ReserveWell Architecture Notebook

Version History Table

Version	Date	Description
v1.0	25.11.2023	-
v1.1	02.12.2023	- Part 4 is revised to be coherent with System Wide Requirements document. - Part 5 is revised to be coherent with Part 4. - Diagrams are added to Part 7 and Part 9.
v1.2	16.12.2023	-Updated according to the feedback of Altan Hoca
v1.3	23.12.2023	-Updated according to the feedback of the reviewer group
v1.4	01.01.2024	- Deployment view is updated and client side is added. - Data view is added. - Decisions and constraints part is updated.

1. Purpose

This document outlines the architectural philosophy, decisions, constraints, justifications, significant elements, and other overarching aspects of the Restaurant Reservation Management System (RRMS) that shape its design and implementation. It serves as a comprehensive guide for developers and stakeholders to understand the system's architecture and its underlying principles.

The RRMS is a cloud-based system designed to facilitate restaurant reservations for diners, enabling them to easily book tables and manage their reservations. The system caters to both diners and restaurant staff, providing a user-friendly interface for both parties.

2. Architectural goals and philosophy

The Restaurant Reservation Management System (RRMS) is designed to be a highly available, scalable, and performant system that caters to both diners and restaurant staff. The system will be deployed to a cloud environment and utilize Next.js and React for the front-end, Firebase for the back end, and a no-SQL database for storing reservation data.

Architectural Goals

- High Availability: Ensure continuous system operation and seamless user experience, even during high traffic volumes.

- Scalability: Accommodate a growing user base and increasing reservation volume without compromising performance.
- Performance: Process reservation requests quickly and efficiently, providing a responsive user experience.
- Security: Safeguard sensitive user data from unauthorized access, maintaining data privacy and integrity.
- User-friendliness: Provide a straightforward and intuitive interface for both diners and restaurant staff for easy navigation and operation.

Critical Issues Addressed

- Hardware Dependencies: Isolate hardware dependencies to minimize the impact of hardware failures and simplify maintenance.
- Performance under Unusual Conditions: Employ mechanisms to handle unexpected situations like traffic spikes or data anomalies without compromising performance.

Key Decisions and Justifications

- Front-end Technology: Utilize Next.js and React due to their popularity, performance, and ease of use.
- Back-end Platform: Employ Firebase for its cloud-based services, including authentication, database storage, and real-time communication.
- Database Choice: Optimal for a no-SQL database for its suitability in storing and retrieving structured data like reservation information.

Architectural Significance

The RRMS architecture prioritizes continuous availability, scalability, performance, security, and user-friendliness to provide a robust, efficient, and user-centric reservation management system for both diners and restaurant staff.

3. Assumptions and dependencies

Assumptions:

- The development team possesses expertise in Next.js, React, Firebase, and no-SQL databases.
- The system will handle a moderate volume (in the range of a thousand users or requests per second) of traffic during peak hours, with occasional spikes in demand.
- The system will adhere to industry standards of SO/IEC 27001 and best practices for data security and privacy.

Dependencies:

- Adherence to data security and privacy regulations, such as GDPR and CCPA.
- Continuous maintenance and updates to ensure system stability and security.
- Synchronization for integration with individual restaurant websites.
- Capability to deploy on the cloud (Firebase).

4. Architecturally significant requirements

The following are the architecturally significant requirements for the Restaurant Reservation Management System referred to in the System Wide Requirements document.

2.2.1 The system needs to integrate with individual restaurant websites to ensure reservation information is synchronized and up to date.
2.2.2 Reservation, payment, and capacity data should be seamlessly shared between the system and restaurant websites.

2.3.1 The system should ensure the security of user data and protect sensitive information using industry-standard encryption and access controls.
2.3.2 Users must have control over their data, including the ability to update or delete their information under appropriate authorization conditions.

2.5.1 The system should handle a growing user base and maintain optimal performance, with a maximum response time of 3 seconds under standard (25 to 100 Mbps) internet connections.
2.5.2 Performance monitoring for memory consumption and response time optimization should be ongoing.
2.5.3 Number of concurrent users will be a thousand at maximum.

2.8.1 The system should integrate with external services such as payment platforms, Google Maps, and other relevant third-party systems.
2.8.2 Seamless integration with these services is crucial for system functionality.

2.10.1 The system should support multiple languages to accommodate diverse user segments, starting with English and expanding to other languages.
2.10.2 Cultural considerations should be considered for a seamless user experience.

5. Decisions, constraints, and justifications

Decisions:	**Justification**	**Related Requirements**
Next.js and React for Front-end Development:	Next.js and React are widely used and well-supported frameworks for building performant and user-friendly web applications. Their popularity provides a wealth of resources and community support for efficient development and troubleshooting.	2.10.1, 2.10.2
Firebase for Back-end Development:	Firebase offers a comprehensive suite of cloud-based services, including authentication, database storage, real-time communication, and hosting. This simplifies back-end development and reduces the need to manage and maintain infrastructure.	2.5.1, 2.10.1, 2.10.2
Cloud Fire Store No-SQL Database for Reservation Data:	Cloud Fire Store No-SQL databases provide flexibility and scalability for storing and retrieving reservation data, which is often unstructured or semi-structured in nature. They efficiently handle the dynamic nature of reservation data and allow for future growth.	2.3.1, 2.3.2, 2.5.3

Microservices architecture and restful web services	The system must employ microservices architecture, where each service is deployed as an independent unit. By using them, the system becomes more scalable and more flexible and agile. Alternatives to Microservices architecture might be Monolithic Architecture, or Service Oriented Architecture (SOA). Cons of Monolithic Architecture: Lack of flexibility, scalability challenges, and a single point of failure. Cons of SOA: Can be complex to implement, and services may be tightly coupled. Alternatives to Restful Web Services might be SOAP (Simple Object Access Protocol) or WebSocket. Cons of SOAP: Heavier and more complex compared to REST, may have higher overhead. Cons of WebSocket: May not be suitable for all types of interactions, increased complexity in some scenarios.	2.2.1, 2.3.1
Firebase Cloud Environment Deployment:	The system's architecture must be tailored to seamlessly integrate with Firebase, ensuring compatibility and functionality in accordance with Firebase-specific protocols and limitations. Utilization of Firebase-provided functionalities is mandatory.	2.5.1
Scalability for Traffic Spikes:	Developers must implement mechanisms that Firebase supports, including load balancing, caching, and autoscaling, to ensure the system can gracefully handle intermittent surges in traffic without compromising performance or user experience. The system's design should prioritize resilience under varying levels of load.	2.5.1, 2.5.3

Constraints:	**Justification**	**Related Requirements**
Data Security and Privacy Compliance:	The system must strictly adhere to industry standards and regulations such as GDPR and CCPA, placing emphasis on safeguarding user data and privacy. Implementation of robust data encryption, access controls, and auditing mechanisms is mandatory to meet the specified security and privacy compliance requirements.	2.3.1, 2.3.2, 2.10.1, 2.10.2

6. Architectural Mechanisms

Architectural Mechanism: Message Queue and API Gateway in Making a Reservation

Description: When a user makes a reservation through the front-end application, the reservation details are packaged into a message and sent to a message queue. The API gateway receives the message, extracts the reservation details, and routes the request to the appropriate microservice responsible for handling reservations. The microservice processes the reservation request, checks availability, and creates a new reservation record in the database. Once the reservation is created, the microservice publishes an event indicating a new reservation has been made. The event is consumed by other interested microservices, such as the notification service, which sends a confirmation email to the user.

Architectural Mechanism: API Gateway and Database Sharding in Updating a Reservation

Description: When a user updates an existing reservation, the updated reservation details are sent to the API gateway through an API call. The API gateway validates the request and routes it to the appropriate microservice responsible for managing reservations. The microservice retrieves the existing reservation record from the database, updates it with the new details, and persists the changes to the database. If the reservation data is sharded across multiple database shards, the microservice uses sharding middleware to locate the appropriate shard and update the corresponding reservation record.

Architectural Mechanism: Caching, Load balancing, and Autoscaling Mechanisms

Description: Firebase, a comprehensive mobile and web application development platform by Google, offers robust mechanisms for optimizing application performance. In terms of caching, Firebase Realtime Database automatically caches accessed data, minimizing the reliance on repeated network requests for enhanced app responsiveness. Firestore, another Firebase service, extends this capability with offline data support, synchronizing local caches with servers when the device is back online. Firebase Hosting provides efficient load balancing through a global content delivery network (CDN), distributing web app content across servers worldwide to reduce latency and improve user experience. Additionally, Firebase Cloud Functions and Hosting both support autoscaling, dynamically allocating resources based on demand, ensuring applications scale seamlessly to handle varying workloads while optimizing resource utilization and costs.

Architectural Mechanism: Data Security and Privacy

Description: Firebase incorporates several architectural mechanisms to ensure robust data security and privacy for applications. The platform supports secure data transmission through the use of HTTPS to encrypt data in transit, safeguarding it from unauthorized interception. Firebase Authentication provides a scalable and customizable identity verification system, allowing developers to implement user authentication securely. Firebase Realtime Database and Firestore, the NoSQL database offerings, include rules-based security to control access at the data level, enabling developers to define who can read or write to specific parts of the database. Firebase also integrates with Google Cloud's Identity and Access Management (IAM), allowing for fine-grained access control and permissions management. The platform facilitates secure serverless computing through Firebase Cloud Functions, ensuring that server-side code executes in a controlled environment. Developers can further enhance security by utilizing Firebase Security Rules, which enable the creation of rules to govern data access and modification based on specific conditions. These collective mechanisms form a comprehensive security framework within Firebase, addressing various facets of data protection and privacy.

7. Key abstractions

The following are the key abstractions for the Restaurant Reservation Management System.

- Reservation: A reservation represents a diner's request to reserve a table at a restaurant.
- Restaurant: A restaurant represents a place of business where diners can eat.
- Diner: A diner represents a person who can use the system to make reservations.
- Restaurant Manager: A restaurant manager oversees and manages restaurant operations. They utilize the RW Application to monitor reservation progress, adjust capacity, and optimize workforce allocation.
- Waitstaff: Waitstaff work in the restaurant industry.

Key abstractions' methods are managed by Firebase as the backend, actors communicate with the database from the frontend.

The key abstractions can also be seen in Figure 1.

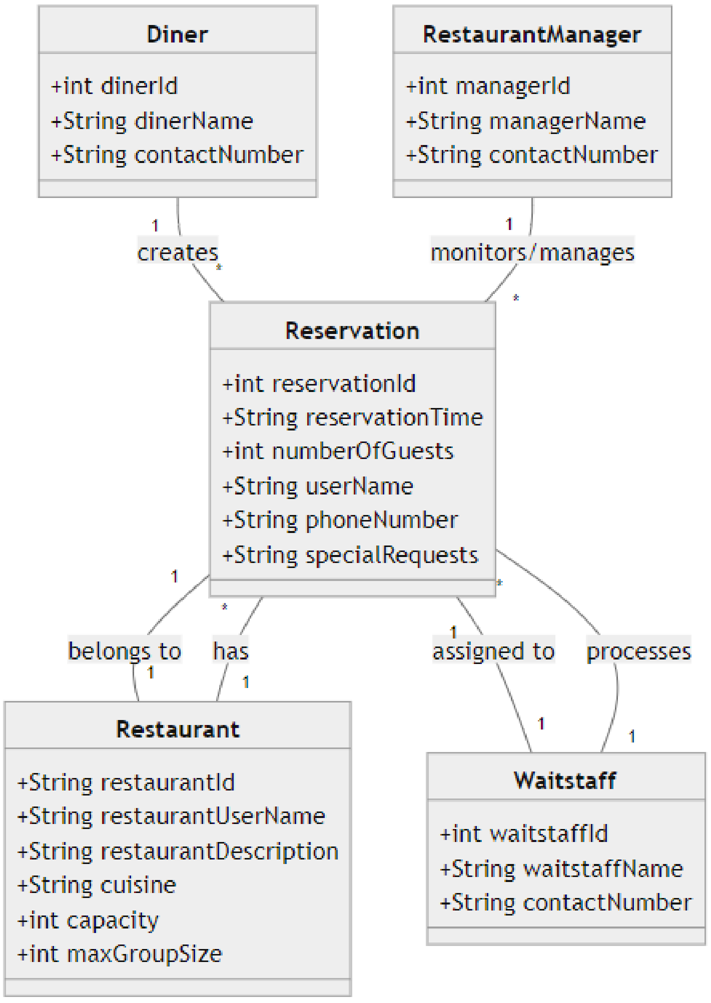


Figure 1: Class Diagram for key abstractions

8. Layers or architectural framework

The Restaurant Reservation Management System (RRMS) will employ a layered architecture pattern to ensure modularity, separation of concerns, and maintainability. This pattern divides the system into distinct layers with well-defined interfaces, promoting independent development, testing, and deployment of each layer.

1. **Presentation Layer:**
   - Responsible for user interaction and presentation logic.
   - Implements the front-end application using Next.js and React.
   - Handles user input, displays reservation information, and communicates with the API layer.
2. **Business Logic Layer:**
   - Encapsulates the core business logic and rules of the system.
   - Implements the back-end services using Firebase.
   - Processes reservation requests, manages data persistence, and handles business logic validation.
   - Enables secure transactions.
3. **Data Access Layer:**
   - Handles interactions with the database.
   - Utilizes no-sql database services, such as MongoDB, Cassandra, or CouchDB, to store reservation data.
   - Provides an abstraction layer between the business logic and data storage.

9. Architectural views

Deployment View

The deployment diagram seen in Figure 2 shows a simple web application deployed on a single server. The server is running an operating system, such Windows, and a web server, such as Firebase. The web application is deployed to the web server and is accessible to clients over the HTTPS protocol.
The client can be a mobile device or a web browser. The client sends a request to the server over HTTPS. The server receives the request and processes it. The server then sends a response to the client over HTTPS.
The web application is divided into two parts: the front end and the back end. The front end is responsible for rendering the user interface and handling user interactions. The back end is responsible for processing data and executing business logic.
The front end is deployed to the web server and is served to clients over HTTPS. The back end is deployed to the server and is accessible to the front end over TCP/IP.
The database is deployed to the server and is accessible to the back end over TCP/IP. The database stores the data used by the web application.

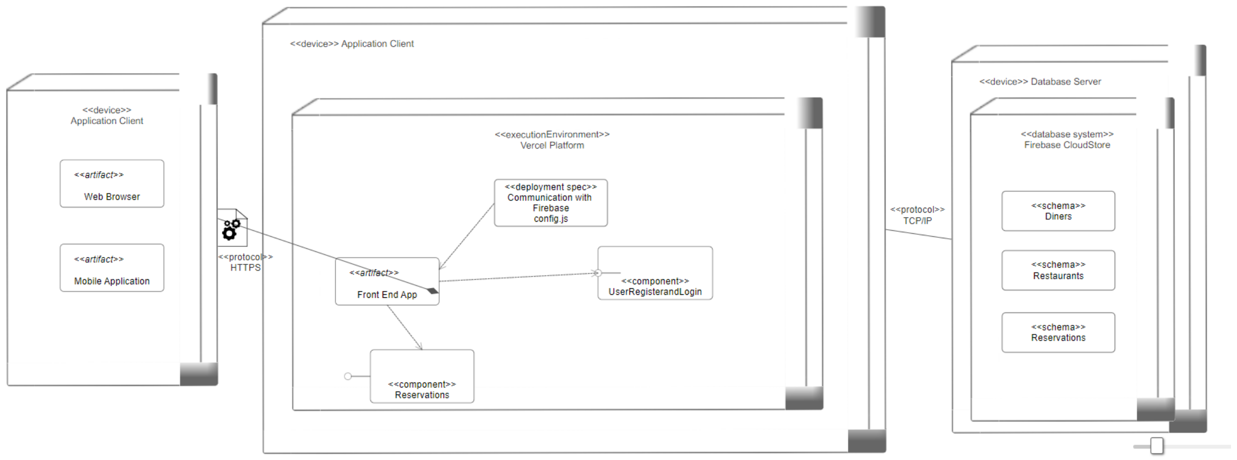

<<device>> Application Client
<<artifact>> Web Browser
<<artifact>> Mobile Application
<<protocol>> HTTPS
<<device>> Application Client
<<executionEnvironment>> Vercel Platform
<<deployment spec>> Communication with Firebase config.js
<<artifact>> Front End App
<<component>> UserRegisterandLogin
<<component>> Reservations
<<protocol>> TCP/IP
<<device>> Database Server
<<database system>> Firebase CloudStore
<<schema>> Diners
<<schema>> Restaurants
<<schema>> Reservations

Figure 2: Deployment View

Class View

Class view can be seen in 7. Key Abstractions part. Key abstractions are seen as classes in our domain model.

Logical View

The logical view describes the structure and behavior of the system from the perspective of its users and developers. It focuses on the key components and their interactions, emphasizing the functional aspects of the system.

1. System Decomposition:
   - The system is divided into three main layers: Presentation Layer, Business Logic Layer, and Data Access Layer.
   - Each layer has defined responsibilities and communicates with other layers through well-defined interfaces.
2. Component Interactions:
   - The Presentation Layer interacts with the user through the front-end application, handling user input and displaying reservation information.
   - The Business Logic Layer encapsulates the core reservation management logic, processing reservation requests and managing data persistence.
   - The Data Access Layer handles interactions with the no-SQL database, storing and retrieving reservation data.
3. Data Model:
   - The system defines a data model that represents key entities, such as Reservation, Restaurant, and User.
   - Relationships between entities are defined to model the system's functionality.

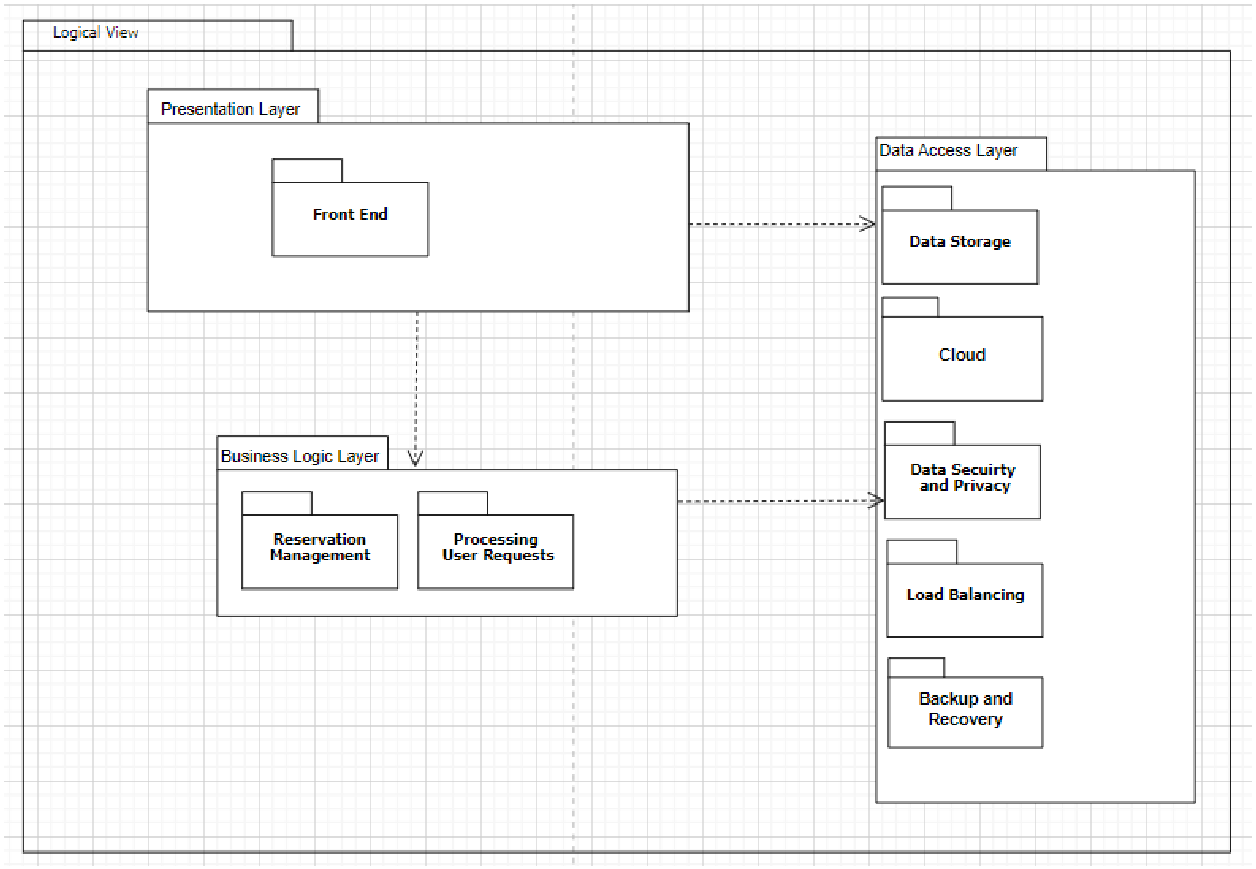


Figure 3: Logical View

Operational View
The operational view describes the physical deployment of the system, including the hardware and software components, their configuration, and how they interact.

1. Microservices Architecture:
    - The system employs a microservices architecture, where each service is deployed as an independent unit.
    - Services communicate through well-defined interfaces, such as RESTful APIs or message queues.
2. Security:
    - Security services are used to protect the system from unauthorized access and attacks.
    - Data encryption is used to protect sensitive data at rest and in transit.

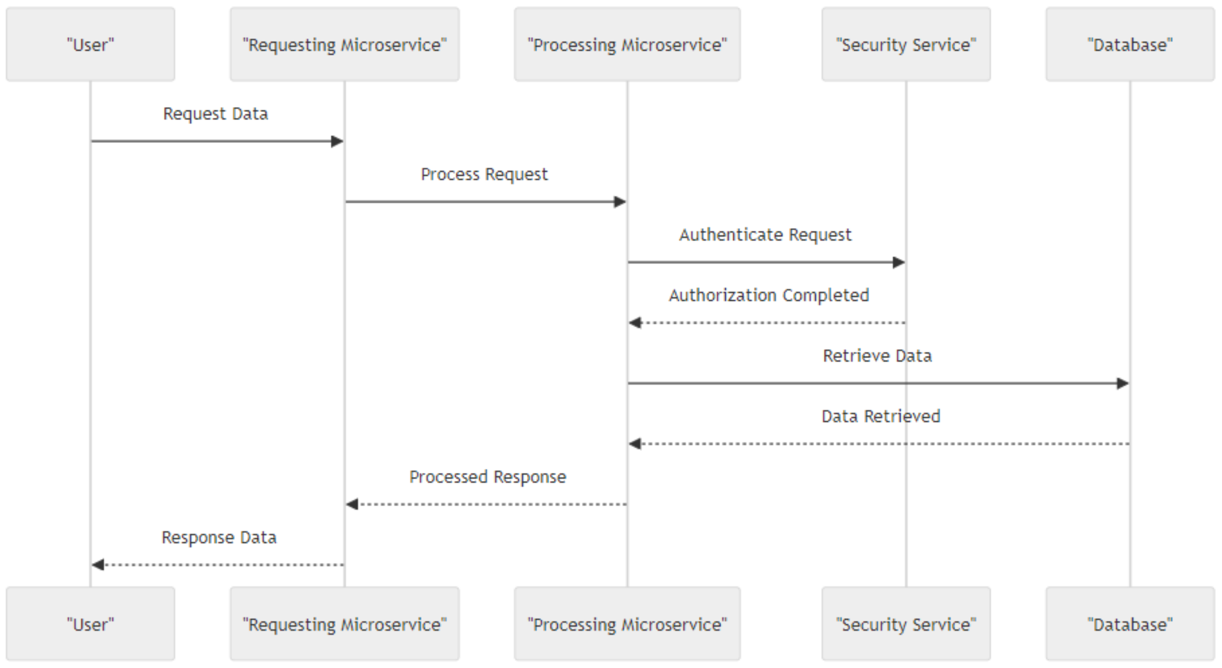


Figure 4: Operational View

Use Case View

The actors in the system are:

- Diner: Makes reservations, registers, updates reservations, exits from a waitlist, manages notifications about waitlists and reservations, rates the restaurant.
- Restaurant Manager: Edits working hours, edits capacity, displays dashboards.
- WaitStaff: Manages the restaurant's waitlist, displays dashboards.

The use cases for diners are:

- Register: Create a new Diner account.
- Make Reservation: Make a reservation for a table at a restaurant.
- Update Reservation: Modify an existing reservation.
- Exit from a Waitlist: Remove themselves from the waitlist for a restaurant.
- Manage Notifications: View and manage notifications about their reservations and waitlist status.
- Rate the Restaurant: Rate their experience at a restaurant.
- Display Reservations: View their upcoming reservations.
- Manage Reservations: Cancel or modify their reservations.

The use cases for restaurant managers are:

- Edit Working Hours: Set the operating hours of the restaurant.
- Edit Capacity: Set the number of tables available for reservations.
- Display Dashboards: View data and charts about the restaurant's reservations and waitlists.

The use cases for waitstaff are:

- Manage Restaurant's Waitlist: Add and remove diners from the waitlist, seat diners from the waitlist.
- Display Dashboards: View data and charts about the restaurant's waitlist.

There are also a few additional elements in the use case diagram:

- Confirm Realized Reservations: This is a note that indicates that the system should be able to confirm that a reservation has taken place.
- Payment Authorization: This is an inclusive relationship, which means that the "Make Reservation" use case includes the "Payment Authorization" use case. This means that when a

diner makes a reservation, they will also need to go through a payment authorization process.

- RegisterR: This is an extension relationship, which means that the "Register" use case can be extended to include additional functionality, such as registering as a restaurant manager or waitstaff member.

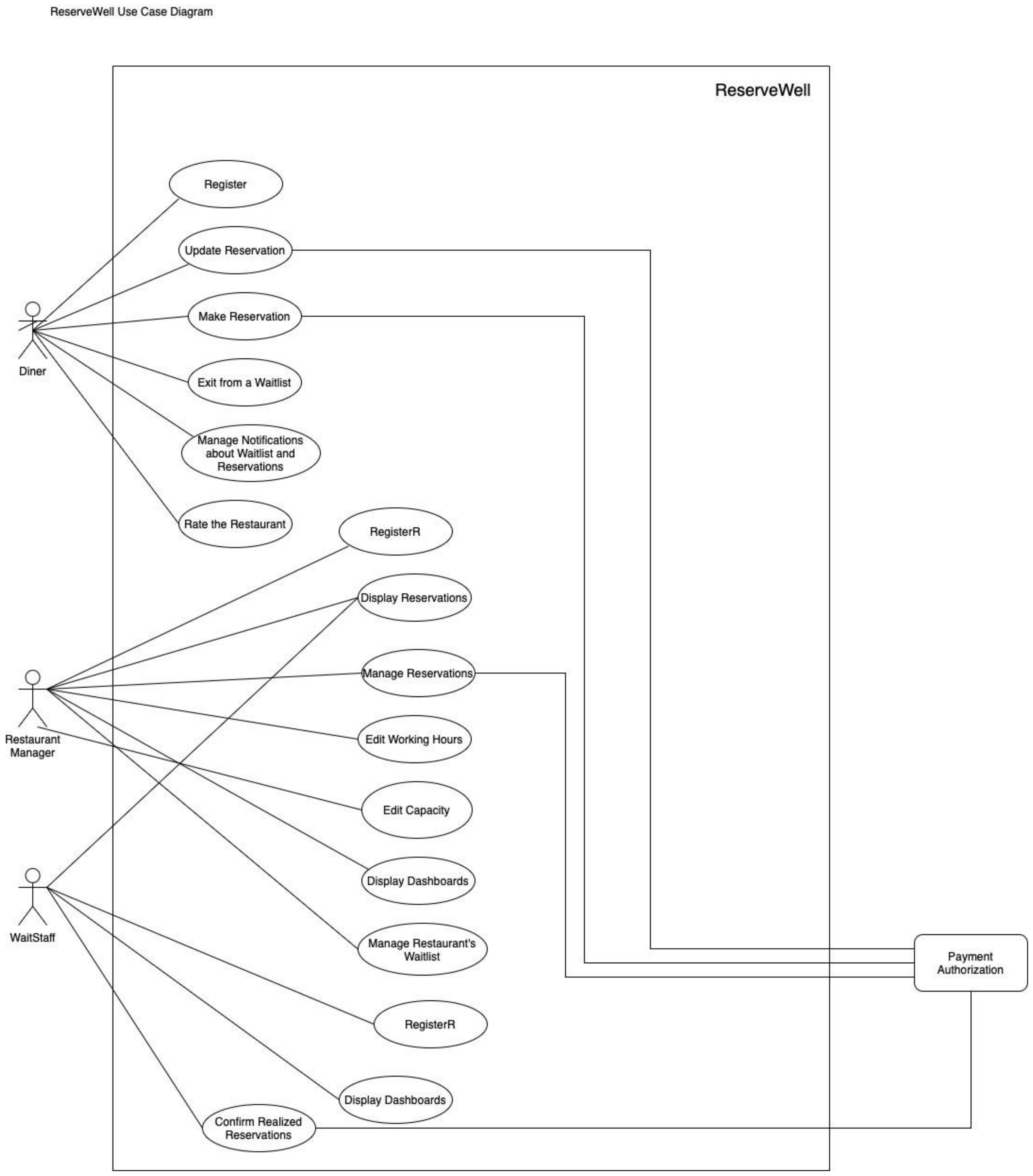
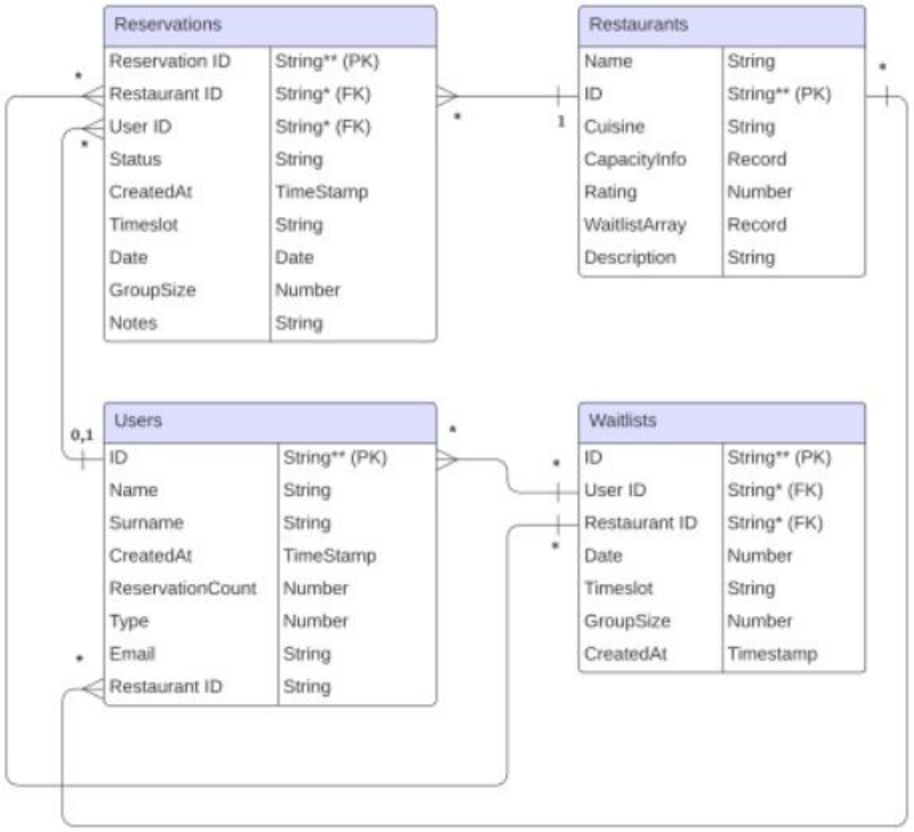


Figure 5: Use Case View

Data View

Data view actors are Reservations, Restaurants, Users and Waitlists as seen in Figure 6.



Reservations	
Reservation ID	String** (PK)
Restaurant ID	String* (FK)
User ID	String* (FK)
Status	String
CreatedAt	TimeStamp
Timeslot	String
Date	Date
GroupSize	Number
Notes	String

Restaurants	
Name	String
ID	String** (PK)
Cuisine	String
CapacityInfo	Record
Rating	Number
WaitlistArray	Record
Description	String

Users	
ID	String** (PK)
Name	String
Surname	String
CreatedAt	TimeStamp
ReservationCount	Number
Type	Number
Email	String
Restaurant ID	String

Waitlists	
ID	String** (PK)
User ID	String* (FK)
Restaurant ID	String* (FK)
Date	Number
Timeslot	String
GroupSize	Number
CreatedAt	Timestamp

Figure 6: Data View