Revisions			
Version	**Description**	**Date**	**Person**
1.0	The document was created.	21.11.2023	Elif Beril Sayli Özde Uysal Annie Yang
1.1	The document was revised according to the Iteration 2 feedback.	02.12.2023	Elif Beril Sayli Özde Uysal Annie Yang
1.2	The document was updated for Iteration 3.	16.12.2023	Elif Beril Sayli Özde Uysal Annie Yang
1.3	Assumptions and dependencies were clarified according to Iteration 3 feedback.	23.12.2023	Elif Beril Sayli
1.4	Data Diagram was updated. Domain Model was updated. Logical View was updated.	01.01.2024	Özde Uysal Annie Yang Elif Beril Sayli

Samwise Service App
Architecture Notebook

1. Purpose

This document outlines the core goals, principles, and vital aspects influencing the design and development of the application. It covers essential architectural requirements, decisions, constraints, and justifications, emphasizing their impact on the system's structure and functionality. It also highlights critical dependencies, key design elements, and significant mechanisms crucial for shaping the Samwise App's overall architecture and guiding its implementation.

2. Architectural goals and philosophy

The philosophy of the Samwise Service App architecture is simplicity and understandability. The main goal behind that architecture is straightforward and intuitive design so architectural components are designed and logically organized and the structure of the system avoids unnecessary complexities. Also, it reduces the learning curve for developers and new team members who don't have enough experience in development. There is a minimum number of concepts, patterns, and abstractions to help developers understand. Also, clear code principles and concise documentation increase the understandability of code. Meaningful variable and function names, and structure the code in a way that aligns with common programming practices and well-organized code enhances readability. A straightforward and intuitive structure simplifies deployment workflows, reducing the chances of errors during this critical phase. Moreover, the simplicity facilitates the seamless integration of modern components with legacy systems. New team members can quickly understand the integration points, ensuring the preservation of data integrity and functionality during the adaptation process. Moreover, optimized code and efficient architectural decisions like layered architecture contribute to performance optimization, mitigating potential bottlenecks and ensuring a responsive user experience. By minimizing unnecessary complexities and providing clear documentation, the system becomes more maintainable over time. By isolating hardware dependencies, such as device-specific

interactions with web development, the application ensures consistent functionality across various devices and browsers. Moreover, Samwise incorporates mechanisms to handle unusual conditions, ensuring the application remains responsive even during peak usage via load balancing, caching strategies, optimized server-side processing, and automated backup and recovery processes to isolate potential failures. To provide data consistency and integrity, client-side data validation and server-side validation provide a reliable representation of data.

In terms of low-level design, the architecture adheres to SOLID Principles (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion). Each class and module has a single responsibility, is open for extensions but closed for modifications, follows the substitution principle, adheres to interface segregation, and utilizes dependency inversion to decouple high-level modules from low-level details. This ensures that each component is modular, maintainable, and can be extended without modifying existing code.

3. Assumptions and dependencies

Assumptions and dependencies that drive architectural decisions include:

3.1. Internet connection is stable for users.

3.2. Users have the most current versions of their respective browsers to ensure safety, security, and browser support.

3.3. Users must access the Samwise Service App either as a Service Provider or a Service Requester in order to offer or request services.

3.4 The code assumes that related modules are installed such as express for web server functionality and body-parser for form data parsing, are installed and properly configured for development. Also, the Universally Unique Identifiers (UUIDs) library for generating UUIDs is installed for development.

3.5 Deployment requirements which are specified at Project Plan Document will be satisfied.

3.6 It is assumed that third party libraries will ensure reliability.

3.7 It is assumed that cloud service will ensure reliability and security.

4. Architecturally significant requirements

For SamWise App, architecturally significant requirements are as heading at that reference document.

Requirement / Reference	References Heading	Reference
User Authentication and Account Management	2.1	SystemwiseReqSpect_v1.3
Service Scheduling and Provider Accountability	2.2	SystemwiseReqSpect_v1.3
Average Rating Calculation	2.3	SystemwiseReqSpect_v1.3
Payment Processing and Transaction Handling	2.4	SystemwiseReqSpect_v1.3
Customer Support	2.5	SystemwiseReqSpect_v1.3
Privacy and Security	2.6	SystemwiseReqSpect_v1.3
Data Backup and Recovery	2.7	SystemwiseReqSpect_v1.3
Usability	3.1	SystemwiseReqSpect_v1.3
Reliability	3.2	SystemwiseReqSpect_v1.3
Performance	3.3	SystemwiseReqSpect_v1.3

Supportability	3.4	SystemwiseReqSpect_v1.3
Premium User Discounts	5.4	SystemwiseReqSpect_v1.3
User Interaction and Feedback	5.5	SystemwiseReqSpect_v1.3
Service Provider Management and Verification	5.6	SystemwiseReqSpect_v1.3
Support and Maintenance	5.7	SystemwiseReqSpect_v1.3
System Constraints	6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7	SystemwiseReqSpect_v1.3
Licensing Requirements	7.1	SystemwiseReqSpect_v1.3
Legal, Copyright, and Other Notices	7.2	SystemwiseReqSpect_v1.3
Applicable Standards	7.3	SystemwiseReqSpect_v1.3
System Documentation	8.1, 8.2, 8.3, 8.4, 8.5	SystemwiseReqSpect_v1.3

5. Decisions, constraints, and justifications

Decisions	Justification	Reference
The system architecture benefits from cloud development to prioritize maintaining a high level of availability, ensuring a minimum uptime of 99.9%.	Although serverless computing allows developers to focus on writing code without worrying about the underlying infrastructure, during peak times, cloud development scales up resources such as computing power, storage, and bandwidth to meet demand without the need for significant upfront investments.	SystemwiseReqSpect_v1.2
The system requirements provide data integrity with monthly incremental data backup, data consistency via synchronous replication, and data recovery procedures. It ensures the reliability of data stored within the system. Therefore, data management is preceded by MySql.	MySQL adheres to ACID (Atomicity, Consistency, Isolation, Durability) properties, ensuring the reliability and integrity of transactions. This is vital for maintaining data consistency so MySQL stands out from others. Also, MySQL offers robust security features, including user authentication, encryption, and access control.	SystemwiseReqSpect_v1.2

The system should be capable of handling 100 concurrent users and requests, with a requirement for a minimum throughput of 100 requests per second during peak usage. Therefore, load balancing mechanisms will be used.	If one server in the pool fails or undergoes maintenance, the load balancer automatically redirects traffic to the remaining healthy servers.	SystemwiseReqSpect_v1.2
The server-side is designed with Node.js.	Node.js is an asynchronous, scalable, event-driven architecture and provides handling numerous concurrent connections efficiently. Node. js allows developers to write both the front-end and back-end of an application from a single codebase rather than React.	SystemwiseReqSpect_v1.2
The Samwise App adopts a hybrid approach, integrating layered architecture, event-driven architecture, and aspects of a monolithic approach.	Each architectural style offers distinct advantages; layered architecture provides modularity, event-driven architecture enhances responsiveness and scalability, while aspects of a monolithic approach ease centralized management.Therefore, hybrid approach is best for SamWise.	Samwise_Architecture_Notebook
The server-side is designed with Express.js.	Express.js leverages the efficient and high-performance nature of Node.js, making SamWise fast and responsive.	Samwise_Architecture_Notebook

6. Architectural Mechanisms

6.1. Modular Design:

The SamWise codebase exhibits a modular design by organizing functionality into separate components to increase maintainability and scalability.

6.2. Database Abstraction:

The use of MySQL and the abstraction of database interactions through the mysql2 library provide a clear separation between business logic layer and data layer and it provides better organization and maintainability of the code.

6.3. RESTful API for CRUD Operations:

The application follows RESTful principles for creating, reading, updating, and deleting (CRUD) data. The endpoints for user registration, login, and service management adhere to RESTful conventions, providing a standardized and predictable API.

6.4. Middleware for Form Data Parsing:

The use of the body-parser middleware allows the application to parse form data easily. This mechanism simplifies the handling of incoming data from HTML forms.

6.5. Asynchronous Database Queries:
The MySQL queries are executed asynchronously, preventing the application from being blocked during database interactions and it increases the responsiveness of the application.

6.6. Dynamic HTML Rendering:
HTML files are dynamically rendered and served based on user interactions.

6.7. Foreign Key Constraint (Database Relationship):
The use of a foreign key constraint in the services table, referencing the users table, establishes a relationship between the services and users entities. This enforces referential integrity in the database.

6.8. UUID Generation for Service Identifiers:
A UUID (Universally Unique Identifier) is generated for each service during creation. This ensures a globally unique identifier for each service, reducing the likelihood of collisions and enhancing data integrity.

6.9. Error Handling and Response Codes:
The application handles errors gracefully by providing meaningful error messages and appropriate HTTP status codes in the responses. This mechanism enhances the user experience and aids in debugging and increases the readability of code.

6.10. Static File Serving (CSS and Images):
The application serves static files such as CSS stylesheets and images using Express's static file serving middleware. This ensures efficient delivery of static assets to clients.

7. Key abstractions

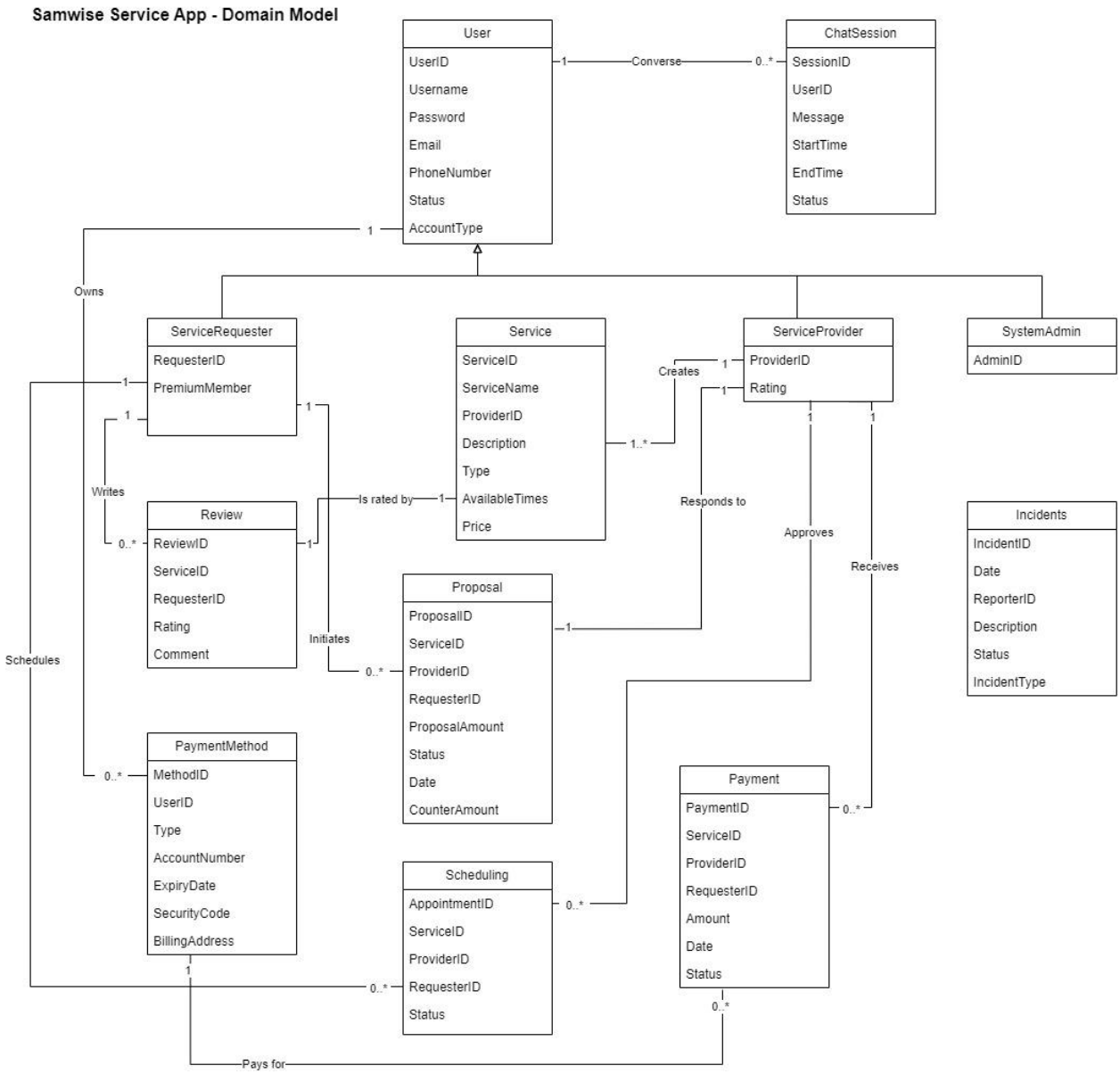
7.1. User: Represents registered individuals interacting with the application, including Service Requesters and Service Providers, and System Admins.

7.2. Service: Represents various home maintenance and care services offered on the platform.

7.3. Review: Represents the user ratings and reviews given to Service Providers and services.

7.4. Proposal: Represents a premium feature where Service Requesters can propose discounts on service prices, which can be accepted, rejected, or modified by Service Providers.

7.5. Domain model



8. Layers or architectural framework

Initially, the Samwise App was designed with layered architecture. On the other hand, the app is very appropriate for event-driven architecture and a monolithic approach. Therefore, the Samwise App has a hybrid approach as software architecture and each architectural style brings its own advantages.

8.1 Layered Architecture

The layered architecture which is demonstrated at ***Figure 1.1*** provides modularity in which each layer can be developed, tested, and maintained independently. Moreover, different layers can be scaled based on requirements. Also, changes in one layer do not necessarily impact other layers.

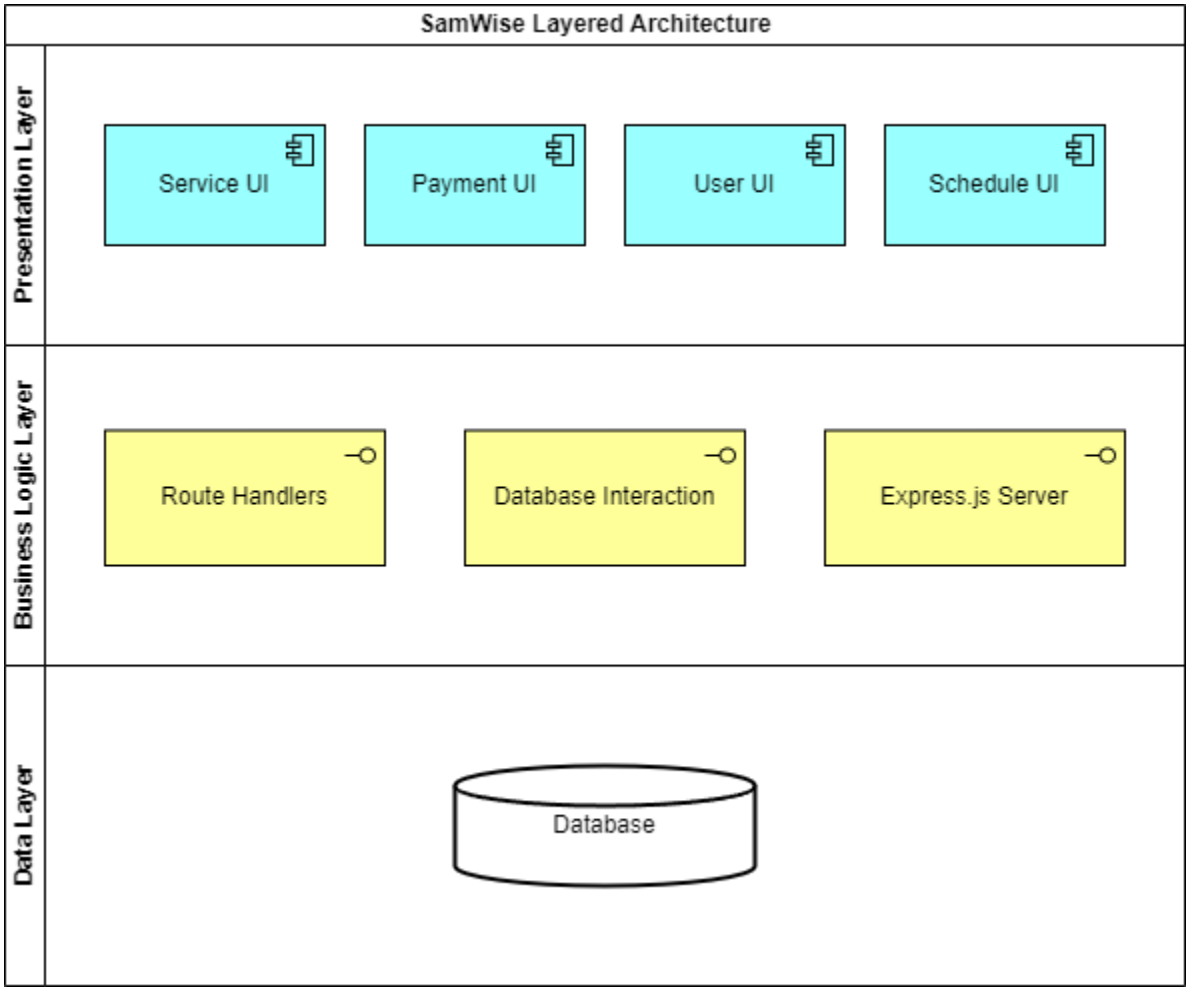


Figure 1.1 Layered Architecture Diagram

8.1.1. Presentation Layer:

- This layer handles user interaction and presentation of information for services, user-related processes, payment, and scheduling pages.
- The HTML files serve as templates for generating the presentation layer. They define the structure and layout of the user interface.
- CSS is responsible for the layout, colors, and overall styling of the user interface.
- The JavaScript functions embedded in the HTML file handle client-side interactions to dynamically modify the content of the web pages.

8.1.1.1 Service UI

- The Service UI is a component of the presentation layer responsible for providing users with a graphical interface to interact with the service-related functionalities.

8.1.1.2 Payment UI

- The Payment UI, situated within the presentation layer, facilitates users in managing financial transactions and payment-related activities.

8.1.1.3 User UI

- The User UI is an integral part of the presentation layer, serving as the interface for users to interact with their accounts and personal information like creating accounts and login.

8.1.1.4 Schedule UI

- Within the presentation layer, the Schedule UI is designed to assist users in managing schedules, appointments, or events within the system.

8.1.2. Business Logic Layer:

- This layer contains the business logic and application-specific functionalities.
- The part of the application layer, responsible for routing requests to the appropriate controllers.
- The Express.js server defines route handlers and handles HTTP requests.
- This layer provides database interactions with the various SQL queries in route handlers involving interactions with the MySQL database.

8.1.2.1 Route Handlers

- Route Handlers form a crucial part of the business logic layer, responsible for processing and responding to incoming requests from the presentation layer. These handlers interpret user actions and trigger the appropriate business logic.

8.1.2.2 Database Interaction

- The Database Interaction module in the business logic layer manages the communication between the application and the underlying database. It encapsulates the logic related to data retrieval, storage, and modification.

8.1.2.3 Express.js Server

- The Express.js Server, situated in the business logic layer, acts as the middleware that facilitates communication between the presentation layer and the underlying business logic. It handles incoming HTTP requests, routes them to the appropriate business logic components, and sends back the corresponding responses.

8.1.3. Data Layer:

- This layer manages data storage and retrieval at MySQL database

8.2 Event- driven Architecture

The structure of the implementation aligns well with the principles of event-driven architecture, especially in the context of web applications using Node.js and Express.js. There are some characteristics of code that align with event-driven architecture. Node.js is used to handle asynchronous operations using an event-driven, non-blocking I/O model. Therefore, database queries, file reading, and other potentially blocking operations are handled asynchronously. In an event-driven architecture, components communicate through events. Express.js framework relies on event emitters because the handling of HTTP requests and responses is event-driven, where routes are triggered by specific HTTP methods. The route handlers themselves can be seen as event handlers for specific HTTP events. Moreover, Express.js middleware functions allow you to execute code before and after the request is processed. Database queries using the mysql2 package are typically asynchronous and non-blocking. The queries are executed, and the results are handled through callback functions, adhering to the event-driven nature of Node.js. Redirects and responses are actions triggered in response to specific events, such as successful or unsuccessful user login or service creation.The JavaScript functions in the HTML file that handle user interactions, such as toggling between login and account creation sections, are also event-driven.

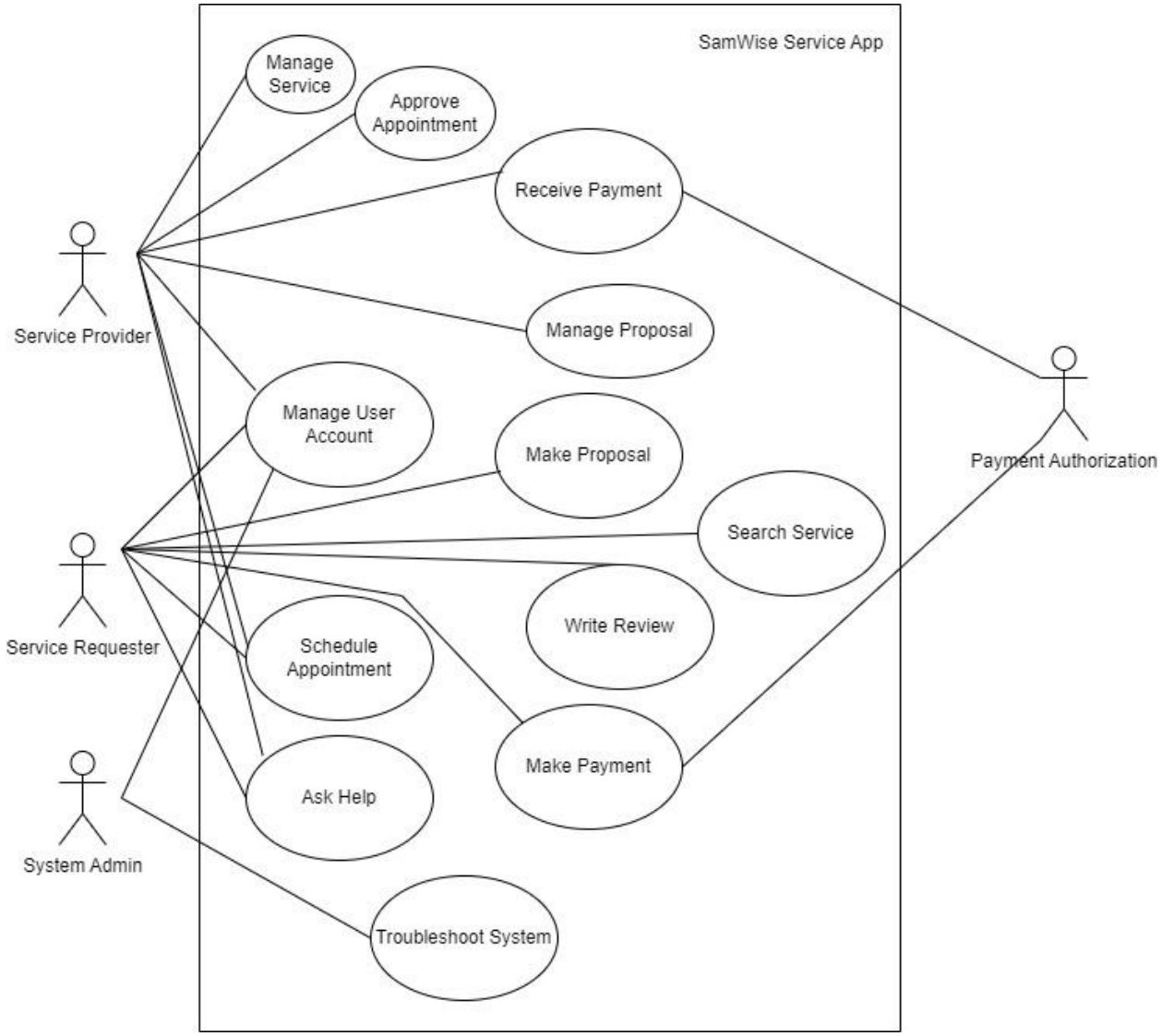
8.3 Monolithic Architecture

The deployment of monolithic architecture is simpler because the entire application is deployed as a single unit. Moreover, a monolithic architecture is often simpler to develop and maintain, making it suitable for smaller

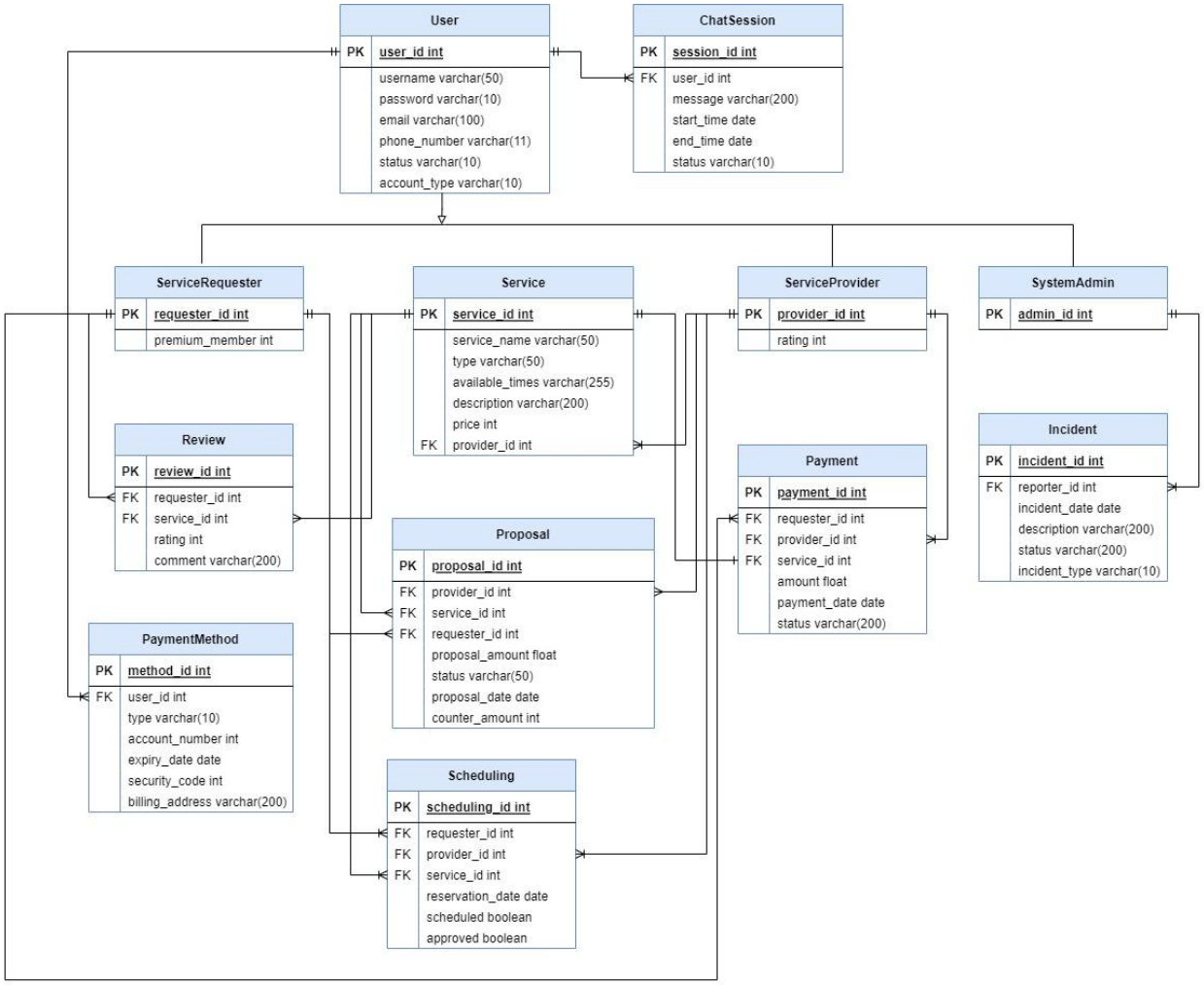
applications or projects with limited complexity. On the other hand, according to needs, as a future development, microservice architecture can be considered to obtain independent development and deployment, better fault isolation and increase technological diversity.

9. Architectural views

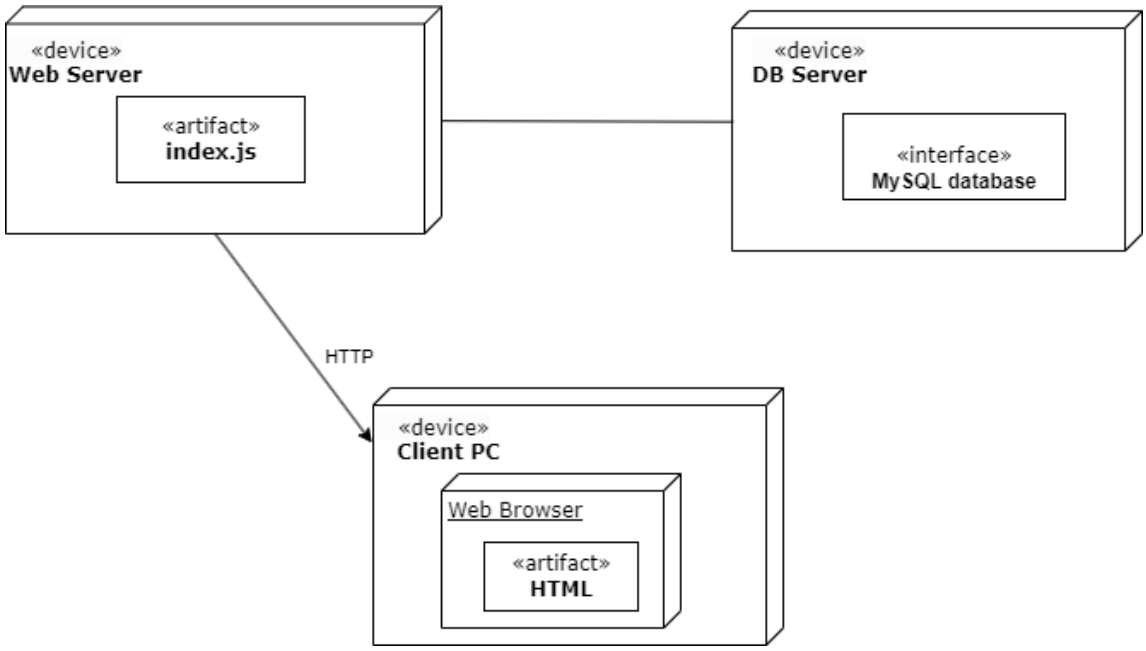
9.1 Use case



9.2 Data view


User
PK user_id int
username varchar(50)
password varchar(10)
email varchar(100)
phone_number varchar(11)
status varchar(10)
account_type varchar(10)
ChatSession
PK session_id int
FK user_id int
message varchar(200)
start_time date
end_time date
status varchar(10)
ServiceRequester
PK requester_id int
premium_member int
Service
PK service_id int
service_name varchar(50)
type varchar(50)
available_times varchar(255)
description varchar(200)
price int
FK provider_id int
ServiceProvider
PK provider_id int
rating int
SystemAdmin
PK admin_id int
Review
PK review_id int
FK requester_id int
FK service_id int
rating int
comment varchar(200)
Payment
PK payment_id int
FK requester_id int
FK provider_id int
FK service_id int
amount float
payment_date date
status varchar(200)
Incident
PK incident_id int
FK reporter_id int
incident_date date
description varchar(200)
status varchar(200)
incident_type varchar(10)
Proposal
PK proposal_id int
FK provider_id int
FK service_id int
FK requester_id int
proposal_amount float
status varchar(50)
proposal_date date
counter_amount int
PaymentMethod
PK method_id int
FK user_id int
type varchar(10)
account_number int
expiry_date date
security_code int
billing_address varchar(200)
Scheduling
PK scheduling_id int
FK requester_id int
FK provider_id int
FK service_id int
reservation_date date
scheduled boolean
approved boolean

9.3 Deployment View



9.4 Logical View