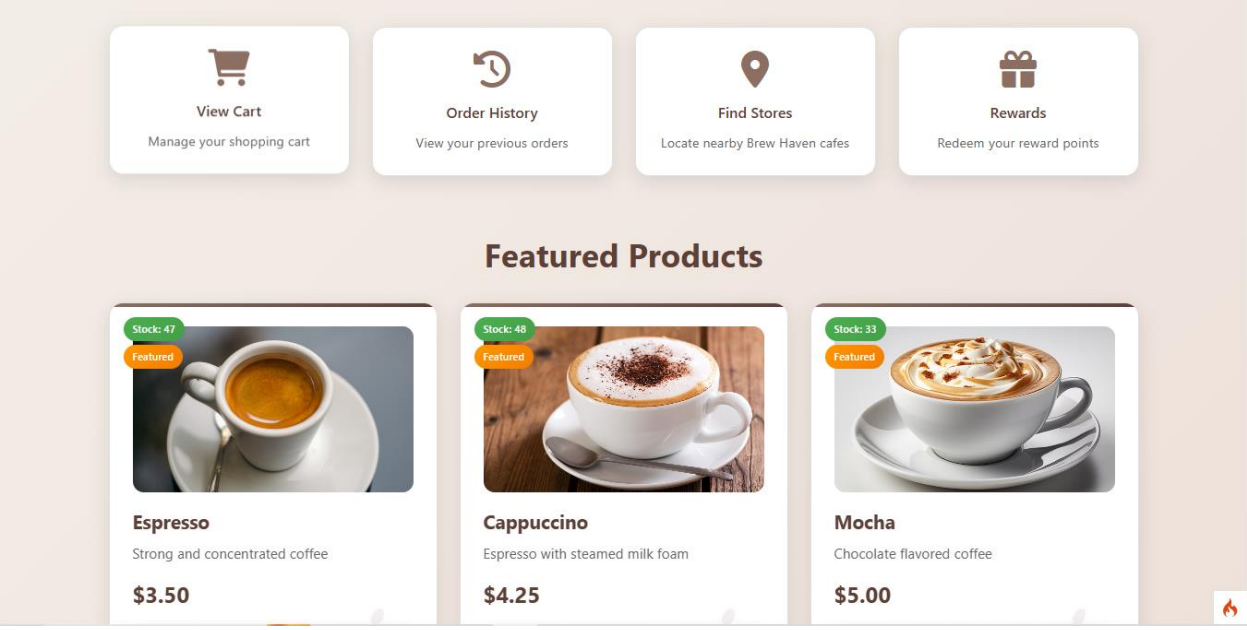

Figure 2.2 View Products

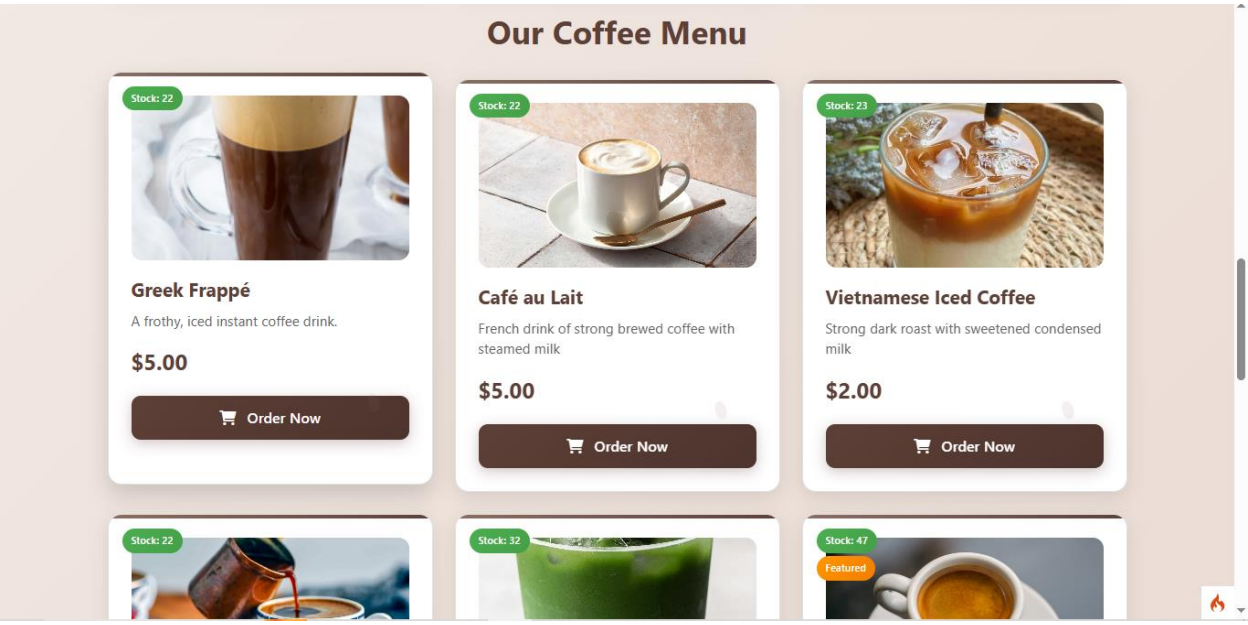

Figure 2.3 View Products

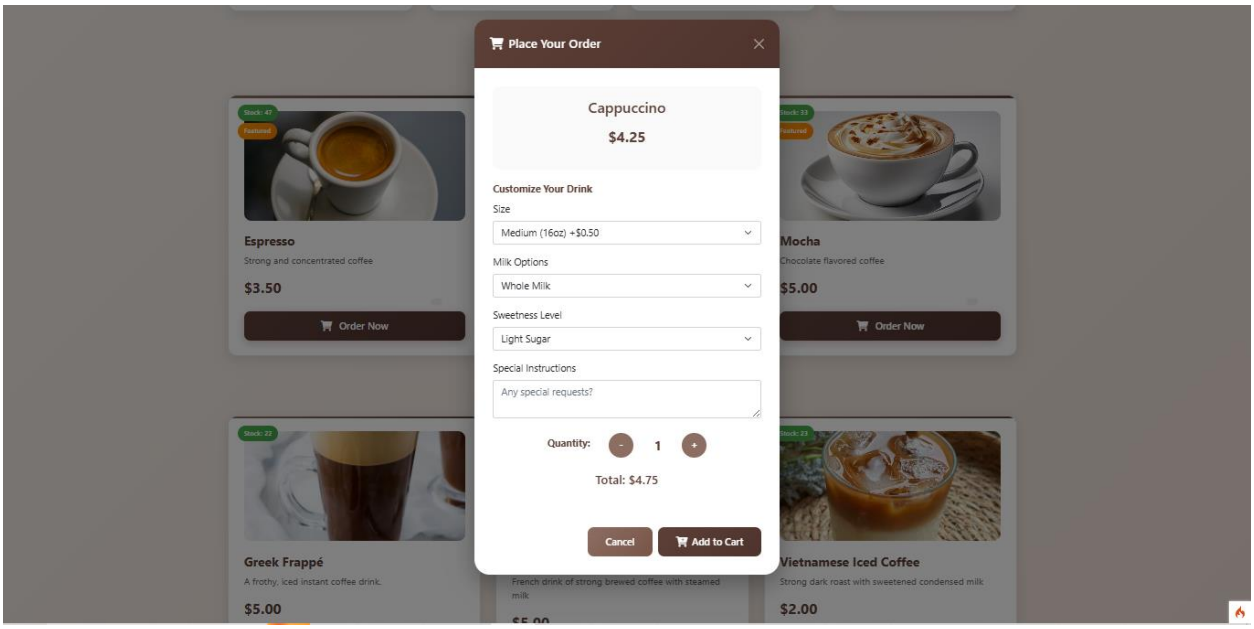

Figure 2.4 Pop- up cart

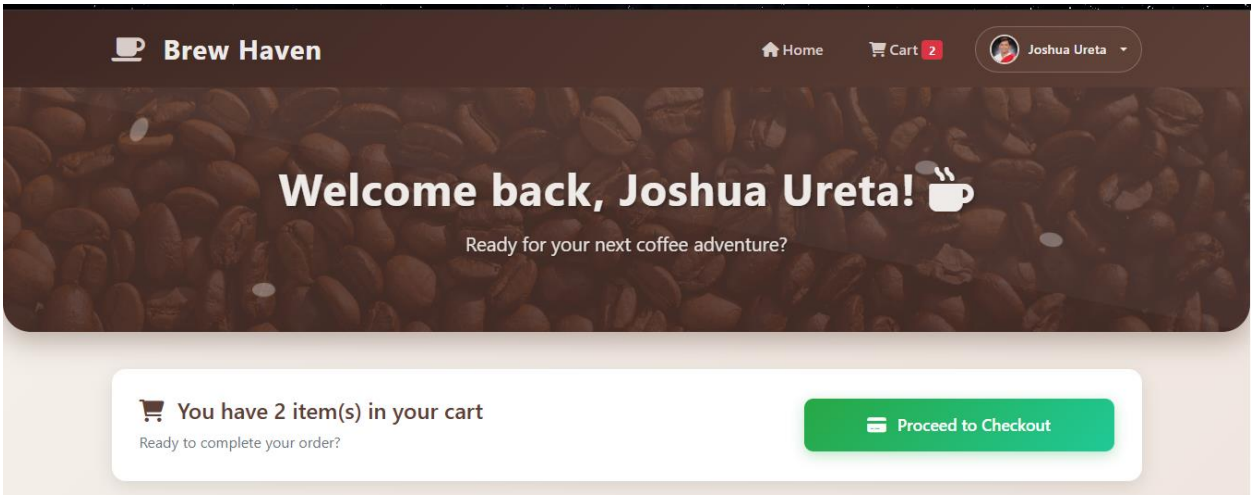

Figure2.5 Items Cart Indicator

Description:

This customer dashboard provides a central hub for users to view their order history, reward points, and explore a menu of available coffee drinks. The design presents a personalized greeting alongside clear navigation options and account statistics, for it to be an engaging and user-friendly experience. For ordering, the interface displays menu items with detailed descriptions and prices, culminating in a customization modal that allows users to tailor their drink before confirming their purchase.

Codes and Explanations:

UserModel

```
// In profile() method
$user = $this->userModel->find($userId);
```

What happens: Fetches user data for profile display and session updates

ProductModel

```
// In index() method - Main dashboard data
$data['products'] = $this->productModel->getProductsForDashboard();
$data['featured_products'] = $this->productModel->getFeaturedForDashboard();

// In cart() method
$data['products'] = $this->productModel->getAvailableProducts();

// In addToCart() method
$product = $this->productModel->find($productId);
```

What happens:

- Gets all products for main menu display
- Gets featured products for highlighted section
- Checks product existence and stock availability
- Provides product details for cart operations

CartModel

```
// Throughout the controller
$data['cart_count'] = $this->cartModel->getCartCount(session()->get('id'));
$data['cartItems'] = $this->cartModel->getCartItems($userId);
$data['cartTotal'] = $this->cartModel->getCartTotal($userId);

// In addToCart() method
$existingItem = $this->cartModel->getCartItem($userId, $productId, $size, $milk_type,
```

```
$sweetness);
$this->cartModel->insert($data);
$this->cartModel->update($existingItem['id'], ['quantity' => $newQuantity]);

// In removeFromCart() method
$this->cartModel->delete($itemId);
```

What happens:

- Manages shopping cart operations (add, update, remove)
- Calculates cart totals and item counts
- Handles cart item merging for duplicate products

OrderModel & OrderItemModel

```
// In orders() method
$orderModel = new OrderModel();
$data['orders'] = $orderModel->getUserOrders(session()->get('id'));

// In processCheckout() method
$db->table('orders')->insert($orderData);
$db->table('order_items')->insert($orderItemData);
```

What happens:

- Manages order creation and order items during checkout
- Retrieves user's order history for display

Summary: The dashboard.php view relies entirely on the Dashboard Controller, which orchestrates data from **multiple models** (UserModel, ProductModel, CartModel, OrderModel) to populate the dashboard with user-specific data, products, cart information, and order history.

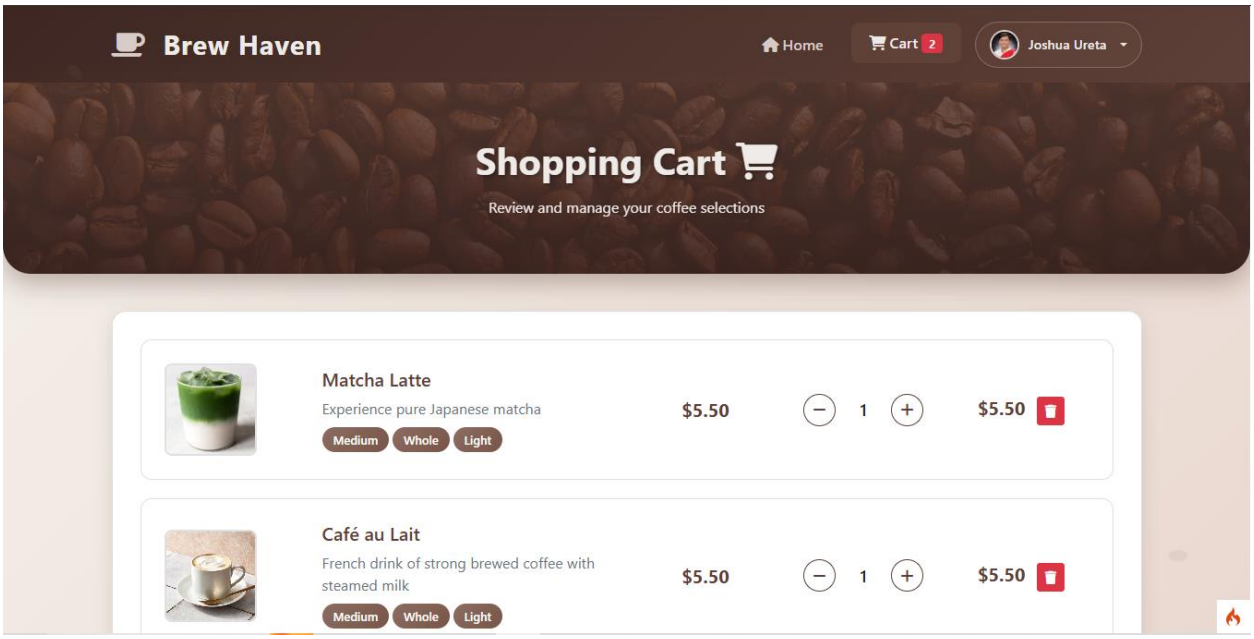

Figure 3.1 Cart page

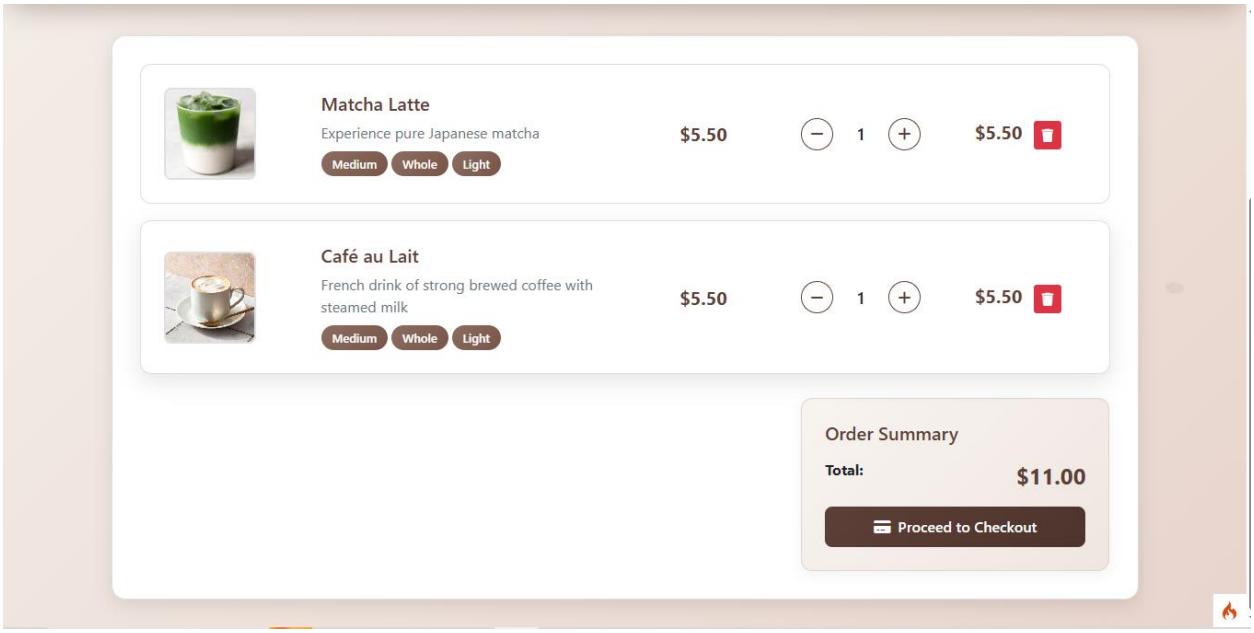

Figure 3.2 Cart page

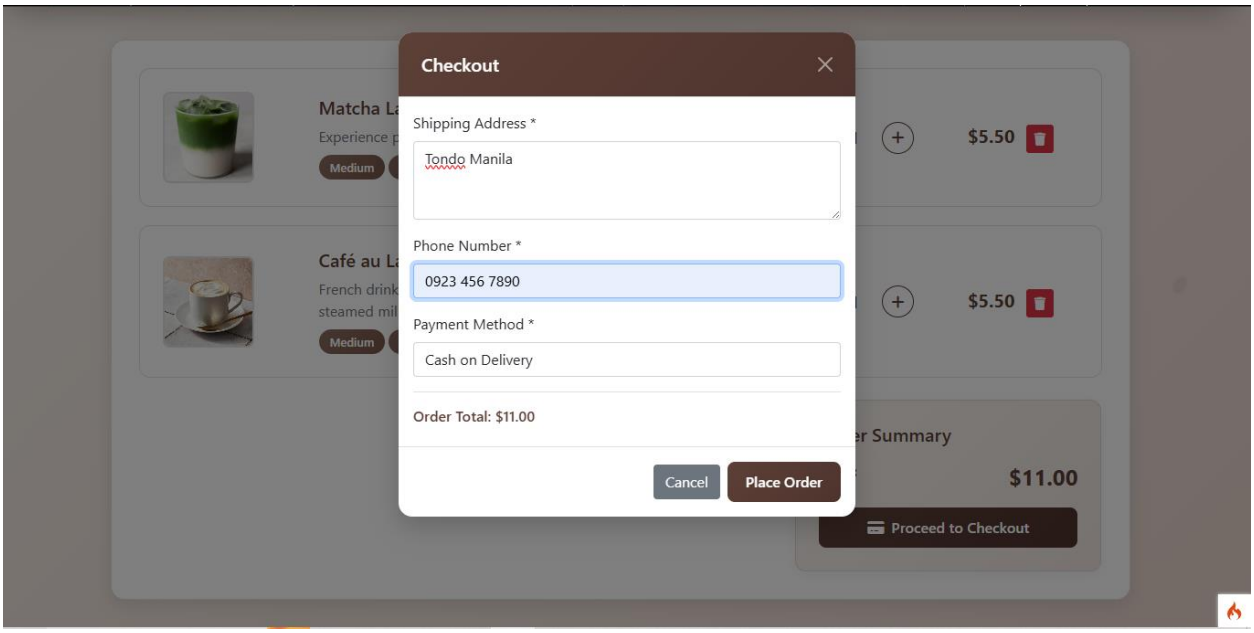

Figure 3.3 Pop-up place order

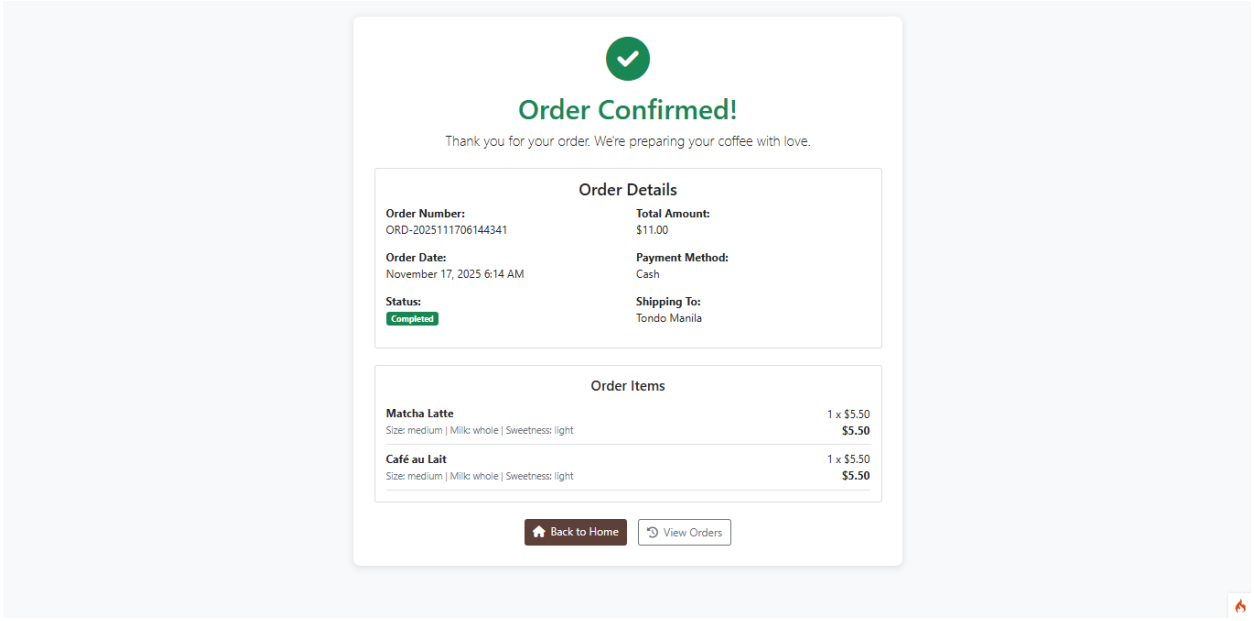

Figure 3.4 Order Confirmation

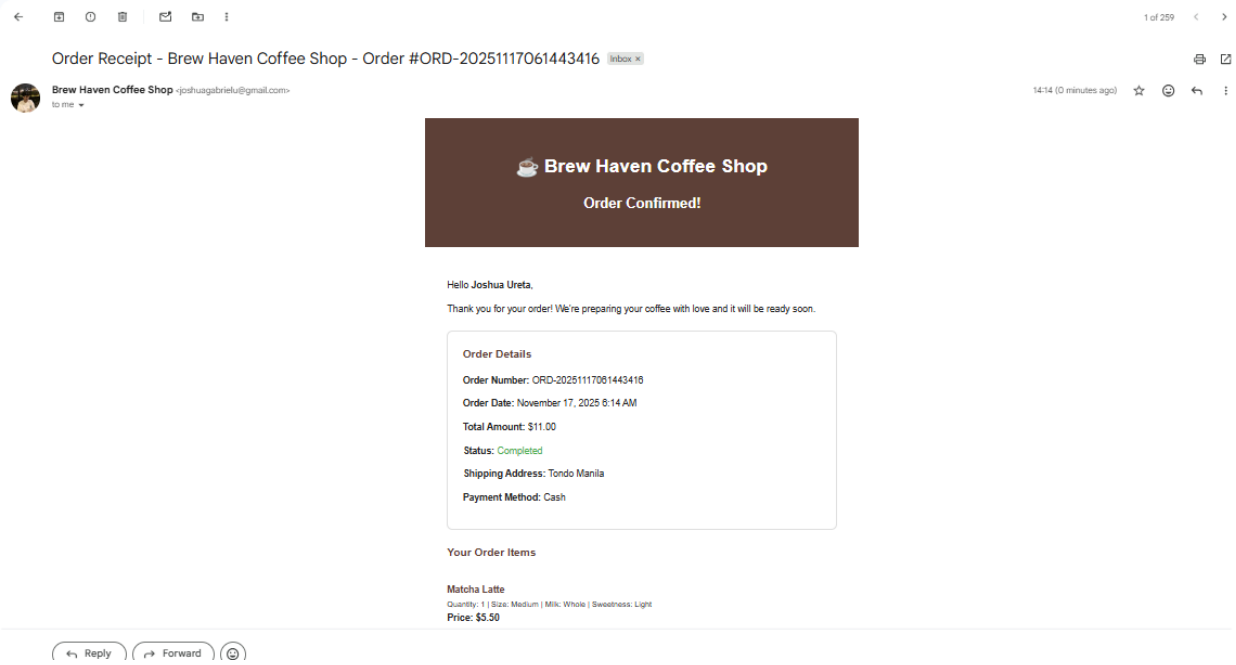

Figure 3.5 Order Confirmation through Gmail

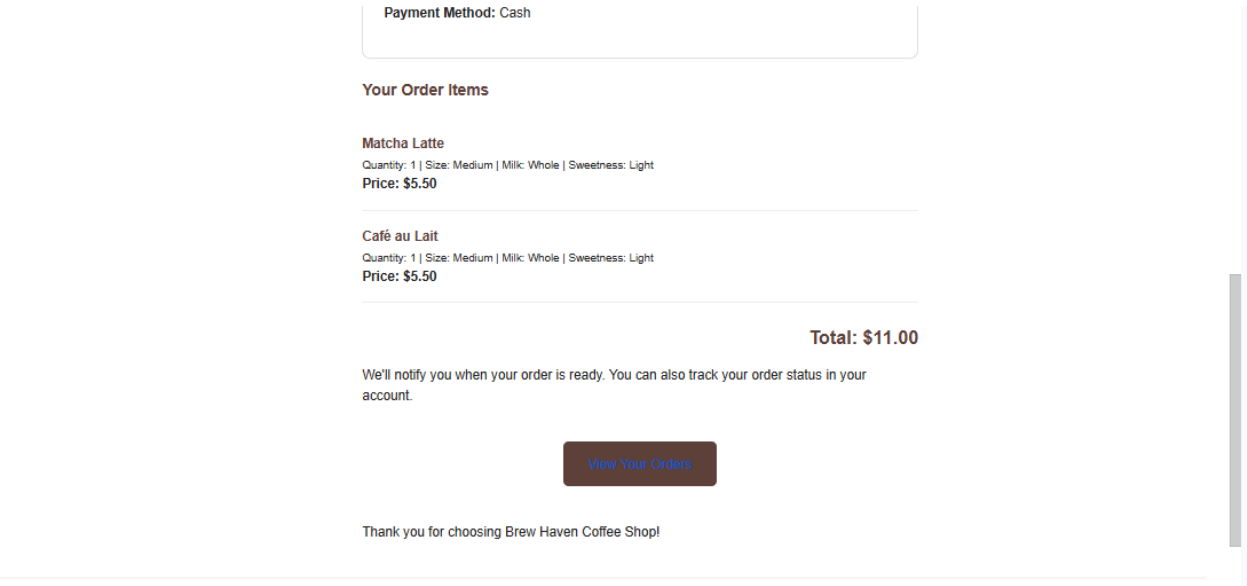

Figure 3.6 Order Confirmation through Gmail

Description:

This feature allows users to view a summary of their ordered items, including names, quantities, individual prices, and the total cost. Users can review their order and proceed to order confirmation to ensure everything is correct before finalizing. Once confirmed, the system stores the order details and updates inventory as needed. A confirmation email is automatically sent via Gmail, containing the order details, total price, order reference number, and estimated pickup or delivery time.

Codes and Explanations:

OrderController.php

Authentication & Validation

```
if (!session()->has('user_id')) {
    return redirect()->to('/login')->with('error', 'Please login to place order');
}
```

What happens: Checks if user is logged in before allowing order placement. Redirects to login with error message if not authenticated.

```
$validation->setRules([
    'shipping_address' => 'required|min_length[10]',
    'phone' => 'required|min_length[10]',
    'payment_method' => 'required'
]);
```

What happens: Validates checkout form fields - ensures address is provided and at least 10 characters, phone number is valid, and payment method is selected.

Database Transaction & Order Creation

```
$db->transStart();
```

What happens: Begins database transaction to ensure all operations succeed or fail together, maintaining data integrity.

```
$orderId = $orderModel->insert($orderData);
```

What happens: Creates main order record in database and retrieves the auto-generated order ID for linking order items.

```
foreach ($cartItems as $item) {
    $orderItemModel->insert($orderItemData);
}
```

What happens: Loops through each cart item and creates corresponding order item records with customization details preserved.

Cart Clearing & Transaction Completion

```
$cartModel->where('user_id', $userId)->delete();
```

What happens: Removes all items from user's cart after successful order creation to prevent duplicate orders.

```
$db->transComplete();
if ($db->transStatus() === FALSE) {
    throw new \Exception('Transaction failed');
}
```

What happens: Finalizes database transaction and checks if all operations were successful, throwing error if any failed.

CartController.php

Add to Cart with Customization

```
$price = $this->calculatePrice($product['price'], $size);
```

What happens: Calculates final price based on selected size (small/medium/large) with price adjustments.

```
$cartData = [
    'user_id' => $userId,
    'product_id' => $productId,
    'quantity' => $quantity,
    'size' => $size,
```

```
    'milk_type' => $milkType,
    'sweetness' => $sweetness,
    'special_instructions' => $specialInstructions,
    'price' => $price
];
```

What happens: Prepares complete cart item data including all customization options for storage in database.

Real-time Cart Updates

```
return json_encode([
    'success' => true,
    'summary' => $summary
]);
```

What happens: Returns JSON response for AJAX cart updates, allowing real-time quantity changes without page refresh.

Price Calculation Logic

```
private function calculatePrice($basePrice, $size)
{
    switch ($size) {
        case 'small': return $basePrice - 0.5;
        case 'large': return $basePrice + 0.5;
        default: return $basePrice;
    }
}
```

What happens: Applies fixed price adjustments based on drink size selection (-$0.50 for small, +$0.50 for large).

OrderItemModel.php

Related Data Retrieval

```
public function getOrderItems($orderId)
{
   return $this->select('order_items.*, products.name, products.image,
products.description')
            ->join('products', 'products.id = order_items.product_id')
            ->where('order_id', $orderId)
            ->findAll();
}
```

What happens: Fetches order items with complete product information by joining with products table for detailed order display.

Batch Order Items Retrieval

```
public function getItemsForOrders($orderIds)
{
   return $this->select('order_items.*, products.name, products.image')
            ->join('products', 'products.id = order_items.product_id')
            ->whereIn('order_id', $orderIds)
            ->findAll();
}
```

What happens: Retrieves items for multiple orders simultaneously, optimized for order history views showing multiple orders.

cart.php View

Empty Cart State

```
<?php if (empty($cartItems)): ?>
   <div class="empty-cart">
      <i class="fas fa-shopping-cart"></i>
```

```
        <h3>Your cart is empty</h3>
        <a href="/dashboard" class="btn btn-coffee-sm">
            <i class="fas fa-coffee me-2"></i>Browse Menu
        </a>
    </div>
<?php else: ?>
```

What happens: Displays friendly empty cart message with navigation button when no items are in cart.

Cart Item Display with Customization

```
<div class="d-flex flex-wrap gap-1">
    <span class="custom-badge"><?= ucfirst($item['size']) ?></span>
    <span class="custom-badge"><?= ucfirst(str_replace('_', ' ',
$item['milk_type'])) ?></span>
    <span class="custom-badge"><?= ucfirst($item['sweetness']) ?></span>
</div>
```

What happens: Shows customization options as styled badges for easy visual identification of drink preferences.

Quantity Update Forms

```
<form action="/update-cart" method="post" class="quantity-control">
    <?= csrf_field() ?>
    <input type="hidden" name="item_id" value="<?= $item['id'] ?>">
    <button type="submit" name="quantity" value="<?= $item['quantity'] - 1 ?>">
        <i class="fas fa-minus"></i>
    </button>
    <span class="quantity-display"><?= $item['quantity'] ?></span>
    <button type="submit" name="quantity" value="<?= $item['quantity'] + 1 ?>">
        <i class="fas fa-plus"></i>
    </button>
</form>
```

What happens: Provides increment/decrement buttons that submit form updates directly, maintaining security with CSRF protection.

CartModel.php

Smart Cart Item Management

```
public function addToCart($data)
{
    $existing = $this->getCartItem(
        $data['user_id'],
        $data['product_id'],
        $data['size'],
        $data['milk_type'],
        $data['sweetness']
    );

    if ($existing) {
        return $this->update($existing['id'], [
            'quantity' => $existing['quantity'] + $data['quantity']
        ]);
    } else {
        return $this->insert($data);
    }
}
```

What happens: Checks for identical customized items before adding to cart - merges quantities if same customization exists, otherwise creates new cart entry.

Stock Validation

```
public function validateCartStock($userId)
{
    foreach ($items as $item) {
        if ($item['quantity'] > $item['stock']) {
            $errors[] = "Insufficient stock for {$item['name']}. Only {$item['stock']} available.";
        }
    }
```

```
    return $errors;
}
```

What happens: Validates that cart quantities don't exceed available product stock before allowing checkout.

OrderModel.php

Automatic Order Number Generation

```
private function generateOrderNumber(array $data)
{
    $lastOrder = $this->orderBy('id', 'DESC')->first();
    $lastId = $lastOrder ? $lastOrder['id'] : 0;
    $nextId = $lastId + 1;
    $data['data']['order_number'] = 'ORD-' . str_pad($nextId, 6, '0', STR_PAD_LEFT);
    return $data;
}
```

What happens: Automatically creates sequential order numbers (ORD-000001, ORD-000002, etc.) for easy order tracking and reference.

Order Statistics

```
public function getOrderStats()
{
    return [
        'total_orders' => $this->countAll(),
        'pending_orders' => $this->where('status', 'pending')->countAllResults(),
        'completed_orders' => $this->where('status', 'completed')->countAllResults(),
        'total_revenue' => $this->selectSum('total_amount')->where('status', 'completed')-
>get()->getRow()->total_amount ?? 0
    ];
}
```

What happens: Provides comprehensive order statistics including counts by status and total revenue calculation for reporting.

order_confirmation.php

Order Summary Display

```
<p><strong>Order Number:</strong> <?= $order['order_number'] ?? 'ORD-' .
$order['id'] ?></p>
<p><strong>Order Date:</strong> <?= date('F j, Y g:i A',
strtotime($order['created_at'])) ?></p>
<p><strong>Total Amount:</strong> $<?= number_format($order['total_amount'],
2) ?></p>
```

What happens: Formats and displays key order information in readable format for customer confirmation.

Item Customization Details

```
<small>
   Quantity: <?= $item['quantity'] ?> |
   Size: <?= ucfirst($item['size']) ?> |
   Milk: <?= ucfirst(str_replace('_', ' ', $item['milk_type'])) ?> |
   Sweetness: <?= ucfirst($item['sweetness']) ?>
</small>
```

What happens: Shows all customization options for each ordered item ensuring customer receives exactly what they ordered.

Email Integration

```
// In Dashboard Controller processCheckout() method
if (!sendOrderConfirmationEmail($finalOrderData, $cartItems, $userEmail, $userName))
{
   log_message('error', 'Order confirmation email failed to send for order: ' . $orderId);
} else {
   log_message('info', 'Order receipt email sent successfully to: ' . $userEmail);
```

}

What happens: Sends formatted order confirmation email to customer after successful order placement and logs email delivery status for monitoring.

Summary: Users add items to their cart with customizations through the `CartController`, while the `CartModel` manages item storage and merges quantities as needed. At checkout, addresses and payment details are validated, and the `OrderController` processes the order within a database transaction, using `OrderModel` to generate the order number and store order data, and `OrderItemModel` to save individual item details and customizations. Finally, a confirmation email is sent via the `order_confirmation.php` template, the cart is cleared, and the user is redirected to the confirmation page

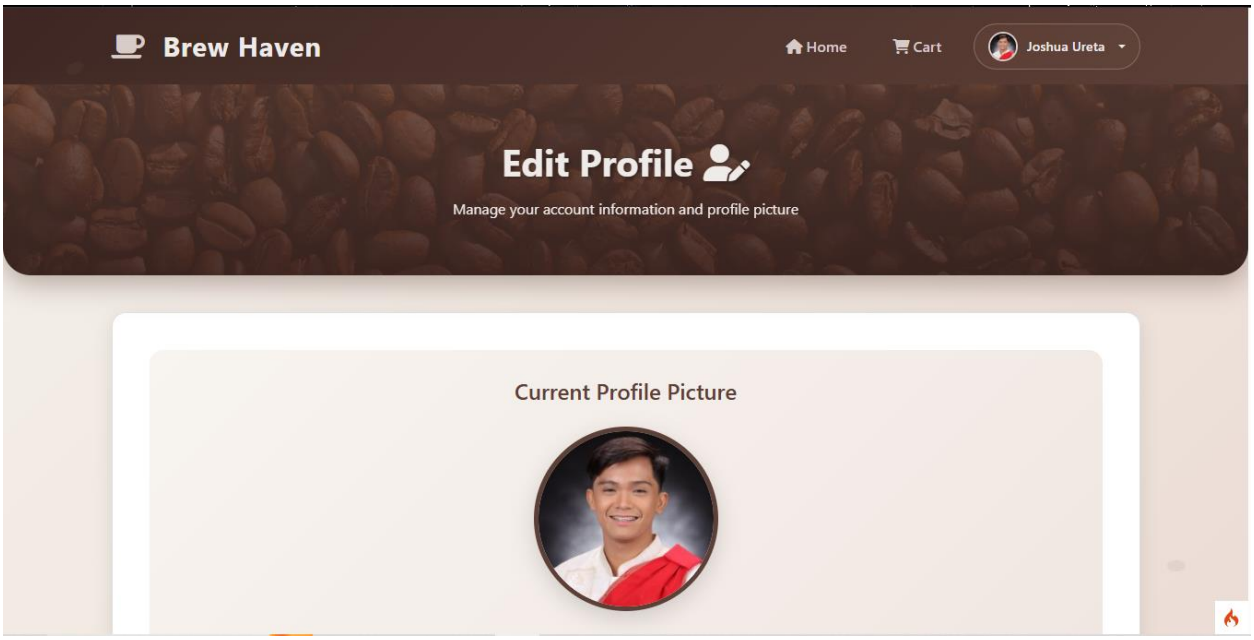

Figure 4.1 Profile Page

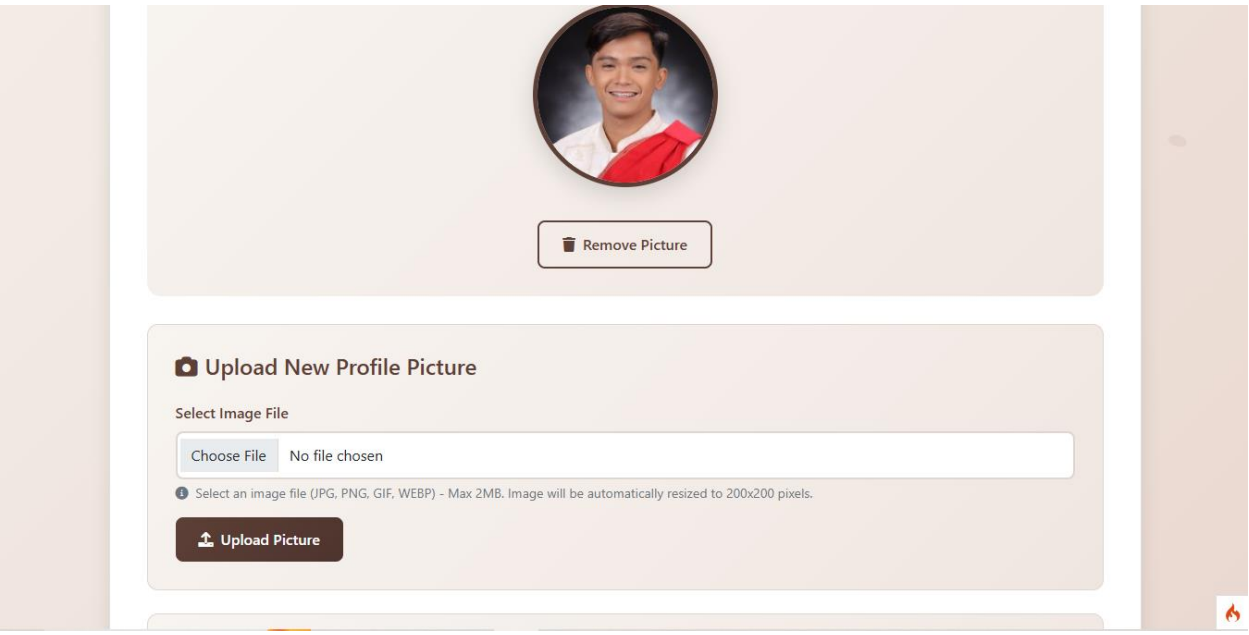

Figure 4.2 Profile Update Section

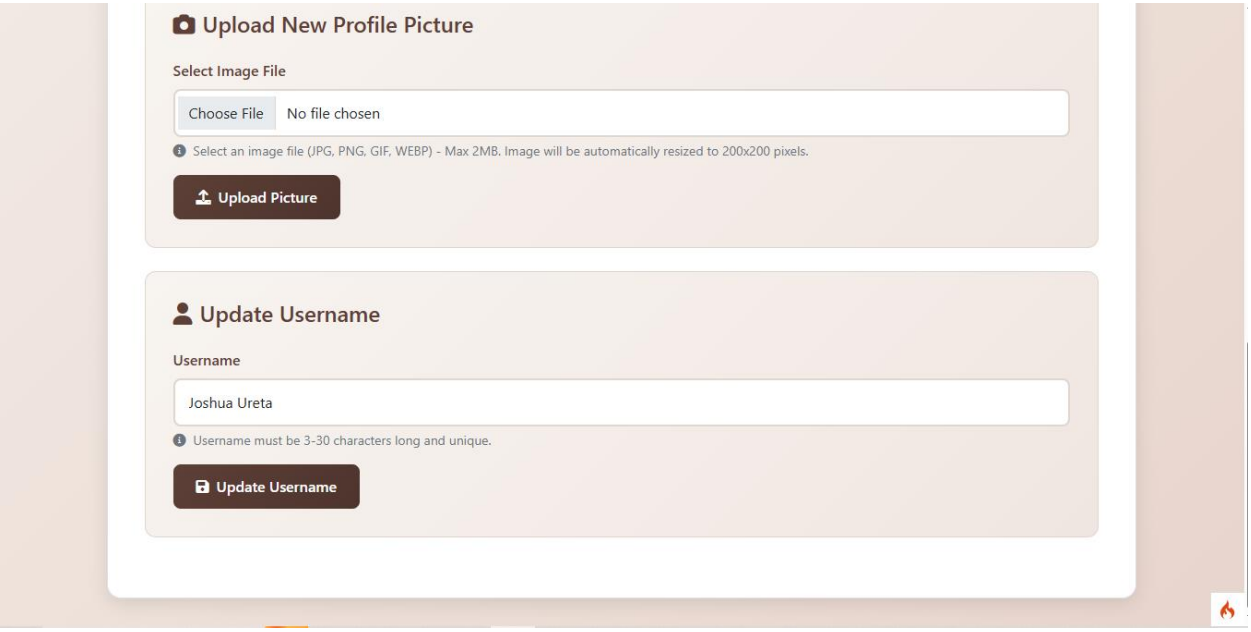

Figure 4.3 Profile Update Username Section

Description:

This page provides a user profile interface where customers can view and manage their core account information, including username, email address, and assigned role. Users can upload and change their profile picture, as well as update their username. A navigation button is included for easy return to the main dashboard, ensuring a seamless experience within the application.

Code and Explanation:

profile.php View

Profile Picture Display & Management

```
<?php if (!empty($user['profile_picture'])): ?>
   <img src="/uploads/profile_pics/<?= $user['profile_picture'] ?>"
        class="profile-picture"
        onerror="this.src='https://via.placeholder.com/200'"
        alt="Profile Picture">
<?php else: ?>
   <div class="profile-icon-main">
      <i class="fas fa-user"></i>
   </div>
<?php endif; ?>
```

What happens: Displays user's current profile picture if exists, otherwise shows a placeholder user icon with elegant styling.

```
<?php if (!empty($user['profile_picture'])): ?>
   <form action="/profile/deleteProfilePicture" method="post" class="d-inline">
      <?= csrf_field() ?>
      <button type="submit" class="btn btn-outline-coffee"
            onclick="return confirm('Are you sure you want to remove your profile
picture?')">
         <i class="fas fa-trash me-2"></i>Remove Picture
      </button>
   </form>
<?php endif; ?>
```

What happens: Shows remove picture button only when profile picture exists, with confirmation dialog to prevent accidental deletion.

File Upload Form with Validation

```
<form action="/profile/update" method="post" enctype="multipart/form-data">
   <?= csrf_field() ?>
```

```
<input type="file" name="profile_picture" class="form-control"
       accept="image/jpg,image/jpeg,image/png,image/gif,image/webp" required>
```

What happens: Creates secure file upload form with multipart encoding and restricts file types to common image formats for safety.

```
const fileInput = document.querySelector('input[type="file"]');
fileInput.addEventListener('change', function() {
    const file = this.files[0];
    const fileSize = file.size / 1024 / 1024;
    const validTypes = ['image/jpeg', 'image/jpg', 'image/png', 'image/gif', 'image/webp'];

    if (!validTypes.includes(file.type)) {
        alert('Please select a valid image file (JPG, PNG, GIF, WEBP).');
        this.value = '';
    } else if (fileSize > 2) {
        alert('File size must be less than 2MB.');
        this.value = '';
    }
});
```

What happens: Client-side validation checks file type and size before upload, preventing server processing of invalid files.

Dashboard.php Controller

Profile Data Retrieval & Session Update

```
public function profile()
{
    if (!session()->get('isLoggedIn')) {
        return redirect()->to('/login');
    }

    $userId = session()->get('id');
    $user = $this->userModel->find($userId);
```

```
    if (!$user) {
        return redirect()->to('/login');
    }

    $this->updateUserSession($user);
    $data['user'] = $user;
    $data['cart_count'] = $this->cartModel->getCartCount($userId);

    return view('profile', $data);
}
```

What happens: Fetches current user data from database and updates session to ensure profile information is synchronized across the application.

```
private function updateUserSession($user)
{
    $sessionData = [
        'id' => $user['id'],
        'username' => $user['username'],
        'email' => $user['email'],
        'profile_picture' => $user['profile_picture'] ?? null,
        'role' => $user['role'],
        'isLoggedIn' => true
    ];

    session()->set($sessionData);
}
```

What happens: Refreshes session data with latest user information from database after profile updates to maintain consistency.

ProfileModel.php

Profile Picture Update

```
public function updateProfilePicture($userId, $filename)
{
```

```
    return $this->update($userId, [
        'profile_picture' => $filename,
        'updated_at' => date('Y-m-d H:i:s')
    ]);
}
```

What happens: Updates user's profile picture filename in database and records update timestamp for tracking changes.

Username Validation & Update

```
public function isUsernameUnique($username, $excludeUserId = null)
{
    $builder = $this->where('username', $username);
    if ($excludeUserId) {
        $builder->where('id !=', $excludeUserId);
    }

    return $builder->countAllResults() === 0;
}
```

What happens: Checks if username is available by excluding current user from the search, preventing duplicate usernames in the system.

```
public function updateUsername($userId, $username)
{
    return $this->update($userId, [
        'username' => $username,
        'updated_at' => date('Y-m-d H:i:s')
    ]);
}
```

What happens: Updates username in database with current timestamp to track when the change occurred.

UserModel.php

User Validation Rules

```
protected $validationRules = [
    'username' => 'required|min_length[3]|max_length[100]',
    'email'    => 'required|valid_email',
    'password' => 'permit_empty|min_length[8]'
];
```

What happens: Defines validation rules for user data ensuring usernames are 3-100 characters and emails are properly formatted.

Email Uniqueness Check

```
public function isEmailUnique($email, $id)
{
    return $this->where('email', $email)
                ->where('id !=', $id)
                ->countAllResults() === 0;
}
```

What happens: Verifies email uniqueness while allowing current user to keep their existing email during profile updates.

File Upload Processing Logic

Image Upload & Processing

```
// In ProfileController update() method
$profilePicture = $this->request->getFile('profile_picture');
if ($profilePicture->isValid() && !$profilePicture->hasMoved()) {
    $newName = $profilePicture->getRandomName();
    $profilePicture->move(WRITEPATH . 'uploads/profile_pics', $newName);

    // Resize image to 200x200 pixels
    $image = \Config\Services::image()
```

```
        ->withFile(WRITEPATH . 'uploads/profile_pics/' . $newName)
        ->fit(200, 200, 'center')
        ->save(WRITEPATH . 'uploads/profile_pics/' . $newName);

    $this->profileModel->updateProfilePicture($userId, $newName);
}
```

What happens: Processes uploaded image file by generating unique filename, moving to uploads directory, and resizing to consistent 200x200 pixel dimensions.

File Security & Error Handling

```
// File validation before processing
$validationRules = [
    'profile_picture' => [
        'uploaded[profile_picture]',
        'mime_in[profile_picture,image/jpg,image/jpeg,image/png,image/gif,image/webp]',
        'max_size[profile_picture,2048]'
    ]
];

if (!$this->validate($validationRules)) {
    return redirect()->back()->with('error', 'Invalid file type or size exceeded 2MB limit.');
}
```

What happens: Server-side validation ensures uploaded files are valid images under 2MB size limit before processing.

Old File Cleanup

```
// Remove old profile picture when updating
$oldPicture = $this->profileModel->getUserById($userId)['profile_picture'];
if ($oldPicture && file_exists(WRITEPATH . 'uploads/profile_pics/' . $oldPicture)) {
    unlink(WRITEPATH . 'uploads/profile_pics/' . $oldPicture);
}
```

What happens: Deletes previous profile picture file from server when user uploads a new one to prevent storage waste.

Summary: Users access their profile page through the `Dashboard` controller, with profile data loaded from `UserModel` and `ProfileModel`. When uploading a new profile picture, client-side validation is performed, and the server validates the file type, size, and security before processing the image with a unique filename, deleting the old picture, and updating the database via `ProfileModel`. Username changes are validated, the session is refreshed with updated information, and success or error messages are displayed to the user via flashdata. This system provides secure, user-friendly profile management with proper file handling, validation, and data consistency across the application.

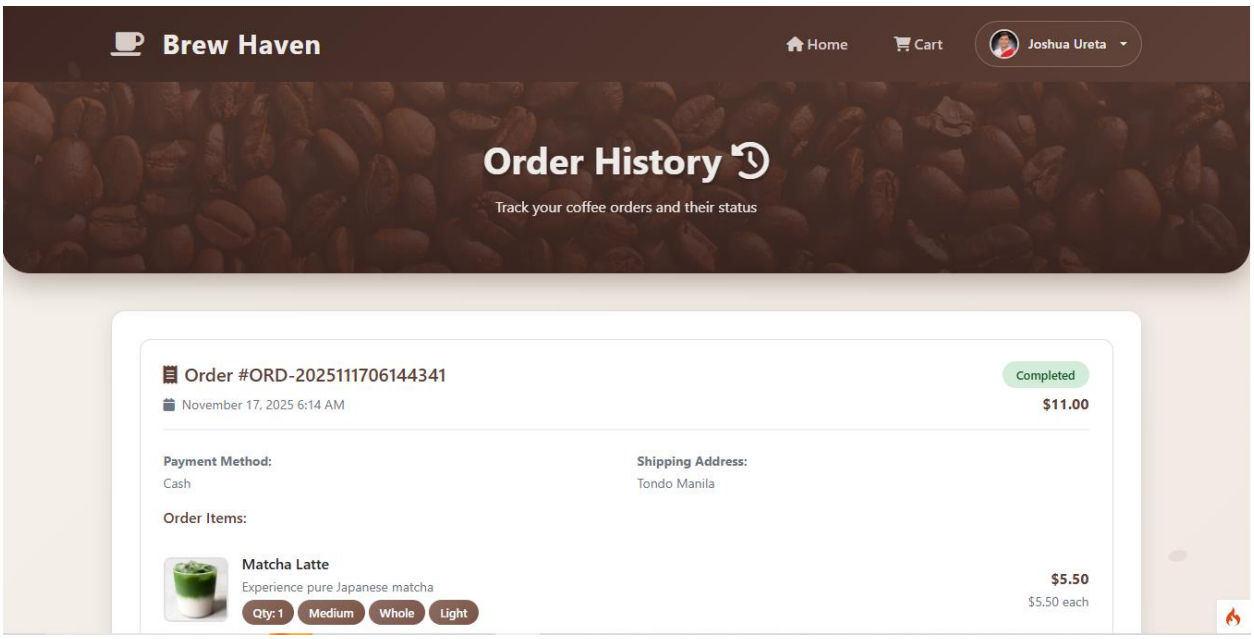

Figure 5.1 Order History Page

Description:

This page displays the user's past order history, allowing them to track all the items they have ordered. Each order shows detailed information, including product names, quantities, customization options, and order status. Users can quickly identify completed, pending, or canceled orders through status badges, and view individual order items for reference.

Code and Explanation:

OrderController.php

Order History Retrieval

```
public function orderHistory()
{
   if (!session()->has('user_id')) {
      return redirect()->to('/login');
   }

   $userId = session()->get('user_id');
   $orderModel = new OrderModel();
   $orderItemModel = new OrderItemModel();
   $cartModel = new CartModel();

   $orders = $orderModel->where('user_id', $userId)
                  ->orderBy('order_date', 'DESC')
                  ->findAll();
```

What happens: Retrieves all orders for the current user sorted by most recent first. Ensures users can only access their own order history.

```
$orderIds = array_column($orders, 'id');
$orderItems = [];
if (!empty($orderIds)) {
   $orderItems = $orderItemModel->getItemsForOrders($orderIds);
}
```

What happens: Fetches all order items for the user's orders in a single query for efficient data loading and display.

OrderModel.php

Automatic Order Number Generation

```
private function generateOrderNumber(array $data)
{
```

```
    if (!isset($data['data']['order_number'])) {
        $lastOrder = $this->orderBy('id', 'DESC')->first();
        $lastId = $lastOrder ? $lastOrder['id'] : 0;
        $nextId = $lastId + 1;
        $data['data']['order_number'] = 'ORD-' . str_pad($nextId, 6, '0', STR_PAD_LEFT);
    }
    return $data;
}
```

What happens: Automatically generates unique sequential order numbers in format ORD-000001 when new orders are created, ensuring easy tracking and reference.

User Order Retrieval

```
public function getUserOrders($userId)
{
    return $this->where('user_id', $userId)
                ->orderBy('created_at', 'DESC')
                ->findAll();
}
```

What happens: Fetches all orders belonging to a specific user, sorted by creation date descending to show newest orders first in the history.

OrderItemModel.php

Multi-Order Items Retrieval

```
public function getItemsForOrders($orderIds)
{
    return $this->select('order_items.*, products.name, products.image')
                ->join('products', 'products.id = order_items.product_id')
                ->whereIn('order_id', $orderIds)
                ->orderBy('order_id', 'DESC')
                ->orderBy('created_at', 'ASC')
```

```
            ->findAll();
}
```

What happens: Efficiently retrieves all items for multiple orders in one query, including product details for display, sorted by order ID and creation time.

orders.php View

Empty State Handling

```
<?php if (empty($orders)): ?>
   <div class="empty-orders">
      <i class="fas fa-clipboard-list"></i>
      <h3>No orders yet</h3>
      <p class="mb-4">Start your coffee journey by placing your first order!</p>
      <a href="/dashboard" class="btn btn-coffee">
         <i class="fas fa-coffee me-2"></i>Browse Menu
      </a>
   </div>
<?php else: ?>
```

What happens: Displays friendly empty state with call-to-action when user has no orders, encouraging them to start shopping.

Order Status Display

```
<span class="status-badge status-<?= $order['status'] ?>">
   <?= ucfirst($order['status']) ?>
</span>
```

What happens: Shows order status with color-coded badges (pending, confirmed, preparing, ready, completed, cancelled) for quick visual recognition.

Order Items with Customization

```
<div class="d-flex flex-wrap gap-1">
   <span class="custom-badge">Qty: <?= $item['quantity'] ?></span>
```

```
    <span class="custom-badge"><?= ucfirst($item['size']) ?></span>
    <span class="custom-badge"><?= ucfirst(str_replace('_', ' ',
$item['milk_type'])) ?></span>
    <span class="custom-badge"><?= ucfirst($item['sweetness']) ?></span>
</div>
```

What happens: Displays all drink customizations as visual badges including quantity, size, milk type, and sweetness level for complete order details.

Order Summary Display

```
<div class="order-header">
    <div class="row align-items-center">
        <div class="col-md-6">
            <h5 class="text-coffee mb-2">
                <i class="fas fa-receipt me-2"></i>Order #<?= $order['order_number'] ?>
            </h5>
            <small class="text-muted">
                <i class="fas fa-calendar me-1"></i>
                <?= date('F j, Y g:i A', strtotime($order['created_at'])) ?>
            </small>
        </div>
        <div class="col-md-6 text-md-end">
            <span class="status-badge status-<?= $order['status'] ?>">
                <?= ucfirst($order['status']) ?>
            </span>
            <div class="mt-2">
                <strong class="text-coffee">$<?= number_format($order['total_amount'],
2) ?></strong>
            </div>
        </div>
    </div>
</div>
```

What happens: Shows comprehensive order header with order number, date, status, and total amount in a clean, organized layout for easy scanning.

Summary: Users access their order history page through the `OrderController`, with an authentication check ensuring only logged-in users can view their history. Orders are retrieved from `OrderModel` for the current user, and order items are efficiently fetched via `OrderItemModel`, with data prepared to include cart count for navigation display. The view renders orders with status badges and customization details, shows an empty state with browse encouragement if no orders exist, and features a responsive design that adapts for both mobile and desktop viewing.

Admin Side

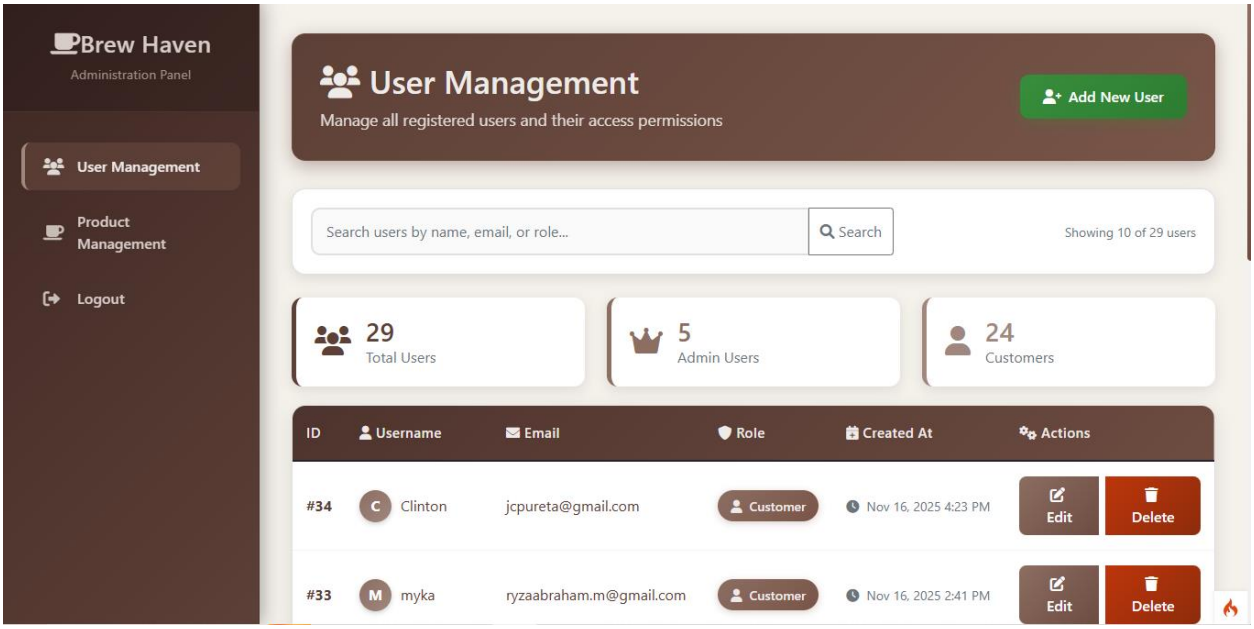

Figure 6.1 User Management Page

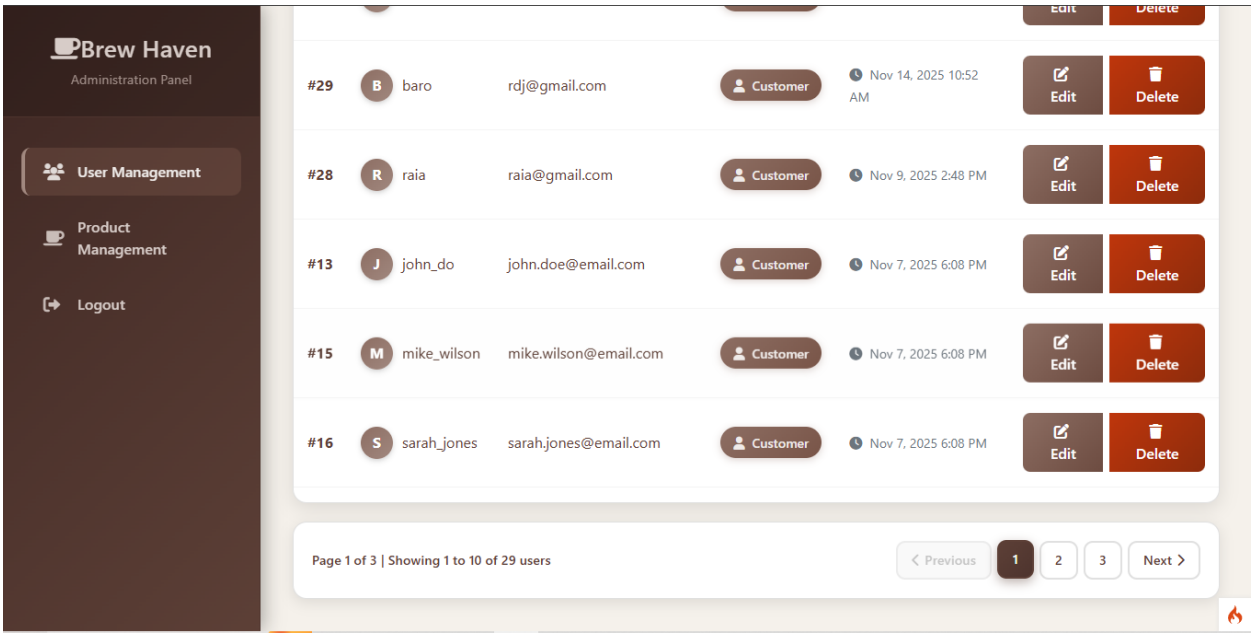

Figure 6.2 User Management Pagination

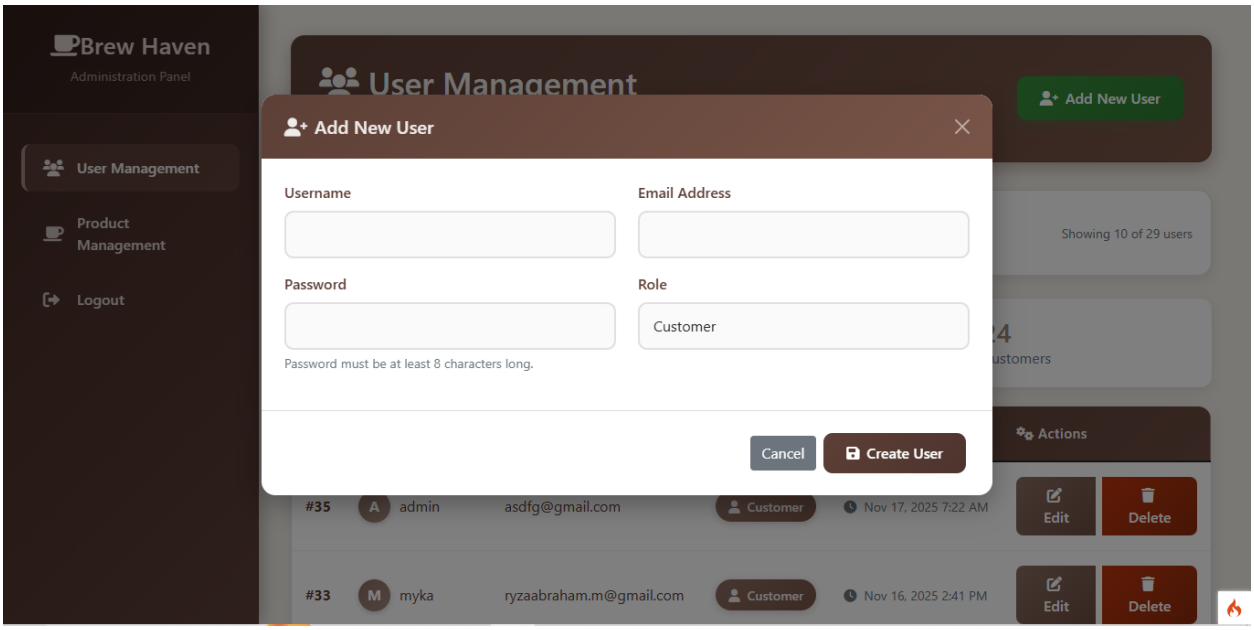

Figure 6.3 Add User

Description:

The user management panel provides a comprehensive user management functionality, allowing administrators to view a table of all users, add new ones with specific roles, and edit or delete existing accounts. It shows a layout for presenting user data and incorporates modal forms for the "Add New User" and deletion confirmation actions to maintain an organized interface. Moreover, it has a confirmation prompt to ensure secure deletion of users.

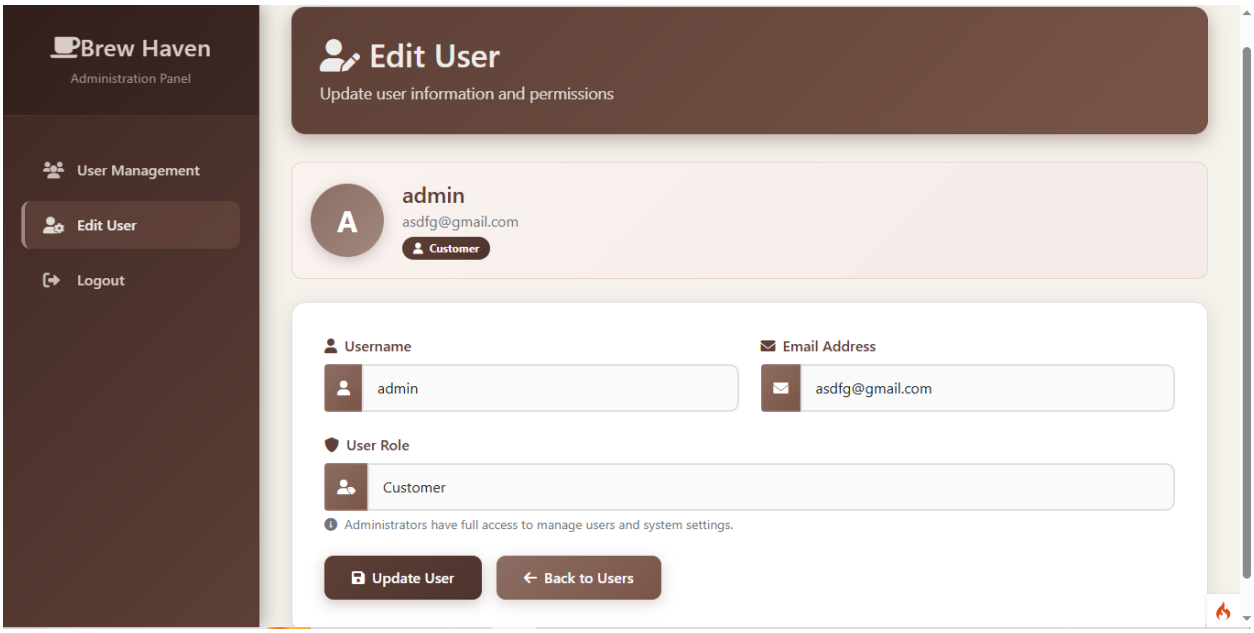

Figure 6.4 Edit User

Description:

This page of the admin provides the functionality for administrators to edit an existing user's information, including their username, email address, and system role. It includes informative role-based descriptions and offers clear action buttons to either confirm the update or cancel the changes which ensures a straightforward and secure user management process.

Codes and Explanation:

Admin.php Controller

Admin Authentication & Authorization

```
public function users()
{
   if (!session()->get('isLoggedIn') || session()->get('role') !== 'admin') {
      return redirect()->to('/dashboard');
   }
```

What happens: Checks if user is logged in AND has admin role before allowing access to admin functions. Redirects regular users to dashboard.

User Search & Pagination

```
$search = $this->request->getGet('search');

if ($search) {
    $this->userModel->groupStart()
        ->like('username', $search)
        ->orLike('email', $search)
        ->orLike('role', $search)
        ->groupEnd();
}
```

What happens: Implements search functionality that filters users by username, email, or role using database LIKE queries with proper grouping.

```
$data = [
    'users' => $this->userModel->orderBy('created_at', 'DESC')->paginate($this->perPage),
    'pager' => $this->userModel->pager,
    'stats' => $this->userModel->getUserStats(),
    'search' => $search
];
```

What happens: Prepares data for view including paginated users, pagination object, user statistics, and search term for display.

UserModel.php

User Statistics Calculation

```
public function getUserStats()
{
    return [
        'total'    => $this->countAll(),
        'admin'    => $this->where('role', 'admin')->countAllResults(),
        'customer' => $this->where('role', 'customer')->countAllResults()
```

```
    ];
}
```

What happens: Calculates and returns user statistics including total users, admin count, and customer count for dashboard display.

Email Uniqueness Validation

```
public function isEmailUnique($email, $id)
{
    return $this->where('email', $email)
                ->where('id !=', $id)
                ->countAllResults() === 0;
}
```

What happens: Checks if email already exists for other users during profile updates, preventing duplicate emails while allowing current user to keep their email.

users.php View

User Statistics Display

```
<div class="row">
    <div class="col-md-4">
        <div class="stats-card total-users">
            <div class="d-flex align-items-center">
                <div class="me-3">
                    <i class="fas fa-users fa-2x" style="color: #5d4037;"></i>
                </div>
                <div>
                    <h3 class="mb-0" style="color: #5d4037;"><?= $stats['total'] ?? 0 ?></h3>
                    <p class="mb-0 text-muted">Total Users</p>
                </div>
            </div>
        </div>
```

```
    </div>
```

What happens: Displays user statistics in visually appealing cards with icons and counts for quick admin overview.

Role-Based Badge Styling

```
<span class="role-badge <?= $user['role'] === 'admin' ? 'role-admin' : 'role-customer' ?>">
    <i class="fas fa-<?= $user['role'] === 'admin' ? 'crown' : 'user' ?> me-1"></i>
    <?= ucfirst($user['role'] ?? 'customer') ?>
</span>
```

What happens: Shows user roles with different colored badges and icons - crown for admins, user icon for customers for easy visual identification.

User Avatar Generation

```
<div class="user-avatar">
    <?= strtoupper(substr($user['username'], 0, 1)) ?>
</div>
```

What happens: Creates visual avatars using the first letter of usernames with consistent styling when profile pictures aren't available.

Advanced Pagination System

```
<div class="custom-pagination">
    <?php
    $currentPage = $pager->getCurrentPage();
    $pageCount = $pager->getPageCount();

    // Previous page button
    if ($currentPage > 1): ?>
        <a href="<?= $pager->getPreviousPageUri() ?>" class="page-btn">
            <i class="fas fa-chevron-left"></i> <span>Previous</span>
        </a>
```

```
<?php else: ?>
    <span class="page-btn disabled">
        <i class="fas fa-chevron-left"></i> <span>Previous</span>
    </span>
<?php endif; ?>
```

What happens: Implements custom pagination with previous/next buttons, page numbers, and disabled states for better user navigation.

Admin.php

Create User with Validation

```
public function createUser()
{
    if (!session()->get('isLoggedIn') || session()->get('role') !== 'admin') {
        return redirect()->to('/dashboard');
    }

    $validation = \Config\Services::validation();
    $validation->setRules([
        'username' => 'required|min_length[3]|max_length[100]',
        'email' => 'required|valid_email|is_unique[users.email]',
        'password' => 'required|min_length[8]',
        'role' => 'required|in_list[customer,admin]'
    ]);
```

What happens: Validates new user data including username length, email format and uniqueness, password strength, and valid role selection.

```
$data = [
    'username' => $this->request->getPost('username'),
    'email' => $this->request->getPost('email'),
    'password' => password_hash($this->request->getPost('password'),
PASSWORD_DEFAULT),
```

```
    'role' => $this->request->getPost('role')
];
```

What happens: Prepares user data with securely hashed password before saving to database for security.

Update User with Email Uniqueness Check

```
if ($email !== $currentUser['email']) {
    if (!$userModel->isEmailUnique($email, $id)) {
        $session->setFlashdata('error', 'Email already exists for another user.');
        return redirect()->to('/admin/users/edit/' . $id);
    }
}
```

What happens: Checks email uniqueness only when email is changed, preventing duplicate emails while allowing unchanged emails.

Delete User with Confirmation

```
public function deleteUser($id)
{
    if (!session()->get('isLoggedIn') || session()->get('role') !== 'admin') {
        return redirect()->to('/dashboard');
    }

    $userModel = new UserModel();
    $session = session();

    if ($userModel->delete($id)) {
        $session->setFlashdata('success', 'User deleted successfully!');
    } else {
        $session->setFlashdata('error', 'Failed to delete user.');
    }

    return redirect()->to('/admin/users');
```

}

What happens: Safely deletes user accounts with proper authorization checks and provides success/error feedback.

edit_user.php View

User Profile Header

```
<div class="user-info-card">
   <div class="d-flex align-items-center">
      <div class="user-avatar-large">
         <?= strtoupper(substr($user['username'], 0, 1)) ?>
      </div>
      <div>
         <h4 class="mb-1" style="color: #5d4037;"><?= esc($user['username']) ?></h4>
         <p class="mb-1 text-muted"><?= esc($user['email']) ?></p>
         <span class="badge" style="background: linear-gradient(135deg, #5d4037,
#4e342e); color: white; padding: 6px 12px; border-radius: 15px;">
            <i class="fas fa-<?= $user['role'] === 'admin' ? 'crown' : 'user' ?> me-1"></i>
            <?= ucfirst($user['role']) ?>
         </span>
      </div>
   </div>
</div>
```

What happens: Displays current user information with large avatar, username, email, and role badge for context during editing.

Pre-filled Edit Form

```
<select class="form-select" id="role" name="role" required>
   <option value="customer" <?= $user['role'] == 'customer' ? 'selected' :
" ?>>Customer</option>
   <option value="admin" <?= $user['role'] == 'admin' ? 'selected' :
```

" ?>>Administrator</option>
</select>

What happens: Pre-selects the current user role in dropdown to show existing value and allow easy modification.

Enhanced Form Inputs

  <input type="text" class="form-control" id="username" name="username" value="<?= esc($user['username']) ?>" required>

What happens: Uses input groups with icons for better visual design and includes proper escaping of user data for security.

Summary: Admins access user management through the `Admin` controller with role verification, retrieving users via `UserModel` with search and pagination, while statistics are calculated for dashboard overview cards. The view displays a users table with roles, avatars, and action buttons, allowing admins to create new users with validation and secure password hashing, edit users with email uniqueness checks, and delete users with confirmation and feedback, supported by search and pagination for efficient management. This system provides comprehensive user management for administrators with proper security, validation, and a user-friendly interface for managing all registered users in the application.

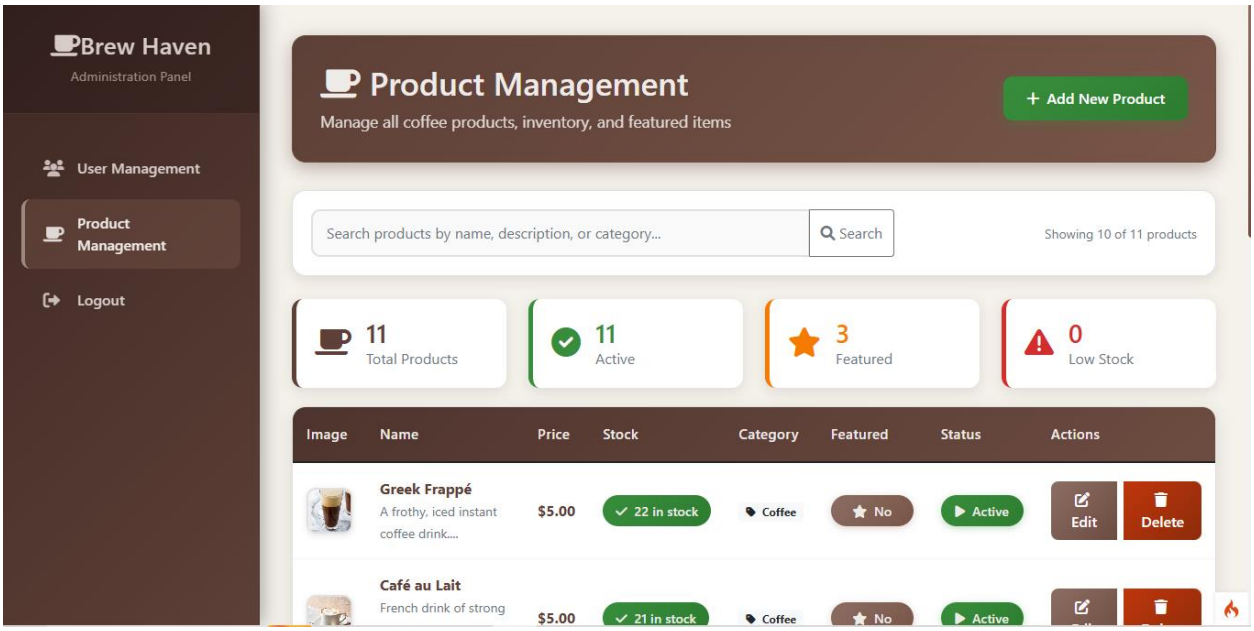

Figure 7.1 Product Management

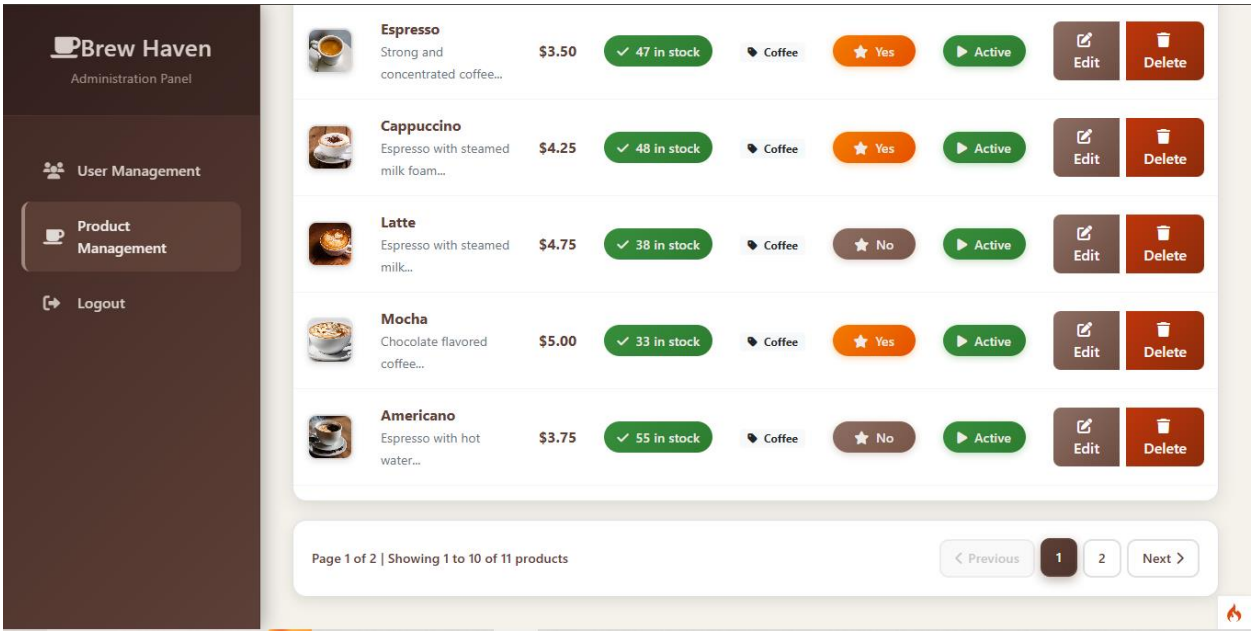

Figure 7.2 Product Management

Description:

This feature provides administrators with complete control over the product catalog, enabling the addition of new items and modification of all existing details. It features a pagination system to facilitate efficient navigation through extensive inventories. The interface consolidates key management functions, allowing for streamlined updates to product information, categories, and stock levels.

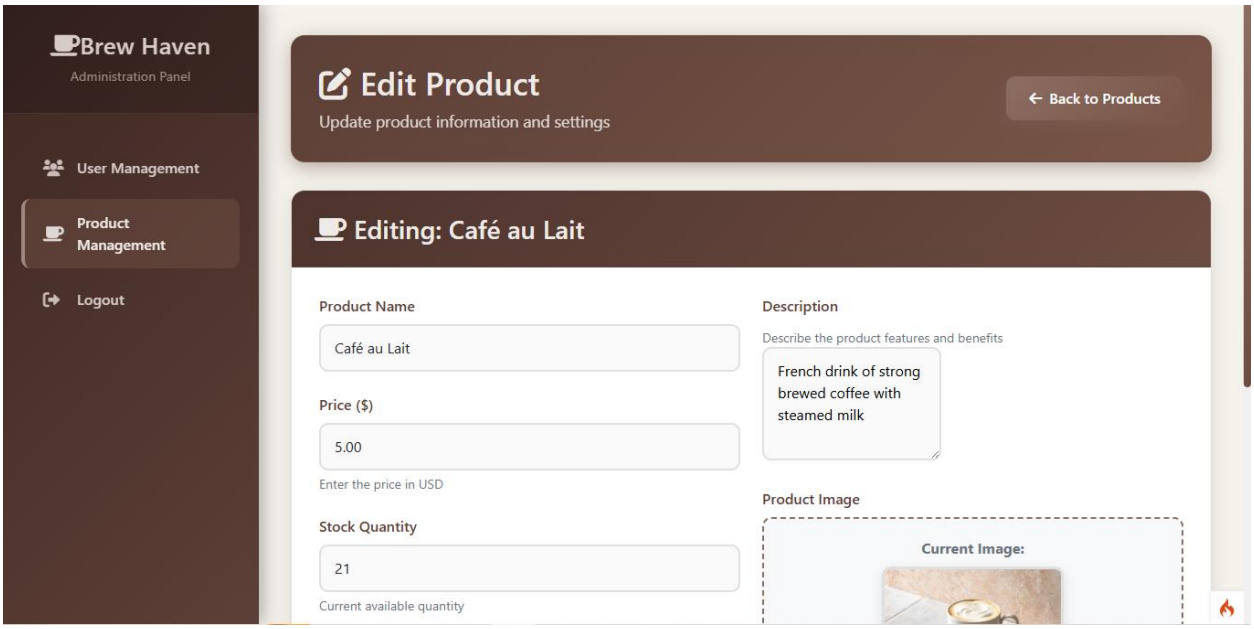

Figure 7.3 Edit Product

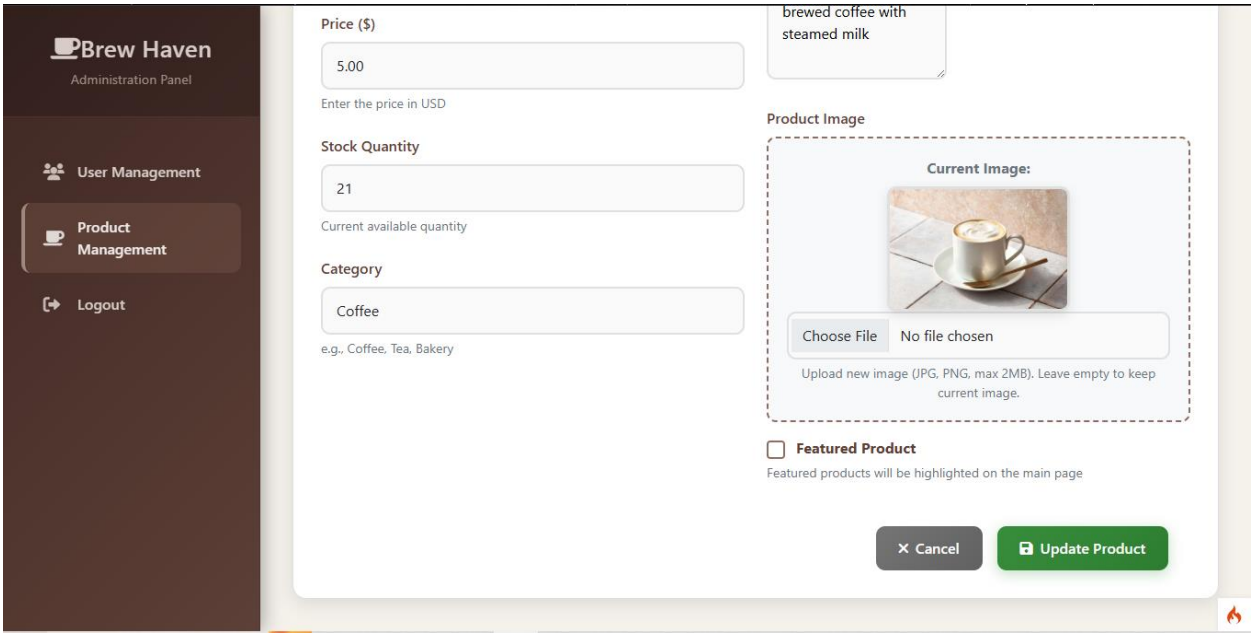

Figure 7.4 Edit Product

Description:

This feature allows you to edit every aspect of a product, including its price, stock quantity, description, category, image, and featured product status.

Codes and Explanations:

ProductModel.php

Product Availability Filtering

```
public function getAvailableProducts()
{
    return $this->where('status', 'active')
                ->where('stock >', 0)
                ->orderBy('created_at', 'DESC')
                ->findAll();
}
```

What happens: Retrieves only active products with positive stock for customer-facing displays. Ensures customers only see purchasable items.

Comprehensive Product Statistics

```
public function getProductStats()
{
    return [
        'total' => $this->countAll(),
        'active' => $this->where('status', 'active')->countAllResults(),
        'inactive' => $this->where('status', 'inactive')->countAllResults(),
        'featured' => $this->where('is_featured', 1)->countAllResults(),
        'low_stock' => $this->where('stock <', 10)->where('stock >', 0)->countAllResults(),
        'out_of_stock' => $this->where('stock', 0)->countAllResults()
    ];
}
```

What happens: Calculates detailed product statistics including stock levels and status counts for admin dashboard insights.

Admin.php Controller

Product Search & Pagination

```
public function products()
{
    if (!session()->get('isLoggedIn') || session()->get('role') !== 'admin') {
        return redirect()->to('/dashboard');
    }

    $search = $this->request->getGet('search');

    if ($search) {
        $this->productModel->groupStart()
            ->like('name', $search)
            ->orLike('description', $search)
            ->orLike('category', $search)
            ->groupEnd();
    }
```

What happens: Implements product search functionality that filters by name, description, or category using grouped database queries.

Product Creation with Image Upload

```
public function createProduct()
{
    if (!session()->get('isLoggedIn') || session()->get('role') !== 'admin') {
        return redirect()->to('/dashboard');
    }

    $validation = \Config\Services::validation();
    $validation->setRules([
        'name' => 'required|min_length[3]|max_length[255]',
        'description' => 'required|min_length[10]',
        'price' => 'required|decimal',
        'stock' => 'required|integer',
        'category' => 'required',
```

```
        'image' =>
'uploaded[image]|max_size[image,2048]|is_image[image]|mime_in[image,image/jpg,imag
e/jpeg,image/png]'
    ]);
```

What happens: Validates product data including image upload requirements for type, size, and format before processing.

```
$image = $this->request->getFile('image');
$imageName = null;

if ($image->isValid() && !$image->hasMoved()) {
    $imageName = $image->getRandomName();
    $image->move(ROOTPATH . 'public/uploads/products', $imageName);
}
```

What happens: Processes uploaded product images by generating unique filenames and moving files to the products upload directory.

Product Update with Image Management

```
$image = $this->request->getFile('image');
$imageName = $currentProduct['image'];

if ($image && $image->isValid() && !$image->hasMoved()) {
    if ($imageName && file_exists(ROOTPATH . 'public/uploads/products/' .
$imageName)) {
        unlink(ROOTPATH . 'public/uploads/products/' . $imageName);
    }

    $imageName = $image->getRandomName();
    $image->move(ROOTPATH . 'public/uploads/products', $imageName);
}
```

What happens: Handles image updates by deleting old product images and replacing them with new uploaded images to manage storage.

Product Deletion with File Cleanup

```
public function deleteProduct($id)
{
    if (!session()->get('isLoggedIn') || session()->get('role') !== 'admin') {
        return redirect()->to('/dashboard');
    }

    $product = $productModel->find($id);

    if ($product) {
        if ($product['image'] && file_exists(ROOTPATH . 'public/uploads/products/' .
$product['image'])) {
            unlink(ROOTPATH . 'public/uploads/products/' . $product['image']);
        }
```

What happens: Removes product image files from server storage when products are deleted to prevent orphaned files.

products.php View

Product Statistics Dashboard

```
<div class="row">
    <div class="col-md-3">
        <div class="stats-card total-products">
            <div class="d-flex align-items-center">
                <div class="me-3">
                    <i class="fas fa-coffee fa-2x" style="color: #5d4037;"></i>
                </div>
                <div>
                    <h3 class="mb-0" style="color: #5d4037;"><?= $stats['total'] ?? 0 ?></h3>
                    <p class="mb-0 text-muted">Total Products</p>
                </div>
            </div>
        </div>
```

```
    </div>
```

What happens: Displays product statistics in color-coded cards with icons for quick admin overview of inventory status.

Dynamic Stock Status Badges

```
<?php
$stockClass = 'stock-high';
if ($product['stock'] == 0) {
    $stockClass = 'stock-out';
} elseif ($product['stock'] < 10) {
    $stockClass = 'stock-low';
}
?>
<span class="stock-badge <?= $stockClass ?>">
    <i class="fas fa-<?= $product['stock'] == 0 ? 'times' : ($product['stock'] < 10 ?
'exclamation-triangle' : 'check') ?> me-1"></i>
    <?= $product['stock'] ?> in stock
</span>
```

What happens: Shows stock levels with color-coded badges and icons that change based on inventory quantity for visual alerts.

Product Status & Featured Badges

```
<span class="status-badge <?= $product['status'] === 'active' ? 'status-active' : 'status-
inactive' ?>">
    <i class="fas fa-<?= $product['status'] === 'active' ? 'play' : 'pause' ?> me-1"></i>
    <?= ucfirst($product['status']) ?>
</span>

<span class="featured-badge <?= $product['is_featured'] ? 'featured-yes' : 'featured-
no' ?>">
    <i class="fas fa-<?= $product['is_featured'] ? 'star' : 'star' ?> me-1"></i>
    <?= $product['is_featured'] ? 'Yes' : 'No' ?>
```

```
</span>
```

What happens: Displays product status and featured status with different colored badges and appropriate icons for quick identification.

Product Image Display

```
<td>
   <?php if ($product['image']): ?>
      <img src="<?= base_url('uploads/products/' . $product['image']) ?>" alt="<?=
esc($product['name']) ?>" class="product-image">
   <?php else: ?>
      <div class="product-image-placeholder">
         <i class="fas fa-coffee"></i>
      </div>
   <?php endif; ?>
</td>
```

What happens: Shows product thumbnails with graceful fallback to coffee icon placeholder when no image exists.

edit_product.php View

Current Image Display with Preview

```
<?php if ($product['image']): ?>
   <div class="image-preview-container mb-3">
      <p class="text-muted mb-2"><strong>Current Image:</strong></p>
      <img src="<?= base_url('uploads/products/' . $product['image']) ?>" alt="<?=
esc($product['name']) ?>" class="current-image" style="max-width:200px;max-
height:200px;">
   </div>
<?php else: ?>
   <div class="image-preview-container mb-3">
      <i class="fas fa-coffee fa-3x text-muted mb-2"></i>
      <p class="text-muted mb-0">No image currently set</p>
   </div>
```

```
<?php endif; ?>
```

What happens: Displays current product image with proper sizing and fallback placeholder when no image exists.

Live Image Preview JavaScript

```
imageInput.addEventListener('change', function(e){
    const file = e.target.files[0];
    if(file){
        const reader = new FileReader();
        reader.onload = function(e){
            if(currentImage){
                currentImage.src = e.target.result;
            } else {
                const previewContainer = document.querySelector('.image-preview-
container');
                if(previewContainer){
                    previewContainer.innerHTML = `<p class="text-muted mb-2"><strong>New
Image Preview:</strong></p>
                        <img src="${e.target.result}" alt="Preview" class="current-image"
style="max-width:200px;max-height:200px;">`;
                }
            }
        };
        reader.readAsDataURL(file);
    }
});
```

What happens: Provides real-time image preview when admin selects a new product image file for upload.

Pre-filled Edit Form

```
<input type="text" class="form-control" id="name" name="name" value="<?=
esc($product['name']) ?>" required>
```

```
<input type="number" step="0.01" class="form-control" id="price" name="price"
value="<?= $product['price'] ?>" required>

<div class="form-check">
   <input class="form-check-input" type="checkbox" id="is_featured"
name="is_featured" value="1" <?= $product['is_featured'] ? 'checked' : '' ?>>
   <label class="form-check-label" for="is_featured"><strong>Featured
Product</strong></label>
</div>
```

What happens: Pre-populates all form fields with current product data including checked state for featured products.

Summary: Admins access to product management with role verification, viewing products with search, pagination, and dashboard statistics. They can create new products with image upload and validation, edit existing products with image replacement, manage stock via visual badges and quantity updates, and flag featured products while controlling product visibility to customers. The system also handles image uploads, previews, and deletions, supports search by name, description, or category, and provides a responsive design across all device sizes. This system provides comprehensive product management for administrators with robust image handling, inventory tracking, and a user-friendly interface for managing the coffee shop product catalog.

Reflection: Learning and Challenges

This project taught me the fundamental importance of thorough planning from the very beginning. I learned that designing a solid database structure first made building all subsequent features significantly smoother. Implementing clear user roles for customers and administrators from the start helped organize system access, and establishing robust form validation early on prevented messy data issues down the line. These experiences solidified my understanding that good initial planning is crucial for preventing major headaches later in development.

Furthermore, I discovered how security and user experience are deeply interconnected. Proper login protection and input validation not only kept user information safe but also contributed to a professional feel. Developing features like pagination for product listings and order history taught me how to manage large datasets effectively, preventing slow load times and overwhelming interfaces, which was vital for maintaining

the site's speed and usability. Crucially, I also learned that you need a strong foundation of personal knowledge to guide a project to completion; you cannot rely on AI, as troubleshooting technical issues like database connections and refining the ordering process into something intuitive required my own problem-solving skills and understanding. Ultimately, this project helped me see how every component of a website must work together seamlessly, and that this integration depends on the developer's own grasp of the system.

JOSHUA GABRIEL P. URETA

FRONT-END DEVELOPER

CONTACT

0972281234

joshuagabrielu@gmail.com

Tondo, Manila

PROFILE SUMMARY

Joshua is a dedicated and reliable worker known for his strong work ethic and commitment to task completion. Despite not having direct work experience, he is highly motivated to learn and adapt to new challenges. His technical skills in web development are noteworthy, with experience in wireframing and front-end frameworks, and he is proficient in coding languages such as C++, HTML, CSS, and JavaScript. Joshua's eagerness to learn, combined with his problem-solving skills and attention to detail, makes him a valuable addition to any team.

EDUCATION

2016-2020
LIAN INSTITUTE
Junior Highschool

2020 - 2022
LYCEUM OF THE PHILIPPINES
Senior Highschool

2022 - 2027
FEU INSTITUTE OF TECHNOLOGY
Bachelor of Information In Techology With the Specialization in Web and Mobile Development

SKILLS

- Programming Proficiency: Knowledge of coding languages such as C++, HTML, CSS, and JavaScript.
- Web Design and Development: Experience with wireframing, UX/UI design, and front-end frameworks.
- Collaboration and Communication: Strong teamwork abilities, effective verbal and written communication skills.
- Problem-Solving and Analytical Thinking: Ability to analyze problems, think creatively, and propose effective solutions.
- Project Management and Time Management: Skills in planning, organizing tasks, and meeting deadlines.

ACHIEVEMENTS

Successfully obtained multiple technical certifications, including:

- Python Programming Certification - Demonstrated proficiency in coding, automation, and software development fundamentals.
- HTML & CSS Certification - Acquired advanced front-end web design and responsive website development skills.
- Database Management Certification - Gained strong knowledge of relational databases, SQL, and data modeling techniques.
- Networking Certifications (Cisco - Networking 1 & 2) - Developed a solid understanding of network fundamentals, configuration, and troubleshooting using Cisco technologies.

Academic & Research Contributions:

- Presented at various research paper seminars, showcasing technical expertise and effective communication skills.
- Published a research paper in a recognized academic platform/journal.