Software Architectures

Assignment 2 : Smart Essential List Application

BITS Pilani

Harmalkar Rahul Rajan Sayali
ID No.: 2024TM93073

Objective

Objective: To gain experience in architecting real-life applications in diverse domains like Retail, Transportation, and Hospitality.

Examples: Inspired by real-world systems like Swiggy (food delivery), Uber (transportation), and IoT health monitoring.

Application Domain: Smart Essential List for groceries, food items, and travel expenses.

Purpose and Key Functional Requirements

Purpose :

The Smart Essential List App aims to provide users with an efficient platform to manage groceries, food items, and travel expenses. It simplifies the creation, organization, and sharing of lists, enhancing planning and purchasing experiences with features like smart suggestions and expense tracking.

Key Functional Requirements

User Account Management
Secure account creation and login with authentication options.

List Creation and Management
Create, edit, and delete lists; categorize items for organization.

Item Management
Add, edit, and remove items; track quantities and prices.

Real-Time Collaboration
Share lists with others; receive notifications for updates.

Smart Suggestions
Recommend items based on past usage for quick addition.

Expense Tracking
Log expenses and visualize spending trends.

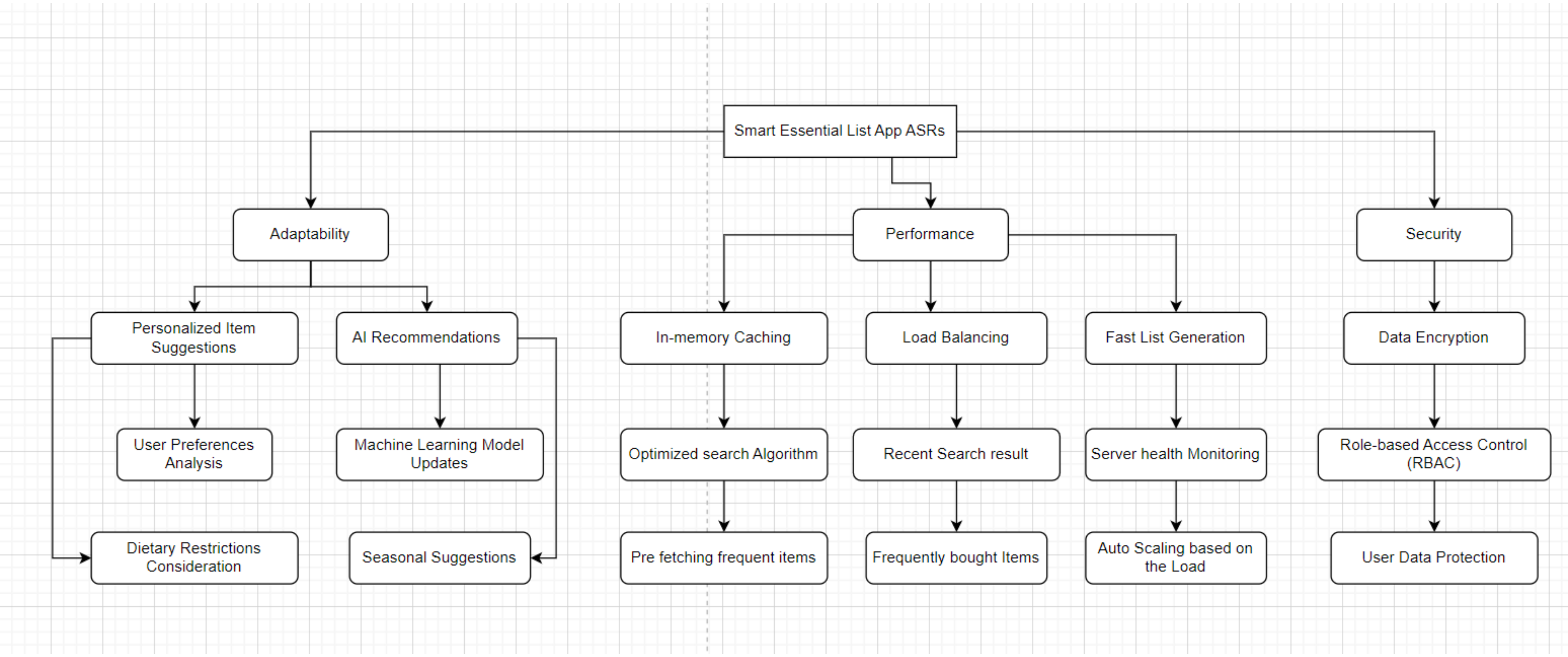
Notifications and Reminders
Set reminders for shopping trips and important items.

Data Syncing and Backup
Synchronize lists across devices and ensure data backup.

Search and Filter
Search and filter items for easy retrieval.

User-Friendly Interface
Intuitive design for easy navigation on all devices.

Architecturally Significant Requirements


Smart Essential List App ASRs
Adaptability
Performance
Security
Personalized Item Suggestions
AI Recommendations
In-memory Caching
Load Balancing
Fast List Generation
Data Encryption
User Preferences Analysis
Machine Learning Model Updates
Optimized search Algorithm
Recent Search result
Server health Monitoring
Role-based Access Control (RBAC)
Dietary Restrictions Consideration
Seasonal Suggestions
Pre fetching frequent items
Frequently bought Items
Auto Scaling based on the Load
User Data Protection

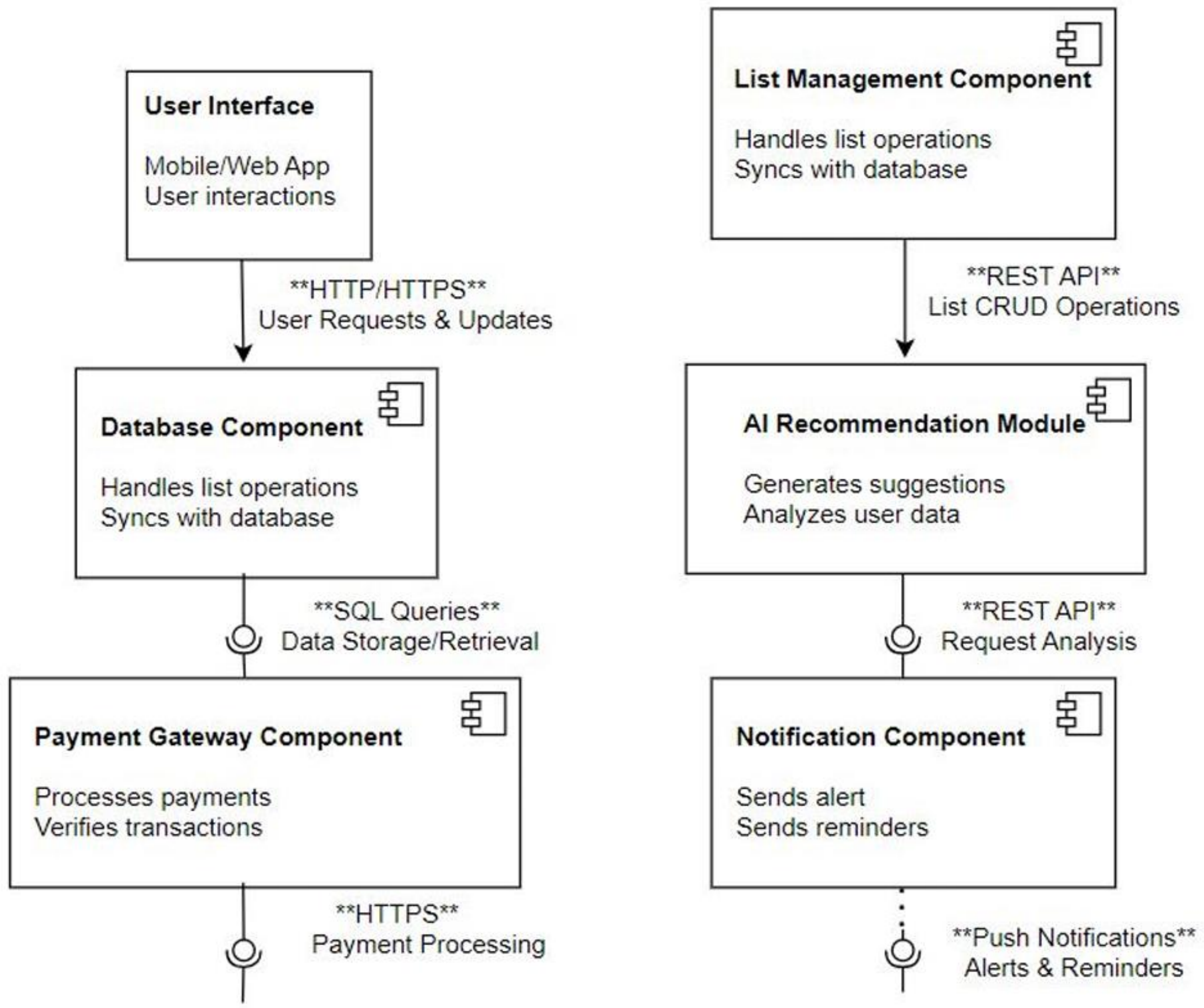
Tactics for Each ASR

1. Adaptability
Modular Design: Independent updates for features; reduces dependencies.
Config-Driven: Quick, centralized updates via external configurations.
Plug-in Architecture: Adds new features easily; keeps core functions separate.

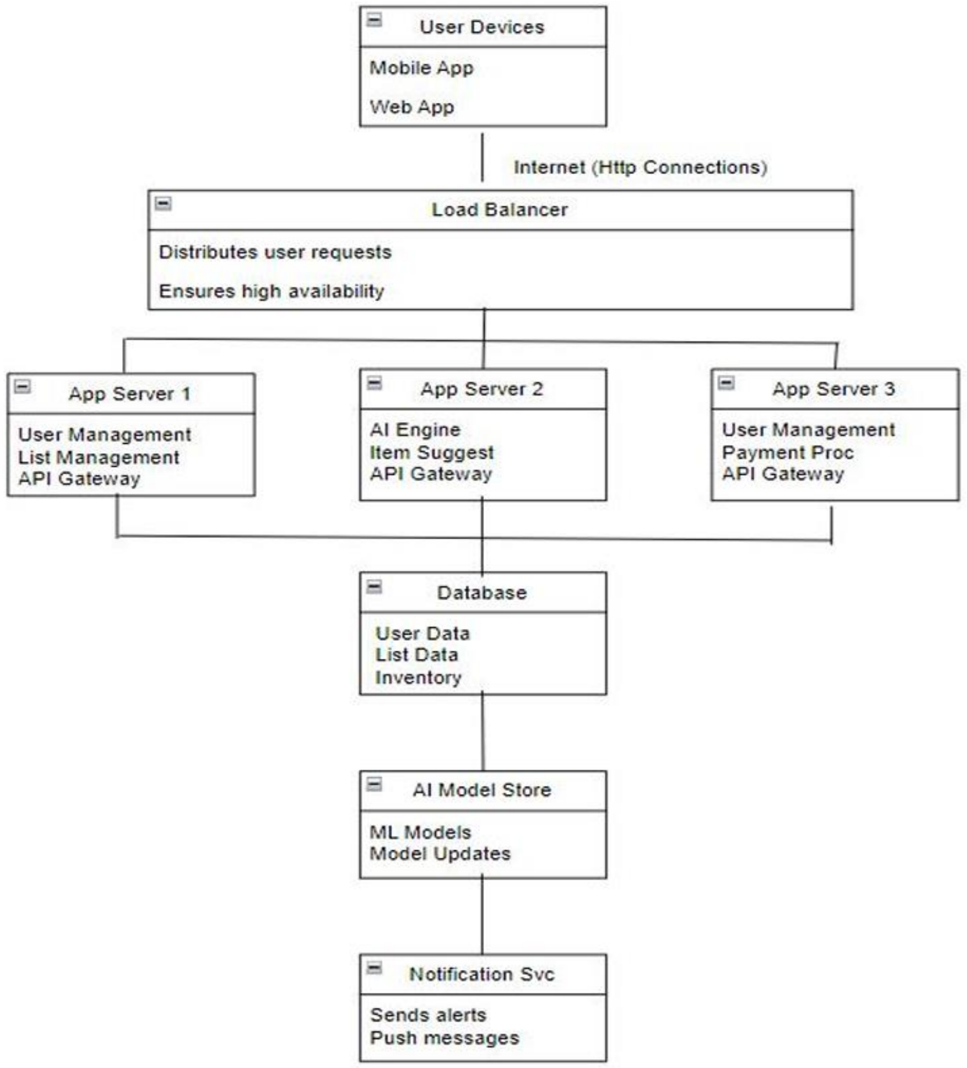
2. Performance
Caching: Speeds up access to frequent items; reduces database load.
Load Balancing: Distributes requests to avoid overload; ensures smooth performance.
Database Indexing: Fast retrieval for frequent queries; minimizes scan time.

3. Security
Auth & Authorization: MFA and role-based controls protect sensitive data.
Data Encryption: Encrypts data at rest and in transit for security.
Logging & Monitoring: Tracks activity, detects issues, and supports audits.

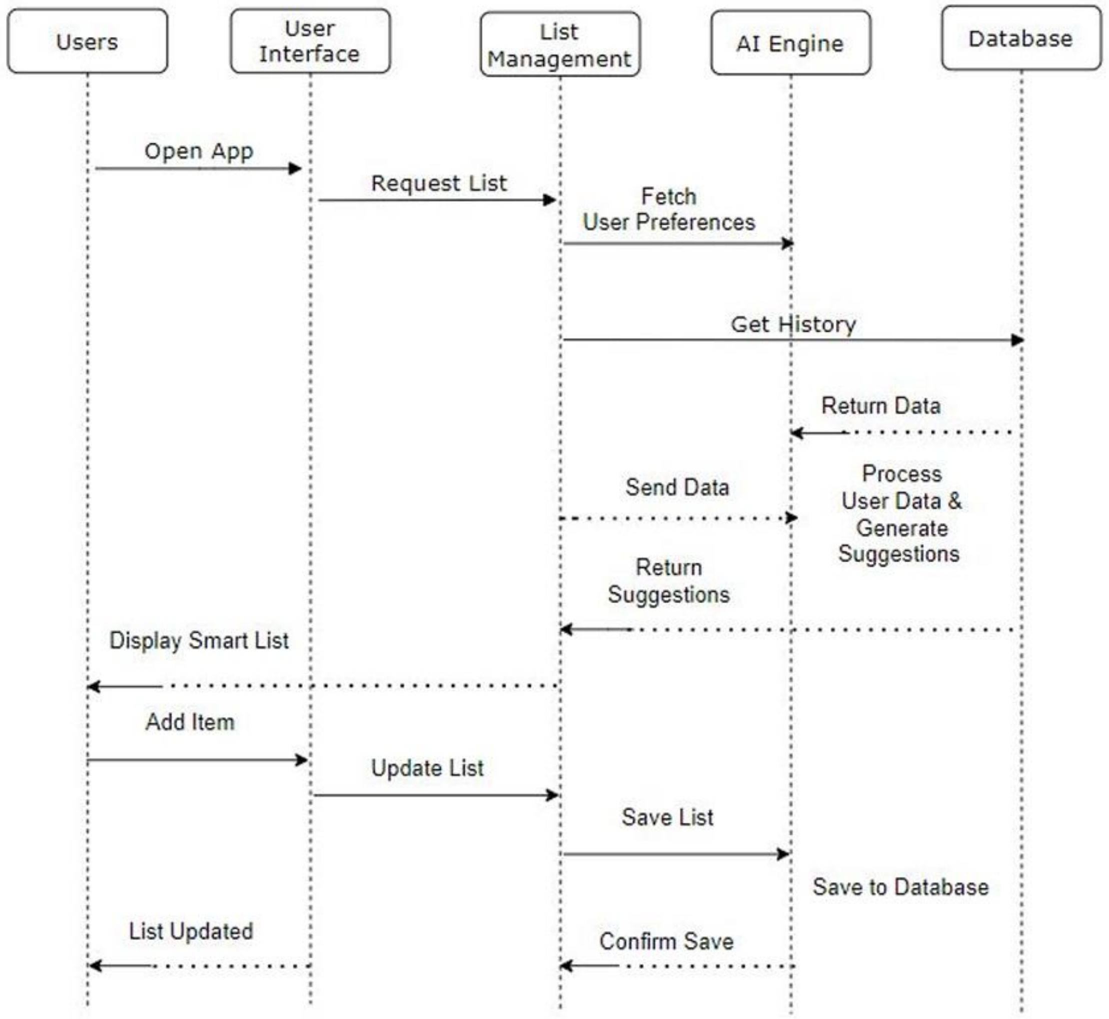
Component & Connection View



Deployment View



Sequence Diagram



Architecture Patterns

1. Microservices Pattern

- **Description:** Breaks down app into small, independent services (e.g., list management, user authentication).
- **Benefits:** Enables easy scaling, independent updates, and fault isolation.

2. Repository Pattern

- **Description:** Centralizes data access logic, separating it from business logic.
- **Benefits:** Simplifies data handling, enables data source flexibility, and improves testability.

3. Event-Driven Pattern

- **Description:** Uses events to communicate between services, especially for notifications or list updates.
- **Benefits:** Supports real-time updates, improves responsiveness, and scales well for high-frequency changes.

4. MVC (Model-View-Controller) Pattern

- **Description:** Separates app into three layers: data (Model), user interface (View), and business logic (Controller).
- **Benefits:** Enhances modularity, making it easier to update UI, business logic, or data handling independently.

5. Client-Server Pattern

- **Description:** Divides responsibilities between the client (app UI) and server (backend processing).
- **Benefits:** Ensures secure, centralized data management and enables lightweight client applications.

Key Learnings

1. Prioritizing Architecturally Significant Requirements (ASRs)

- ASRs (e.g., adaptability, performance, security) are critical for aligning the app's architecture with user needs and business goals.
- Identifying and addressing these requirements early creates a solid foundation for system reliability, scalability, and user satisfaction.

2. Selecting Effective Architectural Tactics

- Thoughtful selection of tactics (e.g., caching for performance, load balancing for scalability, encryption for security) enhances system quality and meets ASRs effectively.
- Each tactic directly contributes to the app's resilience, efficiency, and long-term maintainability, making it adaptable to future growth.

3. Leveraging Architecture Patterns for Modularity and Scalability

- Patterns like Microservices, Event-Driven, and Repository pattern simplify the architecture by isolating features and creating reusable components.
- This modularity not only improves scalability and fault tolerance but also makes future updates and feature additions more manageable and cost-effective.

Thank You