Samwise
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 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.

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. The development team is proficient in JavaScript, Node.js, and MySQL to complete the project plan in the respective iterations.

3.4. The system is compliant with data protection laws such as GDPR and KVKK such that user data is handled responsibly.

3.5 The assumption is that the MySQL database specified in the code is already set up, running, and accessible.

3.6 The development team should have Node.js and npm (Node Package Manager) for running JavaScript server-side code and managing dependencies.

3.7 The use of the Express framework as a web server is a critical dependency for the application's routing, middleware, and server functionality for development.

3.8 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.

4. Architecturally significant requirements

Requirement / Reference	Hybrid Architecture	Reference
Authentication and User Management	2.1	SystemwiseReqSpect_v1.2
Scheduling	2.2	SystemwiseReqSpect_v1.2
Average Rating Calculation	2.3	SystemwiseReqSpect_v1.2
Payment Processing	2.4	SystemwiseReqSpect_v1.2
Customer Support	2.5	SystemwiseReqSpect_v1.2
Privacy and Security	2.6	SystemwiseReqSpect_v1.2
Data Backup and Recovery	2.7	SystemwiseReqSpect_v1.2
Usability	3.1	SystemwiseReqSpect_v1.2
Reliability	3.2	SystemwiseReqSpect_v1.2
Performance	3.3	SystemwiseReqSpect_v1.2
Supportability	3.4	SystemwiseReqSpect_v1.2
User Authentication and Account Management (Business Rules)	5.1.1	SystemwiseReqSpect_v1.2
Scheduling and Provider Accountability (Business Rules)	5.2.1	SystemwiseReqSpect_v1.2
Payment Processing and Transaction Handling (Business Rules)	5.3.1	SystemwiseReqSpect_v1.2
Premium User Discounts (Business Rules)	5.4.1	SystemwiseReqSpect_v1.2
User Ratings and Reviews Transparency (Business Rules)	5.5.1	SystemwiseReqSpect_v1.2
Service Provider Qualification Verification (Business Rules)	5.6.1	SystemwiseReqSpect_v1.2
Regular System Updates and Bug Fixes (Support and Maintenance)	5.7.1	SystemwiseReqSpect_v1.2
System Constraints	6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7	SystemwiseReqSpect_v1.2

Licensing Requirements	7.1	SystemwiseReqSpect_v1.2
Legal, Copyright, and Other Notices	7.2	SystemwiseReqSpect_v1.2
Applicable Standards	7.3	SystemwiseReqSpect_v1.2
System Documentation	8.1, 8.2, 8.3, 8.4, 8.5	SystemwiseReqSpect_v1.2

5. Decisions, constraints, and justifications

Reference	Requirement	Decision	Constraints	Justifications
Systemwise ReqSpect_v 1.2	3.2.1. The Samwise Service App should maintain a high level of availability, with a minimum requirement of 99.9% uptime.	High Availability with 99.9% Uptime	The system architecture shall prioritize maintaining a high level of availability, ensuring a minimum uptime of 99.9%.	• *DO*: Design the system with redundancy and fault tolerance to prevent single points of failure. • *DO*: Implement load balancing and failover mechanisms to distribute traffic and ensure continuous operation. • *DON'T*: Avoid dependencies on individual components to prevent service disruptions in case of failures.
Systemwise ReqSpect_v 1.2	3.2.5. 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.	Data Integrity and Reliability Measures	The system shall implement monthly incremental data backups, synchronous replication for data consistency, and establish data recovery procedures.	• *DO*: Conduct monthly incremental data backups to prevent data loss and ensure the availability of historical data. • *DO*: Utilize synchronous replication to maintain consistent and up-to-date data across multiple locations, enhancing data reliability. • *DON'T*: Avoid asynchronous

				replication for critical data to prevent inconsistencies and maintain data reliability.
	3.3.2. 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.	Scalability for 100 Concurrent Users and 100 Requests per Second	The system shall be designed to handle a load of 100 concurrent users and maintain a minimum throughput of 100 requests per second during peak usage.	• *DO*: Employ scalable architecture, utilizing horizontal scaling to accommodate increasing user demands. • *DO*: Implement efficient caching mechanisms and optimize database queries to ensure optimal performance. • *DON'T*: Avoid bottlenecks in the system architecture that could limit scalability and throughput.
Samwise_Architecture_Notebook	-	The Samwise App adopts a hybrid approach, integrating layered architecture, event-driven architecture, and aspects of a monolithic approach.	Developers are tasked with ensuring seamless integration and compatibility between architectural styles while avoiding over-engineering.	• 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.

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 providers.

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. Time Slot: Represents a service provider’s available time to provide a service. After a service is scheduled by a service requester, the Time Slot becomes unavailable.

7.5. Proposal: Represents a premium feature where service requesters can propose discounts on service prices, which can be accepted, rejected, or modified by service providers.

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.

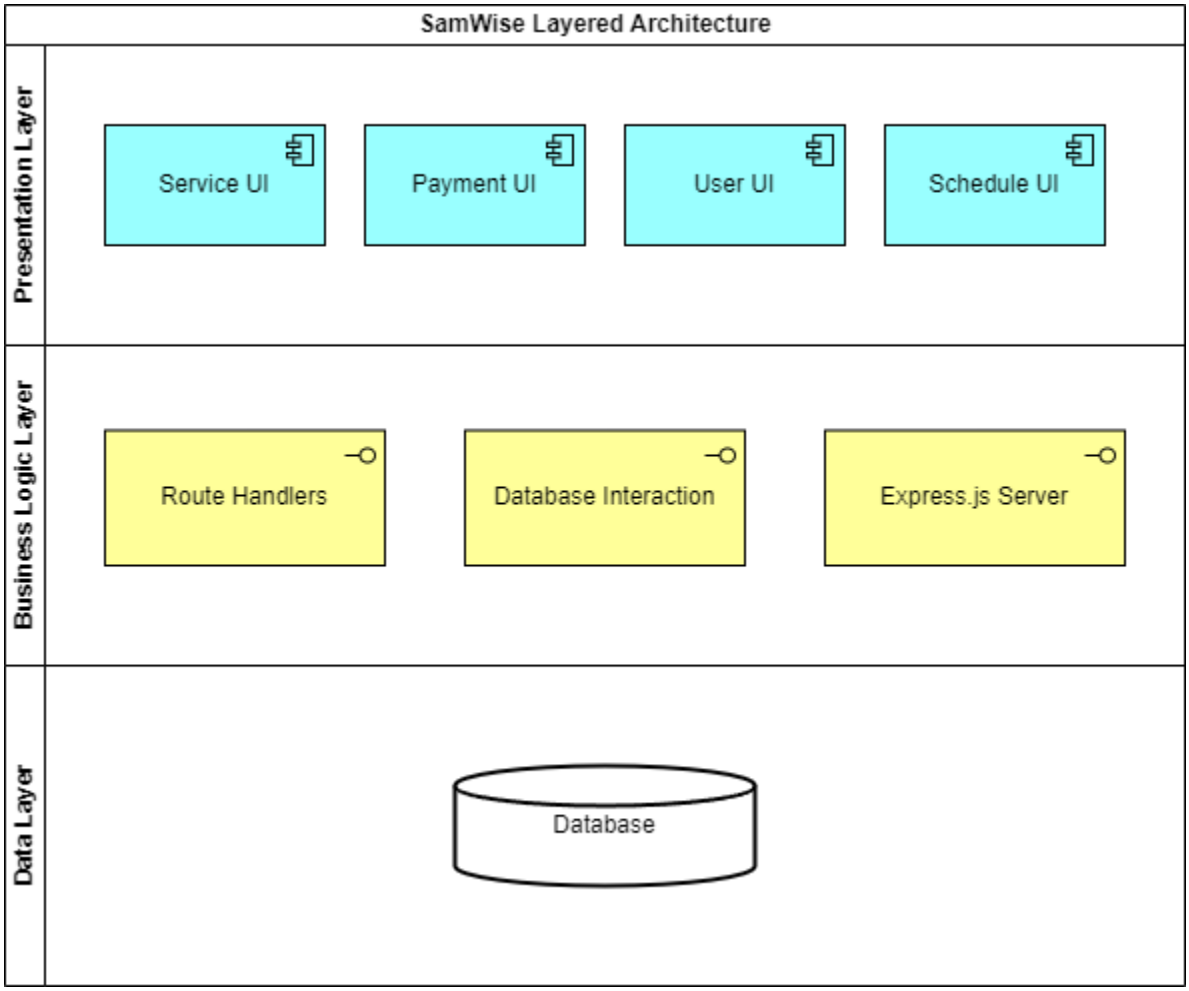


Figure 1.1 Layered Architecture Diagram

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.

8.1.2. 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.

8.1.3. Data Layer:

- This layer manages data storage and retrieval at MySQL database

8.2 Event- driven Architecture

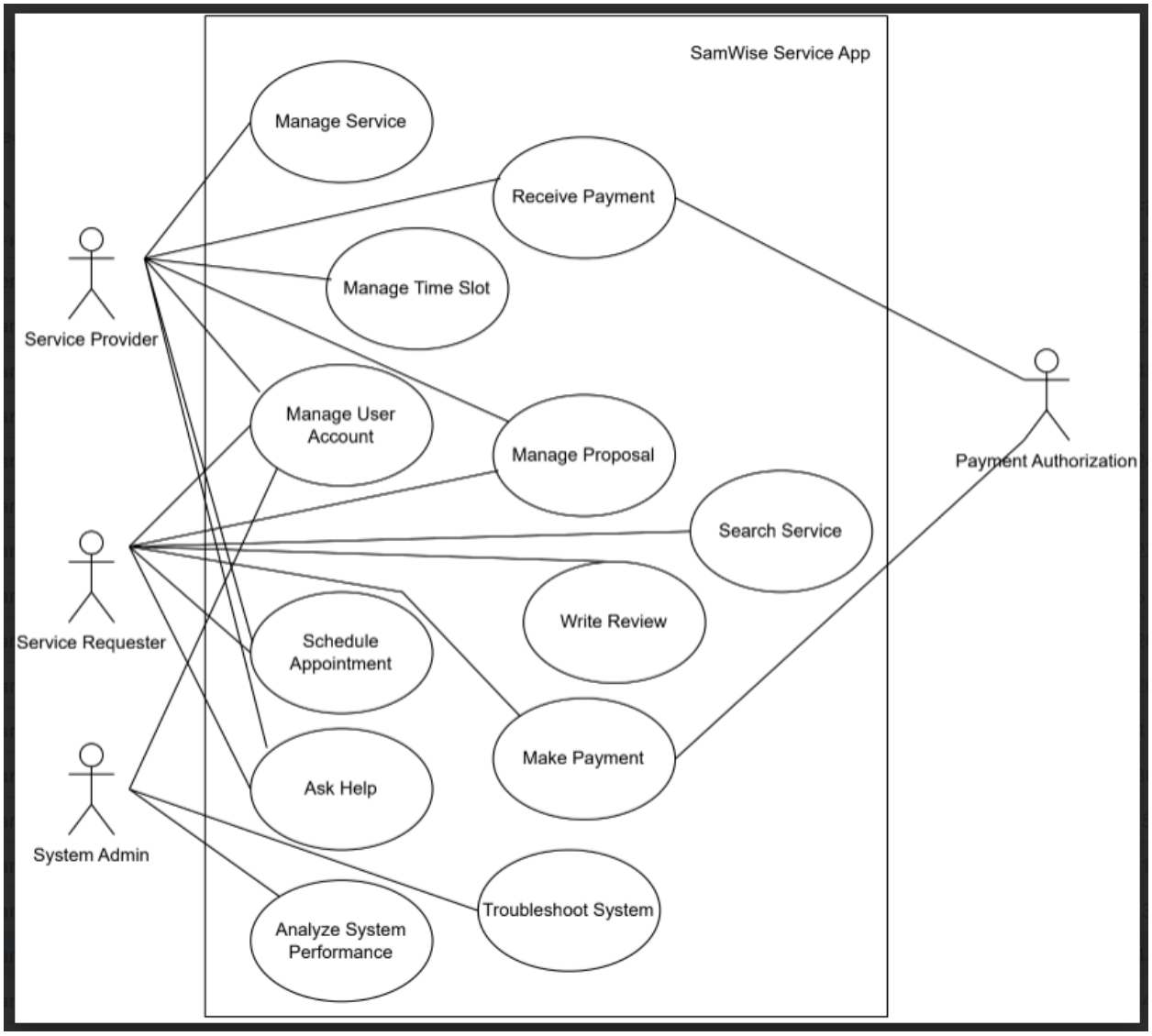
The structure of the implementation aligns well with the principles of event-driven architecture, especially in the context of web applications using Node.js and Express.js. There are some characteristics of code that align with event-driven architecture. Node.js is used to handle asynchronous operations 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

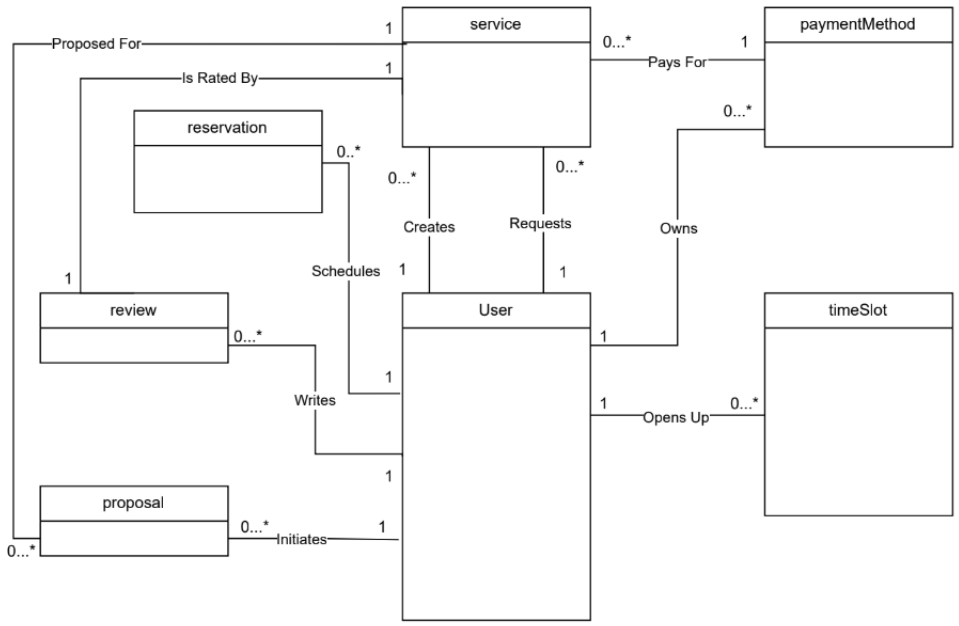
The deployment of monolithic architecture is simpler because the entire application is deployed as a single unit. Moreover, a monolithic architecture is often simpler to develop and maintain, making it suitable for smaller applications or projects with limited complexity. On the other hand, according 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

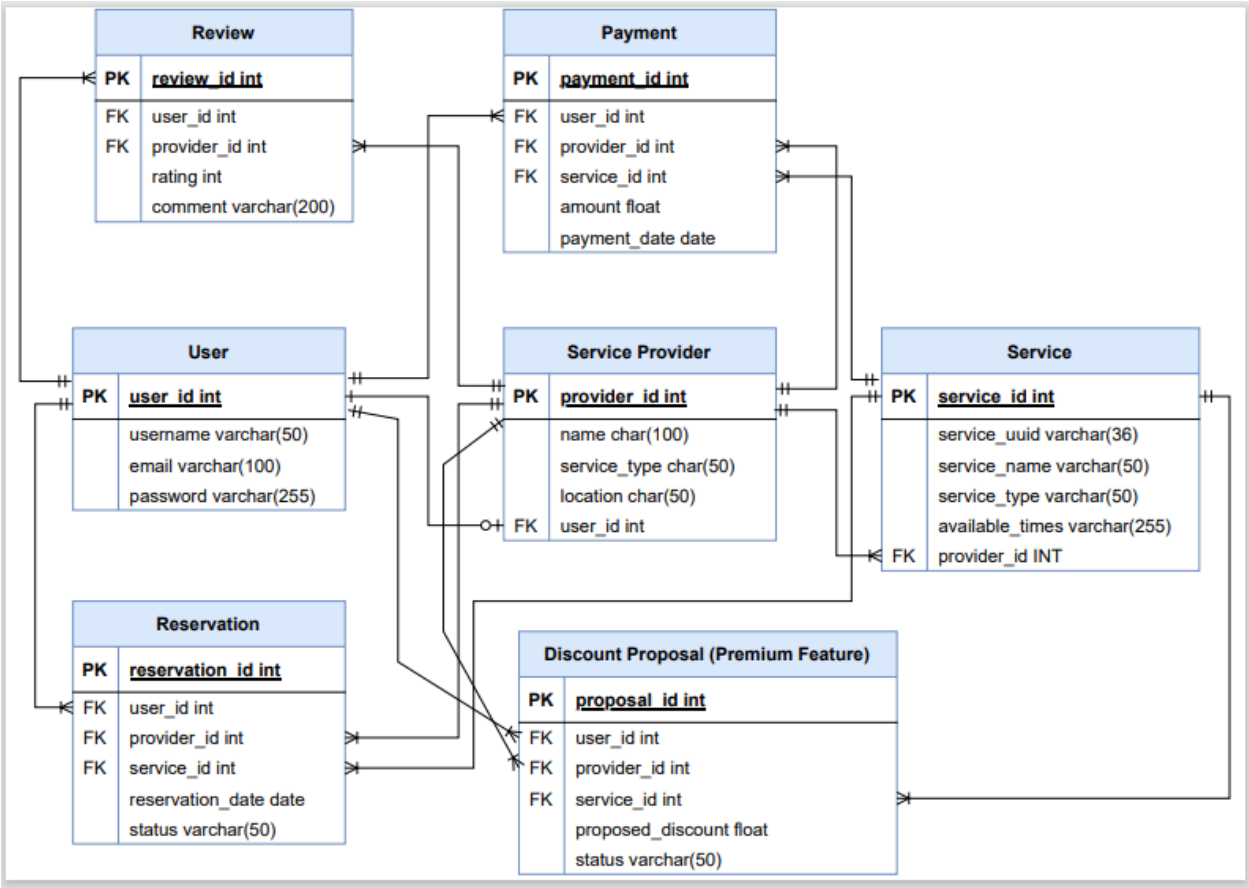
9.1 Use case



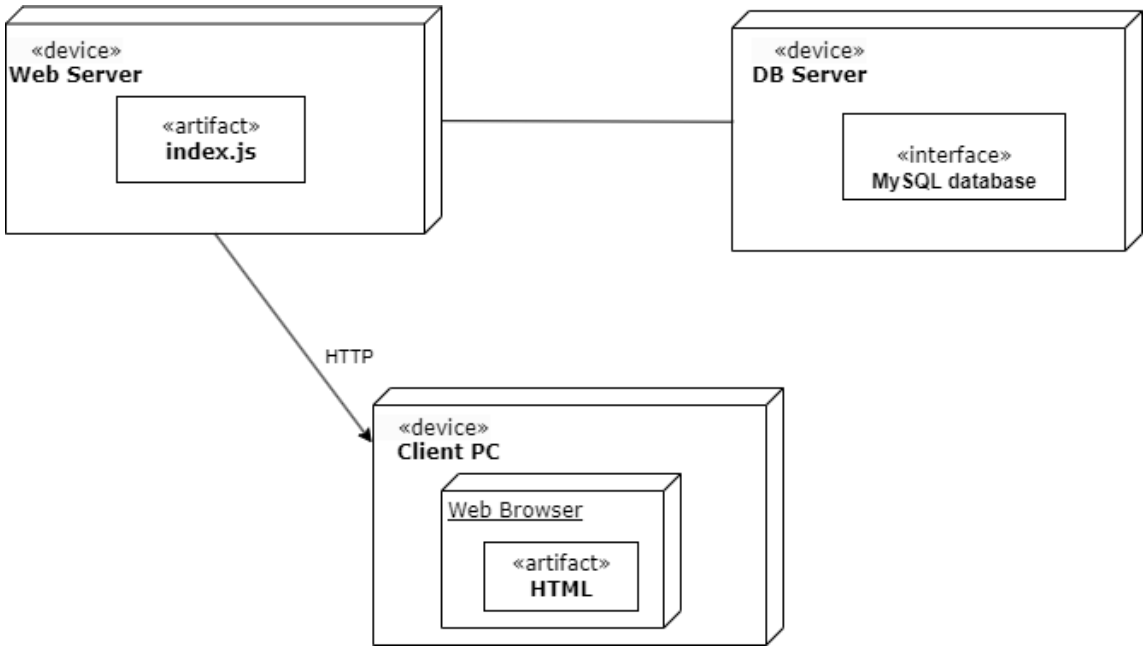
9.2 Domain model



9.3 Data view



9.4 Deployment View


«device»
Web Server
«artifact»
index.js
«device»
DB Server
«interface»
MySQL database
HTTP
«device»
Client PC
Web Browser
«artifact»
HTML