Implementation

This chapter details our application and source code structure, highlighting how models, views, controllers, apis, routing are handled at the source code level.

High-level implementation

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.

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.

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.

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
```

```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.