Project Documentation: GenAI Web Platform

Part I: Getting Started

1. Introduction 1.1. What is GenAI Web Platform? 1.2. Core Features & Capabilities 1.3. Who Is This For? 1.4. Technology Stack

2. Quick Start Guide 2.1. Your First 5 Minutes 2.2. Running the Application Locally 2.3. Key Concepts at a Glance

3. Installation 3.1. Server Requirements & Prerequisites 3.2. Automated Installation (Recommended) 3.3. Manual Installation (Advanced) 3.4. Environment Configuration (`.env`) 3.5. Post-Installation Steps & Security

---

Part II: Guides & Tutorials

4. Core Concepts 4.1. Architectural Overview (MVC-S) 4.2. The Request Lifecycle 4.3. Service Container & Dependency Injection 4.4. Directory Structure Explained 4.5. Security Principles

5. Tutorial: Building Your First Feature 5.1. Creating a New Route 5.2. Building the Controller & Service 5.3. Interacting with the Database (Model & Entity) 5.4. Displaying Data in a View

6. Feature Guides (Deep Dives) 6.1. User Authentication 6.1.1. Registration & Login Flow 6.1.2. Email Verification & Password Resets 6.1.3. Access Control with Filters 6.2. Payment Gateway Integration 6.2.1. Configuration 6.2.2. Initiating a Transaction 6.2.3. Verifying a Payment 6.3. AI Service Integration 6.3.1. Generating Content 6.3.2. Conversational Memory System 6.3.3. Handling Multimedia Inputs 6.4. Cryptocurrency Data Service 6.4.1. Querying Balances 6.4.2. Fetching Transaction Histories 6.5. Administrative Dashboard 6.5.1. User Management 6.5.2. Sending Email Campaigns

---

Part III: Technical Reference

7. API Reference 7.1. Authentication 7.2. Endpoints 7.2.1. `GET /resource` 7.2.2. `POST /resource` 7.2.3. `PUT /resource/{id}` 7.2.4. `DELETE /resource/{id}` 7.3. Rate Limiting 7.4. Error Codes & Responses

8. Command-Line Interface (CLI) 8.1. Overview of Custom Commands 8.2. `php spark train` 8.3. `php spark [another:command]`

9. Configuration Reference 9.1. Application (`App.php`) 9.2. Database (`Database.php`) 9.3. Custom Configurations (`AGI.php`, etc.)

10. Testing 10.1. Running the Test Suite 10.2. Writing Unit Tests 10.3. Writing Feature Tests

---

Part IV: Operations & Community

11. Deployment 11.1. Production Server Setup 11.2. Deployment Checklist 11.3. Performance Optimization

---

Part IV: Operations & Community

Part I: Getting Started

1. Introduction

1.1. What is GenAI Web Platform?

The GenAI Web Platform is a comprehensive, multi-functional application built on the CodeIgniter 4 framework. It serves as a portal for registered users to access a suite of powerful digital services, including AI-driven content generation and analysis, real-time cryptocurrency data queries, and robust user and content management capabilities. Designed with a modular architecture, it features a secure user authentication system, an account dashboard with an integrated balance and payment system (supporting M-Pesa, Airtel, and Card), and a complete administrative panel for user oversight.

1.2. Core Features & Capabilities

- **User Authentication:** Secure registration, login, email verification, and password reset functionality.
- **Payment Gateway Integration:** Seamless payments via Paystack, a popular African payment gateway.
- **AI Service Integration:** Advanced text and multimedia interaction with Google's Gemini API, featuring a sophisticated conversational memory system.
- **Cryptocurrency Data Service:** Real-time balance and transaction history queries for Bitcoin (BTC) and Litecoin (LTC) addresses.
- **Administrative Dashboard:** Robust tools for user management, balance adjustments, financial oversight, and sending email campaigns to all users.
- **Secure & Performant:** Built with modern security best practices and optimized for production environments.

1.3. Who Is This For?

This platform is designed for developers, creators, and businesses, particularly in Kenya and the broader African market, who require a flexible, pay-as-you-go solution for accessing advanced AI and blockchain data services. It serves as both a functional application and a robust foundation for building more complex systems.

1.4. Technology Stack

- **Backend:** PHP 8.1+, CodeIgniter 4
- **Frontend:** Bootstrap 5, JavaScript, HTML5, CSS3
- **Database:** MySQL
- **Web Server:** Apache2
- **Key Libraries:**
    - `google/gemini-php`: For interacting with the Gemini API.
    - `dompdf/dompdf`: For PDF generation.
    - `nlp-tools/nlp-tools`: For Natural Language Processing tasks.
    - `php-ffmpeg/php-ffmpeg`: For audio and video processing.
- **System Dependencies:** Pandoc, ffmpeg
- **Development & Deployment:** Composer, PHPUnit, Spark CLI, Git, Bash

2. Quick Start Guide

2.1. Your First 5 Minutes

For a fresh Ubuntu server, the fastest way to get started is with the automated setup script.

1. Clone the repository: `git clone https://github.com/nehemiaobati/genaiwebapplication.git`
2. Navigate into the directory: `cd genaiwebapplication`
3. Make the script executable: `chmod +x setup.sh`
4. Run with sudo: `sudo ./setup.sh`
5. After completion, edit the newly created `.env` file to add your API keys.

2.2. Running the Application Locally

1. **Clone the Repository:** `git clone https://github.com/nehemiaobati/genaiwebapplication.git`
2. **Install Dependencies:** `composer install`
3. **Create Environment File:** Copy env to `.env` and configure your local database and `app.baseURL`.
4. **Run Migrations:** `php spark migrate`
5. **Start the Server:** `php spark serve`
6. Access the application at `http://localhost:8080`.

2.3. Key Concepts at a Glance

- **MVC-S Architecture:** The application separates concerns into Models (database), Views (presentation), Controllers (request handling), and Services (business logic).

- **Services:** Core functionality like payment processing (`PaystackService`), AI interaction (`GeminiService`), and crypto queries (`CryptoService`) are encapsulated in their own service classes for reusability.
- **Pay-As-You-Go:** Users top up an account balance, and this balance is debited for each AI or Crypto query they perform.

3. Installation

3.1. Server Requirements & Prerequisites

- **OS:** Ubuntu (Recommended)
- **Web Server:** Apache2 or Nginx
- **PHP:** Version 8.1 or higher with `intl`, `mbstring`, `bcmath`, `curl`, `xml`, `zip`, `gd` extensions.
- **Database:** MySQL Server
- **Tools:** Composer, Git, Pandoc, ffmpeg

3.2. Automated Installation (Recommended)

The `setup.sh` script is designed for a clean Ubuntu server and automates the entire installation process. It will:

- Install Apache2, PHP 8.2, and MySQL.
- Create a dedicated database and user.
- Install Composer and Node.js.
- Clone the project repository.
- Install all project dependencies.
- Create the `.env` file with generated database credentials.
- Run database migrations.
- Configure an Apache virtual host.

Usage:

```
chmod +x setup.sh
sudo ./setup.sh
```

3.3. Manual Installation (Advanced)

1. **Clone Repository:** `git clone https://github.com/nehemiaobati/genaiwebapplication.git .`
2. **Install Dependencies:** Run `composer install`.
3. **Configure Environment:** Copy `env` to `.env`.
4. **Database Setup:** Create a MySQL database and user.
5. **Edit `.env` file:** Fill in your `app.baseURL`, database credentials, API keys, and email settings.
6. **Run Migrations:** Run `php spark migrate` to create all necessary tables.
7. **Set Permissions:** Ensure the `writable/` directory is writable by the web server: `chmod -R 775 writable/`.
8. **Configure Web Server:** Point your web server's document root to the project's `public/` directory.

3.4. Environment Configuration (`.env`)

The `.env` file is critical for configuring the application. You must fill in the following values:

- `CI_ENVIRONMENT`: `development` for local, `production` for live.
- `app.baseURL`: The full URL of your application (e.g., `http://yourdomain.com/`).
- `database.default.*`: Your database connection details.
- `encryption.key`: A unique, 32-character random string for encryption.
- `PAYSTACK_SECRET_KEY`: Your secret key from your Paystack dashboard.
- `GEMINI_API_KEY`: Your API key for the Google Gemini service.
- `recaptcha_siteKey` & `recaptcha_secretKey`: Your keys for Google reCAPTCHA v2.
- `email.*`: Configuration details for your SMTP email sending service.

3.5. Post-Installation Steps & Security

1. **Secure `.env`:** Ensure the `.env` file is never committed to version control.
2. **Set DNS:** Point your domain's A record to the server's IP address.
3. **Enable HTTPS:** For production, install an SSL certificate. Using Certbot is recommended:

   ```
   sudo apt install certbot python3-certbot-apache
   sudo certbot --apache
   ```

Part II: Guides & Tutorials

4. Core Concepts

4.1. Architectural Overview (MVC-S)

The project extends the traditional Model-View-Controller (MVC) pattern with a **Service layer (MVC-S)** to better organize business logic.

- **Models (`app/Models`)**: Handle all direct database interactions. They are responsible for querying, inserting, and updating data.
- **Views (`app/Views`)**: Contain the presentation logic (HTML). They receive data from controllers and render it for the user.
- **Controllers (`app/Controllers`)**: Act as the bridge between Models and Views. They handle incoming HTTP requests, orchestrate calls to services, and pass data to the appropriate view.
- **Services (`app/Libraries`)**: Contain the core business logic. This includes interacting with third-party APIs (Paystack, Gemini), processing complex data, and performing calculations. This keeps controllers lean and focused on handling the request-response cycle.

4.2. The Request Lifecycle

1. The request first hits `public/index.php`.
2. CodeIgniter's routing (`app/Config/Routes.php`) matches the URL to a specific controller method.
3. Any defined Filters (`app/Config/Filters.php`) are executed before the controller is called.
4. The Controller method is executed. It may validate input, call one or more Services, and retrieve data from Models.
5. The Controller passes the prepared data to a View.
6. The View is rendered into HTML and sent back to the browser as the final response.

4.3. Service Container & Dependency Injection

The application uses CodeIgniter's service container to manage class instances. Core services are defined in `app/Config/Services.php`. This allows for easy instantiation and sharing of service objects throughout the application.

- **Registration:** Custom services like `PaystackService` and `GeminiService` are registered as static methods in `app/Config/Services.php`.
- **Usage:** Services are accessed anywhere in the application using the `service()` helper function (e.g., `$geminiService = service('geminiService');`).

4.4. Directory Structure Explained

- `app/Commands`: Houses custom `spark` CLI commands, like the `train` command.
- `app/Config`: Contains all application configuration files, including `Routes.php` and `Services.php`.
- `app/Controllers`: Handles web requests.
- `app/Database`: Contains database migrations and seeders for schema management.
- `app/Entities`: Object-oriented representations of database table rows.
- `app/Filters`: Middleware for route protection (e.g., authentication).
- `app/Libraries`: This directory is used for the **Service layer**, containing all core business logic.
- `app/Models`: Handles database interactions.
- `app/Views`: Contains all HTML templates for the user interface.
- `public/`: The web server's document root, containing the main `index.php` file and public assets.
- `writable/`: Directory for logs, cache, and file uploads. Must be server-writable.

4.5. Security Principles

The application adheres to security best practices to protect against common vulnerabilities.

- **Public Webroot:** The server's document root is set to the `public/` directory, preventing direct web access to application source code.
- **CSRF Protection:** Cross-Site Request Forgery tokens are used on all POST forms to prevent malicious submissions.
- **XSS Filtering:** All data rendered in views is escaped using `esc()` to prevent Cross-Site Scripting attacks.
- **Environment Variables:** Sensitive information like API keys and database credentials are stored in the `.env` file, which is never committed to version control.

- **Query Builder & Entities:** All database queries use CodeIgniter's built-in methods, which automatically escape parameters to prevent SQL injection.

5. Tutorial: Building Your First Feature

This tutorial demonstrates how to build a simple "Notes" feature following the MVC-S pattern.

5.1. Creating a New Route

Open `app/Config/Routes.php` and add routes for viewing and creating notes.

```
$routes->group('notes', ['filter' => 'auth'], static function ($routes) {
    $routes->get('/', 'NoteController::index', ['as' => 'notes.index']);
    $routes->post('create', 'NoteController::create', ['as' => 'notes.create']);
});
```

5.2. Building the Controller & Service

1. **Create the Controller** using the command line: `php spark make:controller NoteController`
2. Edit `app/Controllers/NoteController.php`:

```
<?php
namespace App\Controllers;

class NoteController extends BaseController
{
    public function index()
    {
        // For simplicity, we call the model directly here.
        // In a complex app, this would go through a NoteService.
        $noteModel = new \App\Models\NoteModel();
        $data['notes'] = $noteModel->where('user_id', session()->get('userId'))->findAll();
        return view('notes/index', $data);
    }

    public function create()
    {
        $noteModel = new \App\Models\NoteModel();
        $noteModel->save([
            'user_id' => session()->get('userId'),
            'content' => $this->request->getPost('content')
        ]);
```

```
        return redirect()->to(url_to('notes.index'))->with('success', 'Note saved!');
    }
}
```

5.3. Interacting with the Database (Model & Entity)

1. **Create a Migration:** `php spark make:migration create_notes_table`
2. Edit the new migration file in `app/Database/Migrations/` to define the table schema.
3. **Create the Model:** `php spark make:model NoteModel`
4. **Create the Entity:** `php spark make:entity Note`

5.4. Displaying Data in a View

Create a new file at `app/Views/notes/index.php`.

```
<?= $this->extend('layouts/default') ?>
<?= $this->section('content') ?>
<div class="container my-5">
    <h1>My Notes</h1>
    <!-- Form to create a new note -->
    <form action="<?= url_to('notes.create') ?>" method="post">
        <?= csrf_field() ?>
        <textarea name="content" class="form-control"></textarea>
        <button type="submit" class="btn btn-primary mt-2">Save Note</button>
    </form>

    <!-- List existing notes -->
    <ul class="list-group mt-4">
        <?php foreach($notes as $note): ?>
            <li class="list-group-item"><?= esc($note->content) ?></li>
        <?php endforeach; ?>
    </ul>
</div>
<?= $this->endSection() ?>
```

6. Feature Guides (Deep Dives)

6.1. User Authentication

- **6.1.1. Registration & Login Flow**: Managed by `AuthController.php`, this feature handles user registration with validation and reCAPTCHA, credential verification for login, and session management.

- **6.1.2. Email Verification & Password Resets**: Upon registration, a unique token is generated and emailed to the user. `AuthController::verifyEmail()` handles this token. The password reset flow also uses a secure, expiring token sent via email.
- **6.1.3. Access Control with Filters**: The `AuthFilter` (`app/Filters/AuthFilter.php`) is applied to routes in `app/Config/Routes.php` to protect pages that require a user to be logged in.

6.2. Payment Gateway Integration

- **6.2.1. Configuration**: The Paystack secret key is configured in the `.env` file (`PAYSTACK_SECRET_KEY`).
- **6.2.2. Initiating a Transaction**: `PaymentsController::initiate()` collects the amount and email, creates a local record in the `payments` table with a `pending` status, and calls `PaystackService::initializeTransaction()`. This service sends the request to Paystack, which returns a unique authorization URL. The user is then redirected to this URL to complete the payment.
- **6.2.3. Verifying a Payment**: After payment, Paystack redirects the user to the `callback_url` (`payment/verify`). `PaymentsController::verify()` retrieves the transaction reference and uses `PaystackService::verifyTransaction()` to confirm the payment status with Paystack. If successful, the local payment record is updated to `success` and the user's balance is updated within a database transaction.

6.3. AI Service Integration

- **6.3.1. Generating Content**: `GeminiController::generate()` is the core method. It prepares the user's prompt, adds contextual data from the memory system, and sends it to `GeminiService::generateContent()`. The service makes the API call to Google Gemini and returns the response.
- **6.3.2. Conversational Memory System**: This advanced feature is managed by `MemoryService.php`. When a user submits a prompt, the service:
    1. Generates a vector embedding of the user's input using `EmbeddingService`.
    2. Performs a hybrid search (semantic vector search + keyword search) on past interactions stored in the database.
    3. Constructs a context block from the most relevant past interactions.
    4. Prepends this context to the user's new prompt before sending it to the AI.
    5. After receiving a response, it stores the new question-and-answer pair and updates the relevance scores of all memories.
- **6.3.3. Handling Multimedia Inputs**: Users can upload files via the AI Studio. `GeminiController::uploadMedia()` handles the file, and `GeminiController::generate()` processes it, converting the file to base64 and including it in the API request to Gemini, which is a multimodal model.

6.4. Cryptocurrency Data Service

- **6.4.1. Querying Balances**: `CryptoController::query()` calls `CryptoService::getBtcBalance()` or `getLtcBalance()`. The service makes an API call to a third-party blockchain explorer (e.g., blockchain.info, blockchair.com) and formats the response.

- **6.4.2. Fetching Transaction Histories**: Similarly, the controller calls `CryptoService::getBtcTransactions()` or `getLtcTransactions()`. The service fetches the transaction data and formats it into a readable structure for the view.

6.5. Administrative Dashboard

- **6.5.1. User Management**: `AdminController.php` provides methods to list, search, view details, update balances, and delete users. All actions are protected to ensure only administrators can perform them. Balance updates are handled within a database transaction for data integrity.
- **6.5.2. Sending Email Campaigns**: `CampaignController.php` allows an administrator to compose an email that is then sent to every registered user in the `users` table. The process iterates through users and sends individual emails.

Part III: Technical Reference

7. API Reference

While this project is primarily a web application, its controllers can be adapted to serve a RESTful API. The following is a conceptual reference.

7.1. Authentication

Current authentication is session-based. For a stateless API, this would be replaced with a token-based system (e.g., JWT).

7.2. Endpoints

Endpoint definitions would follow standard REST conventions.

- **`GET /resource`**: List all items of a resource.
  - **Example:** `GET /api/users` - Would be handled by `AdminController::index()` to return a JSON list of users.
- **`POST /resource`**: Create a new resource item.
  - **Example:** `POST /api/users` - Would be handled by `AuthController::store()` to create a new user.
- **`PUT /resource/{id}`**: Update a specific resource item.
  - **Example:** `PUT /api/users/{id}/balance` - Would be handled by `AdminController::updateBalance()` to modify a user's balance.
- **`DELETE /resource/{id}`**: Delete a specific resource item.
  - **Example:** `DELETE /api/users/{id}` - Would be handled by `AdminController::delete()`.

7.3. Rate Limiting

CodeIgniter's Throttler filter can be applied to API routes in `app/Config/Routes.php` to prevent abuse by limiting the number of requests a user can make in a given time.

7.4. Error Codes & Responses

The API would use standard HTTP status codes to indicate the outcome of a request.

- `200 OK`: Request was successful.
- `201 Created`: The resource was successfully created.
- `400 Bad Request`: The request was malformed (e.g., validation error).
- `401 Unauthorized`: Authentication failed or is required.
- `403 Forbidden`: The authenticated user does not have permission.
- `404 Not Found`: The requested resource does not exist.
- `500 Internal Server Error`: A server-side error occurred.

8. Command-Line Interface (CLI)

CodeIgniter's CLI tool, `spark`, is used for various development and maintenance tasks.

8.1. Overview of Custom Commands

You can list all available commands by running `php spark`. Custom application commands are located in `app/Commands/`.

8.2. `php spark train`

This is a custom command defined in `app/Commands/Train.php`.

- **Purpose:** To run the AI text classification training service offline.
- **Action:** It invokes `TrainingService::train()`, which reads a training dataset, processes it, trains a Naive Bayes classifier, and saves the serialized model files to the `writable/nlp/` directory. This ensures that performance-intensive model training does not impact the live web application.

8.3. `php spark [another:command]`

Other essential built-in commands include:

- `php spark migrate`: Applies pending database migrations.
- `php spark db:seed <SeederName>`: Runs a database seeder to populate tables with data.
- `php spark make:controller <Name>`: Generates a new controller file.
- `php spark optimize`: Caches configuration and file locations for improved performance.

9. Configuration Reference

9.1. Application (`App.php`)

Located at `app/Config/App.php`, this file contains the base configuration for the application, including the `baseURL`, `indexPage`, and `appTimezone`.

9.2. Database (`Database.php`)

Located at `app/Config/Database.php`, this file defines the connection parameters for your databases. The `default` group is used for the main application, while the `tests` group is used for PHPUnit testing.

9.3. Custom Configurations (`AGI.php`, etc.)

Custom configuration files are placed in `app/Config/Custom/`.

- `AGI.php`: Contains all settings related to the AI service, including embedding models, hybrid search parameters, and memory logic (e.g., decay scores, context budget).
- `Recaptcha.php`: Stores the site and secret keys for the Google reCAPTCHA service, which are loaded from the `.env` file.

10. Testing

10.1. Running the Test Suite

The project is configured to use PHPUnit for testing. The test suite can be run using a Composer script.

```
composer test
```

This command executes `phpunit` as defined in `composer.json` and uses the configuration from `phpunit.xml.dist`.

10.2. Writing Unit Tests

Unit tests focus on testing individual classes (like a Service or Model) in isolation. Test files should be placed in the `tests/` directory, mirroring the `app/` structure.

10.3. Writing Feature Tests

Feature tests are designed to test a full request-response cycle, simulating a user interacting with the application. They allow you to test controllers, views, and redirects together.

Part IV: Operations & Community

11. Deployment

11.1. Production Server Setup

The `setup.sh` script provides a complete, automated setup for a production-ready Ubuntu server, including installing the web server, PHP, database, and all project dependencies, as well as configuring a virtual host.

11.2. Deployment Checklist

1. Set `CI_ENVIRONMENT` to `production` in your `.env` file.
2. Install production-only dependencies: `composer install --no-dev --optimize-autoloader`.
3. Run database migrations: `php spark migrate`.

4. Optimize the application: `php spark optimize`.
5. Ensure the web server's document root points to the `/public/` directory.
6. Verify that `writable/` directory has the correct server permissions.

11.3. Performance Optimization

- **Caching:** The application can be configured to use various caching strategies. The `app/Config/Cache.php` file is set to use Redis as the primary handler with a file-based fallback.
- **Autoloader Optimization:** The `--optimize-autoloader` flag in the composer install command creates an optimized class map for faster class loading.
- **Spark Optimize Command:** `php spark optimize` caches configuration and speeds up the framework's file locator.
- **Database Queries:** Use pagination (`paginate()`) for lists and select only necessary columns to keep database interactions efficient.

12. Troubleshooting

12.1. Frequently Asked Questions (FAQ)

- **Why did my payment fail?** Ensure you have sufficient funds and that your payment provider (e.g., M-Pesa) is active. If the problem persists, contact support with your transaction reference.
- **Why can't I log in after registering?** You must click the verification link sent to your email address before you can log in.
- **Why is my AI query failing?** This could be due to insufficient balance or a temporary issue with the Gemini API. Check your balance and try again after a few moments.

12.2. Common Error Resolutions

- **"Whoops! We hit a snag."**: This is the generic production error message. Check the server logs at `writable/logs/` for the specific error details.
- **"File upload failed."**: This usually indicates a permissions issue. Ensure the `writable/uploads/` directory and its subdirectories are writable by the web server.
- **"Could not send email."**: Verify that your SMTP credentials in the `.env` file are correct and that your email provider is not blocking the connection.

12.3. Logging & Debugging

- **Log Location:** All application logs are stored in `writable/logs/`, with a new file created for each day.
- **Log Levels:** The logging sensitivity can be adjusted in `app/Config/Logger.php`. In a `development` environment, the threshold is low to capture all messages. In `production`, it is higher to only log errors and critical issues.

13. Contributing

13.1. Contribution Guidelines

1. Fork the repository.
2. Create a new feature branch (`git checkout -b feature/AmazingFeature`).
3. Commit your changes (`git commit -m 'Add some AmazingFeature'`).
4. Push to the branch (`git push origin feature/AmazingFeature`).
5. Open a Pull Request.

13.2. Code Style (PSR-12)

The project enforces the PSR-12 coding standard. All contributions must adhere to this standard.

13.3. Submitting a Pull Request

Before submitting a pull request, ensure that your code is well-documented, follows the project's architectural patterns (MVC-S), and that all existing tests pass.

14. Appendices

14.1. Glossary of Terms

- **MVC-S:** Model-View-Controller-Service, an architectural pattern that separates database logic (Model), presentation (View), request handling (Controller), and business logic (Service).
- **Service:** A class in the `app/Libraries` directory that contains reusable business logic.
- **Entity:** A class that represents a single row from a database table, allowing for object-oriented interaction with data.
- **Embedding:** A numerical vector representation of text, used for semantic understanding and similarity searches in the AI memory system.
- **Pay-as-you-go:** A pricing model where users pay only for the services they consume, rather than a fixed subscription fee.

14.2. Changelog & Release History

(This section would be maintained with a list of versions and the changes introduced in each release.)

- **v1.0.0 (Initial Release)**
    - Core features implemented: User Authentication, Paystack Payments, Gemini AI Integration, Crypto Data Service, Admin Dashboard.

Part V: Documentation Maintenance Guide

15. A Guide for the Project Owner

This section serves as the standard operating procedure (SOP) for maintaining this project's documentation. Its purpose is to ensure accuracy, consistency, and longevity, whether updates are performed by you or an AI

assistant.

15.1. The Philosophy of Living Documentation

Treat this documentation as a core part of the codebase. It should evolve in lockstep with every feature change, bug fix, or architectural adjustment. An undocumented change is an incomplete change. The goal is to ensure that a new developer, or you in six months, can understand the *what*, *how*, and *why* of the system just by reading this document.

15.2. Your Role vs. the AI's Role

- **The AI's Role (Efficiency & Accuracy):** The AI is the primary documentation writer. It excels at systematically analyzing code changes (`git diff`), identifying affected components, and generating accurate, detailed descriptions based on the established structure. It is responsible for the heavy lifting of drafting content.
- **Your Role (Clarity & Context):** Your role is that of an editor and strategist. You review the AI-generated content for clarity, human readability, and high-level context that the code alone cannot provide. You ensure the "why" behind a change is captured, not just the "what."

15.3. The Documentation Update Workflow

This workflow applies to any code changes committed to the main branch.

A. For Simple Changes (e.g., typos, clarifications, minor updates):

1. **Identify:** Locate the relevant section in the documentation file.
2. **Edit:** Make the necessary correction or addition directly.
3. **Commit:** Commit the change with a clear message, prefixed with `docs:`.
    - *Example:* `docs: Correct typo in Installation section 3.4`

B. For Complex Changes (e.g., new features, architectural modifications):

1. **Identify the Scope:** Use the procedure in section **15.4** to identify all changed files and map them to the corresponding sections of this documentation.
2. **Draft Updates (or Prompt the AI):** For each affected section, draft the new content. If using the AI, provide it with the list of changed files and instruct it to update the documentation accordingly.
    - *Example Prompt for AI:* "A new email campaign feature has been added. The following files were created or modified: `CampaignController.php`, `CampaignModel.php`, `create.php` view, and the routes were updated. Please update the documentation, including a new sub-section in the Feature Guides (6.5.2) and updating the directory structure."
3. **Update Table of Contents:** If new sections or sub-sections were added, update the table of contents at the beginning of the document.
4. **Update Changelog:** Follow the procedure in section **15.5** to add an entry to the Changelog and determine if the version number needs to be updated.

5. **Review and Commit:** Read through all changes from the perspective of someone unfamiliar with the update. Is it clear? Is anything missing? Once satisfied, commit the changes.

15.4. Procedure: How to Review the Codebase for Changes

The most efficient way to find what needs documenting is by analyzing the difference between your feature branch and the main branch using Git.

1. **Generate a File List:** From your feature branch, run the following command to get a list of all files that have been added (`A`), modified (`M`), or renamed (`R`):

   ```
   git diff main --name-status  | git diff main --name-status > changed_files.txt
   git diff main  | git diff main > code_changes.diff
   ```

2. **Map Files to Documentation Sections:** Use the output from the command above and this checklist to determine which parts of the documentation to review and update.

If This File/Directory Changed...	...Then Review and Update These Documentation Sections
`setup.sh` or `.env` (new variables)	**3. Installation** (Prerequisites, Automated/Manual Setup, Environment Config)
`app/Config/Routes.php`	**5. Tutorial**, **6. Feature Guides**, **7. API Reference** (for new endpoints/URLs)
`app/Controllers/*`	**6. Feature Guides** (logic for a specific feature), **7. API Reference** (endpoint behavior)
`app/Libraries/*` (Services)	**4. Core Concepts** (if a fundamental service changed), **6. Feature Guides** (detailed business logic)
`app/Models/*` or `app/Entities/*`	**5. Tutorial**, **6. Feature Guides** (how data is handled for a feature)
`app/Database/Migrations/*`	**5. Tutorial**, **6. Feature Guides** (mention new database tables/columns)
`app/Commands/*`	**8. Command-Line Interface (CLI)**
`app/Config/Custom/*`	**9. Configuration Reference** (document new custom settings)
`app/Views/*`	Usually doesn't require a doc change unless a major new UI feature is introduced.

If This File/Directory Changed...	...Then Review and Update These Documentation Sections
`composer.json` (new dependencies)	**1.4. Technology Stack**, **3.1. Server Requirements**

15.5. Procedure: Updating the Changelog and Managing Releases

This project follows **Semantic Versioning (SemVer)**: `MAJOR.MINOR.PATCH`.

- **`PATCH` (e.g., 1.0.0 -> 1.0.1):** For backward-compatible bug fixes or documentation corrections. No new features.
- **`MINOR` (e.g., 1.0.1 -> 1.1.0):** For new features or functionality added in a backward-compatible manner. This will be your most common version bump.
- **`MAJOR` (e.g., 1.1.0 -> 2.0.0):** For incompatible API changes or significant architectural shifts that break backward compatibility.

Changelog Update Criteria:

1. **Locate the Changelog:** Find section **14.2. Changelog & Release History**.

2. **Create a New Entry:** For every set of changes merged to `main`, add a new version heading. If it's the first change for a new version, create the heading; otherwise, add to the existing one.

3. **Format the Entry:** Use the following structure, inspired by Keep a Changelog. Only include the categories you need for that release.

```
**v[MAJOR.MINOR.PATCH] - YYYY-MM-DD**

### Added
- New feature A.
- New feature B.

### Changed
- Updated user dashboard for better UX.
- Switched payment provider logic.

### Fixed
- Resolved login bug affecting Safari users.

### Removed
```

```
- Deprecated the old reporting feature.
```

4. **Be Concise and User-Focused:** Describe the *impact* of the change, not just the code that was altered.
    - **Good:** "Added email notifications for successful payments."
    - **Bad:** "Modified the `PaymentsController` and created a `PaymentNotification` class."