VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

WEB PROGRAMMING

MUSIC ECOMMERCE WEBSITE - HIPHOP

Major: Computer Science

Supervisors: Nguyễn Hữu Hiếu

—o0o—

Students: Phạm Võ Quang Minh - 2111762
Đỗ Nguyễn An Huy - 2110193

Email: huy.do862003@hcmut.edu.vn

HCMC, 05/2025

Contents

ID	Name	Shared work	Personal work
2110193	Đỗ Nguyễn An Huy	• Design application's MVC architecture • Design user's templates • Design admin's templates • Implement user administration system	Personal task #1 • Home page • Contact page • Home page management page • Contact management page

Chapter I: Introduction

Music is one of our passions. Therefore, as an idea, we tried to incorporate music elements into this assignment and build a music instrument ecommerce website. This assignment is about building a music ecommerce website that advertises and sells musical instruments, which aims at children as our target audience. We're directly inspired by the design of the MusicPlace (https://musicplace.com/), which is a website whose purpose is to advertise children music courses. The design is bright, colorful, friendly and especially captivating to children, so we decided to challenge ourselves with this design. This website mainly acts as a marketing media that is within accessible reach to everyone. It contains information about various courses, snippets of their classes and user reviews. It also shows guidance on how to register their courses and provide customer service. Our website differs from theirs in one important aspect: ours serves as both a marketing outlet and a commercial site, while theirs is for pure educational and informational purposes.

The stack we've chosen:

- Vanilla Javascript for the frontend, with the aid of jquery.
- TailwindCSS for ease of building user-friendly and responsive websites.
- Pure PHP backend.
- MySQL as the database management system.

Chapter II: Technology

2.1 jQuery and Pure JavaScript

2.1.1 Advantages

- Simplifies DOM manipulation and AJAX requests with concise syntax
- Broad browser compatibility and extensive plugin ecosystem
- Combined with pure JavaScript for optimal performance where needed
- Lightweight compared to full frameworks

2.1.2 Disadvantages

- Somewhat outdated compared to modern frameworks (React, Vue)
- Can lead to spaghetti code in larger applications if not properly structured
- Performance overhead for simple DOM operations compared to vanilla JS

2.2 Tailwind CSS

2.2.1 Advantages

- Utility-first approach enables rapid UI development
- Highly customizable through configuration
- Reduces CSS file size in production with PurgeCSS

- Consistent design system through predefined values

2.2.2 Disadvantages

- Steep learning curve for developers used to traditional CSS
- HTML can become verbose with multiple utility classes
- Requires build process for optimal production performance

2.3 MySQL

2.3.1 Advantages

- Reliable and well-established relational database
- Excellent documentation and community support
- Good performance for structured data
- Strong data integrity through ACID compliance

2.3.2 Disadvantages

- Scaling horizontally can be challenging
- Less flexible than NoSQL databases for rapidly changing data structures
- Performance can degrade with very large datasets without proper optimization

2.4 PHP

2.4.1 Advantages

- Specifically designed for web development
- Easy to deploy and widely supported by hosting providers
- Large ecosystem of libraries and frameworks
- Simple learning curve for beginners

2.4.2 Disadvantages

- Inconsistent function naming conventions
- Performance limitations compared to compiled languages
- Type safety issues in older versions (improved in PHP 7+)

2.5 AJAX

2.5.1 Advantages

- Enables asynchronous data loading without page refreshes
- Improves user experience with dynamic content updates
- Reduces server load by fetching only required data
- Seamless integration with jQuery

2.5.2 Disadvantages

- Can complicate application state management
- Requires careful error handling
- Potential accessibility issues if not implemented properly

2.6 Security vulnerabilities and mitigations

2.6.1 SQL Injection

2.6.1.1 Vulnerability

Raw SQL queries with unsanitized user input allow attackers to manipulate database queries.

2.6.1.2 Mitigation

- Implemented prepared statements with parameter binding
- Used PDO with parameterized queries
- Applied input validation and sanitization
- Limited database user privileges

2.6.2 Cross-Site Scripting (XSS)

2.6.2.1 Vulnerability

Unsanitized user input rendered as HTML/JavaScript allows attackers to inject malicious scripts.

2.6.2.2 Mitigation

- Applied context-appropriate output encoding (htmlspecialchars())
- Implemented Content Security Policy (CSP)
- Used HTTPOnly cookies to prevent JavaScript access
- Validated and sanitized all user inputs

2.6.3 CSRF (Cross-Site Request Forgery)

2.6.3.1 Vulnerability

Attackers can trick users into performing unwanted actions on authenticated sessions.

2.6.3.2 Mitigation

- Implemented unique CSRF tokens for forms
- Validated token and origin on form submissions
- Added SameSite cookie attributes
- Required confirmation for sensitive operations

2.6.4 Session Hijacking

2.6.4.1 Vulnerability

Attackers can steal or manipulate session identifiers to impersonate legitimate users.

2.6.4.2 Mitigation

- Implemented secure session handling with session regeneration
- Used HTTPS throughout the application
- Applied proper session timeout controls
- Implemented IP-based session validation for critical operations

2.7 SEO Optimization

Search Engine Optimization (SEO) is the practice of improving a website's visibility and ranking in search engine results pages (SERPs). When implemented effectively, SEO helps increase organic (non-paid) traffic to your website by making it more attractive to search engines like Google, Bing, and Yahoo.

Utilized strategies:

- Semantic HTML structure with appropriate heading hierarchy
- Server-side rendering for faster initial page loads
- Mobile-responsive design using Tailwind's responsive utilities
- Optimized meta tags (`title`, `description`, Open Graph)
- URL structure optimization with clean, descriptive URLs
- Image optimization with appropriate `alt` tags and lazy loading

Chapter III: Design

3.1 MVC Architecture

The Model-View-Controller (MVC) architectural pattern represents a fundamental approach to application design that emphasizes separation of concerns. This separation creates a modular codebase where components have distinct responsibilities, facilitating maintenance, testing, and scalability. Our PHP application embraces these core principles while implementing them in a pragmatic manner suited to web development requirements.

3.1.1 Conceptual overview

The MVC implementation divides the application into three primary component types:

Model Layer: Responsible for data retrieval, storage, validation, and business rule enforcement. This layer encapsulates all data operations and presents a clean interface to controllers, abstracting database operations and ensuring data integrity.

View Layer: Handles the presentation logic, transforming data from models into user interface elements. Our implementation uses PHP-based templates with layout composition, allowing for consistent UI elements across pages.

Controller Layer: Orchestrates the application flow by receiving HTTP requests, interacting with appropriate models, and selecting views for rendering. Controllers serve as the intermediary that connects user interaction with system functionality.

3.1.2 Directory Structure

The physical organization of the codebase reflects the conceptual divisions of MVC, with additional supporting directories for configuration, middleware, and deployment resources:

```
├─── api/              = API endpoints for AJAX interactions
├─── config/           = Configuration files and environment settings
├─── controllers/      = Controller classes organized by domain
├─── middleware/       = Request preprocessing components
├─── migrations/       = Database schema version files
├─── models/           = Data and business logic components
├─── public/           = Publicly accessible assets
├─── views/            = Templates and presentation logic
│    ├─── layouts/     = Layout templates for page composition
│    └─── [feature dirs]/  = Feature-specific view templates
├─── .htaccess         = Web server configuration
├─── index.php         = Application entry point
└─── routes.php        = URL routing definitions
```

This structure balances conceptual purity with practical organization, grouping files by both responsibility and feature context.

3.2 Application Flow

The system processes user interactions through carefully defined execution paths. These flows can be categorized into two primary patterns with several distinct phases:

3.2.1 Standard Web Request Flow

This flow represents the traditional request/response cycle where the server generates complete HTML pages:

1. **Request Initiation**:
   - User requests a URL through their browser
   - Web server receives the request and routes it to the application entry point
   - .htaccess settings ensure all requests are directed to index.php
2. **Request Preprocessing**:
   - Session is initialized (session_start())
   - Configuration files are loaded (config/index.php)

- Routes are registered (routes.php)
- View utilities are made available (views/index.php)
- Middleware is prepared (middleware/UserMiddleware.php)

3. **Route Resolution**:
   - Application extracts the request path and method
   - System checks if the path exists in the registered routes
   - For undefined routes, the system renders a 404 page
4. **Middleware Processing**:
   - UserMiddleware validates session state and authentication
   - Middleware may redirect unauthenticated users to login page
   - Request context is established (e.g., current user)
5. **Controller Delegation**:
   - Appropriate controller is instantiated based on the route
   - Controller's route() method receives HTTP method and path
   - Controller determines which action method to execute
6. **Model Interaction**:
   - Controller instantiates required model objects
   - Models connect to the database through the Database singleton
   - Models execute queries and retrieve data
   - Domain objects are created from database records
   - Business rules are applied to the retrieved data
7. **View Preparation**:
   - Controller organizes model data into view-friendly structures
   - Data array is passed to the view rendering function
8. **View Rendering**:
   - Output buffering is initiated to capture rendered content
   - View template is included and executes, generating HTML
   - View may access passed data to populate dynamic elements
   - Layout template is loaded with the buffered content
   - Final HTML is sent to the browser
9. **Response Completion**:
   - Buffer is flushed to the client
   - Connection is closed
   - Browser renders the received HTML

3.2.2 AJAX Interaction Flow

This pattern supports dynamic interactions without requiring full page reloads:

1. **Client-side Initiation**:
   - JavaScript event triggers on user interaction

- AJAX request is constructed with appropriate headers
- Request is directed to an API endpoint with payload data

2. **Server Preprocessing**:
   - Request arrives at the application entry point
   - System detects API path prefix (/api/)
   - Direct file inclusion occurs instead of controller routing
3. **API Endpoint Processing**:
   - Specific API PHP script is loaded (e.g., search-photos.php)
   - Authentication and authorization are verified
   - Request parameters are validated
4. **Model Interaction**:
   - API script interacts with appropriate models
   - Database operations are performed
   - Results are processed according to business rules
5. **Response Formatting**:
   - Data is structured as JSON or required format
   - Headers are set for content type and caching
   - Response is encoded and sent to client
6. **Client-side Processing**:
   - JavaScript receives and parses the response
   - DOM is updated based on received data
   - User interface reflects the changes without page reload

This dual-flow architecture provides flexibility to handle both traditional page requests and dynamic interactions while maintaining consistent structural organization and separation of concerns.

Chapter IV: Implementation

4.1 Model Implementation

The Model layer consists of two primary component types:

1. **Domain Objects**: Plain PHP classes that represent business entities with properties and minimal behavior.
2. **Model Services**: Classes that handle data operations for specific entity types.

Here is a high-level implementation example of the Model layer:

```
// Domain Object Example
class User {
  public int $id;
```

```
  public DateTime $dob;
  public string $firstName;
  public string $lastName;
  public string $email;
  public bool $isAdmin;

  public function __construct(int $id, DateTime $dob, string $firstName,
                              string $lastName, string $email, bool $isAdmin) {
    $this->id = $id;
    $this->dob = $dob;
    $this->firstName = $firstName;
    $this->lastName = $lastName;
    $this->email = $email;
    $this->isAdmin = $isAdmin;
  }

  public function getFullName(): string {
    return $this->firstName . ' ' . $this->lastName;
  }
}

// Model Service Example
class UserModel {
  private $db;

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

  public function fetchById(int $id): ?User {
    // Database interaction logic
    // Returns User object or null
  }

  public function fetchAll(): array {
    // Database interaction logic
    // Returns array of User objects
  }

  public function create(array $userData): ?int {
    // Validation and insertion logic
    // Returns new ID or null on failure
  }

  public function update(int $id, array $userData): bool {
    // Validation and update logic
    // Returns success status
  }

  public function delete(int $id): bool {
    // Deletion logic
    // Returns success status
```

```
  }
}
```

The implementation employs several technical approaches:

- **PDO for Database Access**: Provides secure parameterized queries and database abstraction
- **Transaction Management**: Ensures data integrity during operations
- **Type Declarations**: Leverages PHP's type system for code clarity and error prevention
- **Null Handling**: Returns null for failed operations allowing graceful error management
- **Feature-based Organization**: Models are grouped in directories by feature when appropriate

This approach creates a data layer that encapsulates database operations while providing clean interfaces to the rest of the system.

4.2 View Implementation

The View layer employs a template-based system with layout composition. Here's a high-level implementation:

```
// View rendering utilities
function renderContentInLayout(string $layout, string $content, array $data): void {
  // The $content variable is made available to the layout
  // The $data array is extracted to variables for the layout
  extract($data);
  include $layout;
}

function renderView(string $view, array $data): void {
  // Start output buffering to capture view content
  ob_start();
  // Extract data to variables for the view
  extract($data);
  // Include the view file, which now has access to extracted variables
  include $view;
  // Get buffered content
  $content = ob_get_clean();
  // Render content within the layout
  renderContentInLayout('views/layouts/default.php', $content, $data);
}

// Example layout file (views/layouts/default.php)
/*
<!DOCTYPE html>
<html>
<head>
```

```
  <title>Application</title>
  <link rel="stylesheet" href="/css/style.css">
</head>
<body>
  <header>
    <!-- Header content -->
  </header>

  <main>
    <?php echo $content; ?>
  </main>

  <footer>
    <!-- Footer content -->
  </footer>
</body>
</html>
*/

// Example view file (views/home/index.php)
/*
<div class="welcome">
  <h1><?php echo $introduction->title; ?></h1>
  <p><?php echo $introduction->content; ?></p>
</div>

<div class="quote">
  <blockquote><?php echo $quote->text; ?></blockquote>
  <cite><?php echo $quote->author; ?></cite>
</div>

<div class="newsletter">
  <?php foreach ($newsLetters as $letter): ?>
    <article>
      <h2><?php echo $letter->title; ?></h2>
      <p><?php echo $letter->excerpt; ?></p>
      <a href="/newsletter/<?php echo $letter->id; ?>">Read more</a>
    </article>
  <?php endforeach; ?>
</div>
*/
```

Key technical features include:

- **Output Buffering**: Captures rendered content for inclusion in layouts
- **Layout Templates**: Provides consistent page structure across the application
- **Context-specific Rendering**: Different rendering functions for various user contexts
- **Data Passing**: Controllers supply data arrays to views for template variable rendering

This implementation balances simplicity with the flexibility needed for a multi-faceted user interface.

4.3 Controller Implementation

Controllers serve as the coordinators between HTTP requests, business logic, and presentation. Here's a high-level implementation:

```
// Base Controller (optional)
abstract class Controller {
  protected function requireAuthentication(): void {
    if (!isset($_SESSION['user_id'])) {
      header('Location: /login');
      exit;
    }
  }

  protected function requireAdmin(): void {
    if (!isset($_SESSION['user_id']) || !$_SESSION['is_admin']) {
      header('Location: /login');
      exit;
    }
  }
}

// Feature-specific Controller
class HomeController extends Controller {
  public function route(string $method, string $path): void {
    if ('/' === $path && 'GET' === $method) {
      $this->index();
    } else {
      // Handle invalid method/path combinations
      header('HTTP/1.1 405 Method Not Allowed');
      exit;
    }
  }

  public function index(): void {
    // Instantiate models
    $newsLetterModel = new NewsLetterModel();
    $introductionModel = new IntroductionModel();
    $quoteModel = new QuoteModel();

    // Fetch data from models
    $newsLetters = $newsLetterModel->fetchAll();
    $introduction = $introductionModel->fetch();
    $quotes = $quoteModel->fetchAll();

    // Pass data to view
    renderView('views/home/index.php', [
      'newsLetters' => $newsLetters,
      'introduction' => $introduction,
```

```
      'quotes' => $quotes
    ]);
  }
}

// Admin-specific Controller
class AdminController extends Controller {
  public function route(string $method, string $path): void {
    // First ensure admin privileges for all routes
    $this->requireAdmin();

    // Route to appropriate method
    if ('/admin/' === $path && 'GET' === $method) {
      $this->dashboard();
    } else if ('/admin/contacts/' === $path && 'GET' === $method) {
      $this->viewContacts();
    } else if ('/admin/contacts/' === $path && 'POST' === $method) {
      $this->updateContact();
    } else {
      // Handle invalid path/method
      header('HTTP/1.1 404 Not Found');
      exit;
    }
  }

  private function dashboard(): void {
    // Dashboard implementation
  }

  private function viewContacts(): void {
    // Contact list implementation
  }

  private function updateContact(): void {
    // Contact update implementation
  }
}
```

The implementation approach includes:

- **Method-based Routing**: Controllers determine which method to call based on HTTP method and path
- **Model Coordination**: Controllers instantiate and utilize multiple models as needed
- **Data Preparation**: Controllers gather and organize data before passing to views
- **HTTP Method Validation**: Controllers enforce appropriate HTTP methods for actions
- **Authorization Logic**: Controllers may include access control checks for protected routes

This pattern creates a clean coordination layer that keeps business logic in models and presentation logic in views.

4.4 Router Implementation

The Router directs incoming requests to appropriate controllers. Here's a high-level implementation:

```
// routes.php - Route definitions
$routes = [
  '/' => new HomeController(),
  '/login/' => new LoginController(),
  '/logout/' => new LoginController(),
  '/signup/' => new LoginController(),
  '/contact/' => new ContactController(),
  '/admin/' => new AdminController(),
  '/admin/home-page/' => new AdminController(),
  '/admin/contacts/' => new AdminController(),
  '/account/' => new AccountController(),
  '/shop/' => new ShopController(),
];

// index.php - Application entry point
<?php
session_start();

// Load dependencies
require_once 'config/index.php';
require_once 'routes.php';
require_once 'views/index.php';
require_once 'middleware/UserMiddleware.php';

// Extract request information
$path = $_SERVER['PATH_INFO'] ?? '/';
$method = $_SERVER['REQUEST_METHOD'];

// Route the request
if (str_starts_with($path, "/api/")) {
  // Handle API requests
  $apiFile = trim($path, '/') . '.php';
  if (file_exists($apiFile)) {
    require_once($apiFile);
  } else {
    header('HTTP/1.1 404 Not Found');
    echo json_encode(['error' => 'API endpoint not found']);
  }
} else if (array_key_exists($path, $routes)) {
  // Handle controller-based routes
  $controller = $routes[$path];
  $controller->route($method, $path);
} else {
  // Handle 404 for undefined routes
```

```
  header('HTTP/1.1 404 Not Found');
  renderView('views/404.php', []);
}
```

The routing implementation uses:

- **Path-based Mapping**: Routes are defined as URL paths mapped to controller instances
- **Array Structure**: Simple associative array provides readable route definitions
- **Controller Instance Reuse**: Multiple paths can map to the same controller instance
- **API Detection**: Special handling for API endpoints
- **Error Handling**: Proper HTTP status codes for undefined routes

This approach provides flexibility while maintaining simplicity and understandability.

Chapter V: Installation

References