SE Project 3

fARMBnbSe

Functional Requirements

User Management Subsystem

- **User Registration:** Supports secure user registration, validating and storing user data such as usernames, passwords, and emails.
- **User Authentication:** Ensures secure login for users using password verification.

Farm Management Subsystem

- **Farm Listing and Details:** Provides comprehensive listings and detailed descriptions of farms to assist users in making informed decisions.
- **Farm Booking:** Manages farm bookings with integrated availability checks and pricing calculations.

Payment Subsystem

- **Payment Processing:** Handles secure transactions using multiple payment methods, including card payments, UPI, and digital wallets.

Subscription Subsystem

- **Subscription Management:** Allows users to subscribe to and unsubscribe from farm updates, managing their notification preferences effectively.
- **Notification Dispatch:** Sends timely notifications to subscribers about updates and events related to their subscribed farms.

These functional requirements focus on the critical functions that each subsystem must perform to ensure the system is efficient, secure, and user-friendly.

Non-Functional Requirements

1. Scalability

- **Description:** The system should be able to scale horizontally to handle varying loads. We should be able to support peak loads, especially during high-demand periods like holiday seasons.

2. Availability and Reliability

- **Description:** High availability and reliability are critical. Each module/part of our system should be capable of functioning independently, reducing the risk of system-wide failures.

3. Maintainability

- **Description:** The system should be easy to update and maintain without significant downtime. The various parts of our systems should be loosely coupled and highly cohesive, enabling teams to update them independently and deploy new features or fixes rapidly.

4. Data Management

- **Description:** There should be data consistency and we should manage the data coherently. We do not want to create a situation where various parts of the system deal with modifying the data in an inconsistent way.

Subsystem Overview

User Management Subsystem

Role: The User Management subsystem is responsible for managing all user-related operations such as registration, login, profile management, and user listing. This subsystem ensures secure user authentication and the integrity of user data management within the farm management system.

Functionality:

1. **User Registration**
   - **Endpoint:** `/user/register`
   - **Method:** `POST`
   - **Purpose:** Allows new users to register by providing their username, password, and email. The system validates the data (ensuring the username is alphanumeric, the email is valid, age is valid), hashes the password for security, and stores the new user in the database.
2. **User Login**
   - **Endpoint:** `/user/login`
   - **Method:** `POST`
   - **Purpose:** Handles user login by verifying email and password credentials against stored data. It uses bcrypt for secure password verification.
3. **User Profile Management**
   - **Endpoint:** `/user/profile`
   - **Method:** `GET`
   - **Purpose:** Allows users to retrieve their profile information based on their user ID. The endpoint ensures sensitive information like passwords are not exposed.
4. **List Users**
   - **Endpoint:** `/user/list`
   - **Method:** `GET`
   - **Purpose:** Provides a list of all users, excluding sensitive information such as passwords. This is particularly useful for administrative purposes or user directory features.

This subsystem leverages MongoDB for data storage, with Flask as the web framework, and implements essential security measures like password hashing and data validation to protect user data.

Subsystem Overview

Farm Management Subsystem

Role: The Farm Management subsystem is dedicated to managing all operations related to farms, such as listing available farms, providing detailed information about specific farms, handling farm bookings, and gathering user feedback on farms.

Functionality:

1. **List Farms**
   - **Endpoint:** `/farm/list`
   - **Method:** `GET`
   - **Purpose:** Retrieves a list of all farms, which typically includes details such as farm name, description, location, and availability. This helps users in making informed decisions about which farms to explore or book.
2. **Get Farm Details**
   - **Endpoint:** `/farm/<id>`
   - **Method:** `GET`
   - **Purpose:** Provides detailed information about a specific farm, including location, area, ownership, contact information, and type. This is crucial for users considering booking or visiting the farm.
3. **List User-specific Farms**
   - **Endpoint:** `/farm/listuserfarm`
   - **Method:** `GET`
   - **Purpose:** Displays farms associated with a specific user, either through previous bookings or ownership. This feature is valuable for users to manage and review their farm interactions.
4. **Book Farm**
   - **Endpoint:** `/farm/book`
   - **Method:** `POST`
   - **Purpose:** Enables users to book a farm for specific dates. The system ensures the farm is available for the requested dates and calculates the total price based on the duration and farm's rates.
5. **Check Availability**
   - **Endpoint:** `/farm/checkavailability`
   - **Method:** `POST`
   - **Purpose:** Checks the availability of a farm for specified dates before booking. This is essential to prevent double bookings and to ensure that users can plan their visits without conflicts.
6. **Rate Farm**
   - **Endpoint:** `/farm/rate`
   - **Method:** `POST`
   - **Purpose:** Allows users to provide ratings and feedback after visiting a farm. This feedback is useful for other users and helps farm owners improve their offerings.

Subsystem Overview

Payment Subsystem

Role: The Payment subsystem handles all financial transactions related to farm bookings within the platform. It supports various payment methods, including direct card payments, UPI (Unified Payments Interface), and wallet transactions, ensuring secure and efficient processing of payments.

Functionality:

1. **Process Payment and Book**
    - **Endpoint:** `/pay/book`
    - **Method:** `POST`
    - **Purpose:** Manages the payment processing using specified payment methods (card or UPI) as part of the farm booking process. It employs different payment strategies depending on the method chosen to ensure flexibility and security in transactions.
2. **Pay Using Wallet**
    - **Endpoint:** `/pay/wallet`
    - **Method:** `POST`
    - **Purpose:** Allows users to pay for farm bookings using funds available in their digital wallet. This endpoint checks if the user has sufficient balance and updates the wallet balance upon successful transaction.

This subsystem is integrated with MongoDB through a `PaymentDao`, which manages payment-related data operations. The system employs a strategy pattern for payment processing, enabling easy adaptation to various payment methods while maintaining a consistent approach to handling payments.

Subsystem Overview

Subscription Subsystem

Role: The Subscription subsystem manages all aspects of user subscriptions related to farms, enabling users to receive notifications about farm updates. It allows users to actively manage their subscriptions and receive customized alerts.

Functionality:

1. **Get Subscribers**
   - **Endpoint:** `/subscription/getsubscribers`
   - **Method:** `GET`
   - **Purpose:** Retrieves a list of subscribers for a specific farm. This is useful for farm owners or administrators to see who is interested in updates from their farms.
2. **Subscribe**
   - **Endpoint:** `/subscription/subscribe`
   - **Method:** `POST`
   - **Purpose:** Allows users to subscribe to updates from a specific farm, ensuring they receive notifications about important changes or availability.
3. **Unsubscribe**
   - **Endpoint:** `/subscription/unsubscribe`
   - **Method:** `POST`
   - **Purpose:** Permits users to unsubscribe from receiving updates about a farm, giving them control over the notifications they receive.
4. **Notify**
   - **Endpoint:** `/subscription/notify`
   - **Method:** `POST`
   - **Purpose:** Sends notifications to all subscribers of a specific farm. This feature is crucial for communication during events like booking availability, special offers, or changes in farm status.
5. **Get Subscriptions**
   - **Endpoint:** `/subscription/getsubscriptions`
   - **Method:** `GET`
   - **Purpose:** Retrieves all subscriptions associated with a specific user, helping users manage and review their subscribed farms.

IEEE 42010

Stakeholders Identification

1. **End Users (Farmers and Customers)**
    - **Concerns:** Usability, reliability, performance, and availability of the system. Access to real-time information and ease of managing bookings and payments.
2. **System Administrators**
    - **Concerns:** System maintenance, uptime, data backups, security, and the ability to monitor system performance effectively.
3. **Developers and Maintainers**
    - **Concerns:** System scalability, maintainability, understandability, and ease of implementing updates or fixes.

IEEE 42010

Viewpoints and Views

1. Operational Viewpoint (End Users)

- **Views:** User interface mockups.
- **Addressed Concerns:** Usability, reliability, and availability.

2. System Administration Viewpoint (System Administrators)

- **Views:** Infrastructure configuration diagrams, backup and recovery procedures, security protocol descriptions.
- **Addressed Concerns:** System maintenance, security, monitoring capabilities.

3. Development Viewpoint (Developers and Maintainers)

- **Views:** Software architecture diagrams (e.g., component, deployment diagrams), API documentation, code repository structures.
- **Addressed Concerns:** Scalability, maintainability, ease of development.
    -

Architecture Decision Records (ADRs)

ADR 1: Adoption of Microservices Architecture

Decision: Adopt a microservices architecture for the system.

Status: Accepted

Context: The system requires flexibility, scalability, and independent deployment capabilities to accommodate various functionalities such as user management, farm management, payment processing, and subscription services.

Considered Options:

1. **Monolithic Architecture:** A single unified code base and deployment unit.
2. **Microservices Architecture:** A suite of small, independently deployable services.

Decision: Opt for microservices architecture.

Rationale:

- **Scalability:** Each service can be scaled independently based on demand, which is crucial for handling varying loads efficiently.
- **Development Agility:** Allows multiple teams to work independently on different services, accelerating development cycles.
- **Technological Diversity:** Each microservice can utilize technology stacks that are best suited to its requirements.

Consequences:

- Requires robust network management.
- Increases complexity in handling inter-service communications and data consistency.

Architecture Decision Records (ADRs)

ADR 2: Choice of Database Technology

Decision: Use MongoDB as the database solution for all microservices that require NoSQL storage.

Status: Accepted

Context: The system requires a flexible schema to accommodate various data types and structures used by different services.

Considered Options:

1. **Relational Databases (e.g., MySQL, PostgreSQL):** Traditional SQL databases.
2. **NoSQL Databases (e.g., MongoDB, Cassandra):** Schema-less databases designed for high scalability.

Decision: Use MongoDB.

Rationale:

- **Schema Flexibility:** Supports the dynamic data models of the system.
- **Scalability:** Well-suited for horizontal scaling and handling large volumes of data.
- **Developer Familiarity:** Widely used and supported by the developers.

Consequences:

- Not suited for highly relational data models.
- Potential issues with data consistency and transactions.

Architecture Decision Records (ADRs)

ADR 3: Integration of Payment Gateways

Decision: Integrate multiple payment gateways using a strategy pattern.

Status: Accepted

Context: The system must support various payment methods, including credit/debit cards, UPI, and digital wallets.

Considered Options:

1. **Single Payment Gateway:** Restrict to one payment gateway.
2. **Multiple Payment Gateways:** Support various gateways using a flexible strategy pattern.

Decision: Use a strategy pattern to integrate multiple payment gateways.

Rationale:

- **Flexibility:** Allows adding or changing payment methods without affecting other system parts.
- **User Convenience:** Offers users multiple payment options, enhancing user experience.

Consequences:

- Increases complexity in managing multiple integrations.
- Requires careful security management across different payment platforms.

Architectural Tactics

1. Exception → Availability

- **Non-functional Requirement Addressed:** Availability and Reliability
- Implementing fault detection through exceptions ensures that potential failures are caught and managed promptly. This helps maintain the system's overall availability and reliability by allowing it to handle errors gracefully, potentially logging them for further analysis and automatically recovering or alerting system administrators as needed.

2. Ping/Echo → Availability

- **Non-functional Requirement Addressed:** Availability and Reliability
- The ping/echo mechanism implemented via the `/health` endpoint serves to monitor server health. By regularly accessing this endpoint, which returns an 'OK' response, system health and responsiveness are confirmed. This simple yet effective approach helps ensure high availability by enabling quick detection and resolution of server issues, thus minimizing downtime.

3. Maintain Data Confidentiality → Security

- **Non-functional Requirement Addressed:** Security
- User authentication and encrypting or hashing passwords before storage significantly enhances data confidentiality. This practice ensures that even if data storage is compromised, the actual passwords remain protected, thus maintaining the security of user accounts.

Architectural Tactics

4. Restrict Communication Path → Modifiability

- **Non-functional Requirement Addressed:** Maintainability, Security
- Restricting communication paths among components or services in a system can simplify the interactions between them, reducing the complexity of the system architecture. This modifiability improvement makes it easier to modify, replace, or update individual components without impacting others, enhancing overall system maintainability.
- By restricting communication paths, you can control which components communicate with each other, reducing the attack surface by limiting potential entry points for unauthorized access.

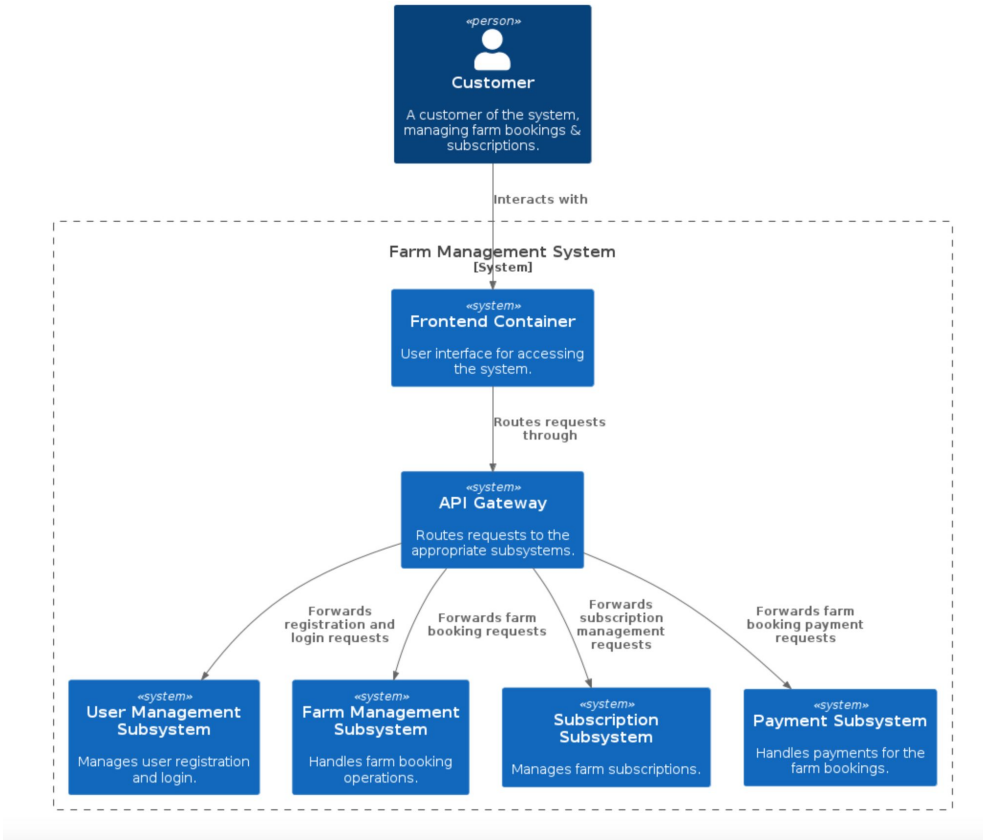
5. Use an intermediary → Modifiability

- **Non-functional Requirement Addressed:** Maintainability, Scalability
- Using intermediaries such as APIs or middleware facilitates loose coupling between system components. This architectural style allows individual components or services to be modified or replaced without affecting others, thus improving the system's adaptability and ease of updates.
- The intermediary allows scalability in terms of allowing the list of farms & users to be decoupled and allow for dynamic changes in the list.

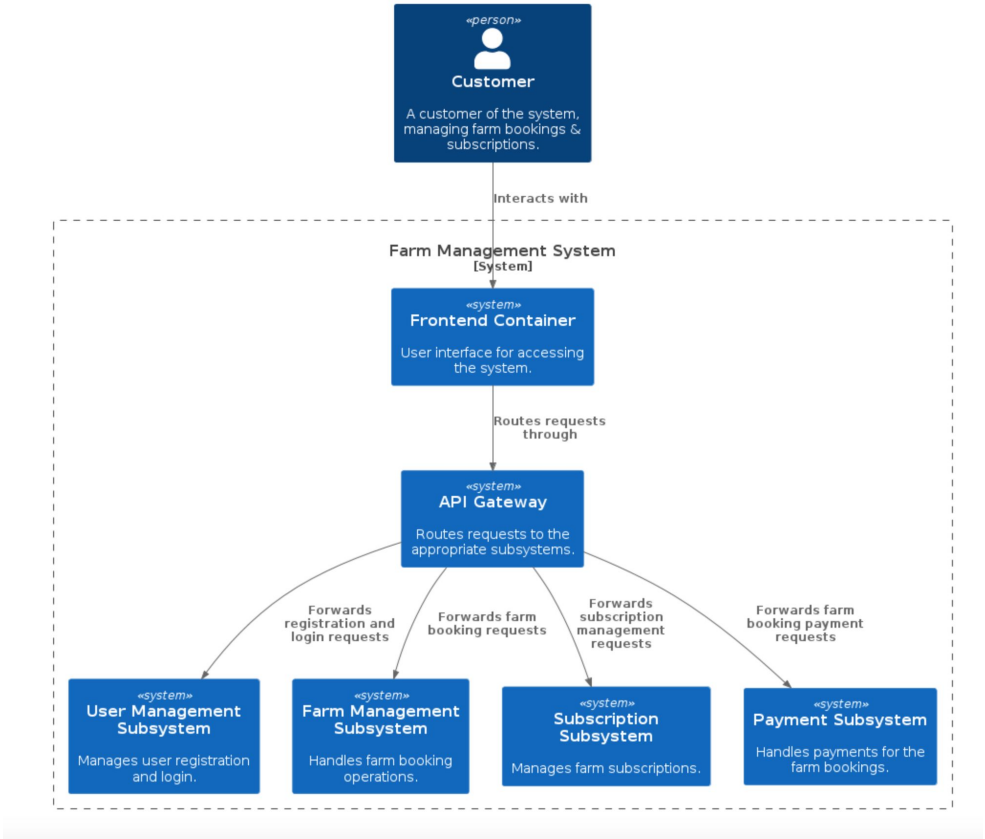
6. Separate UI from the rest of the system → Usability (Design Time) (Model View Controller)

- **Non-functional Requirement Addressed:** Usability, Performance
- Separating the user interface (UI) from the business logic and data model (using the MVC pattern) enhances usability by allowing developers to focus on user interface design independently of backend development. This separation also facilitates easier updates and maintenance of the UI without affecting the underlying business logic, leading to a more flexible and user-friendly system design.
- The separation allows for the potential optimization of the backend and frontend independently, potentially improving overall system performance by optimizing resource usage on both ends.

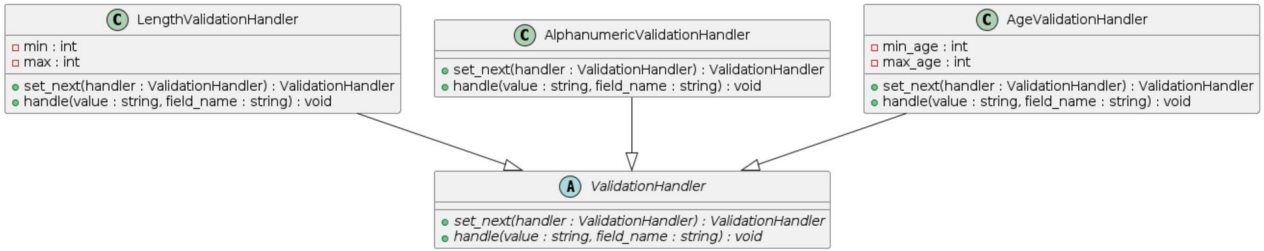
Context Diagram



Container Diagram



Design Pattern Example (Chain of Responsibility)




```
# Validation handlers setup
length_for_username = LengthValidationHandler(3, 50)
alphanumeric = AlphanumericValidationHandler()
length_for_password = LengthValidationHandler(8, 50)
length_for_email = LengthValidationHandler(3, 50)
length_for_phone = LengthValidationHandler(10, 10)
email_validator = EmailValidationHandler()
number_only_phone = PhoneNumberValidationHandler()
age_validator = AgeValidationHandler(18, 65)

# Setting up validation chains
length_for_username.set_next(alphanumeric)
length_for_password.set_next(None)
length_for_email.set_next(email_validator)
length_for_phone.set_next(number_only_phone)
```

Quantitative Analysis - Architecture

Concurrency/Throughput:

Microservices:

Throughput: 19.6876129907987 requests/second Total Time taken: 5.079335927963257 Average Response Time: 0.05079335927963257

Monolith

Throughput: 6.709276388363637 requests/second Total Time taken: 14.904736995697021 Average Response Time: 0.1490473699569702

Response Time

Microservices

Time taken: 23.4591281414032

Monolith

Time taken: 7.165441989898682

Monolithic vs Microservices

Monolithic Architecture:

Performance:

In our test scenario, the monolithic architecture demonstrated better response time for individual requests. This efficiency can be attributed to the tight integration of components within the monolith, resulting in minimal inter-component communication overhead.

Scalability:

However, when it comes to scalability, the monolithic architecture faces limitations. Scaling involves replicating the entire stack, which can be resource-intensive and costly. With our test environment, scaling beyond a certain point might become challenging, especially given the increasing demand for concurrent requests.

Microservices Architecture:

Performance:

Our test results show that the microservices architecture exhibited better throughput and average response time under concurrent requests. Despite slightly higher latency due to inter-service communication, the modular nature of microservices allows for better parallelism and concurrency handling, resulting in improved performance.

Scalability:

Microservices architecture excels in scalability. Each service can be independently deployed and scaled, enabling granular control over resource allocation. With our test scenario, we observed efficient scaling, where only the services experiencing high demand needed to be scaled out, optimizing resource utilization.

Monolithic vs Microservices

Monolithic Architecture:

Energy Efficiency:

From an energy efficiency perspective, the monolithic architecture may not be optimal. Since the entire application runs continuously, even during periods of low demand, there's a constant energy consumption. This lack of adaptability to workload fluctuations can lead to wasted energy.

Microservices Architecture:

Energy Efficiency:

One of the key advantages of microservices architecture is its energy efficiency. In our code, services are designed to spin up only when needed, reducing overall energy consumption during periods of low activity. This dynamic resource allocation ensures efficient energy usage, especially compared to the continuous operation of a monolithic architecture.

Trade-offs and Insights

1. **Complexity**: While microservices offer scalability and energy efficiency benefits, they introduce greater complexity in deployment, management, and monitoring compared to monolithic architectures. Teams must invest in robust infrastructure and tooling to handle this complexity effectively.
2. **Development and Testing**: Developing and testing microservices requires additional effort due to the distributed nature of the architecture. Ensuring consistency, reliability, and version compatibility across services can be challenging but is crucial for maintaining system integrity.
3. **Fault Isolation**: Microservices offer better fault isolation, but managing inter-service communication and maintaining data consistency across distributed systems require careful design and implementation.
4. **Resource Overhead**: Despite the scalability benefits, microservices architecture introduces overhead associated with managing multiple services. This overhead must be carefully balanced against the benefits gained from scalability and efficiency.

Contributions

Akshit - architecture analysis and setting up monolith & microservice architectures

Mukta - payment & subscription subsystem, frontend, microservices

Ronak - farm subsystem, booking frontend, architecture diagrams

Bala - user subsystem, subscription, architecture diagrams

Sriya - farm, subscription subsystem, frontend, microservices