***Abstract***

This project presents the development of ServiceHub MVM, a scalable and modular multi-vendor service marketplace designed to streamline service listings, bookings, and transactions between vendors and customers in a centralized ecosystem. The system allows vendors to register, authenticate, and manage their service offerings, while customers can seamlessly browse categorized services, make real-time bookings, and manage personal profiles and order histories. Additionally, an administrator dashboard facilitates system-wide oversight, including user management, performance analytics, and order regulation.

ServiceHub MVM is engineered to function independently or integrate with popular content management systems (CMS) such as WordPress, making it highly extensible and compatible with existing web infrastructures. The platform emphasizes modularity, maintainability, and role-based access control through the implementation of custom user roles and dedicated dashboards for Admin, Vendor, and Customer tiers.

Core features include a dynamic service booking system with real-time form validation and asynchronous submission, custom order generation and tracking mechanisms, role-specific dashboards for streamlined user experiences, advanced filtering and search capabilities based on taxonomies and price ranges, secure Stripe payment integration for transactional operations, administrative dashboard widgets to monitor platform metrics, and automated email notifications for booking confirmations and vendor alerts. The system also ensures robust security through role-based authentication and session validation. Built using modern development practices, ServiceHub MVM provides a robust foundation for launching a commercial-grade service marketplace connecting digital service providers with end-users.

The system architecture follows modern web development paradigms using a mix of backend and frontend technologies, ensuring scalability, extensibility, and performance. This project stands as a blueprint for building a commercial-grade online service marketplace that bridges the gap between digital service providers and consumers.

***Chapter 1: Introduction***

In today’s digital age, the way people find and book services has changed dramatically. From hiring electricians to booking photography sessions, users now expect fast, convenient, and secure ways to connect with professionals. Traditional service models often involve time-consuming phone calls or visits, while many local businesses struggle with limited online presence. To address this gap, online service marketplaces have become increasingly popular, acting as a bridge between customers and service providers.

This project presents a modern solution: a Multi-Vendor Service Marketplace that simplifies the entire process for all stakeholders. Service providers (vendors) can register, showcase their offerings, manage orders, and receive payments. At the same time, customers can explore services, apply filters to find relevant options, and book services with ease. The platform is user-friendly, scalable, and designed to support integration with widely-used content management systems, including WordPress.

**1.1 Background**

In today’s digital age, service-based businesses are increasingly moving online to reach more customers and streamline operations. While product-based eCommerce has seen massive growth and support from major platforms, service-based businesses—especially local and independent providers—have limited access to structured, scalable platforms. Many vendors still rely on traditional communication methods like calls or physical visits to get leads, schedule appointments, and handle payments. This results in inefficiencies, limited reach, and lack of automation.

To bridge this gap, a comprehensive digital marketplace that can bring together vendors from different industries under a unified system is needed. A Multi-Vendor Service Marketplace empowers individual service providers to register, showcase their services, manage bookings, and receive payments, all within a controlled environment. It also gives customers a centralized platform to compare services, check availability, and place bookings. This concept has been implemented in some global platforms, but customized, affordable, and CMS-compatible solutions are still lacking, especially for regional and small-scale use.

**1.2 Problem Statement**

Despite the rapid digitization of commerce and services, there is a significant gap in the availability of structured, user-friendly platforms tailored for service-based businesses—especially those that support multiple vendors. Most existing online solutions are either tailored to product-based eCommerce, require extensive technical knowledge, or demand high development costs, making them inaccessible for small to mid-sized service providers.

Local service providers often lack the technical infrastructure to create their own websites or manage online bookings, payments, and customer communications. As a result, they remain dependent on outdated methods like phone bookings or word-of-mouth promotions, leading to limited market reach, inefficient operations, and lost revenue opportunities.

On the other hand, customers frequently encounter fragmented experiences when attempting to discover and book services. There is no unified system where users can explore services by category, location, or price, compare providers, view availability in real time, and complete bookings with integrated payment options. This results in a lack of transparency, trust, and convenience.

Moreover, many existing multi-vendor platforms do not provide personalized dashboards for different user roles (e.g., admin, vendor, customer), leading to confusion and poor user engagement. They also lack essential features like order tracking, automated notifications, or secure payment integration.

Therefore, there is a clear need for a dedicated, scalable, and CMS-compatible Multi-Vendor Service Marketplace—a platform that can streamline vendor onboarding, service discovery, booking management, and secure transactions in a single ecosystem. This project aims to address this gap by offering an extensible, role-based solution that enhances both vendor capabilities and customer experience.

**1.3 Objectives**

The primary aim of this project is to develop a functional Multi-Vendor Service Marketplace that simplifies service discovery, booking, and vendor management using a web-based system. The platform should be flexible enough to support a variety of service categories and user interactions while ensuring usability, performance, and security.

The key objectives of this system include the creation of dedicated dashboards for each user type—Admin, Vendor, and Customer—along with support for real-time service listings, booking forms, order generation, and payment processing. The platform must also provide a responsive interface, advanced service filtering (by location, type, and price), and role-based access controls. Admins should have oversight of user registrations, orders, and vendor approvals. The system should be extensible and compatible with popular content management frameworks to ensure broader usability.

**1.4 Scope**

This project focuses on developing a full-stack web application that can support multiple service vendors across various industries. The platform will allow vendors to register, list their services with relevant details and media, and manage their own orders and availability. Customers will be able to explore these services, place bookings, track orders, and manage their profiles from a personalized dashboard.

The project also includes a role-based access system, meaning different interfaces and permissions are available based on user type. It incorporates features like service filtering, secure user authentication, order status management, and integrated email notifications. Although the project is built to be CMS-compatible, it is developed in a way that the core logic can be adapted for non-CMS platforms as well. Features such as review systems, messaging, or mobile apps may be considered outside the current scope but could be added in future expansions.

**1.5 Methodology**

To ensure a structured and efficient development process, this project followed a combination of the Waterfall and Iterative methodologies. The Waterfall model was applied during the initial stages of the project to define the requirements clearly, plan the development phases, and establish a complete understanding of the project's goals and scope. This helped in maintaining a clear roadmap and ensuring that each core feature—such as the user roles, service listings, and booking mechanism—was addressed systematically. Once the foundation was laid, the Iterative model was used to refine, improve, and test functionalities through cycles of development, testing, and feedback.

The development was divided into key modules, including the creation of custom user roles (Admin, Vendor, Customer), implementation of service posting features, dynamic booking systems, and secure payment integration. Each module was developed separately and tested individually before being integrated into the main system. Backend logic and database management were handled using PHP and MySQL, with custom post types and metadata structures ensuring flexibility and scalability. Frontend design was done using responsive frameworks to provide an optimal user experience on both desktop and mobile devices.

Testing and validation were a critical part of the methodology. Functional testing was conducted at each phase to ensure that all features worked as intended. Role-based testing ensured that each user had access only to relevant data and actions. Edge cases, such as invalid form submissions, duplicate user registrations, and payment failures, were also tested thoroughly. The methodology allowed for real-time feedback and continuous improvements throughout the development cycle. As a result, the system is not only functionally robust but also flexible enough for future enhancements and scalability.

***Chapter 2: Literature Review***

A solid understanding of prior research and related systems is essential for developing a well-informed and technically sound software solution. This chapter investigates existing literature, tools, and technologies that are relevant to the development of a multi-vendor service marketplace. The review focuses on platforms that enable service listing, user role management, booking systems, and transaction handling. Additionally, the research includes existing CMS platforms, plugin architectures, and service-oriented portals to highlight both the strengths and limitations of current systems.

Several online platforms facilitate service-based transactions between vendors and customers, such as Fiverr, TaskRabbit, and UrbanClap. These systems follow a core principle of service aggregation, enabling users to access various services under a single unified portal. While these platforms provide end-to-end functionality, they are proprietary and lack customization freedom for developers seeking tailored solutions for niche industries. Open-source alternatives, including various WordPress-based systems, offer a modular approach where features can be integrated via plugins, making them flexible and extensible.

In terms of architecture, many of the successful systems rely on a component-based modular framework where different user roles such as admin, vendor, and customer interact through distinct yet interconnected dashboards. Payment gateways like Stripe or PayPal are integrated to streamline financial transactions and commissions. Literature also highlights the use of REST APIs for real-time data synchronization between users and services, ensuring a smooth and responsive user experience.

A recurring limitation observed in many platforms is the lack of personalized dashboards for different user roles. Vendors often lack comprehensive tools to manage bookings, while customers experience limited tracking options post-purchase. The administrative dashboards are often generic, without real-time performance indicators or analytics. These gaps present an opportunity for systems that not only centralize marketplace features but also focus on the experience and functionality unique to each user type.

Several academic papers and technical journals also examine the use of taxonomies and metadata in filtering services, which is especially relevant to this project’s search and filter functionality. Moreover, case studies around Stripe Connect demonstrate its capability to facilitate direct vendor payouts in multi-vendor setups, eliminating the need for complex manual intervention. This provides a solid technical foundation for implementing secure and automated payment flows.

In conclusion, the literature review reflects a growing demand for flexible, modular, and role-specific service marketplaces. The reviewed platforms and research works have influenced the design choices in this project, especially in the implementation of features such as dynamic dashboards, booking systems, custom user roles, and scalable transaction mechanisms. This chapter provides the contextual basis for decisions made in system architecture, user flow, and feature prioritization in the development phase.

***Chapter 3: Requirements Specification***

A well-defined requirements specification is essential to the success of any software project. It establishes a clear understanding between the development team and stakeholders by outlining what the system must do, how it should behave, and the constraints it must operate within. This chapter documents all the functional and non-functional requirements for the **Multi-Vendor Service Marketplace** project. These requirements serve as a foundation for design, development, testing, and validation phases.

The requirements have been categorized based on their purpose and scope, and each is assigned a unique identifier for easy tracking. Functional requirements define what the system should do, such as user registration, service booking, and order management. Non-functional requirements specify the quality attributes of the system, such as performance, security, and usability.

**Requirement ID Format**

Each requirement is assigned an ID with three components, structured as follows:

* **Requirement Type**: A two-letter code indicating the nature of the requirement.
* FR: Functional Requirement
* NR: Non-Functional Requirement
* **Group Index**: A two-digit number representing the group or category.
* **Requirement Index**: A three-digit serial number for individual tracking within the group.

**Example**: FR-01-002 refers to a Functional Requirement under Group 01 (e.g. User Management), and it is the second requirement in that category.

All requirements are assigned a **priority level** from 1 to 3:

* **Priority 1**: Critical – Must be implemented.
* **Priority 2**: Important – Should be implemented.
* **Priority 3**: Optional – May be skipped if time does not allow.

**3.1 System Overview**

The purpose of the **Multi-Vendor Service Marketplace** system is to provide a robust and scalable platform that allows users to seamlessly interact with a network of service providers. The system enables vendors to register and offer their services to a broader customer base while customers can browse, filter, and book these services as per their preferences. The platform also empowers administrators to manage users, services, and platform-wide operations through a centralized dashboard.

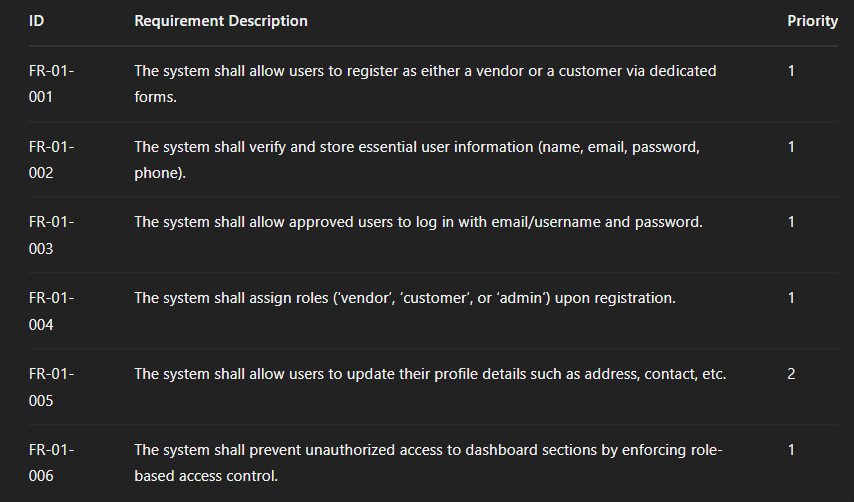
This platform offers a smooth onboarding experience for both vendors and customers, supports real-time order handling, and ensures secure financial transactions through integrated payment gateways. The entire ecosystem is designed to be modular, secure, and user-friendly—capable of scaling as the service demand and vendor base grow.

**3.2 Functional Requirements**

Functional requirements define the core capabilities and operations the system must perform to meet the expectations of its users. These requirements are categorized into logical groups, and each is assigned a unique identifier that aligns with the format specified in Figure 3-1 of this report (e.g., FR-01-001). Additionally, every requirement is associated with a priority level (1 = High, 2 = Medium, 3 = Low), allowing efficient resource allocation during implementation.

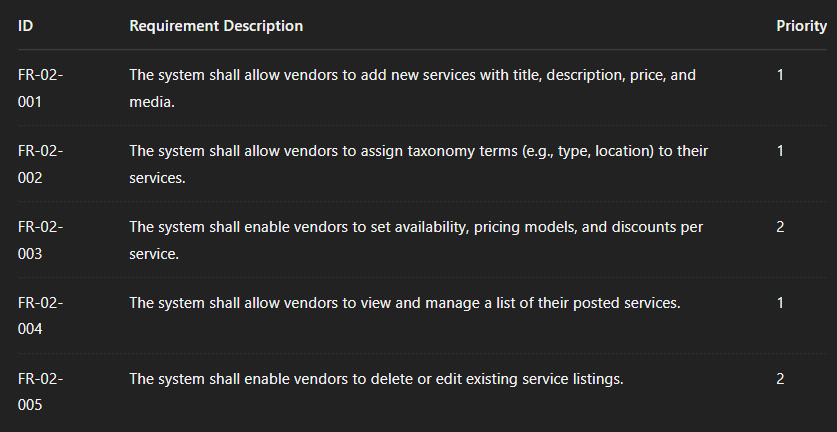
**3.2.1 Category 1 – User Management**

This category addresses user-related operations including registration, login, role assignment, and profile management.



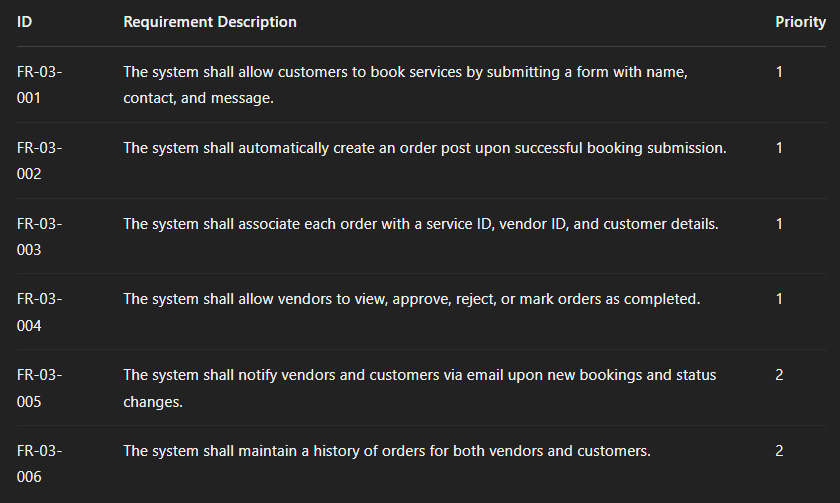
**3.2.2 Category 2 – Service Management**

This category focuses on vendor capabilities such as adding, editing, and managing service listings.



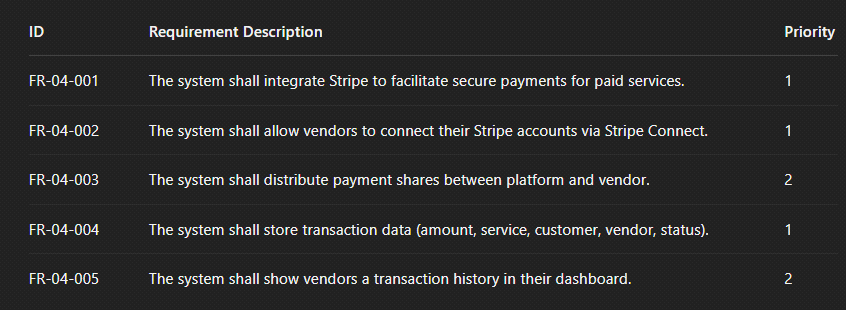
**3.2.3 Category 3 – Booking & Order Management**

This category outlines how customers interact with services through bookings, and how vendors manage received orders.



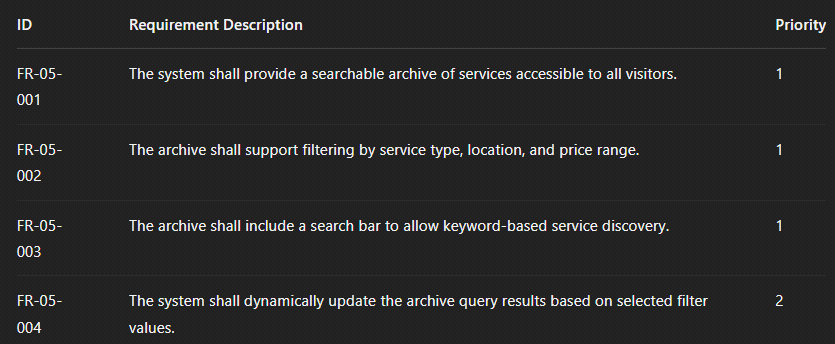
**3.2.4 Category 4 – Payment Integration**

This category includes functionality for processing payments through Stripe and managing transaction data.



**3.2.5 Category 5 – Search and Filtering**

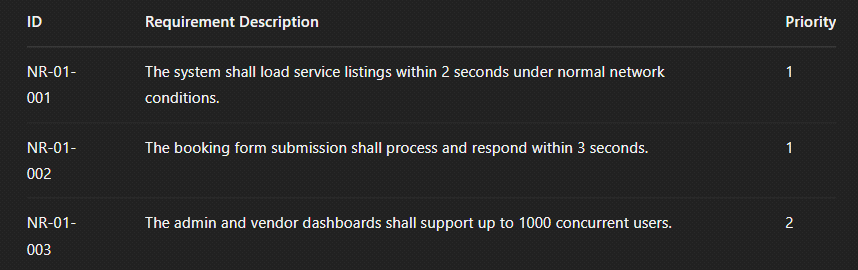
This category covers service discovery by users through dynamic filters and keyword search.



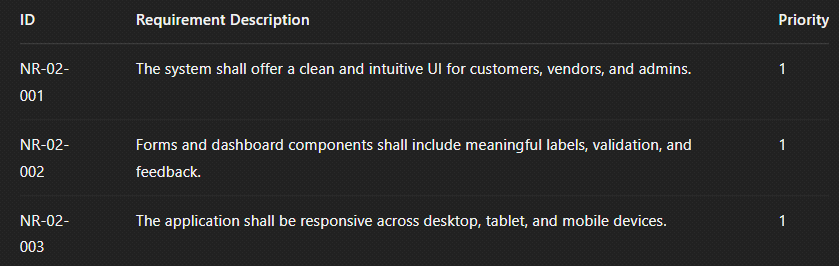
**3.3 Non-Functional Requirements**

Non-functional requirements (NFRs) define the quality attributes and operational constraints of the system rather than specific functionalities. These ensure the system is reliable, secure, and user-friendly under real-world usage.

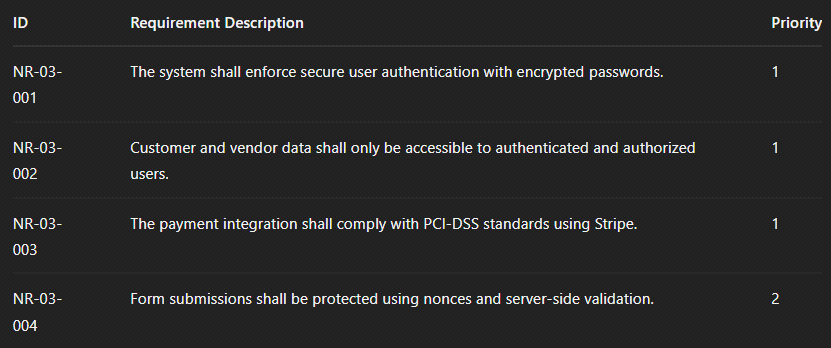
**3.3.1 Performance Requirements**



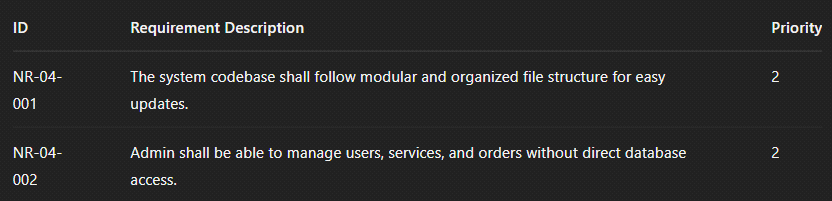
**3.3.2 Usability Requirements**



**3.3.3 Security Requirements**



**3.3.4 Maintainability Requirements**



***Chapter 5: Implementation***

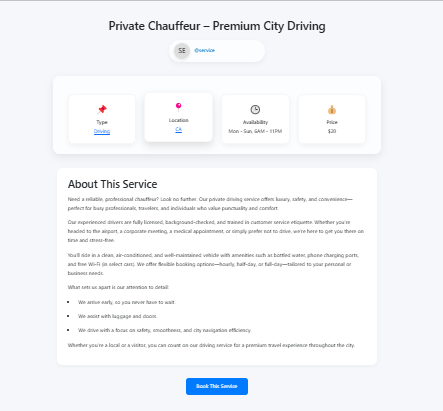
**5.1 Tools and Technologies**

This project leverages a combination of widely-used technologies and platforms to provide an efficient, scalable, and secure web-based software solution. While the platform is independent and can be adapted to any CMS, for this specific implementation, it has been developed and tested using WordPress due to its robust ecosystem and widespread use for building dynamic websites.

**1. Development Environment**

The development of this platform was primarily carried out using Visual Studio Code (VS Code), a powerful and lightweight code editor, known for its extensive feature set and integration with version control systems like Git. GitHub was used to manage the project's source code, allowing for seamless version control and collaboration. The software was developed in a local environment using XAMPP and Laragon, both popular solutions for running PHP, MySQL, and Apache locally. This setup made testing, debugging, and local development easier and faster.

**2. Frontend Development**   
 The frontend was developed using HTML, CSS, JavaScript, and jQuery. Bootstrap was employed for responsive design, ensuring that the user interface is both aesthetically appealing and functional on a wide range of devices. To add more styling flexibility, Sass (Syntactically Awesome Stylesheets) was utilized to make the CSS code more maintainable and modular. These technologies ensured that the platform is easy to use, responsive, and accessible.



**3. Backend Development**  
The backend of the web-based software was developed using **PHP**, one of the most widely used server-side scripting languages in the industry. PHP was chosen due to its versatility, ease of integration with various systems, and its ability to handle server-side processing for dynamic web applications. This choice enables the application to handle tasks such as form submissions, user authentication, data processing, and server-side rendering of content.

For database management, **MySQL** was utilized, a reliable and efficient relational database management system. The database was administered via **phpMyAdmin**, a powerful and user-friendly interface that simplifies database creation, modification, and maintenance. This tool enabled smooth interactions with the database, including running queries, managing tables, and performing data backup and restoration operations.

In terms of data storage, the platform leverages the custom tables and custom tables to efficiently manage and store data related to the plugin. For example, user-specific data is stored in **wp\_usermeta**, which holds information about customers, vendors, and their respective roles and permissions. Configuration settings are stored in **wp\_options**, which provides a central location for managing various site-wide settings, such as API keys or default preferences.

The core plugin data, including **service listings**, **orders**, and other platform-specific entities, are stored in the **wp\_posts** and **wp\_postmeta** tables. The use of these custom tables ensures compatibility with the overall system, maintaining consistency across the application while benefiting from the built-in features of WordPress, such as post revisions, metadata storage, and custom post types.

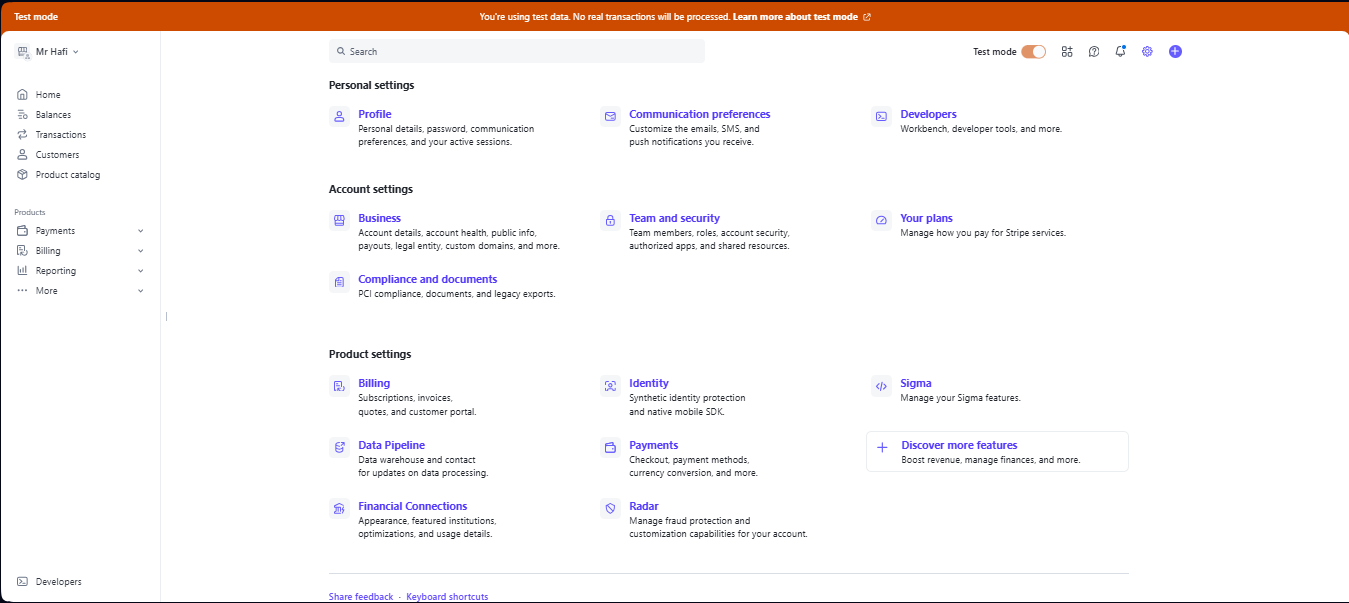
In addition to using the custom tables, a custom database called **service\_marketplace** was created to store plugin-specific data. This custom database houses important data related to services, customer orders, payment details, and additional settings. By creating dedicated tables in the **service\_marketplace** database, the plugin is able to manage and track services offered by vendors, customer orders, and vendor-specific data separately, without conflicting with the core data. This also ensures that the plugin can scale and evolve without impacting the site's overall performance.

A key advantage of using custom tables within the existing database structure is that it allows for seamless integration while maintaining optimal performance. Custom queries can be run against the plugin’s tables, allowing for efficient retrieval and manipulation of the service and order data. Additionally, utilizing the default database structure enables the plugin to benefit from built-in security features, such as user permissions and role management, which helps in protecting sensitive data.

This approach ensures that the plugin’s data is well-organized, avoids redundancy, and integrates smoothly with the existing framework. It also makes use of wp’s flexible and powerful infrastructure, allowing the plugin to function as a cohesive and scalable web-based software solution.

By carefully selecting and structuring the database schema, the backend of the web-based software not only facilitates seamless data storage and retrieval but also ensures that the plugin performs optimally, even as it grows in complexity and user load.

**4. Payment Integration**  
For payment processing, the project integrates **Stripe**, a leading online payment service. Using **Stripe's API**, the system handles payments for service bookings securely and efficiently. Customers can pay directly through the platform, and their transactions are securely processed via Stripe’s infrastructure, ensuring a seamless payment experience.



**5. Email Configuration**  
The project uses **Google's SMTP server (smtp.gmail.com)** for handling email communications between customers, vendors, and administrators. This was integrated through **WP Mail SMTP**, a tool that ensures reliable email delivery, bypassing issues associated with WordPress’ default PHP mail function. Nonces are implemented to secure email requests and prevent unauthorized access to user data during interactions.

**6. Security Features**  
Security is of utmost importance in the development of this platform. All user inputs are carefully sanitized to prevent malicious actions such as **SQL injections**, **Cross-Site Scripting (XSS)**, and other common attacks that target web applications. By utilizing **prepared statements** and ensuring that only sanitized input is processed and stored in the database, the risk of data breaches and vulnerabilities is significantly reduced. This approach ensures that no malicious data can manipulate the database or execute unintended actions.

To further enhance the security of the platform, **nonces** are used to verify that a request has been intentionally made by a valid user. Nonces are unique tokens that are generated for each request, making sure that forms or actions are not submitted by unauthorized third parties. This prevents attacks like **Cross-Site Request Forgery (CSRF)**, where a malicious user could attempt to trigger an action on behalf of another user without their consent.

In addition to these protections, user passwords are stored securely using **password hashing techniques**. This ensures that even if a database breach were to occur, sensitive data such as user passwords remains encrypted and inaccessible. Only the hashed version of the password is stored, and during login, the entered password is hashed and compared with the stored hash, preventing the exposure of the original passwords.

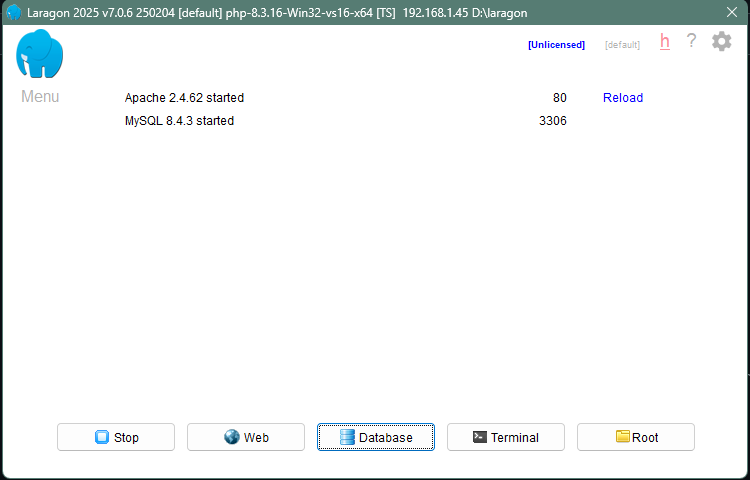
The platform also makes use of **role-based access control (RBAC)** to manage user permissions effectively. By assigning specific roles to users—such as admin, customer, or vendor—the system ensures that users can only access data and features relevant to their role. This minimizes the chances of unauthorized access to sensitive data or administrative functions, effectively preventing privilege escalation attacks.

Further security measures include the implementation of **SSL/TLS encryption** to secure data transmitted between the user’s browser and the server, especially for sensitive information like passwords and payment details. All communication is encrypted to prevent interception by malicious entities.

To prevent brute-force attacks, measures like **login attempt throttling** and **two-factor authentication (2FA)** can be integrated, although they are not part of the initial setup. Additionally, regular **security audits** and the use of **firewalls** can be employed to detect and block suspicious activity, adding an extra layer of defense against external threats.

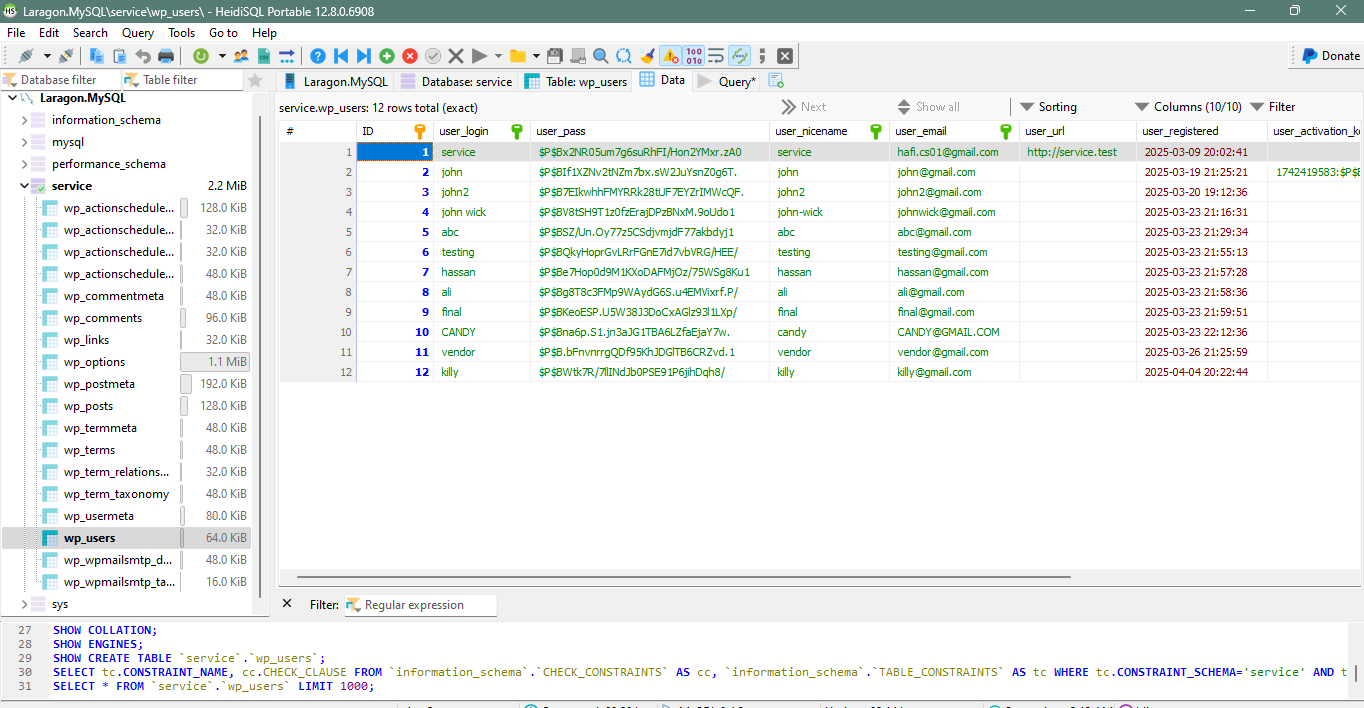
In summary, this platform incorporates robust security practices such as input sanitization, password hashing, nonces for action validation, secure data transmission, and role-based access control, ensuring that user data and system integrity are protected at all times.

**7. Local Development Environment**  
For local development and testing, **XAMPP** and **Laragon** were used to simulate a real server environment. These tools provide a robust local server with support for Apache, MySQL, and PHP, enabling smooth local development and testing. Once the platform is ready for deployment, it can be easily migrated to a live server environment.



**8. Version Control and Collaboration**  
The entire codebase was managed and stored in a **GitHub** repository, ensuring proper version control, collaboration, and traceability of code changes. GitHub also made it easier to collaborate with other team members, allowing for code review, feature branches, and issue tracking throughout the development process.

**9. Database Management**  
For managing the platform’s database, **phpMyAdmin** was used. It provided a user-friendly interface for managing database tables, running queries, and performing backups. The data related to services, users, orders, and other important details are stored in the default WordPress tables, ensuring seamless integration with the WordPress ecosystem.



**5.2 MODULES DESCRIPTION**

1. **User Roles and Permissions**
2. **Login and Registration System (Vendor & Customer)**
3. **Admin Dashboard**
4. **Vendor Dashboard**
5. **Customer Dashboard**
6. **Custom Post Type: Services**
7. **Custom Post Type: Orders**
8. **Service Archive & Single Page Display + Vendor Public Profile Pages**
9. **Booking and Order Workflow**
10. **Email & Notification System**
11. **Transactions and Stripe Payment Integration**
12. **Database Integration & Data Structure**
13. **Security Implementation**
14. **Plugin Architecture & File Structure**

**5.2 MODULES DESCRIPTION**

The development of this web-based service marketplace involved structuring the system into multiple functional modules, each responsible for handling a specific part of the application workflow. These modules were designed with scalability and maintainability in mind, ensuring a clean separation of responsibilities across user management, data storage, interface design, booking lifecycle, payment handling, and security. In this section, each of these modules will be discussed in detail to provide insight into how the complete system was architected and implemented.

The following subsections outline the internal workings and technical considerations of each module.

### ****1. MODULE 1 : User Roles and Permissions****

User role management forms the backbone of access control and defines the capabilities of each type of user within the system. The project implements a **Role-Based Access Control (RBAC)** approach to ensure that each user interacts only with the features meant for their role.

#### **1.1 Defined User Roles**

The system is designed with a **multi-user architecture**, making it crucial to define distinct roles with dedicated permissions. This ensures streamlined user experiences, robust access control, and proper segregation of duties. To achieve this, **three custom user roles** were implemented—**Administrator**, **Vendor**, and **Customer**—each tailored to perform specific tasks within the platform's ecosystem.

#### **Administrator Role**

The Administrator serves as the **superuser** of the entire system, possessing unrestricted control over both the frontend and backend operations. Administrators are responsible for managing the lifecycle of users (approving or rejecting vendors), monitoring service activity, managing payment gateways, and overseeing the overall health of the platform. Key administrative privileges include:

* Managing and moderating all users and their associated data.
* Accessing and editing all services, bookings, and orders.
* Viewing comprehensive analytics and statistics via **custom dashboard widgets** built exclusively for administrative insight.
* Setting platform-wide configurations and preferences.
* Maintaining security settings, reviewing transactions, and monitoring logs.

Administrators ensure that only verified vendors are approved to publish services and maintain a high standard of trust and compliance across the platform.

#### **Vendor Role**

Vendors are the **service providers** on the platform. They are responsible for creating, managing, and delivering services to customers. Upon successful registration and administrative approval, a vendor gains access to a **custom front-end vendor dashboard**, specifically developed for usability, accessibility, and operational efficiency.

Key capabilities granted to vendors include:

* Publishing and managing their own services through a custom post type.
* Handling service bookings and updating order statuses in real-time.
* Uploading portfolios, managing social links, and specifying service radius and availability.
* Viewing and filtering transaction logs and customer requests.
* Editing their profile details, such as business name, location, years of experience, and emergency contact information.

The vendor role has been encapsulated within a **secure and intuitive interface**, ensuring that service providers can operate independently without accessing or interfering with other system modules.

#### **Customer Role**

Customers are the **end-users** of the platform, primarily responsible for browsing services, placing bookings, and communicating with vendors. Upon registration, each customer gains access to a **personalized customer dashboard**, giving them a user-friendly environment to manage their interactions on the platform.

Customer permissions include:

* Viewing service listings and accessing detailed service pages.
* Booking services by submitting structured booking forms, including contact details, address, and preferred appointment date.
* Reviewing and managing their bookings under the "My Orders" tab.
* Updating personal information such as name, phone number, and address.
* Managing their account credentials via secure password update functionality.

The customer role is crafted with **usability, simplicity, and security** in mind, ensuring seamless interaction and engagement with the platform.

#### **1.2 Capabilities and Access Control**

The integrity of a multi-role system largely depends on how effectively **user capabilities and access rights** are defined, enforced, and maintained throughout the application. In this project, each user role—**Administrator**, **Vendor**, and **Customer**—has a unique set of permissions that govern what they can access, create, edit, or delete within the platform.

Access control is implemented through a **two-tiered approach**:

1. **Backend validation**, where server-side logic ensures that only authorized users can perform critical actions, even if malicious attempts are made through external tools or scripts.
2. **Frontend visibility control**, where UI elements and options are dynamically rendered based on the current user's role to ensure a clean and intuitive experience while preventing unauthorized actions.

#### **Role-Based Restrictions**

* **Vendors** are explicitly restricted from accessing administrative dashboards or sensitive system settings. They cannot modify global platform configurations, access other users’ data, or interfere with the services or orders of other vendors. Their access is limited to content authored by themselves, such as their own services, orders, and profile data.
* **Customers** are given strictly limited access intended for end-user interactions. They cannot create or manage services, access revenue information, view vendor data unrelated to their bookings, or interact with backend functionalities. Their interaction with the system is limited to browsing, booking, and profile management.
* **Administrators** possess the highest level of authority and can access all system modules, including user management, plugin settings, data analytics, and transaction logs. Only administrators have the ability to **approve or reject vendor registrations**, an action that is hidden and restricted from all other roles.

#### **1.3 Front-End vs Back-End Role Management**

Instead of relying on the default WordPress admin interface, the project introduces fully isolated **front-end dashboards** for both vendors and customers. This provides a more intuitive and secure experience while preventing accidental access to sensitive backend tools.

When a vendor or customer logs in, they are redirected to their respective dashboards using conditional login logic. Users trying to access restricted areas are redirected and shown error messages, maintaining integrity of role-based segregation.

#### **1.4 Approval Workflow for Vendors**

To maintain the authenticity and integrity of service providers on the platform, a manual approval workflow has been implemented for all newly registered vendors. This mechanism ensures that only legitimate, verified, and authorized users are allowed to offer services to customers.

Upon vendor registration, the system automatically assigns a custom user meta field called \_vendor\_approval\_status, which is initialized with the value pending. This status acts as a control flag, determining whether the user should be granted access to vendor-specific functionalities such as the dashboard, service publishing, and order management.

Until approved, the vendor is restricted from accessing the core vendor features. They are redirected to a “Registration Pending” page upon login, informing them that their application is under review. This creates a controlled onboarding funnel, ensuring that unverified users cannot operate as vendors or interact with customers through the system.

#### **Administrator Approval Interface**

The approval and rejection process is streamlined through the WordPress Admin Users panel. A custom column is added to the users table, displaying the current approval status of each vendor. Admins can **manually approve or reject** vendors with a single click, triggering the appropriate change in the \_vendor\_approval\_status field.

Once approved:

* The vendor gains immediate access to the **front-end vendor dashboard**.
* They are notified that they can now begin offering services.
* Their status is updated to approved, allowing unrestricted access to vendor functionalities.

If rejected:

* The user is blocked from accessing vendor features.
* Their status is updated to rejected.
* This prevents misuse of the platform and reinforces quality control.

This **approval workflow adds an additional layer of security and trust**, ensuring that vendors are properly vetted before being allowed to operate. It protects the platform from fraudulent users, enhances customer confidence, and gives the admin full control over the vendor ecosystem.

#### **1.5 Flexibility and Scalability**

The role system was designed to be extensible. New roles or capabilities can be introduced easily in the future without restructuring the current system. Capability-based control (rather than hardcoded role names) makes the platform modular and adaptable.

Whether the system is scaled up for enterprise use or integrated with third-party tools, this structured role-based approach ensures long-term sustainability.

### ****2. MODULE 2 :**** Login and Registration System (Vendor & Customer)

**2.1 Dual Authentication System**

The platform implements a **robust dual authentication system** to accommodate the needs of two distinct user types — **Vendors** and **Customers**. Recognizing that each user type interacts with the system in unique ways, the login and registration processes are **fully separated**, ensuring that users have a tailored experience based on their role. This separation also simplifies user management and reduces the risk of role conflicts or unauthorized access.

To support this structure, the system includes **dedicated pages for each user type**, namely:

* **Vendor Login and Vendor Registration**
* **Customer Login and Customer Registration**

These pages are not shared or generalized, but are **specifically built and optimized** for each role. This ensures that all form submissions are routed through their **own independent handling logic**, allowing for role-specific checks, validations, and actions. For instance, vendors may be required to provide business-related details such as company name, category, and service radius, while customers follow a simpler registration process with just basic contact and identification information.

Each of these pages is automatically **created at the time of plugin activation** and is connected to a **shortcode**, which renders a well-structured and styled form using **Bootstrap components**. This not only ensures responsiveness across devices but also contributes to a user-friendly and professional-looking interface.

The form handlers behind these pages are responsible for:

* Assigning the appropriate role (vendor or customer)
* Storing additional metadata such as phone numbers and addresses
* Redirecting users to their respective dashboards after successful login or registration
* Ensuring security through nonce verification and input sanitization

This dual-system architecture helps prevent misuse (e.g., a customer accidentally gaining vendor access) and strengthens overall access control. By isolating the login and registration processes for each user role, the platform maintains **a clear boundary between functionalities**, leading to a **more secure, maintainable, and scalable** application environment.

**2.2 Registration Workflow & Validation**

The registration system has been meticulously designed to ensure data integrity, security, and accurate role assignment. Both **Vendor** and **Customer** registration forms follow a structured workflow and incorporate comprehensive validation mechanisms to eliminate errors and prevent malicious data entry.

Each form is built with clearly labeled fields and intuitive input constraints to guide the user. Behind the scenes, the system employs a series of **server-side validations** to verify and sanitize all submitted data before proceeding with account creation. These validations are essential for protecting the platform from SQL injections, XSS attacks, and accidental data corruption.

Some of the key validation features include:

* **Input Sanitization**: All submitted fields, including full name, email, phone number, address, and business information, are sanitized using native functions like sanitize\_text\_field(), sanitize\_email(), and esc\_url() to ensure only clean, safe data is processed and stored.
* **Password Verification**: The password and confirm password fields must match exactly. Mismatches trigger immediate errors and halt the registration process.
* **Email Duplication Check**: Before creating a new account, the system checks whether the submitted email already exists using the email\_exists() function. This ensures each user is uniquely identified by their email address.
* **Conditional Role Assignment**:
  + **Vendors**: When a vendor completes the registration form, they are assigned the vendor role. However, their account is not immediately activated. A custom user meta field named \_vendor\_approval\_status is set to pending, requiring manual approval from an admin before access to the vendor dashboard is granted.
  + **Customers**: Customers, on the other hand, are directly assigned the customer role and are redirected to the login page upon successful registration. They can immediately access their dashboard and start browsing and booking services.
* **Meta Data Storage**: Additional details collected during registration, such as phone number, business category, years of experience (for vendors), gender, and street address, are stored using update\_user\_meta(). This data is later used to pre-fill profile forms and personalize the dashboard experience.

This strict and well-defined registration workflow ensures that all users enter the system with properly structured, secure data. It also helps the platform enforce its approval workflows for vendors while streamlining customer access — all without compromising security or user experience.

.

**2.3 Login Process & Role Verification**

During login, the system ensures that the user logs in via the correct portal:

* **Vendors** logging in via the vendor form are checked for the vendor role and approved status.
* **Customers** are only allowed to proceed if their role is customer.
* If the user does not belong to the expected role, they are **automatically logged out**, and an error message is displayed (e.g., “Only vendors can log in from this form”).

Role-based redirection is handled post-login:

* Vendors → Vendor Dashboard
* Customers → Customer Dashboard

**2.4 Security and CSRF Protection**

The platform places a strong emphasis on safeguarding user credentials and sensitive data during the login and registration processes. Multiple layers of security are embedded into the authentication system to protect against common vulnerabilities such as CSRF (Cross-Site Request Forgery), data tampering, and unauthorized access attempts.

One of the most critical implementations in this regard is the use of **nonces** (a cryptographic token that stands for “number used once”). These are generated using wp\_nonce\_field() and verified upon form submission with wp\_verify\_nonce(). This mechanism ensures that only legitimate users are able to submit forms and that the request originates from the authorized page. If a nonce fails verification, the submission is immediately blocked, effectively preventing CSRF attacks.

Additionally, all form submissions are handled exclusively on the **server side**, which allows for stricter validation and prevents any sensitive operations from being exposed to or manipulated via frontend JavaScript. This ensures that unauthorized scripts or bots cannot bypass the validation layer or tamper with user data.

Furthermore, all user passwords are securely hashed and stored using WordPress’s built-in authentication system, which utilizes strong one-way hashing algorithms like bcrypt. At no point are plain-text passwords stored, retrieved, or exposed in the application.

The combination of **nonce-based request validation**, **input sanitization**, **password hashing**, and **strict backend processing** establishes a robust security framework. This ensures that the authentication workflows for both vendors and customers are protected against injection attacks, session hijacking, and data theft, maintaining the trust and safety of the platform.

**2.5 Enhanced User Experience**

The login and registration forms were built with **Bootstrap 5 UI components** for a clean and responsive experience across devices. Error handling, redirection, and messages are all handled with user clarity in mind.

To further improve user interaction:

* A “Registration Pending” message is shown to unapproved vendors.
* Redirects are applied conditionally based on role and status.
* After registration, users are clearly informed about next steps (e.g., “Your application is under review”).

### ****MODULE 3 :**** Admin Dashboard

The Admin Dashboard serves as the centralized control panel for the system’s highest-privileged users. It provides full oversight of users, services, orders, transactions, and platform analytics. The dashboard has been tailored to offer both granular management capabilities and high-level overviews, streamlining the operational aspects of the service marketplace.

### ****3.1 Overview and Accessibility****

The Admin Dashboard is the nucleus of operational control, providing the system administrators with privileged access to all critical functionalities. It is purposefully designed to maintain oversight over the entire ecosystem of the marketplace—users, services, orders, and transactions—while preserving a high level of security and usability.

Upon logging in, administrators are automatically redirected to the CMS’s backend interface, where the custom admin dashboard features are seamlessly integrated using hooks and menu registration functions. This integration ensures a native experience, allowing administrators to manage the platform without navigating away from the familiar WordPress admin panel. The dashboard components are appended using functions like add\_menu\_page() and add\_submenu\_page() that create organized entries in the sidebar navigation for quick access to various modules.

To maintain strict access control, only users assigned the administrator role are permitted to access the Admin Dashboard. This role-based access is programmatically enforced using conditional checks such as current\_user\_can('administrator'), ensuring that sensitive data and administrative tools remain inaccessible to unauthorized users.

Additionally, the dashboard’s architecture supports **modular expandability**, allowing future extensions such as analytics charts, user activity logs, plugin configurations, or notification logs to be added without restructuring the existing layout. Every admin-facing interface is styled for clarity and consistency, utilizing a blend of native CMS styles and light custom enhancements to improve visual hierarchy and information readability.

The interface not only empowers admins to perform direct actions—such as approving vendors, updating orders, or managing service listings—but also provides **high-level summaries** through live statistics and real-time data widgets. These tools enable the administrators to track the health of the platform, identify bottlenecks, and proactively manage user engagement and service quality.

### ****3.2 Custom Widgets and Analytics****

To provide a snapshot of platform activity, a set of **real-time dashboard widgets** were implemented using the wp\_add\_dashboard\_widget() function. These include:

* **Total Vendors** – Counts and displays all registered vendor accounts.
* **Total Customers** – Displays total number of registered customers.
* **Orders Today** – Uses a date query to count orders placed on the current day.
* **Pending Orders** – Displays how many orders are still awaiting vendor approval or processing.
* **Total Services** – Shows the number of published services on the platform.

These widgets are rendered with custom styling and logic, making it easier for administrators to monitor platform metrics at a glance. Each widget pulls live data from the database, ensuring the information is always up to date.

### ****3.3 Vendor Approval Management****

A dedicated vendor approval system was built directly into the **Users panel** of the admin interface. Upon registration, all vendor accounts are flagged as **“pending”** using a custom user meta field (\_vendor\_approval\_status). Admins can then manually approve or reject vendors via clickable links next to each user's name. Once approved, vendors gain full access to their dashboard and service management tools.

This manual approval step ensures that only verified service providers are allowed to operate on the platform, improving both quality control and user trust.

### ****3.4 Order and User Management****

From the admin area, administrators can:

* View and manage all orders (regardless of vendor ownership)
* Change the status of any service order (pending, approved, completed, canceled)
* Access vendor and customer profiles
* Edit, delete, or reassign services
* Manually create or delete user accounts

All of these operations are handled securely through native CMS functions, with backend validation, user capability checks, and audit-friendly logging mechanisms in place.

### ****3.5 Security and Role Enforcement****

Every admin tool or menu option is role-restricted, ensuring no unauthorized user can view or access sensitive data. Admins also have access to **plugin settings**, where they can update email templates, manage API keys (such as Stripe), and configure global behavior for services or orders.

These features are only visible and executable to users with the administrator capability, as verified using current\_user\_can('manage\_options') and similar functions.

### 4. ****MODULE 4 :**** Vendor Dashboard

### ****4.1 Purpose and Role-Based Access****

The Vendor Dashboard serves as a dedicated control panel for service providers registered on the platform. Its primary purpose is to offer vendors a user-friendly interface for managing their business operations—from service creation and order management to profile updates and tracking transactions. This dashboard is strictly accessible to users assigned the vendor role, and access is validated programmatically using role-based authentication mechanisms like current\_user\_can() or role array checks to prevent unauthorized entry.

Unlike the admin area, vendors operate exclusively on the frontend through a fully customized interface. This ensures that vendors remain within the branded ecosystem and never access the core CMS backend, preserving both security and a consistent UX.

**4.2 Layout and Navigation Structure**

The dashboard is constructed using a tabbed layout, where each tab corresponds to a specific function or module. The main tabs include:

* **Profile** – For viewing and editing personal and business information such as name, contact, business name, category, website, social links, and address.
* **My Services** – Vendors can list, update, or delete their services. This tab displays all services submitted by the vendor using a card-based responsive UI.
* **Orders** – Displays incoming service bookings with details like customer name, contact info, service type, date, status, and notes. Vendors can change order statuses (e.g., Approved, Completed, Canceled).
* **Transactions** – Summarizes all completed orders along with transaction dates, payment amounts, and relevant service IDs.
* **Logout** – A one-click logout button that immediately logs the vendor out and redirects them to the homepage or login screen.

Each section is built with responsive front-end frameworks like **Bootstrap**, ensuring accessibility across devices.

### ****4.3 Dashboard Tabs Breakdown****

#### **4.3.1 Profile Tab**

The **Profile Tab** is the central space for vendors to manage and update their personal and professional information. Designed with an intuitive two-column layout and Bootstrap styling, it presents a clean, modern, and accessible interface. The information is prefilled dynamically using stored user meta, enhancing the experience and reducing redundant input for the user.

This section includes the following major features:

#### **Greeting Header**

At the top of the profile card, the system greets the vendor using their full display name and displays their email address beneath. This dynamic greeting not only adds a personal touch but also confirms to the user that they're logged into the correct account.

#### **Profile Picture Display**

The vendor’s profile photo is prominently shown in a circular thumbnail. If a profile image is not uploaded, a fallback image or placeholder is used. This visual identity element helps differentiate vendors and gives a more human-centric feel to the platform.

#### **View Profile & Edit Buttons**

On the right-hand side of the header, two interactive buttons are available:

* **View Profile**: Opens the vendor’s public profile in a new tab, allowing them to preview how clients see their information.
* **Edit Profile**: Unlocks form fields for editing, toggling the mode between read-only and editable.

#### **Personal and Business Information Fields**

Each field is dynamically populated and editable when "Edit Profile" is activated. The following fields are included:

* **Nickname**: Vendor's display name or username.
* **Gender**: Drop-down or text input, used for profile personalization.
* **Phone**: Contact number for customer communication.
* **Country**: Country or regional location of the vendor.
* **Website**: Personal or business website, displayed as a hyperlink.
* **Service Location**: City or area where services are provided.
* **Business Name**: Official name under which services are offered.
* **Business Category**: Dropdown or text input showing the industry or specialization (e.g., Electrical, Plumbing).
* **Emergency Contact**: Optional backup contact number.
* **Social Links**: Optional field for linking LinkedIn, Facebook, or portfolio profiles.
* **Service Radius**: Shows how far the vendor is willing to offer services from their base location (e.g., 10km).
* **National ID**: Optional identification number field (may be hidden based on use case).
* **Portfolio File**: Displays a thumbnail or download link of an uploaded document or image showcasing the vendor’s past work.

All fields are validated both client-side and server-side. Updates are securely stored using update\_user\_meta() with proper sanitization, and security tokens (wp\_nonce\_field()) ensure that only authenticated users can submit changes.

#### **4.3.2 Services Tab**

The **Services Tab** within the vendor dashboard is a core feature that empowers service providers to manage their offerings in a visual and organized way. This tab displays all services added by the currently logged-in vendor in a card-based grid layout, providing a quick overview of each listing’s details and current status.

Each card represents a single service, populated dynamically from the custom post type (service), and includes relevant metadata such as price, location, duration, discount, and availability. This user-friendly layout enhances navigation and allows vendors to efficiently manage their catalog of services.

#### **Key Features and Functionalities**

##### **Service Cards Display**

Each card in the grid view presents essential information about the service:

* **Service Image**: Pulled from the featured image (post thumbnail).
* **Title**: The service title as set by the vendor.
* **Status Badge**: Indicates if the service is “Active” or “Inactive”.
* **Price & Discount**: Shows the standard price and any applicable discount.
* **Duration**: Specifies how long the service takes ex. 2hrs.
* **Location**: Displays the city or area (retrieved from taxonomy or meta).
* **Availability**: Shows the vendor’s defined availability window (optional).

This gives vendors a real-time summary of their offerings at a glance.

##### **➕ Add New Service Button**

Located at the top right of the page, the **Add New Service** button allows vendors to create new service listings. When clicked, the user is redirected to a custom service submission form which consist of all the fields of Service.

All inputs are securely handled, validated, and stored using the WordPress custom post type infrastructure (wp\_posts and wp\_postmeta).

##### **✏️ Edit Button**

Each service card includes an **Edit** button, which redirects the vendor to a seperate custom service submission form, but in pre-filled mode. Vendors can modify any details of their previously submitted services. Changes are saved using wp\_update\_post() and update\_post\_meta() functions.

This allows vendors to keep their listings up to date without having to delete and recreate services from scratch.

##### **🔍 View Details Button**

This button opens the **service single page** in a new tab, letting the vendor see what their listing looks like to customers. This preview includes a full-width image, detailed description, taxonomy labels, and booking CTA (Call to Action) button.

##### **🗑️ Delete Button**

The delete button triggers a soft-delete (moves the post to trash) after confirming via a popup. This ensures that vendors can remove outdated or no-longer-available services while still retaining the data in the WordPress trash system.

#### **4.3.3 Orders Tab**

The **Orders Tab** provides vendors with a structured and centralized interface to manage all customer bookings for their services. It enhances order visibility and enables vendors to take timely actions—like updating the order status or viewing customer details—directly from their dashboard without needing to access the WordPress backend.

This section significantly contributes to the platform’s goal of a self-contained front-end experience for service providers.

#### **Tabbed Layout Based on Status**

Orders are grouped by their current status into collapsible sections for streamlined access:

* 🟡 **Pending Orders**: New service bookings awaiting vendor approval.
* 🟢 **Approved Orders**: Orders that have been confirmed and accepted by the vendor.
* 🔵 **Completed Orders**: Orders that have been fulfilled and marked as completed.
* 🔴 **Canceled Orders**: Orders that were rejected or canceled due to any reason.

Each section dynamically shows the count of orders within its respective status using badges and can be expanded or collapsed as needed.

#### **Order Table Columns**

Within each tab, the following details are displayed for each order:

* **Service ID**: The ID of the booked service.
* **Customer Name**: Full name of the customer who placed the order.
* **Email & Phone**: Contact information for easy communication.
* **Status Dropdown**: A dynamic dropdown that lets the vendor update the order status in real-time.
* **Details Button**: Opens a modal containing the full booking request details, including message, preferred date, address, and service information.

#### **Status Update Feature**

Vendors can change the status of any order directly from the dropdown. Upon selection, a confirmation popup appears to avoid accidental updates. Once confirmed, the order status is updated in the database and:

* The customer is notified via email about the status change.
* The order is moved to its corresponding section.

This interaction is handled using secure form submissions or AJAX for better UX, depending on your implementation.

#### **Email Notifications (Behind the Scenes)**

Each time an order is updated (approved, rejected, or completed), the system sends automated notifications to the customer, keeping them informed in real-time. These emails are handled via SMTP (using Gmail’s secure gateway) to ensure proper delivery.

#### **Vendor Convenience & Efficiency**

This tab allows vendors to:

* Avoid backend navigation.
* Quickly respond to new bookings.
* View consolidated booking history.
* Maintain professionalism with timely responses and email confirmations.

#### **4.3.4 Transaction Tab**

The **Transactions Tab** provides vendors with a dedicated space to manage and monitor their payment-related activities. This tab plays a vital role in enabling financial connectivity between the platform and service providers by integrating **Stripe**, one of the most reliable and secure payment gateways available.

#### **Stripe Account Connection**

At the core of this tab is a **Connect with Stripe** button, which prompts vendors to link their Stripe account with the platform.

* **Purpose**: Without connecting to Stripe, vendors will not be able to receive payouts for booked services.
* **Process**: Clicking the button redirects the vendor to Stripe’s secure onboarding flow. Once completed, the vendor's Stripe account is linked to their user profile via a stored Stripe account ID.
* **Security**: The connection process uses Stripe’s OAuth protocol, ensuring secure token-based communication without exposing sensitive information.

#### **Why Stripe?**

Stripe was chosen for its developer-friendly APIs, excellent security features, global compliance, and support for various payout models. This makes it ideal for marketplace-style platforms like this, where vendors act as sub-merchants.

#### **Payment Flow Overview**

1. **Customer Books a Service**: The customer is charged via Stripe during the booking.
2. **Platform Receives the Payment**: Funds are held temporarily until the vendor accepts and completes the order.
3. **Vendor Receives Payout**: Once the order is completed, the vendor's connected Stripe account is credited.

This mechanism ensures financial fairness, transparency, and flexibility for both customers and vendors.

**4.4 Order and Booking Management**

When a customer books a service, a new entry is created in the service\_orders post type. These orders are listed under the **Orders** tab for the respective vendor who owns the booked service. Vendors can:

* View full customer details
* Change order statuses using dropdown menus (e.g., Pending → Approved)
* Use modals/popups for full order detail previews

Email notifications are triggered automatically on key status changes to keep both vendor and customer informed.

### ****4.5 Security and Access Control****

Strict validation is enforced to ensure vendors can only manage their own data. Sensitive actions like editing/deleting services or updating order statuses are protected using **nonces** to prevent CSRF attacks. File uploads are handled through wp\_handle\_upload() with MIME type restrictions to prevent execution of malicious files.

### 5. ****MODULE 5 :**** Customer Dashboard

#### **5.1 Overview & Accessibility**

The Customer Dashboard is a dedicated, front-end interface built to facilitate interaction for users registered under the "customer" role. It eliminates the need for customers to access the backend admin area, providing them instead with a clean, intuitive, and role-specific environment. Upon login, customers are automatically redirected to this dashboard, which is dynamically rendered based on their user session.

This dashboard acts as the central hub for customers to manage their activities within the platform — from profile updates to booking reviews. All data displayed is fetched in real-time using secure WordPress functions and user meta information. The interface is fully responsive and styled using Bootstrap, ensuring optimal experience across devices.

#### **5.2** Dashboard – Detailed Overview

#### The Customer Dashboard is designed to deliver a smooth, intuitive user experience, using a combination of Bootstrap and custom styling for responsiveness and visual clarity. Upon login, customers are immediately redirected to this dashboard, bypassing the backend entirely, reinforcing role-based access control.

#### At the top of the dashboard is a personalized greeting banner such as “Hello, John Doe 👋”, which dynamically fetches the user's display name. Adjacent to this greeting is a circular avatar placeholder showing the first letter of the customer's name, creating a simple yet elegant identity block.

#### A logout button is clearly positioned on the top right to allow easy session termination. Below the header, the main navigation bar uses pill-style tabs for switching between “Profile” and “My Orders” — the two core functionalities for customers.

**Profile Tab**

The **Profile** tab allows customers to view and update their personal details securely. The form is presented in a neatly styled card layout, split into two columns for visual balance and usability.

##### **1. Full Name**

* Editable field allowing customers to update their display name.
* Data is saved using wp\_update\_user() with proper sanitization (sanitize\_text\_field()).

##### **2. Email Address**

* Displayed in a read-only input field with a label: “Email (cannot be changed)”.
* This restriction is intentional to maintain login identity consistency and prevent accidental lockouts.

##### **3. Phone Number**

* Editable input field used for contact purposes in service orders and notifications.
* Stored via update\_user\_meta() under a unique meta key (phone).

##### **4. Address**

* Multiline textarea that stores the customer’s street address.
* This information is used during booking submissions and displayed to vendors as part of order details.

##### **5. Save Changes Button**

* Triggers a secure form submission protected with a nonce (wp\_nonce\_field()).
* Upon submission, all updated values are validated and sanitized before being saved.

##### **6.** Change Password Feature

To the right of the address field, a **“Change Password”** button is placed. When clicked, it reveals additional fields for:

* Current Password
* New Password
* Confirm New Password

This feature enforces strong security practices:

* Verifies old password using wp\_check\_password().
* Confirms that new passwords match.
* Updates password using wp\_set\_password() followed by immediate logout to prevent session misuse.

**Orders Tab**

The **My Orders** tab provides customers with a streamlined and visually organized view of all the service bookings they’ve made through the platform. Designed for clarity and efficiency, this section empowers customers to track the status of their engagements and stay updated on their requests.

#### **1. Layout and Presentation**

Each order is encapsulated in a card-like structure, making it easy to distinguish between multiple bookings. Orders are sorted in descending order, with the most recent ones appearing first. At the top of each card, a heading such as **Order #1 – 1234512** clearly displays the order number and associated service ID, giving users a unique reference point.

The interface uses clean columns with the following sections for each order:

* **Date & Time**: Displays the preferred date chosen by the customer at the time of booking.
* **Vendor Name**: The service provider associated with the order.
* **Vendor Email**: For contact purposes or follow-ups, the registered email of the vendor is shown.
* **Additional Notes**: Any custom notes or messages the customer left for the vendor during booking.
* **Order Status**: A colored status badge (e.g. Pending, Approved, Completed, or Canceled) indicates the current state of the order. This is updated by the vendor in their dashboard and reflects in real-time for customers.

#### **2. Real-Time Order Tracking**

The tab provides **live updates on order statuses**, enabling customers to monitor the progress without needing to contact the vendor or admin. Status changes triggered by vendors (such as approving, completing, or canceling an order) are dynamically reflected in this tab. This ensures that communication remains transparent and minimal friction is introduced in post-booking interactions.

#### **3. Security & Data Handling**

* Order details are displayed only to authenticated users with the customer role.
* Orders are fetched from the database using a filtered WP\_Query, ensuring that only orders placed by the currently logged-in user are visible.
* Input fields like additional notes are escaped using esc\_html() or nl2br() for safe rendering.
* No edit or delete functionality is available to customers, safeguarding the integrity of historical data.

#### **5.3 Security & User Restrictions**

To maintain the integrity and privacy of the system, strong security practices are enforced throughout the Customer Dashboard. First and foremost, access to the dashboard is role-restricted — only users with the "customer" role can access this area. Role verification is done through both session checks and current\_user\_can() validations, ensuring that unauthorized users are denied entry and redirected appropriately.

Each form submission on the dashboard is protected using **nonces** — cryptographic tokens that prevent **Cross-Site Request Forgery (CSRF)**. These are generated using wp\_nonce\_field() and validated via wp\_verify\_nonce() before any action is processed. In addition, all inputs are strictly sanitized using WordPress sanitization functions like sanitize\_text\_field() and sanitize\_email() to prevent **SQL injection**, **XSS**, and other attack vectors.

By integrating both frontend restrictions and backend validations, the Customer Dashboard ensures a secure, private, and seamless user experience while maintaining platform trust and usability.

### 6. ****MODULE 6 :**** Services – Custom Post Type

### ****6.1 Overview****

The **Services** module is one of the core components of the platform, built using a **Custom Post Type (CPT)** structure. Each service that a vendor publishes is treated as an individual post under this custom type. This approach leverages the underlying content management system's post handling capabilities while allowing full customization of service-related metadata, taxonomies, and templates.

This CPT provides a clean and scalable way to manage service listings, making it easy to integrate with other modules like booking, filtering, and vendor profiles. The system ensures that each service entry is tightly linked with the vendor who created it and stores all necessary information such as title, price, duration, availability, images, and more.

**6.2 Registration and Configuration**

The CPT service is registered using register\_post\_type() with the following properties:

* **Label:** Services
* **Visibility:** Public with support for frontend viewing and custom query access
* **Supports:** Title, Editor, Thumbnail
* **Rewrite Slug:** /services/
* **Capability Mapping:** Tied to the vendor role to restrict unauthorized access
* **Menu Icon:** Custom icon in admin dashboard for clarity

**6.3 Meta Fields Structure**

Each service includes a comprehensive set of **custom meta fields** to capture all essential service attributes. These fields are stored using add\_post\_meta() and update\_post\_meta():

* **\_service\_price** – Numeric cost of the service (required)
* **\_service\_discount** – Optional discount value offered by the vendor
* **\_service\_duration** – Estimated delivery time or service length
* **\_service\_schedule** – Text input or structured availability (e.g., Mon-Fri 9am–5pm)
* **\_service\_location** – Selected city or region from a predefined taxonomy
* **\_service\_gallery** – Array of image/media IDs uploaded via WordPress Media Library
* **\_service\_status** – Indicates whether service is Active or Inactive
* **\_service\_availability\_type** – Defines service mode: Online, In-person, or Both
* **\_service\_max\_bookings\_per\_day** – Integer input defining the vendor’s daily booking capacity
* **\_service\_availability\_schedule** – Optional field allowing vendors to define recurring availability (structured or free-text)
* **Featured Image** – Standard WordPress post thumbnail, shown on archive and single service page

These fields allow the platform to capture all necessary business information while offering a dynamic and scalable structure for service management.

**6.4 Taxonomy Classification**

Two custom taxonomies are registered to aid in filtering and classification:

* **service\_type** – Categorizes services by type (e.g., Plumbing, Cleaning, Electrical)
* **service\_location** – Assigns a geographic region or city for the service

These taxonomies are used extensively on the **service archive page** to enable dynamic filters via dropdowns and query parameters.

**6.5 Vendor Binding and Permissions**

Every service post is authored by a vendor and linked using the post\_author parameter. This ensures:

* Vendors can only manage their own services
* Admins retain full editing/deletion rights
* Vendor public profiles can easily fetch all services using author query filters
* Data security and role-based access control are maintained

**6.6 Frontend Vendor Management**

From the vendor dashboard, users can manage services through a simplified, Bootstrap-powered interface:

* **Add New Service** – Opens a form to input all custom fields
* **Edit Service** – Pre-populates an update form for editing existing services
* **Delete Service** – Removes the service with confirmation prompts
* **View Service** – Directs to the single service page

All operations are protected using WordPress nonces and capability checks to prevent unauthorized actions.

**6.7 Display Templates**

* **Archive Page (archive-service.php)** – Renders services in a grid layout with filters for location, category, and price
* **Single Page (single-service.php)** – Displays full service details including image gallery, vendor contact info, availability, and booking form

Both templates are mobile-friendly and stylized using Bootstrap and custom CSS, offering a modern and intuitive UI for visitors.

### 7. ****MODULE 7 :**** Orders – Custom Post Type

**7.1 Overview**

The **Orders** module was implemented as a dedicated **Custom Post Type (CPT)** named service\_orders to manage all service bookings made by customers. Each time a customer submits a booking request through a service page, an order is automatically created and linked to the relevant vendor, service, and customer. This approach enables complete separation between content and transactional data, providing scalability and extensibility for future enhancements like analytics, payment tracking, and automation.

Using the native post infrastructure ensures smooth integration with the platform’s existing features such as user roles, metadata storage, and access control, while allowing administrators to review and manage all orders directly from the WordPress dashboard.

**7.2 CPT Registration and Structure**

The service\_orders post type is registered using register\_post\_type() with the following configurations:

* **Visibility:** Admin-only (not publicly queryable)
* **Supports:** Title only (meta-driven architecture)
* **Admin Menu:** Displays under a custom menu
* **Permissions:** Only vendors and admins can view their relevant orders

The post title is typically generated dynamically, such as "Booking: John Doe", based on the customer’s name for easy identification.

**7.3 Meta Fields and Booking Details**

Each order stores essential details in custom meta fields using update\_post\_meta() and get\_post\_meta(). These fields ensure all transactional information is structured and easily retrievable:

* **\_service\_id** – References the booked service’s post ID
* **\_vendor\_id** – Stores the vendor (author) of the service
* **\_customer\_name** – Full name of the customer submitting the booking
* **\_customer\_email** – Contact email of the customer
* **\_customer\_phone** – Customer’s phone number for direct communication
* **\_order\_status** – Status of the order (Pending, Approved, Completed, Canceled)
* **\_order\_notes** – Any optional message or special request added by the customer

Each order post also inherits a post\_status of publish, enabling backend visibility while still relying on meta values for data segmentation.

**7.4 Order Status Workflow**

The \_order\_status meta field governs the entire lifecycle of an order:

* **Pending** – Default status upon submission
* **Approved** – When a vendor accepts the request
* **Completed** – After successful service delivery
* **Canceled** – If the vendor rejects or the customer cancels

Vendors can change the status from their dashboard using a dropdown interface, which triggers an email notification to the customer using wp\_mail().

**7.5 Vendor & Customer Binding**

Orders are bound to:

* **Vendor:** Using the \_vendor\_id field, ensuring each vendor only sees their orders
* **Customer:** Using email and optional future extension of user ID
* **Service:** Via \_service\_id to link the booking with a particular service

This ensures that only relevant stakeholders have access to sensitive data, maintaining security and privacy.

**7.6 Admin Management Interface**

Admins can manage all service orders via the backend:

* View and edit order metadata
* Filter by order status
* Manually update status or reassign vendors/services
* Use the built-in post list table for streamlined access

Custom meta boxes are added in the order editor to simplify viewing and editing of stored metadata.

**7.7 Frontend Vendor Order Dashboard**

On the vendor dashboard (Orders tab), vendors can:

* View a categorized list of orders by status (Pending, Approved, Completed, Canceled)
* Change order status using a dropdown with SweetAlert confirmation
* View full customer details via a modal popup (phone, address, notes)
* Get real-time updates via refresh or soft reload after status change

This allows vendors to manage all incoming service requests without needing backend access.

**7.8 Security & Validation**

All order submissions and updates follow strict validation:

* Nonce protection on form submission and status changes
* Role-based access restrictions
* Sanitization and validation of all user inputs
* Error handling and feedback using AJAX and server-side logic

### 9. ****MODULE 9 :**** Booking and Order Workflow

### ****8.1 Overview****

The booking and order system forms the core transaction mechanism of the platform, seamlessly bridging the gap between customers and vendors. The workflow was engineered to ensure intuitive usability, accurate data capture, real-time feedback, and a fully trackable lifecycle from booking submission to order completion.

It involves both frontend and backend components, AJAX-powered form handling, and integration with the order management infrastructure built using custom post types.

### ****8.2 Booking Form on Service Single Page****

Each service has its own dedicated single page that contains detailed service information and a **“Book This Service”** button. Once clicked, a dynamic form toggles into view that allows customers to submit a service booking request.

The form includes the following fields:

* Full Name
* Email Address
* Phone Number
* Street Address
* Preferred Booking Date
* Additional Message

If the user is logged in as a **customer**, the form auto-populates using profile data stored in the user meta fields, improving user experience and ensuring data consistency.

All form fields are validated client-side and revalidated on the server to prevent unauthorized access or malformed data.

### ****8.3 AJAX-Based Submission Handling****

To enhance performance and avoid full page reloads, the form uses fetch() to submit data via AJAX to admin-ajax.php with a dedicated action hook (servicehub\_mvm\_book\_service). The submission is processed server-side with the following steps:

1. **Nonce Verification:** To prevent CSRF attacks.
2. **Input Sanitization:** All fields are sanitized using WordPress functions like sanitize\_text\_field() and sanitize\_email().
3. **Meta Preparation:** Data is structured and prepared for insertion into a custom post type.
4. **Order Creation:** A new post of type service\_orders is created with all submitted data stored as meta fields.

If successful, the form is hidden and replaced with a confirmation alert, giving users instant visual feedback.

### ****8.4 Order Generation and Linking****

Once submitted, the booking creates an order post (service\_orders) with the following associations:

* **Linked Service:** Via \_service\_id meta
* **Linked Vendor:** Via \_vendor\_id
* **Customer Details:** Captured via custom fields
* **Order Status:** Defaulted to pending

This order is now visible to the vendor from their dashboard, where they can review it, approve, complete, or cancel it.

### ****8.5 Notifications and Acknowledgements****

Upon successful booking:

* The **customer** receives a confirmation email with booking details.
* The **vendor** receives an email alert about the new order.
* The **admin** is also notified for monitoring and auditing purposes.

All emails are sent using wp\_mail() and configured via SMTP for improved deliverability.

The system uses dynamic templates with personalized content to ensure that each party receives the correct information in a branded, readable format.

### ****8.6 Order Status Lifecycle****

The order lifecycle is managed using the \_order\_status meta key. It supports the following states:

* **Pending**: Initial status after booking
* **Approved**: When vendor accepts the order
* **Completed**: After service is successfully delivered
* **Canceled**: If the vendor or customer cancels the booking

Vendors can manage order status from their dashboard. Changes are confirmed with modals (via SweetAlert), and customers are instantly notified via email.

### ****8.7 Role-Based Access Control****

Only vendors can view and act upon orders related to their services. Customers can only view their own bookings from the customer dashboard. Admins have full visibility and can modify or delete any order.

Backend editing is also supported for admins via custom meta boxes in the order editor, allowing status updates, field corrections, and manual entries.

### ****8.8 Error Handling & Validation****

All submission and workflow steps are covered with fallback mechanisms:

* Users cannot book without being logged in.
* Mandatory fields are enforced with both client- and server-side validation.
* Booking throttling is possible via limits like “Max Orders per Day” (from service settings).

This ensures integrity, reliability, and security in the booking pipeline.

### 10. ****MODULE 10 :**** Booking and Order Workflow

### ****9.1 Overview****

A robust email and notification system is vital for maintaining communication between the platform’s core stakeholders — Admins, Vendors, and Customers. The platform’s architecture integrates a modular and event-driven notification framework that ensures real-time alerts, confirmations, and updates are delivered during critical interactions such as bookings, order updates, approvals, and status changes.

The system leverages server-side mailing protocols and dynamic email templating to ensure reliability, accuracy, and personalization.

### ****9.2 SMTP Integration for Reliable Delivery****

To enhance email deliverability and avoid issues with native PHP mail functions, the platform is integrated with **Google SMTP (smtp.gmail.com)**. This configuration ensures emails are sent securely and reliably via authenticated SMTP credentials.

* **Protocol:** SMTP (Simple Mail Transfer Protocol)
* **Host:** smtp.gmail.com
* **Authentication:** OAuth/Password-based (secured)
* **Port:** 587 (TLS enabled)

All outgoing emails are processed through this secure SMTP relay, ensuring that messages reach their recipients without being flagged as spam.

### ****9.3 Trigger-Based Notification Flow****

The email system is designed to react to specific user actions and system events. When a customer submits a service booking, three separate emails are generated. One is sent to the customer as a confirmation with all relevant booking details. Another is sent to the vendor, alerting them of a new order with the service and customer information. A third is optionally sent to the site administrator to keep them informed of activity on the platform.

Similarly, when a vendor updates the status of an order (e.g., from "pending" to "approved" or "completed"), the system automatically sends an email notification to the customer, updating them about the change in real-time. This helps customers stay informed and builds trust in the service process.

Additionally, when a new vendor registers, the administrator receives an approval request email. After reviewing the application, the admin can approve or reject the vendor from the backend. The system then sends a corresponding email to the vendor—either congratulating them on being approved and inviting them to access their dashboard, or notifying them of rejection with a polite message.

In payment-related cases, transactional summaries and alerts are sent to vendors and admins, indicating successful bookings, payment confirmations, and order completions. These updates are all automated and dynamically populated using real-time data.

### ****9.4 Email Templating and Dynamic Content****

Emails are dynamically generated using HTML templates that include:

* **Personalized greetings** (e.g., “Hi John”)
* **Service and order details** (e.g., service name, date, customer/vendor info)
* **Call-to-action links** (e.g., view order, visit dashboard)
* **Styling** consistent with the platform’s branding

For example, when a booking is submitted:

* The **customer** receives a professional-looking order confirmation
* The **vendor** gets an alert with booking details and a link to their dashboard
* The **admin** is looped in for oversight

Each message is composed using PHP concatenation and sent via the wp\_mail() function with appropriate headers to ensure HTML rendering.

php

CopyEdit

$headers = ['Content-Type: text/html; charset=UTF-8'];

This ensures emails render correctly across various email clients.

### ****9.5 Centralized Email Dispatch Logic****

To maintain scalability and consistency, a centralized email dispatch layer is abstracted in the backend. Each function responsible for an event delegates the composition and sending of emails to a dedicated method, improving code readability and maintainability.

Additionally, fail-safes are implemented to handle mail delivery issues — fallback logging is included to notify the admin in case of errors using error\_log() and conditional success checks with is\_wp\_error().

### ****9.6 Data Security and Privacy****

To ensure data integrity in the email system:

* All personal data in emails (name, phone, address) is sanitized before output.
* Emails never expose sensitive data like passwords or internal IDs.
* Recipients are validated using filter\_var($email, FILTER\_VALIDATE\_EMAIL) to avoid delivery to incorrect addresses.

This safeguards the platform against information leaks and ensures GDPR-friendly practices.

### ****9.7 Extensibility and Future Enhancements****

The current implementation supports all essential touchpoints of user interaction. However, the email framework is modular and designed for future enhancements, such as:

* Admin-configurable templates
* In-platform notification center (in-app alerts)
* Email queueing with retry mechanisms
* Webhooks for external integrations (e.g., CRMs or analytics platforms)

### 11. ****MODULE 11 :**** Booking and Order Workflow

The platform includes a secure and streamlined transaction system powered by **Stripe**, a leading payment gateway trusted globally for handling online payments. This integration allows customers to complete service bookings with real-time payments, while vendors receive a record of confirmed transactions directly within their dashboard.

### ****10.1 Payment Integration with Stripe****

Stripe is integrated using its secure API to handle the entire payment workflow. Upon submitting the service booking form, the system securely collects the payment information and sends it to Stripe’s servers for processing. The system ensures PCI compliance by never storing sensitive card details on the server. Instead, Stripe handles tokenization and encryption behind the scenes.

### ****10.2 Transaction Records and Logging****

Each successful payment automatically generates a corresponding service order in the backend and stores transaction metadata (such as amount, timestamp, and Stripe payment ID). These transactions are visible to both the **admin** (via the WordPress backend) and the **vendor** (under the “Transactions” tab in their dashboard), ensuring transparency and easy tracking of revenue.

### ****10.3 Secure Payment Flow****

All communication with Stripe is conducted over HTTPS, and responses are verified to prevent tampering. The use of **webhooks** can be introduced in future iterations to automate payment status updates or refunds. The transaction logic is built to fail gracefully in case of payment errors, displaying user-friendly messages and avoiding partial or duplicate bookings.

### 12. ****MODULE 12 :**** Database Integration & Data Structure

The backend of the system is tightly coupled with a relational database architecture powered by **MySQL**, accessed and managed via **phpMyAdmin** during development. Rather than introducing a separate schema, the plugin makes efficient use of the existing database structure provided by the CMS it integrates with. This approach ensures compatibility, maintainability, and performance — especially when scaling the platform or integrating third-party solutions.

### ****11.1 Use of Default Database Tables****

The system stores all plugin-related data within the default database tables of the CMS, eliminating the need to manually manage additional custom tables. The most commonly used tables include:

* **wp\_users**: Stores all registered users (administrators, vendors, and customers). Each user has a unique ID, login credentials, and email.
* **wp\_usermeta**: Extensively used to store custom fields such as:
  + Vendor details (business name, service location, years in business, social links, portfolio uploads)
  + Customer details (phone number, address, gender)
  + Custom approval statuses (e.g., \_vendor\_approval\_status)
  + User roles and profile metadata
* **wp\_posts**: Serves as the primary repository for both service and service\_orders custom post types.
  + Services are stored with their titles, content, thumbnails, and status (publish, draft, etc.)
  + Orders are created programmatically upon successful booking and saved as individual post entries with associated meta fields.
* **wp\_postmeta**: Every service and order has a variety of metadata:
  + Services: \_service\_price, \_service\_schedule, \_service\_duration, \_service\_status, \_service\_gallery, etc.
  + Orders: \_customer\_email, \_preferred\_date, \_order\_notes, \_order\_status, and more.
  + Meta queries are structured and optimized for performance through indexed lookups when filters or queries are run on the archive pages.
* **wp\_terms, wp\_term\_taxonomy, wp\_term\_relationships**: These tables manage all taxonomy-related data.
  + Service Type and Service Location are custom taxonomies used to categorize services.
  + Each taxonomy is tied to services via term relationships, supporting filter-based navigation in the service archive.
* **wp\_options**: Utilized for global plugin settings, configurations, email templates, and stored Stripe keys or API toggles. This central configuration storage makes the system adaptable for different deployments.
* **wp\_comments and wp\_commentmeta**: These can be extended in the future for customer reviews or vendor feedback if the system evolves into a marketplace with rating systems.
* **wp\_users\_sessions, wp\_user\_level, wp\_capabilities**: These are indirectly used to manage role-based authentication and login sessions through the CMS’s native authentication system.

### ****11.2 Custom Post Type Architecture****

The system defines and registers two primary custom post types:

* **Services (service)**: Represents vendor offerings. Each service contains metadata such as price, type, gallery images, availability, and booking limitations. Services can be published, edited, or removed by the vendor through the vendor dashboard.
* **Orders (service\_orders)**: Each order is a post object created during a service booking. It holds details like the selected service, booking customer, scheduled date, and current status (pending, approved, completed, cancelled).

This model not only leverages the CMS's native CRUD operations but also allows flexibility in implementing hooks, filters, and meta queries for searching, filtering, and sorting content.

### ****11.3 Data Integrity & Structure****

Data integrity is preserved through rigorous input validation, use of nonces, and well-structured database operations. All plugin-generated content is linked through consistent use of post IDs, term relationships, and user IDs — making the data easy to query, report on, and export.

Each piece of data stored follows a naming convention (e.g., \_service\_ or \_order\_) which separates plugin-specific data from core fields, aiding maintainability and scalability.

### ****11.4 Scalability & Optimization****

The use of built-in CMS tables ensures the system scales efficiently. With proper indexing on meta keys (e.g., \_service\_price, \_order\_status), the plugin can support large-scale operations involving thousands of services, users, and bookings. Additionally:

* Transient caching can be implemented to reduce database load for frequently accessed data such as dashboard stats.
* The database design is modular and can support expansion like customer reviews, ratings, multi-vendor commission tracking, or third-party API integrations.

### ****11.5 Summary****

This database architecture follows a smart hybrid approach — combining the reliability of default CMS tables with the flexibility of custom post types and metadata. By avoiding custom table creation and relying on native architecture, the system ensures better future-proofing, plugin compatibility, and easier migration or backup processes.

### 13. ****MODULE 13 :**** Security Implementation

Security stands as a cornerstone of this web-based service marketplace solution. From user input handling to data transmission and access control, every aspect of the platform has been carefully developed with robust security strategies to protect sensitive data, ensure authorized access, and defend against common web vulnerabilities.

**12.1 Input Validation and Sanitization**

One of the core layers of security in this system lies in its meticulous **input validation and sanitization**:

* All user-submitted data—whether during registration, login, profile updates, or service creation—is sanitized using server-side functions that strip malicious or malformed input.
* Email addresses are validated using format-checking routines to ensure authenticity.
* Text fields are passed through escaping and stripping mechanisms before being stored or displayed, helping to prevent **Cross-Site Scripting (XSS)** and **SQL Injection** attacks.
* Numeric and URL fields are type-cast and filtered to eliminate unsafe characters and potential payloads.

This approach ensures that only clean, expected data enters the system and reaches the database.

**12.2 Role-Based Access Control (RBAC)**

The system employs a strict **Role-Based Access Control (RBAC)** model that dynamically governs user capabilities across the platform:

* Users are assigned specific roles (admin, vendor, or customer) that are tied to predefined sets of permissions.
* Access to sensitive areas—such as service management, order approvals, or transaction logs—is restricted based on role-specific checks.
* Conditional logic such as current\_user\_can() and role arrays are implemented at all permission checkpoints to prevent privilege escalation or unauthorized actions.
* Even at the UI level, role-based controls determine what buttons or forms are rendered to a user.

**12.3 Nonce Verification for CSRF Protection**

To mitigate **Cross-Site Request Forgery (CSRF)** attacks, **nonces (numbers used once)** are deployed throughout the system:

* Every critical form—be it login, registration, service creation, order management, or profile updates—includes a unique nonce token.
* On form submission, the system verifies this token before processing the request, ensuring that the request originated from a valid source and has not been hijacked or forged.

This token-based security mechanism is a proven standard in modern web applications and drastically reduces the attack surface for session-related threats.

**12.4 Password Protection & Authentication Security**

* All passwords are stored using **one-way hashing algorithms**, ensuring that even if the database is compromised, password data remains secure and unreadable.
* User authentication is managed via native hashing verification during login attempts, ensuring safe credential handling.
* Failed login attempts are handled gracefully, avoiding exposure of whether a user exists—thereby limiting **enumeration attacks**.

Furthermore, no sensitive session tokens or authentication data are ever exposed to the frontend or included in URLs.

**12.5 File Upload & Media Security**

The system restricts media uploads to authenticated users only and enforces file type validations and maximum file size limits. Uploaded files are stored in protected server directories, and paths are filtered to avoid direct exposure or access to sensitive uploads.

Future implementations can extend this security with:

* MIME-type verification
* Virus/malware scanning on upload
* Secure storage on external object storage (e.g., S3-compatible services)

**12.6 Plugin Isolation & Hardening**

To enhance modular security, the system follows **plugin isolation principles**:

* All custom business logic is contained within a single plugin structure, separated by folders and namespaces.
* Data access functions are encapsulated and reused to avoid accidental exposure or logic duplication.
* Security-related hooks and filters are registered on system-level actions like init, admin\_post, or AJAX calls, ensuring controlled execution environments.

**12.7 Server & Hosting Layer Security**

While primarily developed and tested on a local server stack (e.g., XAMPP, Laragon), the architecture is deployment-ready for secure environments:

* The system supports **HTTPS** encryption via TLS for secure data transmission.
* SMTP authentication is used for email delivery, avoiding reliance on untrusted mail agents.
* Error reporting is disabled in production environments to prevent data leakage through debug logs.

**12.8 Future Enhancements for Security**

To further bolster the platform's resilience against advanced threats, the following enhancements are proposed:

* **2-Factor Authentication (2FA)** for vendors and admins
* **Rate limiting** or CAPTCHA on login pages
* **Audit logs** to track account activity
* **Automatic logout/inactivity timers**
* **Encrypted API keys** and external webhook verification for payment gateways

**12.9 Summary**

The security implementation in this system follows the best practices of secure software engineering, with layered protections across authentication, data input, access control, and session management. By combining server-side validations, token-based verification, role-based access, and careful user input handling, the platform achieves a robust level of defense suitable for commercial deployment.

### 14. ****MODULE 14 :**** Plugin Architecture & File Structure

To maintain scalability, modularity, and clarity in development, the system is built using a well-organized plugin-based architecture. This approach encapsulates all core business logic, custom features, and user interfaces inside a self-contained plugin, making the system highly portable, maintainable, and independent of external themes or extensions.

### ****13.1 Plugin Architecture Philosophy****

The plugin was designed to function as a **standalone system module**, offering complete service marketplace functionality without relying on third-party extensions. This architectural style not only ensures consistency but also provides full control over:

* Custom dashboards for different user roles
* Service and order workflows
* Email notifications and transactions
* Database integrations and AJAX functionalities

The design promotes separation of concerns, ensuring each component (authentication, form handlers, templates, etc.) remains isolated and reusable.

***Chapter 6: Testing***

Testing is a crucial phase in the software development lifecycle, particularly in web-based platforms that involve complex user interactions, data storage, and real-time operations. The purpose of testing in this project was to ensure the stability, reliability, and security of the Service Marketplace platform and to validate its performance under different user roles and actions. This chapter provides a detailed explanation of how testing was conducted during and after the development of the system.

Testing was carried out in multiple phases, starting from the initial development phase with unit testing, followed by integration and functional testing. Each module—whether it be a user-facing feature such as booking or a backend process such as user approval—was put through various test scenarios to check expected behavior, handle exceptions, and confirm that unauthorized access or misuse was effectively prevented.

The platform was tested across different browsers (Chrome, Firefox, and Edge) and on varying screen resolutions to ensure cross-browser and responsive compatibility. Additionally, form validation, error handling, and real-time data submission through AJAX were tested thoroughly to prevent any broken flows or data loss.

## **6.1 Unit Testing**

Unit testing was performed on an individual module level to ensure that each function, form handler, and data operation worked as intended in isolation. This included both frontend and backend components of the system.

**Key areas tested during unit testing included:**

* **Custom Roles and Permissions:** Each role (admin, vendor, and customer) was tested to ensure they only had access to their respective dashboards and operations. For instance, a customer should not be able to view the vendor dashboard or create a service.
* **Form Validation:** Vendor and customer login and registration forms were tested to ensure that incorrect inputs were handled gracefully. Mismatched passwords, empty required fields, and invalid emails were flagged with proper feedback.
* **Custom Post Types:** The service and order post types were validated to ensure their custom meta fields were stored and retrieved correctly. This involved checking how data like price, availability, and booking notes were saved to the database.
* **AJAX Handlers:** AJAX-based booking submission was tested using mock data to ensure it handled both success and failure cases correctly. Validation errors were returned as JSON, and success messages were displayed upon order creation.
* **Security Validations:** Unit tests ensured nonces were verified before any form submission and that unauthenticated users couldn’t perform restricted actions such as creating services or viewing dashboard content.
* **File Uploads:** Portfolio and service gallery image uploads were tested to ensure only valid file types and sizes were accepted.

## **6.2 Functional Testing**

Once the individual components were verified, they were integrated into full user workflows for functional testing. This ensured that each role could complete its tasks from end to end, without breaking the flow.

### ****Vendor Functional Testing****

* Registering as a vendor.
* Logging in after approval.
* Accessing vendor dashboard securely.
* Creating, editing, and deleting services.
* Viewing and updating order status.
* Connecting with Stripe.
* Uploading portfolios and profile updates.
* Receiving booking emails from customers.

### ****Customer Functional Testing****

* Creating a customer account.
* Logging in and accessing the dashboard.
* Browsing services via the archive and filter system.
* Booking a service and confirming success message.
* Viewing booking history on the orders tab.

### ****Admin Functional Testing****

* Logging into the default backend.
* Viewing new vendor registrations.
* Approving and rejecting vendor applications.
* Viewing total vendors, orders, customers, and services via dashboard widgets.
* Managing all services and orders from the admin panel.

Each of these user flows was simulated multiple times using different inputs to ensure robustness and proper session handling.

### ****6.2.1 Testing Requirements****

To ensure the Service Marketplace platform met its intended functional and non-functional requirements, a set of targeted testing requirements were established. These requirements were based on the system specifications, user stories, and core use cases defined during the initial planning and requirement gathering phases.

The following outlines the testing criteria followed:

#### **A. Functional Testing Requirements**

These tests focused on verifying that each component of the system performed its designated function as expected:

* **User Authentication:** Ensure that login and registration work independently for vendors and customers, and only valid credentials are allowed access.
* **Role-based Access:** Confirm that users can only access functionalities assigned to their roles (e.g., customers cannot access vendor dashboards).
* **Service Management:** Verify that vendors can successfully add, update, delete, and view their services from the dashboard.
* **Order Lifecycle:** Test the full flow from service booking (customer) to order approval/completion (vendor).
* **Dashboard Functionality:** Ensure each tab in the vendor and customer dashboards (Profile, Orders, Services, Transactions) works correctly and displays accurate data.
* **Admin Panel Controls:** Test admin features like vendor approval, dashboard widgets, and content moderation.

#### **B. Security Testing Requirements**

* **CSRF Protection:** Validate all form submissions and AJAX requests using nonces to ensure only authenticated requests are processed.
* **Data Sanitization:** Test for input sanitization across all forms to prevent SQL injection, XSS, and other malicious injections.
* **Unauthorized Access Prevention:** Attempt access to unauthorized pages as different roles and verify redirection or denial with proper messaging.
* **Email Verification:** Simulate spoofed or forged email headers to test for vulnerabilities in the SMTP mail system.

#### **C. Usability and UI Testing**

* **Responsive Design:** Verify dashboard and service listings behave consistently across various screen sizes (desktop, tablet, mobile).
* **Navigation Flow:** Ensure intuitive user flow across login → dashboard → booking → logout with no broken pages or dead ends.
* **Feedback Messages:** Confirm that success and error alerts (e.g., booking confirmations, invalid login) are clear and informative.
* **Loading Speed:** Pages, especially dashboard panels and service archives, should load within a reasonable time even with high post volume.

## **6.3 Performance and Usability Checks**

Beyond core functionality, basic performance and usability checks were also conducted to ensure smooth user experience:

* **Page Load Time:** Ensured that service archive, dashboard pages, and forms load within acceptable time, even with multiple posts.
* **Responsiveness:** Tested on desktop, tablet, and mobile viewports to verify layout consistency.
* **Email Delivery:** Simulated booking and approval scenarios to verify SMTP email delivery to vendors, customers, and admin.
* **Data Persistence:** Ensured that no user data was lost upon form reloads, navigation, or session expiry.
* **Error Handling:** Verified that appropriate messages were shown for invalid actions, e.g., unauthorized access, duplicate emails, failed logins.

## **6.4 Summary**

The testing strategy adopted in this project allowed each component to be thoroughly examined at every stage of the development. Thanks to layered testing—starting with unit testing and progressing to functional workflows—critical bugs were detected early, preventing last-minute issues. By testing with various input values, invalid states, and user roles, the system was validated for real-world scenarios and edge cases.

This meticulous testing phase contributed significantly to the system’s robustness, ensuring that users (vendors, customers, and admins) have a secure, error-free, and smooth experience throughout the platform.

***Chapter 7: Conclusions & Future Work***

The Service Marketplace project has successfully delivered a comprehensive, modular, and scalable platform for managing service-oriented interactions between vendors, customers, and administrators. By incorporating custom user roles, tailored dashboards, booking workflows, and integrated payment systems, the platform demonstrates a robust real-world solution. Special care was taken to ensure secure user interactions, data validation, and smooth navigation throughout the user experience.

The system’s design allows for high flexibility and extendability, meaning it can evolve with growing user needs or adapt to different commercial use cases such as home services, repair providers, freelance platforms, etc. The implementation choices, especially custom post types and a dynamic frontend interface, made the platform fully functional while remaining independent from any specific CMS — though currently built to integrate seamlessly with WordPress.

Despite its current capabilities, there are several areas for enhancement that could enrich functionality, performance, and user engagement in the future. Below are the ideas explored in detail:

### ****Idea 1: Real-Time Chat System****

**Future Scope:**  
Integrating a secure real-time messaging system within the dashboard for communication between vendors and customers.

**Benefits:**

* Enables faster query resolution.
* Reduces the likelihood of miscommunication regarding service expectations.
* Promotes transparency and builds trust between both parties.
* Can be extended into a dispute-resolution channel with admin oversight.

**Implementation Suggestions:**

* Use tools like WebSockets, Firebase, or AJAX long-polling.
* Allow file attachments or image sharing in messages.
* Include message timestamps and read receipts.

### ****Idea 2: Advanced Filtering and Sorting****

**Future Scope:**  
Introduce multi-layered filtering mechanisms on the service archive page.

**Benefits:**

* Allows customers to quickly discover services based on granular preferences.
* Improves user engagement and reduces bounce rate.

**Proposed Filters:**

* Service Category, Location, Vendor Ratings.
* Real-time price range sliders.
* Date-based availability filters.

**Possible Implementation:**

* AJAX-based filter reloads for performance.
* Frontend libraries like Select2, jQuery UI sliders, etc.

### ****Idea 3: Review & Rating System****

**Future Scope:**  
Implement a review and rating mechanism that appears publicly on vendor profiles and service pages.

**Benefits:**

* Enhances credibility for both vendors and customers.
* Helps new users make informed decisions.
* Can serve as a metric for vendor ranking or promotions.

**Structure:**

* Star rating out of 5.
* Optional written reviews.
* Admin moderation before publishing.

### ****Idea 4: Mobile App Development****

**Future Scope:**  
Develop native mobile applications for both Android and iOS that mirror dashboard functionality.

**Benefits:**

* Offers on-the-go access to vendors and customers.
* Enables push notifications for new bookings, messages, or payment status.
* Access to phone features like GPS, camera (for uploading portfolio), and direct calling.

**Implementation Suggestion:**

* Use Flutter or React Native for cross-platform compatibility.
* Integrate with existing REST APIs or GraphQL.

### ****Idea 5: AI-Based Service Recommendations****

**Future Scope:**  
Deploy a simple AI recommendation engine that suggests services based on user interests and booking history.

**Benefits:**

* Personalizes the browsing experience.
* Increases user engagement and service bookings.

**Examples:**

* “Services you may like”
* “Popular in your area”
* “Recommended by others who booked X”

**Implementation:**

* Use tag similarity, user behavior tracking, or collaborative filtering algorithms.

### ****Idea 6: Vendor Analytics Dashboard****

**Future Scope:**  
Provide vendors with a visual analytics panel in their dashboard to view trends and data insights.

**Features to Include:**

* Total bookings and revenue over time.
* Peak booking times/days.
* Customer feedback and ratings summary.
* Top performing services.

**Benefits:**

* Helps vendors optimize their services.
* Promotes data-driven decisions for pricing, marketing, and availability.

### ****Idea 7: Booking Calendar with Availability Management****

**Future Scope:**  
Allow vendors to visually manage availability using a calendar-based interface.

**Features:**

* Drag-and-drop time slots.
* Disable dates or set service limits per day.
* Sync with external calendars (Google Calendar, Outlook).

### ****Idea 8: Subscription Model for Vendors****

**Future Scope:**  
Introduce tier-based subscription plans for vendors with benefits like featured listing, more visibility, or extra services.

**Benefits:**

* Adds a monetization layer to the platform.
* Encourages vendors to upgrade for better reach.

**Examples:**

* Basic (Free): Limited services.
* Premium: Featured on homepage, extra analytics.
* Enterprise: Full access + priority support.

### ****Idea 9: Admin Reporting & Exporting Tools****

**Future Scope:**  
Equip the admin dashboard with detailed reports and data export capabilities (CSV, PDF).

**Features:**

* Daily/Weekly/Monthly reports.
* Filter by date, status, or user role.
* Export orders, payments, vendor performance, etc.

### ****Idea 10: Enhanced Email Template Editor****

**Future Scope:**  
Add a dynamic email template system where admin can configure the design and content of system emails.

**Benefits:**

* Personalize communication.
* Use branding and tone aligned with the business.

**Tools:**

* WYSIWYG editor with placeholder tags.
* Toggle specific notifications on/off.

These ideas offer a roadmap for transforming the current version of the Service Marketplace into a highly professional, intelligent, and scalable product. The foundational architecture already supports modular expansion, meaning that most of these features can be implemented incrementally without affecting existing functionality.